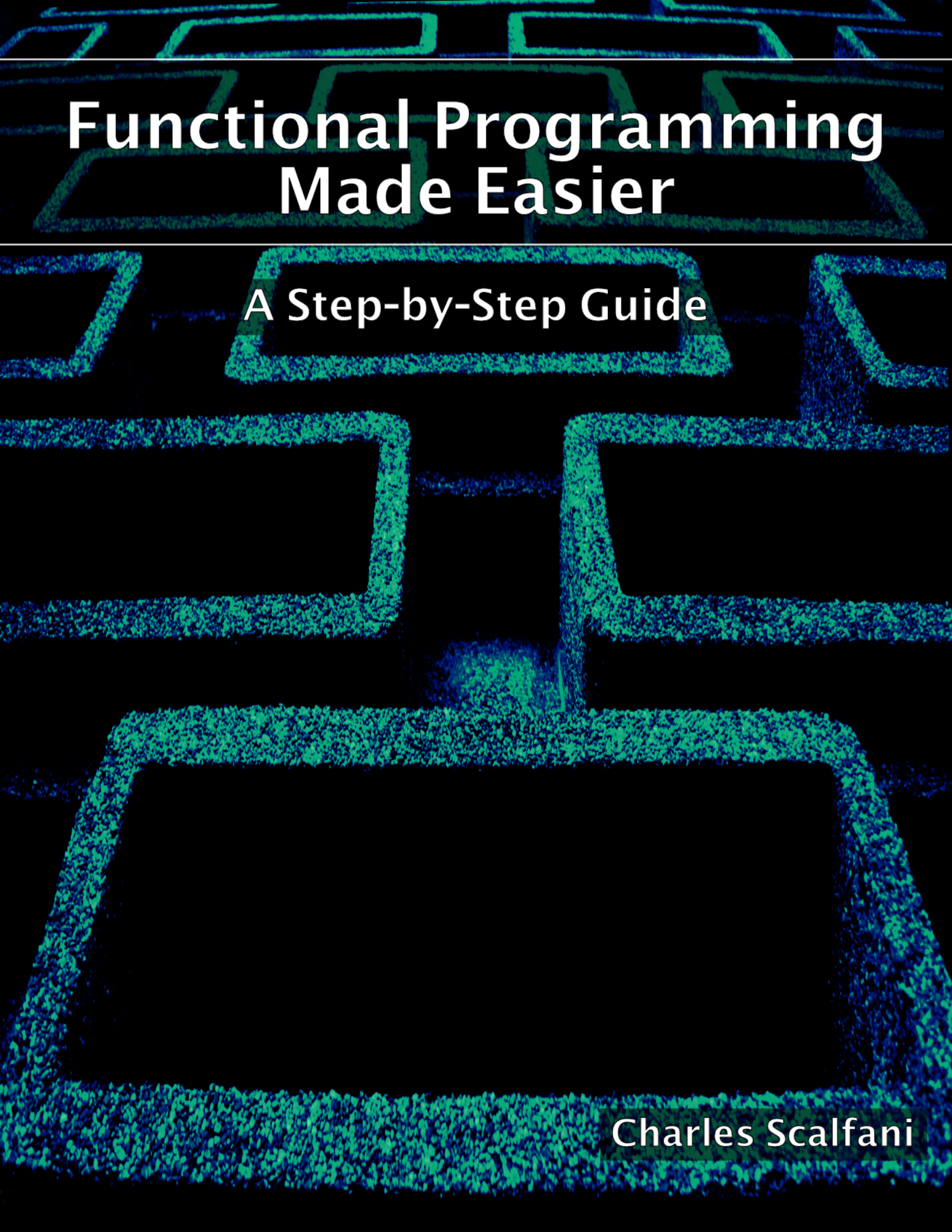

Functional Programming Made Easier
A Step-by-Step Guide
Charles Scalfani

Functional Programming Made Easier

A Step-by-Step Guide

Charles Scalfani

Table of Contents

Introduction . . . 1
- Learning Functional Programming is Hard. . . . 1
- Learning Programming is Learning How to Think . . . 1
- Learning Functional Programming is Learning to Think Differently . . . 2
- Why Functional Programming? . . . 2
- You Cannot Describe an Experience . . . 3
- Barrier to Entry . . . 4
- Costs of Ownership . . . 4
  - Case Study: Javascript . . . 5
  - Case Study: Elm . . . 5
- Finding Developers . . . 6
- Finding Jobs . . . 6
- Who is This Book For? . . . 7
- From A to Z . . . 7
  - A, B, X, Y, Z . . . 7
  - A, B, C . . . 8
  - X, Y, Z . . . 8
  - A to Z . . . 8
- Why PureScript? . . . 9
  - What's PureScript? . . . 9
  - Why not Haskell or Elm or ...? . . . 9
    - Haskell . . . 9
    - Elm . . . 10
    - Some Other Functional Language . . . 10
  - Why PureScript? . . . 11
- Four Part Harmony . . . 11
  - Beginner . . . 11
  - Intermediate . . . 12
  - Advanced . . . 12
  - Beyond . . . 12
- Exercises . . . 12
- So Many Pages . . . 13

I: Beginner . . . 15
- 1. Discipline is Freedom . . . 16
  - 1.1. Global State . . . 16
  - 1.2. Mutable State . . . 17

1.2.1. No Variables . . . 17
1.2.2. No Loops . . . 19
1.3. Purity . . . 22
1.4. Optimization . . . 24
1.4.1. Memoization . . . 24
1.4.2. Compiler Optimization . . . 25
1.5. Types . . . 27
1.5.1. Signal to Noise . . . 27
1.5.2. Perceived Limits . . . 28
1.5.3. Cake and Eat It . . . 28
1.5.4. Cutting Through the Noise . . . 29
1.5.5. Static Type Costs and Benefits . . . 29
1.6. Summary . . . 29
2. The Power of Functions . . . 31
2.1. Functions as Parameters . . . 32
2.2. Functions as Return Values . . . 35
2.3. Higher-order Functions . . . 37
2.4. Composition . . . 38
2.4.1. Point-free Notation . . . 42
2.5. Currying . . . 44
2.5.1. Partial Application . . . 48
3. The Basics of PureScript . . . 52
3.1. Types . . . 52
3.1.1. Javascript Primitives . . . 53
Boolean Type . . . 53
Char Type . . . 53
String Type . . . 53
Number Type . . . 54
3.1.2. PureScript Primitives . . . 55
Int Type . . . 55
Array Type . . . 55
Record Type . . . 56
Syntactical Oddity . . . 57
3.1.3. User Types . . . 57
Type Alias . . . 57
Data Type . . . 58
Algebraic Data Types (ADTs) . . . 59
New Types . . . 67

3.1.4. Common Library Types . . . 69
`Void` . . . 69
`Unit` . . . 71
`Maybe` . . . 71
Either . . . 76
Maybe vs Either . . . 78
Tuple . . . 81
Either vs Tuple . . . 82
List . . . 83
3.2. Pattern Matching . . . 85
3.2.1. Case Expression . . . 85
3.2.2. String Patterns . . . 86
3.2.3. Array Patterns . . . 87
3.2.4. List Patterns . . . 87
3.2.5. Array vs List . . . 88
3.2.6. Record Patterns . . . 89
3.3. Logical Control . . . 92
3.3.1. If-Then-Else Expression . . . 92
3.3.2. Case Expression . . . 93
3.3.3. Pattern Matching . . . 93
3.3.4. Guards . . . 94
3.4. Lambda Functions . . . 96
3.5. Wildcards . . . 99
3.5.1. Case Expression . . . 99
3.5.2. Operator Sections . . . 100
3.5.3. Records . . . 100
3.6. Bindings . . . 101
3.6.1. Where . . . 101
3.6.2. Let . . . 102
3.7. Binary Operators . . . 103
3.7.1. Associativity . . . 104
3.7.2. Precedence . . . 104
3.7.3. Fixity . . . 105
3.8. Comments . . . 105
3.9. Inferring Functionality from Type Signatures . . . 106
3.10. Summary . . . 112
4. Installing PureScript . . . 113
4.1. Compiler and Tools . . . 113

4.1.1. Installing Node . . . 114
For Node Already Installed . . . 114
For Node NOT Installed . . . 114
4.1.2. Create Project . . . 115
Initialize Project for npm and npx . . . 116
Future-proofing . . . 116
Install purescript and spago into your Project . . . 116
Initialize Project for git (Optional) . . . 117
4.1.3. Initialize PureScript Project . . . 117
4.1.4. A Second Project . . . 118
4.2. Editor and Plugins . . . 118
4.2.1. Install Extensions . . . 119
4.2.2. Configure Extensions . . . 119
4.2.3. Configure Editor . . . 120
5. Basic Coding in PureScript . . . 124
5.1. Prelude . . . 124
5.2. Exercise Program . . . 126
5.3. Pursuit . . . 127
5.4. Writing flip . . . 128
5.4.1. Hint for flip . . . 129
5.4.2. Code for flip . . . 130
5.4.3. Alternative Coding for flip . . . 130
5.5. Writing const . . . 131
5.5.1. Hint for const . . . 131
5.5.2. Code for const . . . 132
5.6. The Apply Operator ($) . . . 133
5.7. Writing apply . . . 134
5.7.1. Hint for ($) . . . 135
5.7.2. Code for ($) . . . 135
5.8. Writing the Apply Flipped Operator (#) . . . 136
5.8.1. Code for applyFlipped and (#) . . . 137
5.9. Preparing to Write Data.List Functions . . . 141
5.10. Why Data.List and not Data.Array . . . 144
5.11. Writing singleton . . . 144
5.11.1. Code for singleton . . . 145
5.12. Writing null . . . 145
5.12.1. Code for null . . . 146
5.13. Writing snoc . . . 146

5.13.1. Hint for `snoc` . . . 147
5.13.2. Code for `snoc` . . . 147
5.14. Writing `length` . . . 150
5.14.1. Hint for `length` . . . 150
5.14.2. Code for `length` . . . 151
5.15. Tail Recursion . . . 152
5.15.1. Observations regarding Tail Recursion . . . 156
5.16. Writing `head` . . . 156
5.16.1. Hint for `head` . . . 157
5.16.2. Code for `head` . . . 160
5.17. Writing `tail` . . . 161
5.17.1. Hint for `tail` . . . 161
5.17.2. Code for `tail` . . . 161
5.18. Writing `last` . . . 162
5.18.1. Hint for `last` . . . 163
5.18.2. Code for `last` . . . 163
5.19. Writing `init` . . . 165
5.19.1. Hint for `init` . . . 166
5.19.2. Code for `init` . . . 166
5.20. Writing `uncons` . . . 170
5.20.1. Code for `uncons` . . . 171
5.21. Writing `index` . . . 171
5.21.1. Hint for `index` . . . 172
5.21.2. Code for `index` . . . 172
5.22. Writing `!!` . . . 175
5.22.1. Hint for `!!` . . . 176
5.22.2. Code for `!!` . . . 176
5.23. Writing `findIndex` . . . 177
5.23.1. Hint for `findIndex` . . . 178
5.23.2. Code for `findIndex` . . . 178
5.24. Writing `findLastIndex` . . . 180
5.24.1. Hint for `findLastIndex` . . . 180
5.24.2. Code for `findLastIndex` . . . 181
5.25. Local Function Type Signatures . . . 182
5.26. Writing `reverse` . . . 184
5.26.1. Hints for `reverse` . . . 185
5.26.2. Code for `reverse` . . . 186
5.27. Writing `concat` . . . 187

5.27.1. Hint for `concat` 188
5.27.2. Code for `concat` 188
5.28. Writing `filter` 190
5.28.1. Code for `filter` 191
5.28.2. Alternative Coding for `filter` 192
5.29. Tail Recursive version of `filter` 193
5.30. Time vs Space 195
5.31. Writing `catMaybes` 195
5.31.1. Hint for `catMaybes` 195
5.31.2. Code for `catMaybes` 195
5.32. Writing `range` 196
5.32.1. Hint for `range` 196
5.32.2. Code for `range` 197
5.33. Optimizing `range` 197
5.34. Writing `take` 200
5.34.1. Hint for `take` 201
5.34.2. Code for `take` 201
5.35. Writing `drop` 203
5.35.1. Code for `drop` 204
5.36. Writing `takeWhile` 204
5.36.1. Code for `takeWhile` 205
5.37. Writing `dropWhile` 206
5.37.1. Code for `dropWhile` 207
5.38. Writing `takeEnd` 207
5.38.1. Hint for `takeEnd` 207
5.39. Another Hint for `takeEnd` 208
5.39.1. Code for `takeEnd` 209
5.40. Writing `dropEnd` 210
5.40.1. Code for `dropEnd` 211
5.41. Writing `zip` 212
5.41.1. Code for `zip` 212
5.42. Writing `unzip` 213
5.42.1. Code for `unzip` 213
II: Intermediate 215
6. Typeclasses 216
6.1. The Problem 216
6.2. The Solution 222
6.3. Constraints 224

6.4. Typeclass Requirement . . . 226
6.5. Built-in Typeclasses . . . 227
6.5.1. Eq Typeclass . . . 228
6.5.2. Ord Typeclass . . . 230
6.5.3. Show Typeclass . . . 235
6.6. Derived Instances . . . 235
6.7. Newtype Typeclass . . . 237
6.8. Deriving Instances using `newtype` . . . 239
6.9. Overlapping Instances . . . 242
6.10. Orphaned Instances . . . 244
6.11. Instance Dependencies . . . 248
6.12. Multi-Parametric Typeclasses . . . 249
6.13. Functional Dependency . . . 250
7. Coding Typeclasses . . . 259
7.1. Coding Preparation . . . 259
7.2. Maybe Data Type . . . 259
7.3. Code for `Maybe` Data Type . . . 259
7.4. Writing `Eq` for `Maybe` . . . 259
7.5. Code for `Eq` for `Maybe` . . . 261
7.6. Writing `Ord` for `Maybe` . . . 263
7.7. Hint for `Ord` for `Maybe` . . . 265
7.8. Code for `Ord` for `Maybe` . . . 265
7.9. Writing `>=` . . . 266
7.10. Hint for `>=` . . . 266
7.11. Code for `>=` . . . 267
7.12. Writing `Show` for `Maybe` . . . 268
7.13. Code for `Show` for `Maybe` . . . 268
7.14. Deriving `Eq`, `Ord` and `Show` for `Maybe` . . . 270
7.15. Hint for Deriving . . . 270
7.16. Hint for Deriving `Show` . . . 270
7.17. Code for `Eq`, `Ord` and `Show` for `Maybe` . . . 272
7.18. Deriving `Eq`, `Ord` and `Show` for `Either` . . . 273
7.19. Writing `Eq`, `Ord` and `Show` for `Either` . . . 273
7.20. Creating Our Own Typeclass . . . 278
7.21. Hint for `ToCSV` . . . 279
7.22. Code for `ToCSV` . . . 279
7.23. Using `ToCSV` . . . 279
7.24. Another Hint for `ToCSV` . . . 280

7.25. Code for `ToCSV` 280
7.26. Testing `ToCSV` 281
7.27. Writing `FromCSV` 282
7.28. Code for `FromCSV` 283
7.29. Testing `FromCSV` 287
8. Abstract Algebra 294
8.1. Binary Operators 294
8.1.1. Associative Property 295
8.1.2. Commutative Property 296
8.2. Closure 296
8.3. Magma 297
8.4. Semigroup 298
8.5. Semigroup Typeclass 300
8.6. Monoid 300
8.7. Monoid Typeclass 302
8.8. Monoids in Programming vs Math 303
8.9. Monoids in Programming 303
8.10. Groups 306
8.11. Group Typeclass 307
8.12. Modular Arithmetic and Groups 308
8.13. Abelian Group 311
8.14. Abelian Group Type Alias 311
8.15. Arithmetic Operators in PureScript 312
8.16. Semiring 314
8.17. Semiring Typeclass 315
8.18. Semiring for Functions 316
8.19. Semiring Laws for Functions 318
8.20. Ring 319
8.21. Ring Typeclass 320
8.22. Commutative Ring Typeclass 320
8.23. Euclidean Ring 321
8.24. Summary 321
9. Coding Abstract Algebra 323
9.1. Writing `Semigroup` Typeclass 323
9.2. Hint for `Semigroup` Typeclass 323
9.3. Code for `Semigroup` Typeclass 323
9.4. Writing `Monoid` Typeclass 324
9.5. Code for `Monoid` Typeclass 324

9.6. Writing `Semigroup` for `AndBool` . . . 324
9.7. Hint for `Semigroup` for `AndBool` . . . 325
9.8. Code for `Semigroup` for `AndBool` . . . 326
9.9. Writing `Monoid` for `AndBool` . . . 326
9.10. Code for `Monoid` for `AndBool` . . . 327
9.11. Verify Semigroup Laws for `AndBool` . . . 327
9.12. Writing `verifyAndBoolSemigroup` . . . 327
9.13. Verify Monoid Laws for `AndBool` . . . 328
9.14. Writing `verifyAndBoolMonoid` . . . 328
9.15. Writing `Semigroup` and `Monoid` for `OrBool` . . . 329
9.16. Code for `Semigroup` and `Monoid` for `OrBool` . . . 329
9.17. Verify Semigroup and Monoid Laws for `OrBool` . . . 330
9.18. Code for Semigroup and Monoid Laws for `OrBool` . . . 330
9.19. Writing `Mod4` Data Type . . . 331
9.20. Code for `Mod4` Data Type . . . 331
9.21. Writing `Semigroup` for `Mod4` . . . 332
9.22. Hint for `Semigroup` for `Mod4` . . . 332
9.23. Code for `Semigroup` for `Mod4` . . . 332
9.24. Writing `Monoid` for `Mod4` . . . 334
9.25. Code for `Monoid` for `Mod4` . . . 334
9.26. Writing `Group` Typeclass . . . 334
9.27. Code for `Group` Typeclass . . . 334
9.28. Writing `Group` for `Mod4` . . . 335
9.29. Code for `Group` for `Mod4` . . . 335
9.30. Writing `Eq` and `Show` for `Mod4` . . . 336
9.31. Code for `Eq` and `Show` for `Mod4` . . . 336
9.32. Writing `verifyMod4Semigroup` for `Mod4` . . . 336
9.33. Code for `verifyMod4Semigroup` for `Mod4` . . . 337
9.34. Writing `verifyMod4Monoid` for `Mod4` . . . 337
9.35. Code for `verifyMod4Monoid` for `Mod4` . . . 337
9.36. Writing Semigroup for Maybe . . . 337
9.37. Writing `Semigroup` and `Monoid` for `First` . . . 339
9.38. Code for `Semigroup` and `Monoid` for `First` . . . 339
9.39. Writing `Semigroup` and `Monoid` for `Last` . . . 340
9.40. Code for `Semigroup` and `Monoid` for `First` . . . 340
10. Folds . . . 343
10.1. Fold by other Names . . . 344
10.2. Foldable Typeclass . . . 345

10.3. Fold Associativity . . . 346
10.4. Foldable for `List` . . . 347
10.5. Rewriting `length` with Folds . . . 353
11. Coding Folds . . . 355
11.1. Writing `reverse` . . . 355
11.2. Code for `reverse` . . . 356
11.3. Writing `max` . . . 356
11.4. Hint for `max` . . . 357
11.5. Code for `max` . . . 357
11.6. Writing `findMax` . . . 358
11.7. Code for `findMax` . . . 359
11.8. Improving `findMax` . . . 359
11.9. Improved `findMax` . . . 360
11.10. Code for `findMax` using Fold . . . 362
11.11. Code for `findMax` using `NonEmptyList` . . . 363
11.12. Postmortem of `findMax` and `findMaxNE` . . . 365
11.13. Writing `findMaxNE` using `foldl1` . . . 365
11.14. Code for `findMaxNE` using `foldl1` . . . 366
11.15. Writing `foldl1` . . . 366
11.16. Hint for `foldl1` . . . 367
11.17. Code for `foldl1` . . . 367
11.18. Writing `sum` . . . 368
11.19. Code for `sum` as a Recursive Function . . . 368
11.20. Code for `sum` as Tail Recursive . . . 369
11.21. Code for `sum` using `foldl` . . . 370
11.22. Hint for `sum` for `Ints` and `Numbers` . . . 371
11.23. Code for `sum` for `Ints` and `Numbers` . . . 373
11.24. Code for `sum` using `Foldable` . . . 375
11.25. Using `sum` with `Tree` . . . 375
11.26. Hint for `Foldable` for `Tree` . . . 378
11.27. Code for `toList` Instance for `Tree` . . . 378
11.28. Code for `Foldable` for `Tree` . . . 379
11.29. Improving on `Foldable` for `Tree` . . . 380
12. Functors . . . 383
12.1. Defining ADTs . . . 383
12.2. Semantics . . . 386
12.2.1. Name . . . 386
12.2.2. Implicit Behavior . . . 387

12.2.3. Explicit Behavior . . . 387
12.3. Contexts . . . 388
12.4. Higher-kinded Types . . . 389
12.5. Functions with Simple Types . . . 390
12.6. Functors to the Rescue . . . 392
12.7. Two Perspectives of `map` . . . 395
12.8. The Power of Abstractions . . . 396
12.9. A Little Category Theory . . . 398
12.10. Concrete Examples in `Hask` . . . 403
12.11. Functors in Category Theory . . . 406
12.12. Functor Laws . . . 408
12.13. Verifying the Functor Laws for `Maybe` . . . 410
12.14. Functor Instance for `Either` . . . 411
12.15. Functor Instance for `Choice` . . . 415
12.16. Bifunctor Typeclass . . . 415
12.17. Bifunctors in Category Theory . . . 417
12.18. Bifunctor Laws . . . 422
12.19. Functors in Perspective . . . 423
13. Coding Functors . . . 426
13.1. Writing Functor Instance for `Maybe` . . . 426
13.2. Code for Functor Instance of `Maybe` . . . 427
13.3. Writing Functor Instance for `Either` . . . 432
13.4. Code for Functor Instance of `Either` . . . 432
13.5. Writing Functor Instance for `Tuple` . . . 441
13.6. Code for Functor Instance of `Tuple` . . . 441
13.7. Writing Functor Instance for `Threeple` . . . 442
13.8. Writing Functor Instance for `Threeple` . . . 442
13.9. Validating the Functor Laws . . . 443
13.10. Writing Bifunctor Instance for `Either` . . . 446
13.11. Code for Bifunctor Instance for `Either` . . . 449
13.12. Writing Bifunctor Instance for `Tuple` . . . 451
13.13. Code for Bifunctor Instance for `Tuple` . . . 451
13.14. Writing Bifunctor Instance for `Threeple` . . . 452
13.15. Code for Bifunctor Instance for `Threeple` . . . 452
13.16. Validating the Bifunctor Laws . . . 453
14. More Functors . . . 455
14.1. Functors of Values . . . 455
14.2. Functors of Functions . . . 456

14.3. Contravariant Functor . . . 459
14.4. Contravariant Laws . . . 460
14.5. Contravariant in Category Theory . . . 460
14.6. Choosing `Functor` or `Contravariant` . . . 461
14.7. Polarity . . . 466
14.8. Invariant Functor . . . 470
14.9. Invariant Laws . . . 471
14.10. Homomorphisms . . . 472
14.11. Natural Transformations . . . 472
14.12. Monoid Homomorphisms . . . 475
14.13. Isomorphisms . . . 479
14.14. Functor Instances for Function . . . 482
14.15. Profunctor . . . 486
14.16. Profunctor Laws . . . 487
14.17. Profunctor and Isomorphisms . . . 488
14.18. Profunctor Perspective . . . 490
14.19. Functor Summary . . . 491
14.20. Functor Intuition . . . 492
15. Coding More Functors . . . 494
15.1. Writing the Predicate `odd` . . . 495
15.2. Code for the Predicate `odd` . . . 497
15.3. Writing a `Predicate` Type . . . 498
15.4. Code for a `Predicate` Type . . . 499
15.5. Code for `runPredicate` . . . 500
15.6. Writing `Contravariant` Instance for `Predicate` . . . 500
15.7. Code for `Contravariant` Instance for `Predicate` . . . 501
15.8. Folds and Moore Machines . . . 502
15.9. Hint for `Profunctor` for `Moore` . . . 509
15.10. Code for `Profunctor` for `Moore` . . . 510
15.11. Modeling Folds with Moore Machines . . . 514
15.12. Writing a Moore Machine that Folds . . . 516
15.13. Code for `addr` . . . 516
15.14. Code for a general `addr` . . . 517
15.15. Writing `runFoldL` . . . 518
15.16. Code for `runFoldL` . . . 518
15.17. Leveraging `Moore's Profunctor` Instance . . . 519
15.18. Hint for `sizer` . . . 520
15.19. Code for `sizer` . . . 521

15.20. Power of the Functor . . . 523
III: Advanced . . . 524
16. Applicative Functors, Traversables and Alternatives . . . 525
16.1. Applicative in Haskell . . . 528
16.2. Applicative Functors in Category Theory . . . 529
16.3. Apply Laws . . . 530
16.4. Applicative Laws . . . 532
16.5. Applicative Instance for `Maybe` . . . 534
16.6. Applicative Instance for Product Types . . . 536
16.7. Applicative Instance for Sum Types . . . 538
16.8. An Applicative Example . . . 540
16.9. Applicative Effects . . . 544
16.10. Functors vs Applicatives with Effects . . . 548
16.11. Examples of Applicative Effects . . . 549
16.12. Applicative Effects and Commutativity . . . 549
16.13. Traversables . . . 550
16.14. Traversable for `List` . . . 553
16.15. A Few Words on Context . . . 557
16.16. Alt . . . 558
16.17. Alt Laws . . . 563
16.18. Plus . . . 564
16.19. Plus Laws . . . 566
16.20. Alternative Functor . . . 566
17. Coding Applicatives . . . 569
17.1. Writing `Applicative` Instance for `Maybe` . . . 570
17.2. Code for `Applicative` Instance for `Maybe` . . . 570
17.3. Writing `Applicative` Instance for `Either` . . . 572
17.4. Code for `Applicative` Instance for `Either` . . . 574
17.5. Validation . . . 576
17.6. Using Validation . . . 580
17.7. Applicative Parsers . . . 597
17.8. Parsers in General . . . 597
17.9. Writing an Applicative Parser . . . 598
17.10. Looking Deeper at our Parser . . . 615
17.11. A Common Pattern with Our Parser . . . 616
17.12. Using Our Parser . . . 617
18. Monads . . . 637
18.1. Side-effects . . . 637

18.1.1. Composing Side-effect Functions . . . 640
18.1.2. Composing Side-effect Function with Pure Ones . . . 642
18.1.3. Function Application with Side-effect Functions . . . 643
18.2. `Debuggable` Type . . . 645
18.3. Generalizing `Debuggable` . . . 648
18.4. We Created a `Monad` and More . . . 654
18.5. An Alternative `Monad` Implementation . . . 656
18.6. Back to PureScript's `Monad` Implementation . . . 659
18.7. Recap `Bind`, `Monad` and Supporting Functions . . . 660
18.8. Haskell's Monad Implementation . . . 662
18.9. Monad Laws . . . 663
18.10. Monad Instance for `Maybe` . . . 668
18.11. Working with `Maybe` Monad . . . 673
18.12. Do Notation . . . 677
18.13. Cheating with the `Applicative` Instance and `ap` . . . 682
18.14. Monad Instance for `Either` . . . 686
18.15. Working with `Either` Monad . . . 688
18.16. Adding `Monad` Instance for `Validation` . . . 689
18.17. `Writer` Monad . . . 690
18.18. `Writer` Helper Functions . . . 696
18.19. Parallel Computations . . . 699
18.20. `Reader` Monad . . . 701
18.21. `State` Monad . . . 714
18.22. Using `State` as a Monadic `Validation` . . . 725
18.23. The Kleisli Category . . . 732
19. Coding Monads . . . 743
19.1. Writing `Monad` Instance for `Maybe` . . . 744
19.2. Code for `Monad` Instance for `Maybe` . . . 745
19.3. Writing `Monad` Instance for `Either` . . . 747
19.4. Code for `Monad` Instance for `Either` . . . 748
19.5. Monadic Parsers . . . 750
19.6. Coding with Monadic Parsers . . . 761
19.7. Writing a Date Parser . . . 774
19.8. How Does `Parser` Actually Work . . . 809
19.9. `some` and `many` Combinators . . . 812
19.10. Using `some` and `many` . . . 832
19.11. The `RWS` Monad . . . 838
20. Monad Stacks . . . 855

20.1. Reducing the need for `lift` 863
20.2. Monad Transformers and their APIs 866
20.3. The Error Monad Transformer, `ExceptT` 868
20.4. Monad Stack Order 868
20.5. Coding with Monad Transformers 872
20.6. Coding with `Effect` at the Base 884
20.7. Coding `WriterT` 886
20.8. Coding `ReaderT` 897
20.9. Making `ReaderT` Easier to Use 905
20.10. The Power of the Combinator 917
21. Coding Monad Transformers 919
21.1. Writing the `StateT` Monad Transformer 920
21.2. Type Holes and Undefined 926
21.3. Back to Writing the `StateT` Monad Transformer 929
21.4. Testing our StateT Implementation 957
21.5. Problems with using StateT with ExceptT 971
21.6. Guidelines for Using `ExceptT` 979
IV: Beyond 981
22. Synchronous and Asynchronous Effects 982
22.1. Working with the `Effect` Monad 982
22.2. Working with the `Aff` Monad 986
22.2.1. Fibers 988
22.2.2. Cancelers 993
22.2.3. AVars 994
22.2.4. Busses 997
23. Coding With Effects 1000
23.1. Simple AVar Program Specification 1001
23.2. Effects Program Specifications 1013
23.2.1. Creating a Random Number 1015
23.2.2. Making a Monad Stack 1022
23.2.3. Creating a `Fiber` 1024
23.2.4. Running a Monad Stack in a `Fiber` 1025
23.2.5. Making a `Bus` 1027
23.2.6. Publish to a `Bus` 1033
23.2.7. Subscribing to a `Bus` 1034
23.3. Coding our Effects Program 1034
23.4. Javascript Runtime with AVars and Busses 1055
24. JSON and Ajax 1058

24.1. Generic JSON Encoding and Decoding . . . 1058
24.1.1. `Foreign` Module . . . 1059
24.1.2. `Foreign.Generic` Module . . . 1060
24.1.3. `Foreign.Generic.Class` Module . . . 1060
24.2. `Argonaut` JSON Encoding and Decoding . . . 1065
24.3. Ajax . . . 1078
24.3.1. GET Request . . . 1078
24.3.2. POST Request . . . 1088
25. Coding With Ajax and JSON . . . 1101
25.1. Ajax Program Specifications . . . 1102
25.2. Coding Ajax and JSON with Foreign . . . 1104
25.2.1. Encoding and Sending to the Echo Server . . . 1111
25.2.2. Decoding the Response Modeled with Reversed Names . . . 1127
25.2.3. Decoding the Response Using `Argonaut` . . . 1142
25.2.4. Serial and Parallel `Aff` . . . 1183
25.2.5. Using Parallel with Ajax . . . 1189
26. Foreign Function Interface (FFI) . . . 1200
26.1. FFIs That Return PureScript Types . . . 1202
26.2. FFIs and `Effect` . . . 1203
26.3. FFIs and `Aff` . . . 1206
27. Coding With FFIs . . . 1209
27.1. Wrapping an NPM library . . . 1210
27.2. Reversing JSON Keys in Javascript . . . 1222
28. Writing a Backend using HTTPure . . . 1254
28.1. A Brief Primer on `HTTPure` . . . 1255
28.1.1. Creating an HTTP Server . . . 1255
28.2. Routers . . . 1259
28.3. API For a Front-end Application . . . 1264
28.4. Modeling the API . . . 1271
28.5. Coding the Server's Infrastructure . . . 1294
28.6. Coding the Account Handler . . . 1315
28.6.1. Coding `loadAccounts` . . . 1316
28.6.2. Coding `createAccount` . . . 1341
28.7. Coding the Account Manager . . . 1349
28.7.1. Code for Password Hash . . . 1350
28.7.2. Coding `startup` and `shutdown` for Account Manager . . . 1364
28.7.3. Coding `verifyLogon` . . . 1369
28.7.4. Finishing the `logon` Request . . . 1372

28.7.5. Fleshing out `router` 1381
28.8. Coding the Session Manager 1402
28.8.1. Code for `startup`, `shutdown` and `verifySession` 1402
28.9. Server Logging 1418
28.10. Coding the API 1427
28.10.1. Coding `LogoffRequest` API Handler 1427
28.10.2. Coding `CreateUserRequest` API Handler 1467
28.10.3. Improving Our Parser 1525
28.10.4. Coding `QueryUsers` API Handler 1535
28.11. Servers in PureScript 1550
29. Building Front-ends using Halogen 1551
29.1. Halogen Component Overview 1551
29.1.1. State 1551
29.1.2. Actions 1552
29.1.3. Queries 1552
29.1.4. Inputs 1552
29.1.5. Outputs 1552
29.1.6. Emitters 1552
29.1.7. Lifecycle 1553
29.1.8. Slots 1553
29.1.9. Halogen Model 1553
29.2. Halogen Example Application 1554
29.2.1. Creating New Project 1554
29.2.2. Writing a Counter Component 1558
29.2.3. Rendering the Counter Component 1574
29.2.4. Counter Component Query Algebra 1581
29.2.5. Counter Component as a Child Component 1584
29.2.6. Emitters 1601
30. Writing a Front End using Halogen 1632
30.1. Halogen CSS Library Setup 1634
30.2. Hash Routing 1635
30.2.1. Defining Routes 1638
30.3. Application Monad 1645
30.4. Capabilities 1650
30.5. Capability Implementations 1656
30.6. Router Component 1662
30.7. Page Component 1674
30.8. Logon Page 1683

30.9. Calling the Backend . . . 1705
30.9.1. CORS Solution . . . 1724
30.9.2. Static File Server Solution . . . 1727
30.10. ChangePassword Page . . . 1740
30.11. Users Page . . . 1748
30.12. Modal Dialog Component . . . 1791
30.13. Message Modal Component . . . 1799
30.14. Customizing the Modal Dialog Component . . . 1810
30.15. Create Users Modal Component . . . 1825
30.16. Logoff Page . . . 1888
Appendix A: Epilogue . . . 1904

Introduction

Learning Functional Programming is Hard.

I failed many times to learn Haskell, the Granddaddy of all Functional Programming Languages, before I finally broke through. But not before trying to use Functional Concepts in Javascript. And not before learning Elm (a Purely Functional DSL for web-based, front end GUIs).

It was so hard that I tried anything to avoid the pain. When I tried to learn from free sources on the Internet, I was unable to tell the difference between something I was ready to learn and something I would never need to know as a professional Functional Programmer.

Learning on my own took too long and was too painful. Even though, Functional Programming is more advanced than Imperative Programming, [1] I realized that it didn't have to be that difficult.

The reason that it's so difficult is because there is no definitive guide to turn you from an Everyday Programmer, who programs in Imperative Languages, into a Functional Programmer, who can use Functional Languages like Elm, PureScript, Haskell, etc. to solve everyday problems encountered at work.

This doesn't mean there aren't lots of great blog posts or books. The problem is that there wasn't a single source that would take me by the hand from A to Z [2]. And as a novice, I couldn't tell which books were good and which ones were not. Which blog posts I was ready to read and which ones were far beyond my current reach.

And as a busy CTO of a small company, it was difficult to find the time and energy required to learn Functional Programming. It baffled me that after 35 years of programming experience, that I was struggling to learn yet another language.

And that's the thing. I was not learning just another programming language.

Learning Programming is Learning How to Think

When we learn our first programming language, we also learn how to think like the computer. This way of thinking isn't that foreign to us. Many people use a recipe as an example of a computer program.

First, get out the bread. Then the mustard. Then apply mustard to the bread. And so on.

This way of thinking is exactly how we see our world and so we've been learning how to think that way our whole lives. And this closely matches how the computer operates.

First fetch a value from an address and place it into a Register. Then fetch another value into another Register. Then add them putting the result into yet another Register. And so on.

A sequential, step-by-step process that we're all too familiar with. We still have to learn how to think like the computer since the computer is a pretty simple machine. So complex problems have to be broken down

into something a simpleton can do.

But thinking of a program as a sequence of instructions that will be followed by the computer isn't that different from how we accomplish tasks in the real world. So the only real struggle is learning how to map complex problems to a set of basic functions that can be efficiently executed by a computer and yet still understood by a human.

Learning Functional Programming is Learning to Think Differently

Many people find programming to be much easier than Math. To think like a Mathematician is difficult and unnatural. Our natural way of thinking is to jump to unsubstantiated conclusions quickly.

While this probably served us well when we thought we saw a predator on the savannas of Africa, it doesn't always produce accurate results. This is why producing a proper Mathematical Proof is a learned skill that takes many years of practice.

While Programming requires some practice, an unskilled person can produce a working program pretty easily. Professionals may cringe at the sight of their code and it may fail on some edge cases or perform poorly, but, nevertheless, it is a working program.

No such luck as a Mathematician. If you miss an edge case, the whole proof is wrong and unusable. Math holds us to a higher standard and therefore requires us to think differently, i.e. more precisely.

The same is true for Functional Programming. We must think differently and unnaturally, very much like the Mathematician.

However, I want to be very clear here. You do NOT need to be a Mathematician to use Functional Programming but by the end of this book, you will think more Mathematically even if you don't realize you're doing so.

Why Functional Programming?

There have been many attempts to answer this question [3] but I'm going to take a different tact. Instead, I'm going to tell you some of the things I value in programming.

I value:

- Software whose behavior can be easily reasoned about by simply reading the code.
- Programs that don't require extensive testing.
- Powerful abstractions that allow me to get a lot of work done with very little effort.
- Beauty in code and design.
- Reduction or complete elimination of boilerplate code.

- Reusable code.
- Confidence in code correctness.
- Elimination of unnecessary complexity.
- Coding mostly the Happy Path.

I cannot imagine many programmers who don't value the things on this list. We all want powerful tools that help us solve problems without requiring us to do the same work over and over again while producing code that's predictable and maintainable.

We don't want to deploy our code into production only for it to break requiring us to stay up all night trying to understand code that we wrote a few months ago under the pressure of production being down.

Essentially, we want to be in control of our code to minimize pain, and anything that makes our job easier and less painful is valuable to us.

If you value these things, then Functional Programming is the best and most powerful tool that we have to date to achieve everything on this list and more. That's not to say that something more powerful may not come along in 10 to 20 years, because it could.

But unlike Imperative Programming, Functional Programming has its roots in Mathematics that goes back over 100 years and that Math is built on a foundation of Mathematics that goes back hundreds if not thousands of years. In contrast, conventional programming languages only go back about 60 years.

It is said that Mathematics is discovered not invented. This implies that Functional Programming is also discovered, whereas most other languages are simply invented. [4]

And since Mathematics has proven itself so valuable for solving complex problems in nearly every human endeavor, it's pretty likely that Functional Programming is our best hope to control the scourge of complexity that's pervasive in our modern software.

You Cannot Describe an Experience

Ever hear someone say, "You had to be there"? Well, trying to explain why Functional Programming is so great is somewhat like that.

I know now that, when given the choice, I'll never choose an Imperative Language over a Functional one ever again.

However, if you pressed me for an explanation, I'd say because the *experience* of programming in a Functional Language is just so far superior that I enjoy programming infinitely more.

Unfortunately, this explanation is terrible because I cannot describe an experience well enough so that you will understand, unless, of course, you've shared that experience.

It's like trying to explain to a couple who are about to have their first baby what it's like to become parents and how the center of their universe will now be on the child. I don't even try explaining this. Instead, I

simply wait until after their child is born and then I can have a conversation with them regarding the ups and downs of parenthood.

Functional Programming is similar. As you work through this book, you will slowly experience what so many other programmers have experienced on their journey. You'll fully understand what I mean by "far superior experience" after you build your first real-world project, which we will do in the last Chapters of the book.

Barrier to Entry

Javascript probably has the lowest Barrier to Entry of all programming languages because it's already installed on nearly every computer inside of every Browser. Even your Grandmother has access to Javascript and, if so inclined, could start writing a line or two of Javascript code today.

You don't have to be a programmer to use Javascript. Many people just copy code from StackOverflow and paste it into their editor without ever understanding how it works. This doesn't always bode well and often they need to hack haphazardly at the code to cajole it to work. With very little skill, people can bootstrap themselves to become Javascript programmers, albeit not very good ones.

The Barrier to Entry for Functional Programming is the highest I've ever encountered. Partly why I wrote this book was to help reduce (not eliminate) the amount of effort required to enter into the Functional landscape.

So far, it may seem like I'm making an argument for why NOT Functional Programming, which would be true if not for one very important thing, Costs of Ownership.

Costs of Ownership

One thing that most people dismiss when buying a new car is the cost of owning that new car. When deciding between two different cars most people simply consider the initial upfront costs, i.e. the sticker price, while completely overlooking the fact that they have very different Costs of Ownership. It is the rare individual who takes this into consideration when they are sitting in the car salesman's office.

Ideally, one would want to know what the hourly cost of labor is at the dealership or what the manufacturer charges for a new alternator or radiator.

Another useful piece of information is reliability. The cost to fix the more expensive car may be higher than the cheaper one, but if it's more reliable, then that's another factor to consider. Repairing a cheap car often will add up quickly.

Other useful information is insurance costs and fuel efficiency.

These are all Costs of Ownership for a car and armed with this knowledge, we could make far better decisions regarding which car to buy.

Case Study: Javascript

It’s no different for software. We developers consider mostly language popularity and job prospects when choosing which language to hang our hat on. However, we rarely consider the Costs of Ownership.

One of the things I really liked about Javascript when I first started using it was how quickly I could write something and get it to work. I was coming from Java and welcomed the freedom from what I considered, at the time, "its tyrannical adherence to Types".

In the early days of Javascript, I’d sometimes get a Runtime Exception when I ran my code but I could easily and quickly fix it. Little did I realize, that this experience would scale up exponentially with the size of my program.

Having a dynamic language felt freeing and more efficient because I didn’t have to wait for it to compile and I could do "anything" to my program and "not have to worry about Types". I could have an Array that contained BOTH Strings and Numbers if I wanted to. In my mind, this was a big win regarding the development phase.

The Cost associated with writing Javascript code was far less than it was with a Statically Typed language, which is why I initially loved it.

But because Javascript isn’t compiled, it routinely crashes in the hands of users. So to combat this, the industry embraced Test Driven Development. It was all the rage and was touted to ferret out bugs and reduce Runtime Errors. Unfortunately, this approach has saddled us with the cost of having to write and maintain test code, which quickly grows to be an appreciable percentage of our codebase with its own set of bugs.

For me, this cost began to outweigh any gains I originally got from using a Dynamic Language.

Case Study: Elm

When our company decided to use Elm instead of Javascript for our next big project, I had to train the developers in basic Functional Concepts which was a one-time, up front cost.

In the beginning, a lot of time was spent trying to decipher the compiler errors. For example, Elm has Type Inference meaning that you don’t always have to specify the Types since the compiler can infer them from how you use your variables. However, this means that the error isn’t always reported where you made your mistake.

It takes developers time to realize that they used a variable differently in two places, e.g. as a String and a Number, and that the second usage is where the error will show up. Sometimes that isn’t always the wrong usage. Maybe it’s supposed to be a Number and the usage as a String is where the real error is.

Even after this initial developer acclimation period is over, an appreciable amount of time is spent dealing with compiler error messages, i.e. mistakes made by the developer that the compiler catches. These bugs are not caught by testing code or by users, but, instead, by the compiler, code that we do not have to write specifically for our program and code that we do not have to maintain.

When we put our Elm code into production, it just seemed to work with no Runtime Errors. [5]

The Upfront Cost of Elm was huge compared to Javascript, but the Costs of Ownership were drastically reduced so much so that we are able to add complex features to our product faster than ever before.

It's worth noting that this gain wasn't solely because of the difference between a Statically Typed language and a Dynamically Typed one. Many of the gains in development came from the developer's ability to more easily reason about their code and to be able to leverage powerful abstractions.

Finding Developers

If you are involved in the hiring process and you're reading this wondering how you'll find trained Functional Programmers if you adopt a Functional Language at work, the answer, at the time of this writing, is that you're probably not, at least not in your area.

If your company is open to remote workers then finding developers is not a problem. There are plenty of Developers who would gladly leave their current employment to get a chance to work in a Functional Programming Language. I saw this personally when we were hiring Elm programmers.

Also, in my experience, Functional Programmers are a cut above most. They've typically had to endure the hardship of learning this craft on their own time and are usually lifelong learners. Hopefully, this book will go a long way to reduce that hardship, but it will never be eliminated completely. Anything worth doing is going to require hard work.

But the truth of the matter is that you'll most likely have to train them. I hope that this book will help you in that process and hopefully more books will be written with similar goals.

Finding Jobs

One thing I've always prided myself on the fact that I always stay up to date with technology. Since, I can't know all technologies, I have to pick and choose. The way I did this early in my career was to look at the industry and try to predict what skills would be in demand in the next 5 years.

So, will Functional Programming be a requirement in 5 years? If you asked me at the time of this writing, I'd have to say that it probably won't be. Even though many programming languages have adopted Functional features, e.g. Java now has Lambdas, it's still difficult to find jobs where you can program in a Functional Language.

Stories of the benefits of using Functional Programming Languages in Greenfield Projects, i.e. projects that are written from start, will need time to be told and retold before the industry will start to consider this approach. Also, more books to help developers get up to speed quickly will need to be written.

That's why I believe that somewhere between 2020 and 2030, Functional Programming will become the paradigm of choice for most Greenfield Projects.

Even then, there will still be a huge need for Imperative Programmers. There's still a need for Cobol

Programmers some 60 years after its incarnation, but these jobs will be maintaining antiquated legacy systems.

Having said all of this doesn't detract from the fact that Functional Programming Languages are readily available today with more than sufficient ecosystems and are being used in Production for Artificial Intelligence, Finance, Cryptocurrencies, Web Services, and Web Applications, to name a few, in companies like Facebook, IBM, Twitter, AT&T, Bank of America, Barclays Capital, NVIDIA, Microsoft and many smaller unknown companies all over the world.

Also, keep in mind that Functional Programming ideas have begun to permeate the language landscape. Many more languages are adopting these concepts. It's only a matter of time.

The programming world is becoming more functional every day. It's better to be ahead of that curve.

Who is This Book For?

This book is for the working programmer. However, that doesn't mean that students cannot benefit. They just won't appreciate the benefits like someone who has suffered and struggled with one or more Object Oriented Programming languages.

My idealized reader is someone who has used at least one of the following languages or something similar: Javascript [6], C#, Java, Python, Ruby, etc. They have been working for at least 2 years as a programmer. They value the same things I listed earlier and are looking to improve themselves and make programming a more enjoyable experience.

They want the most powerful tools at their disposal and are willing to put in the effort upfront to become skilled at using them.

This idealized reader is just the imaginary person that I've written this book for. If that's not you then it's more a sign of a lack of my imagination than yours. Please don't let that discourage you.

From A to Z

I've purchased thousands of dollars worth of books in my career and I've read many different types of books. Here how I'd characterize some of them:

A, B, X, Y, Z

My favorite complaint for many technical books is this:

- Chapter 1: Your A, B, C's
- Chapter 2: Your 1, 2, 3's
- Chapter 3: Differential Calculus

These books always start off simple and lull you into a false sense that you'll be able to follow every step

along the way. The beginning is easy and understandable. So far so good. Then, inevitably, you turn a page and hit a wall.

You frantically turn back to the previous page wondering if you missed something. Maybe your eyes glazed over while you were reading and if you just reread the last few paragraphs, it'll be a smooth transition into the next section.

Alas, it is not to be. You have gone from ***A*** to ***B*** to ***X***, ***Y Z***.

A, B, C

Then there are books that are just not enough. They do a great job of explaining the basics, but don't contain enough information to make you proficient in the subject matter.

I felt this way with many Haskell books. I read them and thought that I had a pretty good understanding. I'd then go online and look at some Haskell code and be completely and utterly lost.

The reason there are so many books like this is because it's easy to talk about simple things and in the beginning of any subject is the easy stuff. So many books that you try to learn from will leave you hanging.

They start off at ***A*** but leave you coming up short at ***C***.

X, Y, Z

Then there are books that you know are wrong for you right from Chapter 1. You read the Preface and Intro and felt that everything was going to be normal. That is until you cracked open the first chapter.

These books start at ***X*** and are clearly advanced.

A to Z

It's easy to claim to take someone from ***A*** to ***Z*** but what's difficult is understanding what exactly is meant by ***Z***.

When I think of ***Z***, I think of someone who could be hired as a Functional Programmer in a Purely Functional Programming Language to write standard Business software, e.g. Front end Web GUIs, command line programs, servers (web or otherwise), etc.

Z should represent what most of us do in our everyday work but using a Functional Language instead. ***Z*** does NOT mean that there isn't more to learn. One thing about Programming is that there's always more to learn.

The goal of most people when they learn a new technology is to be able to use it to solve problems that they encounter daily. That is what ***Z*** means to me and this book will take you to that ***Z***.

My goal here is to take a proficient programmer, i.e. someone who can write code and has done so multiple times with varying levels of success, and take them baby step by baby step from the basics through

intermediate levels at the same rate all the way through to advanced until they can regularly take real world problems and use Function Programming Languages to efficiently and effectively solve them.

Why PureScript?

There are 3 reactions that I imagine people will have when they find out this books is going to use PureScript:

1. PureScript? What is that?
2. PureScript? Why not Haskell or Elm or ...?
3. PureScript? Why PureScript?

So let me answer them one at a time.

What's PureScript?

PureScript is a ***purely*** Functional Language that compiles to Javascript. [7]

It can run in both the Browser and on the Server (via `Node.js`). Pretty much anything that you can write in Javascript can be written in PureScript.

PureScript has stolen a lot from Haskell. In fact, the syntax coloring for Markdown and AsciiDoc (what this book's written with) simply use Haskell's. Most of the developers behind PureScript got their start in Haskell and the PureScript compiler is written in Haskell (as is Elm's compiler).

PureScript has also improved on Haskell making it more powerful in some cases but also making it easier to use.

If you already know Haskell, you can learn PureScript very quickly. If you learn PureScript you can easily learn Haskell and you've surpassed learning Elm.

To get started with PureScript takes very little setup and can be accomplished in as little as a few minutes.

Why not Haskell or Elm or ...?

Haskell

Haskell is an old language at 30 years of age. It started off as a research language and has lots of language extensions that one must learn to do modern programming. This complexity can be distract the student from learning the concepts of Functional Programming.

PureScript has no such extensions.

Haskell also doesn't fair well in the Browser. Sure there are technologies that allow Haskell code to run in the Browser but as anyone in the Haskell community will tell you, they are not for the faint of heart and definitely not for the student.

So if you're a Front End Developer, Haskell doesn't really offer much hope in your everyday work.

PureScript on the other hand, was built from the ground up with the Browser in mind.

Haskell's setup, while greatly improved over the years, is a bit more heavyweight than most other languages. There are so many different options for editor plugins and build systems that it becomes difficult to determine which to use and their reliability is often substandard or requires the student to compile it from source code.

PureScript has only a couple of ways and, in this author's opinion, one good way to build PureScript programs. The choice for editor plugins is smaller and more reliable and has pretty good support for modern day editors.

Haskell has 3 types of Strings (technically 6). This flexibility allows a Haskell developer to use the Type of String that best suits the situation but many times multiple Types of Strings are used in a single application. This complexity can be difficult and distracting for students.

PureScript only has one Type of String.

Elm

Elm is a great ***A***, ***B***, ***C*** language but it doesn't take you all the way. It's meant only for Front End Web work so anyone working on the Backend will not be able to transfer what they learn to their everyday work.

Elm is very opinionated and dictates a single architecture to all Elm Programs. This architecture was developed with Web Front Ends in mind. In my opinion, it is not a great architecture for Front Ends and it's worse for Backends.

Some Other Functional Language

I could give you lots of reasons for why not some other language. Like why I think Scala is a compromise language that prioritized interoperability with Java and because of that, is unnecessarily verbose and complex. Or like how F# only works in Microsoft's .NET virtual machine and like Scala is not a Pure Functional Language.

I could make equally disparaging remarks regarding ReasonML, Exlir, Closjure, Idris, Agda, etc. but the truth of the matter is that I only know Haskell, PureScript and Elm.

To be fair, part of the reason for only knowing these languages is because of the reasons that I just mentioned. I've looked at the others and, in my opinion, they all fall short in one area or another so I didn't learn them.

At the moment, I only program in Haskell for the backend and Elm or PureScript for the front end at work, which gives me a lot of experience with using these languages to solve real world problems.

In the past, I've also programmed in Assembly, C, C++, Pascal, FORTRAN, Cobol, Smalltalk, Java, and Javascript to name a few. I'm am by no means a language expert, but I've used my share of languages to solve real world problems.

Now that I've probably disparaged a few reader's favorite environments (JVM, .NET) or languages, I want to say that the concepts you'll learn here are completely transferable to most, if not all, of the aforementioned.

For example, the Monad shows up in Scala, F#, Elixr (in the form of Libraries) and even in Javascript (Promise is a Monad). So learning these concepts will directly translate over into every one of these languages I so quickly dismissed.

Learning useful ideas is never a waste of time even if you can't use them right away.

Why PureScript?

I've sort of explained along the way the benefits of PureScript related to Haskell or Elm but there are 2 major reasons I chose it over the other two.

First, it's complete. For everything that I want to teach in this book, only Haskell and PureScript are advanced enough that they would work. Keep in mind that both languages are more advanced than what you'll learn in this book. This is a good thing since it means that you can keep growing on your own.

Second, PureScript can run on both the front end and the backend. Haskell only runs on the backend (with much pain on the front) and Elm only runs on the front end (with even more pain on the backend).

Four Part Harmony

This book is broken into 4 parts:

- Beginner
- Intermediate
- Advanced
- Beyond

Beginner

This section will slowly introduce the reader to Functional Concepts that are general to the paradigm and found in nearly all Functional Programming languages while introducing simple illustrative examples in PureScript along the way.

If you already have some Functional experience, you may be tempted to skip this section but I'd suggest that you do not. For two reasons.

First, this is where I will introduce you to PureScript and you'll get a chance to do some coding in PureScript.

Second, and most importantly, reviewing what you already know is always a worthwhile venture. It helps strengthen what you know and many times when I've done this, I've seen something that I thought I fully understood in a different light, solidifying my understanding even further.

Intermediate

Armed with a Functional Foundation and some experience writing simple functions in PureScript, this section will expand on that by delving into Typeclasses, Folds and other Mathematical Concepts that show up in Functional Languages, e.g. Functors.

Don't worry if you're Math phobic, any and all Math concepts will be fully explained with the assumption that the reader doesn't have the requisite background.

Category Theory is a very small portion of this book. I almost left it out completely but if you hope to exist in this field, you will definitely bump into people who do understand Category Theory as it applies to Programming and it will do you good to have a general idea of what it is and how it applies to Functional Programming.

Advanced

I joke that when I learned C, the first thing I learned was "Hello, world!" and when I learned Haskell, it was the last thing I learned.

There's a lot of truth in this. Doing I/O in a Purely Functional Programming language is actually Advanced. Seems crazy at first but by the time you get to that part of the book, I hope that it won't seem so.

This section will delve into the everyday kinds of abstractions that are used to get real work done. The kinds of things you'll encounter on a daily basis. It will give you the tools to accomplish these tasks effortlessly and you'll wonder how you ever got along without them.

Beyond

In this section, we'll build a backend server and its corresponding front end. This will be a very simple program but will help you see how to use full-stack PureScript. We'll use the ***Halogen*** framework for the web interface on the front end and the ***HTTPure*** framework to build our web server on the backend.

Our front end will run inside the Browser and our backend will run inside of Node. [8]

Exercises

I hate Exercises in books. It all started with my Math books in school. They'd only give you the answers not how they solved the problems or what they were thinking along the way or what was learned by solving them. All we got were answers.

To make matters worse, most books would only give you the answers to the odd numbered questions most likely, to discourage cheating, which makes no sense when nearly every Math teacher on the planet lives by the Mantra "Show Your Work".

So the Exercises presented here will be different. I will first give you the problem. Then you will stop reading and try them on your own. If you get stuck along the way, return to the book and keep reading.

Sometimes the very next section will provide a hint but there will always be a section on how to code the implementation one line at a time. While we code together, I will explain the thinking behind each and every step. And when there are reasonable alternative solutions, I will give you those as well.

I will do this for every part of the book, not just a few exercises in the beginning.

So please, please, if you're like me and usually skip the Exercises, I implore you to work through them. I cannot tell you how many times I read something or listened to a lecture and thought I understood everything only to find my self frozen the moment my hands touched a keyboard.

Everything seemed to make sense when I was reading or watching the learning materials but the fact that I couldn't use this new found knowledge told me that I really didn't understand it.

My goal with these Exercises is to take what you've just learned and use it immediately to help solidify it in your mind and make it your own. Along the way, we'll discuss and think about what we just did and what kind of thinking goes into being able to write such code. Sometimes we'll contemplate larger concepts from only a few lines of code in order to really hammer home the right abstractions and models by doing a post-mortem analysis of our work.

Most books don't help you go from a blank page to a complete solution. This book will not be one of those. You will stare at many blank pages and when you get stuck, this book will take you one step at a time, i.e. one line of code at a time, from nothing to a working solution. That's one of the reasons this book is so large.

I've also put in a lot of mistakes. Some I made when I was coding. Others were added on purpose. This is why you will see hundreds and hundreds of compiler errors. Too many books give you an antiseptic experience where all errors have been removed making the process of programming seem like magic or a skill that only a select few can truly master.

I don't care how long you've been programming. Everyone makes mistakes and the sooner that you get used to the compiler errors and determining how to fix them, the better.

So Many Pages

Because of how I've chosen to write this book, it comes in at a hefty size. Please do not be discouraged by this. There are many reasons for the large size.

For one, when I show you how to code the exercises, I will write one line at a time and sometimes only a fraction of that line. Each time I do, I will show you what our function looks like at that step so most of the code gets copied over and over again.

I will also copy code from earlier to minimize the need for scrolling back through the book to understand what is being referenced. This helps you while reading, but definitely adds pages.

Another thing I do is break up the paragraphs into very few sentences. This helps give the reader time to digest the previous idea before being thrust into the next. I'll do this even if the paragraph only has a single sentence.

And finally, the book is mostly code and I've chosen a large font to reduce eye fatigue.

The goal of the look of this book is to have a lot of whitespace. This is to help reduce the amount of effort it takes to read it. The most important part isn't to keep the page count down but to make your job learning the subject matter as easy as I possibly can.

What you bring to the table is far more important. The effort you put in will be directly proportional to the value you get out. Remember to give yourself time; don't rush through the material.

All non-trivial subjects and skills take time. So be patient with yourself, work diligently and take breaks. A lot of learning happens in our sleep as our brains replay the day's events.

Do this and I promise that you will not regret it.

[1] Languages like Javascript, C#, Python and Java are Imperative Programming Languages.

[2] Here ***Z*** represents good enough to write in a Functional Language professionally. My personal goal was to write a Web Server in Haskell.

[3] See Why Functional Programming Matters by John Hughes for one approach to answering this question.

[4] See Phillip Wadler's talk Propositions as Types on YouTube or read his paper of the same name.

[5] We do still have Runtime Errors in our application, but it's almost always a crash statement that we put in to signal that our program logic is flawed or it's in our Javascript code since not everything we need to do can be written in Elm.

[6] Many early examples assume a modicum of Javascript knowledge.

[7] At the time of this writing there are other actively maintained backends to the compiler, e.g. C++11, Go and Erlang.

[8] At the time of this writing, Deno is still in its infancy but I suspect PureScript bindings will emerge for it some point in the near future.

Part I: Beginner

Chapter 1. Discipline is Freedom

In the mid-to-late 1960s, the Structured Programming paradigm aimed to improve the quality of programming while reducing the development time needed. One of the most notable features of Structured Programming Languages such as ALGOL, Pascal and C was not a feature at all. It was the absence of the GOTO statement.

This was highly controversial and hotly debated through the 1970s and well into the 1980s. The arguments against the elimination of GOTO usually focused on problems of the past, e.g. the number of machine instructions produced by GOTO-less code was more than code with GOTOs. The biggest opponents of this movement were, not surprisingly, experienced programmers who were used to using GOTOs to short-circuit their logic whenever it was most convenient. There was less programming overhead to simply pull the trigger on a GOTO and they were not willing to give that up for something they saw as less powerful.

In the examples routinely put forth by proponents, GOTO would prove to be more powerful and more terse than their structured counterparts. However, in practice, this was not the case. The term **Spaghetti Code** described the overuse of GOTO's, which made following a single control flow in a program as difficult as following a single strand of pasta in a bowl of spaghetti.

Today, no one debates the benefits of not using GOTO and it's been decades since a language came out with such a powerful yet dangerous feature. I can personally attest to the fact that we are far better off with its demise. I remember trying to follow Assembly Code, which, by its low-level nature, can only branch with a form of GOTO via Jump Instructions. If the programmer overused Jump Instructions, then the program was an incomprehensible mess and rewriting was much easier than trying to understand how it worked.

You may wonder how programs were written at all in such a wild-west sort of mentality. In the case of Assembly Language, the programmer just had to be very disciplined about control flow. In the case of higher-leveled languages, one would choose a language that was absent of such destructive options. With such options removed, you couldn't accidentally (or purposely) do something that you'd later regret.

This process of using discipline in the absence of appropriate built-in safety measures followed by language developers adding such features into new languages and retrofitting old ones, has been going on ever since.

It's at this point, you may be wondering, what things have I gotten used to doing in my programs that we'll all look back on as dangerous and unnecessary.

1.1. Global State

Global State (or Global Variables) has been known to cause problems in programs for quite some time. Some of these problems are:

- Anyone can change the data at any time in any module
- There can be tight coupling between Global Variables that can easily be broken by non-compliant code
- Concurrency requires contention-management logic that's not enforced

- Variable name collisions can occur since the Variable names are in every namespace

Many would argue that Object-Oriented Programming solves these problems. Unfortunately, it does not. It only hides them.

You can easily create a Singleton Object that contains all of your Globals and then expose them to be public. Now you have the equivalent of Global Variables in any Object-Oriented language. They are simply in the namespace of that Singleton Object, which itself is a Global.

Like with GOTO, the only real way to remove this scourge is to make it impossible to have Global State and Functional Programming Languages do exactly this.

1.2. Mutable State

The evils of Mutable State have also been known for quite sometime and recently we've seen many popular frameworks and languages moving, albeit slowly, towards giving developers tools to help control this.

Some of the problems caused by Mutable State are:

- Code is much harder to reason about since values can change drastically altering the code Semantics
- Code is more fragile since program reasoning is greatly reduced

To partially address these issues, Javascript added `const` (with all of its flaws) and libraries have been developed to support Immutable Data Structures, e.g. the `immutable.js` library.

However, these language features require the developer to be disciplined on a daily basis to avoid mutations and unfortunately, many of the standard library functions will mutate your data structures whether you want them to or not. It's also really easy to use the wrong Function, e.g. `splice` mutates but `slice` does not. In Javascript, you're just one typo away from having a mutation bug in production.

The popular framework React has a general immutable approach to its architecture. The problem is that it does so in a language that is anything but, putting the burden on the developer.

In Functional Programming, everything is immutable but what are the ramifications of removing mutability?

1.2.1. No Variables

Do you remember when you first learned programming and someone showed you the following:

```
x = x + 1
```

They told you to "Forget everything you learned in Math". While it's true that in Math, there is no solution for this formula, i.e. there is no value of `x` such that it equals `x + 1`, in programming, this means something completely different. It basically says, take the value of `x`, add `1` to it, and then place it back into `x`.

In these kinds of programming languages, `x` represents a placeholder in memory where numbers can be stored. We look to the right side of the equation first and evaluate it much like we would in Math and then we take the result of that evaluation and place it into a memory location named `x`.

In Functional Programming, there are no Variables. We still call them Variables like we did in Math but they don't vary like we are used to in Programming. Our thinking has to go back to middle school where we learned Algebra.

Remembering our Algebra word problems:

```
Mrs. Johnson has 20 students in her class. 12 of the students have brown hair,
3 have red and the rest have blonde. How many students have blonde hair?
```

Our fist step in solving this is to define our "Variables". So we'd write down the following:

```
Let s = number of students
Let b = students with brown hair
Let r = students with red hair
Let y = students with blonde hair
```

Then we'd write our formula:

```
y = s - b - r
```

To do this in PureScript, we write something very similar:

```
let s = 20         -- number of students
    b = 12         -- students with brown hair
    r = 3          -- students with red hair
    y = s - b - r -- students with blonde hair
```

Just like in Math, the variable `s` stands for `number of students`. The number of students doesn't change in the middle of our problem. It's always `20`. In Math and in Functional Programming `s` is immutable.

So how did we solve for `y` in Math?

We substituted:

```
y = s  - b  - r
  = 20 - 12 - 3
```

In Functional Programming, the compiler does the same substitution that we learned as kids and we can do this at any time since we don't have to worry that `s` or `b` or `r` will change.

When we say `s = 20`, we aren't asking the compiler to allocate some memory for our `20` and then call it `s`. What we're doing is defining `s` as the value `20`. Technically, `s = 20` is an ***expression***.



An expression is ***Referentially Transparent*** if it can be replaced with its value without changing the program's behavior.

Here `s` is defined to be the same as `20`. It's equal to `20`and since they are equivalent, we can use them interchangeably and whenever we see fit. In Haskell, `s` will be replaced only when it's needed since the language is by default ***lazy***, i.e. it only evaluates what it needs when it needs it.

PureScript by default is ***strict*** meaning it eagerly evaluate all definitions, which means we don't automatically benefit from this runtime optimization. However, we still get all the other benefits of Referential Transparency.

By removing mutability from our language, we gain the ability to do simple substitution at any time we see fit. We also can reason more easily about our code and in many cases can understand code better by doing simple mathematical substitutions that we learned in school.

It's at this point you're probably asking yourself, "How do I do a loop in a language with no Variables". The answer is you can't.

1.2.2. No Loops

In Math, there are no loops but it's still a powerful enough language (yes Math is a language) to define things like loops found in programming.

Let's look at the Factorial Function in Math where `n > 0`:

```
n! = 1 · 2 · 3 ··· (n − 2) · (n − 1) · n        (Eq. #1)
```

So then 5! is:

```
5! = 1 · 2 · 3 · 4 · 5
```

When we write this backwards:

```
5! = 5 · 4 · 3 · 2 · 1
```

we can see the following grouping:

```
5! = 5 · (4 · 3 · 2 · 1)
```

where the parenthesized value is just `4!`:

```
5! = 5 · 4!
```

which leads us to an alternative definition for Factorial:

```
n! = n · (n – 1)!
```

But what happens when `n – 1` is `0`?

How do we calculate `0!`?

We can figure this out by looking at `1!`:

```
1! = 1 · (1 – 1)!
   = 1 · 0!
```

From our original definition of Factorial (see `Eq. #1`), `1! = 1`. So, let's substitute and reduce:

```
1! = 1 · 0!
1  = 1 · 0!
1  = 0!
```

Therefore, we can reasonably define `0!` as:

```
0! = 1
```

So here's our Mathematical Definition of Factorial:

```
0! = 1
For all n > 0, n! = n · (n – 1)!
```

In more Mathematical Symbology:

```
0! = 1
∀n ∈ ℕ, n! = n · (n − 1)!
```

where `∀n ∈ ℕ` means `For all values of n that are elements of the set of Natural Numbers`.

Natural Numbers are also known as Counting Numbers, `1, 2, 3, 4, …`, which is equivalent to `n > 0` from our previous definition.

So how would we define a Factorial Function using loops (in Javascript):

```
const factorial = n => {
  var result = 1;
  for (var i = 1; i <= n; ++i)
    result = result * i;
  return result;
};
```

First notice how complex this definition is compared to our Mathematical one. You might dismiss this as being the expected difference between Math and Programming but it doesn't have to be that way.

You may expect that Functional Programming will magically reduce this complexity. Well, yes and no.

It's not a special feature of Functional Programming that reduces the complexity. Instead, it's just a different way to think about the problem that even most Imperative Programming languages support.



Imperative Programming languages, like Javascript, consist of a set of commands that change the program's state.

Looking back on our final Mathematical definition of the Factorial Function:

```
0! = 1
∀n ∈ ℕ, n! = n · (n − 1)!
```

We can see that Factorial is defined in terms of itself, i.e. there's a Factorial on the right side of the equation. This is known as a ***Recursive Definition***.

In programming, we can write recursive functions (here in Javascript):

```
const factorial = n => n === 0 ? 1 : n * factorial (n − 1);
```

With recursion, our Function parallels the Mathematical definition and is as easy to understand as its

Mathematical counterpart.

Turns out that you don't need loops if you have recursion in your language and all Functional Programming languages support recursion since they don't have loops.

Also, you can turn any loop into a Recursive Function and any Recursive Function into a loop.

The upside to Recursion is that many times it simplifies the Function making it easier to think about than a loop.

A downside to Recursion is that it's not as intuitive as loops but with practice it gets a lot easier. We'll do a whole set of exercises writing recursive functions. So much so that it'll become second nature to think recursively.

Also, many recursive functions have the same pattern so there are library functions that help us leverage these similarities. We will examine these functions in detail as well.

1.3. Purity

Let's look at some simple Mathematical functions:

```
f(x) = 25x - 17
g(y,z) = 42y + z
```

They do the following:

- Take one or more inputs
- Perform a computation
- Return one result

```
f(0) = -17
g(1,2) = 44
```

Functions in Math are very rigorous. They all abide by these restrictions.

However, in Imperative Programming, functions have no such requirements. We've all written functions that return no values. For example, in Javascript:

```
var workCounter = 0;
const doWork = str => {
  ++workCounter;
  console.log("Working with: " ++ str);
};
```

Notice that there is no `return` in this Function. Its return value is `undefined`. So why call it?

We call it for its Side Effects, which are:

- Increment `workCounter`
- Write to the `console`

In Purely Functional Programming Languages based on Lambda Calculus, we have the following rigorous requirements :

- Take one parameter
- Perform a computation
- Return one result

All functions like this are called **Pure**.



A ***Pure*** Function has NO Side Effects and given the same input will ALWAYS produce the same output.

Notice that these requirements differ from the Mathematical ones. Here our functions can only take a single parameter. You might think that this is terribly limiting but it's not.

Turns out that any Function with multiple Parameters can be rewritten as a set of functions that each only take a single parameter. This is a powerful concept called ***Currying***, which we will delve into more later.

Once again, Functional Programming has taken away something that we've all gotten use to, doing Side Effects whenever and wherever we like. If you think about it, any program that does NOT produce Side Effects is producing no valuable work.

It just runs for a while heating up the CPU and then exits with a single value. No files are written. No HTML code is sent or received. No updates are made to the Database. So how is it possible to write anything of any value if you're restricted to only Pure functions?

Trust me, you can. My first real Haskell program was a backend server that had a WebSocket API and maintained a Postgres Database. You'll have to wait a bit to see just how since doing Side Effects is unexpectedly a more advanced subject.

I often joke that when I learned C, the first thing I learned was Hello World and when I learned Haskell, it

was the last.

In the early days of Haskell, there was no way to do Side Effects until they figured out how to do them in a Pure way. I know this sounds like an oxymoron but it's not as you'll see later.

1.4. Optimization

When you only use Pure functions, there are some hidden benefits that you get, viz. the ability to use certain optimizations.



Bad designs produce unexpected consequences whereas good designs produce unexpected benefits.

1.4.1. Memoization

This hard-to-pronounce term defines a technique of storing the results of expensive calculations so that the result can be looked up instead of recalculated.

To better understand this, let's look at a toy example in Javascript:

```
const g = v => {
  for (var i = 0; i < 10000000; ++i) {
    // do some complex calculation ❶
    console.log ("processing index: " + i); ❷
  }
  return v;
};
```

❶ Imagine some really long complex calculation here.

❷ Thanks to this line, g is not a Pure Function.

Unfortunately, since g is not Pure, we cannot Memoize the computation. But why?

If we call g twice with the same v, it will log to the console twice.

If we call g once with v and remember its result (memoize) and then if we simply look up its stored result instead of calling it a second time, we will only log to the console ONCE.

Our program will not behave the same and so we cannot optimize by using Memoization. One could argue that this difference is trivial but just imagine that our Side-Effect was writing to a Database.

Let's change g to be a Pure Function and build a Memoized version of g called mg:

```
const gCache = {}; ❶
const mg = v => {
  if (gCache[v] === undefined) { ❷
    const result = g(v); ❸
    gCache[v] = result; ❹
  }
  return gCache[v]; ❺
};
const g = v => { ❻
  for (var i = 0; i < 10000000; ++i) {
    // do some complex calculation ❼
  }
  return v;
};
```

❶ This is our cache of previously calculated values indexed by our input parameter (I know that this is global but this is Javascript and just an example).

❷ We check to see if we've already called `g` with the parameter `v`.

❸ If we haven't call `g` before, then we do so now and save the `result`.

❹ We put the `result` into our cache.

❺ We return the cached value.

❻ `g` is now Pure since we removed the Side-Effect of logging.

❼ Let's assume this complex calculation is also Pure.

In this case, we don't call `g` directly, unless we want to waste time computing, but instead call `mg`, our Memoized `g` Function.

The first time we call `mg` with `v`, the resulting computation won't be in our cache. So we compute it and save it indexed by the input parameter, `v`. If we call it a second time with the same input parameter, then we can simply look up the previous result and return immediately with our computation.

With Pure functions, an optimization solution could be hand-built or done by a smart Compiler via some form of annotation to mark `g` as expensive.

Without Purity, certain Compiler Optimizations are impossible.

1.4.2. Compiler Optimization

Let's consider another toy example in Javascript:

```
const g = v => {
  // do some complex calculation ❶
  console.log ("g was called with: " + v); ❷
  return v;
};

const f = n => n * g(x) - g(x); ❸
```

❶ Imagine we do a complex calculation here.

❷ `g` is NOT Pure due to this Side-Effect.

❸ There is a case for optimization here when `n = 1`.

It would be nice if the compiler would automatically optimize out the 2 calls to `g` when `n = 1` but it cannot since `g` is NOT Pure.

If `g` isn't called twice in `f`, then our program won't execute the Side-Effect of logging to the console twice. Even though we know the answer is always going to be `0`, we still have to do an expensive calculation twice.

Let's make `g` Pure:

```
const g = v => { ❶
  // do some complex calculation ❷
  return v;
};

const f = n => n * g(x) - g(x); ❸
```

❶ `g` is now Pure having removed the console logging.

❷ Same expensive computation.

❸ Now this can be automatically optimized.

The compiler can optimize `f` by generating code as if we had written the following:

```
const f = n => n === 1 ? 0 : n * g(x) - g(x);
```

Without Pure functions, this sort of automatic optimization is impossible since it's too difficult for the compiler to determine if optimizing will affect the program's behavior.

There are many more optimizations that compilers can make if it knows up front that a Function is Pure. These are just some simple examples.

1.5. Types

There are two kinds of Types in programming languages, ***Static Types*** and ***Dynamic Types***. Static Types are checked at compile time and Dynamic Types are checked at runtime.

Programmers have been debating which is better since the beginning of Programming. And it's no different today.

1.5.1. Signal to Noise

If you've worked with languages that have Static Types, e.g. Java, you may have gotten used to complex types such as:

```
private static final Map<Integer, List<String>> map =
  new HashMap<Integer, List<String>>;
```

The signal to noise ratio is very high. Scala, a Functional/OO hybrid, has equally noisy type definitions:

```
class NonEmpty[A](elem: A, left: Set[A], right: Set[A]) extends Set[A] {
  // redacted
}
```

So you can understand why programmers are willing to abandon Static Types for the simplicity of the following Javascript:

```
var map = {}
```

Or even simpler in Python:

```
dict = {}
```

However, there is a high price to pay for Dynamic Types:

- Type checking is deferred until runtime
- Tests to catch Type errors must be written and maintained, i.e. Technical Debt
- Refactoring is so risky that it's often avoided
- Nearly zero help from the IDE

One could argue that the downsides of Static Types are:

- Limits the programmer's freedom
- Requires explicit Typing
- Signal to Noise is high

1.5.2. Perceived Limits

There's a common belief that with a Dynamically-Typed language, you don't need to know what Type of data you will get until runtime and that this makes Dynamically-Typed languages more flexible than Statically Typed ones.

This just isn't true. It's impossible to write code that can deal with random data structures. What Statically-Typed languages force you to do is to be explicit with Types to guarantee that your program is correct.

Dynamically-Typed languages are great for a quick-and-dirty hack or for q very small program that can be debugged quickly and won't be a maintenance headache, i.e. will never change.

Even so, if you care about code quality and correctness, then you'll choose a Statically-Typed language because it will find a whole set of bugs at compile time as opposed to runtime. Pure Functional Programming languages that are Statically Typed can be automatically tested without burdening the developer with having to write, debug and support Unit Test code.

1.5.3. Cake and Eat It

Some languages force the developer to explicitly define every single Type but there are a whole set of languages, mostly Functional ones, that support ***Type Inference***.



Type Inference frees the programmer from explicitly defining Types since the Compiler can usually infer the Type based on its usage.

Take the following bit of PureScript code:

```
add x y = x + y
```

We can infer that x and y are some sort of Number. A Mathematician would say that x and y are members of a Semi-Ring (more on this much later).

So how did we "know" what Types were involved? We knew because of the + operator. It takes 2 Numbers and adds them together. We inferred based on how x and y were being used and this is, in essence, how the compiler infers types.

Specifying Types is important to help document code, especially with complex Types, but with Type Inference, we don't always have to specify Types, e.g. in the case of short-lived, local Variables. This reduces the burden found in many Statically-Typed languages.

1.5.4. Cutting Through the Noise

Let's revisit our Java example:

```
private static final Map<Integer, List<String>> map =
  new HashMap<Integer, List<String>>;
```

What makes this difficult to read is that the Type information is intertwined with the Variable names. In most ML-based languages, e.g. PureScript, Haskell, and Elm, they separate the Type information, i.e. the ***Type Signature***, from the Function definition.

Here's an overly simplistic definition for `add` in PureScript:

```
add :: Int -> Int -> Int  -- Type definition
add x y = x + y           -- Function definition
```

Notice how the Types are defined on a separate line from the Function. This small change has a huge impact on our ability to understand what this Function does. In fact, sometimes we can tell everything about a Function by just looking at its Type Signature. We'll spend some time looking at Type Signatures and guessing what those functions do later.

If you think that the Type Signature looks strange at first glance, you're not alone. Don't worry, we'll spend plenty of time with Type Signatures until they become second nature.

1.5.5. Static Type Costs and Benefits

Languages that employ Static Types impose upon the programmer an extra burden to define Types and use them when creating new values. The benefit of this disciplined approach is the confidence gained that a program will operate as specified.

The code we will write in PureScript will be mostly debugged using the compiler. I've heard Haskell and Elm programmers say "If it compiles, it works." While this is an exaggeration, I can personally attest to this experience. More times that not, once my program compiles, it just works.

1.6. Summary

A disciplined approach to programming has proven itself over the years . It's not surprising that such discipline is met with resistance from those in the profession who are under pressure daily to produce results. Eventually, the better ideas do make it to industry even if it takes a few decades.

Functional Programming's restrictions are no different from the ones introduced by Structured Programming. Those restrictions made programming better back in the 1960's as Functional Programming is doing today.

We've seen the restrictions that Functional Programming imposes on us in order to make our programs more stable and easier to understand:

- All functions are Pure (no Side Effects)
- All values are immutable (no Variables)
- Iteration is performed via Recursion (no Loop Constructs)

Next, we'll take a look at some power features of Functional Programming that are seldom found in other paradigms.

Chapter 2. The Power of Functions

Way back when I was programming in Java, there were so many times I wanted to refactor my code, but could not. Here's a simple example where I got stuck:

```
public class HelloWorld {
    public static String appendIfSmall(String s, String append) {
        if (s.length() < 10) ❶
            return s + append;
        else
            return s;
    }
    public static String appendIfOddLength(String s, String append) {
        if (s.length() % 2 != 0) ❷
            return s + append;
        else
            return s;
    }
    public static void main(String[] args) {
        System.out.println(appendIfSmall("Hello World", "!!!"));
        System.out.println(appendIfOddLength("Hello World", "!!!"));
    }
}
```

❶ This `if` checks to see if the length is "small".

❷ This `if` checks to see if the length is odd.

The output of the above code is:

```
Hello World ❶
Hello World!!!
```

❶ This doesn't have `!!!` appended to it since it's not "small" since its length is 11.

While this is a toy example, it demonstrates a very real problem. I cannot refactor this code into a single `append` Function. Here's a failed attempt at doing so:

```
public class HelloWorld {

    public static String appendIf(String s, String append) {
        if (??? WHAT DO I PUT HERE ???) ❶
            return s + append;
        else
            return s;
    }
}
```

❶ I can't put BOTH the check for odd length and small length here. Besides, I don't want both checks. Sometimes I want the odd length check and other times I want the small length check. What I'd like is a way to pass in the `if` statement.

To solve this dilemma, we'll need something called a ***Predicate***, i.e. a Function that takes an input Parameter and returns a `Boolean`. Here are two Functions that encapsulate the two `if` checks.

```
public class HelloWorld {
    public static Boolean isSmall(String s) {
        return s.length() < 10;
    }
    public static Boolean isOddLength(String s) {
        return s.length() % 2 != 0;
    }
    // ...
}
```

What I want is a way to pass these Functions as a Parameter to a single `append` Function. Back when I was programming in Java, there was no way to do this. Now, Java has adopted a very powerful feature that Functional Programming has had for over 50 years, i.e. passing Functions as Parameters.

2.1. Functions as Parameters

In Functional Programming, Functions are first-class citizens. This idea has been around since the late 1950s. In Lisp, the saying goes:



Code is Data, Data is Code

This means that code can be manipulated like data and data can be executed.

While this is powerful, it's terribly dangerous and in very rare cases, e.g. Genetic Algorithms, it's a feature that one should generally avoid.

However, it's the idea that Functions are no different from `Integers` where the real power is. In our earlier `append` example, we struggled to pass a Predicate Function because the language simply had no mechanism.

Now that Java pilfered this feature from Functional Languages, we can write:

```
import java.util.function.Function;

public class HelloWorld {

    public static Function<String, Boolean> isSmall =
        s -> s.length() < 10; ❶
    public static Function<String, Boolean> isOddLength =
        s -> s.length() % 2 != 0; ❶
    public static Boolean isSmall(String s) {
        return s.length() < 10;
    }
    public static String appendIf(Function<String, Boolean> pred,
                                  String s, String append) {
        if (pred.apply(s)) ❷
            return s + append;
        else
            return s;
    }
    public static void main(String[] args) {
        System.out.println(appendIf(isSmall, "Hello World", "!!!"));
        System.out.println(appendIf(isOddLength, "Hello World", "!!!"));
    }
}
```

❶ These special types of `Functions` in Java can be passed to `appendIf` just like we passed the `Strings`, `s`, and `append`.

❷ We apply the Parameter, `s`, to our Predicate Function, `pred`.

I wish I had this facility back in the early days of Java but this kind of thing is commonplace in Functional Languages.

Let's look at the PureScript equivalent:

```
module Main where

import Prelude

import Data.String.CodePoints (length)
import Effect (Effect)
import Effect.Console (log)

isSmall :: String -> Boolean ❶
isSmall s = length s < 10

isOddLength :: String -> Boolean ❶
isOddLength s = length s `mod` 2 /= 0

appendIf :: (String -> Boolean) -> String -> String -> String
appendIf pred s append = if pred s then s <> append else s

main :: Effect Unit
main = do
  log $ appendIf isSmall "Hello World" "!!!"
  log $ appendIf isOddLength "Hello World" "!!!"
```

❶ These Functions are just run-of-the-mill Functions. No special syntax needed.

I don't expect you to understand this code just yet but I wanted to show you the terseness of PureScript compared with Java. Granted, if you're a Java programmer, the PureScript version will feel very foreign at first.

Regardless, there's a beautiful lack of parentheses, commas, semi-colons, curly-brackets and accessor types, i.e. `public` and `static`. While this may be jarring at this point, I promise that you will get spoiled by this very quickly.

Where Java uses `Function<String, Boolean>`, PureScript uses `String -> Boolean`.

In Java, there is a special protocol to use a Function that's been passed as a Parameter, `pred.apply(f)`. In PureScript, it's no different than calling any other Function, `pred f`.

Just for fun, let's look at the same code in Haskell:

```haskell
module Main where

import Prelude hiding (pred) ❶ ❷

isSmall :: String -> Bool ❸
isSmall s = length s < 10

isOddLength :: String -> Bool
isOddLength s = length s `mod` 2 /= 0

appendIf :: (String -> Bool) -> String -> String -> String
appendIf pred s append = if pred s then s <> append else s ❷

main :: IO () ❹
main = do
  putStrLn $ appendIf isSmall "Hello World" "!!!" ❺
  putStrLn $ appendIf isOddLength "Hello World" "!!!"
```

❶ In PureScript, there's a finer granularity of libraries, so there's more to import.

❷ We hide `pred` from the import of `Prelude` because we don't want to conflict with our `pred` in `appendIf`.

❸ `Bool` is the Type for Booleans in Haskell.

❹ Instead of `Effect Unit`, we do effects in Haskell using `IO ()` where `Unit` is the equivalent to `()` in Haskell.

❺ In PureScript, we need to specially import `log` from `Effect.Console` because we could be writing for the Browser so `log` isn't as readily available as the Haskell equivalent, `putStrLn`.

While I've glossed over a lot of details, which we will get to later in this book, I hope seeing how similar PureScript and Haskell are will give you confidence that once you're done with this book, you can easily transition over to writing in Haskell.

Unfortunately, I cannot easily give you an example in Elm because Elm doesn't have facilities for outputting to the console since it can only run inside the Browse but the syntax is very similar.

The important thing here is that all three languages support the ability to pass Functions as Parameters.

2.2. Functions as Return Values

If Functions are Values and Functions return Values, can Functions return other Functions?

Yes we can and as it turns out, Javascript also supports Functions as Values. So here's an example in Javascript of a Function returning another Function:

```
const isEven = n => n % 2 === 0;
const upperLower = n => { ❶
  if (isEven(n)) return String.prototype.toUpperCase;
  else return String.prototype.toLowerCase;
};

console.log(upperLower(0).apply("this should be output in uppercase"));
console.log(upperLower(1).apply("THIS SHOULD BE OUTPUT IN LOWERCASE"));
```

❶ If n is even then we return a Function to convert strings to uppercase. Otherwise, we return a Function to convert strings to lowercase.

Let's see how we'd write this in PureScript:

```
module Main where

import Prelude

import Data.String.Common (toLower, toUpper)
import Effect (Effect)
import Effect.Console (log)

isEven :: Int -> Boolean
isEven n = n `mod` 2 == 0

upperLower :: Int -> (String -> String)
upperLower n = if isEven n then toUpper else toLower

main :: Effect Unit
main = do
  log $ upperLower 0 "this should be output in uppercase"
  log $ upperLower 1 "THIS SHOULD BE OUTPUT IN LOWERCASE"
```

Once again, the PureScript example will not completely make sense to you yet but we're only concerned with how it contrasts with Javascript.

It's also possible that you are not familiar with Javascript either and in that case, the best you can hope to discern from these two examples is that the syntax in PureScript is far less dense than in Javascript.

You may also notice that there's a lot more overhead in PureScript because we have to import libraries. In this particular example, Javascript has all of the Functions we need already built-in. This is not always the case.

If you've spent any time working with Javascript, you're well aware of how complicated adding libraries

can be especially if you're targeting the Browser but using a library originally written for Node.

Such is the life of a front-end developer these days.

2.3. Higher-order Functions

To summarize, Functional Languages treat Functions as Values, which means that they can be passed to other Functions as Parameters and can be returned from other Functions.

The technical term for this is ***Higher-order Function***.



A ***Higher-order Function*** is a Function that takes a Function as a Parameter, returns a Function or both.

In contrast, Functions that do not do this are known as ***First-order Functions***.

Look back at the PureScript code and see if you can find the Higher-order Functions and the First-order Functions.

If you get stuck, try looking at the Javascript code. There are two First-order Functions and one Higher-order.

Let's see how you did:

```
module Main where

import Prelude

import Data.String.Common (toLower, toUpper)
import Effect (Effect)
import Effect.Console (log)

isEven :: Int -> Boolean ❶
isEven n = n `mod` 2 == 0

upperLower :: Int -> (String -> String) ❷
upperLower n = if isEven n then toUpper else toLower

main :: Effect Unit ❶
main = do
  log $ upperLower 0 "this should be output in uppercase"
  log $ upperLower 1 "THIS SHOULD BE OUTPUT IN LOWERCASE"
```

❶ First-order.

❷ Higher-order.

In PureScript, the syntax `(String -> String)` stands for a Function that takes a `String` and returns a `String`, and since this is the return type of `upperLower`, it means that `upperLower` is a Higher-order Function.

Don't worry if this isn't making much sense just yet, because we'll delve into Type Signatures later, which will clarify all of this. We're just dabbling in the code to help us get acclimated to PureScript.

2.4. Composition

Building larger things from smaller components is seen in nature, engineering, manufacturing and in programming. Building larger, more complex Functions by composing smaller, simpler ones is a very powerful technique.

In Math, we learned, and most likely forgot, that we can combine two Functions using Functional Composition:

```
f : Y → Z, g : X → Y ⇒ f ∘ g (x) = f(g(x)) : X → Z ∀x ∈ X
```

If you're not used to reading Mathematical statements, this can be daunting. So let's break this syntax down:

```
f : Y → Z
```

This states that `f` is a Function that maps elements from the Domain `Y` to the Codomain `Z`. Don't forget that `Y` and `Z` are Sets, e.g. `Y` could be the set of Real Numbers, `ℝ` (this is the symbol Mathematicians use to denote Real Numbers).

What Mathematicians call `Domain` and `Codomain`, we programmers call `Input` and `Output` respectively.

What does `maps` mean in this context?

It means that `f` takes an element of `Y`, traditionally depicted by `y` and produces an element of `Z`, usually depicted by `z`.

The arrow syntax is very similar to the arrow syntax we've seen used in Haskell and PureScript when defining its Functions. For example:

```
negate :: Int -> Int
negate n = (-1) * n
```

`negate` is just like `f` above. Its Domain or Input is `Int` and its Codomain or Output is `Int`.

Let's continue parsing:

```
g : X → Y
```

This states that g is a Function that maps elements from the Domain X to the Codomain Y.

Notice how g's Codomain is the same as f's Domain, i.e. the Output of g is the Input of f. This is a critical requirement if we are to *glue* these two Functions together.

```
f ∘ g (x) = f(g(x)) : X → Z ∀x ∈ X
```

The first portion states that f composed with g applied to x is equal to applying x to g and then applying those results to f. The ∘ symbol is the compose operator.

The remainder of this line states that the `f composed with g` maps elements from the Domain X to the Codomain Z for all elements x that are in X.



Functional Composition works from Right to Left.

```
f ∘ g
```

This is spoken as f ***COMPOSED WITH*** g or f ***AFTER*** g or f ***FOLLOWING*** g. The words ***AFTER*** and ***FOLLOWING*** help to remind us that composition is Right to Left.

```
f ∘ g (x)
```

This is spoken as f ***COMPOSED WITH*** g ***APPLIED TO*** x.

Now let's attack the whole thing:

```
f : Y → Z, g : X → Y ⇒ f ∘ g (x) = f(g(x)) : X → Z ∀x ∈ X
```

This states that if f maps Y to Z and g maps X to Y then f ***COMPOSED WITH*** g applied to x is equal to x applied to g applied to f, which maps X to Z for all x that are elements of X.

That's a mouthful when written out. You can see why Mathematicians use symbols like programmers do.

Speaking of programmers, how do we compose two small Functions into a larger one in code?

Here's an example in PureScript:

```
toString :: Int -> String ❶
toString n = show n

toArray :: String -> Array String ❷
toArray s = [s]

intToStringArray :: Int -> Array String ❸
intToStringArray n = toArray (toString n) ❹
```

❶ `toString` maps `Int` to `String`, i.e. give this Function an `Int` and it'll return a `String`.

❷ `toArray` maps `String` to `Array String`, i.e. given a `String` it'll return an array of `String`.

❸ Maps `Int` to `Array String`.

❹ `intToStringArray` is a MANUAL composition of `toString` and `toArray`. Notice how this takes an `Int` and uses `toString` to produce a `String` that is then passed to `toArray`, which produces an `Array String`, i.e. our return Value.

Let's rewrite `intToStringArray` using PureScripts Composition Operator, `<<<`:

```
intToStringArray :: Int -> Array String
intToStringArray = toArray <<< toString ❶
```

❶ Here we are composing the 2 Functions using `<<<`. Notice how the Composition Operator helps to remind us that Functional Composition works from Right to Left like it does in Math. But in PureScript, unlike Math, we can also go left to right using `>>>`.

Let's revisit a shortened version of the Mathematical definition from above:

```
f : Y → Z, g : X → Y ⇒ f ∘ g : X → Z
```

Notice how the ***OUTPUT*** of `g` matches the ***INPUT*** of `f`. The PureScript compiler would give us an error if `toString's` Output Type didn't match `toArray's` Input Type.

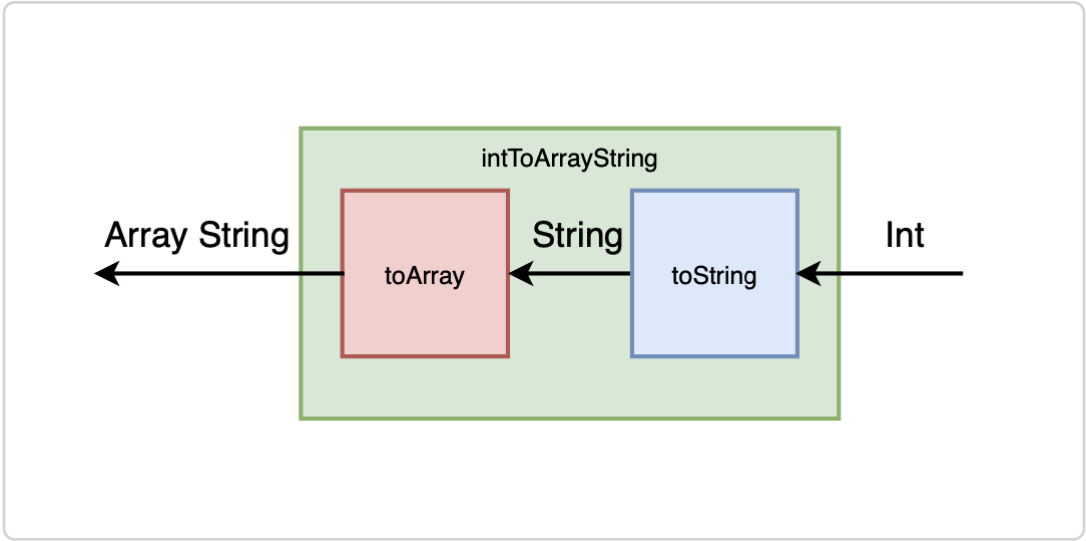
To help show how code fits together, take the following code:

```
toString :: Int -> String
toString n = show n

toArray :: String -> Array String
toArray s = [s]

intToStringArray:: Int -> Array String
intToStringArray = toArray <<< toString
```

and display it pictorially:



`toString` takes an `Int` and outputs a `String`, which is the input to `toArray` that outputs `Array String`.

Composing `toArray <<< toString` gives us `intToStringArray` which takes an `Int` and produces an `Array String`.

Composing Functions is a powerful feature of Functional Languages and is used often in most Functional codebases. Haskell uses the `.` operator to mimic the Mathematical ∘, whereas PureScript uses the `<<<` operator to help us remember that Composition in Math is from Right to Left:

```
notLarge :: Int -> Boolean
notLarge = not <<< isLarge ❶
```

❶ You don't need to use `<<<` with `not` in this example. It's just used here as an example. The code could have been just `not isLarge`.

PureScript also provides the `>>>` operator when we want to compose in the opposite direction. This is usually done because the code reads better. In the above example, it reads better using `<<<`. As we progress, we'll see examples where `>>>` is preferable.

2.4.1. Point-free Notation

Notice how our `notLarge` Function has NO formally defined Parameters. That's because it's written in a form called ***Point-free***.



Point-free notation (or style) means that a Function is defined WITHOUT explicitly mentioning one or more of its Parameters.

We could write `notLarge` by explicitly mentioning its Parameters but now we have the burden of naming and remembering them:

```
notLarge :: Int -> Boolean
notLarge num = not (isLarge num) -- NOT Point-free
```

This version is not as easy to read as our Point-free one. Writing a Point-free version of a Function doesn't always make the Function easier to read but when it does, it's definitely worth doing.

Most Imperative languages don't have direct support for Composition or Point-free Notation. For example, in Javascript, `notLarge` must be written like:

```
const notLarge = num => not (isLarge num);
```

Compare that with our PureScript, Point-free version:

```
notLarge :: Int -> Boolean
notLarge = not <<< isLarge
```

The Point-free version reads like English (`notLarge` is `not <<< isLarge`). This is why I chose to use `<<<` as opposed to `>>>`, which reads backwards (`isLarge >>> not`).

Now, let's assume that you wrote the following Function to pad a `String` with zeros on the Left to a specified size:

```
zeroPad :: Int -> String -> String
zeroPad size s = padLeft '0' size s
```

Is it possible to rewrite this Function to be Point-free? The fact that `size` and `s` appear on both sides of the equal sign should signal that it ***may*** be possible.

If you don't see how to write this Point-free, then maybe adding redundant Parentheses will help:

```
zeroPad :: Int -> String -> String
zeroPad size s = (padLeft '0') size s
```

If we look at our Function as a simple equation, we can see that `(padLeft '0')` is equal to `zeroPad`, giving us:

```
zeroPad :: Int -> String -> String
zeroPad = padLeft '0'
```

Now we have `zeroPad` in Point-free notation. It also reads better (`zeroPad` is `padLeft` with `0`).

Notice how we didn't touch the signature. That's because the Function still takes those Parameters. They're just Unnamed Parameters.

There's a simpler way to look at this sort of reduction using what I call ***Cancel on the Right***:

```
zeroPad :: Int -> String -> String
zeroPad size s = padLeft '0' size s
```

First, notice how `s` is the rightmost Parameter on both sides of the equal sign. That means we can cancel it from both sides of the equation:

```
zeroPad :: Int -> String -> String
zeroPad size = padLeft '0' size
```

Now `size` is the rightmost Parameter on both sides of the equal sign, so we cancel it as well:

```
zeroPad :: Int -> String -> String
zeroPad = padLeft '0'
```

Since there are no more Parameters on the left side of the equation, `zeroPad` is in its most reduced Point-free form.

The technical term for this reduction is ***Eta-Reduction*** (notated by `η-reduction`). The name comes from the use of the 7th letter of the Greek Alphabet, ***Eta***, which is uppercase `H` or lowercase `η`. This term is from ***Lambda Calculus***, which an area of Mathematics that Functional Programming has "borrowed" heavily from.

One thing to keep in mind is that the order of Parameters matters for Eta-Reduction. Imagine that `zeroPad` had a slightly different definition:

```
zeroPad :: String -> Int -> String
zeroPad s size = padLeft '0' size s
```

In this case, we would be unable to reduce this because `size` is the rightmost Parameter on the left-side of the equation but `s` is the rightmost on the right-side. Just like in Algebra, we cannot cancel `size` on one side with `s` on the other.

Many times when coding, I'll change the order of my Parameters to allow for this cancelation to help reduce the complexity of my Function definitions.

2.5. Currying

Do you remember that one of the restrictions of Functional Programming languages is that all Functions only take 1 Parameter? Well, this restriction comes from Lambda Calculus, which was created by the mathematician ***Haskell Curry***.

In Lambda Calculus, Functions can only take one Parameter. They are written in the following mathematical notation:

```
λx. x + 1
```

In PureScript, we'd write this as:

```
\x -> x + 1
```

Notice how the `\` sort of looks like a Lambda, `λ`. The `.` is replaced by `->`, but the idea is the same.

In both cases, this defines an anonymous Function that takes a single argument and adds 1 to it.

How do we write a Function in Lambda Calculus that adds two numbers if our Functions can only take one Parameter?

Don't forget that Functions can return other Functions:

```
λx. λy. x + y
```

Here is the PureScript equivalent:

```
\x -> \y -> x + y
```

When this Function is called with 3:

```
(\x -> \y -> x + y) 3
```

the value 3 binds to the variable x and returns the following Function:

```
\y -> 3 + y
```

When we call this returned Function with 5:

```
(\y -> 3 + y) 5
```

the value 5 binds to y returning:

```
3 + 5
```

since + is a Function, it returns:

```
8
```

In PureScript (and in Haskell and Elm) we can write:

```
\x y -> x + y
```

While this is illegal in Lambda Calculus, in programming, we are able to define multiple Parameters to reduce visual complexity. How is this possible when we stated earlier that all Functions in Functional Programming only take a single Parameter.

This is where ***Currying*** comes in. Currying takes a Function like:

```
\x y -> x + y
```

and turns it into a Function like:

```
\x -> \y -> x + y
```

The compiler does this for us ***automatically*** so we can have the nicety of writing Functions with far less

cruft.

Let's look at a named Function:

```
add :: Int -> Int -> Int ❶
add x y = x + y
```

❶ The Type Signature is Right-Associative, which implies that this Function is Curried.

All Type Signatures have implied Parentheses that are Right-Associative, i.e. associates to the right:

```
add :: Int -> (Int -> Int)
add x y = x + y
```

With the implied Parentheses explicitly defined, we can see how this Function only takes 1 Parameter and returns a Function that also only takes one Parameter, an `Int` and returns an `Int`.

Here's another Function with more Parameters and explicit, Right-associative Parentheses:

```
add3 :: Int -> (Int -> (Int -> Int))
add3 x y z = x + y + z
```

With explicit Parentheses, we can see that Functions only take a single Parameter and return a Function that only takes 1 Parameter until the final Value is returned, which in the above case is an `Int`.

Here's another example with explicit Parentheses:

```
show3 :: Int -> (Int -> (Int -> String))
show3 x y z = show x <> "," show y <> "," show z ❶ ❷
```

❶ `show` takes our `Ints` and turns them into `Strings`.

❷ `<>` concatenates our `Strings`.

When Parentheses are not right-justified, the meaning changes:

```
f :: Int -> (Int -> String) -> String
```

- The first Parameter is `Int`.
- The second Parameter is `Int -> String`
- The result is `String`

This means that the second Parameter is a Function, making `f` a Higher-order Function.

What about a Function like this:

```
g :: ((Int -> Int) -> Int) -> Int
```

- The first Parameter is `((Int -> Int) -> Int)` which takes a Function from `Int -> Int` and returns an `Int`.
- The result is `Int`.

What are the implied Parentheses for:

```
h :: Int -> String -> String -> String
```

Remember they are Right-Associative meaning associate using Parentheses to the right.

So you start on the right side and add the first set of Parentheses:

```
h :: Int -> String -> (String -> String)
```

Then do it again:

```
h :: Int -> (String -> (String -> String))
```

We can stop once we have a Function that takes only 1 Parameter. In this case, `h` takes an `Int` and returns a Function of type `String -> (String -> String)` so we can stop.

These implied Parentheses are there because Functions can only take a single Parameter. The important thing to note is that variables get bound along the way. For example, returning to `add3`:

```
add3 :: Int -> Int -> Int -> Int
add3 x y z = x + y + z
```

When we call `add3` with one Parameter:

```
add3 1 :: Int -> Int -> Int
```

`x` is bound to `1`. When we call that resulting Function with one Parameter:

```
(add3 1) 2 :: Int -> Int
```

y gets bound to 2. Calling that resulting Function with one Parameter:

```
((add3 1) 2) 3 :: Int
```

z gets bound to 3. The Function is now ***Fully Applied*** and returns a value of 6, which is of Type `Int`.

2.5.1. Partial Application

Returning to our `zeroPad` Function:

```
zeroPad :: Int -> String -> String
zeroPad = padLeft '0' ❶ ❷

padLeft :: Char -> Int -> String -> String
padLeft padCh size s = ... ❸
```

❶ `padLeft` is given only 1 of the many Parameters it will need to produce a final result.

❷ We have partially applied `padLeft`.

❸ We're not concerned with `padLeft's` implementation at the moment.



Partial Application is when a Function is called with fewer than all of its Parameters.

When we think of Functions having multiple Parameters, then Partial Application makes sense. However, when we think of Curried Functions, there is no such thing as Partial Application since there's only one Parameter.

The power of Partial Application may not be immediately apparent but consider a simple example:

```
pad :: Boolean -> Char -> Int -> String -> String
pad rightSide padCh size s = ... ❶

padLeft :: Char -> Int -> String -> String
padLeft = pad false ❷

padRight :: Char -> Int -> String -> String
padRight = pad true ❷

zeroPad :: Int -> String -> String
zeroPad = padLeft '0' ❸

spacePad :: Int -> String -> String
spacePad = padRight ' ' ❹

dotPad :: Int -> String -> String
dotPad = padRight '.' ❹
```

❶ We're not concerned with `pad's` implementation at the moment.

❷ Partial Application of `pad`.

❸ Partial Application of `padLeft`.

❹ Partial Application of `padRight`.

I like to think of Partial Application as configuring a more general Function. We're sort of ***baking*** in the configuration Values.

Notice that Partial Application is only possible because those "configuration Parameters" are specified first. `pad's Boolean` Parameter lets us define `padLeft` and `padRight` and then their `Char` Parameter lets us define `zeroPad`, `spacePad` and `dotPad`.

Here order is important. Imagine if the `Char` came before the `Boolean`:

```
pad :: Char -> Boolean -> Int -> String -> String
pad padCh rightSide size s = ...

padZero :: Boolean -> Int -> String -> String ❶
padZero = pad '0'

padSpace :: Boolean -> Int -> String -> String ❶
padSpace = pad ' '

zeroPad :: Int -> String -> String ❷
zeroPad = padZero false

spacePad :: Int -> String -> String ❷
spacePad = padSpace true

padDot :: Boolean -> Int -> String -> String ❶ ❸
padDot = pad '.'

dotPad :: Int -> String -> String ❷ ❸
dotPad = padDot true
```

❶ We have to write these 3 Functions to bake in the pad character first.

❷ Next, we write these 3 Functions that specifies which side to pad on.

❸ To implement `dotPad`, we have to write an extra Function, `padDot` since the pad character is `pad's` first Parameter.

This approach forced us to write a new Function for each new pad character. Our original approach was better since it let us create `padLeft` and `padRight` BEFORE adding the pad character. Now we can leverage those Functions anytime we want to add a new pad character Function.

It's worth spending some time planning the order of your Parameters for this very reason.



The general rule for Parametric Order is to have the Parameters that change the ***least*** be the ***leftmost*** ones and the ones that change the ***most*** to be ***rightmost***.

Revisiting the better version of our padding Functions:

```
pad :: Boolean -> Char -> Int -> String -> String
pad rightSide padCh size s = ...

padLeft :: Char -> Int -> String -> String
padLeft = pad false

padRight :: Char -> Int -> String -> String
padRight = pad true

zeroPad :: Int -> String -> String
zeroPad = padLeft '0'

spacePad :: Int -> String -> String
spacePad = padRight ' '

dotPad :: Int -> String -> String
dotPad = padRight '.'
```

If you look at `pad`, you'll notice that the `size` Parameter comes before `s`, the `String` we want to pad. This is because we may want to write the following Function:

```
dotPad20 :: String -> String
dotPad20 = dotPad 20
```

Now, we can pass a bunch of `Strings` to `dotPad20`. The `String` we want to pad is the thing that will change the most and, therefore, should be dead last in our Parameter List.

All of the other Parameters can be thought of as Configuration Parameters to tell `pad` how to pad whereas `s` is what to pad and since it will change the most, it's the rightmost Parameter.

Sometimes you'll have 2 Parameters that always change and, in that case, just make sure that they are last. Their respective order won't matter.

Chapter 3. The Basics of PureScript

Programming in C by Kernighan and Ritchie started a tradition when learning a new languge by starting with a program that prints `Hello, World!`.`

So here it is:

```
module Main where ❶

import Prelude ❷

import Effect (Effect) ❸
import Effect.Console (log)

main :: Effect Unit ❹
main = log "Hello, world!" ❺
```

❶ Every `module` starts with its definition. Here we called it `Main` and it would be placed in a file `Main.purs`.

❷ You import libraries using `import`. Here we're importing the standard base library called `Prelude`.

❸ We also import a few more libraries to do Side-Effects, i.e. print to the console. We will revisit Side-Effects in detail much later in this book.

❹ This states that `main` is a Side-Effect Function whose computation is `Unit`, the Functional equivalent of `void` in other languages, e.g. C, C++ and Java.

❺ This logs to the console our message to the world.

I/O like this is an advanced subject in Functional Programming, since it's a Side-Effect, there's a lot here that you won't fully understand just yet.

This shouldn't stop us from using the `log` Function for quick and dirty testing of the Functions that we write for exercises. I promise that once we get through Monads, this code will make a lot more sense.

To start with Monads would be like starting with High School before Kindergarten, so you'll just have to trust my superficial explanations for a bit.

Instead of digging into `Hello, world!`, we're going to start by learning about Types, one of the most important parts of PureScript and Haskell since they are both Statically Typed languages.

We'll return to the subtleties of `Hello, world!` in the Advanced part of this book.

3.1. Types

Most of PureScript's code is open source and can easily be perused via the GitHub repositories, but the

implementation for Primitive Types are built into the compiler, which is written in Haskell.

In PureScript, Types always start with an uppercase letter.

3.1.1. Javascript Primitives

The following Primitive Types map directly to Javascript's types:

- Boolean
- Char
- String
- Number

Boolean Type

`Boolean` values are `true` and `false`. Unlike Haskell and Elm, these are true primitives since they compile directly to Javascript's boolean values.

```
t :: Boolean
t = true

f :: Boolean
f = false
```

Char Type

`Char` is defined with single quotes:

```
ch :: Char
ch = 'a'

unicodeCh :: Char
unicodeCh = '\x00E9'
```

String Type

`String` is defined with double quotes:

```
s :: String
s = "This is a multi-line string\nwith embeded newlines"

s2 :: String
s2 = "This is a multi-line string with continuations\
     \ at the end of the lines" ❶

s3 :: String
s3 = """ ❷
This is a multi-line that can contain quotes "" but \n will not be a newline
"""

unicodeStr :: String
unicodeStr = "This is a unicode character: \x00E9" ❸
```

❶ Characters to the left of `\`, in this case spaces, are NOT part of the string.

❷ Triple quotes can have newlines in them. The escape sequence `\n` will NOT be interpreted as a newline but will be interpreted as the literal characters, `'\'` and `'n'`.

❸ Unicode characters can be embedded with the escape sequence `\x` followed by a 4-hex-digit code.

Number Type

`Number` compile directly to Javascript representation for numbers:

```
import Prelude ❶

n :: Number
n = 1.0 ❷

smallestNumber :: Number
smallestNumber = (-5e-324) ❸

largestNumber :: Number
largestNumber = 1.7976931348623157e+308
```

❶ Imports `Prelude` for the negate operator, i.e. `-`.

❷ Must have decimal point otherwise interpreted as `Int`.

❸ Negative numbers MUST be in Parentheses. You will forget this. I know I still do from time to time. So, I'll say it again. Negative numbers MUST be in Parentheses.

3.1.2. PureScript Primitives

These are the PureScript-specific primitives, i.e. they don't map ***directly*** to Javascript's types:

- Int
- Array
- Record

Int Type

The underlying representation of `Int` is Javascript's number type but all of its operators, e.g. `+`, `-`, etc., will limit the results to only `Int` values:

```
import Prelude

i :: Int
i = 42 ❶

i2 :: Int
i2 = 1 + 4

smallestInt :: Int
smallestInt = (-2147483648) -- -2^31 ❷

largestInt :: Int
largestInt = 2147483647   -- 2^31 - 1
```

❶ No decimal point otherwise it'll be interpreted as `Number`.

❷ Just in case you've already forgotten, negative numbers MUST be in Parentheses.

Even though Javascript has a range of -2^{53} to $2^{53} - 1$ for Integers with no loss of accuracy, PureScript limits `Int` to 32-bits, i.e. -2^{31} to $2^{31} - 1$.

There is, however, a PureScript library called `purescript-int-53` that supports the full 53-bit Integer that Javascript supports. This increases the range to from `-9,007,199,254,740,991` to `9,007,199,254,740,991`.

Array Type

The `Array` Type is represented by Javascript arrays at runtime but, unlike Javascript, the elements must be ***Homogenous*** (***Homo***, meaning ***same*** and ***genos*** meaning ***a kind***), i.e. Arrays must contain elements of the same Type.

```
mta :: Array Number
mta = [] ❶

a :: Array Int
a = [1, 2, 3]

a2 :: Array String
a2 = ["abc", "123"]

aa :: Array (Array Int) ❷
aa = [ [1, 2, 3], [4, 5], [6, 7, 8, 9] ]
```

❶ An empty `Array` of `Number`.

❷ An `Array` of an `Array` of `Int`, i.e. array of arrays.

Record Type

`Record` is an aggregate Type with named fields that is represented by Javascript objects at runtime.

```
r :: { firstName :: String, lastName :: String }
r = { firstName: "Joe", lastName: "Mama" }

type Person = ❶
  { name :: String
  , age :: Int
  }

r2 :: Person ❶
r2 = { name: "Jane Doe", age: 37 }

type Nested = ❷
  { val :: Int
  , rec ::
      { val2 :: Int
      , name :: String
      }
  }
```

❶ You can define a Type Alias using the `type` keyword. Here `Person` is an alias for the Record. You can imagine copying and pasting the Record everywhere the Type Alias name is used.

❷ This is a nested record. The field `rec` is itself a Record. Notice the formatting of the records. They are defined on multiple lines. This formatting is idiomatic PureScript formatting.

Syntactical Oddity

To create a `Record`, we use the following syntax:

```
person = { name: "Candy Cane", age: 37 }
```

To modify a one or more elements in the `Record` and return a ***new*** `Record` (remember, all Values are immutable), we use the following syntax:

```
newPerson = person { name = "Randy Cane" }
```

This is the ONLY time an equal sign is used in `Record` syntax. All other times, a colon is used. This is another nuance that's easy to forget.

If you know Javascript, using a `:` won't be foreign but using the `=` for updating a `Record` will be.

3.1.3. User Types

PureScript comes with many useful built-in Types but without the ability to make your own Types, you won't be able to model your problem domain. PureScript has the following facilities for defining your own Types:

- Type Alias
- Data Type
- Algebraic Data Types
  - Product Type
  - Coproduct or Sum Type
- New Type

Type Alias

A Type Alias allows you to make an alias for another Type. Think of it as shorthand for a more complex Type.

```
type Id = String

type Message = { id :: Id, payload :: String }

type MessageHandler = Message -> Result

handler :: MessageHandler ❶
handler messageHandler = ...

handler' :: Message -> Result ❶ ❷
handler' messageHandler = ...
```

❶ These two Type Signatures are identical. The compiler substitutes `MessageHandler` with its definition, `Message -> Result`.

❷ PureScript allows you to name a Function with a ***prime***, i.e. the apostrophe character. Traditionally, this is at the end of the Function name and is related to the unprimed version in some way.

Data Type

We can make our own Data Types from scratch. Here is the simplest example:

```
data MyType = MyType
```

Notice `MyType` is on both sides of the equal sign.

The `MyType` on the left-hand side is defining a new ***Data Type*** called `MyType`.

On the right-hand side is `MyType` again but this defines the ***Data Constructor***.

The namespace for Data Types and Data Constructors are separate so there is no name collision here. You cannot use a Data Constructor where a Data Type is used and vice versa, which is why you will often see the names reused on both sides of the equal sign. Many times the word, Data, will be dropped and these will be referred to as Type and Constructor.

In PureScript, Data Types and Data Constructors always start with an uppercase letter, whereas variables start with a lowercase letter or an underscore:

```
x :: Int
x = 100

_b :: Boolean
_b = true

data MyOtherType = MyOtherType
```

Algebraic Data Types (ADTs)

There are 2 types of `Algebraic Data Types`, `Product Types` and `Coproduct Types` (also called `Sum Types`). The names reflect the process by which you determine how many inhabitants exist in a particular Type.

If the calculation involves a multiplication then it's a `Product Type`. If it involves addition then it's a `CoProduct Type` or `Sum Type`. In Math, the prefix `co` is added to mean the opposite (Domain, Codomain, Sine, Cosine, Tangent, Cotangent, etc.).

Sum Types (Coproduct)

```
data Bool = True | False
```

In Haskell and Elm, this is how they define their boolean values. We can make our own `Bool` in PureScript since it won't collide with `Boolean`.

Here `Bool` is the Data Type and `True` and `False` are the 2 Data Constructors:

```
mkTrue :: Bool ❶
mkTrue = True ❷

mkFalse :: Bool ❶
mkFalse = False ❷
```

❶ `Bool` is the `Data Type`.

❷ `True` and `False` are the Data Constructors

`Bool` can be either `True` or `False` but NOT both. That's what makes it a ***Sum Type***. Sum Types are also called ***Unions*** because a union in Set Theory is an OR operation. The union of Set `A` and Set `B` contains elements that are contained in `A` OR contained in `B`.

Understanding this should help you see why we separate the different possible Values of `Bool` with `|`, which is commonly used as the OR symbol in many languages, i.e. `||`.

Let's look at another `Sum Type`:

```
data FailureReason
  = InvalidSyntax
  | InvaidInput
  | AlreadyExists
  | NotFound
  | Other String
```

Here we have a Type to model a reason for failures with a catch-all case of `Other`. Notice that `Other` takes a `String` as a Parameter. Types can take Parameters just like Functions.

It turns out that the Data Constructor, `Other`, is a Function. Its ***implied*** Type is:

```
Other :: String -> FailureReason
```

This means that we can use `Other` the same way we use Functions because it is a Function.

All of the other Data Constructors for `FailureReason` are Values. Here are their ***implied*** types:

```
InvalidSyntax :: FailureReason
InvalidInput :: FailureReason
...
```

`FailureReason` is a bit limited though. Can you see why?

Notice that `Other` only takes a `String`. This may be fine for most situations but what if we want to use `FailureReason` elsewhere in our codebase where `String` isn't enough to fully describe the `Other` case?

Let's make a more Flexible version of `FailureReason`:

```
data FailureReason' a ❶
  = InvalidSyntax'
  | InvaidInput'
  | AlreadyExists'
  | NotFound'
  | Other' a
```

❶ We added `'` to the end of all of the names to distinguish them from our old definition.

Now we've introduced the ***Type Variable***, `a`, into the definition. This means that `Other'` can still take a `String` if we want, but now it also can take any other Type:

```
failWithString :: FailureReason' String ❶
failWithString = Other' "Because this Function always fails"

type Error = { code :: Int, reason :: String }

failWithError :: FailureReason' Error ❷
failWithError = Other' { code: -123, reason: "Because of reasons"}
```

❶ We pass `String` to the `FailureReason'` Type which unifies `a` with `String`. This means that `Other'` must take a `String`.

❷ Here `a` unifies with `Error` meaning `Other'` expects an `Error`.

Polymorphic vs Monomorphic

Let's look at our two definitions:

```
data FailureReason ❶
  = InvalidSyntax
  | InvaidInput
  | AlreadyExists
  | NotFound
  | Other String

data FailureReason' a ❷
  = InvalidSyntax'
  | InvaidInput'
  | AlreadyExists'
  | NotFound'
  | Other' a
```

❶ This is ***Monomorphic*** because `FailureReason` takes no Type Parameters (***Mono*** means ***one*** and ***morph*** means ***shape***). Our `Other` Data Constructor is also Monomorphic since it can only take one shape or Type, viz. `String`.

❷ This is ***Polymorphic*** because `FailureReason'` takes a Type Parameter, `a` (***Poly*** means ***many*** and ***morph*** means ***shape***). Our `Other'` Data Constructor is Polymorphic since it can take many shapes or Types, e.g. `String`, `Error`, etc.

Monomorphic Types are always in uppercase, e.g. `String`, whereas, Polymorphic Types are always in lowercase, e.g. `a`.

Product Types

Let's look at a simple Type:

```
data Triplet a b c = Triplet a b c
```

Here we have a 3 Polymorphic Type Parameters, `a`, `b` and `c`.

The Data Constructor, `Triplet` has the implied Type:

```
Triplet :: a -> b -> c -> Triplet
```

Let's create a `Triplet`:

```
type StringStats = Triplet String Int Int

getStats :: String -> StringStats
getStats s = Triplet s (length s) (vowelCount s) ❶ ❷
```

❶ The first slot of `Triplet` contains our `String`. The second slot contains the length and the third the number of vowels.

❷ `vowelCount` is an imagined Function whereas `length` is a Function in the `Data.String.CodeUnits` module.

`StringStats` simultaneously contains 1 `String` and 2 `Integers`. That's what makes it a ***Product Type***. It contains a `String` AND an `Int` AND another `Int`. In sets, this corresponds to `Intersection`. The intersection of Set `A` and Set `B` contains elements that are contained in `A` AND contained in `B`.

The trouble with `Triplet` is that we can mix up the 2 `Ints`. There are ways to fix this with `newtype` which we will look at later, but there's an even better way.

We can change `StringStats` to use a `Record` instead of using the potentially ambiguous `Triplet`:

```
data StringStats = StringStats ❶
  { string :: String ❷
  , length :: Int ❸
  , vowelCount :: Int ❹
  }
```

❶ `StringStats` is no longer a Type Alias, but it's a proper Data Type. This change is NOT a requirement to make a `Record` in PureScript (it is in Haskell). We could've left it a Type Alias.

❷ Holds the same as the first slot of `Triplet`.

❸ Holds the same as the second slot of `Triplet`.

❹ Holds the same as the third slot of `Triplet`.

`StringStats` is not ambiguous like `Triplet` is.

Isomorphic

Comparing our `Triplet` to `StringStats`:

```
data Triplet a b c = Triplet a b c

data StringStats = StringStats
  { string :: String
  , length :: Int
  , vowelCount :: Int
  }
```

`StringStats` and `Triplet` both contain the same information. The big difference is the fact that `Triplet` is more flexible since it can take any types `a`, `b` and `c`, whereas `StringStats` takes very specific types `String`, `Int` and `Int`:

They're almost the same, but let's work to make them closer. First, let's make a specialized version of `Triplet`:

```
data StringTriplet = StringTriplet String Int Int

data StringStats = StringStats
  { string :: String
  , length :: Int
  , vowelCount :: Int
  }
```

Now `StringTriplet` and `StringStats` have exactly the same types. That means we could use either Type interchangeably. There is a formal definition for this:



Two Types, T1 and T2, are ***Isomorphic*** (***Iso*** means ***equal*** and ***morph*** means ***shape***) if a Function can be written from T1 to T2 and from T2 to T1 without any loss of information.

Let's see if our new types are Isomorphic by trying to write lossless conversion Functions between them:

```
from :: StringTriplet -> StringStats
from (StringTriplet string length vowelCount) = ❶
  StringStats ❷
  { string: string
  , length: length
  , vowelCount: vowelCount
  }

to :: StringStats -> StringTriplet
to (StringStats { string, length, vowelCount }) = ❸
  StringTriplet string length vowelCount ❹
```

❶ `from's` Parameter is destructured using ***Pattern Matching*** giving us access to the elements of the Product Type, `StringTriplet` (more on this later).

❷ We construct `StringStats` from the parts we destructured from `StringTriplet`.

❸ `to` also uses Pattern Matching to access fields of the Record in `StringStats` using a slightly different syntax since it's destructing a `Record`.

❹ We construct `StringTriplet` from the parts we destructured from `StringStats`.

Don't worry, we'll take a deeper dive into Pattern Matching soon enough. But the point of this code is to prove that we can write Functions to convert back and forth between `StringTriplet` and `StringStats` with NO information loss.

These Funtions prove that `StringTriplet` and `StringStats` are Isomorphic.

Inhabitants

The ***NO information loss*** part of Isomorphisms is very important. At first glance, you might think `String` and `Int` are Isomorphic since any `Int` can be converted into a `String`. But not any `String` can be converted into an `Int`.

Types are similar to Sets, except where Sets have Elements, Types have Inhabitants. And there are far more Inhabitants of `String` than `Int` even though they both have an Infinite number of Inhabitants.

This gets into the mind-bending concept of varying sizes of Infinity, but we can think of it simply by realizing that every `Int` has a string representation in the `String` Type, but `String` has additional inhabitants that don't have any digits, e.g. `""`, `"·_·"`, `"abc"`, etc.

The number of inhabitants in `String` is greater than the number in `Int`, which means that they cannot be Isomorphic.

Let's imagine a Type that only contains the numbers `42` and `79` and let's call it `TwoNum`. Because this has two inhabitants, it makes it Isomorphic to `Boolean` since it also has two inhabitants, `true` and `false`.

How we map between `TwoNum` and `Boolean` can be totally arbitrary. We could simply just make a lookup

table that's used by our `to` and `from` Functions:

```
TwoNum    Boolean
------    -------
42        false
79        true
```

With a lookup table, we can always map back and forth between any two types as long as they have an equal number of inhabitants.



Any two types with an equal number Inhabitants are Isomorphic.

Let's look at Sum Types (Coproducts) and calculate their inhabitants:

```
data Variant = This | That | TheOther
```

It's easy to see that `Variant` has 3 inhabitants, `This`, `That` and `TheOther`.

Let's try another:

```
data DualResult = First Boolean | Second Variant
```

`DualResult` has 2 Data Constructors, `First` and `Second`, but that doesn't automatically mean that it has 2 inhabitants.

Since both `First` and `Second` have Type Parameters, we must take those into consideration. `First` can take a `Boolean` which has 2 inhabitants and `Second` takes a `Variant` which has 3 inhabitants. So let's enumerate all possible values of `DualResult`:

```
First true
First false
Second This
Second That
Second TheOther
```

Notice each Data Constructor takes a Parameter of Type `Boolean` (`2` inhabitants) OR `Variant` (`3` inhabitants). There's that `OR` again like we saw in Unions.

Taking all of this into consideration, it should be clear that `DualResult` has `2 + 3` inhabitants. This addition operation to calculate inhabitants is why we call it a `Sum Type`.

Let's see if this same logic holds with Product Types:

```
data BooleanVariant = BooleanVariant Boolean Variant
```

Be careful not to confuse the Type, `BooleanVariant` (lefthand side), with the Data Constructor, `BooleanVariant` (righthand side). Here are all of the possible values of the Type `BooleanVariant`:

```
BooleanVariant true This
BooleanVariant true That
BooleanVariant true TheOther
BooleanVariant false This
BooleanVariant false That
BooleanVariant false TheOther
```

Notice each Data Constructor has both a `Boolean` AND a `Variant`.

With a Product Type, we have to consider all combinations of these types, i.e. all combinations of the 2 inhabitants from `Boolean` paired with the 3 inhabitants from `Variant`, or `2 · 3` inhabitants. This multiplication operation is why we call this a `Product Type`.

When the Types are ORed as in `DualResult` we ADD (Sum) and when the types are ANDed as in `BooleanVariant` we MULTIPLY (Product).

But what happens when they're combined:

```
data Combo = Dual DualResult | BoolVar BooleanVariant
```

The `|` tells us that we're adding the inhabitants of each side. So we have `(2 + 3) + (2 · 3)` or `5 + 6` inhabitants because `DualResult` has 5 and `BooleanVariant` has 6.

So far we've been calculating inhabitants for Monomorphic Types. How do we calculate the inhabitants for Polymorphic Types?

```
data Sometimes a = HaveIt a | DoNotHaveIt
```

`Sometimes` represents an `a` sometimes, i.e. we may or may not have an `a`.

The number of inhabitants are `a` for `HaveIt` and `1` for `DoNotHaveIt`. That means that the number of inhabitants for `Sometimes` is `a + 1`.

Here `a` stands for ***the number of inhabitants of a***.

If `a` is `Boolean` as in:

```
type SomeBool = Sometimes Boolean ❶
```

❶ `a` unifies with `Boolean`.

then the number of inhabitants of `SomeBool` is `2 + 1`. We got this by replacing the `a` in `a + 1` with `2` for the number of inhabitants in `Boolean`.

New Types

We've just learned how we can create our own Types to model our problem domain. Next, let's look at how we might model some simple problems.

Let's write a simple Function to format a name:

```
fullName :: String -> String -> String -> String
fulName first middle last = first <> " " <> middle <> " " <> last ❶
```

❶ The operator `<>` appends strings to each other.

Forgetting for now that our Function will have 2 spaces between the first and last name if the middle name is an empty string, can you see anything else wrong with our Function?

HINT: Look at the Types.

Imagine we call our Function like:

```
fullName "Smith" "Jay" "John"
```

We accidentally put the last name first. How did that happen?

Looking at just the Type Signature:

```
fullName :: String -> String -> String -> String
```

we can see that it's not very helpful regarding the order of our Parameters. So, let's make it so we can read this better by creating Type Aliases:

```
type FirstName = String
type MiddleName = String
type LastName = String

fullName :: FirstName -> MiddleName -> LastName -> String
fulName first middle last = first <> " " <> middle <> " " <> last
```

Now, when we look at the Type Signature, we will be able to tell how we should be calling this, greatly reducing our chance for errors:

```
fullName "Smith" "Jay" "John"
```

Well, we did it again! We made the same mistake. The Type Aliases are helpful but only if we the programmer read them. How can we make this better?

Ideally, we'd like our program to fail to compile if we get these in the wrong order. So let's create ***unique*** Types for each Parameter in our Function:

```
data FirstName = FirstName String ❶
data MiddleName = MiddleName String
data LastName = LastName String
data FullName = FullName String

fullName :: FirstName -> MiddleName -> LastName -> FullName
fulName (FirstName first) (MiddleName middle) (LastName last) = ❷
  FullName (first <> " " <> middle <> " " <> last) ❸
```

❶ We wrap all of our `Strings` in unique Types.

❷ We destructure all of our Parameters using Pattern Matching.

❸ The final result is wrapped in `FullName`.

Now let's call our Function with our new Types:

```
-- COMPILER ERROR!
fullName (LastName "Smith") (MiddleName "Jay") (FirstName "John")
```

We called `fullName` with the Parameters in the wrong order again. But this time, we get a compiler error because even though `FirstName` and `LastName` both take `Strings`, they are not the same Type anymore.

So when we accidentally passed `LastName "Smith"` where the Function was expecting a `FirstName`, the compiler caught the `Type Mismatch` error, protecting us from ourselves.

This technique just ***wraps*** a Type inside of another Type. In our case, we wrapped our String in another Type which made it unique. We essentially made a new Type for each String in our original Function.

PureScript has a special keyword for this approach called `newtype`:

```
newtype FirstName = FirstName String ❶
newtype MiddleName = MiddleName String
newtype LastName = LastName String
newtype FullName = FullName String

fullName :: FirstName -> MiddleName -> LastName -> FullName
fulName (FirstName first) (MiddleName middle) (LastName last) =
  FullName (first <> " " <> middle <> " " <> last)
```

❶ `data` has been replaced with `newtype`

The `newtype` keyword tells the compiler that we're just making a new Type for another Type, e.g. `FirstName` is a new Type for `String`. That means that the compiler can do some optimizations if it knows that it's just a simple wrapper.

But it can only do these optimizations if certain restrictions are imposed:

- `newtypes` must only have 1 Data Constructor
- The Data Constructor can only take 1 Parameter

The Type `FullName` has a single Data Constructor, also called `FullName` that takes a single Type Parameter, `String` as does `MiddleName`, `LastName` and `FullName`. This is why we could replace `data` with `newtype`.

There are additional features that makes working with `newtypes` convenient. We will leverage these features after we learn about ***Typeclasses***.

3.1.4. Common Library Types

PureScript implements many commonly used Types in it's libraries. We're going to explore a few of them here.

`Void`

In Set Theory, there's the concept of an ***Empty Set*** that has no elements.

In Type Theory, there's the concept of ***Void Type***, which is a Type with Zero Inhabitants. In PureScript (and Haskell) this Type is called `Void`.

Of all the Types we'll explore, `Void` is probably the least used.

You might be wondering how you would define a Type that has no inhabitants. If we try, we might write:

```
data NoInhabitants = NoInhabitants
```

Unfortunately, this has 1 inhabitant, viz. `NoInhabitants`.

There is one possible solution. We can put this Type definition in a Module and then only export out the Type and NOT the Data Constructor. This way no one can construct a `Void`.

But there's another, much more clever and simpler solution:

```
data Void = Void Void
```

At first glance, this may be confusing, so let's break it down from Left to Right:

- The first `Void` is the Data Type.
- The second `Void` is the Data Constructor.
- The third `Void` is the Type Parameter to the `Void` Data Constructor.

So to create a `Void` we have to use the `Void` Data Constructor and pass it a `Void`:

```
v :: Void
v = Void (???)
```

The `???` needs to be of type `Void`, which we can create using the `Void` Data Constructor:

```
v :: Void
v = Void (Void (???))
```

But our next Data Constructor also needs a `Void`, which we can create using the `Void` Data Constructor and so on:

```
v :: Void
v = Void (Void (Void (Void (Void(Void(Void(...)))))))
```

It is impossible to construct a `Void` since it's only Data Constructor requires a `Void`. We have a chicken and egg problem.

The technical term for this is a ***Recursive Definition*** because the definition refers to itself.

Except, in this particular case, it's an ***Infinitely Recursive Definition*** because there is no base-case to terminate the recursion.

Let's look at a made-up, Infinitely Recursive Definition:

```
data NeverEnding = NeverEnding NeverEnding
```

Now, we add a base-case to turn this into just a Recursive Definition:

```
data NeverEnding = NeverEnding NeverEnding | TheEnd ❶
```

❶ `TheEnd` is our base-case since it ***terminates*** the Recursive Definition.

This definition is no longer Infinitely Recursive thanks to the newly add Data Constructor, `TheEnd`. And because of that, we can actually write down something of type `NeverEnding`:

```
actuallyEnds :: NeverEnding
actuallyEnds = NeverEnding (NeverEnding (NeverEnding TheEnd))
```

The first and second `NeverEnding` Data Constructors needed something of Type `NeverEnding` and got it by using the `NeverEnding` Data Constructor.

But the third and final `NeverEnding`, which also needs something of Type `NeverEnding`, got `TheEnd` instead, which terminates our construction.

`Unit`

In Set Theory, there's the concept of a ***Unit Set*** that has exactly 1 element.

In Type Theory, there's the concept of ***Unit Type***, i.e. a Type with only One Inhabitant and in PureScript this Type is called `Unit`.

The only inhabitant of `Unit` is called `unit`.

`Unit` appears often in code with Effects. Our `main` Function in our `Hello World` example has the following signature:

```
main :: Effect Unit
```

In fact, the entry point to all PureScript programs will always have this Type Signature.

`Maybe`

Anyone who has coded in C, Java, Javascript and a host of other languages where values can be NULL knows all too well the pain of dealing with these cases. It seems that NULL values have haunted programmers since the beginning of time. Well, at least since the beginning of NULL values.

It turns out that PureScript solves this problem by having NO NULLs. You may wonder how you handle optional values without NULLs. Well, it's done with the `Maybe` Type.

Here's its definition and an example of how it could be used:

```
data Maybe a = Just a | Nothing ❶

data Person = Person ❷
  { name :: String
  , birthdate :: Date
  , deathdate :: Maybe Date ❸
  }
```

❶ Maybe takes a single, Polymorphic Parameter, `a`, which can be ANY Type.

❷ Person is a `Record` with 3 fields.

❸ `deathdate` is a `Maybe Date` since the `Person` may still be living. In this case, `Maybe's` Type Parameter, `a`, unifies with `Date`.

To set `deathdate` for an existing `person` we'd do the following:

```
import Data.Date (canonicalDate)

person :: Person
person = Person
  { name: "Joe Mama"
  , birthdate: canonicalDate 1962 10 2 ❶
  , deathdate: Nothing ❷
  }

-- somewhere else in our code
deadPerson = person { deathdate = Just today } ❸ ❹
```

❶ We set `birthdate` by calling `canonicalDate`, which takes a `year`, `month` and `day` and returns a `Date`.

❷ When this record is constructed, there is no `deathdate`, so we construct a `Maybe Date` by using the constructor `Nothing`.

❸ Later in our code, we set the `deathdate` by using the Data Constructor `Just`. But we cannot simply use `Just` by itself. `Just` requires a single Parameter of Type `a`, which, in this case, is `Date`. Therefore, we must give it a `Date` and we do so by give the Data Constructor a single Parameter, `today`, which we'll pretend is a `Date`.

❹ We're using the Update `Record` syntax, hence the use of `=` and NOT `:`.

This is all fine and good, but you may still be wondering how this is better than NULL. This can best be

answered by looking at an example.

First in Javascript:

```
const x = 10;
const y; ❶
const add = (x, y) => x + y -- z = NaN ❷
```

❶ We did not initialize `y`.

❷ Here we try to use `y` in a computation and wind up with `NaN`. Now any computation that uses `z` will be `NaN`.

Now in PureScript:

```
add :: Int -> Int -> Int
add x y = x + y

useAdd :: Int
useAdd =
  let x :: Int
      x = 10

      y :: Maybe Int
      y = Nothing ❶
  in
  add x y -- COMPILER ERROR!! ❷
```

❶ We have no value for `y` and it's EXPLICITLY defined to be a `Maybe Int`.

❷ The compiler complains because we tried to add an `Int` with a `Maybe Int`.

The compiler save us from using an incompatible Type. So how do we fix this?

We could try to make the Types compatible:

```
add :: Int -> Int -> Int
add x y = x + y

useAdd :: Int
useAdd =
  let x :: Int
      x = 10

      y :: Int ❶
      y = Nothing  -- COMPILER ERROR!!
  in
  add x y ❷
```

❶ If we try to make `y` an `Int` but set it to `Nothing`, the compiler will complain since our types don't match (`Int ≠ Maybe Int`)

❷ The compiler doesn't complain here because we defined `y` to be of Type `Int` which is fine for adding to another `Int`.

Once again the Type system saves us from assigning a `Maybe Int` to an `Int`. But how do we fix this now?

We could just initialize `y` to some Value:

```
add :: Int -> Int -> Int
add x y = x + y

useAdd :: Int
useAdd =
  let x :: Int
      x = 10

      y :: Int
      y = 42
  in
  add x y
```

Now everything works. But this example is too simplistic. What would happen if `useAdd` got passed `y`:

```
add :: Int -> Int -> Int
add x y = x + y

useAdd :: Maybe Int -> Int (1)
useAdd y = (1)
  let x :: Int
      x = 10
  in
  add x y  -- COMPILER ERROR!! (2)
```

(1) We now get passed `y`, so our Type Signature and Parameter list reflect that.

(2) And we get a compiler error just like before since we cannot add `Int` to a `Maybe Int`.

We haven't fixed anything just yet, but now this problem is more realistic. So now let's fix it by making `useAdd` return a `Maybe Int` to reflect the fact that we MAY fail:

```
add :: Int -> Int -> Int
add x y = x + y

useAdd :: Maybe Int -> Maybe Int (1)
useAdd y' = (2)
  let x :: Int
      x = 10
  in
  case y' of (3)
    Just y -> Just (add x y) (4)
    Nothing -> Nothing (5)
```

(1) Our Type Signature changed to return a `Maybe Int`.

(2) Our Parameter name changed to `y'` so it won't conflict with `y` later.

(3) We are using a `case` expression in PureScript (more on this later).

(4) We are Pattern Matching with the `Just y` where `y` will get bound to the Value ***inside*** the `Maybe`, which is an `Int` and can be safely used in `add x y`.

(5) If we don't get a `y`, i.e. it's `Nothing`, then all we can do is return `Nothing`.

Notice how the `Maybe` Type forced us to handle the fact that `y'` may not exist. The compiler hounded us to get our types in order.

At first, you may curse the compiler for this, I know I did. But over time, you'll begin to love the fact that all possible errors have been "handled". It's still possible to handle this case badly, but at least you are forced to think about each and ever one of them.

In the Javascript case, our Value became a `NaN` which is Isomorphic to `Nothing`. The difference being that the PureScript compiler enforces the use of `Maybe`. And if we use best practices, `Maybe` will show up in our Type Signatures, which will inform us and others to the fact that our Function may fail.

So the Semantics of `Maybe` is something that can fail. In our example, `useAdd` cannot use `add` in the absence of one of the `Ints`.

There are many times were `Maybe` comes in handy:

- Search `String` for a `Char` or another `String`.
- Get a Value from `Map` (a Key/Value store) using a key.
- Get a Value from an `Array` at a particular index.

Each of these Functions have a single ***obvious*** failure, `Char` or `String` not found, key not found or index out of bounds.

But what if that's not the case or our Function can fail for multiple reasons. How do we want return an explicit failure reason?

Either

The `Either` Type is another failure Type, like `Maybe`, except it has the added benefit of having a reason for the failure:

```
data Either a b = Left a | Right b
```

Here Type `a` is used by the `Left` Data Constructor and Type `b` is used by the `Right` Data Constructor. To get a better sense of this, let's look at an example:

```
data QueryError ❶
  = DatabaseConnectionError String ❷
  | InvalidQuery
  | NotFound

query :: Query -> Either QueryError Int ❸
```

❶ `QueryError` models the different errors we can encounter.

❷ `DatabaseConnectionError` takes a `String` which contains more information about the connection failure.

❸ `query` takes a `Query` (not defined here) and either returns a `QueryError` or an `Int`.

So how would we create the different errors:

```
Left (DatabaseConnectionError "The connection was refused by the server")
Left InvalidQuery
Left NotFound
```

With `Either`, the left side is ALWAYS the error type. This isn't just a convention. Later, we'll see why this is the case, but for now, just remember `Left` is the error and `Right` is success. I always think of `Right` as being the right result and anything else is wrong or an error.

Here's how we represent a successful result for `query`:

```
Right 42
```

Let's revisit the definition of `Either`:

```
data Either a b = Left a | Right b
```

`Either` has 2 Type Parameters, `a` and `b`. What this means is that the Type on the left side CAN be different than the Type on the right.

But it doesn't have to be:

```
Either Int Int
```

This would mean that it's going to return an `Int` or an `Int` which is perfectly valid. Just because `a` and `b` are different names only says that they ***can*** be different Types, but they're completely independent from each other. Therefore `a` can be an `Int` and so can `b`.

But how do we tell the difference between an `Int` that's an error and an `Int` that's a success?

We use the `case` expression and Pattern Matching:

```
case result of ❶
  Left errorCode -> "The error code is: " <> show errorCode ❷ ❹
  Right count -> "The number of rows returned is: " <> show count ❸ ❹
```

❶ `result` is NOT an `Int`, it's an `Either Int Int`.

❷ We match on `Left` to extract `errorCode`.

❸ We match on `Right` to extract the `count`.

❹ We have to use `show` with `errorCode` and `count` to convert them to `Strings` so that they can be appended with the rest of the message.

You can think of `Left` and `Right` as ***Tagging*** the `Int` so that we can tell whether it represents an error or a success. This is why `Sum Types` are also known as `Tagged Unions`.

Maybe vs Either

`Maybe` and `Either` can both represent error results, but only `Either` gives us a reason. `Maybe`, on the other hand, can represent an optional Value.

Just for fun, let's look at the inhabitants of both:

```
data Maybe a = Just a | Nothing

data Either a b = Left a | Right b
```

Remember that `Maybe` is a `Sum Type`, so we need to add the inhabitants of all of the Data Constructors. The inhabitants of `Maybe` is `a` for `Just a` and `1` for `Nothing`, therefore the number of inhabitants is `a + 1`.

And for Either, which is also a `Sum Type`, we have `a` for `Left` and `b` for `Right` or `a + b` inhabitants.

```
Maybe:    a + 1
Either:   a + b
```

That was a mildly interesting exercise, but there's actually something more interesting if we try to make `Maybe` and `Either` have the same inhabitants, which would make them `Isomorphic`.

There's 2 ways we can do that.

First, we choose a Type for `b` such that it has only 1 inhabitant:

```
Maybe:    a + 1
Either:   a + 1     (b = 1)
```

One thing to note here is that an `a` in the `Maybe` Type is independent of an `a` in the `Either` Type. So, an easy way to make them have the same number of inhabitants, is to simply choose the same Type for both `a's`. We also could've picked 2 different Types with equal Inhabitants.

Let's choose `Boolean` for both `a's` and `Unit` for `b` (we need `b` to have 1 inhabitant):

```
Maybe Boolean ≅ Either Boolean Unit ❶
```

❶ The symbol `≅` stands for `is Isomorphic to`.

Notic that our `Either` has an error Value but no success Value. While they're technically Isomorphic,

`Either Boolean Unit` doesn't make sense with the Semantics of `Maybe Boolean`. That's because the `Boolean` in `Maybe` is the success Type, whereas the `Boolean` in `Either` is the error Type.

The second way to make them Isomorphic is to swap the `Either` types:

```
Maybe Boolean ≅ Either Unit Boolean
```

Now our `Either` has a success Value but no error Value, which is exactly like `Maybe`. It too only has a success Value. Semantically, this Isomorphism makes sense. `Boolean` in both types is the success Type.

So we have 3 inhabitants for each of the following:

```
Maybe Boolean ≅ Either Boolean Unit ≅ Either Unit Boolean
```

Let's list them:

```
Maybe Boolean
-------------
Just true
Just false
Nothing

Either Boolean Unit
-------------------
Left true
Left false
Right unit

Either Unit Boolean
-------------------
Left unit
Right true
Right false
```

While this was slightly more interesting, you may be wondering what this has to do with using these Types. In turns out that there are times we want to convert between `Maybe` and `Either` and when they're Isomorphic we can easily do this:

```
to :: Maybe Boolean -> Either Unit Boolean
to (Just b)  = Right b
to Nothing   = Left unit

from :: Either Unit Boolean -> Maybe Boolean
from (Right b) = Just b
from (Left _)  = Nothing ❶
```

❶ We can use _ to represent a Variable that we don't care about naming. This is common pattern when working with `unit`.

But, let's be real, we're never going to have an `Either` where one of the Type Parameters is of Type `Unit`. We're just going to use a `Maybe`.

So how do we convert between them when they're not Isomorphic?

Looking at `to` and `from` above gives us some clues.

When we went from `Maybe` to `Either`, we had to make up data for the `Nothing` case. We did that by using `unit`. When we went from `Either` to `Maybe`, we threw out data for the `Nothing` case. We threw out the `unit`.

If we want to convert between ***any*** `Maybe` and `Either`, we'll need to do something similar:

```
hush :: Either a b -> Maybe b
```

Notice there is no `a` in the `Maybe` result. That's should tell us that `hush` "throws away" the error Type on the left side of the `Either`.

Let's look at the rest of the Function:

```
hush :: ∀ a b. Either a b -> Maybe b ❶
hush (Left _)  = Nothing ❷ ❹
hush (Right x) = Just x ❸ ❹
```

❶ When defining Polymorphic Type Parameters for Functions, you must explicitly define them using the keyword `forall` or you can optionally use the symbol, `∀`. I use the symbol because it's easier to read.

❷ _ is used since we aren't going to use the `a` from `Left`.

❸ We destructure the `Right` to get to the `x`, which is of Type `b`, and then we construct a `Maybe` using `Just`.

❹ In PureScript, we can write differnt versions of Functions. The compiler matches the Parameters passed in to determine which version to call. Unlike languages that support overloading, they MUST have the same Type Signatures.

`hush` is in the library `package.either` in module `Data.Either`.

To go the other direction, i.e. from a `Maybe` to an `Either`, we need to provide the Value for the left side of the `Either` since `Maybe` only has `Nothing` for its error case:

```
note :: ∀ a b. a -> Maybe b -> Either a b ❶
note err Nothing = Left err ❷
note _ (Just x) = Right x ❸ ❹
```

❶ The first Parameter is of Type `a`, which is the same Type as the `a` side of the `Either`.

❷ In this case, we have `Nothing` and so we must use the provided error Value, `err`, to construct a `Left`.

❸ We don't need the error Value here since this case is the success case, i.e. the right side of `Either`. So we just use `_` as our "don't care" Variable name.

❹ We destruct the `Just` to get at the `x` and then immediately wrap it in `Right`.

`note` is also a library Function that can be found in the same module as `hush`, viz. `Data.Either`.

Tuple

A `Tuple` is a `Product Type` that contains 2 things. Here's its definition:

```
data Tuple a b = Tuple a b
```

In Math, a Tuple is written as `(x, y)`. Haskell and Elm use the same syntax for Tuples that's used in Math. In other programming languages, this is often called a pair.

A `Tuple` is useful when you need to create a pair of things that are related is some way.

In Math, we used tuples for `x-y coordinates` on a 2D graph, e.g. `(5, -1)`. To full describe a point in a plane, we need both the `x` and `y` coordinates. It makes sense to represent them as a tuple:

```
type Point = Tuple Int Int
```

Another use case for `Tuple` is when you want to return 2 values back from a Function:

```
splitPosAndNeg :: Array Int -> Tuple (Array Int) (Array Int)
```

`splitPosAndNeg` will take an `Array` of `Int` and return a `Tuple` where the first position of the `Tuple` contains an `Array` of Negative Numbers and the second contains an `Array` of Positive.

Here's how we might use it:

```
let Tuple pos neg = splitPosAndNeg [1, -2, 3, -4, 5, -6]
```

We are using destructuing here to assign `pos` to the first of the `Tuple` and `neg` to the second.

We can use library Functions to do the same:

```
let split = splitPosAndNeg [1, -2, 3, -4, 5, -6]
    pos = fst split ❶
    neg = snd split ❷
```

❶ `fst` gets the first Value of a `Tuple`.

❷ `snd` gets the second Value of a `Tuple`.

`fst` and `snd` are in the package `purescript-tuple` in module `Data.Tuple`.

Either vs Tuple

Earlier, we made Monomorphic versions of `Maybe` and `Either` such that they were Isomorphic.

In other words, we came up with concrete types for the Polymorphic Type Parameter `a` for `Maybe` and the Polymorphic Type Parameters `a` and `b` for Either to make `Maybe` and `Either` Isomorphic.

Can we do the same with `Tuple` and `Either`?

The real question that we're asking is can we take make a `Product Type` such as `Tuple` Isomorphic to a `Sum Type` such as `Either`. Remember that a `Product Type` is an ***AND*** and a `Sum Type` is an ***OR***.

`Tuple a b` means that this `Tuple` has an `a` ***AND*** a `b`, whereas `Either a b` means that this `Either` has an `a` ***OR*** a `b`.

To answer this, let's first look at their inhabitants:

```
Tuple:  = a * b
Either: = a + b
```

Remember that `Tuple` and `Either` are Isomorphic when the number of inhabitants are equal, meaning that the following must be true:

```
a * b = a + b
```

There's only 2 times that the addition of 2 numbers and multiplication of those same 2 numbers are equal:

```
0 * 0 = 0 + 0     (a, b = 0)
2 * 2 = 2 + 2     (a, b = 2)
```

Let's write the corresponding Types:

```
Tuple Void Void ≅ Either Void Void
Tuple Boolean Boolean ≅ Either Boolean Boolean
```

Although we've made a `Product Type` and a `Sum Type` Isomorphic, with these 2 cases, the Types are pretty useless as far as Types go.

Because of this, conversions between `Product Types` and `Sum Types` are rarely done, if ever.

Case in point, there are no library Functions to convert from `Tuple` to `Either` or vice versa.

List

`Array` implementations are backed up by Javascript Arrays for efficiency and you'll most likely use `Arrays` more than `Lists`, but there are things you cannot do with Arrays as we'll see in Pattern Matching.

And learning how `Lists` work in PureScript will help you to understand how to work with `Lists` in Haskell where they are used much more often than their array counterparts.

We will also write many of the `List` library Functions as exercises that will get you used to programming functionally and recursively.

Here is the Recursive Definition of a `List` and its corresponding operator:

```
data List a = Cons a (List a) | Nil ❶

infixr 6 Cons as : ❷
```

❶ A `List` of `a's` is either the empty list, `Nil` or an `a` that is `Cons-ed` to the head of another `List` (more on this in a minute).

❷ The `:` is an infix operator for the `Cons` Data Constructor. It has precedence 6 and is Right-Associative, hence the `r` at the end of `infixr` (more on this in a minute).

The `Cons` operation puts a single Value onto the head of a `List`.

We can construct a `List` with the `Cons` operator:

```
nums :: List Int
nums = 1 : 2 : 3 : Nil
```

Remember the operator, `:`, is Right-Associative, which means it associates to the right making the previous example equivalent to the following:

```
nums :: List Int
nums = (1 : (2 : (3 : Nil))) ❶
```

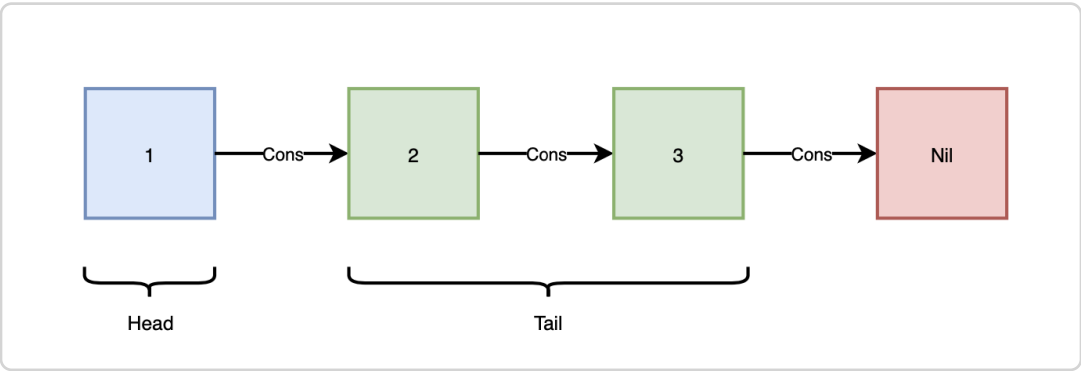
❶ The redundant Parentheses show the Right-Associativity of the operator. Notice how the Parentheses collect on the RIGHT. These Parentheses show that the order of operations move from right to left. First `3` is `Cons`-ed with `Nil`. Then `2` is `Cons`-ed with that resulting `List` and finally, `1` is `Cons`-ed.

We could have construct the list with just the `Cons` Data Constructor instead of the operator:

```
nums :: List Int
nums = Cons 1 (Cons 2 (Cons 3 Nil)) ❶
```

❶ This syntax helps us see the Recursive Definition in action but is more difficult to read and reason about. That's where the `:` operator is helpful.

`Lists` are effectively ***Linked Lists***. Here is a pictoral view of our list:





The `head` of the list is the first item in the list. The `tail` is everything except the `head`.

In the module `Data.List`, there are Functions for getting the `head` and `tail` of a `List`:

```
head :: ∀ a. List a -> Maybe a
head Nil     = Nothing ❶
head (x : _) = Just x ❷

tail :: ∀ a. List a -> Maybe (List a)
tail Nil       = Nothing ❸
tail (_ : xs) = xs ❹
```

❶ If the `List` is empty, then we cannot get the `head` and therefore return `Nothing`.

❷ Here we Pattern Match against the `Cons` operator to get the `head` of the `List`. We don't care about naming the `tail` and so use `_`.

❸ If the `List` is empty, then there is no tail.

❹ We Pattern Match against the `Cons` operator to extract the `tail` and return it. We don't care about the `head` and call it `_`.

We'll see how this Pattern Matching works next.



`Cons` has its origins in Lisp, a Programming Language from the 1950s. Lisp stands for **List Processor**. In Lisp, `cons` is the ***Constructor*** for lists.

3.2. Pattern Matching

In our study of Types, we glossed over the idea of ***Pattern Matching*** in PureScript. Let's take a deeper dive.



Pattern Matching matches values based on both their Shape and their Value, e.g. `Just 10` and `Just 5` have the same Shape but different Values, whereas `Nothing` and `Just "xyz"` have different Shapes.

3.2.1. Case Expression

A simple example:

```
import Data.Maybe (Maybe(..)) ❶

isNothing :: ∀ a. Maybe a -> Boolean ❷
isNothing m = case m of ❸
  Nothing -> true ❹
  _       -> False ❺
```

❶ `Maybe` is in the `Data.Maybe` module. The `(..)` after the `Maybe` say that in addition to importing the Type `Maybe`, we want to include ALL of its Data Constructors.

❷ Don't forget the `∀ a`. Notice that `.` terminates the Type Variable definitions.

❸ We use `case` on `m` to do Pattern Matching on it.

❹ Our first case is to check for `Nothing`. The expression to the right of `->` is the Value of the Function if the pattern matches.

❺ The `_` is normally a dont-care variable and it has a similar semantic here. This is a catchall for any pattern, sort of a dont-care pattern.

There is another, more idomatic, way to write this Function, which is to write it by defining multiple versions of the same Function but with different patterns for the input Parameters:

```
import Data.Maybe (Maybe(..))

isNothing :: ∀ a. Maybe a -> Boolean
isNothing Nothing = true ❶
isNothing _       = false ❷
```

❶ This version of `isNothing` will be called if the input Parameter matches `Nothing`.

❷ This version of `isNothing` will be called if any previous versions did not match.

Our patterns must cover all possible values of each input Parameter. This requirement is also imposed on the `case` expression.

The compiler will let you know if your patterns are not ***exhaustive***, i.e. you missed a possible case.

Pattern Matching starts with the first definition and proceeds to subsequent ones. Same for `case` pattern matching. Therefore, all catchall cases must be last.

The compiler will let you know if a case is ***unreachable***, e.g. you didn't put a catchall last.

3.2.2. String Patterns

Let's look at another example:

```
toString :: Boolean -> String
toString true  = "true"
toString _     = "false" ❶
```

❶ This catchall case is equivalent to `toString false`. We just chose to use `_` for brevity.

And the inverse Function:

```
fromString :: String -> Boolean
fromString "true" = true
fromString _      = false (1)
```

(1) This catchall case is NOT the same as the catchall in `toString`. What `_` represents is any string that's NOT `"true"`. So both `"TRUE"` and `"false"` will match, since pattern matching of `Strings` is case-sensitive.

3.2.3. Array Patterns

`Arrays` can only be Patterned Matched against ***fixed-length*** `Arrays`. So we can write an `isEmpty` Function using Pattern Matching:

```
isEmpty :: ∀ a. Array a -> Boolean
isEmpty [] = true (1)
isEmpty _ = false
```

(1) An empty `Array` literal is written as `[]`.

We can also write:

```
multiplyTwo :: Array Int -> Maybe Int
multiplyTwo [x, y] = Just (x * y)
multiplyTwo _      = Nothing
```

Pattern Matching with `Arrays` is far less flexible than with `Lists`.

3.2.4. List Patterns

When we first looked at `Lists`, we saw two helper Functions, `head` and `tail`:

```
head :: ∀ a. List a -> Maybe a
head Nil     = Nothing
head (x : _) = Just x (1)

tail :: ∀ a. List a -> Maybe (List a)
tail Nil      = Nothing
tail (_ : xs) = Just xs (1)
```

(1) The `Cons` operator, `:`, can be used for Pattern Matching.

Idiomatic PureScript (and Haskell) uses a single character Variable, e.g. `x`, for the `head` of a `List` and that

same variable with an appended `s`, i.e. `xs`, to represent the `tail`.

You can think of the head as a single `x` and the tail as a plural of `x` or `xs`.

Let's look at how we can use `Cons` with Pattern Matching:

```
head :: ∀ a. List a -> Maybe a
head Nil        = Nothing
head (Cons x _) = Just x ❶

tail :: ∀ a. List a -> Maybe (List a)
tail Nil         = Nothing
tail (Cons _ xs) = Just xs ❷
```

❶ We match just like we would any other Data Constructor with 2 Parameters. Here the first Parameter, `x`, is the `head`.

❷ Here the second Parameter of `Cons`, `xs`, is the `tail`.

Take a minute and compare this with `head` and `tail` that use the operator. Personally, I like the operator version better. It turns out that you will use the operator to Pattern Match far more often than `Cons`.

3.2.5. Array vs List

Pattern Matching an `Array`, requires you to know its exact size. You're also forced to deal with all the parts of the `Array` all at once:

```
addThree :: Array Int -> Maybe Int
addThree [x, y, z] = Just (x + y + z)
addThree _      = Nothing
```

With `Lists`, we can write Functions that deal with any length `List` and when we do, we can work with just the `head` and then recurse to continue with the `tail` of the list:

```
sum :: List Int -> Int
sum Nil      = 0 ❶
sum (x : xs) = x + sum xs ❷
```

❶ This Pattern Matches with an empty list, which has a zero length.

❷ We Pattern Match to extract both the `head` and `tail`. Then we add the `head`, `x`, to the sum of the `tail`, `sum xs`.

This Function is recursive, but don't worry if you're not really good with Recursion. We're going to write many Recursive Functions in the following chapters. It will become much easier.

The main point here is that with `Lists`, we don't have know the size of the `List` and we don't have to deal with all of the elements of a `List` at once like we do with `Arrays`.

The limits on Pattern Matching with `Arrays` means that it's impossible to write the `sum` Function for `Array` using Pattern Matching:

```
sum :: Array Int -> Int
sum ???? -- Cannot write a Pattern here
```

We cannot match an `Array` of any size. There are many other solutions to this problem, one of which is using a `List` instead or possibly converting our `Array` to a `List` and using `List` Pattern Matching.

Later, when we learn about ***Folds***, we'll see the best solution for summing `Arrays`.

3.2.6. Record Patterns

Let's start with the following records:

```
type Address =
  { street :: String
  , city :: String
  , state :: String
  , zip :: Int
  }

type Employee =
  { name :: String
  , jobTitle :: String
  , yearsAtCompany :: Int
  , address :: Address
  }

type Company =
  { name :: String
  , yearsInBusiness :: Int
  , address :: Address
  }
```

Now let's look at a bunch of different ways to write a Function to check to see if the `Employee` is the `CEO`. First, the most naïve:

```
isCEO :: Employee -> Boolean
isCEO employee = if employee.jobTitle == "CEO" then true else false ❶
```

❶ Accessing a field of a `Employee` `Record` is done with `employee.jobTitle`.

The syntax for accessing `Records` in PureScript uses the same dot-notation that you see in many other languages, e.g. Javascript. But unlike Javascript, you cannot create an accessor at runtime.

Let's continue by removing the `if` expressions since it's redundant:

```
isCEO :: Employee -> Boolean
isCEO employee = employee.jobTitle == "CEO"
```

Now let's write this using Pattern Matching:

```
isCEO :: Employee -> Boolean
isCEO { jobTitle } = jobTitle == "CEO" ❶
```

❶ `{ jobTitle }` pattern matches the `jobTitle` field and binds it to a local variable with the same name.

When we use `{ jobTitle }`, this is called a ***Record Pun***.

Since we only care about the `jobTitle` in the `Employee` record, we only specified it. We could have specified other fields from that record:

```
isCEOAndGates :: Employee -> Boolean
isCEOAndGates { name, jobTitle } = jobTitle == "CEO" && name == "Bill Gates"
```

Let's write a Function to check the State of a `Company` for California:

```
isCalifornia :: Company -> Boolean
isCalifornia { address: { state } } = state == "CA" ❶
```

❶ This is how you Pattern Match nested records. `address` is of Type `Address`, which is a `Record`. Don't miss the `:` after `address`. That's easy to forget.

What would happen if we wanted Functions to check both the company and the employee's `state`:

```
isCompanyCalifornia :: Company -> Boolean
isCompanyCalifornia { address: { state } } = state == "CA"

isEmployeeCalifornia :: Employee -> Boolean
isEmployeeCalifornia { address: { state } } = state == "CA"
```

These two Functions are nearly the same. Imagine we have other Records with `Addresses` that we'd like to check their states. We'd have to repeat this boilerplate code over and over again.

There is a more general way to write these two Functions:

```
isCalifornia :: ∀ r. { address :: Address | r } -> Boolean ❶ ❷
isCalifornia { address: { state } } = state == "CA"
```

❶ We have to remember to define `∀ r`. Don't forget the `.` after the Variable definition.

❷ The Type `{ address :: Address | r }` says that we have a Record with AT LEAST a field named `address` of Type `Address`. The `| r` says that there MAY be other fields, but we don't care what they are.

What's great about this definition is that it'll match BOTH `Company` and `Employee` even though their other fields don't match. Those other fields are represented by `r` and since this definition is for all `r's`, i.e. `∀ r`, it'll match any set of fields.

The technical term for `r` is a ***Row Type***. More on this later.

There's one more important thing we can do when Pattern Matching Records and that's rename the field Parameter:

```
isCalifornia :: ∀ r. { address :: Address | r } -> Boolean
isCalifornia { address: { state: s } } = s == "CA"
```

Renaming is useful when a field name conflicts with another variable in scope, e.g. a Function name in the same module.

In our example, we can imagine that we renamed the field variable because there's a Function called `state` in the same module as our `isCalifornia` Function.

We can also Pattern Match to specific values of fields in Records:

```
isCalifornia :: ∀ r. { address :: Address | r } -> Boolean
isCalifornia { address: { state: "CA" } } = true ❶
isCalifornia _  = false ❷ ❸
```

❶ Pattern Match with `state` equals `"CA"`.

❷ With this approach, we're forced to provide other versions of the same Function. In this particular case, this approach is more verbose.

❸ Our catchall case is always `false`.

3.3. Logical Control

There are only a few ways to do logical decision making in PureScript:

- `if-then-else` expression
- `case` expression
- Pattern Matching
- Guards

3.3.1. If-Then-Else Expression

Notice, how I used the word ***expression*** when describing `if-then-else`. That's because it's NOT a statement as in many languages. An `if` must always have and `else` part. To understand why, let's look at a simple example:

```
keepPositive :: Int -> Int
keepPositive x = if x < 0 then 0 else x ❶
```

❶ We're forcing `x` to be `0` if it's negative otherwise we leave `x` alone.

If we didn't have the `else` portion then what would `keepPositive x` evaluate to when `x` is `-1`?

In Javascript, there is a similar `if` expression that requires the else part:

```
const keepPositive = x => x < 0 ? 0 : x
```

Javascript also has an `if` statement, but in PureScript everything is an expression and there is no way to evaluate something in one case and nothing in the other. We are always evaluating an expression or calling a Function and then doing something with that result.

If you have an `if` statement with no `else` then you're typically doing one of a few things, performing a calculation or calling a Function and storing it's results into a variable, or you're calling a Function for it's side-effects, etc.

First, there are no side-effects in Pure Functional Programming so that case is out for us.

Second, calling a Function for it's return Value or doing a calculation for it's results is the same thing. And

we can do that too, but we cannot OPTIONALLY assign that result to a variable because that assumes one of two things. Either the variable already has a Value which means we'd be mutating it or it has no Value and therefore is NULL.

Both of these scenarios are forbidden in PureScript. This is why we only have an `if` expression.

3.3.2. Case Expression

We can do logical branching using a `case` expression but it's limited to Pattern Matches:

```
data ContactMethod
  = Phone
  | Email
  | Fax

keepModern :: ContactMethod -> ContactMethod
keepModern preferredContactMethod =
  case preferredContactMethod of
    Phone -> Phone
    Email -> Email
    Fax -> Email ❶
```

❶ If the preferred contact method is `Fax` we use `Email` instead.

Technically, we could rewrite `keepPositive` using `case`:

```
keepPositive :: Int -> Int
keepPositive x = case x < 0 of
  true -> 0
  false -> x
```

While this technically is valid, this version is clunky compared to the `if-then-else` one.

3.3.3. Pattern Matching

Pattern Matching is very similar to using `case`:

```
data ContactMethod
  = Phone
  | Email
  | Fax

keepModern :: ContactMethod -> ContactMethod ❶
keepModern Phone = Phone
keepModern Email = Email
keepModern Fax = Email
```

❶ This version is slightly easier to read than the `case` expression version.

There is no way to write `keepPositive` with ONLY Pattern Matching. We need a way to have an `if-like` operation, which Pattern Matching doesn't allow.

If only there was another way...

3.3.4. Guards

Guards are a way to specify `if` logic in a concise and readable manner. The syntax is a bit strange when first encountered:

```
keepPositive :: Int -> Int
keepPositive x ❶
  | x < 0     = 0 ❷ ❸
  | otherwise = x ❷ ❹
```

❶ Notice NO equal sign when using Guards. This fact is really easy to forget.

❷ The vertical bar, `|`, is the Guard. What follows is a Boolean xpression, an equal sign and the Value of the expression if the Boolean expression is `true`.

❸ This line says if `x < 0` is `true` then return `0`.

❹ `otherwise` is defined to equal `true` and only exists to make reading Guards nicer. It is the catchall case where we simply return `x`.

We can also use Guards in `case` expresssions:

```
data ContactMethod
  = Phone
  | Email
  | Fax

keepModernIfYoung :: Int -> ContactMethod -> ContactMethod
keepModernIfYoung age preferredContactMethod =
  case preferredContactMethod of
    Phone -> Phone
    Email -> Email
    Fax | age < 40  -> Email (1) (3)
        | otherwise -> Fax (2) (3)
```

(1) We first Pattern Match on Fax and then the Guards kick in. If `age` is younger than `40`, we use `Email`.

(2) Our catchall case will be for anyone `40` and older, and we'll use `Fax`.

(3) Don't forget to use `->` instead of `=` for Guards in `case` expressions. This too is easy to forget.

Here's another example of Guards in a `case`:

```
noBiggerThan10 :: Maybe Int -> Int
noBiggerThan10 x = case x of
  Just x  | x > 10    -> 10 (1)
          | otherwise -> x
  Nothing             -> 0
```

(1) The Pattern Match is on the left of `|`, then the conditional, followed by the arrow, and finally the Value. Notice how the `x` in the pattern `Just x` can be used on the right side of the Guard.

Guards and Pattern Matching can work together:

```
takeWhile :: ∀ a. (a -> Boolean) -> List a -> List a (1)
takeWhile p (x : xs) | p x = x : takeWhile p xs (2)
takeWhile _ _ = Nil (3)
```

(1) Takes a Predicate and a `List`, and takes from the head of the `List` while the Predicate is met. It stops as soon as the Predicate returns `false` and returns all the elements it took up to that point in a `List`.

(2) `p x` checks the head of the `List`, `x`, against the Predicate, `p`. If `true` it adds `x` to the head of the recursive call to `takeWhile` with the tail. If, however, it's `false`, there is no catchall Guard and so it goes to the other version of `takeWhile`

(3) This version covers 2 cases. First, it's the `false` case for the first version with the Guard. Second, it's the case for a empty `List`, since `Nil` will not match the second Parameter of the first version of `takeWhile`,

viz. `(x : xs)`.

Let's review what the pattern `x : xs` will match:

```
1 : 2 : 3 : Nil   (x = 1, xs = 2 : 3 : Nil)
1 : 2 : Nil       (x = 1, xs = 2 : Nil)
1 : Nil           (x = 1, xs = Nil)
Nil               (NO MATCH)
```



A ***Predicate*** is a Function that takes a Value and returns a Boolean based on some condition regarding that Value, e.g. `isOdd :: Int -> Boolean` will take an `Int` and return `true` if it's odd.

3.4. Lambda Functions

Earlier, when we were learning a little bit about Lambda Calculus, we saw PureScript's equivalent to Functions in Lambda Calculus:

```
\x -> x + 1
```

This Function is an Anonymous Function, i.e. has no name. We could name it by calling it `f`:

```
f :: Int -> Int
f = \x -> x + 1
```

Since we've named this `f`, there's no reason to use the Lambda. We could just as easily write `f` with the Parameter on the other side of the equal sign:

```
f :: Int -> Int
f x = x + 1
```

So why is this syntax supported. What's the real benefit of a Lambda?

This syntax is helpful when you need to pass a Function to another Function, but it's not worthy of a proper name since it'll only be used once:

```
filter (\x -> x < 10) [1, 2, 3, 10, 20, 30] -- [1, 2, 3]
```

Here the Lambda Function is a Predicate that checks to see if the Value is less than `10`.

`filter` will keep anything from the `Array` that passes the test. It'll iterate through our `Array` and call our Lambda passing the element from the `Array` and if the Lambda returns `true`, then `filter` will keep it, otherwise it's filtered out.

Remember there are NO mutations, therefore the `Array` that you get back from `filter` will be a new `Array`.

To specify multiple Parameters in a Lambda is very similar to normal Functions:

```
\x y -> x + y
```

which is syntactical sugar for:

```
\x -> \y -> x + y
```

If you're used to Javascript, you've seen a similar pattern for manullay writing curried Functions:

```
x => y => x + y // Javascript
```

There are times when Functions have more Parameters than is specified in the Type Signatures. This can be really confusing when you first encounter this, but taking into consideration Lambdas can help.

To illustrate this, let's examine the `compose` Function, usually called via its Binary Operator, `<<<`, which composes two Functions:

```
compose :: ∀ a b c. (b -> c) -> (a -> b) -> (a -> c) ❶
compose f g x = f (g x) ❷ ❸
```

❶ `compose` takes 2 Parameters, both Functions, and returns a Function.

❷ We've specified 3 paramters here. What's up?

❸ `f (g x)` is of Type `c` which does NOT match the return Value in the Type Signature. What's going on?

It turns out that this Function is equivalent to writing it with a Lambda:

```
compose :: ∀ a b c. (b -> c) -> (a -> b) -> (a -> c)
compose f g = \x -> f (g x) ❶ ❷
```

❶ Now it takes ONLY 2 Parameters.

❷ Now it returns a Function from `a` to `c`. The Lambda on the right side of the equal sign takes an `x` of Type `a` and returns `f (g x)` of Type `c`.

We can freely move Parameters across the equals sign as long as we move the rightmost Parameter first and maintain the correct order:

```
f1 x y z =           x + y + z
f2 x y   = \z ->     x + y + z ❶
f3 x     = \y z ->   x + y + z ❷
f4       = \x y z -> x + y + z ❸
```

❶ First we move the rightmost Parameter `z`.

❷ Then we move the next rightmost Parameter `y` and maintain the proper order in the Lambda, i.e. `y` comes before `z`, just like it did in `f1`.

❸ And finally we move the `x` maintaining it's proper position.

To better understand the maintainence of Parameter order, let's look at `f3`. It takes `x` and will return a Function that takes `y` first and then `z`. This maintains the order `x`, `y`, `z` found in `f1`.

In fact, all the Functions are equivalent:

```
f1 = f2 = f3 = f4
```

We've answered the question about the number of Parameters but let's return to our original conumdrum to see another way to look at this:

```
compose :: ∀ a b c. (b -> c) -> (a -> b) -> (a -> c) ❶
compose f g x = f (g x) ❶
```

❶ `f (g x)` is of Type `c` which is NOT what the return Value of this Function shows.

Remember that Type Signatures are Right-Associative which means they have implied Parentheses on the right. This means that the Parentheses on `(a -> c)` are redundant and we can remove them:

```
compose :: ∀ a b c. (b -> c) -> (a -> b) -> a -> c ❶
compose f g x = f (g x) ❷
```

❶ Now we have 3 Parameters!

❷ `f (g x)` is of Type `c` which is our return Type.

Both ways of looking at the Type Signature are valid as are both ways of understanding our original dilemma.

3.5. Wildcards

So far, we've seen _ used as a "don't-care" variable.

There are other times when it's used as a ***Widcard***.

3.5.1. Case Expression

```
data ContactMethod
  = Phone
  | Email
  | Fax

-- verbose version

keepModernIfYoung :: Int -> ContactMethod -> ContactMethod
keepModernIfYoung age preferredContactMethod =
  case preferredContactMethod of ❶
    Phone -> Phone
    Email -> Email
    Fax | age < 40  -> Email
        | otherwise -> Fax

-- simpler with a wildcard

keepModernIfYoung' :: Int -> ContactMethod -> ContactMethod
keepModernIfYoung' age = case _ of ❷
  Phone -> Phone
  Email -> Email
  Fax | age < 40  -> Email
      | otherwise -> Fax
```

❶ Notice that we had to provide a name for a Parameter that would be immediately used in a `case` expression and never again.

❷ Here we write the Function in Point-free notation. This allows the _ to stand for the second Parameter. It's still a "don't-care" Variable since we didn't name it, yet as a Wildcard, we actually use the Value.

The following 2 lines of code are equivalent:

```
keepModernIfYoung' age = case _ of

keepModernIfYoung' age = \someNameIHaveToThinkOf ->
  case someNameIHaveToThinkOf of
```

I think this illustrates well the benefits of using Wildcards.

3.5.2. Operator Sections

Wildcards can also be used in ***Operator Sections***:

```
filter (\x -> x < 10) [1,2,3,10,20,30] -- [1,2,3] ❶
filter (_ < 10) [1,2,3,10,20,30] -- [1,2,3] ❷
filter (10 <= _) [1,2,3,10,20,30] -- [10,20,30] ❸
```

❶ This is our filter Function call from above that uses a Lambda.

❷ This is using an Operator Section with the Wildcard as the left side of the Operator. This is equivalent to the previous Function call.

❸ The is using the Wildcard on the other side of the Operator.



Operator Sections are Binary Operators with one or two Wildcard variables set off by Parentheses.

The following are also a valid Operator Sections:

```
(_ <> "suffix")   -- append the string "suffix" ❶
(_ / 10)          -- div by 10 ❶
(10 / _)          -- div into 10 ❶
(_ == _)          -- equal check ❷
```

❶ These Functions take 1 Parameter.

❷ This Function takes 2 Parameters.

3.5.3. Records

Wildcards can be using in Records:

```
\name age -> {name: name, age: age} ❶
{ name: _, age: _ } ❷
```

❶ A Lambda that takes 2 Parameters and returns a record populated with the values from those Parameters.

❷ This is the Wildcard equivalent of the previous line. It's very important that the order of Parameters is maintained, i.e. the first Wildcard encountered from Left to Right is the first Parameter, the second Wildcard is the second Parameter and so on.

Record updates can also have Wildcards:

```
\name age -> person { name = name, age = age } ❶
person { name = _, age = _} ❷
```

❶ Here's a Lambda that takes `person` and updates 2 of it's fields (`person` could have more than just 2 fields) and returns a new `Record` with just those 2 fields modified. Remember, there are no mutations, so `person` is unchanged.

❷ The same can be accomplished with Wildcards using this syntax.

Another Wildcard example for updates:

```
\person -> person { age = 18 } ❶
_ { age = 18 } ❷
```

❶ The Lambda takes a `person` and updates the `age` returning a new `Record`.

❷ The Wildcard equivalent. This syntax looks odd at first.

3.6. Bindings

We have 2 ways to bind Values to variables. (Don't forget, Functions are also Values in Functional Programming.)

3.6.1. Where

The keyword `where` allows us to define things after the fact in a Function definition:

```
multSum :: Int -> Int -> Tuple Int Int ❶
multSum x y = Tuple mult sum ❷
  where
    mult = x * y ❸ ❺
    sum = x + y ❹ ❺
```

❶ `multSum` takes 2 `Ints` and returns a `Tuple` of `Ints`.

❷ Variables `mult` and `sum` are used here but have not been defined yet.

❸ `mult` is bound to `x * y`.

❹ `sum` is bound to `x + y`.

❺ The `where` section ONLY has access to variables at the same level as the Function, `multSum`, i.e it's Parameters. Any local Function variables are NOT accessible to `where`.

The `where` section is private to the Function `multSum`.

This style of binding doesn't bog down the human reader with too many upfront details. The terms `mult` and `sum` are introduced to the reader first and then their actual definitions are an afterthought in the `where` section.

Sometimes, it's best to defer the gory details of your computation and first present the high-level concepts in the code.

The `Tuple mult sum` tells us a lot. It says we have a multiplication of some sort in the first position of the `Tuple` and an addition in the second.

Once we get to the `where` we can see the details of those calculations. And sometimes, we never need to delve this far, depending on the Function.

For line continuations, the next line has to be indented PAST the first character of variable name:

```
where
  mult = x * y
  sum = x
    + y ❶
```

❶ The first character has to be indented at least one space past the `s` in `sum`.

3.6.2. Let

In constrast, `let` defines things up front:

```
multSum' :: Int -> Int -> Tuple Int Int
multSum' x y =
  let mult = x * y ❶
      sum = x + y in ❷
  Tuple mult sum
```

❶ A `let`, like `where`, can have multiple definitions. With `let`, we are burdening our reader with details here first and then specifying the overall computation.

❷ The `let` block is terminated with the reserved word `in`.

The bindings in the `let` block are only valid within the `in` block.

Unlike `where` which must be part of a NAMED Function, `let` can be used as part of ANY expression:

```
filter (\n -> let n2 = n * n in n == n2) [0, 1, 2] -- [0, 1]
```

If you try this with a `where`, you'll get a compiler error:

```
-- ILLEGAL
filter (\n -> n == n2 where n2 = n * n) [0, 1, 2] -- COMPILER ERROR!!
```

For line continuations, the indentation is similar to `where`:

```
let mult = x * y
    sum = x
      + y in
Tuple mult sum
```

3.7. Binary Operators

We've seen Binary Operators in action but we failed to define them.



A ***Binary Operator*** is an Operator represented by a Symbol that is a Function of 2 Parameters. The Value on the lefthand side of the Operator is the first Parameter and the Value on the right is the second.

Let's look at an example:

```
data List a = Nil | Cons a (List a)

append :: ∀ a. List a -> List a -> List a ❶
append xs Nil = xs ❷
append Nil ys = ys ❸
append (Cons x xs) ys = Cons x (append xs ys) ❹

infixr 5 append as <> ❺
```

❶ `append` will append the second `List` to the first.

❷ The case where the second `List` is empty.

❸ The case where the first `List` is empty.

❹ The general case where we remove the head of the first `List` and PREPEND it to the `append` of the tail of

the first `List` with the whole second `List`.

❺ Definining the operator `<>` as the infixed, Right-Associative operator with Precedence 5.

3.7.1. Associativity

There are 3 types of Associativity for Binary Operators:

- `infixr` = Right-Associative
- `infixl` = Left-Associative
- `infix` = None

The operator `<>` is Right-Associative, which means the following two lines are equivalent:

```
l1 <> l2 <> l3 <> l4
(l1 <> (l2 <> (l3 <> l4)))
```

Notice how the Parentheses collect on the Right.

The operator `+` is Left-Associative, which means the following two lines are equivalent:

```
n1 + n2 + n3 + n4
(((n1 + n2) + n3) + n4)
```

Notice how the Parentheses collects on the Left.

The operator `==` is Non-Associative, i.e. is defined using `infix`. This means that repeated use of the operator (or use with other Non-Associative operators) in an expression MUST have EXPLICIT Parentheses:

```
true == false == false     -- COMPILER ERROR!!
(true == false) == false   -- compiles with Value of true
true == (1 == 42)          -- compiles with Value of false
```

3.7.2. Precedence

The `*` operator has a Precedence of 7, whereas `+` has a Precedence of 6, which means the following two lines are equivalent:

```
2 * 3 + 4
(2 * 3) + 4
```

The higher the Precedence number, the sooner it's evaluated. Here `*` is higher than `+` and so it's evaluated first.

In our `append` definition above, we specified:

```
infixr 5 append as <>
```

That defines the operator `<>` as the equivalent to calling `append` infixed with Right Associativity and a Precedence of 5. That means it's evaluated before any other operators that are part of the same expression that have a lower Precedence, i.e. 4, 3, 2, 1, and 0.

3.7.3. Fixity

`infix` means Infixed Operator, i.e. it comes between it's two values:

```
l1 <> l2
```

Compare that with the Prefixed equivalent:

```
append l1 l2
```

We can use an Infixed Operator in Prefixed Position and we can use a Function in Infixed Position:

```
(<>) l1 l2 ❶
l1 `append` l2 ❷
```

❶ The Operator must be surrounded by Parentheses to be used in Prefixed Position.

❷ The Function name must be surrounded by Back-ticks to be used in Infixed Position.

Infixing a named Function makes sense when it helps readability, e.g. `elem` is a Predicate that returns `true` if a Value is an element of an `Array`:

```
if 42 `elem` [1, -5, 42] then "YES" else "NO"
```

We can read this as: `if 42 is an element of [1, -5, 42]…`

3.8. Comments

We've seen inline comments up to this point, e.g.:

```
add :: Int -> Int -> Int
add x y = x + y -- add 2 numbers
```

The inline comment is prefixed with `--`.

Block comments can be accomplished as follows:

```
{-  This Function will add two numbers
    and return the answer, which is
    the addition of those two numbers.
-}
add :: Int -> Int -> Int
add x y = x + y -- add 2 numbers
```

Comments that start with a pipe character, `|`, are considered Documentation for tools such as `psc-docs` and `Pursuit`, the online repository of PureScript API documentation. Pursuit can be found at https://pursuit.purescript.org/.

3.9. Inferring Functionality from Type Signatures

All Functions have Type Signatures whether defined explicitly by the developer or determined implicitly by the compiler based on how the developer uses the Parameters.

Best practices deems that at least all top-level Functions in a module have explicit Type Signatures.

Not only does this help by improving the readability of the codebase, but it helps the compiler to produce better error messages. But Type Signatures aren't just helpful to the compiler. We can imply a lot from just a Type Signature.

Take the following Type Signature:

```
f :: Int -> Int -> Int
```

Take a guess at what kind of Function `f` might be.

Here is a list of possible Functions for `f`:

```
max :: Int -> Int -> Int
min :: Int -> Int -> Int
add :: Int -> Int -> Int
sub :: Int -> Int -> Int
mult :: Int -> Int -> Int
div :: Int -> Int -> Int
```

And we could go on from here.

But what about more abstract Type Signatures such as:

```
f :: ∀ a. a -> a
```

How many Functions can you write that perform this? Try to list them all out before reading any further.

Remember, that `a` is an unknown Type to the developer of `f`. It can be ANY Type, e.g. an `Int`, `String`, `Array Unit`, `List (Maybe Int)`, anything.

Now how many Functions can you think of?

The way to think this through is to realized that you're given a value of some UNKNOWN Type `a` and you have to return a value of some UNKNOWN Type `a`. You cannot write any code to look at the `a` because you don't know what `a` is. But you still need to produce an `a`. And the only `a` you have is the one you've been given.

Therefore, there is only 1 Function that can have this signature:

```
identity :: ∀ a. a -> a
identity x = x
```

`identity` takes the only `a` it has and returns it back unchanged since it doesn't know anything about `a`.

What about the following Function:

```
f :: ∀ a b. a -> b
```

How many Functions can your write with this Type Signature?

Thinking this through, we are given an `a` and must produce a `b`. But we don't know what Type `a` is and we don't know what Type `b` is. So how are we supposed to convert an UNKNOWN `a` into an UNKNOWN `b`?

The answer is we cannot. There are no Functions with this Type Signature. If you find this hard to believe,

try to imagine writing a Function where `a` is an `Int` and `b` is a `String`. But that SAME Function must also work when `a` is a `String` and `b` is an `Int`. Remember, the caller of your Function determines which Concrete Types `a` and `b` are, not you.

How about this Type Signature:

```
f :: ∀ a. List a -> Maybe a
```

Try to enumerate how many Functions you could write before moving on.

In the above example, we have a `List` of elements which are of Type `a`, i.e. UNKNOWN Type to us. And we must produce an `a` and since our only source of `a's` is the `List`, it must come from the `List`. We also have the possibility of failure as signified by the `Maybe`.

Here's a reasonable list of Functions with that Type Signature:

```
first :: ∀ a. List a -> Maybe a ❶
last :: ∀ a. List a -> Maybe a ❷
constIndex :: ∀ a. List a -> Maybe a ❸
```

❶ Tries to return the first thing in the `List`. Returns `Nothing` if the `List` is empty.

❷ Tries to return the last thing in the `List` or `Nothing` if it fails, i.e. for an empty `List`.

❸ Tries to return the element at some hardcoded index. Returns `Nothing` if the index is out of bounds.

It turns out that `first` and `last` are just special cases of `constIndex`, so for all practical purposes, there is only 1 unique Functions with this Type Signature.

What about:

```
f :: ∀ a. List a -> a
```

What does this Function appear to do?

Given a `List`, this Function ALWAYS produces an `a`. The problem is that this promise cannot be honored if we pass it an empty `List`.

There are no ***Total Functions*** that can be written with this Type Signature. We can, however, write a ***Partial Function***, but it would crash on the empty `List` case.



A Partial Function is one where all of the cases are not handled. A Total Function is one where all possible calling scenarios have been accounted for. PureScript discourages Partial Functions.

How can we take this Partial Function and make it Total?

Here's the Type Signature for a Total version:

```
f :: ∀ a. a -> List a -> a
```

Can you see what this does? Try to imagine the Function that we'd write for this Type Signature before moving on.

The newly added Parameter is a Default Value:

```
alwaysFirst :: ∀ a. a -> List a -> a
alwaysFirst def list = ... ❶
alwaysLast :: ∀ a. a -> List a -> a
alwaysLast def list = ... ❶
```

❶ The details of the Function are not important here.

If given an empty `List`, these Functions will return the supplied default.

Let's try another Type Signature:

```
f :: ∀ a. a -> Int -> List a -> a
```

This is pretty close to the previous Type Signature with the addition of an `Int` Parameter. What can we do with that Parameter?

The most logical explanation is that the `Int` is an index into the `List`:

```
getAt :: ∀ a. a -> Int -> List a -> a
getAt def index list = ... ❶
```

❶ The details of the Function are not important here.

Once again, if given an empty `List`, this Function will return the supplied default.

What would you put in place of the `???` in the following:

```
f :: ∀ a. Int -> List a -> ???
```

This looks like our `getAt` Function above, but it's just missing the default Parameter. That means that we could fail in the case of an empty `List`.

That means we have:

```
getAt' :: ∀ a. Int -> List a -> Maybe a ❶
getAt` index list = ... ❷
```

❶ `???` was replaced by `Maybe a` to account for the possibility of a failure.

❷ The details of the Function are not important here.

Here's another one:

```
f :: ∀ a b. a -> Int -> (a -> b) -> List a -> b
```

This may seem pretty complex, but it's actually not. It looks a lot like `getAt` but with the added Type `b` and the extra Parameter `a -> b`.

So this Function simply does what `getAt` does, but before it returns a Value, it converts it from Type `a` to `b` using the supplied conversion Function (3rd Parameter).

Here's what this Function looks like:

```
getAtConvert :: ∀ a b. a -> Int -> (a -> b) -> List a -> b
getAtConvert def index convert list = ... ❶
```

❶ The details of the Function are not important here.

But wait. I thought we couldn't have a Function from `a -> b`. What gives?

While it's true that we cannot write a Function from ANY `a` to ANY `b`, we could write a Function from `Int` to `String` and then call `getAtConvert` with it:

```
intToString :: Int -> String
intToString i = show i

getAtConvert (-1) 2 intToString (1 : 2 : 3 : Nil) ❶
```

❶ Here our default value is `-1` and our index is `2`.

When we call this Function with `intToString`, our Type Parameter `a` gets unifies with `Int` and `b` gets unifies with `String`. Our `convert` Function's Type is `a -> b` or, in this particular case, `Int -> String`. A Function from `Int` to `String` is a Function we can write.

We cannot have a free-standing Function with the Type Signature of `a -> b` since we cannot write code for converting ANY `a` to ANY `b`. But in this case, the caller of our Function can pick Concrete Types for `a` and `b`,

write a conversion Function for those Concrete Types and then pass that Function to us.

We don't have any idea what `a` and `b` will be when we write our Function, so the best we can do is take something of Type `a` and give it to that Function to get a Value of Type `b`.

Notice that we didn't need to look at the implementation of the Function to guess what that Function does and what roles the Parameters played in the Function.

For example, if we have the following:

```
inBetween :: Int -> Int -> Boolean
```

We have enough information to make a pretty good guess as to what it does and what each Parameter SHOULD be. In this example, convention would dictate that we have a starting and ending number in that order. The only real question is whether `inBetween` is inclusive of it's endpoints or not.

If we see:

```
singleton :: ∀ a. a -> List a
```

We can surmize that this takes a Value of Type `a` and puts it into a `List`.

When we see:

```
f :: ∀ a. a -> a -> Maybe a
```

The best we can surmize here is that `f` takes 2 values of `a` and possibly returns one of them based on some internal criteria.

And with:

```
f :: ∀ a. a -> (a -> Boolean) -> Maybe a
```

We can see that we're passing the criteria (Predicate) for returning `a`. The criteria is the 2nd Parameter which will take a an `a` and return `true` or `false`, which we can reasonably assume will be returned if `true` and not if `false`.

A lot can be determined by just looking at the Type Signatures of Functions. This process will come it handy when we start coding and we're looking at library documentation that somewhat vague or incomplete, which unfortunately is more the rule than the exception.

3.10. Summary

We've learned a lot in this Chapter about some of the basic parts of PureScript from Types to common language constructs. There will be more to come. And while theory is important, getting some hands on experience can help cement that theoretical understanding. And often times, we think we understand something only to learn otherwise when we attempt to use our newfound knowledge.

You will probably find yourself returning to this Chapter as you work out the coming exercises. Now it's time to get our hands dirty.

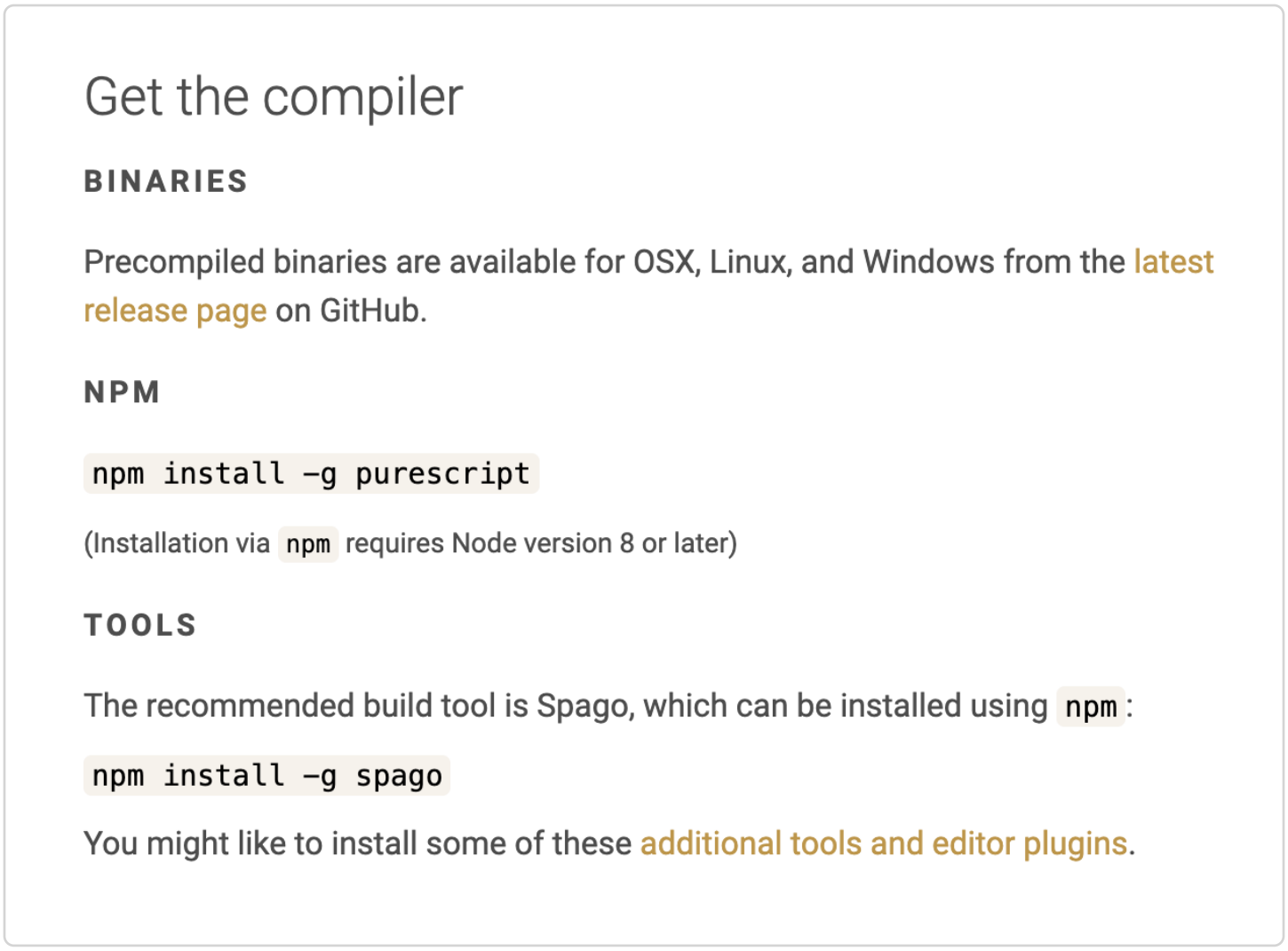
Chapter 4. Installing PureScript

Of all the Chapters of this book, this one will become obsolete the fastest. That's because in the software world things change constantly. The tools I'm going to suggest are the best ones that I know of at the time of this writing and I expect that they will be around for at least the next 5 years.

If you're reading this around 2025 or after, you may want to check to see if there are newer/better tools than I'm suggesting here. Even then, I'd suggest you still read through this to make sure you're getting all of the same features and capabilities.

4.1. Compiler and Tools

Navigating to http://www.purescript.org/ in 2020, we find the following:



This will get you up and running pretty quickly, but I'd like to suggest NOT following this process. I believe installing the tools locally is a much better long term strategy. The choice is totally up to you but at least hear me out first.

The reason for this recommendation comes from my experience working with many different Haskell projects in production. The Haskell build tool, `stack`, helps you to manage multiple versions of `GHC`, the Haskell compiler, and libraries on your system at one time by installing it in each project.

This is really a nice feature since developers can work on any of the projects without worrying about a single global compiler version. Most of the time, our Haskell projects are using different versions of the compiler so this is imperative.

The recommended build tool for PureScript, `spago`, doesn't do this automatically, but with a little effort, you can accomplish this and get the same benefit.

4.1.1. Installing `Node`

When PureScript runs outside of the browser it runs in `Node`. We'll need it installed so we can write simple programs as we learn without all the overhead of the browser.

Also, we're going to use `npm`, which comes with `Node`, to install the compiler and build tool locally into our projects.

Commands are run from the **Terminal** or **Command Prompt**.

For `Node` Already Installed

If you do NOT have `Node` installed in your system, then skip this step. If you're not sure run the following command:

```
node -v
```

If it prints out a version, then you have it installed. If not, then skip this sub-section.

Next step is to figure out if you need to upgrade it. Run the command:

```
npm -v
```

If it prints out `5.2.0` or higher, you can skip upgrading `Node` because you have a version that includes `npx`, which we will use heavily.

Otherwise, you'll need to upgrade to a version of `Node` that's at least `8.2.0` or higher.

For `Node` NOT Installed

I'd recommend installing `n`, a `Node` version manager. Since `Node` is global, you can only have one at a time installed. `n` let's you manage which one you're using at any one time.

This is TOTALLY optional and you can completely skip this and just install `Node` from the Node website but

I'd recommend installing `n`.

You can install `n` (visit The `n` GitHub repository for the latest instructions):

- Mac
  - Using Homebrew - `brew install n`
- Mac or Linux
  - Manual process - `curl -L https://git.io/n-install | bash`
- Windows
  - NOT SUPPORTED (see Node Version Manager for Windows)

Once `n` is installed, you can install `Node` using ONE of the following commands:

```
sudo n latest # install the latest release
sudo n lts # install the newest Long Term Support release
```

For Windows Users using `nvm` you can install `Node` using ONE of the following commands:

```
nvm install latest 64 # install latest release 64-bit version
nvm install latest 32 # install latest release 32-bit version
nvm install latest all # install latest release 64 & 32-bit versions
```

You can test that you have the right version of `Node` by running:

```
node -v # version should be at least 14.15.4 LTS or 15.7.0 for latest
```

You can double check that you have the right version by running:

```
npx -v # should display 5.2.0 or higher
```

If all is right here, then you're ready to create your first PureScript project.

4.1.2. Create Project

First thing we must do is to create a directory for our project. You probably have a location you keep your programs already and so I'd suggest you create a directory under that called `purescript` so you can keep all of your PureScript code together.

If your code is kept in a folder called `/dev`, then you'd run:

```
mkdir /dev/purescript
```

Next, create the project directory:

```
cd /dev/purescript
mkdir fp-book # or some other name you like better
cd fp-book
```

Before we go further ***make sure that you're in your project directory***.

Next, we are going to do the following to fully create our PureScript project:

1. Initialize the project for working with `npm` and `npx`.
2. Install `purescript` and `spago` (Locally).
3. Initialize the project for working with `git` (Optional). This is useful if you use VSCode.

Initialize Project for `npm` and `npx`

To initialize your project so that we can install `npm` packages and use `npx` to run the compiler and build tool from inside your project, run:

```
npm init -y
```

This will create a file call `package.json`. This will keep track of any packages we install for both production and development. For example, we may install `uuid` and `uuid-validate` which are node `npm` packages that are required by the PureScript package `purescript-uuid`.

Future-proofing

Since technology changes so quickly, new versions of libraries, compilers and tools can break code in this book. To guarantee that you're using the exact same version I used when I wrote this book, I will ask you to install `spago` and `purescript` using their explicit versions. You can choose to not do this, but then you'll be on your own regarding breaking changes in the compiler.

Don't worry that you're missing out on something just because you're using an older version of the compiler (circa 2020). The concepts that you'll learn here are universal and will be valuable for decades to come.

Install `purescript` and `spago` into your Project

To locally install the PureScript compiler, `purescript` and the build tool, `spago`, run:

```
npm install --save-dev spago@0.20.0 purescript@0.14.1
```

or to live dangerously and install the latest versions run (remember, you're on your own if you choose this):

```
npm install --save-dev spago purescript
```

Notice the `--save-dev` option. This will tell `npm` that these packages are only for development.

To verify that we have installed the compiler locally, we'll use `npx` (NOTICE the `x`):

```
npx purs --version # should print out 0.14.1
```

`npx` looks in our local `node_modules` directory for `purs`, the name of the PureScript compiler, and then runs it passing all other command line Parameters to it. The parameter `--version` tells `purs` to print out it's version.

To verify that we have installed the build tool locally, run:

```
npx spago version # should print out 0.20.0
```

Initialize Project for `git` (Optional)

To prepare our project for version control using `git`, run:

```
git init # should print `Initialized empty Git repository in <dir>`
```

Although this is optional, I'd still recommend doing this.

4.1.3. Initialize PureScript Project

The final step is to initialize this project for building a PureScript program using `spago`:

```
npx spago init
```

If you're curious to see everything that `spago` creates, you should take a look at the contents of your project directory:

```
ls -l
```

If you're using `git`, you'll be happy to know that `spago` gives you a pre-populated `.gitignore` file.

We can build the project by running:

```
npx spago build
```

or we can build AND run the project by running:

```
npx spago run # should print out the <spaghetti-emoji>
```

4.1.4. A Second Project

While this seems like a lot to do to get up and running, most of it was first-time configuration.

To start a second project involves:

```
# make project directory
mkdir /dev/purescript/second-project
# enter project
cd /dev/purescript/second-project
# initialize for using `npm` and `npx`
npm init -y
# add compiler and build tools to the project
npm install --save-dev spago@0.20.0 purescript@0.14.1
# initialize our `git` repo (Optional)
git init
# create a skeleton PureScript project
npx spago init
# build and run the skeleton project
npx spago run
```

4.2. Editor and Plugins

At the time of this writing, there are PureScript plug-ins for the following editors:

1. Atom
2. Emacs
3. Sublime
4. Vim
5. VSCode

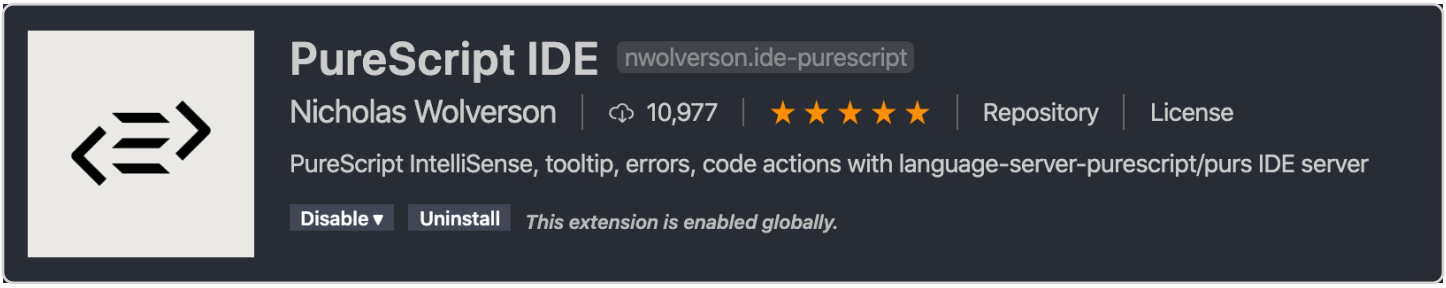
When you're reading this, there may be more. Go to the Purescript website to check for new tool support.

I'm going to cover `VSCode` since it's what I use, but I started in Atom and while the configuration process is different, the developer experience is very similar.

If you're going to use a different editor, then please read on and make sure you do the equivalent things for your editor.

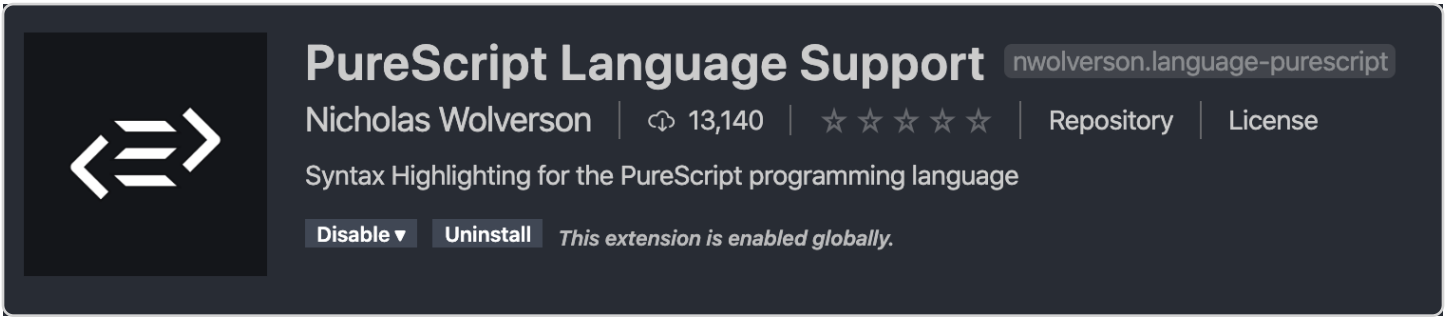
4.2.1. Install Extensions

Open the extensions search in VSCode and search for `PureScript IDE`:



Click on Install.

Next, search for `PureScript Language Support`. It probably will already be installed by the `PureScript IDE` extension:



If it's not already installed, then click on Install.

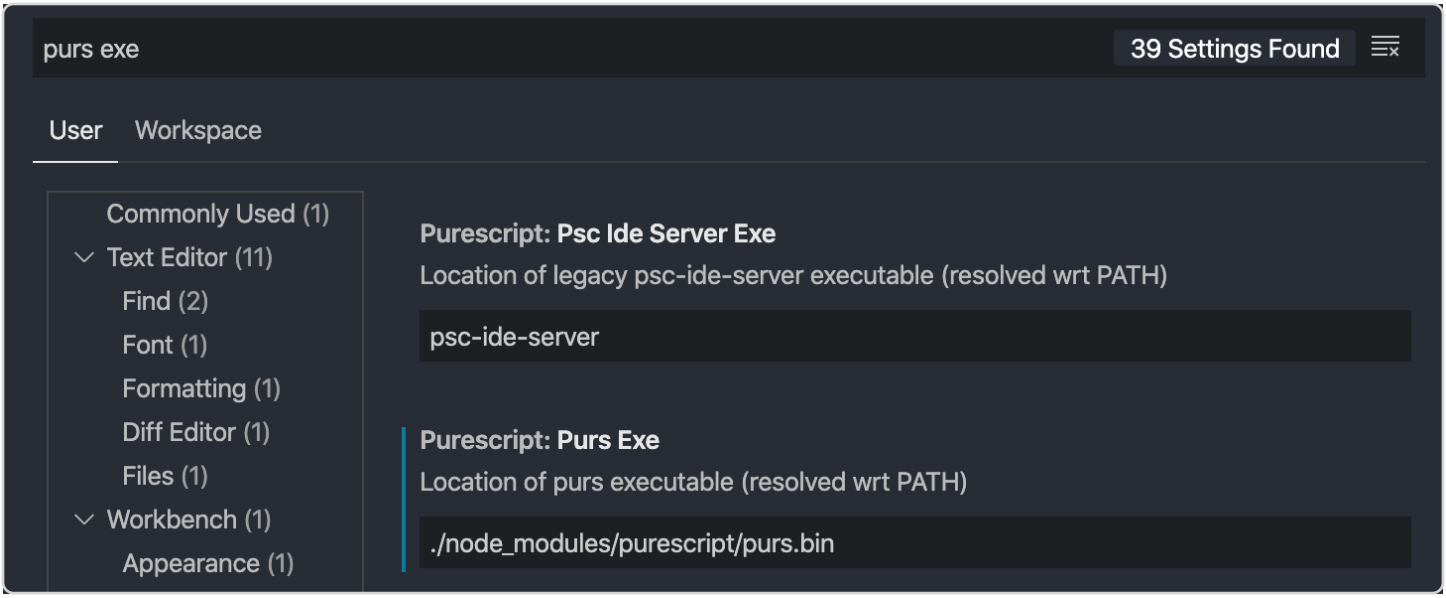
4.2.2. Configure Extensions

Skip this step if you installed your compiler globally.

Open up Preferences:

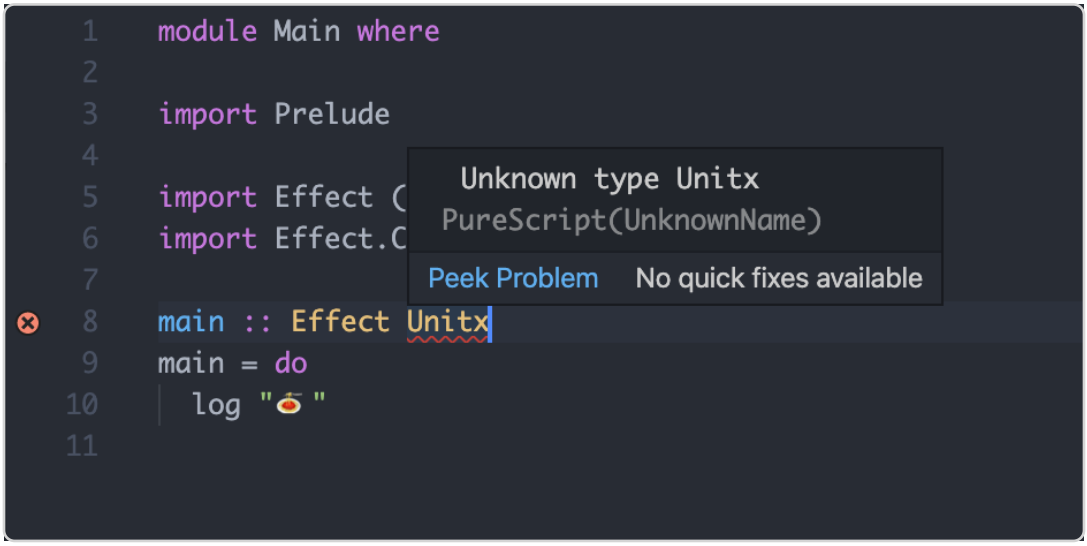
- Mac = Code > Preferences > Settings
- Windows/Linux = File > Preferences > Settings

Search for `purs exe` and change `Purescript: Purs Exe` path to point to our locally installed compiler, i.e. `./node_modules/purescript/purs.bin`:



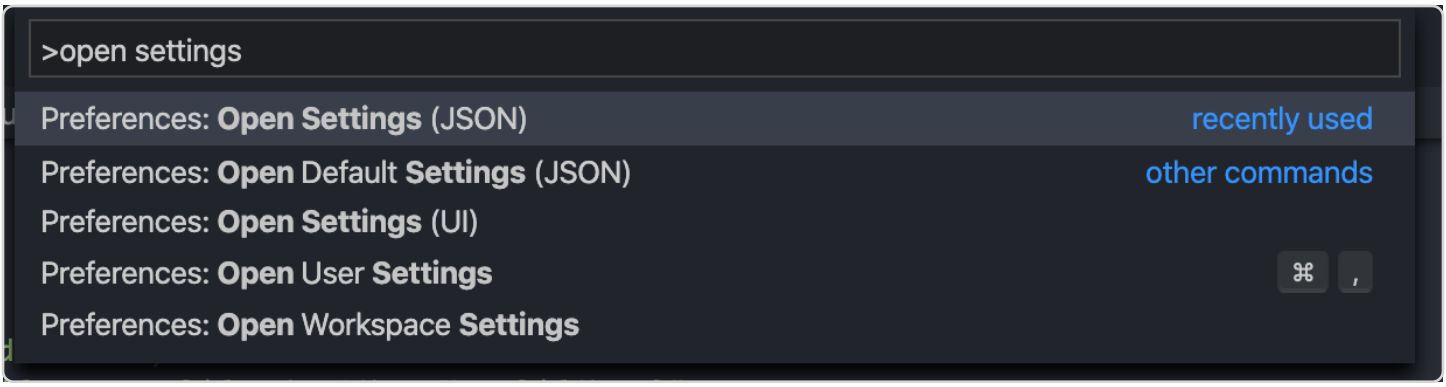
If, in the future, this location changes due to some change in `npm`, you'll have to locate the PureScript compiler in `node_modules` and configure with the appropriate path.

Once properly configured, the editor will be able to compile when we save our code and give us error messages in a pop-up window when we hover our mouse over it:



4.2.3. Configure Editor

Use `Ctrl-Shift-P` on Windows/Linux or `Cmd-Shift-P` on the Mac to open the `Command Pallete`. Then type `open settings`:

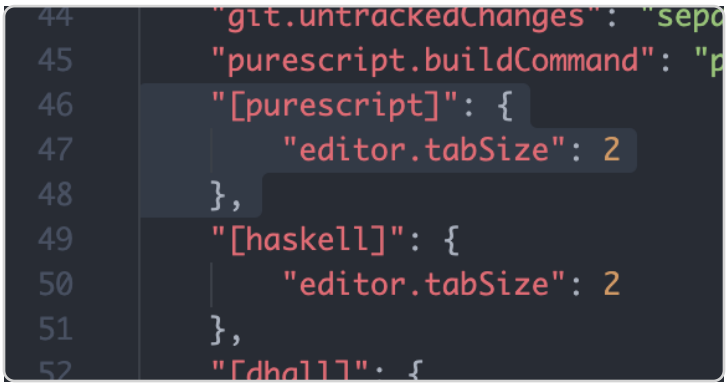


Select `Preferences: Open Settings (JSON)` and you will now be editing your `settings.json`.

Place the following code in the JSON object:

```
"[purescript]": {
    "editor.tabSize": 2
}
```

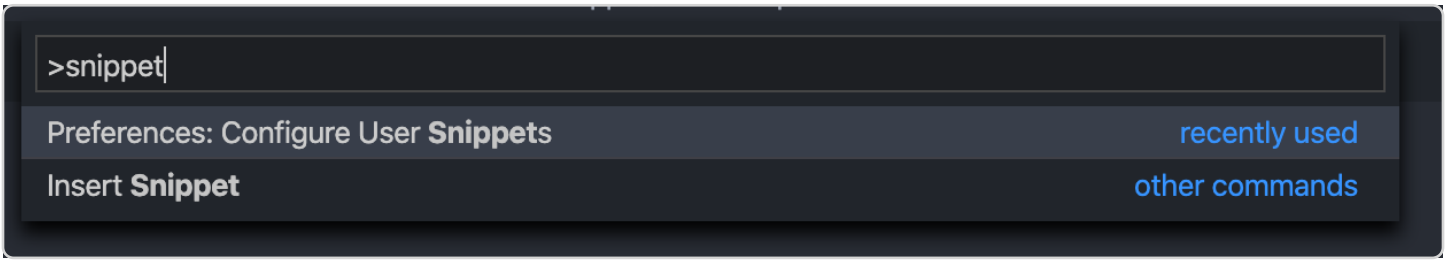
It doesn't matter if you place it first, middle or last:



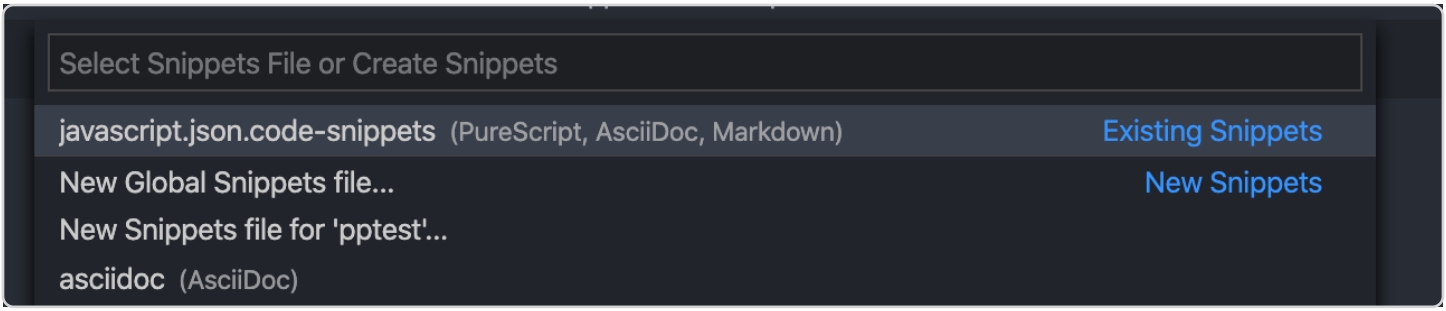
Make sure you save and close the file.

Next let's add a Snippet for the Mathematical Symbol for the `forall` keyword.

Open up the `Command Palette` again and type `snippet`:

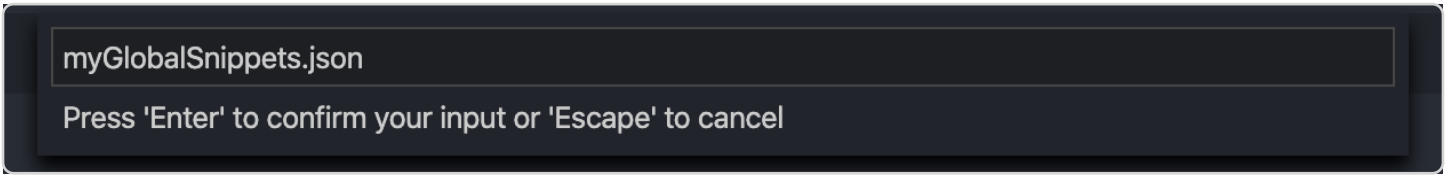


Select `Preferences: Configure User Snippets` and you get another menu:

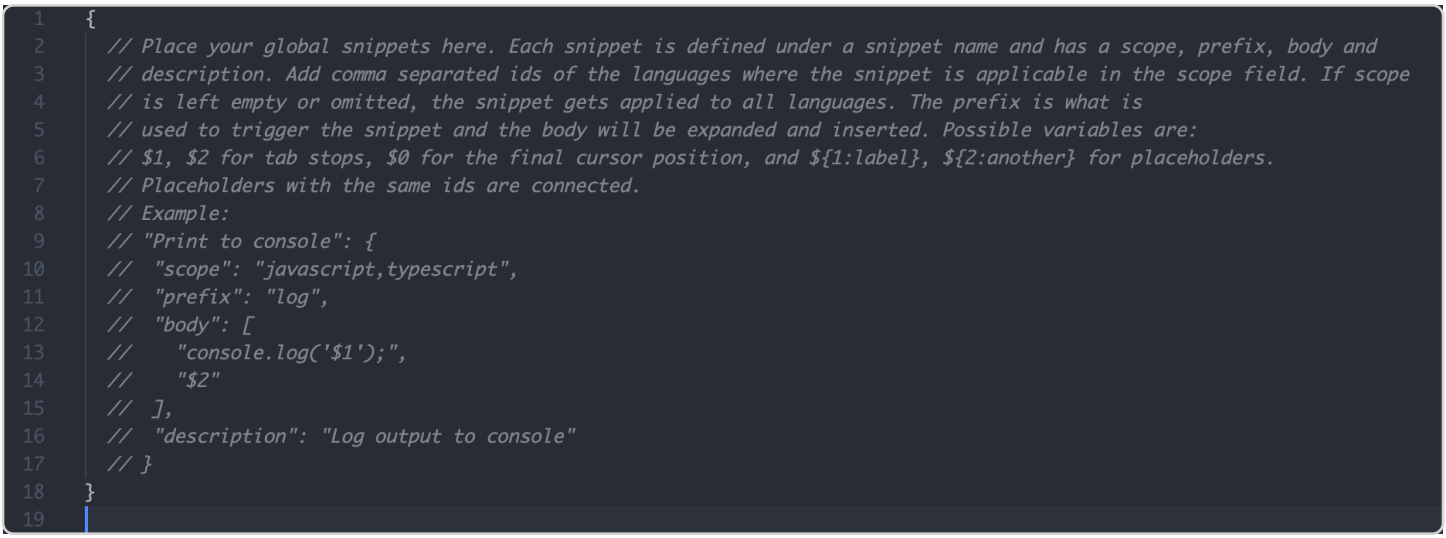


If you already have Snippets file then choose it. Mine here is called `javascript.json.code-snippets`.

If this is your first time creating Snippets then choose `New Global Snippets file...` and name it something you'll remember:



You'll then be presented with your snippets JSON file:



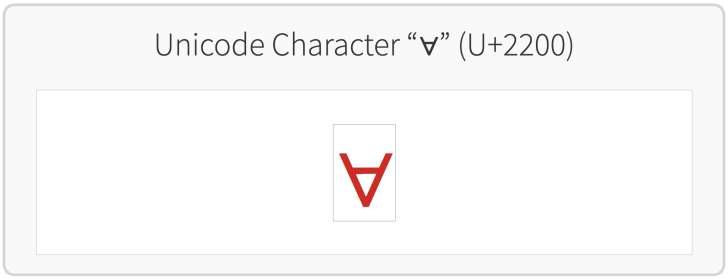
Next add the following text to the end of the JSON object:

```
    "Forall": {
        "scope": "purescript",
        "prefix": "fa",
        "body": "∀"
    }
```

```
  // "description": "Log output to console"
  // }
  "Forall": {
    "scope": "purescript,asciidoc,markdown",
    "prefix": "fa",
    "body": "∀"
  },
}
```

If you're reading this as a PDF, then you can copy/paste from this book. If you're reading this on paper, then you'll need to type this code.

The easiest way to type `∀`, is to copy and paste it from the internet. Either Google for `unicode forall` or get the ForAll here.



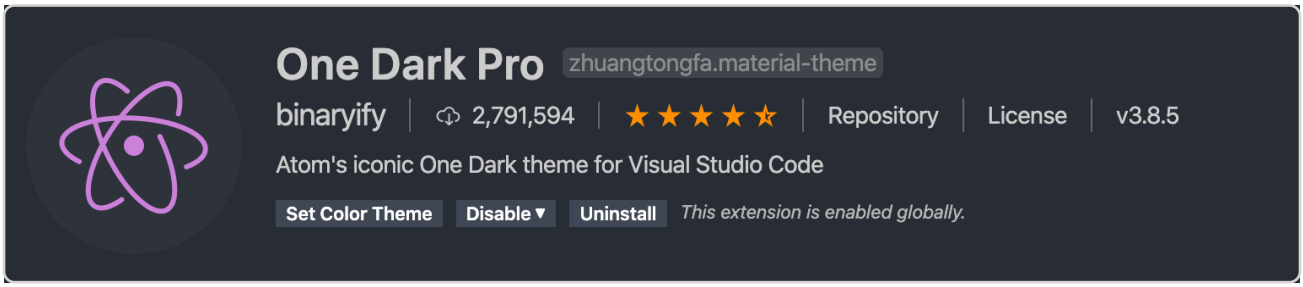
Then select the `∀` and copy and paste it into the JSON file.

Make sure you save and close the file.

Once this is configured (you may have to restart VSCode), you can type `fa` followed by `Tab` and the `fa` will be replaced by `∀`. This is configured to only work when you're editing `purescript` files.

And finally, I'd suggest installing the `One Dark Pro` extension. It's the one I'll use in the screenshots and it's color scheme works well with PureScript and Haskell.

If you'd like to do this, then simply install the extension and then click on the `Set Color Theme` button:



You are now ready to start programming in PureScript.

Chapter 5. Basic Coding in PureScript

Reading a book can lull us into the false sense that we understand what we've read. This happens to me, especially in programming. We understand it as it's being explained, but we haven't fully understood it until we've utilized it.

So that's what the exercises are for. And if you're like me and tend to skip exercises in books, please do not. There's more to these exercises than meets the eye.

Many concepts will get introduced in the exercises. Many commonly used library Functions will also be introduced during this process. You will also learn how to program, and more important, think in a Functional way using PureScript.

5.1. Prelude

`Prelude` is library of commonly used Functions that nearly ever program will use. Unlike Haskell, you have to explicitly `import` this module into your code:

```
import Prelude
```

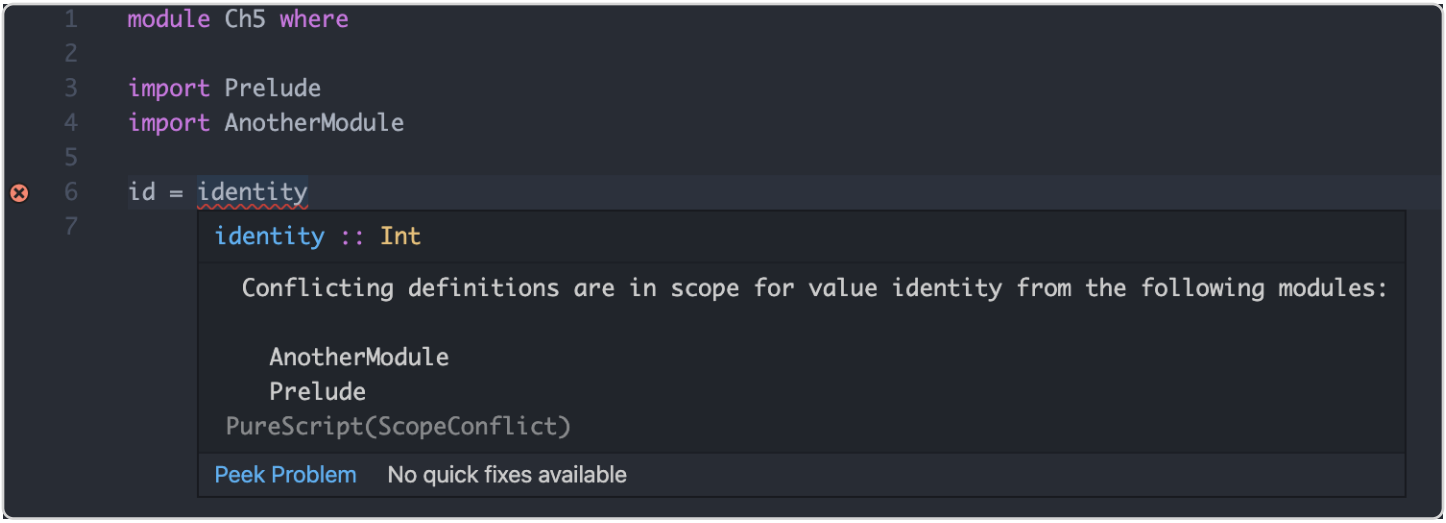
When you import a module like this, ***all*** exported Functions get imported into your module's global namespace. This can be problematic if they collide with Functions from other modules.

Let's assume that we have a module called `AnotherModule` that exports `identity` and `doSomething`. If we import as follows and then try to use `identity`:

```
import Prelude
import AnotherModule

id = identity   -- COMPILER ERROR!!
```

The compiler will complain that `identity` is ambiguous:



We can fix this by ***selectively importing***:

```
import Prelude
import AnotherModule (doSomething) ❶

id = identity
```

❶ Import `AnotherModule` but ONLY include `doSomething`.

Or we can decide to use `identity` from AnotherModule but we want everything else from Prelude:

```
import Prelude hiding (identity) ❶
import AnotherModule (identity, doSomething) ❷

id = identity
```

❶ Import `Prelude` while ***hiding*** `identity`, i.e. do not import it.

❷ Explicitly specify the import of `identity` and `doSomething` from `AnotherModule`.



It is considered good practice to ***explicitly*** specify all Functions that are imported from modules. In small programs, you can get away with not doing this, but as your codebase grows, it becomes increasingly difficult to determine where Functions have been imported from. The only exception to this rule is `Prelude`.

But what if we want to use `Prelude's identity` sometimes and `AnotherModule's` other times?

```
import Prelude
import AnotherModule (doSomething) ❶
import AnotherModule as AM ❷

id = AM.identity ❸
```

❶ Still import `doSomething` into the global namespace.

❷ Import the whole module, `AnotherModule`, but require that it's Functions must be qualified with the prefix `AM.`, e.g. `AM.identity`.

❸ Qualify that `identity` is from module `AM`.

5.2. Exercise Program

We can use the project that we created in the previous chapter for coding the coming exercises. Let's make our `Main` module, named `Main.purs` look like the following:

```
module Main where ❶

import Prelude
import Effect (Effect)

import Ch5 as Ch5 ❷

main :: Effect Unit ❶ ❸
main = Ch5.test ❹
```

❶ All PureScript programs must have a `Main` module with a `main` Function.

❷ We import our code module as `Ch5`.

❸ Don't worry about the Type Signature just yet.

❹ We call a `test` Function in the `Ch5` module. That's where we're going to put our code to test our versions of some of the `Prelude` Functions we're going to write as exercises.

And then create a file under the `src` directory called `Ch5.purs` with the following code:

```
module Ch5 where ❶

import Prelude (Unit) ❷

import Effect (Effect) ❸
import Effect.Console (log)

test :: Effect Unit ❸
test = do ❸
  log "test" ❹
```

❶ We define and name our module.

❷ We are only importing the `Unit` Type because we're going to be writing our own versions of many of the `Prelude` Functions here and we don't want name collisions.

❸ Don't worry about these lines. We will eventually get to them when we talk about Effects.

❹ This logs the `String` `test` to the console. We'll delete this later.

Now to run this program in `Node`, we can open the Terminal in VSCode by typing `Ctrl+`` and then typing:

```
npx spago run
```

Our program should print out the `String` `test`.

We are now ready to start coding.

5.3. Pursuit

Since we are going to writing a lot of Functions that already exist in the `Prelude` library, we should get used to looking up Function definitions.

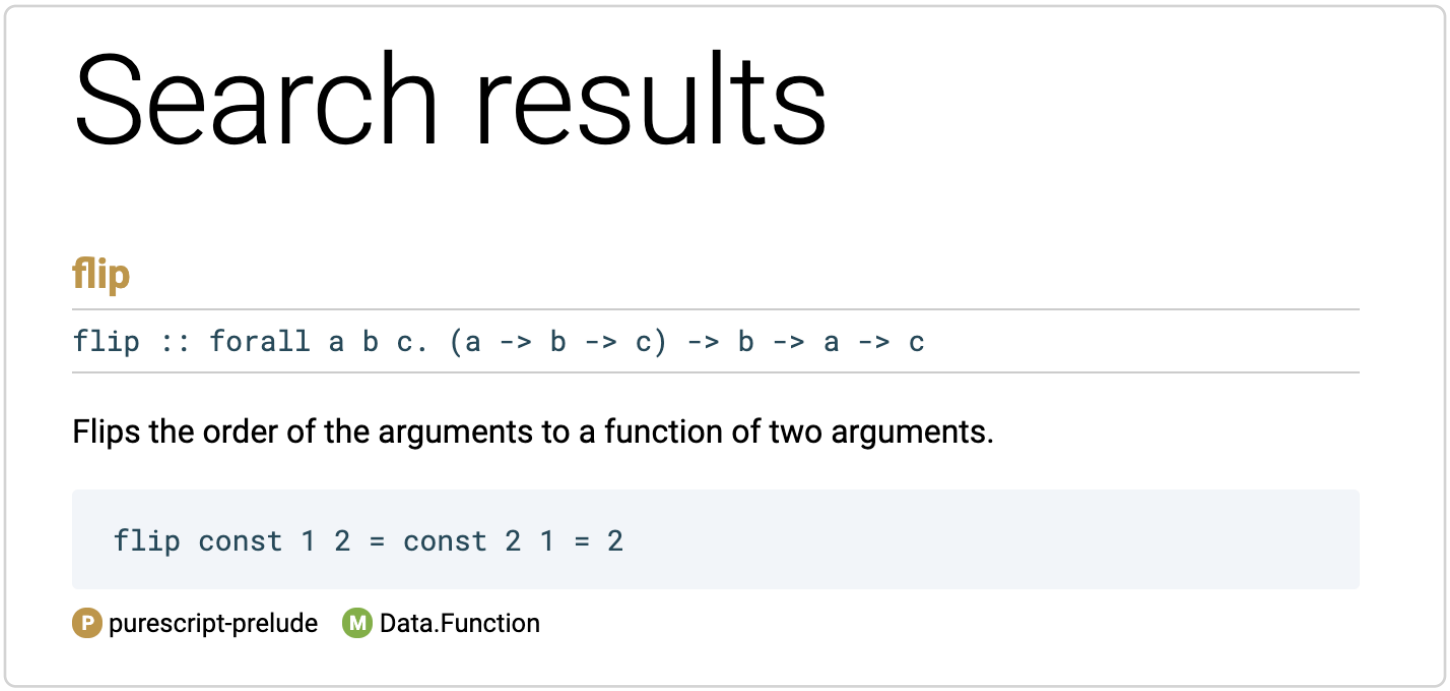
There is a website for doing this called `Pursuit`. It can be found on the web at https://pursuit.purescript.org/. When you're there, you can search for Functions using the Search field.

Before you code any of the Functions in the Chapter, please first consult Pursuit to find its Type Signature and what it's supposed to do.

Let's look for `flip` by typing the name of the Function and hitting `Enter`:



You should see the following definition:



This Function's Type Signature is:

```
flip :: ∀ a b c. (a -> b -> c) -> b -> a -> c
```

I've used `∀` instead of `forall` as shown in `Pursuit`.

We want to understand what this Function does and sometimes we'll get a nice example (other times not):

```
flip const 1 2 = const 2 1 = 2
```

Unfortunately, we don't know what `const` does just yet so this is marginally helpful. It says that calling `flip` on the Function `const` (whatever that does) with Parameters `1` and `2` is the same as calling `const` with those Parameters reversed. It also shows the result of the `const` call being `2`.

Another thing to notice is both the package and module where `flip` can be found. The package is `purescript-prelude` and the module is `Data.Function`.

5.4. Writing `flip`

Sometimes a Function's Parameters are in the wrong order for how we'd like to use them. This is where `flip` comes in. Try to code `flip` in the `Ch5` module and then write the following line into your `test` Function:

```
log (show (flip const 1 2))
```

Also. delete the line:

```
log "test"
```

Your module should look like:

```
module Ch5 where

import Prelude (Unit, show, const) ❶ ❷

import Effect (Effect)
import Effect.Console (log)

test :: Effect Unit
test = do
  log (show (flip const 1 2)) ❸
```

❶ We need the `show` Function from `Prelude` to turn the `Int` we'll ultimately get from the `flip const` call into a `String` that we can pass to `log`.

❷ We also need `const` for now. Ultimately, we'll write this Function and so we'll need to remove this once we get there.

❸ Here's our test code.

Now all you have to do is write the flip Function. Please try to do so without looking ahead. Once you're done or if you get stuck continue on.

5.4.1. Hint for `flip`

Let's take a look at the details of the Type Signature for `flip`:

```
flip :: ∀ a b c. (a -> b -> c) -> b -> a -> c
```

It tells us that `flip` takes a Function of 2 Parameters (`a` and `b` returning `c`), a Parameter of Type `b`, Parameter of Type `a` and returns a value of Type `c`.

Our first step is to name these Parameters:

```
flip :: ∀ a b c. (a -> b -> c) -> b -> a -> c
flip f x y = ???
```

See if that helps you write the Function now.

5.4.2. Code for `flip`

Here's the code for `flip`:

```
flip :: ∀ a b c. (a -> b -> c) -> b -> a -> c
flip f x y = f y x ❶
```

❶ We call `f` with the original Parameter order flipped.

We can test this by running our program:

```
npx spago run
```

It should print out `2`.

5.4.3. Alternative Coding for `flip`

What would have happened if `flip's` Type Signature had the redundant Parentheses added:

```
flip :: ∀ a b c. (a -> b -> c) -> (b -> a -> c)
```

Now it takes a single Function and returns a Function. Since Type Signatures are Right-Associative, these Parentheses are implied, but seeing them changes how we think about the Function.

How might you have written it differently if you saw this version of the Type Signature?

```
flip :: ∀ a b c. (a -> b -> c) -> (b -> a -> c)
flip f = \x y -> f y x ❶
```

❶ We return a Lambda Function.

Remember, when we talked about Lambda Functions, we talked about moving Parameters across the equals sign? Well, here's a case where we've done exactly that.

By the way, it turns out that our first version of `flip` is idiomatic PureScript. But it's good to see other ways of writing and thinking about `flip`.

Just for fun, here's another way that's also valid:

```
flip :: ∀ a b c. (a -> b -> c) -> b -> (a -> c) ❶ ❷
flip f x = \y -> f y x
```

❶ The redundant Parentheses are specified differently here.

❷ Takes a Function and a Parameter of Type b and returns a Function of 1 Parameter.

By the way, all 3 versions are equivalent:

```
flip f = \x y -> f y x
flip f x = \y -> f y x
flip f x y = f y x
```

As are the Type Signatures:

```
flip :: ∀ a b c. (a -> b -> c) -> (b -> a -> c)
flip :: ∀ a b c. (a -> b -> c) -> b -> (a -> c)
flip :: ∀ a b c. (a -> b -> c) -> b -> a -> c
```

If this is still confusing, go back and read the section on Lambda Functions.

5.5. Writing const

Before reading on, look up const on Pursuit and see if you can figure out what it does and how it might be used.

const is used when you're calling a Function that needs a Function to perform an operation but that operation never varies. Instead, it's always returns a ***constant*** value.

Before you write const, remember to remove it from the Prelude import:

```
import Prelude (Unit, show) ❶
```

❶ Remove const from the list of Functions imported from Prelude since we're writing our own.

So now, you're ready to write const.

5.5.1. Hint for const

Look at const's Type Signature:

```
const :: ∀ a b. a -> b -> a
```

We can see that the first Parameter, a, is returned and the second Parameter b is ignored.

That should inform us on how to write this. So give it a try.

5.5.2. Code for const

```
const :: ∀ a b. a -> b -> a
const x _ = x ❶
```

❶ We don't care about the second Parameter, hence the _, and we just return the first Parameter.

If it's not obvious how this is useful, maybe a simple example will help:

```
filter (\n -> n > 2) [1, 2, 3]  -- [3] ❶
filter (const true) [1, 2, 3]   -- [1, 2, 3] ❷
```

❶ The Predicate to `filter`, which is a Lambda Function, takes a single Parameter that is `n` and returns a `Boolean`, which tells `filter` whether to keep the element in the resulting `Array`.

❷ Here `const true` is the Predicate, which when passed an element from the `Array` will ignore it (since it's the second Parameter to `const`) and just return `true` (the first Parameter to `const`).

Notice how `const true` is a Partial Application of the `const` Function, i.e. we haven't quite given it all of its Parameters. By partially applying, we are effectively "baking" in the constant value that we want this Function to return. Remember that `const` will ignore its second Parameter and always return its first.



Think of `const` as a ***Constant Function*** not a constant value. `const true` is a Constant Function whereas `true` is a Constant Value.

Your module should now look like this:

```
module Ch5 where

import Prelude (Unit, show)

import Effect (Effect)
import Effect.Console (log)

flip :: ∀ a b c. (a -> b -> c) -> b -> a -> c
flip f x y = f y x

const :: ∀ a b. a -> b -> a
const x _ = x

test :: Effect Unit
test = do
  log (show (flip const 1 2))
```

And when run should still print 2, assuming we wrote `const` correctly since now our test code is using our `const` and not `Prelude's`.

5.6. The Apply Operator ($)

There's a really helpful Operator that is often used in idiomatic PureScript, `$`. While this Operator is very easy to implement (we'll do so shortly), it's very powerful in reducing coding noise by eliminating Parentheses.

Let's look at our `test` Function:

```
test :: Effect Unit
test = do
  log (show (flip const 1 2)) ❶
```

❶ It would be nice to reduce these Parentheses.

With the Apply Operator we can remove all Parentheses from that line of code:

```
test :: Effect Unit
test = do
  log $ show $ flip const 1 2 ❶
```

❶ Equivalent statement to the one with Parentheses.

We can think of `$` as adding a Begin Parenthesis at the point where the `$` is and an End Parenthesis at the

end of the line of code even if it's continued on the next line.

So how does this magical Operator work?

First, we need to write its underlying Function, `apply`.

5.7. Writing `apply`

Try to write `apply`, the named version of the infixed Binary Operator, `$`. Don't forget to look up `apply` on `Pursuit` to get its Type Signature. When you do you, you'll probably see `Apply` as the first in the Search Results.

That's the Type `Apply` not the Function `apply`. Remember Type names start with an UPPERCASE letter. Keep looking through the results until you come upon the Function in package `purescript-prelude`.

When you do, you may notice that there are 2 `apply` Functions in that package, one in module `Control.Apply` and the other in `Data.Function`. Since this is dealing with Function calls, we want the one from `Data.Function`.

An alternative search could be made for `$`, which is only defined once in `purescript-prelude`. It shows us that it's in `Data.Function`. And when we click on the `($)` link we'll see the first line of documentation on the Operator:

```
Operator alias for Data.Function.apply (right-associative / precedence 0)
```

This tells us that it's an alias for `Data.Function.apply`, which tells us that we're interested in the `apply` Function from the module `Data.Function`. And if you scroll up just a little on that page, you'll see that the documentation for `apply` is directly above.

Now, try to write `apply` before reading on.

Once you see it's Type Signature, you might think that it's a pretty useless Function:

```
apply :: ∀ a b. (a -> b) -> a -> b
```

This seems an especially silly Function when you include the implied Parentheses:

```
apply :: ∀ a b. (a -> b) -> (a -> b)
```

It appears that `apply` takes a Function and then just returns that Function. What's the point?

Well the benefit isn't in the Function. The real benefit of this Function is found in the use of its Operator, `$` and its Precedence, as we shall soon see.

For now, let's write the Function:

```
apply :: ∀ a b. (a -> b) -> a -> b
apply f x = f x
```

Well, that was pretty trivial.

Next, try to write the Operator.

5.7.1. Hint for ($)

Most likely, when you read the last sentence of the previous section, you were stumped. I know I would be. Even though we've talked about operators and we've looked at how they were defined, we've never ***created*** our own operators.

This is why doing exercises is so important. It forces you to really understand things.

So here's a quick refresher on how to create an Operator:

1. Decide what the associativity is going to be for your infixed Operator and use one of the following accordingly:
    1. `infixl` (Left-Associative).
    2. `infixr` (Right-Associative).
    3. `infix` (Non-Associative).
2. Decide the precedence (Higher gets done first).

Here's an example of a fictitious Operator `!!!` that's Right-Associative and has a Precedence of `3`:

```
infixr 3 notReal as !!!
```

Now see if you can write the Operator for `apply` that's Right-Associative with Precedence 0. (These choices will become apparent soon enough.)

5.7.2. Code for ($)

Here is the code for `apply` and its Operator:

```
apply :: ∀ a b. (a -> b) -> a -> b
apply f x = f x

infixr 0 apply as $
```

You may be wondering how we knew what to choose for Associativity and Precedence. For this Operator, it's a very simple process, which is not true for all Operators.

If we think about the Precedence of Parentheses, it would be 9 because it has the highest Precedence and therefore gets evaluated first.

Let's return to our code with Parentheses:

```
test :: Effect Unit
test = do
  log (show (flip const 1 2))
```

We are saying that we want everything in the Parentheses to happen first. This implies that we want the result of the Parenthetical to be applied to `log` ***last***.

So if we're going to represent that application with our new Operator, `$`, we want it's Precedence to be ***last*** this way ***everything*** to the right of the `$` gets evaluated ***first*** as if it were surrounded by Parentheses.

That's why it has the Precedence of 0 and why we can use it in place of Parentheses.

When considering the Associativity for this Operator, we need to remember how the Operator was defined. We said that it would be the equivalent of adding a Begin Parenthesis where the `$` is specified and an End Parenthesis at the end of the line.

That tells us that repeated use of the Operator on the SAME line of code should work like its Parenthetical Equivalent, which has Parentheses collecting at the end of the line:

```
log $ show $ flip const 1 2   -- log (show (flip const 1 2))
```

Right-Associativity associates to the right, which means that the Parentheses collect on the Right. And that's exactly what we have here, which is why we picked `infixr`.

Now change your test code to use `$` and rerun it. It should still print `2`.

5.8. Writing the Apply Flipped Operator (#)

There's another very helpful Operator in PureScript, `#`. But it's a little more difficult to understand why, so let's look at an example:

```
f x =
  step3 (step2 (step1 x))
```

Here we have a Function that performs 3 Functions, `step1`, `step2` and `step3` in that order. But the order in

which we write these, from left to right, feels inside-out when compared to the order of execution. The inside Parenthetical gets executed first, then the next Parenthetical and finally `step3` is called with the result.

So let's rewrite it using our newly minted Operator, `$`:

```
f x =
  step3 $ step2 $ step1 x
```

It's not inside-out anymore, but since we read code from left to right, it feels backwards. `step1` is called then `step2` then `step3`. Execution is right to left.

Many times, we like to think about things as a sequence of operations. Written out in English, this would be "Take `x` and apply it to `step1` and then take that result and apply it to `step2` and then finally apply that result to `step3` ".

It's just a different way of thinking and it would be nice if we could code that way when it makes the most sense.

Well, we can with the flipped version of the `$` Operator, viz. `#`:

```
f x = x
  # step1
  # step2
  # step3
```

This code reads like our English sentence above does. And since we read code from top to bottom, the steps are in the order they're applied.

Now, before continuing, write `applyFlipped` and its Operator, `#`.

5.8.1. Code for `applyFlipped` and `(#)`

Looking first at the signature:

```
applyFlipped :: ∀ a b. a -> (a -> b) -> b
```

we can see that it lives up to its name. It's Type Signature is just a flipped version of `apply's` Type Signature, which means we can use `flip` on `apply` to write this Function:

```
applyFlipped :: ∀ a b. a -> (a -> b) -> b
applyFlipped = flip apply
```

If this implementation isn't as clear enough, take a look at the manually written version:

```
applyFlipped :: ∀ a b. a -> (a -> b) -> b
applyFlipped x f = f x
```

Next, we turn our attention to the Operator Precedence and Associativity.

Let's look back at our example usage of #:

```
f x = x
  # step1
  # step2
  # step3
```

Since we want this to apply x to step1 first then step2, etc., we should add the Parentheses to reflect that fact:

```
f x = (((x ❶
  # step1)
  # step2)
  # step3)
```

❶ Notice how Parentheses collect on the left.

This informs us that we want to use infixl for Left-Associativity.

To determine Precedence, let's look once again at our example usage of #:

```
f x = x ❶
  # step1
  # step2
  # step3
```

❶ Imagine for a minute that our x is a more complex computation, e.g. x + 10.

If we were to replace x with x + 10, we'd want that computation to happen ***before*** we apply it to step1:

```
f' x = (x + 10) ❶
  # step1
  # step2
  # step3
```

❶ The Parentheses will guarantee that `x + 10` happens first.

We don't really want to have to type in the Parentheses if we don't have to. This means that we want the precedence of `#` to be ***last***.

Based on all of this, we choose the following Operator definition:

```
infixl 0 applyFlipped as #
```

And we could live with this definition for a while, but when we use it in the real world, we're going to eventually bump into the following problem:

```
g x = lastStep $ x
  # step1     -- COMPILER ERROR!!
  # step2
  # step3
```

But why?

Here's the compiler error:

```
# :: ∀ a b. a → (a → b) → b

  Cannot parse an expression that uses operators of the same precedence but mixed associativity:

    Ch5.(#) is infixl
    Ch5.($) is infixr

  Use parentheses to resolve this ambiguity.
 PureScript(MixedAssociativityError)

Peek Problem    No quick fixes available
```

The compile is forcing us to add ***explicit*** Parentheses. But the whole point of `$` was to reduce Parentheses and now our definition for `#` is ruining that.

How do we fix this?

The compiler error says the problem is that operators of the ***same*** Precedences with ***mixed*** Associativity are problematic.

That means that there's only 2 things we can do to fix this:

1. Change Associativity so that they match.
2. Change Precedence so that they're not longer the same.

Option 1 is out since it would completely change the Semantics of our Operator:

```
f x = (x
  # (step1     -- COMPILER ERROR!!
  # (step2
  # step3)))
```

And it would no long Type Check since Functions are being passed to other Functions, which is not what we want.

Option 2 is our only real option here.

How should we change the Precedence then?

Well, the compiler is unhappy because they are the exact same Precedence and we want # to evaluated ***last***:

```
g x = lastStep $ x
  # step1     -- COMPILER ERROR!!
  # step2
  # step3
```

But, if we think about this code for a minute, we can see that we really don't want # to be dead last. We just want it last, right before $.

And therein lies our solution:

```
infixl 1 applyFlipped as #
```

We make # have Precedence 1, which means, when used in conjunction with other operators, it will be evaluated last except for $, which will be evaluated dead last.

Our final solution is:

```
applyFlipped :: ∀ a b. a -> (a -> b)  -> b
applyFlipped = flip apply

infixl 1 applyFlipped as #
```

We should test this code by adding the following line to our test Function:

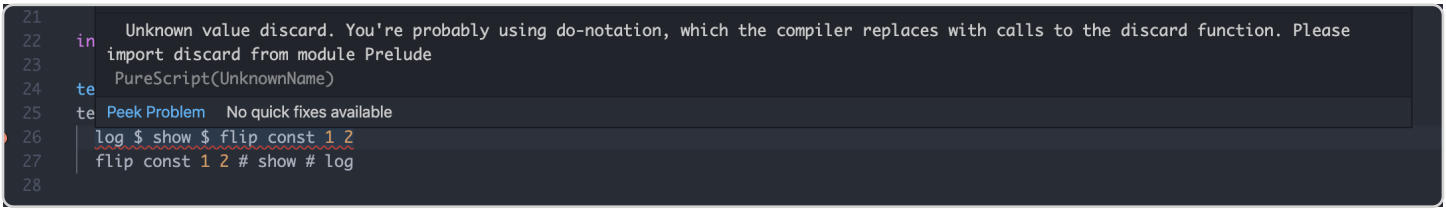
```
flip const 1 2 # show # log
```

But then the compiler complains:

```
test :: Effect Unit
test = do ❶
  log $ show $ flip const 1 2   -- COMPILER ERROR!!
  flip const 1 2 # show # log
```

❶ The do block is special syntax that we'll learn about later in more advanced parts of the book.

This error seems to come out of nowhere and make no sense:



There are some internal Functions that are normally imported by Prelude that are used for do syntax. Since we decided to only import a few items from Prelude in our module, since we're rewriting some of the Prelude Functions, we left out the do syntax Functions.

According to the error message, we can fix this by adding discard to the Prelude import list:

```
import Prelude (Unit, show, discard)
```

And now our program will compile and when it runs, it should print out 2 twice.

5.9. Preparing to Write Data.List Functions

The remainder of the Functions we'll write in this chapter will be from Data.List.

We'll need to selectively import from Data.List so we don't have to write everything.

To include the List Functionality add the following to your imports ***keeping the import List alphabetized (except for Prelude which is first)***:

```
import Data.List (List(..), (:)) ❶ ❷
```

❶ List(..) is a shorthand for saying to import the Type List and all of it's Data Constructors. Longhand for this is List(Nil, Cons).

❷ (:) imports the Cons Operator.

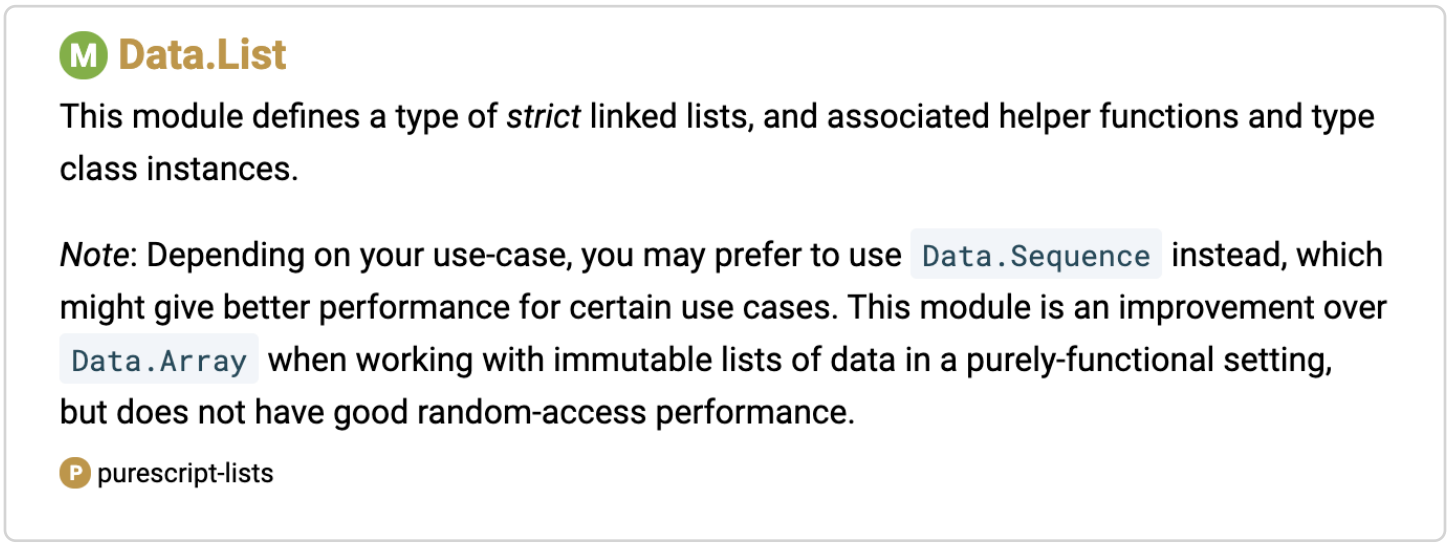
Your imports should now look like:

```
import Prelude (Unit, show, discard)

import Data.List (List(..), (:)) ❶
import Effect (Effect)
import Effect.Console (log)
```

❶ This should have a compiler error.

It turns out that we need don't have the `List` package. If you search `Pursuit` for `Data.List` you'll find the package that contains this Type:



At the bottom, we see `purescript-lists` is the package name.

To add packages to our build, we need to edit the `spago.dhall` file. When adding a package, we always leave off the `purescript-` prefix. So that means we need to add `lists` to the `dependencies`:

```
{-
Welcome to a Spago project!
You can edit this file as you like.
-}
{ name = "my-project"
, dependencies =
  [ "console"
  , "effect"
  , "lists" ❶
  , "prelude"
  , "psci-support"
  ]
, packages = ./packages.dhall
, sources = [ "src/**/*.purs", "test/**/*.purs" ]
}
```

❶ This will add the package that has the module we need, `Data.List`.

Once you do this, rebuild your program:

```
npx spago build
```

This will install the `purescript-lists` package and compile it.

You will get a bunch of warnings. These are because of 2 reasons.

First, we aren't using `Data.List` yet and hence the following warning:

```
Warning 9 of 9:

  in module Ch5
  at src/Ch5.purs:5:1 - 5:33 (line 5, column 1 - line 5, column 33)

    The import of Data.List is redundant


  See
https://github.com/purescript/documentation/blob/master/errors/UnusedImport.md
for more information,
  or to contribute content related to this warning.
```

All of the other 8 warnings are because there are things in the PureScript libraries that haven't caught up

with deprecated features that were introduced in version 14 of the compiler.

We can safely ignore those warnings.

Now return to VSCode and save your file, which will trigger the build. If the compiler error doesn't go away on the `import Data.List` and it still thinks that the module cannot be found, you'll have exit the editor and go back in. This happens from time to time.

It should be underlined with yellow for warning. This warning is the same one from above. You can see it if you hover over the offending line of code.

5.10. Why `Data.List` and not `Data.Array`

The `List` type is written totally in PureScript, whereas `Array` leverages the low-level Javascript Functions for `Arrays`. So for learning how to write PureScript, and especially, recursive Functions in PureScript, `Lists` are the perfect study subject.

Another limitation with `Array` is that Pattern Matching only works if the pattern is the ***exact*** size of the `Array`. This makes writing a simple Function like `length` on Array impossible in PureScript.

Keep in mind that most, if not all, of the Functions we'll write for `List` will have an `Array` equivalent and when writing production code, we'll use `Array` far more often than `List`.

5.11. Writing `singleton`

When you search `Pursuit` for the next Function, `singleton`, you'll see a bunch of Functions listed. One is from `Data.Array`. Another from `Data.List.Lazy`. The one we're concerned with is the one from `Data.List` with the Type Signature:

```
singleton :: ∀ a. a -> List a
```

We know it's the one we want because the `List` Type is in the Signature.

As a reminder, here is the definition of `List` with its `Cons` Operator, `:`.

```
data List a = Nil | Cons a (List a) ❶

infixr 6 Cons as :
```

❶ `Cons` takes a Value and effectively puts that Value at the head of another `List`. This is a Recursive Definition and is only terminated by `Nil`.

You can construct a `List` with its constructor, `Cons`:

```
Cons 1 (Cons 2 (Cons 3 Nil))
```

You can also use the `Cons` Binary Operator:

```
1 : 2 : 3 : Nil
```

Now add the following line to `test`:

```
log $ show $ singleton "xyz" ❶
```

❶ Should print out `("xyz" : Nil)`.

Now, write the `singleton` Function without reading ahead.

5.11.1. Code for `singleton`

This Function is quite simple:

```
singleton :: ∀ a. a -> List a
singleton x = x : Nil ❶
```

❶ Uses the `Cons` Operator.

It could also be written using `Cons`:

```
singleton :: ∀ a. a -> List a
singleton x = Cons x Nil ❶
```

❶ This is merely the prefixed version, whereas the Operator version is the infixed version.

I prefer the Operator version due to its terseness.

5.12. Writing `null`

If you come from a language that has NULLs, I want you to realized that this is ***not*** the same thing. This Function checks to see if a `List` is empty.

Although, you can write this Function with an `if` or a `case`, it's idiomatic to use Pattern Matching instead. So try to take that approach.

Also, test your code with the following lines added to `test`:

```
log $ show $ null Nil ❶
log $ show $ null ("abc" : Nil) ❷
```

❶ Should print out `true`.

❷ Should print out `false`.

Try to code it before proceeding.

5.12.1. Code for `null`

Can you tell what kind of Function `null` is by just looking at it's signature?

```
null :: ∀ a. List a -> Boolean
```

Any Function whose signature is of the form `a -> Boolean` is a Predicate. In this case, the `a` in our Predicate's Type Signature stands for `List a` in `null's` Type Signature.

And now the code for `null` using Pattern Matching:

```
null :: ∀ a. List a -> Boolean
null Nil = true ❶
null _ = false ❷
```

❶ Empty `List` case.

❷ The catchall case.

Notice that we didn't use an `if` or a `case` expression. Pattern Matching is so much cleaner.

5.13. Writing `snoc`

The name of this Function is `Cons` reversed. Since `Cons` puts an element at the head of a `List`, what do you think `snoc` does?

It puts an element at the end. Think about how you want to attack this problem before coding it.

Also, test your code with the following line added to `test`:

```
log $ show $ snoc (1 : 2 : Nil) 3 ❶
```

❶ Should print `(1 : 2 : 3 : Nil)`.

5.13.1. Hint for snoc

Did you tried to write something like the following:

```
snoc :: ∀ a. List a -> a -> List a
snoc l x = l : x    -- COMPILER ERROR!!
```

The reason this won't work is because `l` is the list and `x` is the element. That's not what the `Cons` Operator wants. It expects to put an element at the head of the `List`, not the end.

Remember, `Lists` are effectively `linked-lists`. Adding to the head of a `List` is easy and fast. Adding to the end of the `List` requires the following Strategy:

1. Unwinding the `List` until you get to the end. This will require recursion.
2. At the end of the `List`, return a `List` with just one element in it, i.e. the element that we're adding to the end.
3. Put the `List` back together, one element at a time, as we return from the recursive calls.

Hopefully, this has helped. Try to write the Function again using this Strategy.

5.13.2. Code for snoc

Looking at our Strategy, let's implement the `Base Case` (#2) of the Recursion first:

```
snoc :: ∀ a. List a -> a -> List a
snoc Nil x = singleton x ❶ ❷
```

❶ This is our `Base Case` from our Strategy (#2). We've reached the end of the `List` (or started with an empty `List`) and we simply return a `singleton` list with the element that we want at the end of the `List`.

❷ We're not done yet. This Function is a `partial` Function and the compiler will complain because not all cases have been exhausted.



The `Base Case` of a recursive Function is the case that terminates the recursive calls. If a recursive Function doesn't have a `Base Case` then it's infinitely recursive and will never produce an output value.

Next step is to figure out what the next Pattern Match should be. Looking at the Function as it is so far, we can see that we've handled the empty `List` case, so we need a Pattern Match for a non-empty `List`.

Let's add just that part:

```
snoc :: ∀ a. List a -> a -> List a
snoc Nil x      = singleton x
snoc (y : ys) x = ??? ❶
```

❶ This will match non-empty lists with the Pattern `(y : ys)`.

The next question we need to ask ourselves is whether there are any other cases. We have the empty `List` case and the non-empty `List` case. Since that covers all of the cases, we now have a total Function.

Now all we have to do is figure out what goes in place of `???`.

Revisiting our Strategy #1, it says:

```
Unwinding the `List` until you get to the end. This will require recursion.
```

Unwinding the list is essentially what we're doing by pulling the head off of the `tail` in the non-empty `List` case:

```
snoc :: ∀ a. List a -> a -> List a
snoc Nil x      = singleton x
snoc (y : ys) x = ??? ❶
```

❶ Pulling the head off.

But we still need to recurse to continue down the `List` until we get to the end of the `List`:

```
snoc :: ∀ a. List a -> a -> List a
snoc Nil x      = singleton x
snoc (y : ys) x = snoc ys x ❶
```

❶ Recurse so we get to the end of the `List`. We use the tail of the `List ys`, which will get progressively smaller and smaller with each recursive call to `snoc` until it's empty, which will terminate the recursion.

If you run your program with this code, you'll see that the last thing printed assuming you're adding your tests to the end of `test` Function:

```
(3 : Nil) ❶
```

❶ NOT the right answer!!

What did we miss?

We didn't reassemble the list , i.e. #3 in our Strategy. Let's examine that case to figure out what's wrong:

```
snoc (y : ys) x = snoc ys x
```

Notice that we haven't done anything with `y`. We pull off the head and then we don't use it. So we're effectively throwing it away.

Once we are done adding `x` to `ys` via `snoc ys x`, we should put `y` back onto the head of the `List`:

```
snoc :: ∀ a. List a -> a -> List a
snoc Nil x      = singleton x
snoc (y : ys) x = y : snoc ys x ❶
```

❶ We put `y` at the head of the `List`.

Now when we run our program, we get:

```
(1 : 2 : 3 : Nil)
```

This works!

But let's make sure we understand how it works. Typically, the recursive part is always the most difficult:

```
snoc (y : ys) x = y : snoc ys x
```

We can reason about this line of code as follows:

```
We pull the head off of the List and then add the element, `x`, to the tail.
Then we put the head back onto that new List.
```

Here's the full implementation again:

```
snoc :: ∀ a. List a -> a -> List a
snoc Nil x      = singleton x ❶
snoc (y : ys) x = y : snoc ys x ❷
```

Next, let's look at how each call works by starting with the initial call and using simple Mathematical Substitution with the two different equations annotated above:

```
snoc (1 : 2 : Nil) 3  =     -- original call
1 : snoc (2 : Nil) 3  =     -- substitute using ❷
1 : (2 : snoc Nil 3)  =     -- substitute using ❷
1 : (2 : singleton 3) =     -- substitute using ❶
1 : (2 : (3 : Nil))   =     -- singleton definition
1 : 2 : 3 : Nil             -- remove redundant Parentheses since
                            -- `:` is Right-Associative
```

Notice that each time we do a Mathematical Substitution, it's equivalent to a Recursive call to `snoc` in our program. We substitute 3 times and we recurse 3 times.

5.14. Writing `length`

The next Function should be easier after working on `snoc`. It's the `length` Function to calculate the length of a `List`.

Make sure you're looking at the right version of `length` from `Data.List` when you search `Pursuit`.

Add this line to your `test` Function for testing `length`:

```
log $ show $ length $ 1 : 2 : 3 : Nil ❶
```

❶ Prints 3.

Also, at the risk of spoiling the solution, add the `(+)` Operator to the `Prelude` import list:

```
import Prelude (Unit, (+), show, discard)
```

Now, try to write `length` before moving forward.

5.14.1. Hint for `length`

Imagine I tell you that I have a `List` that you must determine the length of, but you're not allowed to look at the `List`. And I also tell you that you can ask me one question about that `List` but not about it's length.

What question might you ask?

You might think about it for a minute, and pull a dirty trick on me by asking what's the length of the tail of the `List`.

Why is this a dirty trick? Well, you didn't ask about the length of the original `List`, but instead asked about the length of the tail.

Why is this a cheat? Well, all you have to do is add 1 to the answer to that question and you know how long the original `List` is.

To reiterate, the length of a `List` is 1 plus the length of its tail.

Use this to help you write `length` before reading on.

5.14.2. Code for `length`

As usual, we start with the Type Signature:

```
length :: ∀ a. List a -> Int
```

Then we write the trivial case, i.e. the `Base Case` of a Recursive Function:

```
length :: ∀ a. List a -> Int
length Nil = 0 ❶
```

❶ An empty list has length `0`.

Finally, we write the general or Recursive case, i.e. a non-empty `List`:

```
length :: ∀ a. List a -> Int
length Nil       = 0
length (x : xs) = ???
```

But how do we code the `???` part?

If we know the length of `xs` then we only have to add 1 to that and we have the length of our `List`:

```
length :: ∀ a. List a -> Int
length Nil       = 0
length (_ : xs) = 1 + length xs ❶
```

❶ Using `_` where `x` was in the previous step since we're not using it. The compiler will produce a warning if we don't use Variables.

And what does this look like when we use Algebraic Substitution:

```
length (1 : 2 : 3 : Nil)    -- original call
1 + length (2 : 3 : Nil)    -- substitute using ❷
1 + (1 + length (3 : Nil))  -- substitute using ❷
1 + (1 + (1 + length Nil))  -- substitute using ❷
1 + (1 + (1 + 0))           -- substitute using ❶
1 + (1 + 1)                 -- reduce
1 + 2                       -- reduce
3                           -- reduce
```

We have 4 substitutions, which means we have 4 recursive calls. The longer the `List`, the more recursive calls we have.

5.15. Tail Recursion

The `length` Function is great example to use to learn about `Tail Recursion`.

Before we delve into that subject, we need to talk about how most computers work when calling Functions at the lowest level. When a Function is called, its code must be executed and when the Function is finished, the program needs to return to the location that called it.

The way it knows how to return to the calling Function is by implementing a Data Structure called a ***Stack***.

The Stack keeps a full record of the calling process. Just like a stack of plates, the last plate on is the first plate off of the stack. That's why a Stack is called a ***LIFO*** Data Structure or ***Last In First Out***.

Here's an example:

```
f :: Int -> Int
f x = g x ❶

g :: Int -> Int
g x = h x ❷

h :: Int -> Int
h x = x + 10 ❸
```

❶ `f` calls `g`.

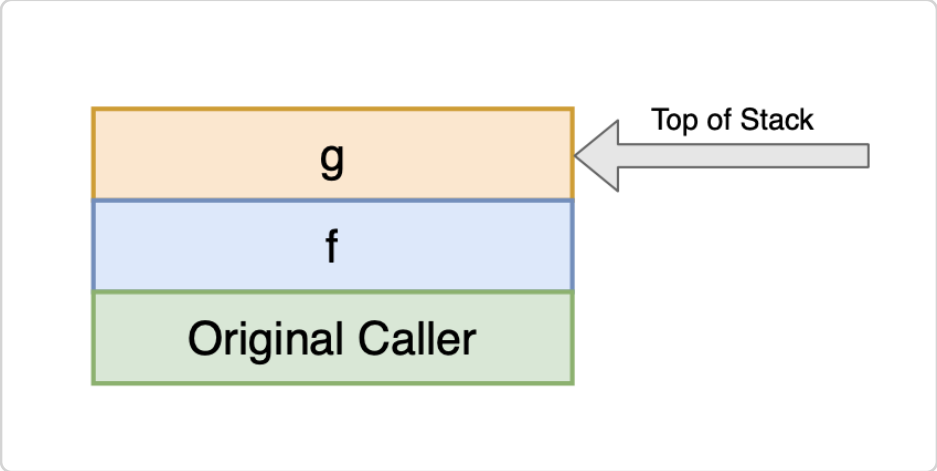
❷ `g` calls `h`.

❸ `h` returns the computation.

Next, let's look at the low-level process of calling `f` with `10`:

```
f 10      -- original caller calls f
          -- push return address of original caller onto the Stack
g 10      -- f calls g
          -- push return address of f onto Stack
h 10      -- g calls h
          -- push return address
```

At this point, the Stack looks like:



Looking at the Stack, we will return to `g`, then `f` then the `original caller` in that order. This ordering should make sense if you think of a stack of plates.

So far, we've only looked at the execution right up to the point where we've called `h`. Let's continue executing our code at that point:

```
h returns 20  -- pops the top of the Stack (getting g)
              -- returns 20 to g
g returns 20  -- pops the top of the Stack (getting f)
              -- returns 20 to f
f returns 20  -- pops the top of the Stack (getting original caller)
              -- returns 20 to the original caller
```

That was a simple case where we only needed 3 slots on the Stack.

Let's look back at our `length` Function:

```
length :: ∀ a. List a -> Int
length Nil       = 0
length (_ : xs) = 1 + length xs ❶
```

❶ Recursive Function call.

We can see that it calls itself once for each element of the `List`. If the `List` has `2` elements, then we'll make `2` calls and require `2` slots of the Stack.

If we have a `List` of `1000` elements, we'll need `1000` slots in the Stack. A million elements will need a million slots.

At some point, we will encounter a `List` long enough that it uses up all of the available slots in the Stack. This is called a ***Stack Overflow***. You may have heard of a website named after this unfortunate event.

The problem with our `length` Function is that it's Recursive but not ***Tail Recursive***.



A Function is Tail Recursive if the very last operation of the Function is a Recursive call.

When a Function is Tail Recursive, the compiler can be smart and not save the return location on the Stack since there's no reason to return to the calling Function that has nothing left to do.

This saves our Stack from overflowing.

Here's `length` again:

```
length :: ∀ a. List a -> Int
length Nil       = 0
length (_ : xs) = 1 + length xs ❶
```

❶ `length` appears to be Tail Recursive since the `length` call is the rightmost thing we've typed but it's not the last thing executed. The last thing that's executed in this case is the addition of `1` to the result of the Recursive call to `length`. That addition is why this Function is ***not*** Tail Recursive.

So can we rewrite `length` so that it is Tail Recursive?

I'm sure you've guessed that it is possible, otherwise why would I have waited until now to bring up the subject.

To rewrite `length` to be Tail Recursive, we must do all of our calculations ***before*** we recurse so that there's nothing left for us to do.

But how?

To do the calculations first, in the case of `length`, means that we're going to have add the `1` to a running length Value and then pass it to each recursive call.

Let's write `length` where that is the case:

```
length :: ∀ a. Int -> List a -> Int ❶
length acc Nil      = acc ❷
length acc (x : xs) = length (acc + 1) xs ❸
```

❶ Added `Int` to the Type Signature for the running length Value.

❷ `acc` is our accumulator for our running length Value. So when we hit the end of the list, we can return `acc`, which will be the final length Value.

❸ We are now Tail Recursive because we're adding `1` ***before*** we recurse and there's no reason to return from the recursive call because there's nothing left to do.

So we can congratulate ourselves for writing this Function to be Tail Recursive and all is right with the world, right?

Wrong.

Our new version burdens the caller of `length` with having to pass in the `acc` Parameter, which must be `0` to start otherwise our length calculation will be wrong:

```
list = 1 : 2 : 3 : 4 : Nil
len = length 0 list ❶ ❷
badLen = length 7 list ❶ ❸
```

❶ We have to pass the initial length value which ***should be*** `0`.

❷ This call will produce a good length value.

❸ This call will produce a bad length value since it'll be off by `7`.

This is bad. So how do we have our cake and eat it?

The way to solve this dilemma is to write the `length` Function with the original Type Signature, i.e. no `acc` Parameter. And then we immediately call a private Tail Recursive version of the `length` Function, which we'll call `go` with the `acc` Parameter set to `0`:

```
length :: ∀ a. List a -> Int
length l = go 0 l where  ❶ ❷ ❸
  go :: Int -> List a -> Int ❹
  go acc Nil      = acc
  go acc (_ : xs) = go (acc + 1) xs ❺
```

❶ `length` makes sure to pass in `0` to start. We only have to get this code right once, not every time `length` is called like before.

❷ It's idiomatic to call our private Function in this situation `go`. I'm not sure the history behind this, but this tradition started in Haskell.

❸ `go` is private because we are declaring it with the keyword `where`.

❹ This is our Tail Recursive version of `length` renamed to `go` for historical reasons.

❺ Here's our Tail Recursive call.

Now the callers of `length` don't have worry about the Tail Recursion requirements. `length` manages that by making sure that `0` is the starting value for the running length.

5.15.1. Observations regarding Tail Recursion

Tail Recursion is great for saving Stack space, but it's more difficult to write, reason about and makes the codebase more complex. To write all of your Recursive Functions this way to start is known as ***Premature Optimization***. We only want to optimize our code when we have to.

It turns out that not all Recursive Functions can easily be written to be Tail Recursive and even when you can, the complexity of the Function can make it unmaintainable. And there are other cases where you save Stack space but wind up using Heap space instead. Sometimes this can be an okay solution since we usually have more Heap than Stack.



The Heap is a memory region that is available to your program. Variables are allocated from the Heap as your program runs. Usually, the Heap is much larger than the Stack.

So if we can easily write something to be Tail Recursive, it may be worth doing so. But, it's best not to prematurely optimize our code, but instead wait until we need it. It's important to know that such a thing as Tail Recursion exists and that compilers such as PureScript (and Haskell) are smart enough to take advantage of it.

But that doesn't mean we should burden ourselves with the added complexity for the 1% case.

5.16. Writing head

The `head` Function should be relatively easy after `length`. Once again, make sure when you search `Pursuit`, that you're looking at the `head` from `Data.List`.

Add the following to `test`:

```
log $ show $ head Nil ❶
log $ show $ head ("abc" : "123" : Nil) ❷
```

❶ Test the outlier case of an empty `List`. Prints `Nothing`.

❷ Test the general case. Prints `(Just 2)`.

There are a couple of compiler issues you're going to run into along the way. Try and tackle them yourself and if you get stuck, refer to the following Hint section.

5.16.1. Hint for head

If you haven't finished writing head yet, go ahead and comment out what you did write and add the following stubbed out version of head:

```
head :: ∀ a. List a -> Maybe a
head _ = Nothing
```

This function is just so you can follow along.

If you have a completed Function for head, then you probably noticed that the compiler is complaining that it didn't know what Maybe was. That's because you need to import it from Data.Maybe.

So, if you haven't tried that, please do so now before reading on.

If you simply added the following:

```
import Data.Maybe (Maybe)
```

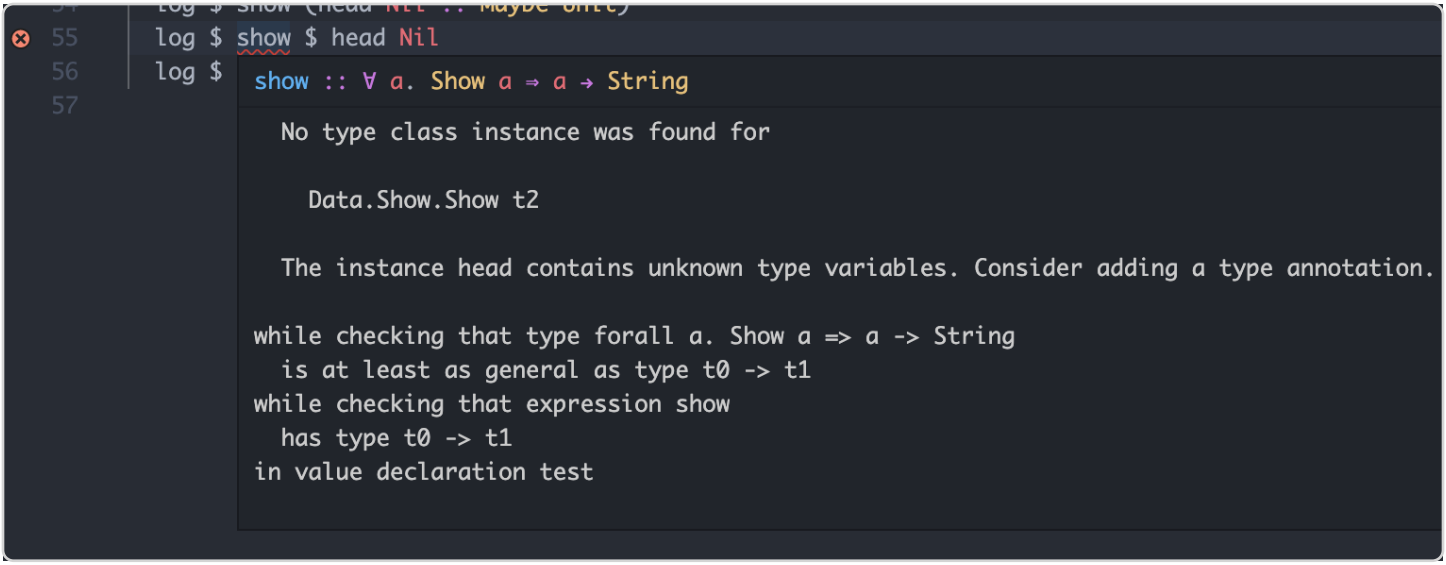
but you're now you're getting an error on Nothing and/or Just, it's because the above import only imports the Type Maybe. It doesn't import the Data Constructors, Nothing and Just.

Remember, the only way we can construct something of Type Maybe is to use its Data Constructors.

So we need to change our import statement to:

```
import Data.Maybe (Maybe(..))
```

That should fix all of your problems with Maybe, et al. But, now the compiler has a problem with the test code I gave you. You're probably getting an error like:



And if you're like me, the first time you see this, all you can do is scratch your head and wonder. There's a lot of information here and you can get overloaded with all of it.

The most import part for our situation is the following:

```
No type class instance was found for Data.Show.Show t2
```

The real key here is `t2`. That tells us that the Type Inference engine has failed to figure out a Concrete Type. If it had, it would have been something like `Int`, `Boolean` or some uppercased Type name.

But it's lowercased meaning that it's still a Type Parameter that the Type Inference couldn't figure out based on how you're using it. Why though?

To answer that, we need to look at the code in question:

```
log $ show $ head Nil
```

Now, imagine we are doing the Type Inference instead of the compiler, and we're trying to figure out the Types in this line of code. First, we know the Types for `log`, `show` and `head` because those have been explicitly typed, e.g. here's `head`:

```
head :: ∀ a. List a -> Maybe a
```

But what about `Nil`? What's its Type?

Let's refer back to `List's` definition to see if we can ferret out it's Type:

```
data List a = Nil | Cons a (List a)
```

If we use the Data Constructor, `Nil`, we will construct a Type, `List a`. But what Type is `a`?

There is no way to tell.

If we look at the other test code line I gave you, we can determine `List's` Type easily:

```
log $ show $ head ("abc" : "123" : Nil)
```

It's Type is `List String`.

But with an empty `List`, there are no elements to interrogate. It could be a `List` of ***anything***, viz. `List a`, where `a` is our ***anything***. The problem is that the compiler wants to verify that `a` can be passed to `show` (more on that later).

How do we fix this then?

Type Inference infers what Types our values are by how we use them. But when there isn't enough evidence, as in the case of our empty `List`, then we must ***explicitly*** specify the Type.

Here's how to properly do that:

```
log $ show (head Nil :: Maybe Unit)
```

We parenthesize the call to `head` and specify an ***explicit*** Type for its return value. But why `Unit` as the `List` element Type?

It really doesn't matter. It just needs any valid Type that we can `show` (more on that later). By using `Unit`, we are signaling to anyone reading our code that the element Type really doesn't matter. Had we used `Int` or some other non-trivial Type, the reader of our code (which could very well be us some time in the future), may think there's more to that list than meets the eye.

Returning to the error message for a minute:

```
No type class instance was found for Data.Show.Show t2
```

It's worth noting that during Type Inference, the compiler will "lose" your names and create any unknown type names with a `t` and a number. Here it has come up with `t2` for our `a`.

It can be very difficult from time to time to do that mapping between the type the compiler is referring to and what we called it. This is an understandable issue, since it may be referring to multiple Function, like, `log`, `show` and `head` which all call their Type Parameters `a`.

Hopefully, in the future, the PureScript compiler will attempt to match the name to the original, e.g. instead of `t2`, it would print `a2` which would at least give us some clue to look for some `a`.

Another thing to note is that when I explicitly specified the Type, I added Parentheses to make sure that I specified the Type that `head` would return. With no Parentheses, the explicit Type Signature refers to the whole line of code:

```
log $ show $ head Nil :: Maybe Unit     -- COMPILER ERROR!!
```

Also, I removed the `$` Operator from after the `show`. I did this because it's completely redundant in the following:

```
log $ show $ (head Nil :: Maybe Unit)   -- `$` is redundant
```

I chose to specify the return Type of `head`, but I could've just as easily specified the `List` Type instead:

```
log $ show $ head (Nil :: List Unit)
```

The choice is really up to you.

Now with all of the compiler errors out of the way, see if you can finish writing `head` before continuing on.

5.16.2. Code for head

Start with the Type Signature:

```
head :: ∀ a. List a -> Maybe a
```

Next, write the trivial case, i.e. empty `List`:

```
head :: ∀ a. List a -> Maybe a
head Nil = Nothing ❶
```

❶ There is no head if the list is empty.

Next, we write the general case by Pattern Matching on the `List`:

```
head :: ∀ a. List a -> Maybe a
head Nil     = Nothing
head (x : _) = Just x ❶
```

❶ Don't forget the `Just`. Also, we don't name the `tail`, i.e. `_`, here because we don't need it.

5.17. Writing `tail`

After writing `head`, you should be able to write `tail` in a flash.

Here is the test code to add to `test`:

```
log $ show $ tail Nil
log $ show $ tail ("abc" : "123" : Nil)
```

Code it before moving on.

5.17.1. Hint for `tail`

Yes, yes, I know. I set you up. I purposely left off the explicit Type Signature from the test code. But, hopefully, if I did a good enough job explaining why it was needed, you were able to quickly figure out what was wrong and knew how to fix it.

If you weren't able to, please reread the section on `head` and pay close attention to the Type Signature to `tail` (it's slightly different than `head`). That should help you figure out what the explicit Type should be. Try again before moving on.

5.17.2. Code for `tail`

Just in case you are still struggling with the explicit Type in the test code:

```
log $ show (tail Nil :: Maybe (List Unit))
```

Unlike `head`, `tail` returns `Maybe (List a)`. And since the Type Inference engine cannot determine what Type `a` is, we need to give it one, viz. `Unit`.

Here's the other way to specify the explicit Type for `a`:

```
log $ show $ tail (Nil :: List Unit)
```

This Function is so close to `head`, we can pretty much jump right into it:

```
tail :: ∀ a. List a -> Maybe (List a)
tail Nil      = Nothing ❶
tail (_ : xs) = Just xs ❷
```

❶ Same idea as in head. There is no tail of a empty List

❷ Here we don't name the head since we're just interested in the tail.

You may have realized that the definition for tail could have been the following:

```
tail :: ∀ a. List a -> List a     -- ALTERNATIVE definition
```

This definition would mean that you'd get an empty tail for an empty List, which may seem like a valid return value. But that really depends on your definition of tail.

Consider the following definition for tail:

```
The tail of a List is what is remaining after the first element, i.e. the
head, is removed.
```

If the List is empty then no head can be removed and therefore there can be no remainder to represent the tail.

This definition matches the Type Signature for tail:

```
tail :: ∀ a. List a -> Maybe (List a)
```

And since head can fail, so can tail therefore tail also returns a Maybe. In Haskell, this definition is not the case. tail throws an exception given an empty list. This is also the case with head in Haskell. It is the rare case, where this is desirable over returning Maybe as PureScript does.

5.18. Writing last

last is just like head, except that the single element removed from the List is at the end.

This process is more difficult since Lists are built by adding elements to the front. That fact makes writing head easier than writing last.

Before you try to code this, devise a strategy for how you're going to attack this problem before moving on to the coding. If you get stuck for an obvious approach, then proceed to the hint.

Here is the code you should add to your test:

```
log $ show $ (last Nil :: Maybe Unit) ❶ ❷
log $ show $ last ("a" : "b" : "c" : Nil) ❸
```

❶ Just like we've done before, we need to let the compiler know what Type a is in the Maybe a return Type

of last.

❷ Prints Nothing.

❸ Prints (Just "c").

By the way, you can also write the last test line as:

```
log $ show $ last $ "a" : "b" : "c" : Nil
```

Since $ has the lowest precedence, the : Operator gets evaluated first. In this particular case, I prefer using Parentheses because it helps highlight that we have a List. Most other cases, I prefer $ over Parentheses.

5.18.1. Hint for last

Since we can only work with the head and tail of a List using Pattern Matching, we must proceed down the list one element at a time, removing and discarding the head along the way until we get to a list with only 1 element left and that is our last.

So we're going to have a Pattern Match case for a List of one element.

If you struggled writing this, try to take this approach in your code before moving on.

5.18.2. Code for last

Start with the Type Signature from Data.List:

```
last :: ∀ a. List a -> Maybe a
```

This Type Signature tells us a lot. The fact that it returns a Maybe tells us that it can fail. That's an important clue.

We should ask ourselves, "When can this fail". After our experience with head and tail, we may guess that it'll fail if it's given an empty List. But does that make sense?

If last is just like head but from the other end of the List, then it seems logical that it fails on the empty List just like head does since you cannot return the last element of an empty List because there is no last element.

Let's add that case:

```
last :: ∀ a. List a -> Maybe a
last Nil = Nothing ❶
```

❶ The empty List is our failure case.

With head and tail, our next step was to consider the general case, so let's try that approach and see if it works for writing last.

Reviewing our strategy from the section Hint for last, we need to discard the head until we're left with a list of only one element. So let's write that:

```
last :: ∀ a. List a -> Maybe a
last Nil      = Nothing
last (_ : xs) = if length xs == 1 then head xs else last xs ❶ ❷ ❸
```

❶ Don't forget to add (==) to the import Prelude import list.

❷ The head is discarded in the Pattern Match by simply not naming it, i.e. using _.

❸ If the length is 1 then we take the one and only element out of xs otherwise we keep recursing until we've exhausted all but the last element. Notice that since head returns a Maybe and we ***know*** that xs is not empty, in fact its length is 1, we'll get a Just with the one and only element and, therefore, do not need to add it.

Just for fun, look at this code and see if it's Tail Recursive? Remember that a recursive Function is Tail Recursive if the last thing it does is Recurse.

It turns out that it is Tail Recursive, but it's hard to see when written with the if. And it turns out that there's a more elegant solution for this case that doesn't use an if but uses Pattern Matching instead.

Can you think of how to do that? Go back and try to use Pattern Matching on a List of one element before proceeding.

The case where length xs == 1, i.e. we have a List of one element, is the same as the following:

```
last (x : Nil) = Just x ❶ ❷
```

❶ The if part as a Pattern Match, i.e. length equals 1.

❷ (x : Nil) does the same thing that head does in the if–then–else version, but we don't get a Maybe here and so must add the Just ourselves.

And with this case added to our code, we don't need an if anymore. Essentially, the Pattern Matching is doing the if for us. So we just have to code the else case:

```
last (_ : xs) = last xs ❶
```

❶ The else part as a Pattern Match.

And putting it all together:

```
last :: ∀ a. List a -> Maybe a
last Nil      = Nothing
last (x : Nil) = Just x ❶
last (_ : xs) = last xs ❷
```

❶ This match MUST precede the general case, otherwise the 1 element `List` will match the general case and we'll never execute this logic. Remember, when using Pattern Matching the compiler is doing the `if` checks for us in the order that we define them.

❷ Tail Recursion here.

Now we can really see that this Function is Tail Recursive since it only Recurses in a single case and all it does is Recurse.

One thing you may not have noticed is that we're no longer calling `length` like we did in our first version. This means that we aren't traversing the `List` multiple times anymore making this version far more efficient for large `Lists` than our previous version.

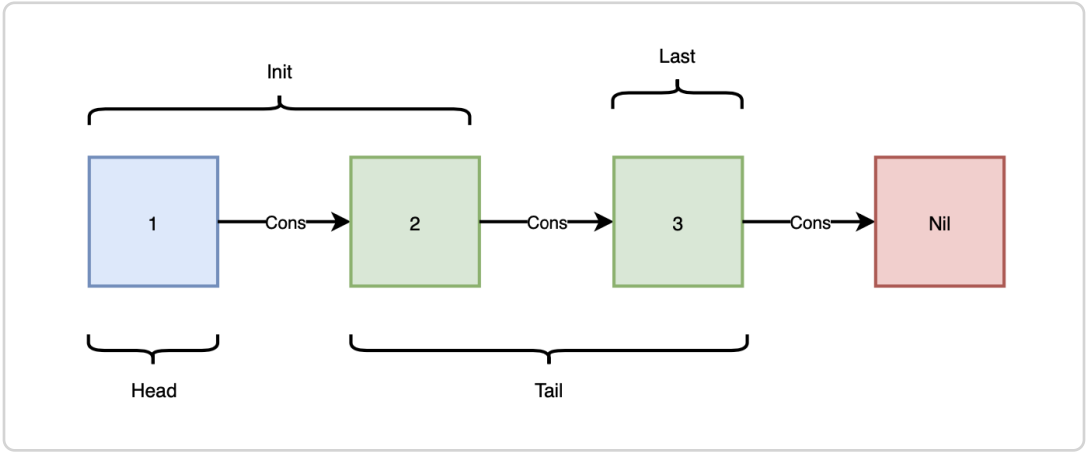
Remember that `length` traverses the list once. The previous version called `length` for EVERY element in the `List`. That means a `List` of size `n` will traverse n^2 times, albeit with a smaller and smaller `List` each iteration.

This improvement is definitely worth doing since it's so much more efficient and easier to read and understand. It's a win-win-win scenario.

5.19. Writing `init`

`init` is just like `tail`, except that the `List` returned is at the beginning minus the `last` element.

You may have figured out by now that `head` is to `tail` as `last` is to `init`.



Here is the code you should add to your `test`:

```
log $ show $ init Nil ❶
log $ show $ init (1 : Nil) ❷
log $ show $ init (1 : 2 : Nil) ❸
log $ show $ init (1 : 2 : 3 : Nil) ❹
```

❶ Prints `Nothing`.

❷ Prints `(Just Nil)`.

❸ Prints `(Just (1 : Nil))`.

❹ Prints `(Just (1 : 2 : Nil))`.

Once again, try to devise a strategy before jumping into the code.

5.19.1. Hint for `init`

As we did in `last`, we want to recurse until we get to a singleton `List`, i.e. a `List` with only one element. But unlike `last`, we do NOT want to dispose of the elements along the way. Instead, we want to collect the elements in a new `List` as we recurse.

Also, think back to the last few Functions that we' wrote. We always asked the question, "What's the failure case".

If you haven't been able to write this yet, try again with these hints in mind.

5.19.2. Code for `init`

As always, we start with the Type Signature:

```
init :: ∀ a. List a -> Maybe (List a)
```

What can we determine by just looking at this and knowing what `init` is supposed to do?

First, we know it's going to take a `List`, which, based on our previous experience with `head`, `tail` and `last`, tells us that we're going to iterate, i.e. our Function will be Recursive. That means we'll need to have a way to terminate the recursion, i.e. a base-case.

Just like in `last`, our base-case is a singleton `List`:

```
init :: ∀ a. List a -> Maybe (List a)
init (x : Nil) = Just Nil ❶ ❷
```

❶ We return a `List` with everything EXCEPT the last element, `x`, which is `Nil`.

❷ Don't forget the `Just`.

Our failure case is the empty List:

```
init :: ∀ a. List a -> Maybe (List a)
init Nil = Nothing ❶
init (x : Nil) = Just Nil
```

❶ If we initially get an empty List, we will fail to produce a List minus its non-existent last element.

Finally, we need to deal with the general case where we have a non-empty List. We want to remove the head of the List and add it to the init of the tail recursively:

```
init :: ∀ a. List a -> Maybe (List a)
init Nil = Nothing
init (x : Nil) = Just Nil
init (x : xs) = x : init xs  -- COMPILER ERROR!! ❶
```

❶ Why is this an error?

The compiler complains with:



Although, this approach worked in the past, it doesn't work here because init doesn't return a List. Instead, it returns a Maybe List. Notice that these 2 Types are the Types in our Type mismatch error.

We'll need a different strategy than we took with the earlier Functions.

Let's wind back our code to just our failure case:

```
init :: ∀ a. List a -> Maybe (List a)
init Nil = Nothing
```

We have handled the case where we have an empty List. All other cases that we need to write will be cases

where we have a non-empty list.

The only reason that `init` returns `Maybe List` is because it can initially get passed an empty `List`. We no longer have that restriction in writing the recursive case.

So let's write a local Function, proverbially named `go`, that returns a `List` since we have guaranteed that it will never get an empty `List`:

```
init :: ∀ a. List a -> Maybe (List a)
init Nil = Nothing
init l = go l where ❶
  ???
```

❶ Here `l` is a non-empty `List` thanks to the previous line of code which catches the empty `List` case and returns `Nothing`.

Now we just have to figure out how to write `go`. Once again, we know we're going to recurse and we're going to eventually reduce the non-empty `List` until it has only 1 element left:

```
init :: ∀ a. List a -> Maybe (List a)
init Nil = Nothing
init l = go l where
  go (_ : Nil) = Nil ❶
  ???
```

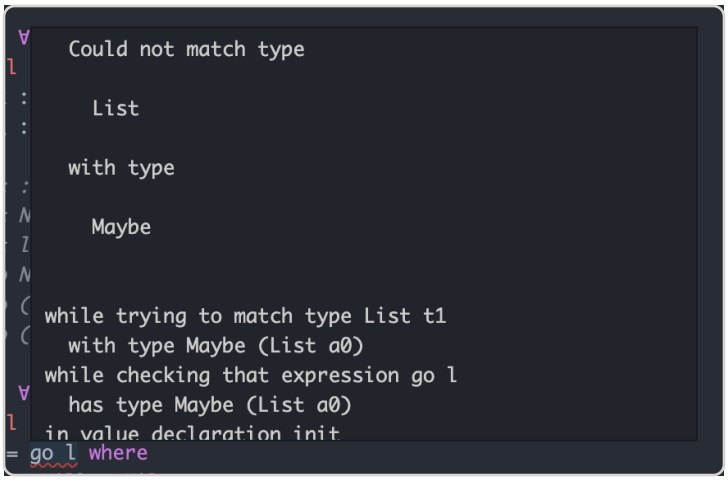
❶ Just like we did before, we exclude the last element of the list, `x`, which is the whole point of `init`. This is our base-case for our recursion.

We've written the base-case and now we must write the recursive case.

```
init :: ∀ a. List a -> Maybe (List a)
init Nil = Nothing
init l = go l where       -- COMPILER ERROR!!
  go (_ : Nil) = Nil
  go (x : xs) = x : go xs ❶
```

❶ This case is exactly what we tried to write earlier with `init`. Now the compiler doesn't complain anymore because `go` returns a `List`, as opposed to `Maybe List` that `init` returns.

But now we have a different compiler complaint at a different location:

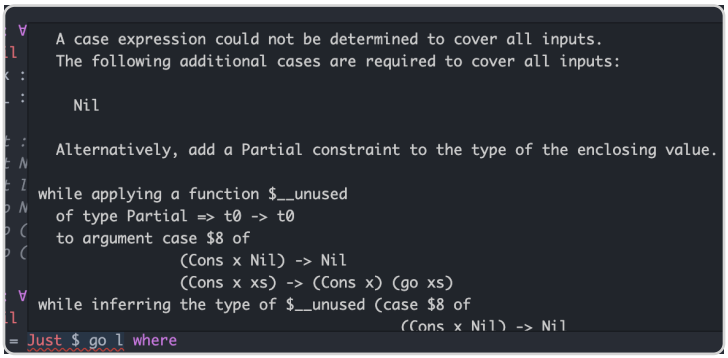


Here `go` returns a `List` but `init` returns a `Maybe`. We need to add the Data Constructor for `Maybe`, i.e we need to add `Just` to our call to `go` turning a `List` into a `Maybe List`:

```
init :: ∀ a. List a -> Maybe (List a)
init Nil = Nothing
init l = Just $ go l where        -- COMPILER ERROR!! ❶
  go (_ : Nil) = Nil
  go (x : xs) = x : go xs
```

❶ Now `init l` returns a `Maybe List`. Note the use of `$` instead of Parentheses.

And once again, we have a different error:



This error says that we're missing a case for `go` when the `List` is empty, i.e. `Nil`.

This may seems wrong at first since we ***know*** that `go` will never get an empty `List`. Why? Because we will never initially call it with an empty `List` and we'll terminate our recursion when the list has only 1 element in it.

But the problem is that the compiler doesn't know any of this. There is a non-empty `List` Type in PureScript that we haven't learned about yet, but that would unnecessarily complicate this particular Function.

So the easiest thing to do is just add that case to appease the compiler, even though we know it will never be executed.

```
init :: ∀ a. List a -> Maybe (List a)
init Nil = Nothing
init l = Just $ go l where
  go Nil = Nil ❶
  go (_ : Nil) = Nil ❷
  go (x : xs) = x : go xs
```

❶ This will never be executed.

❷ The real base-case that terminates the recursion.

And now we have the same compiler error we've seen before on the following line:

```
log $ show $ init Nil
```

We'll need to fix this like before:

```
log $ show $ init (Nil :: List Unit)
```

5.20. Writing uncons

`uncons` is a Function that's the opposite of the `Cons` constructor. Instead of taking a head and a tail and producing a `List`, `uncons` takes a `List` and produces a head and tail.

But Functions in PureScript can only return one value from a Function, not two. So `uncons` solves this problem by returning a `Record` with two fields:

```
uncons :: ∀ a. List a -> Maybe { head :: a, tail :: List a }
```

Since `uncons` can fail to produce both a `head` and a `tail` if given an empty `List`, it returns a `Maybe`.

Add the following code to your test code:

```
log $ show $ uncons (1 : 2 : 3 : Nil) ❶
```

❶ Prints `(Just { head: 1, tail: (2 : 3 : Nil) })`.

This Function is pretty simple compared to what we've written so far, so give it try before moving on.

5.20.1. Code for uncons

Since this Function is so simple, there's no need for a Hint section.

Starting with the Type Signature, we write:

```
uncons :: ∀ a. List a -> Maybe { head :: a, tail :: List a }
```

Next we add the failure case, which is when we have an empty List:

```
uncons :: ∀ a. List a -> Maybe { head :: a, tail :: List a }
uncons Nil = Nothing ❶
```

❶ There is neither a head nor a tail in an empty List.

And finally, we add the non-empty List case:

```
uncons :: ∀ a. List a -> Maybe { head :: a, tail :: List a }
uncons Nil       = Nothing
uncons (x : xs)  = Just { head: x, tail: xs } ❶
```

❶ Destructuring the List we add the parts to our the Record and return it.

5.21. Writing index

index takes a List and an index value (an Int) and returns the value at that index.

We can immediately tell from this description, that this Function can fail. The obvious case is when the index is out of bounds, i.e. it's passed a List of length 3 but an index of 99.

Another case for failure is when given empty List or a negative index.

Add the following code to your test code:

```
log $ show $ index (1 : Nil) 4 ❶
log $ show $ index (1 : 2 : 3 : Nil) 1 ❷
log $ show $ index (Nil :: List Unit) 0 ❸ ❹
```

❶ Prints Nothing.

❷ Prints (Just 2).

❸ Here we have to explicitly specify the Type for the empty List.

❹ Prints Nothing.

Try to think through the strategy before jumping into coding this Function. Then once you have a strategy, give it a try before moving on.

5.21.1. Hint for index

Our strategy for writing this Function should be to remove the head of the List keeping track of what our current index is until we get to the requested index and then return that value, which should be the head we just removed.

At any point, we can exhaust our list before we've gotten to the requested index. This is our failure case.

Since we will need to keep track of our current index as we go, we will need a go Function that takes the current index as an extra Parameter, when compared to index.

If you want a review of how this is done, take a look back at how we made length, tail recursive (see Tail Recursion).

5.21.2. Code for index

Our Type Signature:

```
index :: ∀ a. List a -> Int -> Maybe a
```

One thing to note here. The Parameter order seems off here. Typically, we want the thing that changes most to be last, yet, the List is first. Can you think of any reason this would be defined this way?

Keep your guess in mind as we will see why in a bit.

Next, we add the failure case:

```
index :: ∀ a. List a -> Int -> Maybe a
index Nil _ = Nothing
```

All that's left is the general case. We need to have a go Function so we can keep track of our extra information, i.e. the current index:

```
index :: ∀ a. List a -> Int -> Maybe a
index Nil _ = Nothing
index l i   = go 0 l where ❶
  ???
```

❶ go takes 1 more Parameter than index, i.e. the current index.

We know go is going to be recursive, so we need to write the base-case for go:

```
index :: ∀ a. List a -> Int -> Maybe a
index Nil _ = Nothing
index l i = go 0 l where
  go _ Nil     = Nothing ❶
```

❶ If we get to the base-case for go, we have failed to find an element at the requested index. This is why we return Nothing.

And finally, we need to code the general case for go:

```
index :: ∀ a. List a -> Int -> Maybe a
index Nil _ = Nothing
index l i = go 0 l where
  go _ Nil     = Nothing
  go ci (x : xs) = if ci == i then Just x else go (ci + 1) xs ❶ ❷
```

❶ Don't forget to add (==) to the import Prelude import list, if it's not already there.

❷ Here ci is the current index and i is the request index. When the current index equals the requested index, we have found our element and we simply return it, i.e. Just x. Otherwise, we increment the current index and recurse, i.e. we keep looking.

It's worth noting that we don't bother passing i to go. Can you think of any reason why not?

First of all, i is in go's scope, i.e. go's code can "see" i.

Second, i, the requested index, doesn't change. ci and l do change as we recurse and so they are passed to go.

Based on those two facts, we don't need to pass i to go.

But, what if i did change? A completely different strategy could have been to ***decrement*** the requested index each recursion until it reaches 0 at which point we've found our element:

```
index :: ∀ a. List a -> Int -> Maybe a
index Nil _ = Nothing
index l i = go l i where ❶
  go Nil _     = Nothing ❷
  go (x : xs) ci = if ci == 0 then Just x else go xs (ci - 1) ❸
```

❶ Now we pass i to go since it will change each recursion.

❷ Slight change of Parameter order here (you'll see why in a minute).

❸ ci now stands for our current countdown index and once we have decremented it down to 0, then we've found our element.

Either implementation is fine. Sometimes more than one strategy can work. Also, we could have coded this with Guards:

```
import Prelude (Unit, (+), (==), show, discard, otherwise) ❶

index :: ∀ a. List a -> Int -> Maybe a
index Nil _ = Nothing
index l i = go l i where ❷
  go Nil _     = Nothing
  go (x : xs) ci
    | ci == 0   = Just x
    | otherwise = go xs (ci - 1)
```

❶ Added `otherwise` to our import list.

❷ It appears that we can write this in Point-free notation since the rightmost 2 Parameters are the same on both sides of the equal sign (this is why I swapped the Parameter order earlier).

Now, let's write it Point-free:

```
index :: ∀ a. List a -> Int -> Maybe a
index Nil _ = Nothing   -- COMPILER ERROR!! ❶
index  = go where
  go Nil _     = Nothing
  go (x : xs) ci
    | ci == 0   = Just x
    | otherwise = go xs (ci - 1)
```

❶ Compiler complains due to Parameter list length difference.

Here's the official error:



The reason this error occurs is because we've taken away the Parameters for Pattern Matching. So while ETA-Reduction (what I call cancel on the right) is typically a nice final clean-up step on your Functions, we cannot write Point-free notation if we are also using Pattern Matching.

So put back the parameters to both sides of the equal sign and take a look at your working code. Can you see a failure case that we missed?

What happens when `i` is negative?

Let's fix that oversight:

```
import Prelude (Unit, (+), (==), (<), show, discard, otherwise) ❶

index :: ∀ a. List a -> Int -> Maybe a
index Nil _ = Nothing
index l i = go 0 l where
  go _ Nil     = Nothing
  go ci (x : xs)
    | i < 0     = Nothing ❷
    | ci == i   = Just x
    | otherwise = go (ci + 1) xs
```

❶ Added `<` to import list.

❷ Check for negative index. To add this code, Guards are the cleanest approach.

Adding guards to `index` would've be clunky, which is why the check is done in `go` instead. Technically, it's not as efficient since `i` never changes, but it's such a small overhead, it's worth it to gain readability of the code.

Add the following to your test code and make sure it prints out `Nothing`:

```
  log $ show $ index (1 : 2 : 3 : Nil) (-99) ❶ ❷
```

❶ Make sure you have `negate` imported from Prelude or you'll get an error regarding `Unknown value negate`, since `negate` is used under the covers to make `99` negative.

❷ All negative numbers in PureScript must be surrounded by Parentheses.

5.22. Writing !!

`!!` is the Binary Operator for `index` that lets us write code like:

```
addresses !! 7 ❶
```

❶ Get the element at index 7.

Add the following code to your test code:

```
log $ show $ (1 : 2 : 3 : Nil) !! 1 ❶
```

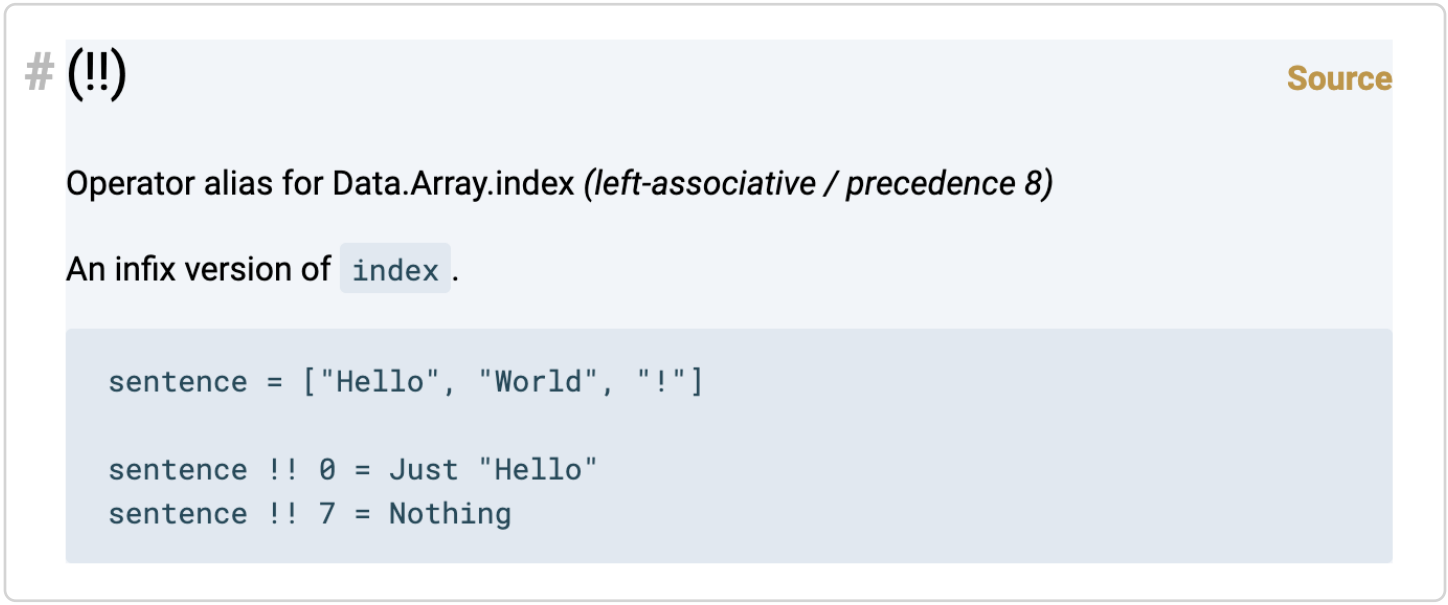
❶ Prints `(Just 2)`.

Go ahead and look up the Associativity and Precedence for `!!` on `Pursuit` and write the code for this Operator.

Make sure you're looking at the `Data.List` version of the Operator even though the other versions have the same values. It's still good practice to make sure you're looking at the correct documentation in `Pursuit` since there are lots of libraries with similar Operators and Function names.

5.22.1. Hint for !!

In case you do not have access to the Internet while reading this:



5.22.2. Code for !!

Since `!!` Operator is Left-Associative and has a Precedence of 8, we write:

```
infixl 8 index as !!
```

Let's look at the usage of this Operator in some detail:

```
index addresses 7
addresses `index` 7
addresses !! 7
```

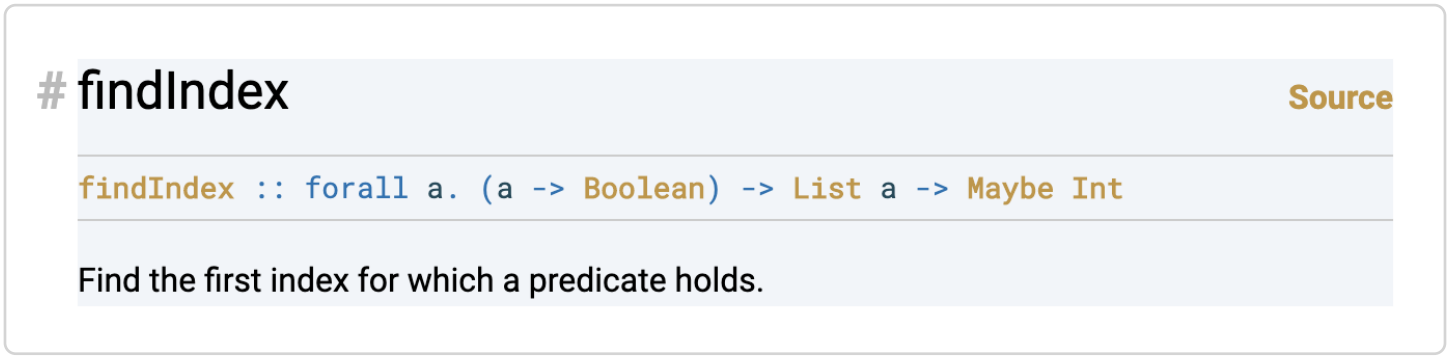
All of the above lines are equivalent. This now explains why the Parameter order for `index` seemed backwards. It was to facilitate the desired syntax for `!!` Operator.

If we need the Parameters in the other order, we can always `flip index`:

```
flip index 7 addresses
```

5.23. Writing `findIndex`

Here is `Pursuit's` explanation for `findIndex`:



So what exactly does this mean?

First, thing to remember is the definition of Predicate. A Predicate is a Function that has the following Type Signature:

```
somePredicateFunction :: ∀ a. a -> Boolean
```

It takes some value of Type `a` and makes a decision based on its Value and returns a `Boolean`. Here's some examples of Predicate Functions:

```
isEven :: Int -> Boolean
isOdd :: Int -> Boolean
(_ > 7) ❶ ❷
("" /= _) ❶ ❸
```

❶ These are wildcard Functions.

❷ This has a Type Signature of `Int -> Boolean`.

❸ This has a Type Signature of `String -> Boolean`.

Let's return to the Type Signature for `findIndex`:

```
findIndex :: ∀ a. (a -> Boolean) -> List a -> Maybe Int
```

We can now see that the first Parameter is a Predicate. It will be used to determine if the element is the one we're looking for.

We can also see that we are returning a `Maybe Int`, which means we are returning the index that we found and it's possible that we fail to find it.

Add the following test code:

```
log $ show $ findIndex (_ >= 2) (1 : 2 : 3 : Nil) ❶
log $ show $ findIndex (_ >= 99) (1 : 2 : 3 : Nil) ❷
log $ show $ findIndex (10 /= _) (Nil :: List Int) ❸ ❹
```

❶ Prints `(Just 1)`.

❷ Prints `Nothing`.

❸ Explicitly specify Type of empty `List` to `Int` (not `Unit`) since the Predicate wants an `Int`.

❹ Prints `Nothing`.

Also, add `(>=)` and `(/=)` operators to the Prelude imports.

Writing this Function isn't too different from other Functions we've written so far. Give it a try and see how you do.

5.23.1. Hint for `findIndex`

The only real hint here is that `findIndex` will need to keep track of the current index as it recurses. We've done this before with `index`. So, look back at the code we wrote for index to see how we've kept track of extra Parameters as we recurse if you cannot remember.

Try to finish coding `findIndex` before reading on.

5.23.2. Code for `findIndex`

First, the Type Signature

```
findIndex :: ∀ a. (a -> Boolean) -> List a -> Maybe Int
```

And then the failure case:

```
findIndex :: ∀ a. (a -> Boolean) -> List a -> Maybe Int
findIndex _ Nil = Nothing ❶
```

❶ This is the base-case for the recursion since we've exhausted our `List`. It's also the failure case for when we're given an empty `List`.

And finally, the general case (or recursive case):

```
findIndex :: ∀ a. (a -> Boolean) -> List a -> Maybe Int
findIndex _ Nil         = Nothing
findIndex pred (x : xs) = ??? ❶
```

❶ We have a problem here since `findIndex` doesn't keep track of an index value. Looks like this is a job for `go`.

Let's add a `go` Function like we did in `index` to keep track of the current index:

```
findIndex :: ∀ a. (a -> Boolean) -> List a -> Maybe Int
findIndex _ Nil  = Nothing
findIndex pred l = go 0 l where ❶ ❷
  ???
```

❶ `go` has 1 extra Parameter than `findIndex`, i.e. the current index, which starts off as 0.

❷ We don't pass `pred` to `go` since it's in scope, i.e. `go` can access it, and never changes.

Since we are writing a new Function, `go`, we have to write the base/failure-case, this time for `go`, and it's recursive/general-case.

First the base-case for `go`:

```
findIndex :: ∀ a. (a -> Boolean) -> List a -> Maybe Int
findIndex _ Nil  = Nothing
findIndex pred l = go 0 l where
  go _ Nil = Nothing ❶
```

❶ The base-case is when we've exhausted the `List`, i.e. we did not find what we were looking for.

Now the recursive case for `go`:

```
findIndex :: ∀ a. (a -> Boolean) -> List a -> Maybe Int
findIndex _ Nil  = Nothing
findIndex pred l = go 0 l where
  go _ Nil      = Nothing
  go i (x : xs) = if pred x then Just i else go (i + 1) xs ❶ ❷
```

❶ We call our Predicate Function, `pred`, with `x` and if it passes, i.e. returns `true`, then we've found what we're looking for and we simply return it. Otherwise, we increment our index and recurse with the tail, `xs`.

❷ Don't forget the `Just` since `go` returns a `Maybe` just like `findIndex` does.

`findIndex` is Tail Recursive since there is no reason to return to the caller after the recursive call to `go`.

5.24. Writing `findLastIndex`

This is similar to `findIndex`, except that it finds the ***last*** index that fulfills the Predicate condition. Keep in mind that just because `findLastIndex` found an index doesn't mean that it stops. `findLastIndex` needs to keep looking because there's always a chance that there's another match later in the `List`.

Many times, creating a simple examples to use as you code helps immensely. Let use the following Predicate:

```
(_ == 10)
```

Here are a few examples to keep in mind while coding `findLastIndex`:

```
Nil
10 : 5 : 10 : -1 : 2 : 10 : Nil
11 : 12 : Nil
```

Let's also use these examples in our test code:

```
log $ show $ findLastIndex (_ == 10) (Nil :: List Int) ❶ ❷
log $ show $ findLastIndex (_ == 10) (10 : 5 : 10 : -1 : 2 : 10 : Nil) ❸ ❹
log $ show $ findLastIndex (_ == 10) (11 : 12 : Nil) ❺
```

❶ Explicit Type for empty `List` of `Int` since Predicate wants an `Int`.

❷ Prints `Nothing`.

❸ Make sure you have `negate` imported from Prelude or you'll get an error regarding `Unknown value negate`. All negative numbers actually use `negate` under the covers. Notice that we don't have to put `-1` in Parentheses since it's part of a `Cons` Binary Operator, i.e. `:`.

❹ Prints `(Just 5)`

❺ Prints `Nothing`.

Keep these examples in mind while writing this Function before you move on to get some hints. Also, don't hesitate to look back at `findIndex` for help since it's a similar Function.

5.24.1. Hint for `findLastIndex`

The basic difference between `findIndex` and `findLastIndex` is that `findLastIndex` has to keep searching even after an index has been found. That means that in `findLastIndex`, we have to keep track of the current index (like in `findIndex`) but we also must keep track of what was found previously if any.

That phrase "if any" tells us that it will be a `Maybe Int` since it's possible we haven't found anything yet. So we'd expect to initialize the found index to `Nothing`.

Try again before continuing.

5.24.2. Code for `findLastIndex`

We start with the Type Signature:

```
findLastIndex :: ∀ a. (a -> Boolean) -> List a -> Maybe Int
```

This signature is exactly the same as `findIndex`. The only difference is that we keep searching until we get to the end of the `List`.

The next step is to add the failure case:

```
findLastIndex :: ∀ a. (a -> Boolean) -> List a -> Maybe Int
findLastIndex _ Nil = Nothing ❶
```

❶ Cannot find an element in an empty `List`, so this fails.

Now, before going further, let's think for a minute. What do we know? We know this is very similar to `findIndex`. We also know that we need to keep track of everything we did in `findIndex`, i.e. the current index, and additionally, we need to keep track of our found index.

So it's probably safe to say that we can use `findIndex` as a template for this Function:

```
findLastIndex :: ∀ a. (a -> Boolean) -> List a -> Maybe Int
findLastIndex _ Nil  = Nothing
findLastIndex pred l = go Nothing 0 l where ❶
  go fi _ Nil      = fi ❷
  go fi i (x : xs) = go (if pred x then Just i else fi) (i + 1) xs ❸
```

❶ We have 1 extra Parameter than we did in `findIndex`, i.e. the first Parameter, which is our found index, which we initially set to `Nothing` since we haven't found anything yet.

❷ This is the end of the line. Since we're looking for the last element that satisfies the Predicate, we have to try every single element in the `List`. When we get here, we've completed our search and simply return the last found index, `fi`, which could be `Nothing`.

❸ The first Parameter to the recursive `go` call is an `if` expression. The `if` is where we decide whether this index is our new last found index. If it is, then we set it to `Just i` else we keep our previous found value, which could be `Nothing`. Then we increment our current index like we did in `findIndex` and keep searching with the tail, i.e. `xs`.

Compare this Function with `findIndex` to see the minor differences.

Also, like `findIndex`, this Function is Tail Recursive. It should be even easier to see that fact than it was for `findIndex` since `findIndex's` recursion call was part of the `else` section of an `if` expression.

5.25. Local Function Type Signatures

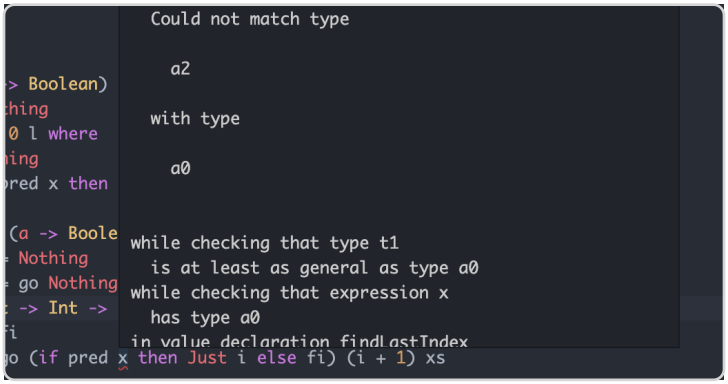
Notice that when we wrote `findLastIndex` that we didn't specify the Type for `go`. So far, we've never specified the Type for `go`. We just rely on the compiler to figure it out.

Let's add it in and see what happens:

```
findLastIndex :: ∀ a. (a -> Boolean) -> List a -> Maybe Int
findLastIndex _ Nil  = Nothing
findLastIndex pred l = go Nothing 0 l where
  go :: ∀ a. Maybe Int -> Int -> List a -> Maybe Int
  go fi _ Nil      = fi
  go fi i (x : xs) =
    go (if pred x then Just i else fi) (i + 1) xs -- COMPILER ERROR!!
```

Notice that we get a compiler error. But why?

Here's the exact error:



It says that it cannot match `a2` with `a0`. The prefix `a` tells us that we're dealing with some sort of `a`, actually two `a's`.

Let's see if we can find those `a's`:

```
findLastIndex :: ∀ a. (a -> Boolean) -> List a -> Maybe Int ❶
findLastIndex _ Nil  = Nothing
findLastIndex pred l = go Nothing 0 l where
  go :: ∀ a. Maybe Int -> Int -> List a -> Maybe Int ❷
  go fi _ Nil      = fi
  go fi i (x : xs) =
    go (if pred x then Just i else fi) (i + 1) xs -- COMPILER ERROR!!
```

❶ Here's an a.

❷ And here's another a.

If we look back on the error message, we can see that the compiler error occurred on the x. We can see this by the red underline under the x, but what gives?

The problem is that pred is defined in terms of the first a and x in terms of the second one.

When we say ∀ a in a local Function, we're telling the compiler that the is a new INDEPENDENT a. Sure we named it a, but it's NOT the same a as in the main Function:

```
findLastIndex :: ∀ a. (a -> Boolean) -> List a -> Maybe Int ❶
findLastIndex _ Nil  = Nothing
findLastIndex pred l = go Nothing 0 l where
  go :: ∀ a. Maybe Int -> Int -> List a -> Maybe Int ❷
  go fi _ Nil      = fi
  go fi i (x : xs) =
    go (if pred x then Just i else fi) (i + 1) xs -- COMPILER ERROR!!
```

❶ Main Function's a.

❷ INDEPENDENT a.

But we don't want the a's to be different from one another. So how do we fix this?

That's easy. We just remove the ∀ a from go's Type Signature:

```
findLastIndex :: ∀ a. (a -> Boolean) -> List a -> Maybe Int
findLastIndex _ Nil  = Nothing
findLastIndex pred l = go Nothing 0 l where
  go :: Maybe Int -> Int -> List a -> Maybe Int ❶
  go fi _ Nil      = fi
  go fi i (x : xs) = go (if pred x then Just i else fi) (i + 1) xs
```

❶ The a in List a is now the same a as in findLastIndex's Type Signature.

Keep this in mind when adding Type Signatures to Local Functions. Most of the time, you're going to want your Type Parameters to match.

5.26. Writing `reverse`

`reverse` takes a `List` and reverses it. Pretty simple Function with a pretty confusing Type Signature:

```
reverse :: List ~> List
```

The `~>` is known as a ***Natural Transformation***. This has to do with the fact that the Type on the left and the Type on the right are both ***Functors***. We don't need to worry about what that means at this point. We'll get to Functors, soon enough.

For now, just realize that the following signatures are equivalent:

```
reverse :: List ~> List ❶
reverse :: ∀ a. List a -> List a ❷
```

❶ Easy to type.

❷ The equivalent signature to the one with the Natural Transformation.

One other thing to keep in mind is that `~>` is a Type Operator, i.e. it's a Binary Operator on Types not Values. That's why it shows up in the Type Signature and not our code.

So you'll need to import this Operator specially:

```
import Prelude (Unit, (+), (==), (<), (>=), (/=), show, discard, negate,
otherwise, type (~>)) ❶
```

❶ `(~>)` needs to be proceeded by the keyword `type`.

Add the following to your test code:

```
log $ show $ reverse (10 : 20 : 30 : Nil) ❶
```

❶ Prints `(30 : 20 : 10 : Nil)`

Try to write the Function using the Natural Transformation in the Type Signature so you'll get used to seeing it. If you get stuck, move forward to read the hints.

5.26.1. Hints for `reverse`

It's common to code `reverse` ***incorrectly*** like:

```
-- THIS IS WRONG
reverse :: List ~> List
reverse Nil      = Nil
reverse (x : xs) = x : reverse xs ❶
```

❶ We've used this pattern a lot and it seems like the right approach without thinking too much about the problem.

When this code is run, the `List` is in the exact same order as before.

The idea was close but the `x` needs to be put at the end of the reversed list not the head:

```
reverse :: List ~> List
reverse Nil      = Nil
reverse (x : xs) = reverse xs : x -- COMPILER ERROR!! ❶
```

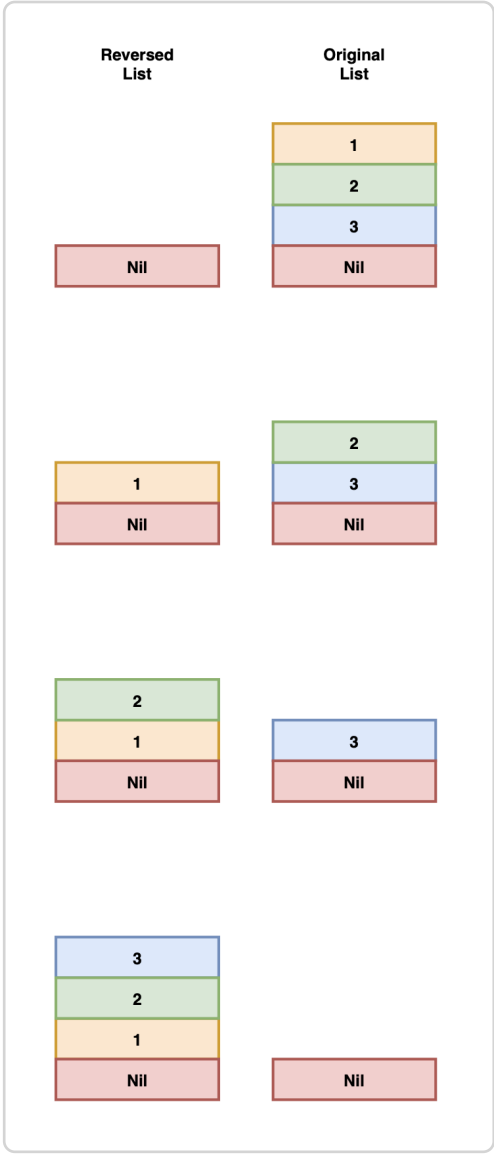
❶ This is invalid since you cannot have a `List` on the left side of the `Cons` Operator, `:`. It must be on the right side.

This is technically what we want, but it's not possible to add to the end of a `List`.

Let's try a completely different tact.

Imagine we have 2 `Lists`, a Reversed List and our Original List. We're going to imagine them as if they were Stacks instead of `Lists` with `Nil` on the bottom and the head of the `List` on top.

And at each step, we take from the top of the Original and put it on the top of the Reverse. (Here top is synonymous with the head.):



Our Original `List` is now reversed.

If you had trouble writing this, try this strategy before moving on.

5.26.2. Code for `reverse`

Starting with the Type Signature:

```
reverse :: List ~> List
```

Adding the base-case:

```
reverse :: List ~> List
reverse Nil = Nil ❶
```

❶ An empty `List` reversed is still an empty `List`.

And now we need to write the recursive case, but based on what we learned in the Hint section, we're going to build up a reversed `List`. This is going to require a local Function that has an extra Parameter for the reversed `List`. We'll stick to tradition and call this Function `go`:

```
reverse :: List ~> List
reverse Nil = Nil
reverse ol = go Nil ol where ❶
???
```

❶ We initialize our reversed `List` to be an empty `List`.

Now we'll write the base-case for `go`. This is when we've exhausted the `List` that we're reversing:

```
reverse :: List ~> List
reverse Nil = Nil
reverse ol = go Nil ol where
  go rl Nil = rl ❶
```

❶ `rl` is our reversed `List` and since our original `List`, `ol`, has no more elements, we can simply return the reversed list.

And finally, we need to build the reverse list in the recursive case:

```
reverse :: List ~> List
reverse Nil = Nil
reverse ol = go Nil ol where
  go rl Nil      = rl
  go rl (x : xs) = go (x : rl) xs ❶
```

❶ Like we saw in the example from the Hint section, we take the head off of our original `List` and add it to the head of our reversed `List`. Go back and look at the Stacks in the diagram in the Hint section if this is confusing.

5.27. Writing `concat`

`concat` takes a `List` of `Lists` and returns a single `List` with all of the elements in the same order as they were. For example:

```
((1 : 2 : 3 : Nil) : (4 : 5 : Nil) : (6 : Nil) : (Nil) : Nil)
  -> (1 : 2 : 3 : 4 : 5 : 6 : Nil)
```

This may be hard to see, so here's what it would look like if it was an Array of Arrays:

```
[[1, 2, 3], [4, 5], [6], []]
  -> [1, 2, 3, 4, 5, 6]
```

Let's use this example for our test code:

```
log $ show $
  concat ((1 : 2 : 3 : Nil) : (4 : 5 : Nil) : (6 : Nil) : (Nil) : Nil) ❶
```

❶ Prints `(1 : 2 : 3 : 4 : 5 : 6 : Nil)`

Give this Function a try before moving forward.

5.27.1. Hint for concat

It's easy to get lost when trying to write this Function for the first time. We want to build a single List, which, so far, has involved building it one element at a time and it's no different here.

The difficulty here is that we can't just take that element from the head of our List since each element is itself a List. So we need a small bit of extra logic to extract an element from our original List of Lists.

See if you can do this using Pattern Matching before moving on to the solution.

5.27.2. Code for concat

As always we start with the Type Signature:

```
concat :: ∀ a. List (List a) -> List a
```

Next, we add the base-case:

```
concat :: ∀ a. List (List a) -> List a
concat Nil = Nil ❶
```

❶ Given an empty List of Lists returns an empty List.

Finally, the recursive case:

```
concat :: ∀ a. List (List a) -> List a
concat Nil = Nil
concat ??? = x : concat ??? ❶ ❷ ❸
```

❶ We need to come up with a Pattern Match that will extract the next element.

❷ We put the extracted element `x` in front of the `concat` of the remainder of the `List`.

❸ We also need to pass the `List` of `Lists` minus the extracted element to recursive call to `concat`.

Extracting that element is the tricky part since it's in the first element of a `List`, which is itself a `List`. Let's think about it for a minute. How do we get to the element, `1` in the following `List`:

```
((1 : 2 : 3 : Nil) : (4 : 5 : Nil) : (6 : Nil) : (Nil) : Nil)
```

We need to get to the `1`, but it's not the head of our `List`, it's the head of the head of our `List`.

Let's see if we can capture that idea in a Pattern Match:

```
concat :: ∀ a. List (List a) -> List a
concat Nil = Nil
concat ((x : xs) : xss) = x : concat ??? ❶ ❷
```

❶ `(x : xs)` is the head of our `List`, which itself is a `List`, and `x` is the head of that `List`.

❷ `xss` is the Plural of `xs`, i.e. the tail of our `List` of `Lists`.

Now we've split our list up into 3 pieces, `x`, `xs` and `xss`. Our element, `x`, is what we'll put on the front of the new `List` so that's accounted for.

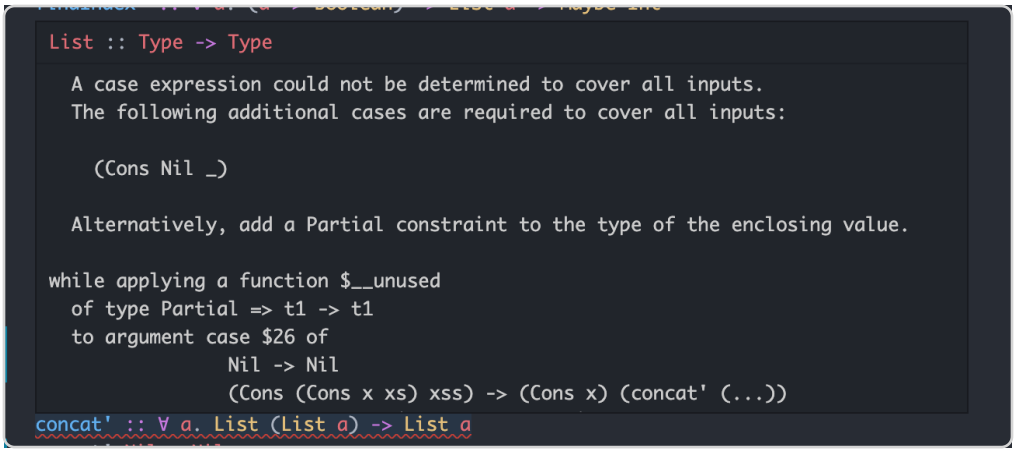
`xs` is the head `List` minus the `x` that we removed. And `xss` is our original `List` of `Lists` minus the head `List`, i.e. minus `xs`. We can put the remaining portion of our head `List`, `xs`, back onto `xss` to get the remaining `List` of `List` to continue `concat`-ing with:

```
concat :: ∀ a. List (List a) -> List a    -- COMPILER ERROR!! ❶
concat Nil = Nil
concat ((x : xs) : xss) = x : concat (xs : xss) ❷
```

❶ The compiler complains because we're missing a case. We'll get to that in a moment.

❷ We put the remaining head `List`, `xs`, back at the head of `xss`, i.e. our `List` of `Lists`.

The exact error from the compiler:



According to the error message, the case that's missing is `(Cons Nil _)`. So let's add it:

```
concat :: ∀ a. List (List a) -> List a
concat Nil = Nil
concat (Nil : xss) = ??? ❶
concat ((x : xs) : xss) = x : concat (xs : xss)
```

❶ This is the case when we've exhausted the head `List`. This is also the case where we encounter an empty `List` in our `List` of `Lists` like our last `List` in our test code.

Once we've depleted the head `List` and processed all its elements, it will be empty and we can just throw it away and continue with the rest of the `Lists`:

```
concat :: ∀ a. List (List a) -> List a
concat Nil = Nil
concat (Nil : xss) = concat xss ❶
concat ((x : xs) : xss) = x : concat (xs : xss)
```

❶ We continue `concat`-ing with the remainder of the `List` of `Lists`.

5.28. Writing `filter`

When I first learned about Functional Javascript, I learned about 3 Functions, `map`, `reduce` and `filter`. `filter` is one of those Functions that cannot be done without being able to pass a Function as an Argument to another Function.

Non-Functional Languages like Java could not do this for a very long time and so this Function seemed so magical because I was never able to write a Function like `filter`. I always had to write a `for-loop`.

But this is not the case in Functional Languages since we have Higher-Ordered Functions, i.e. Functions that take or return Functions.

`filter` takes a Predicate Function to ***KEEP*** or ***filter-in*** elements when the Predicate returns `true`. Take the

time to re-read that last sentence. We filter **IN** not out. That's the opposite of what we mean in English when we say filter. Usually, we're thinking about what we don't want.

Remember to look up the Type Signature on `Pursuit` and make sure you're looking at the definition from `Data.List`. It may be easier just to read ahead and get the Type Signature from there, but practicing looking things up on `Pursuit` has its benefits since this will be what you'll do when not working on problems out of this book.

Add the following test code:

```
log $ show $ filter (4 > _) $ (1 : 2 : 3 : 4 : 5 : 6 : Nil) ❶
```

❶ Prints `(1 : 2 : 3 : Nil)`

You'll need to import `(>)` from Prelude.

This Function is pretty simple. Give it a try before reading on.

5.28.1. Code for `filter`

This Function is simple enough that we can jump right into the code. First, the signature:

```
filter :: ∀ a. (a -> Boolean) -> List a -> List a
```

Since this is an iterative Function, which means recursive, we will have a base-case:

```
filter :: ∀ a. (a -> Boolean) -> List a -> List a
filter _ Nil = Nil ❶
```

❶ Nothing can be filtered from an empty `List`.

And finally, the recursive case:

```
filter :: ∀ a. (a -> Boolean) -> List a -> List a
filter _ Nil = Nil
filter pred (x : xs) = if pred x then x : filter pred xs else filter pred xs
❶
```

❶ If `x` passes the `pred's` test, then we keep it and continue filtering `xs`. Otherwise we just continue filtering `xs`, i.e. the rest of the `List`.

5.28.2. Alternative Coding for `filter`

Here is another way to code `filter` using Guards:

```
filter :: ∀ a. (a -> Boolean) -> List a -> List a
filter _ Nil = Nil
filter pred (x : xs)
  | pred x    = x : filter pred xs
  | otherwise = filter pred xs
```

You may find this version easier to read. I find it a bit more verbose. I find using Guards when a ***single*** `` `if` expression` `` will do, isn't worth the verbosity. Once I need to start nesting `if's`, however, is when I reach for Guards.

It's basically a personal preference. You might like the `` `if` expression` `` but not like it all on one line:

```
filter :: ∀ a. (a -> Boolean) -> List a -> List a
filter _ Nil = Nil
filter pred (x : xs) =
  if pred x then x : filter pred xs
  else filter pred xs
```

Or you might not mind using more lines to express this:

```
filter :: ∀ a. (a -> Boolean) -> List a -> List a
filter _ Nil = Nil
filter pred (x : xs) =
  if pred x then
    x : filter pred xs
  else
    filter pred xs
```

For me, this last version is the worst of them all because it eats up valuable vertical space on the screen.

Code formatting is more than just personal preference. It's also what you're used to looking at. If you're coming from `Elm`, you may find the last version best because you're used to looking at `Elm` code.

I'd suggest you try to get used to looking at code that's more terse, which will help you when you're looking at code written in PureScript and Haskell.

5.29. Tail Recursive version of `filter`

You may have noticed that our version of `filter` is not Tail Recursive:

```
filter :: ∀ a. (a -> Boolean) -> List a -> List a
filter _ Nil = Nil
filter pred (x : xs) =
  if pred x then x : filter pred xs else filter pred xs ❶
```

❶ `x : filter pred xs` must return from `filter` to complete the `Cons` operation making it Non-tail Recursive.

Let's make it Tail Recursive. Typically, you do this by doing the operation, in this case, `Cons`, ***before*** you recurse. That will require an additional Parameter to keep track of the filtered `List` as we recurse. This is where our local `go` Function comes in:

```
filter :: ∀ a. (a -> Boolean) -> List a -> List a
filter pred l = go Nil l where ❶ ❷
  ???
```

❶ `Nil` is the initial value of our new `List`. We'll be adding elements that pass the Predicate to this `List`.

❷ The rightmost Parameter on both sides of the equal sign is `l`. We can ETA-Reduce this (cancel on the right).

The Point-free version:

```
filter :: ∀ a. (a -> Boolean) -> List a -> List a
filter pred = go Nil where ❶
  ???
```

❶ We've removed `l`.

Now we need the base-case for `go`:

```
filter :: ∀ a. (a -> Boolean) -> List a -> List a
filter pred = go Nil where
  go nl Nil = nl ❶
```

❶ Once we've hit the end of our original `List`, we can return our new `List`, `nl`.

Next, we need to code the recursive case for `go`:

```
filter :: ∀ a. (a -> Boolean) -> List a -> List a
filter pred = go Nil where
  go nl Nil      = nl
  go nl (x : xs) = if pred x then go (x : nl) xs else go nl xs ❶ ❷
```

❶ Remember, we don't have to pass `pred` to `go` since it's in `go`'s scope.

❷ If `x` passes `pred`'s requirement, then we first add it to our new `List`, `nl` and then recurse with the remainder of the `List`, `xs`. Otherwise, we just continue with `xs`. Now that we're `Cons`-ing first, our Function is Tail Recursive.

But now we have a big problem. We're adding the elements to our new list ***backwards***. The first element goes onto the head of the list and then the next element goes onto the head and so forth. This will result in reversing our results compared to their original order.

How do we fix this?

Since we can only add to the head of a `List` ***and*** we want this version to be Tail Recursive, we only have one choice and that's to reverse the list at the end.

We can do that with the `reverse` Function we wrote, but before we do, we should make sure that `reverse` is Tail Recursive otherwise there's no point in writing `filter` Tail Recursive if `reverse` is going to overflow our Stack:

```
reverse :: List ~> List
reverse Nil = Nil
reverse ol = go Nil ol where
  go rl Nil      = rl
  go rl (x : xs) = go (x : rl) xs ❶
```

❶ It recurses by calling `go` and nothing else. So this is Tail Recursive.

Now that we can see that `reverse` is Tail Recursive, we can add it to `filter`:

```
filter :: ∀ a. (a -> Boolean) -> List a -> List a
filter pred l = reverse $ go Nil l where ❶ ❷
  go nl Nil      = nl
  go nl (x : xs) = if pred x then go (x : nl) xs else go nl xs
```

❶ We are now reversing the results of `go`.

❷ Notice I had to put back the `l` Parameter that I removed when I was making this Point-free.

To make this version Point-free again, we need to use Function Composition, `<<<`, instead of Function Application, `$`:

```
filter :: ∀ a. (a -> Boolean) -> List a -> List a
filter pred = reverse <<< go Nil where ❶
  go nl Nil      = nl
  go nl (x : xs) = if pred x then go (x : nl) xs else go nl xs
```

❶ The `l` Parameter is gone again. `go Nil` is waiting for 1 Parameter, viz. the `l` we removed. Then `go's` results are passed into `reverse`, which is also waiting for 1 Parameter. Remember, composition only works with Functions waiting for a single Parameter.

5.30. Time vs Space

In our Tail Recursive version of `filter`, we were forced to process the list twice. Once during our filtering and once during reversing. This takes more time than the Non-Tail Recursive version. But it takes less space.

In programming we're always choosing between trading space for time or time for space. The Non-Tail Recursive Function is more Time-Efficient, whereas the Tail Recursive version is more Space-Efficient.

Understanding the differences is critical so that you can make the right choice when you write your own Functions.

5.31. Writing `catMaybes`

`catMaybes` takes a `List` of `Maybe a's` and filters out the `Nothings` and unwraps the `Just a's` giving you a `List` of `a's`.

This is great if you have a `List` of things that may have failed and you only want the successes.

Add the following test code:

```
log $ show $
  catMaybes (Just 1 : Nothing : Just 2 : Nothing : Nothing : Just 5 : Nil) ❶
```

❶ Prints `(1 : 2 : 5 : Nil)`

Try to code this before consulting the next section.

5.31.1. Hint for `catMaybes`

You're going to need to use a `case` expression. Other than that, this is a pretty straightforward Function.

5.31.2. Code for `catMaybes`

Start, as always, with the Type Signature:

```
catMaybes :: ∀ a. List (Maybe a) -> List a
```

And then the base-case:

```
catMaybes :: ∀ a. List (Maybe a) -> List a
catMaybes Nil = Nil
```

And finally, the recursive case:

```
catMaybes :: ∀ a. List (Maybe a) -> List a
catMaybes Nil       = Nil
catMaybes (x : xs) = case x of
  Just y  -> y : catMaybes xs ❶ ❷
  Nothing -> catMaybes xs
```

❶ Here's how we convert the `Maybe a` to simply `a`.

❷ Notice that I used `y` for the Value in the `Just`. I didn't want to call it `x` as that would ***shadow*** the previous `x` in the Pattern Match and `case` expression.

5.32. Writing `range`

`range` is helpful to seed a `List` (or `Array`) with a sequential set of numbers. Obviously, the resulting `List` will be sorted.

`range` includes the endpoints, but there's just one wrinkle. `range` can produce an ascending or descending `List`.

Add the following to your test code:

```
log $ show $ range 1 10 ❶
log $ show $ range 3 (-3) ❷
```

❶ Prints `(1 : 2 : 3 : 4 : 5 : 6 : 7 : 8 : 9 : 10 : Nil)`

❷ Prints `(3 : 2 : 1 : 0 : -1 : -2 : -3 : Nil)`

Stop here and write `range`.

5.32.1. Hint for `range`

There are lots of ways to write this Function. But one thing I'd suggest is to keep in mind that the new `start` Value that you'll pass when recursing is going to be `start + 1` or `start + (-1)` depending on whether

the start is greater than the end or not.

This can be done with an inline if expression to help reduce your code complexity.

5.32.2. Code for range

We start as always with the Type Signature:

```
range :: Int -> Int -> List Int
```

Next, the base-case:

```
range :: Int -> Int -> List Int
range start end   | start == end  = singleton start ❶
                  | otherwise     = ??? ❷
```

❶ Create a List with start as the one and only element.

❷ This is where the recursive case will go.

Notice that we're using Guards and that's because Pattern Matching doesn't handle Boolean logic.

The recursive case is:

```
range :: Int -> Int -> List Int
range start end
  | start == end = singleton start
  | otherwise    =
      start : range (start + (if start < end then 1 else (-1))) end ❶ ❷
```

❶ Place start at the head and recurse with a incremented/decremented start. If start is less than end then we are incrementing else decrementing.

❷ Make sure to import (<) and otherwise from Prelude.

5.33. Optimizing range

You might have written range as follows:

```
range :: Int -> Int -> List Int
range start end =
  let step = if start < end then 1 else (-1) in ❶
  if start == end then singleton start
  else start : range (start + step) end
```

❶ The intent is to do this calculation only once.

If the intent of writing it this way was to do the `step` computation only once, then it actually doesn't accomplish that. Since this Function is recursive, it will recalculate that value each call.

This is a common practice when you are writing loops in Imperative Languages. Typically, you would do this calculation before starting the loop. But for recursion, we have to make sure it's out of the recursive calls, which did not happen here.

Now, it's possible that the compiler can do some magic and optimize this repeated computation out, but I highly doubt it since `start` changes for every recursive call (it's either incremented or decremented).

In reality, the overhead of the `step` computation is so minor that it's not worth doing. But just for argument sake, let's suppose it's a far more expensive computation. How would we optimize this?

Think about it and give it a go before reading on.

We want to compute `step` only once and we want to keep it out of the recursive calls:

```
range :: Int -> Int -> List Int
range start end = go (if start < end then 1 else (-1)) start where ❶
  go step start'  | start' == end = singleton start' ❷
                  | otherwise     = start' : go step (start' + step) ❸ ❹
```

❶ Compute the `step` value only once and pass it to the first call to `go`.

❷ Had to name this variable `start'` so it wouldn't shadow `start`.

❸ The recursion only happens in our `go` Function where we pass in the `step` value, as opposed to computing it each time.

❹ Make sure to import `otherwise` from Prelude if you haven't already.

You may have thought to use a `let-in` expression so you wouldn't have to pass in `step`:

```
-- DOES NOT COMPILE
range :: Int -> Int -> List Int
range start end = let step = if start < end then 1 else (-1) in go start
  where
  go start' | start' == end = singleton start'
            -- COMPILER ERROR!!
            | otherwise     = start' : go (start' + step) ❶
```

❶ `step` is NOT in scope here.

Remember that the `let` values are only in scope for the `in` section, which terminates at the `where`.

But we can accomplish this by moving our `step` calculation to `where` section:

```
range :: Int -> Int -> List Int
range start end = go start where ❶
  go start' | start' == end = singleton' start'
            | otherwise     = start' : go (start' + step)
  step = if start < end then 1 else (-1) ❶
```

❶ We no longer have to pass `step` to `go` since `step` is now in scope for `go` since they are both in the `where` section.

Now, even though we've optimized to this point, we still don't have a Tail Recursive Function.

Let's make that final optimization:

```
range :: Int -> Int -> List Int
range start end = go Nil start where ❶
  go rl start' | start' == end = start' : rl ❷
               | otherwise     = go (start' : rl) (start' + step) ❸
  step = if start < end then 1 else (-1)
```

❶ Pass in an empty `List` as our starting point. It's called `rl` in the `go` Function. It stands for Range `List`

❷ The base-case adds `start'` to the Range `List`.

❸ In this version, we add to the list ***before*** we recurse, which makes this Tail Recursive.

Try running this version and see what happens.

Did you get:

```
10 : 9 : 8 : 7 : 6 : 5 : 4 : 3 : 2 : 1 : Nil) -- BACKWARDS
(-3 : -2 : -1 : 0 : 1 : 2 : 3 : Nil) -- BACKWARDS
```

Look at our latest version of `range` and see if you can figure out what's wrong.

It's because we're putting the `start'` at the head of the `List` first followed by all of the other values and finally `end` is the last number put at the head of the `List`. This produces the `List` backwards.

We could fix this by calling `reverse` since it's Tail Recursive, but that would require making a full second pass over our `List`. While this extra step is often necessary when working with existing `Lists`, we're generating this `List`.

So instead of incurring the extra overhead of reversing the list, we can generate the numbers backwards. This way `end` is put at the head of the list first and `start` is put at the head last:

```
range :: Int -> Int -> List Int
range start end = go Nil end start where ❶
  go rl start' end' | start' == end' = start' : rl ❷
                    | otherwise = go (start' : rl) (start' + step) end'
  step = if start < end then (-1) else 1 ❸
```

❶ Pass `end` as `go's start` and `start` as `go's end`.

❷ `go` now takes an extra Parameter `end'`. It's called `end'` so it won't shadow `end`.

❸ Logic is also swapped here since `step` is calculated based on `start` and `end`, not `start'` and `end'`, which aren't available to us since they're `go's` private Parameters.

We've gone above and beyond here with `range` to make it as efficient as possible. This may be overkill for `range`, but the thought process is a valuable exercise that you can leverage when efficiency does matter.

5.34. Writing `take`

`take` will take the number of specified elements from a `List` or as many as it can, if the `List` is too small.

Look up the definition on `Pursuit` in module `Data.List`.

And add the following code to your test code:

```
log $ show $ take 5 (12 : 13 : 14 : Nil) ❶
log $ show $ take 5 (-7 : 9 : 0 : 12 : -13 : 45 : 976 : -19 : Nil) ❷
```

❶ Prints `(12 : 13 : 14 : Nil)`

❷ Prints `(-7 : 9 : 0 : 12 : -13 : Nil)`

Everything you need to know to write this we've done before. So, try to write this before looking at the hint.

5.34.1. Hint for `take`

`take` needs to keep track of how many items it's taken so it knows when it's done. This would require a `go` Function, but maybe by using the number of elements to take as a countdown, we can avoid keeping track of any extra state and avoid the need for a local Function.

5.34.2. Code for `take`

Type Signature:

```
take :: ∀ a. Int -> List a -> List a
```

Next, the base-case:

```
take :: ∀ a. Int -> List a -> List a
take ??? ❶
```

❶ Turns out there are 2 base-cases.

We actually have 2 base-cases. One when the countdown reaches `0` (or is initially `0`) and the other when the `List` is emptied (or is initially empty):

```
take :: ∀ a. Int -> List a -> List a
take _ Nil = Nil
take 0 _   = Nil
```

Finally, the recursive case:

```
take :: ∀ a. Int -> List a -> List a
take _ Nil       = Nil
take 0 _         = Nil
take n (x : xs) = x : take (n - 1) xs ❶ ❷
```

❶ Add `(-)` to your Prelude import list.

❷ This is NOT Tail Recursive since we need to return back here after the recursive call to add `x` to the head of the `List`.

In a lazy language like Haskell, this implementation would be fine since the `take (n - 1) xs` won't be evaluated until it's needed, if ever, so we get some optimization through laziness. But PureScript is a strict language, meaning that everything is evaluated up front. So, we have to worry about Stack usage for

extremely large Lists.

Try to rewrite this to be Tail Recursive before looking at the solution.

Writing take as a Tail Recursive Function is just an exercise. I would normally just leave the Function as it is, since the chances of overflowing my Stack is extremely rare for the kinds of programs I write.

But it's nice to be able to solve the problem if we ever encounter it:

```
take :: ∀ a. Int -> List a -> List a
take = go Nil where ❶ ❹
  go nl _ Nil       = nl ❷
  go nl 0 _         = nl ❸
  go nl n (x : xs)  = go (x : nl) (n - 1) xs -- WRONG ORDER!! ❹
```

❶ This is written Point-free. The Parameters not named here are the number of items to take and the List. We initially call go with an empty List called nl.

❷ If we exhaust the List or initially get an empty List, then return our new List, which could be Nil.

❸ If we exhaust our count or initially get a count of 0, then return our new List, which could be Nil.

❹ To make this Tail Recursive, we wind up build the List backwards.

Here's the correct version using reverse:

```
take :: ∀ a. Int -> List a -> List a
take n = reverse <<< go Nil n where ❶ ❷ ❸
  go nl _ Nil        = nl
  go nl 0 _          = nl
  go nl n' (x : xs)  = go (x : nl) (n' - 1) xs ❹
```

❶ We had to add the n Parameter back since Functional Composition, <<<, expects Functions of one Parameter.

❷ We call reverse on the final result. We are trading Time for Space here. We're using a Tail Recursive Function because we don't have enough space in our Stack. So, we must spend Time reversing the List at the end.

❸ Don't forget to import (<<<) from Prelude if you haven't already.

❹ Had to rename n to n' so it wouldn't shadow the Parameter n.

One thing to note here is that if you pass in a negative number for the count, this will infinitely recurse and overflow the Stack. Let's fix that.

First, we'll fix the non-Tail Recursive version:

```
take :: ∀ a. Int -> List a -> List a
take n = go (max 0 n) where ❶ ❷
  go _ Nil       = Nil
  go 0 _         = Nil
  go n' (x : xs) = x : go (n' - 1) xs ❸
```

❶ `max 0` will keep the count positive.

❷ Don't forget to import `max` from Prelude.

❸ Had to rename `n` to `n'` so it wouldn't shadow the Parameter `n`.

Now, we'll fix the Tail-Recursive version:

```
take :: ∀ a. Int -> List a -> List a
take n = reverse <<< go Nil (max 0 n) where ❶ ❷
  go nl _ Nil        = nl
  go nl 0 _          = nl
  go nl n' (x : xs)  = go (x : nl) (n' - 1) xs
```

❶ `max 0` will keep the count positive.

❷ Don't forget to import `max` from Prelude.

5.35. Writing `drop`

`drop` differs from `take` in what it returns. Like, `take` it will remove the specified number of elements, but instead of giving you elements the elements taken, it returns the remaining elements. So the following is true:

```
nums = (1 : 2 : 3 : 4 : 5 : 6 : 7 : Nil)
take 2 nums -- (1 : 2 : Nil)
drop 2 nums -- (3 : 4 : 5 : 6 : 7 : Nil)
```

Add the following to your test code:

```
log $ show $ drop 2 (1 : 2 : 3 : 4 : 5 : 6 : 7 : Nil) ❶
log $ show $ drop 10 (Nil :: List Unit) ❷ ❸
```

❶ Prints `(3 : 4 : 5 : 6 : 7 : Nil)`

❷ Explicitly specified Type for our empty `List`.

❸ Prints `Nil`

Try to write drop using what you've learned from take before moving on.

5.35.1. Code for drop

First the Type Signature is not surprisingly the same as take:

```
drop :: ∀ a. Int -> List a -> List a
```

Next the 2 base-cases:

```
drop :: ∀ a. Int -> List a -> List a
drop _ Nil = Nil ❶
drop 0 l   = l ❷
```

❶ If the initial List is empty, then return an empty List. Since we're going to recursively call drop with an ever-decreasing List, this covers the case when we've exhausted our List before our count reaches 0.

❷ If the count is 0, then return the List. This case covers the initial count of 0 or when we countdown to 0.

And finally, the recursive case:

```
drop :: ∀ a. Int -> List a -> List a
drop _ Nil      = Nil
drop 0 l        = l
drop n (_ : xs) = drop (n - 1) xs ❶
```

❶ We dispose of the head and countdown the number of elements.

We're going to skip making this Tail Recursive and safe for negative numbers since it's exactly the same process as in take.

5.36. Writing takeWhile

takeWhile is like take without an explicit count. Instead, it uses a Predicate to determine when to stop. Elements will be taken from the list while the Predicate return true. As soon as it returns false, it stops.

Add this to your tests:

```
log $ show $ takeWhile (_ > 3) (5 : 4 : 3 : 99 : 101 : Nil) ❶
log $ show $ takeWhile (_ == -17) (1 : 2 : 3 : Nil) ❷
```

❶ Prints (5 : 4 : Nil)

❷ Prints Nil

No need for hints since this Function is very similar to what we've been doing. So give it a try. Just reading the code solutions is never enough. You'll learn so much more when you've first tried to code this yourself.

5.36.1. Code for takeWhile

The usual Type Signature:

```
takeWhile :: ∀ a. (a -> Boolean) -> List a -> List a
```

We've replaced Int in take with a -> Boolean in takeWhile.

Next, the one and only base-cases (unlike take):

```
takeWhile :: ∀ a. (a -> Boolean) -> List a -> List a
takeWhile _ Nil = Nil ❶
```

❶ We're giving an initial empty List or we've exhausted our List by recursively calling ourselves with smaller and smaller Lists.

Finally, the recursive case:

```
takeWhile :: ∀ a. (a -> Boolean) -> List a -> List a
takeWhile _ Nil = Nil
takeWhile pred (x : xs) =
  if pred x then x : takeWhile pred xs else Nil ❶ ❷
```

❶ If x passes the Predicate then keep going with the tail, xs. Upon returning, we add x to the head.

❷ If x fails the Predicate, then we're done and return Nil. If this is the first time we've called takeWhile, then the first element doesn't meet our Predicate's criteria and we'll wind up with an empty List and our final return Value. If, however, this is a recursive call, then this Nil will terminate our List as previous x Values are placed at the head of our List one at a time as we return back through all of the previous recursive calls, finally returning our full List.

Let's examine the recursive case in a bit more detail using the following example:

```
takeWhile (_ <= 2) (1 : 2 : 3 : Nil)
```

And let's annotate the different cases of our code:

```
takeWhile :: ∀ a. (a -> Boolean) -> List a -> List a
takeWhile _ Nil = Nil ❶
takeWhile pred (x : xs) = ❷
  if pred x then x : takeWhile pred xs else Nil
```

We're going to use Mathematical Substitution as we've done before. We can do this thanks to Referential Transparency:

```
-- starting with
takeWhile (_ <= 2) (1 : 2 : 3 : Nil)
-- substituting into ❷
if (_ <= 2) 1 then 1 : takeWhile (_ <= 2) (2 : 3 : Nil) else Nil
-- evaluating we get
1 : takeWhile (_ <= 2) (2 : 3 : Nil)
-- substituting into ❷
1 : (if (_ <= 2) 2 then 2 : takeWhile (_ <= 2) (3 : Nil) else Nil)
-- evaluating we get
1 : (2 : takeWhile (_ <= 2) (3 : Nil))
-- substituting into ❷
1 : (2 : (if (_ <= 2) 3 then 3 : takeWhile (_ <= 2) Nil else Nil))
-- evaluating we get
1 : (2 : takeWhile (_ <= 2) Nil else Nil)
-- substituting into ❶
1 : (2 : Nil)
-- removing Parentheses
1 : 2 : Nil
```

5.37. Writing `dropWhile`

`dropWhile` is to `takeWhile` what `drop` is to `take`.

Add this to your tests:

```
log $ show $ dropWhile (_ > 3) (5 : 4 : 3 : 99 : 101 : Nil) ❶
log $ show $ dropWhile (_ == -17) (1 : 2 : 3 : Nil) ❷
```

❶ Prints `(3 : 99 : 101 : Nil)`

❷ Prints `(1 : 2 : 3 : Nil)`

Consult `Pursuit` for the details and write `dropWhile` before moving on.

5.37.1. Code for dropWhile

First things, first:

```
dropWhile :: ∀ a. (a -> Boolean) -> List a -> List a
```

Next, the base-case:

```
dropWhile :: ∀ a. (a -> Boolean) -> List a -> List a
dropWhile _ Nil = Nil
```

Finally, the recursive case:

```
dropWhile :: ∀ a. (a -> Boolean) -> List a -> List a
dropWhile _ Nil         = Nil
dropWhile pred l@(x : xs) = if pred x then dropWhile pred xs else l ❶
```

❶ If `x` passes the Predicate then discard it and continue dropping with the tail, `xs`. Otherwise, we return the FULL `List`, `l`. Notice the `l@` syntax. This creates a variable for the FULL `List` and then proceeds to destructure it using Pattern Matching into `x` and `xs`.

Turns out that `dropWhile` is Tail Recursive, unlike `takeWhile`.

5.38. Writing takeEnd

`takeEnd` does what `take` does, but at the end of the `List`. This makes it a bit more complicated.

Add the following to your tests:

```
log $ show $ takeEnd 3 (1 : 2 : 3 : 4 : 5 : 6 : Nil) ❶
log $ show $ takeEnd 10 (1 : Nil) ❷
```

❶ Prints `(4 : 5 : 6 : Nil)`

❷ Prints `(1 : Nil)`

Try to write `takeEnd` before moving on.

5.38.1. Hint for takeEnd

You may think about writing `takeEnd` as:

```
takeEnd :: ∀ a. Int -> List a -> List a
takeEnd n l = drop (max 0 $ length l - n) l ❶
```

❶ Have to use `max 0` here because `drop` won't work with negative numbers (at least the version we've written).

And while this does work, it traverses the `List` more than once. Once for the `length` call and then partially for the `drop call`. So it's not the most efficient. Normally, if I was just doing this once, I'd be okay with this. But we're writing a library Function and it's going to be used over and over again, so I'd like it to be more efficient.

The strategy for `takeEnd` is a bit different than what we've seen so far. We have to traverse the `List` all the way to the end and as we return from each recursive call, we decide whether we're going to keep the value or not.

To do this, we need to return 2 pieces of information, the `List` that we're building and its `length`. We could call `length` on that `List` each time, but that would be very inefficient since we'd traverse this smaller `List` multiple times as it grows.

Normally, keeping track of 2 things means a `go` routine with 2 more Parameters than our main Function has. But, in this case, we need this information ***after*** the `go` Function returns. How do we do that?

How do you return multiple items from a Function?

A `Record` is one way to go, but in this case, we only need 2 things, so we can use a `Tuple`.

Based on all of this, our approach is going to be to keep calling `go` until we get to the end of the `List`. After we return from `go`, we'll use the `length` information to determine if we should add the current head to the `List` we're building.

If this sounds complex, it kind of is. Take a stab at it before moving on.

5.39. Another Hint for `takeEnd`

If you're lost and don't know where to start, here's the start of the code (the Type Signature and base-case):

```
import Data.Tuple (Tuple(..), snd) ❶

takeEnd :: ∀ a. Int -> List a -> List a
takeEnd n = go >>> snd where ❷ ❸
  go Nil = Tuple 0 Nil ❹
  ???
```

❶ Import `Tuple` and its Data Constructors. Also, import `snd`.

❷ Don't forget to import `(>>>)` from Prelude if haven't already.

❸ Point-free where the `List` isn't named. Once `go` is finished, we get the `List` from the `Tuple` in the second position using `snd`.

❹ Once we fully traverse our original `List`, we will start building our new `List` with `Nil` which clearly has a length of `0`. This is why we start with `Tuple 0 Nil`.

Now the recursive case has to take keep recursing to exhaust the `List` and then build the new `List` ***after*** `go` has returned it's `Tuple`.

To see how this will work, let's take a simple example:

```
takeEnd 2 (1 : 2 : 3 : 4 : Nil)
```

We keep traversing until we get to the base-case. We'll start there and each step thereafter is what's returned to the previous call:

```
Tuple 0 Nil             -- base-case
Tuple 1 (4 : Nil)       -- returning
Tuple 2 (3 : 4 : Nil)   -- returning
Tuple 3 (3 : 4 : Nil)   -- returning ❶
Tuple 4 (3 : 4 : Nil)   -- returning ❶
(3 : 4 : Nil)           -- snd ❷
```

❶ Here we do not change our `List` since we've achieved our desired number of elements. We keep counting (more on that later) but return the same `List`.

❷ We take the `snd` of the `Tuple` at the top and return `(3 : 4 : Nil)` as our final result.

Try to write the recursive case now to accomplish this before moving on.

5.39.1. Code for `takeEnd`

Here's where we left off in the last hint section:

```
takeEnd :: ∀ a. Int -> List a -> List a
takeEnd n = go >>> snd where
  go Nil = Tuple 0 Nil
```

All we need now is the recursive case:

```
takeEnd :: ∀ a. Int -> List a -> List a
takeEnd n = go >>> snd where
  go Nil      = Tuple 0 Nil
  go (x : xs) = go xs ❶
    # \(Tuple c nl) -> Tuple (c + 1) $ if c < n then x : nl else nl ❷ ❸
```

❶ We immediately call `go` on the tail. Remember, we want to work from the end of the `List`, forward. This is why we're building the list ***after*** calling `go`.

❷ We take the return value of `go` and pass it into our Lambda Function using `#`. The `Tuple` is destructured into it's components, `c` for the count of elements and `nl` our new `List`.

❸ We return a `Tuple` with our count incremented and if we haven't met our quota yet, then we add `x` to the head of our new `List`. Otherwise, we just return our final `List`.

Notice that we always increase our count, `c`, even if we aren't adding anything to our new `List`. This is just for terseness. Our count will still be greater than `n` in that case. We could've written so that our count stayed steady, but we only needed it to know when to stop, so it doesn't have to accurately represent the length of our new `List`.

For the curious, here's what it would look like if we wanted to maintain an exact count:

```
takeEnd :: ∀ a. Int -> List a -> List a
takeEnd n = go >>> snd where
  go Nil      = Tuple 0 Nil
  go (x : xs) = go xs
    # \tup@(Tuple c nl) -> if c < n then Tuple (c + 1) (x : nl) else tup ❶
```

❶ We bind the return `Tuple` from `go` to `t` and destructure it with `Tuple c nl`. When we don't add to the list, we return the `Tuple` unchanged, viz. `tup`.

This version is a bit more complex than the one that doesn't worry about the count capping off at the length of our new `List`. And since we dispose of it using `snd`, there's no real reason for the added complexity.

5.40. Writing `dropEnd`

`dropEnd` is like `takeEnd` except it returns what it did not drop off the end of the supplied `List`.

And just like `takeEnd`, we could write this using a `length` calculation:

```
dropEnd :: ∀ a. Int -> List a -> List a
dropEnd n l = take (max 0 $ length l - n) l
```

This approach has all of the same issues we addressed in `takeEnd`. So we should look to write a more

efficient version that this.

Add the following to your tests:

```
log $ show $ dropEnd 3 (1 : 2 : 3 : 4 : 5 : 6 : Nil) ❶
log $ show $ dropEnd 10 (1 : Nil) ❷
```

❶ Prints `(1 : 2 : 3 : Nil)`

❷ Prints `Nil`

Give this a try remembering what you've learned about working from the end of a `List` when we coded `takeEnd`.

5.40.1. Code for `dropEnd`

First the Type Signature:

```
dropEnd :: ∀ a. Int -> List a -> List a
```

We'd expect our code to be similar to `takeEnd` since they both work from the end of the `List`. So until we learn otherwise, let's go down a similar path when adding the base-case:

```
dropEnd :: ∀ a. Int -> List a -> List a
dropEnd n = go >>> snd where
  go Nil = Tuple 0 Nil
```

For the recursive case, we're going to follow what we did with `takeEnd`, but we're going to flip the logic. Our count, `c`, is now going to count the items we're dropping, whereas, in `takeEnd` it was counting what we were taking:

```
dropEnd :: ∀ a. Int -> List a -> List a
dropEnd n = go >>> snd where
  go Nil = Tuple 0 Nil
  go (x : xs) = go xs
    # \(Tuple c nl) -> Tuple (c + 1) $ if c < n then nl else x : nl ❶
```

❶ We've just swapped the `then` part with the `else` part from `takeEnd's` logic, which means that as long as our drop count, `c`, is less than our drop value, `n`, then we leave the new `List` alone. But once we pass that threshold, we start adding `x` to the head.

`takeEnd` counts how many elements were taken and stops adding to new the `List` once its quota is met, whereas, `dropEnd` counts the number of elements dropped until it meets its quota and then it starts adding

to the new List.

5.41. Writing zip

zip will take 2 Lists and zip them into a single List where each element of the output List is a Tuple containing the first List's element in the first position and the second List's in the second position.

One important thing to note is that the shortest List dictates when zip stops.

Consult Pursuit for more information.

Add the following to your test code:

```
log $ show $ zip (1 : 2 : 3 : Nil) ("a" : "b" : "c" : "d" : "e" : Nil) ❶
log $ show $ zip ("a" : "b" : "c" : "d" : "e" : Nil) (1 : 2 : 3 : Nil) ❷
log $ show $ zip  (Nil :: List Unit) (1 : 2 : Nil) ❸ ❹
```

❶ Prints ((Tuple 1 "a") : (Tuple 2 "b") : (Tuple 3 "c") : Nil).

❷ Prints ((Tuple "a" 1) : (Tuple "b" 2) : (Tuple "c" 3) : Nil).

❸ Explicitly specify our empty List Type.

❹ Prints Nil.

Try to write zip without looking further.

5.41.1. Code for zip

Starting with the Type Signature:

```
zip :: ∀ a b. List a -> List b -> List (Tuple a b)
```

Take a minute to look at the signature. Notice how we have to define both a and b, viz. ∀ a b. That's because a Tuple can take 2 different Types. So our Lists don't have to be the same Type. They ***can*** be the same, but do not have to be.

∀ a b says that a and b can be any Type, which means they can both be Int, for example. But the fact that they are named differently means that they are independent and can be different Types just like they are in our test code.

Next, we write the base-case (there are 2):

```
zip :: ∀ a b. List a -> List b -> List (Tuple a b)
zip Nil _ = Nil
zip _ Nil = Nil
```

If either `List` is empty, then there's nothing to `zip`.

And finally, the recursive case:

```
zip :: ∀ a b. List a -> List b -> List (Tuple a b)
zip Nil _             = Nil
zip _ Nil             = Nil
zip (x : xs) (y : ys) = Tuple x y : zip xs ys ❶
```

❶ Take the head off of both lists and wrap it up in a `Tuple`, which we put on the head of the zipped tails.

5.42. Writing `unzip`

`unzip` is the inverse of `zip`. You should be able to take the output of `zip` and apply it to `unzip` and get your original `Lists` back, albeit in a `Tuple`, assuming you started with `Lists` of the same size. That's because `zip` will truncate the longer of the 2 `Lists`.

Add the following to your test code:

```
log $ show $ unzip (Tuple 1 "a" : Tuple 2 "b" : Tuple 3 "c" : Nil) ❶
log $ show $ unzip (Tuple "a" 1 : Tuple "b" 2 : Tuple "c" 3 : Nil) ❷
log $ show $ unzip (Nil :: List (Tuple Unit Unit)) ❸ ❹
```

❶ Prints `(Tuple (1 : 2 : 3 : Nil) ("a" : "b" : "c" : Nil))`.

❷ Prints `(Tuple ("a" : "b" : "c" : Nil) (1 : 2 : 3 : Nil))`.

❸ Explicitly specified the Type of our empty `List`. Notice that `unzip` is expecting a `List` of `Tuples`. We don't care about the underlying Types since our `List` is empty and so we simply specify `Unit`.

❹ Prints `(Tuple Nil Nil)`.

Consult `Pursuit` for the Type Signature and further examples then try to write `unzip` before reading on.

5.42.1. Code for `unzip`

First, the Type Signature and the single base-case:

```
unzip :: ∀ a b. List (Tuple a b) -> Tuple (List a) (List b)
unzip Nil = Tuple Nil Nil
```

Next, the recursive case:

```
unzip :: ∀ a b. List (Tuple a b) -> Tuple (List a) (List b)
unzip Nil = Tuple Nil Nil
unzip (Tuple x y : ts) =
  unzip ts # \(Tuple xs ys) -> Tuple (x : xs) (y : ys) ❶ ❷
```

❶ We pull off the head of the `List` destructuring it in the process into `x` and `y`.

❷ We call `unzip` on the tail and pass the result to a Lambda that destructures the `Tuple` to get the individual `Lists`. Finally, we add our `x` and `y` to the head of their respective `Lists` wrapping the final result into a `Tuple`.

Part II: Intermediate

Chapter 6. Typeclasses

The concepts we've learned so far are sufficient for you to dive into Elm or functional Javascript. But there are abstractions that you'll miss out on that will reduce boilerplate code, e.g. JSON decoding.

With both Haskell and PureScript, the compiler can generate code for us greatly reducing our need to write boilerplate code. Also, with higher level abstractions, we don't have to reinvent the wheel over and over again. Writing this code by hand is tedious and error prone.

From this point forward, we will be looking at language features and concepts that will give you more power than what is available in Imperative Languages with functional features and Elm. At first, you might not miss these features. I didn't. But eventually, you'll find yourself coding the same logic over and over again bloating your codebase. It's at this point, you'll look for solutions to boilerplate and code bloat to no avail. Most languages just don't support boilerplate-reduction abstractions like Typeclasses.

Let's start with some problems that Typeclasses solve.

6.1. The Problem

Imagine we have a Type Alias `Address` for the following `Record`:

```
type Address =
  { street1 :: String
  , street2 :: String
  , city :: String
  , state :: String
  , zip :: String
  }
```

And imagine we also have a function for getting directions like:

```
getDirections :: Address -> Directions ❶
getDirections address = ... ❷
```

❶ Given an `Address`, this will return driving `Directions`.

❷ The implementation details aren't important here.

Let's also assume that we have the following Data Types:

```
data Person = Person
  { name :: String
  , age :: Int
  , address :: Address ❶
  }

data Company = Company
  { name :: String
  , address :: Address ❶
  }

data Residence
  = Home Address ❶
  | Facility Address ❶
```

❶ Each type has an `Address` component.

How would we get `Directions` for each of these types?

Looking at the following concrete examples of each type:

```
person :: Person
person = Person
  { name: "Joe Mama"
  , age: 22
  , address:
    { street1: "123 Main Street"
    , street2: "Apt 152"
    , city: "Jamestown"
    , state: "CA"
    , zip: "95327"
    }
  }

company :: Company
company = Company
  { name: "Acme"
  , address:
    { street1: "987 Tesla Way"
    , street2: "Suite 101"
    , city: "Irvine"
    , state: "CA"
    , zip: "92602"
    }
  }

-- Buford, WY has population of 1
home :: Residence
home = Home
  { street1: "1 1st Street"
  , street2: "Apt 1"
  , city: "Buford"
  , state: "WY"
  , zip: "82052"
  }

facility :: Residence
facility = Facility
  { street1: "54321 Countdown Ave"
  , street2: ""
  , city: "Huntsville"
  , state: "AL"
  , zip: "35805"
  }
```

we can see how the following functions let us get `Directions` for each type:

```
getPersonDirections :: Person -> Directions
getPersonDirections (Person person) = getDirections person.address ❶

getCompanyDirections :: Company -> Directions
getCompanyDirections (Company company) = getDirections company.address ❶

getResidenceDirections :: Residence -> Directions
getResidenceDirections residence = getDirections $ case residence of ❷
  Home address -> address
  Facility address -> address
```

❶ `record.fieldName` is how we access a field of a record.

❷ We have to use a `case` expression for `Residence` since it's a Sum Type.

As a quick aside, we could have written `getResidenceDirections` a few different ways:

```
getResidenceDirections' :: Residence -> Directions
getResidenceDirections' = getDirections <<< case _ of ❶
  Home address -> address
  Facility address -> address

getResidenceDirections'' :: Residence -> Directions
getResidenceDirections'' = getDirections <<< go where ❷
  go (Home address) = address
  go (Facility address) = address
```

❶ Point-free where the `case` expression is using wildcard variables, i.e. the `Residence` value.

❷ Point-free using local function `go`, which uses Pattern Matching instead of a `case` expression.

While writing these functions was pretty simple, what happens when you try to use them in the same function?

Let's try to write a function that will take these three Types:

```
getDirectionsTo :: ??? -> Directions
```

The problem is we want `getDirectionsTo` to accept `Person`, `Company` or `Residence`. But what Type is that?

Notice that we used the word OR. That should clue us into how we can unify these Types. What we need is to create a Sum Type:

```
data HasAddress
  = PersonAddress Person
  | CompanyAddress Company ❶
  | ResidenceAddress Residence ❶
```

❶ Notice the use of the OR symbol, `|`. This says that `HasAddress` equals `PersonAddress` or `CompanyAddress` or `ResidenceAddress`.

Now we have a single Type that encompasses all Types with `Address` to pass to `getDirectionsTo`:

```
getDirectionsTo :: HasAddress -> Directions
getDirectionsTo = case _ of
  PersonAddress (Person person) -> getPersonDirections person
  CompanyAddress (Company company) -> getCompanyDirections company
  ResidenceAddress residence -> getResidenceDirections residence
```

Now we may feel pretty good about our solution at first, but what happens when a new Type with an `Address` appears in our codebase?

```
data EmptyLot = EmptyLot
  { daysEmpty :: Int
  , price :: Int
  , address :: Address
  }
```

Now we have to write the following:

```
getEmptyLotDirections :: EmptyLot -> Directions
getEmptyLotDirections lot = getDirections lot.address
```

Not too bad but we're not done. `getDirectionsTo` needs to be updated as does the Sum Type, `HasAddress`:

```haskell
data HasAddress
  = PersonAddress Person
  | CompanyAddress Company
  | ResidenceAddress Residence
  | EmptyLotAddress EmptyLot ❶

getDirectionsTo :: HasAddress -> Directions
getDirectionsTo = case _ of
  PersonAddress (Person person) -> getPersonDirections person
  CompanyAddress (Company company) -> getCompanyDirections company
  ResidenceAddress residence -> getResidenceDirections residence
  EmptyLotAddress lot -> getEmptyLotDirections lot ❷
```

❶ Added our new Type to the Sum Type.

❷ Added the case for `EmptyLot`.

This may not seem to bad, but imagine we now want to write a function to calculate the distance to the address. Now we need another functions like `getDirectionsTo` but to calculate distances:

```haskell
newtype Miles = Miles Int ❶

getMilesTo :: HasAddress -> Miles
getMilesTo = case _ of
  PersonAddress (Person person) -> getPersonMiles person ❷
  CompanyAddress (Company company) -> getCompanyMiles company ❸
  ResidenceAddress residence -> getResidenceMiles residence ❹
  EmptyLotAddress lot -> getEmptyLotMiles lot ❺
```

❶ We create a `newtype` to wrap `Int` that represents the number of miles as opposed to kilometers.

❷ `getPersonMiles` needs to be written.

❸ `getCompanyMiles` needs to be written.

❹ `getResidenceMiles` needs to be written.

❺ `getEmptyLotMiles` needs to be written.

So now we have a whole set of other functions to write now. Not good.

Let's fix that by writing a single `getMiles` function and use that:

```
getMiles :: Address -> Miles
getMiles address = ... ❶

getMilesTo :: HasAddress -> Miles
getMilesTo = getMiles $ case _ of ❷
  PersonAddress (Person { address }) -> address ❸
  CompanyAddress (Company { address }) -> address ❹
  ResidenceAddress (Home address) -> address ❺
  ResidenceAddress (Facility address) -> address ❻
  EmptyLotAddress (EmptyLot { address }) -> address ❼
```

❶ Details aren't important here.

❷ Here we call `getMiles` with the value of the `case` expression, which is an `Address`.

❸ We pattern match `Person` and destructure its `Record` to get the `address` field.

❹ We pattern match `Company` and destructure its `Record` to get the `address` field.

❺ We pattern match `Home`.

❻ We pattern match `Facility`. Notice we mention `ResidenceAddress` again here with the only difference being which `Residence` constructor that we're matching.

❼ We pattern match `EmptyLot` and destructure its `Record` to get the `address` field.

With this approach we don't have to write as many functions, but it's still a pain to have maintain the Sum Type and update the `case` expression not just in this function, but every single function that accepts our Sum Type.

6.2. The Solution

Everything we've done so far is boilerplate code. You know it's boilerplate when it doesn't take much thought and you can copy and paste existing solutions and make a few minor edits to get your new function or case.

Copying and pasting should be a red flag that tells you that a more generic solution is desirable and, hopefully, possible. Sometimes though, there are technical limitations that force you to create boilerplate code. But in our case, all we need is some way to specify that `Person`, `Company`, `Residence` and `EmptyLot` all have addresses.

If they were all Type Aliases for records then we could simply use `_.address` as our access function (using a wildcard for the `Record`). But one of these is a Sum Type, viz. `Residence`, so that won't work for us.

When we want to treat different things similarly in languages such as C# or Java, we use Interfaces:

```java
// Java
public interface HasAddress {
  Address getAddress();
}
```

```csharp
// C#
interface IHasAddress
{
  Address getAddress();
}
```

In these languages, the Objects need only support the Interface and then anywhere in the code when you want to deal with JUST `Addresses`, you can pass the Interface to your functions.

But in PureScript and Haskell, we don't have Objects. Instead, we have Types.

Luckily for us, we can group Types by creating ***Typeclasses*** and make our Type an ***Instance*** of that Typeclass.

To better understand this, let's first create the Typeclass `HasAddress`:

```
class HasAddress a where ❶ ❷
  getAddress :: a -> Address ❸
```

❶ We use the `class` keyword to create a Typeclass.

❷ Notice that this class has a Single Type Parameter, `a`. That's where we'll put our Type when defining our instance.

❸ Here is where we list ALL of the functions that must be implemented to create an instance of the Typeclass. In this case, we only have 1.

The Type Parameter `a` stands for the Type that implements an instance of the class. So let's write instances for `Person`:

```
instance hasAddressPerson :: HasAddress Person where ❶ ❷
  getAddress (Person p) = p.address ❸
```

❶ All Instances must be named. The idiomatic way to name Instances is to use the Typeclass name followed by all Type Parameters in the specified order. Here we only have 1 Type Parameter, `Person`, so our name is `hasAddressPerson`. N.B. This requirement has been removed in PureScript 0.14.2.

❷ Here Person is the Type Parameter `a` in the `HasAddress` class definition. This means that `getAddress`

will expect a `Person` as its first parameter.

❸ We destructure Person to get to the underlying `Record` and then access the `address` field.

Take a minute to imagine how to write Instances for `Company`, `Residence` and `EmptyLot`. These will mirror `hasAddressPerson` above with the exception of `Residence` since it's a Sum Type.

Here are those Instances:

```
instance hasAddressCompany :: HasAddress Company where
  getAddress (Company c) = c.address

instance hasAddressResidence :: HasAddress Residence where ❶
  getAddress (Home address) = address
  getAddress (Facility address) = address

instance hasAddressEmptyLot :: HasAddress EmptyLot where
  getAddress (EmptyLot l) = l.address
```

❶ `Residence's` Instance has multiple versions of `getAddress` using Pattern Matching.

Armed with this new feature, let's rewrite `getDirections`:

```
-- WILL NOT COMPILE!!!
getDirections :: ∀ a. a -> Directions ❶
getDirections hasAddr = let address = getAddress hasAddr in ❷
  ... ❸
```

❶ Signature is wrong.

❷ By calling `getAddress` we are expecting `hasAddr` to have a `HasAddress` Instance.

❸ Details aren't important here.

But this doesn't compile. Why not?

6.3. Constraints

We've told the compiler that `a` can be ANY Type, yet we're using `a` as if it has an `HasAddress` Instance. We need a way to let the compiler know that `a` can be ANY Type but it also MUST have a `HasAddress` Instance.

We do this using ***Constraints***:

```
getDirections :: ∀ a. HasAddress a => a -> Directions ❶
getDirections hasAddr = let address = getAddress hasAddr in
  ... ❷
```

❶ The Type `a` MUST be an Instance of `HasAddress`.

❷ Details aren't important here.

`HasAddress a =>` is the syntax for constraining `a` to Types which have Instances of `HasAddress`.

You can have multiple Constraints in a single Type Signature:

```
getDirections :: ∀ a. Show a => HasAddress a => a -> Directions ❶ ❷
getDirections hasAddr = let address = getAddress hasAddr in
  ... ❸
```

❶ The Type `a` MUST be an Instance of BOTH `Show` and `HasAddress`.

❷ `Show` is a Typeclass for Types that can be represented as `Strings`.

❸ Details aren't important here.

Notice how the Constraints come between `∀ a.` and the Type Parameters.

You can also define multiple Constraints using Parentheses separating the Typeclasses by commas:

```
getDirections :: ∀ a. (Show a, HasAddress a) => a -> Directions
```

I tend to use the Parenthetical version when my signature is all on one line, but revert to the more verbose signature when my Type Signature spans multiple lines:

```
getDirections
  :: ∀ a
  .  Show a
  => HasAddress a
  => a
  -> Directions
```

Now that we have Typeclasses, we only have to write the following code for `Person`:

```
class HasAddress a where
  getAddress :: a -> Address

data Person = Person
  { name :: String
  , age :: Int
  , address :: Address
  }

instance hasAddressPerson :: HasAddress Person where
  getAddress (Person p) = p.address

getDirections :: ∀ a. HasAddress a => a -> Directions
getDirections hasAddr = let address = getAddress hasAddr in
  ... ❶
```

❶ Details here are not important.

Then if we want to add `Company`, we only need to implement the following:

```
data Company = Company
  { name :: String
  , address :: Address
  }

instance hasAddressCompany :: HasAddress Company where
  getAddress (Company c) = c.address
```

And now we can use `getDirections` with `Company` WITHOUT changing `getDirections`.

With Typeclasses, we have an easy way to treat disperate Types in a similar way without having to create a Sum Type. There's also no need for `case` expressions. The compiler makes the determination of which Type we're referring to at COMPILE-TIME.

This greatly reduces complexity and boilerplate.

6.4. Typeclass Requirement

Every Typeclass function is called a ***Method***:

```
instance hasAddressPerson :: HasAddress Person where
  getAddress (Person p) = p.address ❶
```

❶ `getAddress` is the one and only Method of Typeclass `HasAddress`.

Please don't conflate this with Object Oriented methods. While the words Class and Method are used, they have different meanings here.

There is a requirement for Typeclasses that may not be obvious at first blush.



Every Typeclass Method must have its Polymorphic Parameter in its Type Signature.

The reason for this is because calling `getAddress` isn't enough information to determine which `getAddress` to call since many Types can be instances of `HasAddress`. The compiler needs a Type to determine which `getAddress` to call.

If we look back at our class, we can see that each Method (all 1 of them) requires an `a` as one of its Parameters:

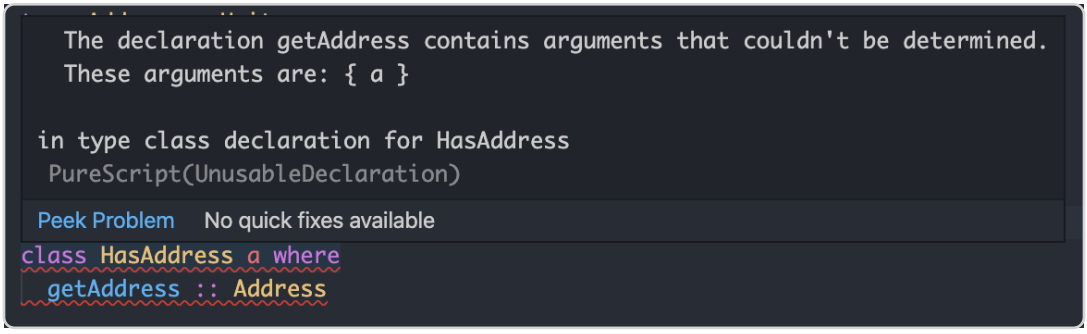
```
class HasAddress a where
  getAddress :: a -> Address ❶
```

❶ Requires `a` as the first Parameter. This information is sufficient for the compiler to determine which `HasAddress` instance to use.

Just for a moment, let's imagine we didn't have this restriction:

```
-- COMPILER ERROR!!
class HasAddress a where
  getAddress :: Address
```

Here is the exact error:



The error specifies that there are "arguments" that cannot be determined, viz. `a`.

6.5. Built-in Typeclasses

In PureScript, there are many built-in Typeclasses. Most of the ones we're going to look at have equivalent,

if not identical, versions in Haskell.

6.5.1. Eq Typeclass

This Typeclass is for comparing two values of the same Type for equality.

Here is the definition:

```
class Eq a where ❶
  eq :: a -> a -> Boolean ❷

infix 4 eq as == ❸
```

❶ The class has a single Type Parameter `a`.

❷ Notice how `eq` has the Type Parameter `a` in its Type Signature.

❸ The operator `(==)` is the infixed version of the `eq` function. It has NO Associativity which means that we are required to add Parentheses when using more than one of these operators in a single expression.

Let's write an instance for `Person`:

```
data Person = Person
  { name :: String
  , age :: Int
  , address :: Address
  }

instance eqPerson :: Eq Person where
  eq (Person p1) (Person p2) = ??? ❶
```

❶ We've destructured both `Person` values to expose their underlying `Records`.

Now we're stuck. We need to determine what does it mean to be the same `Person`. Depending on the Semantics of your Type, which is based on what you're modeling, we may have completely different implementations here.

In this case, let's just assume that if all fields of the `Record` are equal then the `Person` values are equal:

```
instance eqPerson :: Eq Person where
  eq (Person p1) (Person p2) =
    p1.name == p2.name && p1.age == p2.age && p1.address == p2.address ❶
```

❶ `String` (`name`), `Int` (`age`) and `Record` (`Address`) all have instances of `Eq`.

We lucked out here. We didn't have to make any of our field Types an instance of `Eq` because they already are.

However, if Address was defined as a proper Type and not just a Type Alias:

```
data Address = Address
  { street1 :: String
  , street2 :: String
  , city :: String
  , state :: String
  , zip :: String
  }
```

We would have gotten the following error on our instance of `Eq`:

```
instance eqPerson :: Eq Person where
  eq (Person p1) (Person p2) =
    p1.name == p2.name && p1.age == p2.age && p1.address == p2.address
    && :: ∀ a. HeytingAlgebra a ⇒ a → a → a

      No type class instance was found for

        Data.Eq.Eq Address


    while applying a function eq
      of type Eq t0 => t0 -> t0 -> Boolean
      to argument p1.address
    while inferring the type of eq (p1.address)
    in value declaration eqPerson
```

Now the compiler is looking for a Typeclass instance for `Eq Address`. Before, `Address` was just an Alias for `Record`, which has an `Eq` instance.

But now `Address` is its own Type and we need to add an `Eq` instance:

```
instance eqAddress :: Eq Address where
  eq (Address a1) (Address a2) = a1 == a2 ❶ ❷
```

❶ Destructure both `Address` values to expose their underlying `Records`.

❷ We leverage `Record's Eq` instance here when we're using `==`.

`Records` can be compared for equality if and only if all of their field Types also have `Eq` instances. In our case, every field in `Address` is of Type `String`, which has an `Eq` instance.

Here's our final implementation:

```
data Address = Address
  { street1 :: String
  , street2 :: String
  , city :: String
  , state :: String
  , zip :: String
  }

data Person = Person
  { name :: String
  , age :: Int
  , address :: Address
  }

instance eqPerson :: Eq Person where
  eq (Person p1) (Person p2) =
    p1.name == p2.name && p1.age == p2.age && p1.address == p2.address

instance eqAddress :: Eq Address where
  eq (Address a1) (Address a2) = a1 == a2
```

6.5.2. Ord Typeclass

This Typeclass is for determining the ordering of two values of the same Type.

Here's the definition:

```
class Eq a <= Ord a where ❶
  compare :: a -> a -> Ordering

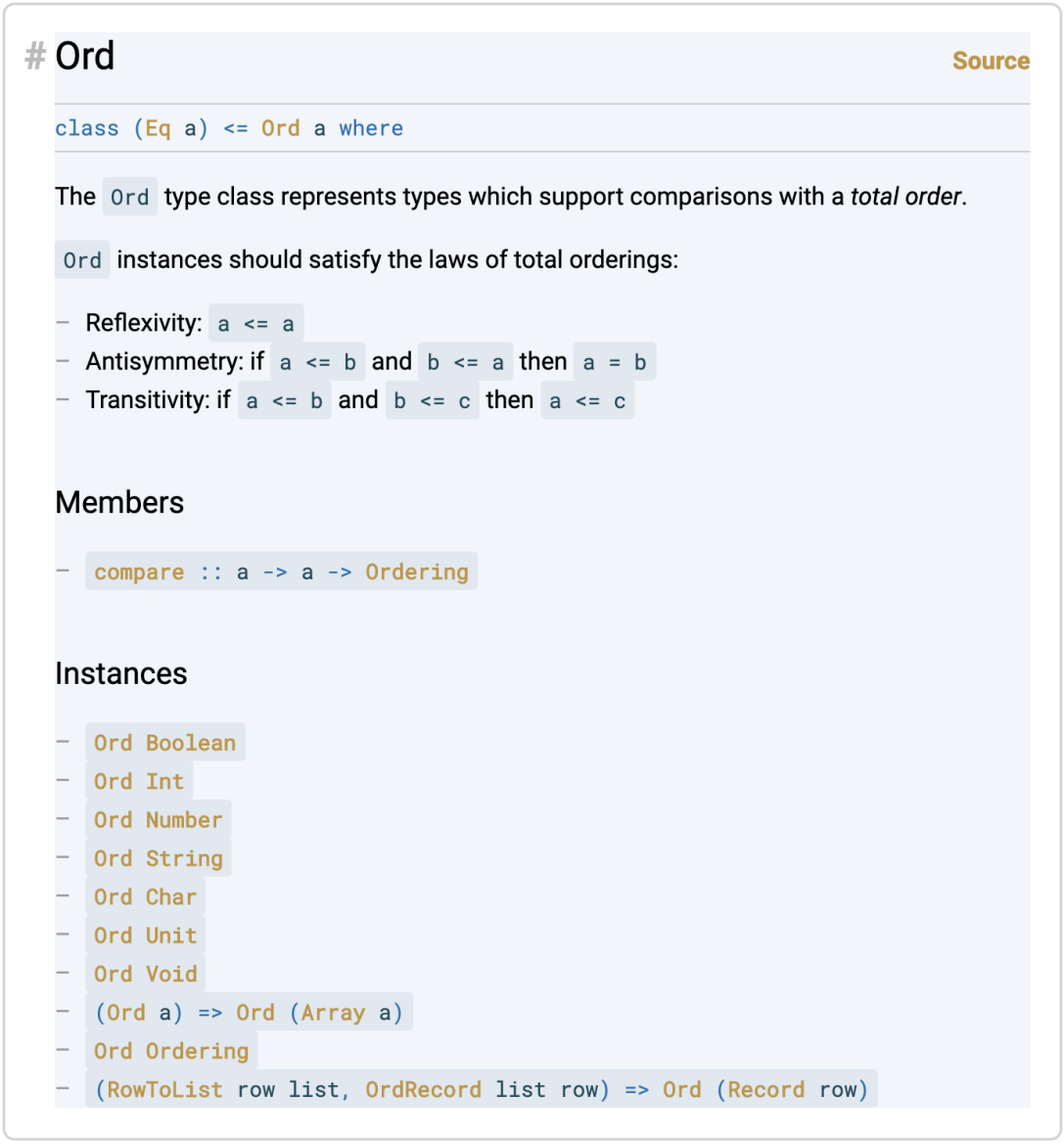
data Ordering = LT | GT | EQ ❷
```

❶ The backward fat-arrow is a Superclass Constraint.

❷ This Sum Type defines the 3 possible Orderings between any 2 values of the same Type.

It is said that `Eq` is a ***Superclass*** of `Ord`. This is NOT like Inheritance. It simply means that any instance of `Ord` MUST also be an instance of `Eq`.

If we consult `Pursuit` for `Ord`, we'll find the following:



Ord Source

```
class (Eq a) <= Ord a where
```

The `Ord` type class represents types which support comparisons with a *total order*.

`Ord` instances should satisfy the laws of total orderings:

- Reflexivity: `a <= a`
- Antisymmetry: if `a <= b` and `b <= a` then `a = b`
- Transitivity: if `a <= b` and `b <= c` then `a <= c`

Members

- `compare :: a -> a -> Ordering`

Instances

- `Ord Boolean`
- `Ord Int`
- `Ord Number`
- `Ord String`
- `Ord Char`
- `Ord Unit`
- `Ord Void`
- `(Ord a) => Ord (Array a)`
- `Ord Ordering`
- `(RowToList row list, OrdRecord list row) => Ord (Record row)`

Notice how most Primitive Types have `Ord` instances, i.e. `Boolean`, `Int`, `String`, etc. But there are a few odd ones, e.g. `Unit`, `Void` and `Ordering`.

What would the instance for `Unit` look like?

Try to write it without reading further.

Writing instances for `Unit` are usually very trivial:

```
instance ordUnit :: Ord Unit where ❶
  compare _ _ = EQ ❷
```

❶ We're using idiomatic instance naming here. First the Class name, `ord`, followed by the Type name, `Unit` resulting in `ordUnit`.

❷ `Unit` has a single inhabitant, viz. `unit`. And everything is equal to itself (Reflexive Property from Math).

What about `Void`? How would write that?

Try to do so without reading further.

`Void` is an odd bird since we can never have a value of `Void`. So you might think it doesn't matter what we pick. Two `Voids` could always `compare` to be `GT`.

But that really flies in the face of logic. How could two values that are impossible to obtain be comparable such that one is greater (or lesser) than the other.

One way to look at this is to use Sets. If we compare the values in Empty Set A with Empty Set B, could we ever show that one would be greater (or lesser) than the other? But we can't prove they are equal either.

But if we decide that all things that don't exist share all discernible attributes with each other. And if we also define that two things are equal when all discernible attributes are equal, then we can say that all non-existent entities are equal to each other resulting in the following implementation:

```
instance ordVoid :: Ord Void where
  compare _ _ = EQ
```

With that brain bender behind us, we can turn our attention to the `Ord` instance for `Ordering`. How would you write such an instance?

Here's the definition to help:

```
data Ordering = LT | GT | EQ
```

Think about some logical ordering for `LT`, `GT` and `EQ`. What order would you put these in?

Seems to me that the most logical ordering would be `LT` would be less than all other Values and `GT` would be greater than all other Values. Based on this logic, it would make more sense if the definition for `Ordering` was:

```
-- NOT the official definition
data Ordering = LT | EQ | GT
```

Then the Values would also be in their natural order, but that's not how the PureScript library defines it.

Try to write `Ord` instance for `Ordering` before moving on.

```
instance ordOrdering :: Ord Ordering where
  compare LT LT = EQ (1)
  compare EQ EQ = EQ
  compare GT GT = EQ
  compare LT _  = LT (2)
  compare EQ LT = GT (3)
  compare EQ GT = LT
  compare GT _  = GT (4)
```

(1) First, we express the 3 `EQ` cases.

(2) Next, we specify that `LT` is less than everything.

(3) Then, we specify that `EQ` is in between `LT` and `GT`.

(4) Finally, we specify that `GT` is greater than everything.

Next, let write `Ord` for our own Type. Here's a simple Sum Type, `Place`:

```
data Place = First | Second | Third
```

To write this, we need to determine what ordering a Sum Type should have. Something logical, intuitive and easy to remember should be our solution.

We could decide that the order in which the Data Constructors or Values, in this case, are defined is the Sum Type's natural ordering. This fits all of our requirements, it's logical, intuitive and easy to remember. And that's exactly the approach that the PureScript compiler takes when it derives it automatically for us (more on this later). So, that will be our solution.

Give it a try before moving on.

Don't forget to create the `Eq` instance for `Place` first.

Here is the `Eq` instance:

```
instance eqPlace :: Eq Place where
  eq First First   = true
  eq Second Second = true
  eq Third Third   = true
  eq _ _           = false
```

Next, is the `Ord` instance for `Place`:

```
instance ordPlace :: Ord Place where
  compare First First   = EQ
  compare First _       = LT
  compare Second Third  = LT
  compare Second Second = EQ
  compare Second First  = GT
  compare Third Third   = EQ
  compare Third _       = GT
```

So what does all this work get us?

We can sort an `Array` of `Places`:

```
import Data.Array (sort) ❶

x :: Array Place
x = [Third, First, Second]

sx :: Array Place
sx = sort x -- [First, Second, Third]
```

❶ Import the `sort` function from `Data.Array`.

Looking at the `sort` Type Signature:

```
sort :: ∀ a. Ord a => Array a -> Array a
```

we notice that `a` is Constrained by `Ord a`, i.e. it must be have an instance of `Ord`, which `Place` does. Without it, `sort` couldn't decide which element of the `Array` goes where.

We can also use our Type as a key in a Map:

```
insert :: ∀ k v. Ord k => k -> v -> Map k v -> Map k v
```

Notice that ONLY the key, `k`, must have an `Ord` instance. That's because we place the value into the `Map` structure based on its key's ordering.

`Maps` are found in the package `purescript-ordered-collections` in module `Data.Map`.

To add the package to your `spago.dhall` file, remember to remove the prefix `purescript-`.

6.5.3. Show Typeclass

Another common built-in Typeclass is `Show`. This Class is for turning a value of your Type into a String so that we can display it.

Here's its definition:

```
class Show a where
  show :: a -> String
```

Try to write a `Show` instance for our `Place` Type.

The instance is as follows:

```
instance showPlace :: Show Place where ❶
  show First  = "First"
  show Second = "Second"
  show Third  = "Third"
```

❶ We use idiomatic naming of our instance, i.e. Class name followed by Type name.

6.6. Derived Instances

Let's imagine we have another Sum Type:

```
data SomeType = This | That | TheOther | AndYetAnother
```

And we want to make it have `Eq`, `Ord` and `Show` instances. What would be your first inclination? Well, if you're like me you'd probably look back at what we did for `Place`, copy and paste it and begin editing.

And when you're finished you'd arrive here:

```
instance eqSomeType :: Eq SomeType where
  eq This This                   = true
  eq That That                   = true
  eq TheOther TheOther           = true
  eq AndYetAnother AndYetAnother = true
  eq _ _                         = false

instance ordSomeType :: Ord SomeType where
  compare This This                   = EQ
  compare This _                      = LT
  compare That TheOther               = LT
  compare That AndYetAnother          = LT
  compare That That                   = EQ
  compare That This                   = GT
  compare TheOther AndYetAnother      = LT
  compare TheOther TheOther           = EQ
  compare TheOther This               = GT
  compare TheOther That               = GT
  compare AndYetAnother AndYetAnother = EQ
  compare AndYetAnother _             = GT

  instance showPlace :: Show Place where ❶
  show This  = "This"
  show That = "That"
  show TheOther  = "TheOther"
  show AndYetAnother  = "AndYetAnother"
```

Just in case you're wondering, that was painful. It was nothing but boilerplate code and I had to check it more than once to make sure it was right. And it could still have a bug in it because going over that makes my eyes glaze over.

This is something that's better for a machine to do than a human. Luckily, PureScript allows us to derive these using the following syntax;

```
data SomeType = This | That | TheOther | AndYetAnother

derive instance eqSomeType :: Eq SomeType ❶
derive instance ordSomeType :: Ord SomeType ❷
```

❶ Derive `Eq` instance.

❷ Derive `Ord` instance.

Since writing these is usually a mechanical process, the compiler makes our jobs much easier by writing

them for us. Keep in mind that deriving will order Sum Types in the order that you define the Data Constructors.

But what about Show?

Turns out that Show can also be derived but with a little bit of extra work.

```
import Data.Generic.Rep (class Generic) ❶
import Data.Show.Generic (genericShow) ❷

derive instance genericSomeType :: Generic SomeType _ ❸
instance showSomeType :: Show SomeType where
  show = genericShow ❹
```

❶ We need to import the Class Generic. Notice the keyword class.

❷ We also need to import the generic Show function, genericShow.

❸ We can derive the Generic instance for our Type. In fact, we can ONLY derive this. We're not allowed to write our own instance. This guarantees that the Internal Representation is done correctly. This is also why the second Type Parameter to the Generic Type is _. The compiler will fill in its Type to guarantee that it too is correct.

❹ We can now leverage the generic show for our Type.

There's a lot to unpack here. To start, the compiler can generate a Internal Representation of SomeType. That Representation has a Type, which is filled in by the compiler in the second Type Parameter to the Generic Class. The one where we put an underscore.

The genericShow function, utilizes the Internal Representation to display values of SomeType. That's why our Type MUST be an instance of Generic otherwise genericShow cannot interpret it at runtime.

This can seem a bit confusing, since it's dealing with representing Types at runtime. We will look into Generics in depth later, but for now, you can simply use it as is shown above.

For another concrete example, here's how we'd use genericShow for Place:

```
derive instance genericPlace :: Generic Place _
instance showSomeType :: Show Place where
  show = genericShow
```

It was a copy/paste of the SomeType implementation with only name changes.

6.7. Newtype Typeclass

The Newtype Class allows for generic wrapping and unwrapping of newtype Types. The Type must be

defined using `newtype` and NOT `data` to derive an instance of this TypeClass.

Here's a simple example using our old friend, `FirstName`:

```
newtype FirstName = FirstName String ❶
derive instance newTypeFirstName :: Newtype FirstName _ ❷
```

❶ Here we wrap `String` in the Data Constructor `FirstName` for safety reasons, i.e. we can't just pass any old `String` to a function expecting a first name.

❷ We derive a `Newtype` instance for `FirstName`. Notice, just like `Generic`, the second Type Parameter, the wrapped Type, must be `_`. The compiler will fill in this Type for us.

We can use `NewType's` Methods, `wrap` and `unwrap` to work with `newtypes` generically.

Let's look at a specific case of using `unwrap`:

```
import Data.Newtype (class Newtype, unwrap) ❶

newtype FirstName = FirstName String
derive instance newTypeFirstName :: Newtype FirstName _

newtype LastName = LastName String
derive instance newTypeLastName :: Newtype LastName _

fullName :: FirstName -> LastName -> String ❷
fullName (FirstName first) (LastName last) = first <> " " <> last

fullName' :: FirstName -> LastName -> String ❸
fullName' first last = unwrap first <> " " <> unwrap last
```

❶ We need to import the `Newtype` Class. Notice the `class` keyword.

❷ This is unwrapping using Pattern Matching.

❸ This is unwrapping using `Newtype's` Methods.

You may think that this isn't worth the trouble. But when we don't know the exact Type, it becomes much more powerful.

Here's an example that illustrates that scenario:

```
glueNames ❶
  :: ∀ a b
   . Newtype a String ❷
  => Newtype b String ❷
  => String
  -> a
  -> b
  -> String
glueNames between n1 n2 = unwrap n1 <> between <> unwrap n2 ❸

lastNameFirst :: LastName -> FirstName -> String ❹
lastNameFirst = glueNames ", " ❺

fullName :: FirstName -> LastName -> String ❹
fullName = glueNames " " ❻
```

❶ When Type Signatures become long, this is an alternative formatting that is commonly used. It takes a bit of getting used to, but definitely helps readability.

❷ We are constraining both `a` and `b` to be `newtypes` that have instances of `Newtype`. They also must both wrap a `String` Type. Both `FirstName` and `LastName` fit this requirement.

❸ Our generic function to glue these names together uses `Newtype's` unwrap Method completely oblivious to the actual Types.

❹ Notice the order of Parameters is flipped between `lastNameFirst` and `fullName`. Since `glueNames` doesn't care about the exact types, we have this flexibility. The only thing `glueNames` requires is that it gets two `newtypes` that both wrap `String`.

❺ We call `glueNames` with a comma as our delimiter. This is written in Point-free notation where last name and first name are not explicitly specified.

❻ We call `glueNames` with a space as our delimiter. This too is written in Point-free notation, where first name and last name are not explicitly specified.

6.8. Deriving Instances using `newtype`

Let's revisit `FirstName` and derive it's `Eq` instance:

```
newtype FirstName = FirstName String
derive instance newTypeFirstName :: Newtype FirstName _
derive instance eqFirstName :: Eq FirstName ❶
```

❶ Deriving `Eq`.

That was easy since we can always derive `Eq`. But things become a bit more complicated when our `newtype`

wraps something more complicated than a `String` or if we want our `newtype` to be an instance of something not derivable, e.g. `HasAddress`.

Let's revisit `Person` and `HasAddress`:

```
newtype Person = Person ❶
  { name :: String
  , age :: Int
  , address :: Address
  }

instance hasAddressPerson :: HasAddress Person where ❷
  getAddress (Person p) = p.address

newtype Ceo = Ceo Person ❸

newtype Janitor = Janitor Person ❸
```

❶ Notice this is a `newtype` whereas previously we defined this with `data`.

❷ Here's the same `HasAddress` instance implementation from before.

❸ We have a `newtype` for `Ceo` and `Janitor`, presumably to make sure we don't call `Ceo` functions with a `Janitor`.

Go ahead and write `HasAddress` for both `Ceo` and `Janitor`. Try not to look ahead, but please, by all means, look back at what we've done so far.

There are multiple ways to implement `HasAddress` instances for `Ceo` and `Janitor`.

First, Hand-coded:

```
instance hasAddressCeo :: HasAddress Ceo where
  getAddress (Ceo p) = getAddress p ❶

instance hasAddressJanitor :: HasAddress Janitor where
  getAddress (Janitor p) = getAddress p ❶
```

❶ We use Pattern Matching to unwrap `Person`.

Next, we write it *generically* using unwrap from `Newtype`:

```
import Data.Newtype (class Newtype, unwrap) ❶

newtype Ceo = Ceo Person
derive instance newtypeCeo :: Newtype Ceo _ ❷

newtype Janitor = Janitor Person
derive instance newtypeJanitor :: Newtype Janitor _ ❷

genericPersonHasAddress :: ∀ a. Newtype a Person => a -> Address ❸
genericPersonHasAddress wrappedPerson =
  getAddress $ unwrap wrappedPerson ❹

instance hasAddressCeo :: HasAddress Ceo where
  getAddress = genericPersonHasAddress ❺

instance hasAddressJanitor :: HasAddress Janitor where
  getAddress = genericPersonHasAddress ❺
```

❶ We need to import `Newtype` Class and `unwrap`.

❷ We need to make `Ceo` and `Janitor` instances of `Newtype` Class.

❸ `a` is any type as long as it's a `Newtype` that wraps `Person`, which is what `Newtype a Person` means.

❹ We unwrap `Person` and then call `getAddress` with it. Remember that since `getAddress's a` is `Person`, the compiler will call `Person's getAddress` Method.

❺ We can now leverage `genericPersonHasAddress` for both `Ceo` and `Janitor`. This is very similar to how we used `genericShow`.

While this is more code than hand coding, the real benefit here is that every single Type that wraps `Person` can now leverage `genericPersonHasAddress`.

And finally, we can write these instances by using `derive newtype`:

```
import Data.Newtype (class Newtype) ❶

newtype Ceo = Ceo Person
derive instance newtypeCeo :: Newtype Ceo _ ❷
derive newtype instance hasAddressCeo :: HasAddress Ceo ❸

newtype Janitor = Janitor Person
derive instance newtypeJanitor :: Newtype Janitor _ ❷
derive newtype instance hasAddressJanitor :: HasAddress Janitor ❸
```

❶ We need to import `Newtype` Class.

❷ We need to make `Ceo` and `Janitor` instances of `Newtype` Class.

❸ I cheated and used `derive newtype`.

In the final version, I cheated by using something we haven't seen yet, `derive newtype`. This tells the compiler that our Type is just a `newtype`, so derive the instance, in this case `HasAddress`, by calling the wrapped Type's implementation, viz. `Person's HasAddress` instance.

This is very convenient and saves us from having to write boilerplate code. It's also something the compiler can do for us very easily.

One very important point when using this technique is that the underlying Type MUST implement the instance. If `Person` didn't have a `HasAddress` instance, then the compiler would have complained.

6.9. Overlapping Instances

Take the following instances:

```
data Overlap = Overlap

instance overlapShow1 :: Show Overlap where
  show = "Overlap 1"

instance overlapShow2 :: Show Overlap where -- COMPILER ERROR!!
  show = "Overlap 2"
```

The compiler isn't happy to see two instances for showing `Overlap`. The truncated error message is:

```
Overlapping Typeclass instances found for

    Data.Show.Show Overlap
```

You can understand why this is not allowed since there'd be no way for the compiler to resolve which of these to use.

But there are times we need to deal with such a situation. Let's look at the following Class to `Combine` values:

```
class Combine a where
  combine :: a -> a -> a ❶
```

❶ Takes an `a` and another `a` and combines them somehow and produces another `a`.

If we want to combine `Ints` using addition, we'd make the following instance of `Combine`:

```
instance combineInt :: Combine Int where
  combine = (+)
```

But what if we also want to combine `Ints` using multiplication, in the same program:

```
instance combineMultInt :: Combine Int where -- OVERLAPPING INSTANCES!!
  combine = (*)
```

The problem is that we have two ways of combining `Ints` in the same module, package or program. It would be fine if we had one way for `Ints` and one way for `Strings`. But, in this case, we have two ways for `Ints`.

There is a way out of our quagmire by creating `newtypes` for `Int`. One Type that combines using addition and another that combines using multiplication:

```
newtype AddInt = AddInt Int ❶
newtype MultInt = MultInt Int ❷

instance combineAddInt :: Combine AddInt where
  combine (AddInt x) (AddInt y) = AddInt (x + y) ❸

instance combineMultInt :: Combine MultInt where
  combine (MultInt x) (MultInt y) = MultInt (x * y) ❹
```

❶ Our `newtype` for additive combinations.

❷ Our `newtype` for multiplicative combinations.

❸ Destructure our Parameters and compute their sum rewrapping them back up into an `AddInt`.

❹ Destructure our Parameters and compute their product rewrapping them back up into an `MultInt`.

So instead of using `Ints`, we use either `AddInts` or `MultInts` and there is no ambiguity and no Overlapping Instances.

But what if our Types aren't Monomorphic? In the above case, we had the concrete Type, `Int`.

Here's a Polymorphic example:

```
class IsRecord a where
  isRecord :: a -> Boolean ❶

instance isRecordRecord :: IsRecord (Record a) where ❷
  isRecord _ = true

instance isRecordOther :: IsRecord a where -- COMPILER ERROR ❸
  isRecord _ = false
```

❶ Here we have a Class whose `isRecord` Method returns `true` if it's a `Record`.

❷ This is `Record's` instance for `IsRecord`.

❸ What we're trying to do here is write an instance for EVERY OTHER Type to have it return `false` since we've already handled the `Record` instance.

The compiler isn't happy with this. Can you see why?

It may be clear to us what we're trying to do here, and it works fine for every Type except for `Record`.

When the Type is `Record a`, it matches both instances. In the first instance, `Record a` unifies with `Record a`. In the second instance, `Record a` unifies with `a`. (Remember the `a` in `Record a` is different than `a` in `class IsRecord a`.)

What we need is a way to specify the first instance as a specific case and to specify the second as a Catch-All. And we can do exactly that with ***Instance Chaining***:

```
class IsRecord a where
  isRecord :: a -> Boolean

instance isRecordRecord :: IsRecord (Record a) where
  isRecord _ = true
else instance isRecordOther :: IsRecord a where ❶
  isRecord _ = false
```

❶ The addition of the keyword `else` "Chains" these two instances together.

When using Instance Chaining, order matters. So if we were to swap the instance order the compiler will NOT complain. It just will NEVER use the second one since `a` will match all Types. This is the same issue we've seen with Pattern Matching.

6.10. Orphaned Instances

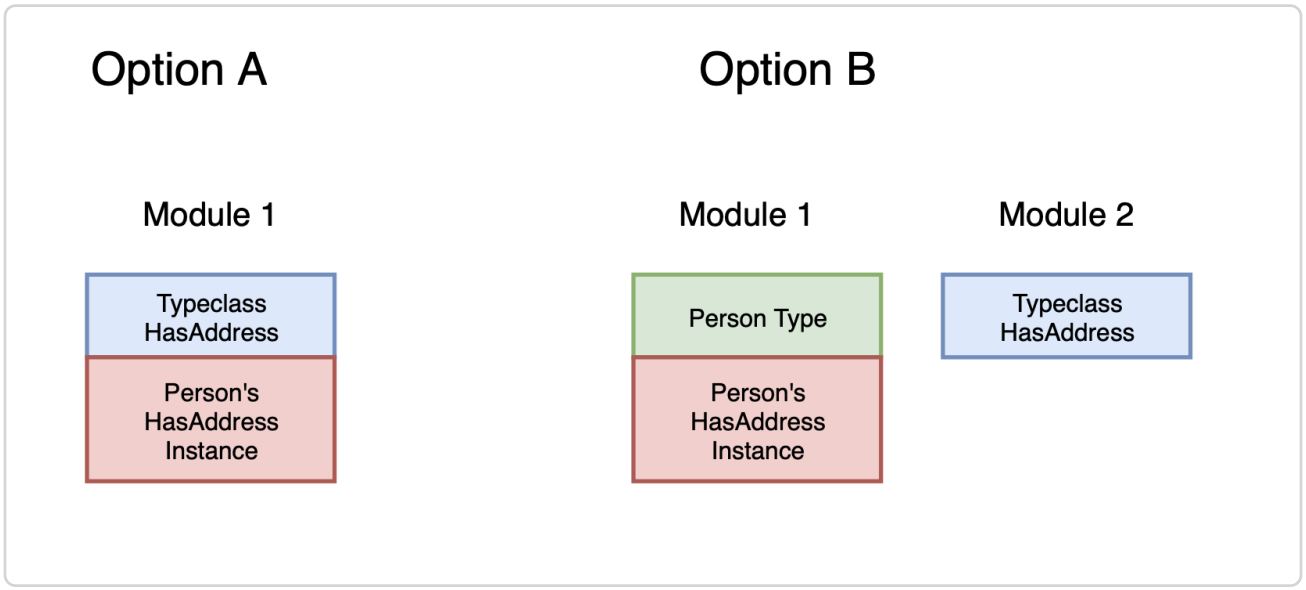
There are some restrictions on where we can define instances of Classes and for good reason, which will become apparent soon enough:



Instances MUST be *either*:

1. Defined in the SAME module as the Typeclass.
2. Defined in the SAME module as the Type.

Here's is a pictorial view of those 2 cases:

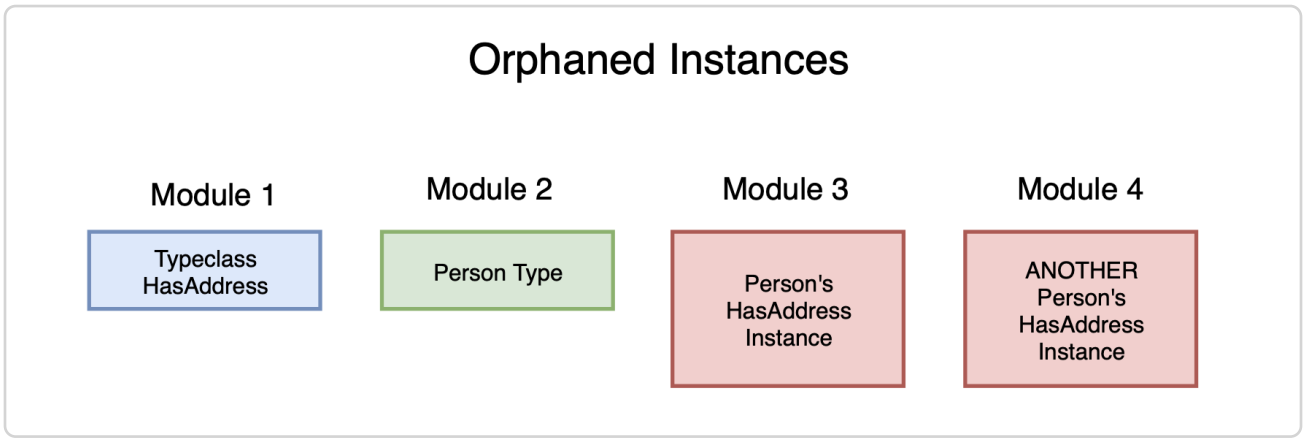


In `Option A`, we have defined `Person's` instance in the SAME module as the Typeclass.

`Option B` defines `Person's` instance in the SAME module as its Type whereas the Typeclass is define in different module.

If either of these two requirements are not followed, then we have an ***Orphaned Instance***.

This restriction may seem odd at first glance. Let's look at the alternative to see were things go wrong:



Here `Module 3` defines `Person's HasAddress` instance. But then so does `Module 4`. Let's imagine that `Module 3` and `Module 4` are fully independent from one another, i.e. neither import the other.

Now we'll have 2 potentially different implementations of `HasAddress` for `Person`. If half of our modules

import `Module 3` and the other half import `Module 4`, we could be in a world of hurt.

What we need is an Authoritative Definition and there really is only two places of Authority, the module where the Typeclass is defined (`Module 1`) and the module where the Type is defined (`Module 2`).

This may seem limiting and frankly, sometimes it is. So much so that in Haskell, there's ways to tell the compiler to allow Orphaned Instances in a particular module, i.e. essentially defining a specific module to be the Authority. However, this is not without risk. If this is overused, then it's possible to wind up with two disparate definitions of the same instance.

So in PureScript, this is disallowed with no override possible. But what happens when you do find yourself with an Orphaned Instance. Let's look at just such a case:

```
import Combine.Class (class Combine) ❶

-- ORPHANED INSTANCE!!
instance combineInt :: Combine Int where ❷
  combine = (+)
```

❶ The Typeclass is defined outside of this module.

❷ The `Int` Type is defined in the PureScript module `Prim` and therefore the Type is defined outside of this module.

This is a classic case of an Orphaned Instance. You don't control either the Typeclass or the Type. But you want to make the Type have additional functionality.

What can be done?

We need a way to control either the Typeclass or the Type. If you make another Typeclass, then all the code that's using the other Typeclass won't work with yours. But if we make our own Type, then we'd be the authoritative voice for that Type.

And how do we do that? With `newtype`, of course:

```
import Combine.Class (class Combine)

newtype AddInt = AddInt Int ❶

instance combineAddInt :: Combine AddInt where ❶
  combine (AddInt x)  (AddInt y) = AddInt (x + y)
```

❶ Now our `AddInt` Type is in the same module as our `Combine` instance for `AddInt`. No more Orphaned Instance.

This solution doesn't come for free. We cannot use `AddInt` as we would `Int`:

```
x :: AddInt
x = AddInt 10

y :: Int
y = 11

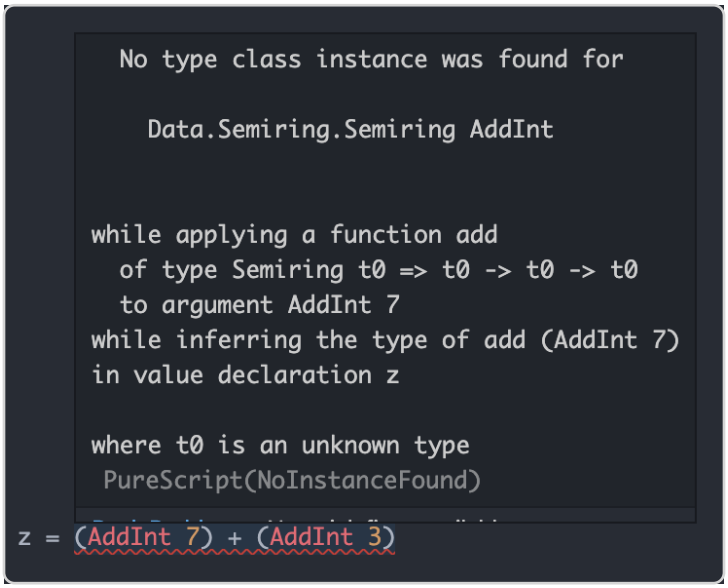
z :: Int
z = x + y -- COMPILER ERROR!! ❶
```

❶ This will cause a Type Mismatch error. `x` is `Int` and `y` is `AddInt` and `+` expects the two Types to be the same.

We also cannot add two different `AddInts`:

```
z :: AddInt
z = (AddInt 7) + (AddInt 3) -- COMPILER ERROR!!
```

If you try you'll get the following compiler error:



The most pertinent portion of the error is:

```
  No Typeclass instance was found for

    Data.Semiring.Semiring AddInt
```

This is confusing and will remain so until we get to the Abstract Algebra section where we'll learn a little bit about `Semirings`. For now let's just accept that we're missing an instance to add two `AddInts`.

For the curious, here's the instance that allows us to add and multiply, which are the operations of a `Semiring`, for `AddInt`:

```
instance semiringAddInt :: Semiring AddInt where
  add (AddInt x) (AddInt y) = AddInt (x + y)
  zero = AddInt 0
  mul (AddInt x) (AddInt y) = AddInt (x * y)
  one = AddInt 1
```

While `newtype` allows us to sidestep the Orphaned Instance problem, it is not without its downsides. Many times we need to implement the underlying instances for our `newtype` using `derive newtype`:

```
import Data.Newtype (class Newtype)

newtype AddInt = AddInt Int
derive instance newtypeAddInt :: Newtype AddInt _ ❶
derive newtype instance semiringAddInt :: Semiring AddInt ❷
derive newtype instance ringAddInt :: Ring AddInt ❸
derive newtype instance commutativeRingAddInt :: CommutativeRing AddInt ❹
derive newtype instance euclideanRingAddInt :: EuclideanRing AddInt ❹
```

❶ We must derive the `Newtype` instance for our Type before we can leverage `derive newtype`.

❷ Now, although we don't understand what a Semiring is yet, we can still leverage the underlying implementation for `Int`. This definition lets us Add and Multiply `AddInts`.

❸ This lets us Subtract `AddInts`.

❹ These let us Divide `AddInts`. `CommutativeRing` is the Superclass of `EuclideanRing` so we have to implement both since `EuclideanRing` gives us the Divide Operation.

Luckily for us, `derive newtype` reduces our work greatly. Unfortunately, we will probably find ourselves adding to this list as we use `AddInt` in different situations.

6.11. Instance Dependencies

Instances can also be constrained, i.e. an Instance depends on one or more Instances:

```
data Maybe = Nothing | Just a

instance showMaybe :: Show a => Show (Maybe a) where
  show Nothing  = "Nothing"
  show (Just x) = "(Just " <> show x <> ")" ❶
```

❶ The call to `show x` is why `a` must be an instance of `Show` since `x's` Type is `a`.

This instance is dependent on `Show`, i.e. we can show `Maybe a` as long as we can show `a`.

Here is another example:

```
data Either a b = Left a | Right b

instance showEither :: (Show a, Show b) => Show (Either a b) where ❶
  show (Left x)  = "(Left " <> show x <> ")" ❷
  show (Right y) = "(Right " <> show y <> ")" ❸
```

❶ We use Parentheses and commas to list multiple constraints.

❷ The call to `show x` is why `a` must be an instance of `Show`.

❸ The call to `show y` is why `b` must be an instance of `Show`.

You may have noticed that in instance definitions, we don't use `∀`. That's because it's implied.

6.12. Multi-Parametric Typeclasses

So far we've only seen Typeclasses with a single Type Parameter, usually `a`, but it is possible to have multiple:

```
import Data.List (List) ❶
import Data.List as List ❷
import Data.String.CodeUnits as String ❸

class Decapitate collection element where ❹
  decapitate :: collection -> Maybe {head :: element, tail :: collection}

instance decapitateList :: Decapitate (List a) a where ❺
  decapitate = List.uncons

instance decapitateString :: Decapitate String Char where ❻
  decapitate = String.uncons
```

❶ We import just the `List` Type so we don't need to type `List.List`.

❷ Then we import everything from `Data.List` in the Namespace `List`. This means that we must prefix all imports from this module with `List.`, e.g. `List.uncons`.

❸ This is the non-Unicode `String` module.

❹ A Class for removing the head `element` from a `collection`.

❺ The `List` instance where we leverage `List's uncons` function. `a` is the element of our `List`.

❻ The `String` instance where we leverage `String's uncons` function. `Char` is the element of our `String`.

`collection` and `element` are the Type Parameters to the `Decapitate` Typeclass. `collection` is the Type of collection and `element` is the Type of each element within that collection.

`decapitate` removes the head of the `collection` and returns it and the remaining elements in a `Record`. Since this can fail, it returns a `Maybe`.

The `List` instance is said to be ***Polymorphic*** in 2nd Type Parameter, i.e. Type `a`. Remember that ***Poly*** means ***many*** and ***morph*** means ***shape***:

```
instance decapitateList :: Decapitate (List a) a where
  decapitate = List.uncons
```

The 2nd Type Parameter, `a`, can take on many shapes, i.e. `a` can unify with many Types.

In contrast, the `String` instance is ***Monomorphic*** in the 2nd Type Parameter, i.e. `Char`. Remember that ***Mono*** means ***one*** and ***morph*** means ***shape***:

```
instance decapitateString :: Decapitate String Char where
  decapitate = String.uncons
```

The 2nd Parameter, `Char`, is a single, fixed Type.

6.13. Functional Dependency

Returning to our `String` instance for `Decapitate`:

```
import Data.String.CodeUnits as String

instance decapitateString :: Decapitate String Char where
  decapitate = String.uncons
```

This is the non-Unicode version of `String`. But what if we want to support Unicode also:

```
import Data.String.CodePoints (CodePoint)
import Data.String as StringUnicode

instance decapitateStringUnicode :: Decapitate String CodePoint where
  decapitate = StringUnicode.uncons
```

You might think that this is a case of Overlapping Instances, but it is NOT. `Decapitate String Char` is different than `Decapitate String CodePoint`.

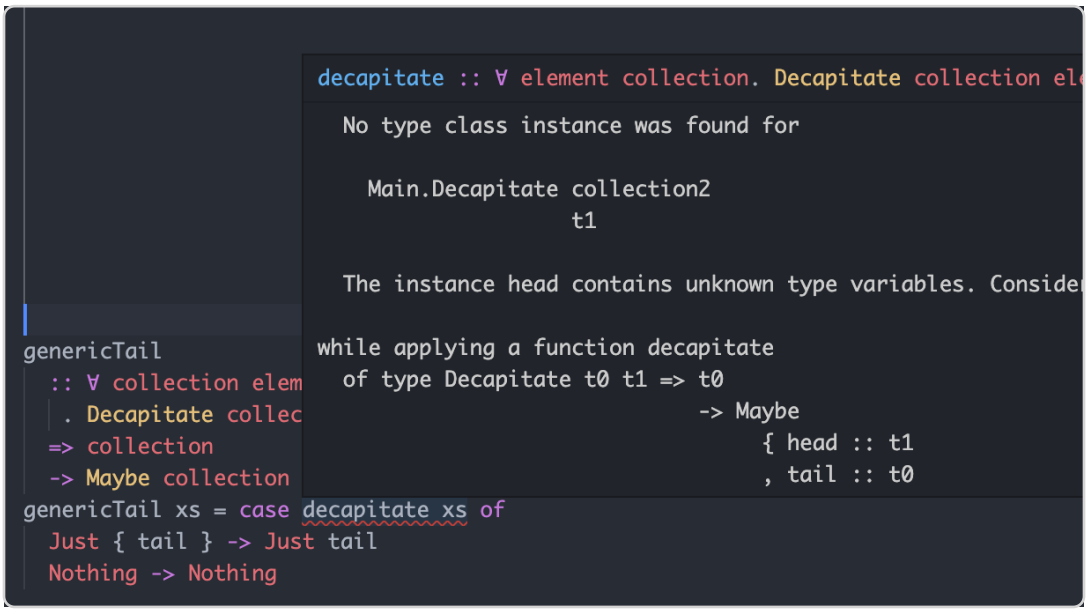
Let's keep these implementations in mind as we look at a function to return the tail by leveraging instances like these:

```
genericTail
  :: ∀ collection element
   . Decapitate collection element ❶
  => collection
  -> Maybe collection
genericTail xs = case decapitate xs of -- COMPILER ERROR!!
  Just { tail } -> Just tail ❷
  Nothing -> Nothing
```

❶ We constrain our `collection` and `element` to be part of a `Decapitation` instance.

❷ We Pattern Match `tail` from the `Record`.

Here is the compiler error:



Notice the Types in the error message, `collection2` and `t1`. The Type, `collection2`, is clearly our `collection`. And `t1` is clearly `element`. But why can't it find an instance?

In `genericTail`, the `element` Type isn't provided as a Parameter to the function, only `collection` is. (`element` only shows up in the Constraint, but not as a Parameter.) But the compiler needs both `collection` and `element` to pick the correct instance implementation of `Decapitate` to use.

In our example above, we have the Unicode and non-Unicode versions of `Decapitate` where the `collection` is `String` in both cases. But without knowing the `element` Type, we cannot decide which one to use and neither can the compiler.

There are 2 ways to fix this problem. One is a bit hacky and the other involves Functional Dependencies.

First, let's look at the hacky way. We need to provide the function an element of the appropriate Type so that the compiler can determine which instance to use for `Decapitate`:

```
genericTail
  :: ∀ collection element
   . Decapitate collection element
  => element ❶
  -> collection
  -> Maybe collection
genericTail _ xs = ❷
  case (decapitate xs :: Maybe {head :: element, tail :: collection}) of ❸
    Just { tail } -> Just tail
    Nothing -> Nothing
```

❶ `element` is now passed to the function. The compiler will have concrete Types for both `collection` and `element`, which means it can now decide which instance to call for `decapitate xs`.

❷ We're not using the `element` parameter. It's only there for its Type. So we don't bother to name it and use `_`.

❸ For some reason (possibly a bug), the compiler requires an explicit Type definition here.

When we want to use `genericTail`, we are burdened with coming up with an inhabitant of our `element` Type:

```
t :: Maybe String
t = genericTail 'c' "abc" ❶
```

❶ Provide the `Char`, `'c'`, as our `element` type. This exemplar lets the compiler know that we want it to use the instance, `decapitateString`.

And for Unicode:

```
tu :: Maybe String
tu = genericTail (codePointFromChar 'c') "abc" ❶
```

❶ `element` is now `CodePoint`. This exemplar lets the compiler know that we want it to use the instance, `decapitateStringUnicode`.

This is definitely a hacky way, but it works. But can we do better than this?

Seems that we're just picking a random value just to define the Type for `element`. There's got to be a cleaner way of doing this without picking a random value. While this isn't a big deal when we're talking about providing a Primitive Type, it's a completely different story when our type is a complex, nested `Record`. Coming up with such a `Record` would not be trivial.

As it turns out, there is a cleaner way to communicate to the compiler what Type our `element` should be:

```
import Type.Proxy (Proxy) ❶

genericTail
  :: ∀ collection element
   . Decapitate collection element
  => Proxy element ❷
  -> collection
  -> Maybe collection
genericTail _ xs = case (decapitate xs :: Maybe {head :: element, tail ::
collection}) of ❸
  Just { tail } -> Just tail
  Nothing -> Nothing
```

❶ `Proxy` is in its own package, `purescript-proxy`, which means `proxy` needs to be added to your Dependencies in `spago.dhall`.

❷ We don't have to pick some random value anymore. We just have to provide a `Proxy` for the Type.

❸ We still disregard the `Proxy` argument.

We can now provide a `Proxy` of the appropriate Type:

```
import Type.Proxy (Proxy(..)) ❶

t :: Maybe String
t = genericTail (Proxy :: Proxy Char) "abc"

tu :: Maybe String
tu = genericTail (Proxy :: Proxy CodePoint) "abc"
```

❶ We need the `..` to include the Constructors of `Proxy`, which is also called `Proxy`.

What's going on here isn't so obvious, so let's break it down.

First, let's look at the definition of `Proxy`:

```
data Proxy a = Proxy
```

The first thing you may notice is that the `a` Type Parameter doesn't show up on the right-hand side of the equation. We've seen this before with `Maybe a` and `Nothing`. But at least with `Maybe`, the `a` Type Parameter is used by the `Just` Data Constructor.

Here it's never used. When that's the case, the Type `a` is called a ***Phantom Type***.

Remember working with `Nil` and how the compiler couldn't determine the `a` Type for the `List`. To solve this problem, we had to explicitly specify the Type:

```
log $ show $ index (Nil :: List Unit) 0
```

Well, we have the same issue with the `Proxy` Data Constructor. We must provide an explicit Type and it's that Type that unifies with `element`. This unification lets the compiler know the Type for `element`:

```
import Type.Proxy (Proxy(..)) ❶

t :: Maybe String
t = genericTail (Proxy :: Proxy Char) "abc" ❶

tu :: Maybe String
tu = genericTail (Proxy :: Proxy CodePoint) "abc" ❷

genericTail
  :: ∀ collection element
   . Decapitate collection element
  => Proxy element ❶ ❷
  -> collection
  -> Maybe collection
genericTail _ xs = case (decapitate xs :: Maybe {head :: element, tail ::
collection}) of
  Just { tail } -> Just tail
  Nothing -> Nothing
```

❶ Type `Proxy Char` unifies with `Proxy element` in `genericTail`. Therefore, `element` unifies with `Char`.

❷ Type `Proxy CodePoint` unifies with `Proxy element` in `genericTail`. Therefore, `element` unifies with

`CodePoint`.

Well, that concludes our hacky solution. It was a bit complex and cumbersome.

Hopefully, a feature called ***Functional Dependencies*** can simplify this mess. Well, they do and they don't.

Here's how we could write the Typeclass `Decapitate` using Functional Dependencies:

```
class Decapitate collection element | collection -> element where ❶
  decapitate :: collection -> Maybe {head :: element, tail :: collection}
```

❶ The portion after the vertical bar specifies the Functional Dependency between `collection` and `element`.

The Functional Dependency defines that for each `collection` Type there can be ONE AND ONLY ONE `element` Type.

This means that when the compiler only knows about the `collection` Type, it can safely infer the `element` Type since there can only be one possibility. But it also means, that you're limited in what instances you can create.

Using Functional Dependencies, all ambiguity is eliminated allowing us to write a simpler version of `genericTail`:

```
genericTail
  :: ∀ collection element
   . Decapitate collection element
  => collection ❶
  -> Maybe collection
genericTail xs = case decapitate xs of ❷
  Just { tail } -> Just tail
  Nothing -> Nothing
```

❶ We no longer need the extra Parameter.

❷ Now the compiler doesn't need an explicit Type on this line.

This seems to solve all of our previous problems… except for that limitation I mentioned earlier:

```
instance decapitateString :: Decapitate String Char where
  decapitate = String.uncons

-- OVERLAPPING INSTANCE ERROR!!
instance decapitateStringUnicode :: Decapitate String CodePoint where
  decapitate = StringUnicode.uncons
```

Why do you think we get an Overlapping Instance error here?

Remember what our Functional Dependency says. Knowing the `collection` will dictate the `element` with no ambiguity. But when we introduce 2 different cases for the `collection` of `String`, we now have ambiguity, hence the Overlapping Instance error.

So is there a solution for this dilemma?

Actually, there is. We simply reverse the Functional Dependency:

```
class Decapitate collection element | element -> collection where ❶
  decapitate :: collection -> Maybe {head :: element, tail :: collection}
```

❶ Reversed dependency.

Now our `String` instances don't overlap since the `elements` are different. With the Functional Dependency reversed, the compiler starts with either `Char` or `CodePoint` and based on that it can figure out the `collection`, which, in both cases, is `String`.

And this is okay, because it's the `element` that dictates the Instance it'll use for `decapitate` and there's no ambiguity between `Char` and `CodePoint`.

But we're forced to add back in our explicit Type Signature for `genericTail`:

```
genericTail
  :: ∀ collection element
   . Decapitate collection element
  => collection
  -> Maybe collection
genericTail xs = case (decapitate xs :: Maybe {head :: element, tail ::
collection}) of ❶
  Just { tail } -> Just tail
  Nothing -> Nothing
```

❶ With the Functional Dependency reversed, we need to explicitly specify the Type for the compiler.

The explicit Type Signature isn't too bad, but now we have a different Overlapping Instance problem in the

following scenario:

```
instance decapitateString :: Decapitate String Char where
  decapitate = String.uncons

instance decapitateStringUnicode :: Decapitate String CodePoint where
  decapitate = StringUnicode.uncons

-- OVERLAPPING INSTANCE ERROR!!
instance decapitateList :: Decapitate (List a) a where
  decapitate = List.uncons
```

But why?

Since `a` can unify with both `Char` and `CodePoint`, the compiler cannot unambiguously determine if the `collection` is `List a` or `String`.

Here's another case where having the Functional Dependency reversed causes a problem:

```
instance decapitateList :: Decapitate (List a) a where
  decapitate = List.uncons

-- OVERLAPPING INSTANCE ERROR!!
instance decapitateArray :: Decapitate (Array a) a where
  decapitate = Array.uncons
```

Once again, the compiler cannot determine for any `a`, whether the collection is a `List` or an `Array`. Remember, we reversed the dependency so the compiler will start with the `element` Type to determine the `collection` Type.

When the Functional Dependency was `| collection → element`, the above example would work just fine, but with it reversed we're overlapping and with it forwards our `Strings` overlap.

There is, however, one more trick up our sleeves. We can remove this error using ***Instance Chaining***:

```
instance decapitateList :: Decapitate (List a) a where
  decapitate = List.uncons
else instance decapitateArray :: Decapitate (Array a) a where ❶
  decapitate = Array.uncons
```

❶ Added keyword `else`.

This tells the compiler what order to pick instances in. There may be multiple valid cases, but the first one

will always be picked removing any ambiguity.

But now `decapitateArray` will NEVER be picked because `List` will always match:

```
ta :: Maybe (Array String) -- COMPILER ERROR!! ❶
ta = genericTail ["abc"]

tl :: Maybe (List String)
tl = genericTail ("abc" : Nil)
```

❶ Could not match type `List` with type `Array`.

On the surface, it may appear that Functional Dependencies don't work well and require a lot of work. But in practice, they are usually the easiest and best solution.

The example with non-Unicode and Unicode `Strings` both needing to be an instance of the same Class is pretty rare and quite contrived to demonstrate some problems you'll run into and some possible solutions.

And in the rare case where Functional Dependencies fail us due to crazy requirements, we can always use `Proxy` to force the compiler to pick the instance we want.

Chapter 7. Coding Typeclasses

It's always good to take something we've just learned and play with it to get a good feeling for the concepts. Please don't just read this section, but make sure you try to code as much as possible.

7.1. Coding Preparation

Working in the same project as we did for our earlier coding sessions, let's simply add a file for this Chapter called `Ch7a`. (The `a` is important since we'll create a `b` file later on in this Chapter.)

Next, write the module definition for `Ch7a` without looking back on what we've done before. See if you can remember it. If not, no worries. It's just good to challenge your memory. If you have to cheat and look, don't copy and paste it. Typing it over will help cement it into your mind.

7.2. Maybe Data Type

Create the definition for `Maybe`. You've seen it more than a few times, so it should be easy.

If you struggle with this, that's a good thing. That means trying to write it is worth doing.

Try to define it before moving on.

7.3. Code for `Maybe` Data Type

Your file should look like:

```
module Ch7a where

data Maybe a = Nothing | Just a ❶
```

❶ The order of the Data Constructors could be reversed, but remember the `Ord` Typeclass derivations we talked about in the last Chapter will order values based on the order of the Data Constructors. So, I've chosen to put `Nothing` first here to cause `Nothings` to sort to the top of `Lists` and `Arrays` of `Maybes`.

7.4. Writing `Eq` for `Maybe`

Add the following imports:

```
import Prelude (Unit, show, discard, (==), ($)) ❶

import Data.Eq (class Eq) ❷
import Effect (Effect) ❸
import Effect.Console (log) ❸
```

❶ Some things we'll need from `Prelude`. Remember, `discard` is implicitly used in the `do` syntax (more on this later in the book).

❷ We need to import the `Eq` Typeclass. Note the `class` keyword.

❸ These imports are for our debugging code, viz. the `test` function.

Edit your `Main.purs` to look like:

```
module Main where

import Prelude
import Effect (Effect)

import Ch7a as Ch7a ❶

main :: Effect Unit
main = Ch7a.test ❷
```

❶ Change from `Ch5` to `Ch7a`.

❷ Change from `Ch5.test` to `Ch7a.test`.

Add the following Function to test our code in `Ch7a.purs`:

```
test :: Effect Unit
test = do
  log $ show $ Just 5 == Just 5 -- COMPILER ERROR!! ❶
  log $ show $ Just 5 == Just 2 ❷
  log $ show $ Just 5 == Nothing ❸
  log $ show $ Nothing == Just 5 ❹
  log $ show $ Nothing == (Nothing :: Maybe Unit) ❺ ❻
```

❶ Prints `true`.

❷ Prints `false`.

❸ Prints `false`.

❹ Prints `false`.

❺ Prints `true`.

❻ Once again, the compiler cannot figure out the `a` in `Maybe a` since the Data Constructor, `Nothing`, doesn't take a Type Parameters. So we have to explicitly specify `a`. We chose `Unit` for `a` because `a` must have an `Eq` instance, which `Unit` does.

The compiler error is expected here. We'll fix it soon enough.

Notice that we only had to explicitly specify the Type on one side of `==`. That's because `==` is an alias for `eq` which expects both Parameters to be of the same Type. Once we let the compiler know that the right-hand side is of Type `Maybe Unit`, it can figure out that the left-hand side is of the same Type.

A friendly reminder, you run your program with the following command (once it compiles):

```
npx spago run
```

Now you're ready to write the `Eq` instance for `Maybe`. We want to write the instance by hand for practice even though we can derive `Eq`.

Use `Pursuit` to look up the definition of the `Eq` Typeclass.

Please give it a try before reading further.

7.5. Code for `Eq` for `Maybe`

First, we code the instance definition:

```
instance eqMaybe :: Eq (Maybe a) where ❶
```

❶ We use canonical naming here for the Instance Name, first the Typeclass name and then the Type name, viz. `eqMaybe`.

Next, we code up the Methods. Based on `Pursuit`, `Eq` only has one Method, `eq`. First, let's code up the trivial case:

```
instance eqMaybe :: Eq (Maybe a) where
  eq Nothing Nothing = true ❶
```

❶ This is a trivial case.

Next, we should code up the `Just` case:

```
instance eqMaybe :: Eq (Maybe a) where
  eq Nothing Nothing   = true
  eq (Just x) (Just y) = ??? ❶
```

❶ Notice that we have an x and a y to work with.

The Pattern Matching tells us we have 2 Justs but we don't want to make all Justs equal like we did with Nothing. Why?

Well, looking back at our test code, we wouldn't expect Just 1 to be equal to Just 2. The reason for this is because 1 is NOT equal to 2. This means that we need to compare the values contained within the Justs:

```
instance eqMaybe :: Eq (Maybe a) where
  eq Nothing Nothing   = true
  eq (Just x) (Just y) = x == y ❶
```

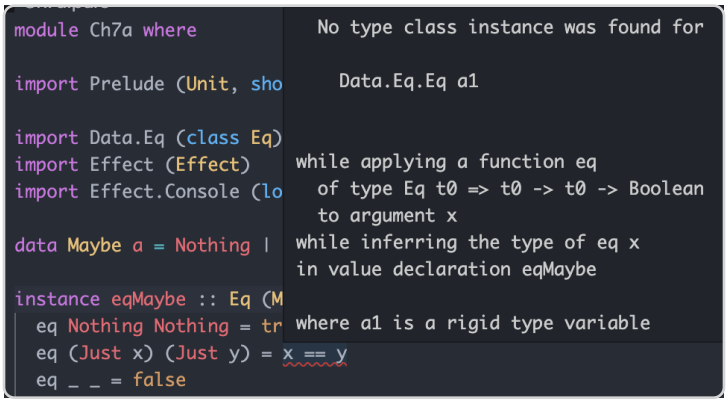
❶ Justs are equal if what they wrap are equal.

And finally, we code the Catch-All case:

```
instance eqMaybe :: Eq (Maybe a) where
  eq Nothing Nothing   = true
  eq (Just x) (Just y) = x == y -- COMPILER ERROR!!
  eq _ _               = false ❶
```

❶ This Catch-All case matches one Just and a Nothing in either order.

The compiler error is as follows:



The pertinent portion of the error is:

```
No type class instance was found for

    Data.Eq.Eq a1
```

`Data.Eq.Eq` is the fully-qualified Typeclass name. `Data.Eq` is the Module name and the second `Eq` is the Typeclass.

Notice the underlined part in the editor is `x == y`. That makes sense based on the error message. It cannot determine an `Eq` Instance for `a1`. But what's `a1`? We don't have an `a1`.

Remember, the compiler adds numbers to the end of the Types to distinguish different variables with the same names. So we need to look for an `a` in our code:

```
instance eqMaybe :: Eq (Maybe a) where ❶
  eq Nothing Nothing   = true
  eq (Just x) (Just y) = x == y -- COMPILER ERROR!!
  eq _ _               = false
```

❶ This is the ONLY `a`.

We get an error when trying to compare `x` and `y` for equality using `==` which is just an Operator Alias for the `eq` Method in the `Eq` Typeclass. The compiler wants to be able to use the appropriate `Eq` instance on this line of code.

Both `x` and `y` are Type `a` since they were constructed with `Just`, which takes an `a`. You can also say they were destructured with `Just`, which gives us a value of Type `a`.

The compiler is complaining that there is NO Instance for our `a`. So we need to provide one. But we cannot just write one since even we don't know what Type `a` will be when our Typeclass gets used.

Therefore, we only have one choice. We must REQUIRE that `a` comes with its own `Eq` instance. And we accomplish this with a Constraint:

```
instance eqMaybe :: Eq a => Eq (Maybe a) where ❶
  eq Nothing Nothing   = true
  eq (Just x) (Just y) = x == y
  eq _ _               = false
```

❶ We require `a` to have an `Eq` instance by constraining `a`.

7.6. Writing `Ord` for `Maybe`

Remember that the `Ord` class lets us determine the ordering of values in a Type. So operations like `<`, `>`, `<=`

and >= all require an Instance of the Ord Typeclass.

Look up Ord in Pursuit to see what Methods are part of the Ord Typeclass.

Next, update your Prelude imports:

```
import Prelude (Unit, show, discard, (==), ($), (<), (>), (<=), (>=))
```

And let's import from Data.Ord that we'll need for implementing an Instance for Ord:

```
import Data.Ord (class Ord, Ordering(..), compare)
```

And add the following lines to your test function:

```
  log "------------------" ❶
  log $ show $ Just 1 < Just 5 -- COMPILER ERROR!! ❷
  log $ show $ Just 5 <= Just 5 ❸
  log $ show $ Just 5 > Just 10 ❹
  log $ show $ Just 10 >= Just 10 ❺
  log $ show $ Just 99 > Nothing ❻
  log $ show $ Just 99 < Nothing ❼
```

❶ This will make reading the output of our test code easier since the previous tests also print out Booleans.

❷ Prints true.

❸ Prints true.

❹ Prints false.

❺ Prints true.

❻ Prints true.

❼ Prints false.

Once again, we are expecting this compiler error since we haven't implemented Ord yet, so we cannot use any of the comparison operators here.

Please consult Pursuit to refresh your memory about the Ordering Type since it's used in the Ord Typeclass.

If you struggle with writing this Typeclass, then proceed to the Hint section that follows. But give it try before giving up.

7.7. Hint for Ord for Maybe

The Ordering Type is as follows:

```
data Ordering = LT | GT | EQ
```

The compare Method of the Ord Typeclass is defined as:

```
compare :: a -> a -> Ordering
```

That means that you'll need to compare the 2 Maybes, probably using Pattern Matching, and return either LT, GT or EQ depending on the values. This means that our underlying values will need to be comparable too.

Give it another try.

7.8. Code for Ord for Maybe

First we write the instance definition:

```
instance ordMaybe :: Ord a => Ord (Maybe a) where
```

We have constrained the a to also have an Ord instance just like we did for Eq in anticipation of comparing 2 Justs.

Next, we code the trivial case:

```
instance ordMaybe :: Ord a => Ord (Maybe a) where
  compare Nothing Nothing     = EQ
```

And now the Just case:

```
instance ordMaybe :: Ord a => Ord (Maybe a) where
  compare Nothing Nothing   = EQ
  compare (Just x) (Just y) = ??? ❶
```

❶ We cannot use an operator like we did in Eq.

So the question marks are where we stop and ask how do we compare the underlying values, x and y. In the Eq Typeclass, we used the == operator. We could certainly use the < and > operators:

```
instance ordMaybe :: Ord a => Ord (Maybe a) where
  compare Nothing Nothing   = EQ
  compare (Just x) (Just y) =
    if x > y then GT else if y < x then LT else EQ (1)
```

(1) Nested if under the first `else`.

But this is unnecessary, since the greater-than and less-than operators are written in terms of `compare` anyway. We'll see how this is true later when we write code for `>=`.

So let's simply avoid the middleman and just use `compare`:

```
instance ordMaybe :: Ord a => Ord (Maybe a) where
  compare Nothing Nothing   = EQ
  compare (Just x) (Just y) = compare x y (1)
```

(1) Notice that the order of `x` and `y` matches the same order as `Just x` and `Just y`.

And finally, we handle the 2 cases where we're comparing a Nothing and a Just:

```
instance ordMaybe :: Ord a => Ord (Maybe a) where
  compare Nothing Nothing   = EQ
  compare (Just x) (Just y) = compare x y
  compare Nothing _         = LT (1)
  compare _ Nothing         = GT
```

(1) Remember that we defined the `Nothing` Data Constructor BEFORE we defined the `Just` Data Constructor, hence, `Nothing` is less than `Just a` and conversely, `Just a` is greater than `Nothing`. This follows the rules used by the compiler when it derives `Ord`.

7.9. Writing >=

You may be wondering why we're writing `compare` which returns `LT`, `GT` or `EQ` and yet we're testing our code with `<`, `>`, `<=` and `>=`.

To better understand how these are related, look on `Pursuit` to see what function `>=` is an Operator Alias for. Then look at that function's Type Signature.

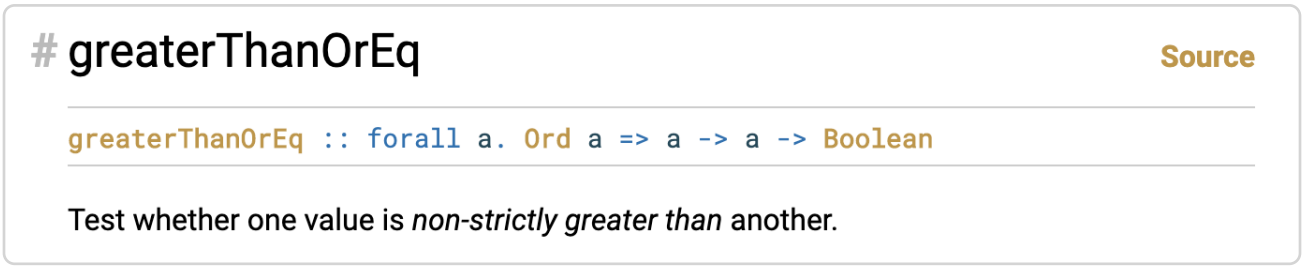
Finally, write that function and define the operator before continuing.

7.10. Hint for >=

Here's the definition of `>=`:



We can see that this operator is an alias for is `greaterThanOrEq`. Let's look up its Type Signature, which is directly above the Operator definition on the same page:



From this Type Signature, we can see that `greaterThanOrEq` doesn't expect a number like we're used to giving it when we use `>=`. Instead, it expects an `a` that's constrained to have an instance of `Ord`.

So keep all of this in mind and try again before reading ahead.

7.11. Code for >=

Let's start with the Type Signature:

```
greaterThanOrEq :: ∀ a. Ord a => a -> a -> Boolean
```

Next, for the implementation:

```
greaterThanOrEq :: ∀ a. Ord a => a -> a -> Boolean
greaterThanOrEq x y = ??? ❶
```

❶ We know that this should leverage `compare` somehow.

We know that `compare` returns `LT`, `GT` and `EQ`. All we have to do is call `compare` and check to see if it's either `GT` or `EQ`:

```
greaterThanOrEq :: ∀ a. Ord a => a -> a -> Boolean
greaterThanOrEq x y = cmp == GT || cmp == EQ where cmp = compare x y ❶ ❷
```

❶ We need to add `(||)` to our `Prelude` import list.

❷ We are using `where` here so we don't actually call `compare` twice.

And finally, we just need to add the Operator Alias, looking back at the Pursuit documentation to see what the Associativity and Precedence are.

```
infixl 4 greaterThanOrEq as >= ❶ ❷
```

❶ We need to remove (>=) from our Prelude import list since we're defining our own operator.

❷ The Associativity is Left so we use infixl and the Precedence is 4.

7.12. Writing Show for Maybe

Use Pursuit to look up the definition of the Show Typeclass.

Add the following import:

```
import Data.Show (class Show) ❶
```

❶ Optionally, you could remove show from your Prelude import list and add it here. Prelude imports it for you from Data.Show. Either way is valid.

Then add the following to your test code:

```
log $ show $ Just "abc" ❶
log $ show $ (Nothing :: Maybe Unit) ❷ ❸
```

❶ Prints (Just "abc").

❷ Once again, we had to explicitly specify the Type.

❸ Prints Nothing.

Now write the Show instance for Maybe before moving on. Write it by hand, i.e. do not use genericShow. We'll do that soon enough.

7.13. Code for Show for Maybe

First we write the instance definition line:

```
instance showMaybe :: Show (Maybe a) where
```

Next, we write the show Method for the trivial case, i.e. for Nothing:

```
instance showMaybe :: Show (Maybe a) where
  show Nothing = "Nothing"
```

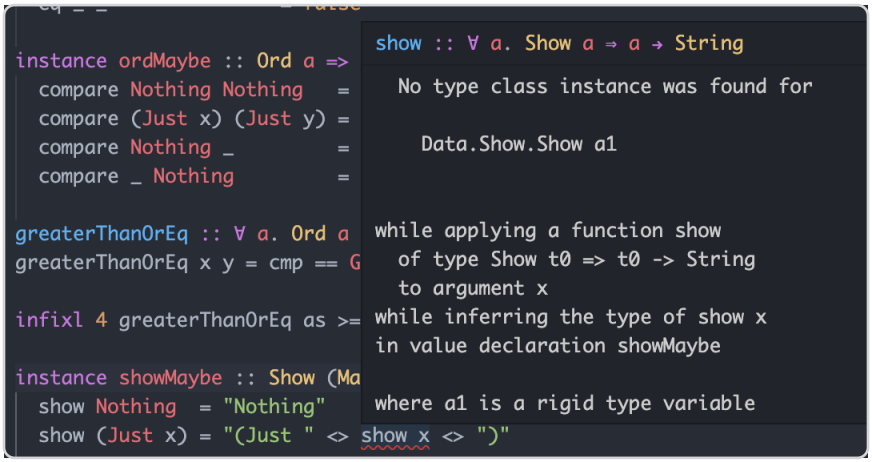
And finally, we write the Just case:

```
instance showMaybe :: Show (Maybe a) where
  show Nothing = "Nothing"
  show (Just x) = "(Just " <> show x <> ")" -- COMPILER ERROR!! ❶ ❷
```

❶ We need to add (<>) to our Prelude import list.

❷ Showing x is a problem. Can you think of why?

Here's the compiler error:



Notice that show x is underlined in the editor and the compiler error says that a1, which corresponds to our a in Maybe a, doesn't have a Show instance.

We've told the compiler to show x and yet x can be ANY a. But it can't by just ANY a. It must be a showable a. So we add that Constraint:

```
instance showMaybe :: Show a => Show (Maybe a) where ❶
  show Nothing = "Nothing"
  show (Just x) = "(Just " <> show x <> ")"
```

❶ We've added a Constraint on a to require that it too is showable.

Now a isn't just ANY a, but ANY a that also has a Show instance.

7.14. Deriving Eq, Ord and Show for Maybe

Now that we've written these instances by hand, which we did to better understand how they work, we can comment them out and replace them with deriving statements instead.

Once you comment out your Show instance, you'll get a warning on the Prelude import line. When you hover your cursor over that line, you'll see that the import for (<>) is no longer necessary.

Go ahead and remove it now.

Try to see if you remember how to derive these instances without looking back or ahead. If you cannot, then just look ahead to the Hint section.

7.15. Hint for Deriving

If you cannot remember the deriving syntax, which is normal at first, then I'd like to take this moment to advocate for why we're doing all of these exercies even though we talked about all of this in the previous Chapter.

I know I had to look up the syntax for things like Operator Alias and Deriving several times before I could just type them without thinking. To be honest, Operator Alias still gets me sometimes.

By doing these exercices and not just reading them, we're solidifying our understanding with experience that will help us when we're coding something real.

And now on to the syntax of derive:

```
derive instance eqMaybe :: ???
```

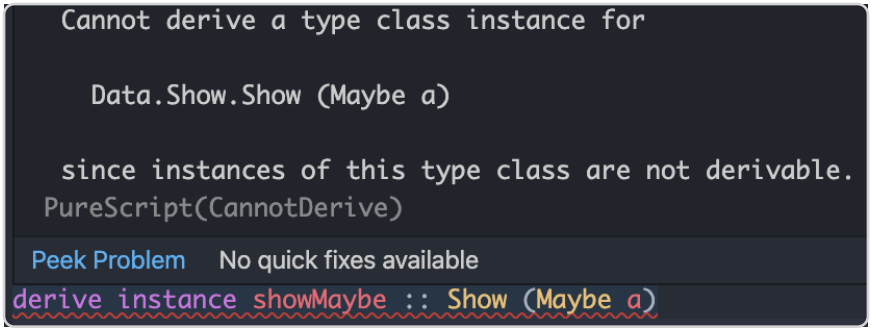
See if this helps and try again. If you get stuck on Show, look to the next Hint section.

7.16. Hint for Deriving Show

You may or may not remember that you cannot do the following:

```
derive instance showMaybe :: Show (Maybe a) -- COMPILER ERROR!!
```

Here is the error you'll get:



The compiler say that it ***Cannot derive a type class for Data.Show.Show***. And it says below that it cannot do so because ***instances of this type class are not derivable***.

In the previous Chapter, we talked about using `genericShow`. But to do this, our Type must have an Internal Representation of the Type so that `genericShow` can use that representation to display the value.

This is done with the following imports:

```
import Data.Generic.Rep (class Generic)
import Data.Show.Generic (genericShow)
```

Next, we want to build using:

```
npx spago build
```

And we get the following error:

```
[error] Some of your project files import modules from packages that are not
in the direct dependencies of your project.
To fix this error add the following packages to the list of dependencies in
your config:
- maybe
- tuples
```

We need to add these packages to `spago.dhall` as Direct Dependencies of our project:

```
{-
Welcome to a Spago project!
You can edit this file as you like.
-}
{ name = "my-project"
, dependencies =
  [ "console"
  , "effect"
  , "lists"
  , "maybe" ❶
  , "prelude"
  , "psci-support"
  , "tuples" ❶
  ]
, packages = ./packages.dhall
, sources = [ "src/**/*.purs", "test/**/*.purs" ]
}
```

❶ Added to maintain alphabetic order.

Try again:

```
npx spago build
```

Finally, we create an instance of `Generic` for our Type and then we can manually write the `Show` instance and simply just use `genericShow`.

Try to do that before moving on.

7.17. Code for `Eq`, `Ord` and `Show` for `Maybe`

Let's start with `Eq`:

```
derive instance eqMaybe :: Eq a => Eq (Maybe a)
```

And now we derive `Ord`:

```
derive instance ordMaybe :: Ord a => Ord (Maybe a)
```

Before we can do `Show`, we have to do `Generic` so we can leverage `genericShow`:

```
derive instance genericMaybe :: Generic (Maybe a) _ ❶
```

❶ Don't forget the second UNSPECIFIED Type Parameter is always _.

The underscore Type Parameter is filled in by the compiler.

And finally, we can manually write the Show instance:

```
instance showMaybe :: Show a => Show (Maybe a) where ❶
  show = genericShow
```

❶ Don't forget the Constraint for a.

The final code should look something like:

```
data Maybe a = Nothing | Just a
derive instance eqMaybe :: Eq a => Eq (Maybe a)
derive instance ordMaybe :: Ord a => Ord (Maybe a)
derive instance genericMaybe :: Generic (Maybe a) _

instance showMaybe :: Show a => Show (Maybe a) where
  show = genericShow
```

Now rerun your tests and see if the output is the same as your manual versions.

7.18. Deriving Eq, Ord and Show for Either

Create the Either Data Type and then derive Eq and Ord, and write Show using genericShow for Either.

Use what you've done with Maybe as a template for Either.

Remember there are some minor differences between Maybe and Either, e.g. Either has 2 Type Parameters a and b.

Try to code it before proceeding.

7.19. Writing Eq, Ord and Show for Either

Let's create the Data Type first:

```
data Either a b = Left a | Right b
```

Next, let's write the Eq instance:

```
derive instance eqEither :: (Eq a, Eq b) => Eq (Either a b) ❶
```

❶ We have to constrain BOTH a and b using this special syntax with Parentheses and Commas.

Then we write Ord:

```
derive instance ordEither :: (Ord a, Ord b) => Ord (Either a b)
```

And before we can write Show, we need to derive Generic so we can use genericShow:

```
derive instance genericEither :: Generic (Either a b) _
```

And finally, we write the Show instance leveraging genericShow:

```
instance showEither :: (Show a, Show b) => Show (Either a b) where
  show = genericShow
```

Let's add some test code:

```
log $ show $ Left "left" -- COMPILER ERROR!! ❶
log $ show $ Right (Just 42)
```

❶ This compiler error we've seen before.

Here's the gory details of the compiler error:

```
log $
log $  show :: ∀ a. Show a ⇒ a → String
log $
log $    No type class instance was found for
log "-
log $
log $       Data.Show.Show t2
log $
log $    The instance head contains unknown type variables. Consider adding a type annotation.
log $
log $  while checking that type forall a. Show a => a -> String
log $    is at least as general as type t0 -> t1
log "- while checking that expression show
log $    has type t0 -> t1
log $  in value declaration test
log "-
log $ show $ Left "left"
log $ show $ Right (Just 42)
```

Here the compiler gives us a Type of t2 which isn't very helpful at all.

```
instance showEither :: (Show a, Show b) => Show (Either a b) where ❶
  show = genericShow
```

❶ We have 2 Type Parameters here, a and b.

Looking at this definition, we could reasonably assume that t2 is for b since it's the 2nd parameter. But why does it care about b?

To see if we're on the right track, comment out the offending line and see if we get a similar error for the next line:

```
-- log $ show $ Left "left"
log $ show $ Right (Just 42) -- COMPILER ERROR!! ❶
```

❶ The same exact error.

Here's the new error:

```
log $ show :: forall a. Show a => a -> String
log $
log "-   No type class instance was found for
log $
log $       Data.Show.Show t2
log $
log $     The instance head contains unknown type variables. Consider adding a type annotation.
log $
log $   while checking that type forall a. Show a => a -> String
log "-    is at least as general as type t0 -> t1
log $   while checking that expression show
log $     has type t0 -> t1
log "-  in value declaration test
-- log
log $ show $ Right (Just 42)
```

Unfortunately, this too gives us t2 as the Type. So maybe our original assessment is wrong about t2 representing b since Right provides the compiler with b.

So both Left and Right gives us an error that suggests that we add a *Type Annotation*. But it's still not clear what to do.

Let's do a test by going to the REPL (Read-Evaluate-Print-Loop), but first comment out both test lines of code so our program will compile:

```
  -- `log $ show $ Left "left" -- COMPILER ERROR!!
  -- log $ show $ Right (Just 42)
```

Go to the terminal, making sure you're in your project directory and run:

```
npx spago repl
```

Then let's type the following commands at the prompt, >:

```
> import Ch7a ❶ ❷
> :t Left "left" ❸
forall t2. Either String t2 ❹
```

❶ Do NOT type > since that's the prompt of the REPL (Read-Evaluate-Print-Loop).

❷ We import our code so we have Either defined in the REPL.

❸ :t is short for :type, which prints the type of Left "left".

❹ This is the output of the :t command.

Notice how the Type is:

```
forall t2. Either String t2
```

The compiler figured out what a in Either a b was because we gave it Left "left" and it unified our String, "left", with a. But it doesn't know what b is.

It says that b or t2 as it calls it, can be ANY Type, hence the forall t2.

Let's try to find the Type for our second test:

```
> :t Right (Just 42) ❶
forall t1. Either t1 (Maybe Int) ❷
> ^D ❸
See ya!
()
```

❶ We request the type of the Right from our test code.

❷ This is the output of the :t command.

❸ Type Control-D to exit the REPL.

Strangely enough, it calls our b, t1 here, which is not the same name that the compiler error had, but that's okay. This tells us what we need to know.

When we give the compiler a Left, then it can infer the a but not the b. But when we give it a Right, it can infer the b but not the a.

This makes sense, since, in both cases, there just isn't enough information. Left doesn't inform us with

regards to b and Right doesn't tell us anything about a.

Let's change our test code to give it enough information.

There are 2 ways of doing this. We can add a Type Annotation like the compiler suggested:

```
log $ show (Left "left" :: Either _ Unit) <1> ❷ ❸
log $ show (Right (Just 42) :: Either Unit _) <1> ❷ ❹
```

❶ The _ is filled in by the compiler since we know it can infer that Type Parameter.

❷ Unit is used since it has a Show instance.

❸ We could replace this _ with String.

❹ We could replace this _ with (Maybe Int).

Another way to fix this, is to use variables, which is more closely aligned to what we'd do in our programs. We're not going to do this, but this is how that would work.

First, let's define a Type Alias:

```
type MyEitherVar = Either String (Maybe Int) ❶
```

❶ A horribly named Type Alias.

Next, lets define some values of Type MyEitherVar and show those:

```
  let x = Left "left" :: MyEitherVar ❶
      y :: MyEitherVar ❶
      y = Right $ Just 42
  log $ show x ❷
  log $ show y ❷
```

❶ We can define x and y using these 2 different syntaxes. The choice is yours.

❷ Both a and b are known to the compiler since both x and y are of Type MyEitherVar. So the compiler can infer the (Show a, Show b) Constraint from only having an x or a y.

At this point, we can clean up our imports:

```
import Prelude (Unit, discard, show, ($), (<), (<=), (==), (>), (>=)) ❶
import Data.Ord (class Ord) ❷
```

❶ Removed (<>) from the import list.

❷ Removed `Ordering(..)` and `compare`.

7.20. Creating Our Own Typeclass

Let's imagine that we have some things we'd like to encode as CSV (Comma-Separated Values).

A very efficient way to do this is to create a Typeclass. Let's call it `ToCSV` with a single Method called `toCSV`.

First though, let's make a new file called `Ch7b.purs` under the `src` directory with the following:

```
module Ch7b where

import Prelude

import Effect (Effect)
import Effect.Console (log)

test :: Effect Unit
test = do
  log "placeholder" ❶
```

❶ We're going to delete this line shortly. It's only here to keep the compiler happy for now.

And change `Main.purs` to be:

```
module Main where

import Prelude
import Effect (Effect)

import Ch7b as Ch7b ❶

main :: Effect Unit
main = Ch7b.test ❷
```

❶ Change import from `a` to `b`.

❷ Call `b's` test function.

Now, go ahead and write the Typeclass for `ToCSV` before moving on. If you get stuck, check out the Hint section.

7.21. Hint for ToCSV

When we think about encoding something, we first have to determine the Type for the encoding.

Our case is simply a CSV, which could easily be encoded with a `String`. But, we don't want to accidentally mix up any old `String` with our `CSV Strings`.

So first let's make a `newtype` for our `CSV` encoded string. Let's call it `CSV`.

And to help remind you of the syntax of Typeclasses, here's our `HasAddress` Typeclass from an earlier Chapter:

```
class HasAddress a where
  getAddress :: a -> Address
```

Take a second stab at writing both the `newtype` and the `class` before venturing further.

7.22. Code for ToCSV

First, let's create our `newtype`:

```
newtype CSV = CSV String
```

`CSV` is just a wrapper for a `String`, but has the added benefit of being Type Safe, i.e. the compiler will complain if we pass a `String` in its place.

Next, we'll write the class:

```
class ToCSV a where
  toCSV :: a -> CSV
```

The Method `toCSV` will take our Type, represented here by `a`, and encode it as a `CSV`.

7.23. Using ToCSV

Take the following Data Types and make an Instance of `ToCSV` for `Person`:

```
newtype FullName = FullName String

newtype Age = Age Int

data Occupation = Doctor | Dentist | Lawyer | Unemployed

data Person = Person
  { name :: FullName
  , age :: Age
  , occupation :: Occupation
  }
```

Give it a try and see how far you can get. If you're struggling, please consult the following Hint section.

7.24. Another Hint for ToCSV

First thing to realize is that it's easiest to implement `show` for `FullName`, `Age` and `Occupation` before attacking `Person's ToCSV` instance.

Give it another try before reading ahead.

7.25. Code for ToCSV

Let's write the `FullName` instance first:

```
newtype FullName = FullName String

instance showFullName :: Show FullName where
  show (FullName name) = name ❶
```

❶ We destructure `FullName` exposing `name`, which is Type `String`.

Then we can write the same for `Age`, but this time we're going to do it a little differently:

```
import Data.Newtype (class Newtype) ❶

newtype Age = Age Int
derive instance newtypeAge :: Newtype Age _ ❷
derive newtype instance showAge :: Show Age ❸
```

❶ We need to import `class Newtype` to derive `Newtype` for `Age`.

❷ The `Newtype` Typeclass lets us derive Typeclasses for our wrapper Type by delegating to the wrapped

Type's implementation. This saves us from having to write and debug our implementation.

❸ Derive `Show` for `Age` (our wrapper Type) by using `Show` for `Int` (our wrapped Type).

We could've implemented `FullName` using this exact techique. Deriving is typically preferable to hand-coding.

Next, we'll write the `show` instance for `Occupation`:

```
import Data.Generic.Rep (class Generic) ❶
import Data.Show.Generic (genericShow)

data Occupation = Doctor | Dentist | Lawyer | Unemployed
derive instance genericOccupation :: Generic Occupation _ ❷

instance showOccupation :: Show Occupation where
  show = genericShow ❸
```

❶ We need to import these to implement `show`.

❷ We need to derive `Generic` to leverage `genericShow`.

❸ We implement `Show` using `genericShow`.

Now, we are poised to write the `ToCSV` instance for `Person`:

```
data Person = Person
  { name :: FullName
  , age :: Age
  , occupation :: Occupation
  }

instance toCSVPerson :: ToCSV Person where
  toCSV (Person {name, age, occupation}) = ❶
    CSV $ show name <> "," <> show age <> "," <> show occupation ❷
```

❶ Destructure `Person` and then destructure the fields of its underlying `Record`.

❷ We `show` each field in the order we've defined them and then delimit them with commas and wrap the result in `CSV`.

7.26. Testing ToCSV

Try to come up with test code for testing `toCSV` for `Person` before looking ahead at my suggestion for a test.

Don't forget to remove our `placeholder` code from the `test` function:

```
log "placeholder" ❶
```

❶ Remove this line.

Here's some test code if you cannot think of what to write:

```
-- COMPILER ERROR!! ❶
log $ show $ toCSV
  (Person
    { name: FullName "Sue Smith" ❷
    , age: Age 23
    , occupation: Doctor
    }) == CSV "Sue Smith,23,Doctor"
```

❶ Cannot compare for equality for a `CSV`.

❷ Notice how `name` isn't just a hardcoded `String`, but is a `FullName`. Same issue with `age`.

To make the compiler happy, we need to implement `Eq` for `CSV` so that we can check for Equality:

```
newtype CSV = CSV String
derive instance newtypeCSV :: Newtype CSV _ ❶
derive newtype instance eqCSV :: Eq CSV ❷
```

❶ First derive `Newtype` to make deriving other Typeclasses easy. We'll need this later.

❷ Leverage the underlying `String's Eq` instance to derive ours. We could also leave off the `newtype` here since we can derive `Eq` for ANY Type.

Now our test code compiles and can be run:

```
log $ show $ toCSV
  (Person
    { name: FullName "Sue Smith"
    , age: Age 23
    , occupation: Doctor
    }) == CSV "Sue Smith,23,Doctor" ❶
```

❶ Prints `true`.

7.27. Writing `FromCSV`

Anytime you encode something, you'll almost always need to decode it. So let's make another Typeclass

called FromCSV with a single Method called fromCSV.

Looking back at what we did with ToCSV should be helpful. Give it a go and then check your work in the next section.

Also, you're going to need a function called split in the Data.String module. Consult Pursuit to learn more about this function.

7.28. Code for FromCSV

The class is pretty straightforward:

```
import Data.Maybe (Maybe) ❶

class FromCSV a where
  fromCSV :: CSV -> Maybe a ❷
```

❶ Import the Type Maybe. Note that the constructors are NOT imported. We'll need to rectify that soon.

❷ Since we are taking a CSV, which is essentially just a String, it's possible that we cannot decode it to our Type, e.g. we encode Type A and then try to decode that to Type B. This potential failure is why we are returning a Maybe.

And now, we can write Person's Instance one step at a time. First, the imports:

```
import Data.Maybe (Maybe(..)) ❶
-- COMPILER ERROR!!
import Data.String (Pattern(..), split) ❷
```

❶ Add (..) to include Just and Nothing which fromCSV will need.

❷ Import for spliting a string based on a Pattern into an Array String.

We're going to need to add the package for the module Data.String. Go look up that module on Pursuit and see what package it belongs in. Then return here.

According to Pursuit, it can be found in purescript-strings.

First, comment out the offending import:

```
-- import Data.String (Pattern(..), split)
```

This will allow us to full build the project.

Add strings to your spago.dhall file and then build the project with:

```
npx spago build
```

We get the following output:

```
[warn] None of your project files import modules from some projects that are
in the direct dependencies of your project.
These dependencies are unused. To fix this warning, remove the following
packages from the list of dependencies in your config:
- strings
[error] Some of your project files import modules from packages that are not
in the direct dependencies of your project.
To fix this error add the following packages to the list of dependencies in
your config:
- newtype
```

The warning is expected, but the error tells us that we need to add `newtype` to `spago.dhall`.

Do so then rerun:

```
npx spago build
```

Don't forget to uncomment out the `import Data.String` line.

The next step is the instance:

```
instance fromCSVPerson :: FromCSV Person where
  fromCSV (CSV str) = case split (Pattern ",") str of ❶ ❷
    [name, age, occupation] -> Just $ Person ❸
      { name: FullName name
      , age Age ??? ❹
      , occupation: ??? ❹
      }
    _ -> Nothing
```

❶ We destructure the `CSV` to expose its underlying `String`. We could have just as easily called `unwrap` in our code. But first, we'd have to derive a `Newtype` instance for `CSV`.

❷ We `split` the `String`, `str`, on Commas. This will return an `Array` of `Strings`.

❸ We Pattern Match on an `Array` of length 3. Remember, we can only Pattern Match `Arrays` of an exact size.

❹ The `???` are places we need to convert a `String` to another `Type`, an `Int` for `age` and `Occupation` for

`occupation`.

To convert a `String` to `Int`, we should first look to `Pursuit`. See if you can find the function even though you don't know the name before reading further.

I searched for `toInt` and didn't find out anything. I was hoping to find a function in the `Data.String` module to convert a `String` to an `Int`.

Next, I searched for `fromString` and I found a function in `Data.Int`:

```
fromString :: String -> Maybe Int
```

Notice it's a `Maybe Int` since it could fail. That is going to complicate our code.

It turns out that I could've searched for `String -> Maybe Int` on `Pursuit` and it would've also have brought up `fromString`. With `Pursuit`, you can search for Type Signatures.

Let's update our instance with `fromString`:

```
import Data.Int (fromString) ❶

instance fromCSVPerson :: FromCSV Person where
  fromCSV (CSV str) = case split (Pattern ",") str of
    [name, age, occupation] -> case fromString age of
      Just age' -> Just $ Person
        { name: FullName name
        , age: Age age' ❷
        , occupation: ??? ❸
        }
      Nothing -> Nothing ❹
    _ -> Nothing
```

❶ Need to import `fromString` from `Data.Int` module.

❷ We use `age'` so we don't shadow `age`.

❸ The `???` still needs to be addressed.

❹ If we cannot decode the `age` then we fail.

Now we need to convert a `String` to `Occupation`. That function clearly needs to be written by us. Also, there's a possibility for failure, so our function will have a Type Signature of `String -> Maybe Occupation`.

Try to code this function before reading further. Call it `toOccupation`.

Coding this is simple but tedious:

```
toOccupation :: String -> Maybe Occupation
toOccupation = case _ of ❶ ❷
  "Doctor" -> Just Doctor
  "Dentist" -> Just Dentist
  "Lawyer" -> Just Lawyer
  "Unemployed" -> Just Unemployed
  _ -> Nothing ❸
```

❶ `toOccupation` is Point-free.

❷ We're using the wildcard `case` expression.

❸ Here's the Catch-All case where we have an unrecognized `String`.

Realize that this conversion Function is fragile. If we add another `Occupation` in the future, we'll have to remember to also update this Function. The compiler will NOT be able to help us here.

To understand why, imagine we add an occupation of `BusDriver`. Notice that there is no way for the compiler to know that the `String`, `"BusDriver"`, should return `Just BusDriver`. That's because of the Catch-All case matches our newly added `Occupation` just fine.

Now let's address the `???` in the above code and finish off the decoder:

```
instance fromCSVPerson :: FromCSV Person where
  fromCSV (CSV str) = case split (Pattern ",") str of
    [name, age, occupation] -> case fromString age of
      Just age' -> case toOccupation occupation of ❶
        Just occupation' -> Just $ Person
          { name: FullName name
          , age: Age age'
          , occupation: occupation'
          }
        Nothing -> Nothing ❷
      Nothing -> Nothing
    _ -> Nothing
```

❶ Convert the `String` to `Occupation` with the chance of failure.

❷ Here's the case where we failed to decode `occupation`.

While this code works, it's messy. We haven't learned enough to clean it up, but just as a preview, here's what the code would look like in production code. I'm putting this here in case you return here someday after finishing this book:

```
instance fromCSVPerson :: FromCSV Person where
  fromCSV (CSV str) = case split (Pattern ",") str of
    [name, age, occupation] -> do ❶
      age' <- fromString age
      occupation' <- toOccupation occupation
      pure $ Person
        { name: FullName name
        , age: Age age'
        , occupation: occupation'
        }
    _ -> Nothing
```

❶ We're executing in the Maybe Monad. Future you will understand and appreciate this.

It should be clear at this point, that this version is cleaner than our version. Once we get to Monads, we'll be able to write simpler code.

7.29. Testing FromCSV

The best way to test encoders and decoders is to do a Round-trip test, i.e. encode then decode and compare the result to the original. They should match.

Try to write a good test for FromCSV before moving on but first write an Eq instance for Person so we can compare the encoded/decoded Person with the original. Do this before reading further.

So first, we must be able to compare for Equality two values of Type Person:

```
data Person = Person
  { name :: FullName
  , age :: Age
  , occupation :: Occupation
  }
derive instance eqPerson :: Eq Person -- COMPILER ERROR!!
```

The compile is complaining because it needs to know how to compare Age. We should add that too. And while we're at it, we should take care to add Eq for FullName and Occupation too.

Let's do Age first:

```
newtype Age = Age Int
derive instance newtypeAge :: Newtype Age _
derive newtype instance showAge :: Show Age
derive newtype instance eqAge :: Eq Age ❶
```

❶ This is the only line we need to add to our existing code for Age.

Let's compare Age above with FullName's implementation:

```
newtype FullName = FullName String

instance showFullName :: Show FullName where
  show (FullName name) = name
```

We don't have the benefit of having a Newtype instance. Try to convert this implementation to be the same as Age before reading on.

Here's FullName using Newtype. It replaces all of the above code:

```
newtype FullName = FullName String
derive instance newtypeFullName :: Newtype FullName _
derive newtype instance showFullName :: Show FullName
```

Now, simply add the Eq instance as we did with Age. Try that first before continuing.

Here's FullName's Eq instance:

```
derive newtype instance eqFullName :: Eq FullName
```

Now we just have to add the Eq for Occupation. Try that before reading ahead.

We can simply add one line of code to implement Occupation's Eq instance:

```
derive instance eqOccupation :: Eq Occupation
```

Now our compiler error disappears:

```
data Person = Person
  { name :: FullName
  , age :: Age
  , occupation :: Occupation
  }
derive instance eqPerson :: Eq Person ❶
```

❶ With `Eq` instances for `FullName`, `Age` and `Occupation`, the compiler can now derive an `Eq` instance for `Person`.

With the ability to check `Person` for Equality, we can now write a Round-trip test:

```
  let person = Person
        { name: FullName "Sue Smith" ❶
        , age: Age 23
        , occupation: Doctor
        }
  log $ show $ (toCSV person # fromCSV) == Just person ❷ ❸ ❹
```

❶ Make sure you indent this line (and all subsequent lines in this expression) past the variable name, i.e. `person`.

❷ We're using `#` to take the output of `toCSV person` and apply it to `fromCSV`.

❸ Since `fromCSV` can fail, we have to compare against `Just person`.

❹ Prints `true`.

Now we can run our test. It produces the following output:

```
[error] Some of your project files import modules from packages that are not
in the direct dependencies of your project.
To fix this error add the following packages to the list of dependencies in
your config:
- integers
```

Update your `spago.dhall` to contain `integers`:

```
{-
Welcome to a Spago project!
You can edit this file as you like.
-}
{ name = "my-project"
, dependencies =
  [ "console"
  , "effect"
  , "integers" ❶
  , "lists"
  , "maybe"
  , "newtype"
  , "prelude"
  , "psci-support"
  , "strings"
  , "tuples"
  ]
, packages = ./packages.dhall
, sources = [ "src/**/*.purs", "test/**/*.purs" ]
}
```

❶ Added here to maintain alpha order.

When you rerun your tests, notice how they both FAIL.

Somehow we broke our tests. Let's figure out what went wrong. Here's our test code:

```
test :: Effect Unit
test = do
  log $ show $ toCSV
    (Person
      { name: FullName "Sue Smith"
      , age: Age 23
      , occupation: Doctor
      }) == CSV "Sue Smith,23,Doctor"
  let person = Person
        { name: FullName "Sue Smith"
        , age: Age 23
        , occupation: Doctor
        }
  log $ show $ (toCSV person # fromCSV) == Just person
```

Both tests are printing `false`, which is wrong. That makes me think that `toCSV` isn't working, since both

tests use `toCSV` but only one test uses `fromCSV`. So, let's test that theory.

Add the following code to the top of our test code to print out the result of `toCSV`:

```
log $ show $ toCSV -- COMPILER ERROR! ❶
  (Person
    { name: FullName "Sue Smith"
    , age: Age 23
    , occupation: Doctor
    })
```

❶ No type class instance was found for `Data.Show.Show CSV`.

We never implemented `Show` for `CSV` so let's add that:

```
newtype CSV = CSV String
derive instance newtypeCSV :: Newtype CSV _
derive newtype instance eqCSV :: Eq CSV
derive newtype instance showCSV :: Show CSV ❶
```

❶ We only had to add this one line.

Now running the test with `npx spago run` prints:

```
"\"Sue Smith\",23,Doctor" ❶
false
false
```

❶ The `\"` doesn't look right.

For some reason we have imbeded quotes in our `String`. But why? And what did we change to break this?

It may not be obvious at first as to why this is. To see why, let's run the REPL again:

```
npx spago repl
```

And type at the prompt:

```
> show "Sue Smith"
"\"Sue Smith\"" ❶

> ^D ❷
See ya!
()
```

❶ There's those embedded quotes again.

❷ Hit Control-D to exit.

From this test, we can see that showing a `String` will display the `String` surrounded by double-quotes.

This means that we broke `FullName` when we derived it's `Show` instance. Let's look at the BEFORE:

```
instance showFullName :: Show FullName where
  show (FullName name) = name ❶
```

❶ We don't call `show` on `name`. We simply return the `String`.

And now AFTER:

```
derive newtype instance showFullName :: Show FullName ❶
```

❶ Delegates to the `Show` instance of the wrapped Type, which is `String`.

We can see that the derived code calls `show` on the underlying value, which is `name`. This is very different than our hand-coded `show`, which just returns the `String`.

Let's put back our old `Show` instance for `FullName` by first removing the derive:

```
derive newtype instance showFullName :: Show FullName ❶
```

❶ Remove this line.

And then by adding the hand-coded `Show` instance:

```
instance showFullName :: Show FullName where
  show (FullName name) = name
```

Now when we run our tests again, it prints:

```
"Sue Smith,23,Doctor"
true
true
```

Much better.

We're going to leave in our debugging code, but just commented out:

```
test :: Effect Unit
test = do
  -- log $ show $ toCSV
  --   (Person
  --     { name: FullName "Sue Smith"
  --     , age: Age 23
  --     , occupation: Doctor
  --     })
  log $ show $ toCSV
    (Person
      { name: FullName "Sue Smith"
      , age: Age 23
      , occupation: Doctor
      }) == CSV "Sue Smith,23,Doctor"
  let person = Person
        { name: FullName "Sue Smith"
        , age: Age 23
        , occupation: Doctor
        }
  log $ show $ (toCSV person # fromCSV) == Just person
```

Chapter 8. Abstract Algebra

There are many concepts from Mathematics that Functional Programming borrows heavily from. One of those areas is Category Theory. We'll see that once we get into Functors, Applicatives and Monads.

The other area is Abstract Algebra. We're going to take a cursory look at these concepts as a whole, since we only need to understand it as it relates to programming. A word of caution here. The concepts here are actually very simple. Simple enough that you could have easily learned these concepts in your Algebra class.

I suspect the reason that these subjects aren't taught in middle school is that they're considered too abstract to teach every student since most students will not need them moving forward. Unfortunately, this kind of thinking is pervasive in an educational system that is still set up to produce Industrial Revolution workers. But we live in the Information Age where these concepts are beneficial to us programmers and so it's worth learning.

Another thing to keep in mind is that these concepts weren't developed by one person over a short period of time. They were developed over hundreds of years and by people from all different countries, many of whom were working in different areas of Mathematics.

This somewhat explains why the names of similar concepts are so different from one another. For example, Monoid and Semigroup are two concepts that differ by a single rule or requirement and yet their names would suggest that they are very different ideas.

This unfortunate circumstance of history is the only difficulty we'll need to overcome. The names here are complicated but the concepts are pretty simple. Trust me when I say that you've done far more difficult things in programming than we'll do here.

8.1. Binary Operators

As a quick review, let's look at Binary Operators. The word `Binary` means `2`. That `2` stands for `2` Parameters. The `Operator` is a symbol that implies a very specific function, e.g. `+`:

```
12 + 73
```

Here the `2` (Binary) Parameters are `12` and `73`. The `+` (Operator) is Numerical Addition.

Here are some more examples of Binary Operators that we should all be quite familiar with:

```
+
−
*
/
&&
||
```

Binary Operators are usually written infixed, in programming and always in Math:

```
1 + 3
51 − 19
4 * 43
100 / 20
true && false
false || true
```

8.1.1. Associative Property

A Binary Operator, `•`, is `Associative` if the following is true:

```
 a • (b • c) = (a • b) • c  ❶
```

❶ Parentheses associate. On the left side, the Parentheses associates `b` with `c` and on the right it associates `a` with `b`.

You can remember what this means if you consider the Parentheses as the mechanism which Associates the Operator with its Parameters.

Here we're using the symbol, `•`, to represent ANY Binary Operator. Could be Addition or Multiplication or Logical And.

When a Binary Operator is Associative, we can rearrange the Parentheses however we like and the results are equal:

```
((a •   b) •  c)  • d   =
 (a •  (b  •  c)) • d   =
 (a •   b) • (c   • d)  =
  a •  (b  • (c   • d)) = ...
```

We're all too familiar with Associative Operators (Order of Operations does NOT matter):

```
(1 + 3) + 47 = 1 + (3 + 47)
2 * (17 * 54) = (2 * 17) * 54
```

And we're equally familiar with NON-Associative Operators (Order of Operations does matter):

```
1 / (3 / 5) /= (1 / 3) / 5
(17 - 42) - 3 /= 17 - (42 - 3)
```

8.1.2. Commutative Property

A Binary Operator, `•`, is Commutative if the following is true:

```
 a • b = b • a ❶
```

❶ `a` and `b` ***Commute*** or travel across the Binary Operator and the results are unchanged.

Again, we've seen plenty of examples of Commutative Operators in Arithmetic:

```
1 + 10 = 10 + 1
3 * 25 = 25 * 3
```

And NON-Commutative Operators:

```
1 - 10 /= 10 - 1
3 / 25 /= 25 / 3
```

8.2. Closure

In Programming, the term `Closure`, has a very different meaning than it does in Mathematics.

Let's investigate the Mathematical definition.

Take the following Set:

```
{ true, false }
```

Let's apply the Binary Operator, `&&`, to every combination of the Elements of the Set:

```
true  && true  = true
true  && false = false
false && true  = false
false && false = false
```

Notice how the result is always a value that's in our Set. When this is the case, we say, the Set is Closed under the Operation of &&.

Take another Set:

```
{ 1, 2, 3 }
```

Is this Set Closed under the Binary Operator, +?

No, it's not since `3 + 3 = 6`, which is NOT in the Set.

What about subtraction? Or multiplication? Or division?

Turns out the answer is NO to all of these questions.

Let's consider the Set:

```
{ 1, 2, 3, ... } ❶
```

❶ The Set of Positive Integers.

Is this Set closed under the Binary Operator, +?

Yes. Adding any two Positive Integers produces a Positive Integer.

What about Subtraction, Division or Multiplication?

Subtraction and Division are NOT closed, e.g. `1 - 3 = -2` and `1 / 3 = 1/3`.

But this Set is closed under Multiplication since multiplying any two Positive Integers is a Positive Integer.

8.3. Magma

Magma is a red, hot, liquid rock. But it's also an Abstract Algebra Structure. It's sort of the bedrock of everything we'll talk about here. All other Algebraic Structures we investigate will build on this.



A ***Magma*** is a Set that's Closed under a Binary Operator.

Here's an example of a Magma:

```
M = { true, false } and • = &&
```

The Set `M` is the set of all `Booleans` and the Binary Operator `•` is the `Logical And` operation, `&&`.

Here's an example of NOT a Magma:

```
X = { 0, 1, 2, 3, ... } and • = −
```

We saw earlier how Subtraction is NOT closed over the Set of Positive Integers. This is also true for the Natural Numbers, which includes `0`.

Here is the Formal Definition of a Magma using Mathematical Notation:

```
∀ a, b ∈ M  ⇒  a • b ∈ M
```

where `M` is the Magma's Set.

Since Math is a Language with its own Syntax that we're probably not too familiar, let's parse this statement:

```
∀ a, b ∈ M    (For all a, and b that is an Element of M)
⇒             (it follows that)
a • b ∈ M     (the result of a • b is an Element of M)
```

8.4. Semigroup

If we start with a Magma and add 1 more requirement, we get a Semigroup.



A ***Semigroup*** is a Magma where the Binary Operator is Associative.

An example of a Semigroup:

```
S = { "a", "b", ..., "z", "aa", "ab", ... } and  • = <> ❶ ❷
```

❶ `S` is the Set of NON-empty Strings.

❷ `<>` is the Concatenation (or Append) operator.

We should validate that our Binary Operator is Associative:

```
("ab" <> "cd") <> "ef" = "ab" <> ("cd" <> "ef")
```

We can easily justify that it is when we see the above example. The order in appending `Strings` doesn't matter. We can first append `"ab"` with `"cd"` and then `"ef"` or we can first append `"cd"` with `"ef"` and then append that to `"ab"`.

Either way, we have the same final `String`, `"abcdef"`.

Here's an example of NOT a Semigroup:

```
X = { 0, 1, 2, 3, ... } and • = –
```

This isn't Closed under the Binary Operator, `–`, which means that it's not even a Magma. So it's definitely not a Semigroup.

Here is the Formal Definition of a Semigroup using Mathematical Notation:

```
∀ a, b, c ∈ S  ⇒  a • (b • c) = (a • b) • c
```

where `S` is the Semigroup's Set.

Let's parse this statement:

```
∀ a, b, c ∈ S              (For all a and b and c
                           that is an Element of S)
⇒                          (it follows that)
a • (b • c) = (a • b) • c  (the result of applying a to
                            the result of b • c is equal to
                            the result of applying a • b to
                            c)
```

Notice how the right-hand side of `⇒` captures the Associative nature of our Binary Operator.

Also, notice how `c ∈ S` is the notation that Mathematicians use to constrain their variables. In PureScript, we constrain by using `∀ c. Semigroup c =>`.

These are conceptually equivalent. We're still talking about ALL `c` but with the added requirement that `c` be an element of `S`, our Semigroup's Set.

8.5. Semigroup Typeclass

In PureScript, we have a Typeclass for `Semigroup`:

```
class Semigroup a where
  append :: a -> a -> a

infixr 5 append <> ❶
```

❶ Our Binary Operator.

Some common instances of `Semigroup` are `String`, `List` and `Array`:

```
"ab" <> "123" = "abc123"
(1 : 2 : 3 : Nil) <> (98 : 99 : Nil) = (1 : 2 : 3 : 98 : 99 : Nil)
[1, 2, 3] <> [98, 99] <> [1, 2, 3, 98, 99]
```

8.6. Monoid

If we add 1 more requirement to Semigroup we get a Monoid:



A ***Monoid*** is a Semigroup where the Set has a Neutral Element, e.

Here is the Formal Definition of a Monoid using Mathematical Notation:

```
e ∈ M, ∀ a ∈ M  ⇒  a • e = e • a = a
```

where `M` is the Monoid's Set.

Let's parse the above definition for the Neutral Element's requirements:

```
e ∈ M                (e is an Element of M)
, ∀ a ∈ M            (and for all a that is an Element of M)
⇒                    (it follows that)
a • e = e • a = a    (applying a to e is equal to
                      applying e to a which is equal to a)
```

In simpler terms, the Neutral Element (also called the Empty Element or Identity) can be applied to any Element of the Set, including itself since it too is an Element of the Set, and have no impact on that Element.

Here's an example of a Monoid to drive home the point:

```
M = { "", "a", "b", ..., "z", "aa", "ab", ... } and  • = <> ❶ ❷ ❸
```

❶ `e` is `""` the `Empty String`, which is the first element of Set `M`.

❷ `M` is the Set of Strings.

❸ `<>` is the Concatenation (or Append) operator.

If we append the `Empty String` to any `String`, we get that same `String` back. Conversely, if we append any `String` to the `Empty String`, we still get that same `String` back.

This is why `e` is called the `Identity` since it simply returns back the exact same thing you started with. There is a function in `Prelude` called `identity` that has a similar effect:

```
identity :: ∀ a. a -> a
identity x = x
```

Here's another example of a Monoid:

```
M = { 0, 1, 2, 3, ...} and • = +, e = 0
```

Remember that `M` is first a Magma:

```
1 + 3 = 4     (Closed)
```

and then a Semigroup:

```
1 + (2 + 3) = (1 + 2) + 3     (Associative)
```

and finally a Monoid:

```
0 + 1 = 1 + 0 = 1       (Identity)
```

Another example:

```
M = { true, false } and • = ||, e = false
```

`M` is first a Magma, then a Semigroup and finally a Monoid:

```
true || false = true                                (Closed)
true || (true || false) = (true || true) || false   (Associative)
false || true = true || false = true                (Identity)
```

Yet another example:

```
M = { 0, 1 } and • = ∗, e = 1
```

`M` is first a Magma, then a Semigroup and finally a Monoid:

```
1 ∗ 1 = 1                    (Closed)
1 ∗ (0 ∗ 1) = (1 ∗ 0) ∗ 1    (Associative)
1 ∗ 0 = 0 ∗ 1 = 0            (Identity)
```

8.7. Monoid Typeclass

In PureScript, we have a Typeclass for `Monoid`:

```
class Semigroup m <= Monoid m where ❶ ❷
  mempty :: m
```

❶ `Semigroup` is a Superclass of `Monoid`, which says that every `Monoid` must first be a `Semigroup` and then we add the Empty Element, `mempty`. This mirrors the Mathematical definition.

❷ Note the use of the backward fat arrow `<=`.

Here is the `String` instances for `Semigroup` and `Monoid`:

```
foreign import concatString :: String -> String -> String ❶

instance semigroupString :: Semigroup String where ❷
  append = concatString

instance monoidString :: Monoid String where ❸
  mempty = ""
```

❶ This is a Foreign Import to let the PureScript compiler know how to call a Javascript function.

❷ The `Semigroup` instance calls Javascript, presumably for speed, to append the two `Strings`.

❸ The Empty Element is the `Empty String`.

8.8. Monoids in Programming vs Math

When programming in Statically-Typed Languages like PureScript and Haskell, we concern ourselves with Types not Sets. So how do we bridge the gap between Mathematics and Programming?

In Math, a Monoid is a Set with a Binary Operator. In PureScript, a `Monoid` is a Type with a Binary Operator `<>`.

Earlier we talked about Types having Inhabitants. We can think of Types as Sets of Inhabitants.

Take the Type `Boolean`. It's Inhabitants is the Set:

```
{ true, false }
```

And `String`, `Int` and `Array Int` all have the following Set of Inhabitants:

```
{ "", "a", "b", ... }                  -- String
{ ..., -3, -2, -1, 0, 1, 2, 3, ... }   -- Int
{ [], [1], [2], ... }                  -- Array Int
```

So Monoids in Math are Isomorphic to `Monoids` in Programming. The Set in Math is the Set of Inhabitants of a Type in Programming.

8.9. Monoids in Programming

Let's say we want to write a function to combine a `List` of `Monoids`:

```
import Data.List (List)

combineListOfMonoids :: ∀ a. Monoid a => List a -> a ❶
combineListOfMonoids = go mempty where ❷ ❸
  go c Nil      = c ❹
  go c (x : xs) = go (c <> x) xs ❺
```

❶ `a` is Constrained to be a `Monoid`.

❷ `combineListOfMonoids` is in Point-free Notation. The `List` variable goes unmentioned.

❸ The initial result is the Empty Element, `mempty`.

❹ This case is when the `List` is exhausted. `c` is the combined value here, which could be `mempty` if we're initially given an Empty List, `Nil`.

❺ We append the `head` of the `List` to our combined value, `c` and then Recurse with the `tail`.

Notice that we start with `mempty`. There are two reasons for this. First, because our `List` could initially be empty and second, because we want an initial value that won't affect the result of the append operation. Much like multiplying by `1` or adding `0`.

We could not do this with just a `Semigroup` Constraint since Semigroups do not have Empty Elements.

But what if we wanted to write a version where we could also allow Semigroups? We could start with the first element instead of the Empty Element. But there'd be no first element if the `List` is empty.

We need a way to guarantee that the list isn't empty and we can do this by using the `NonEmpty` Type.

Let's write a similar function but for NON-Empty `Lists`:

```
import Data.List (List) ❶
import Data.NonEmpty (NonEmpty(..)) ❷

combineNEListOfSemigroups :: ∀ a. Semigroup a => NonEmpty List a -> a ❸
combineNEListOfSemigroups (NonEmpty x xs) = go x xs where ❹
  go c Nil = c
  go c (y:ys) = go (c <> y) ys
```

❶ It's common practice to keep your imports in alphabetical order. That's why `Data.List` is before `Data.NonEmpty`.

❷ We need to import the constructors so we can destructure the Non-empty `List`.

❸ Now that we disallow empty `Lists`, we can relax our requirements on `a` and allow `Semigroups`.

❹ We now have to start with the `head` of the `NonEmpty` since we no longer have `mempty`. But since the `List` is guaranteed by the `NonEmpty` Type to always have at least 1 element, we can get the `head` with no fear of failure.

The body of the code is very similar to `combineListOfMonoids`. But this `NonEmpty` Type is new to us, so let's take a closer look.

Here is the definition of `NonEmpty`:

```
data NonEmpty f a = NonEmpty a (f a)

infixr 5 NonEmpty as :| ❶
```

❶ Binary Operator for constructing a `NonEmpty`. Notice that it's very similar to `List's` Cons operator, `:`.

Notice how `NonEmpty` takes 2 Type Parameters. `f` is the collection Type and `a` is the element Type. It could've been define as:

```
data NonEmpty collection element = NonEmpty element (collection element)
```

This would have been pretty verbose to look at, so it makes sense why single letters where used. But it's not clear as to why `f` was used.

We're used to seeing `a`, but why `f` and `a` instead of `a` and `b`, for example?

It turns out that all collections of elements, at least every collection I can think of, are also `Functors`. We haven't learned about `Functors` yet, but clearly the author of this Type understands `Functors` and so used `f` to represent the `Functor`.

Here is our Type Signature for `combineNEListOfSemigroups`:

```
combineNEListOfSemigroups :: ∀ a. Semigroup a => NonEmpty List a -> a
```

Notice how we specified the `f` as `List`. That's the collection or `Functor` Type. The second Type Parameter is `a`, which is the element Type.

Now let's look at the Constructor for `NonEmpty`, which is also called `NonEmpty`:

```
data NonEmpty f a = NonEmpty a (f a) ❶
```

❶ Constructor is on the right-hand side of the equals sign.

Notice how we MUST have a value of `a` AND a value of `f a` to construct a `NonEmpty`. That's the magic ingredient in this Type. Without both parts, you cannot hope to construct a `NonEmpty`, which is why it can guarantee that the collection, or `List` in our case, is NOT empty. It's constructor REQUIRES at least 1 element.

There is a helper function to convert `NonEmpty` to `List`:

```
newtype NonEmptyList a = NonEmptyList (NonEmpty List a) ❶

toList :: NonEmptyList ~> List
```

❶ `NonEmptyList` is a `newtype` that has only one Type Parameter, the element Type.

Remember that `~>` is the Natural Transformation from one `Functor` to another. (Functors are coming. I promise.) It's shorthand for:

```
toList :: ∀ a. NonEmptyList a -> List a
```

You may be wondering how do you go from a `List` to a `NonEmpty`. There is no helper in the library for this. But we can write our own function for this:

```
import Data.List (List(..), (:))
import Data.List.Types (NonEmptyList(..))
import Data.Maybe (Maybe(..))
import Data.NonEmpty ((:|))

fromEmptyList :: ∀ a. List a -> Maybe (NonEmptyList a) ❶
fromEmptyList Nil = Nothing ❷
fromEmptyList (x : xs) = Just $ NonEmptyList $ x :| xs ❸
```

❶ We can fail if we're given an empty `List`, so we return a `Maybe`.

❷ This is the failure case.

❸ We remove the `head` from the `List` and give it to the `NonEmpty` Cons operator `:|`, and then wrap it in the `NonEmptyList newtype`.

That was a fun diversion, but now back to Abstract Algebra.

8.10. Groups

If we add 1 more requirement to Monoid we get a Group.



A ***Group*** is a Monoid where every element of the Set has an Unique Inverse.

Here is the Formal Definition of a Group using Mathematical Notation:

```
∀ a ∈ G, a' ∈ G, e ∈ G  ⇒  a • a' = a' • a = e
```

Let's parse this out:

```
∀ a ∈ G                (For all a that is an element of G)
, a' ∈ G               (and a' that is an element of G)
, e ∈ G                (and e that is an element of G)
⇒                      (it follows that)
a • a' = a' • a = e    (applying a to a' is equal to
                        applying a' to a which is equal to e)
```

Essentially, `a'` is the Unique Inverse of `a` and `e` is the Empty Element, i.e. Identity Element.

Let's now see how this works by starting with the following Monoid:

```
M = { 0, 1, 2, 3, ...} and • = +, e = 0
```

Now let's add the Group requirement, i.e. Unique Inverses for all elements of the set:

```
G = { ..., -3, -2, -1, 0, 1, 2, 3, ...} and • = +, e = 0
```

Notice that the Empty Element, `e`, is its OWN Inverse and it's Unique since it's not an Inverse for any other element in the Set.

If we take `e` and apply to `e` we get `e`. So it acts as an Identity to all elements but also as an Inverse for itself. It also acts like Identity for itself.

Note that Inverses, `a'`, take any element, `a`, and when applied with the Binary Operator map them to the Identity element, `e`, i.e. `a' • a = e`.

Here's another example of a Group:

```
G = { ..., 1/3, 1/2, 1, 2, 3, ...} and • = *, e = 1
```

Again, `1` is it's own Inverse, i.e. `1 * 1 = 1` and it's an Identity for every element, e.g. `1/3 * 1 = 1/3` including itself, `1 * 1 = 1`.

An example of NOT a Group:

```
X = { ..., -3, -2, -1, 0, 1, 2, 3, ...} and • = -, e = 0
```

Subtraction fails us again due to the fact that it's NOT Associative:

```
(-3) -   3  /= 0
  3  - (-3) /= 0
```

8.11. Group Typeclass

In PureScript, `Group` is a Typeclass defined as:

```
class Monoid g <= Group g where ❶
  ginverse :: g -> g ❷
```

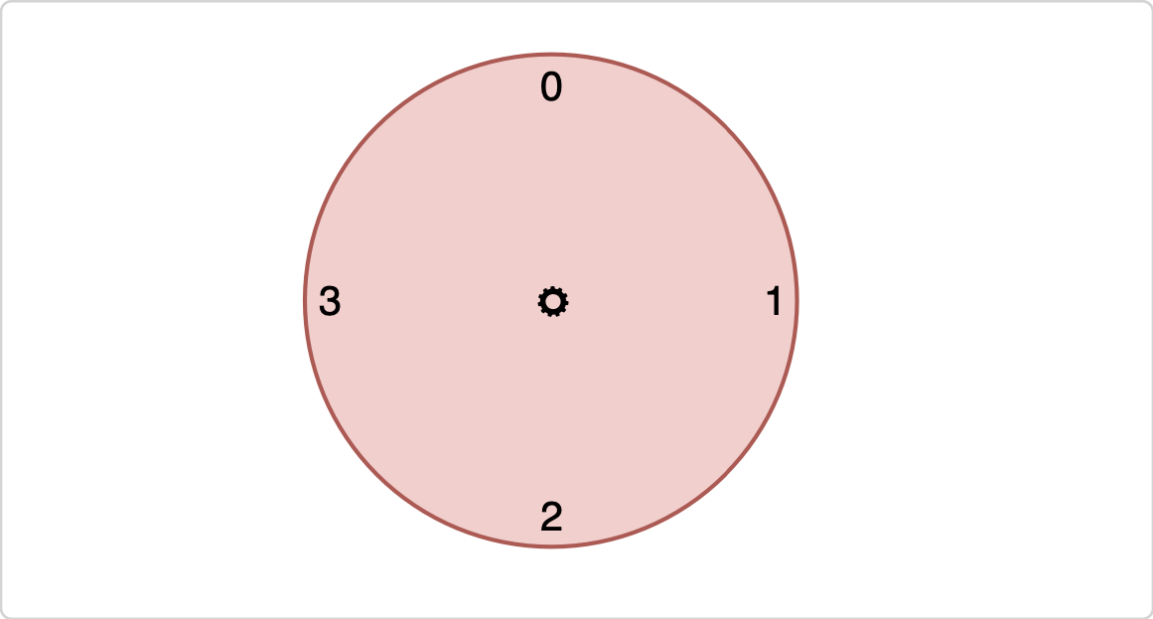
❶ `Monoid` is a Superclass of `Group`. Notice the backward fat arrow.

❷ `ginverse` returns the Inverse of the value it's given.

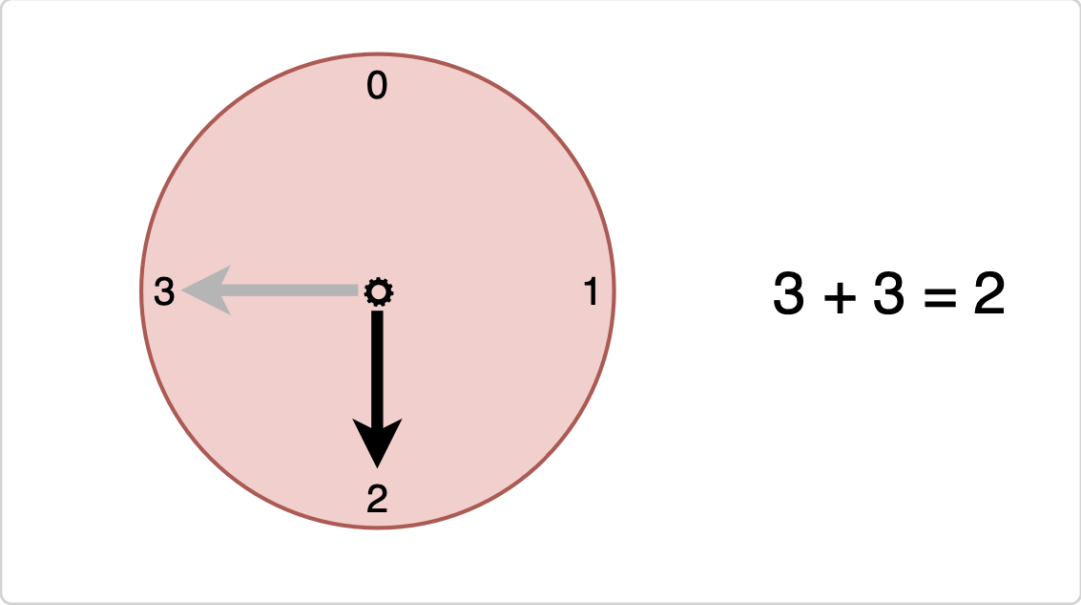
Notice how `ginverse` takes a `g` and returns a `g`. That's because Inverses are part of the same Set which in programming is the same Type.

8.12. Modular Arithmetic and Groups

Imagine you have a clock with 4 positions instead of 12. This clock starts with 0 instead of 1:

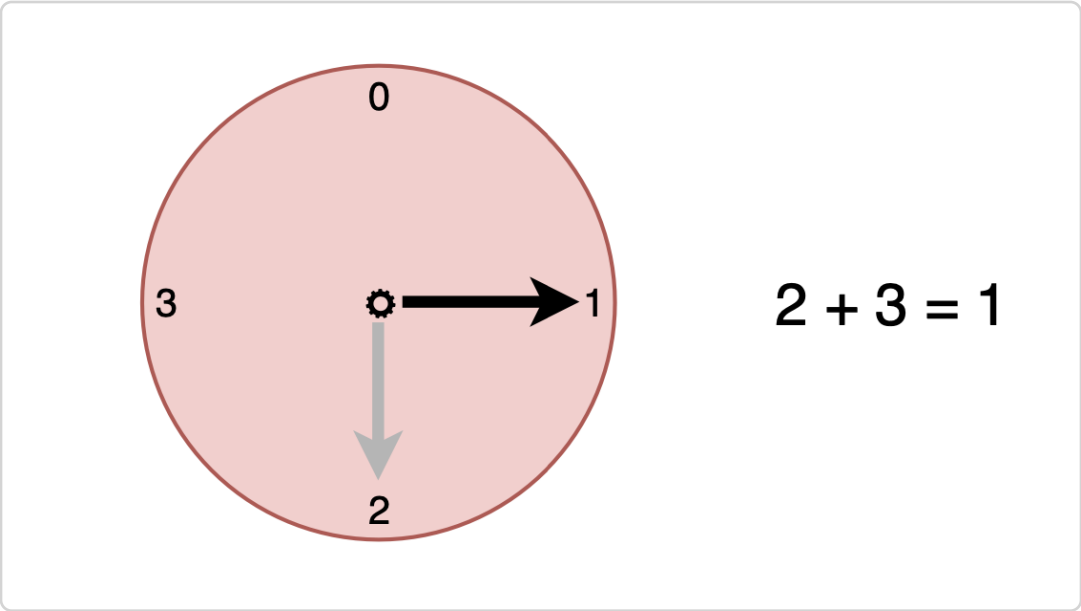


Let's start at position 3 and add 3 by going Clockwise 3 positions, i.e. from 3 to 0 to 1 to 2:



This kind of Arithmetic is also known as ***Modular Arithmetic***.

Next, imagine we start at 2 and add 3. Once again we move Clockwise 3 positions and wind up at 1:



This clock represents the ***Integers Mod 4***. `Mod` is short for Modulo.

Symbolically, it's displayed as $\mathbb{Z}_4$ which is the following Set, `{ 0, 1, 2, 3 }`. Notice that these are same the numbers on our clock face.

But what about Subtraction? Can we subtract?

Actually, we can by going Counterclockwise. But, since we're interested in Modular Arithmetic in the context of `Groups` and a `Group` only has 1 Binary Operator, let's look at how we can get the same result as Subtraction using Addition instead.

In Math, we learned that Subtracting is like adding the Inverse or Negative number, e.g.:

```
10 - 3 = 10 + (-3)
```

Here `-3` is the Inverse of `3`. In Modular Arithmetic, we also have Inverses, but they aren't negatives.

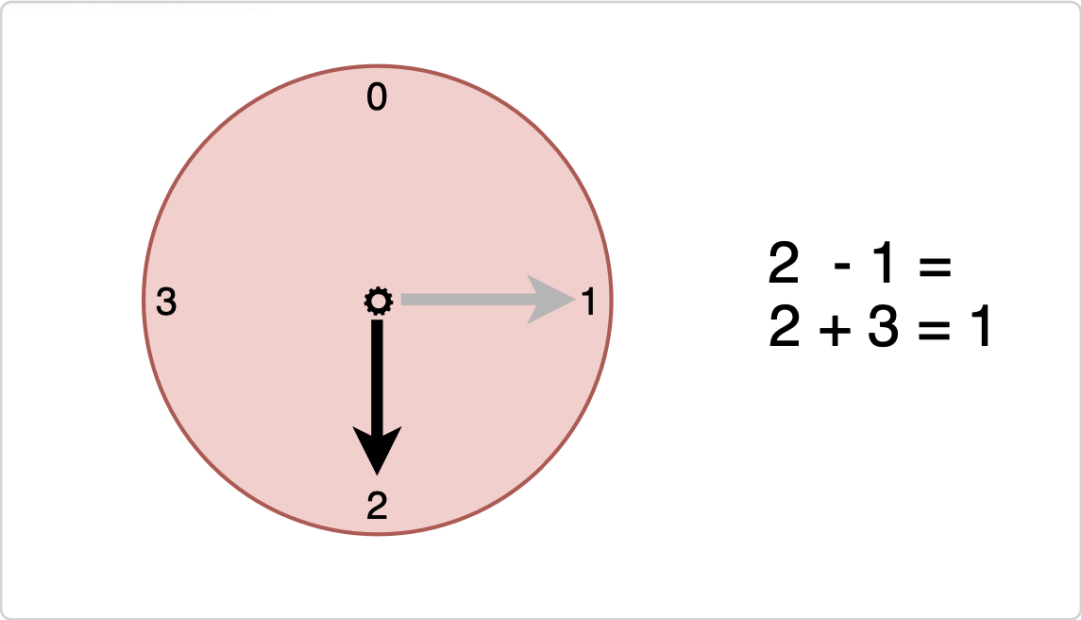
Remember, in the definition of a `Group`, every element in our Set has an Inverse such that:

```
a • a' = a' • a = e
```

where `a'` is the Inverse of `a`.

In regular Arithmetic, this is just `a` and `-a`. But this is not the case for Modular Arithmetic.

We can consult our clock to help us see why this is true:



Notice how Subtracting `1` is the same as Adding `3`. That makes `3` the Inverse of `1` and vice versa. Here are the Inverses:

```
 a   a'
--- ---
 0   0
 1   3
 2   2
 3   1
```

Since our Binary Operator is `+`, adding `a` and `a'` always results in `0`, which is the definition of Inverse that we saw in the Group requirement:

```
 a   a'    a • a'
--- --- -----------
 0   0   0 + 0 = 0
 1   3   1 + 3 = 0
 2   2   2 + 2 = 0
 3   1   3 + 1 = 0
```

(If the above table doesn't make sense, try each of these cases using our clock.)

We've shown that `Integers Mod 4`, $\mathbb{Z}_4$, with a Binary Operator of `+` meets the unique requirement for a Group. But we need to make sure it's Monoid.

To be a Monoid, we need an Identity Element. That would be `0`. Apply it with our Binary Operator to any element of our set and we get back that element.

`Integers Mod 4` is also a Semigroup since the Binary Operator, `+`, is Associative.

And it's a Magma since adding any number to any other number in our Set produces a member of our Set (thanks to Modular Math), i.e. `Integers Mod 4` is closed under Addition.

So `Integers Mod 4` with Addition is a Group:

```
ℤ₄ = { 0, 1, 2, 3 }
•  = +
```

8.13. Abelian Group

An ***Abelian Group*** is a Group where the Binary Operator is also Commutative. (This is named after 19th century Mathematician Neils Henrik Abel.)

Both `+` and `*` are Commutative:

```
a + b = b + a
x * y = y * x
```

So the following Groups are also Abelian Groups (also known as ***Commutative Groups***):

```
{ ..., -3, -2, -1, 0, 1, 2, 3, ... } and + where e = 0
{ ..., 1/3, 1/2, 1, 2, 3, ... } and * where e = 1
```

8.14. Abelian Group Type Alias

In PureScript, Abelian Group is just a Type Alias:

```
type Abelian a b = Group a => Commutative a => b
```

This definition looks odd at first glance. We know that Type Aliases are nothing more than Macros, i.e. you can think of the right-hand side as replacing everywhere the left-hand side is used in your code. The left-hand side is easy to read and type, so Type Aliases help us reduce the amount code and cognitive overhead.

But this Type Alias has Constraints and it doesn't look like a Type Signature for a Function. When I first saw this, I was confused until I realized how it's used, which made me realize that it's poorly named.

Imagine we make the following Type Alias:

```
type ConstrainAbelian a b = Abelian a b
```

That's a much better name since it reminds us that the `Abelian` Type Alias gives us a way to easily constrain `a` to be an Abelian Group, i.e. `a` must be a `Group` and `Commutative`. But what exactly is `Commutative`?

`Commutative` is a Typeclass:

```
class Semigroup a <= Commutative a
```

`Commutative` is just a `Semigroup` with the following implied law:

```
∀ x, y ⇒ x <> y = y <> x      [Commutative]
```

In PureScript, we cannot enforce the Commutativity of the `<>` operator, so the best we can do is just define it as such.

To get a better feel for how this Type Alias can be used, let's look at a simple example:

```
find :: ∀ a. Abelian a (Set a -> Maybe a) ❶
```

❶ The `b` in the Alias Definition is the Function's FULL Type Signature, i.e. `(Set a -> Maybe a)`.

This expands to:

```
find :: ∀ a. Group a => Commutative a => Set a -> Maybe a
```

So the `Abelian` Type Alias is just a HELPER for adding the necessary constraints to a function's Type Signature, viz. `Group a => Commutative a =>`.

8.15. Arithmetic Operators in PureScript

Arithmetic Operators use unfamiliar types in PureScript. In Haskell, the Type is `Num`. Not scary at all.

To see these types, run the PureScript REPL with the following command:

```
npx spago repl
```

And then type `:t (+)` to determine the Type Signature for Addition:

```
> :t (+)
forall a. Semiring a => a -> a -> a
```

The only part of the Type Signature that is baffling is the `Semiring` Constraint. The `a -> a -> a` makes sense. In fact, it implies that `+` is Closed for the Set of Inhabitants of Type `a`. So `Semiring` is at least a Magma.

Next, let's try to see what Multiplication's Type Signature is:

```
> :t (*)
forall a. Semiring a => a -> a -> a
```

It also has a `Semiring` Constraint.

And what about Subtraction:

```
> :t (-)
forall a. Ring a => a -> a -> a
```

That was unexpected. It's got a `Ring` Constraint, whatever that is.

And finally, Division:

```
> :t (/)
forall a. EuclideanRing a => a -> a -> a
```

Once again, this Constraint is quite the surprise.

So we have:

```
Semiring
Ring
EuclideanRing
```

Who would've thought simple Arithmetic Operations that we learned in Grade School would be so complicated.

It turns out that in PureScript, Arithmetic Operators are defined using these Abstract Algebraic Structures. The benefit of doing this is that these Operators can be used for non-numeric things giving us the full power of these abstractions.

The downside to this approach is that we've got to learn a little more Abstract Algebra.

8.16. Semiring



A ***Semiring*** is an Algebraic Structure with Set `R` and 2 Binary Operators, `+` and `·`, called ***Addition*** and ***Multiplication***.

A Semiring also has some laws. First, the laws for Commutative Monoid (a Monoid where the Binary Operator is Commutative) and Addition:

```
(R, +) is a Commutative Monoid with Identity of 0:

(a + b) + c = a + (b + c)   [Associativity] (Semigroup)
0 + a = a + 0 = a           [Identity] (Monoid)
a + b = b + a               [Commutative] (Commutative Monoid)
```

Next, the laws for Monoid and Multiplication:

```
(R, ·) is a Monoid with identity of 1:

(a · b) · c = a · (b · c)   [Associativity] (Semigroup)
1 · a = a · 1 = a           [Idenity] (Monoid)
```

Then the laws for Multiplicative Distribution across Addition on the Left and Right:

```
a · (b + c) = (a · b) + (a · c)       [Left Distributivity]
(a + b) · c = (a · c) + (b · c)       [Right Distributivity]
```

And finally the law for Multiplication with `0` that Annihilates `R`:

```
0 · a = a · 0 = 0         [Annihilation]
```

Notice that Addition must be Commutative but Multiplication does not. This may seem odd at first since the Multiplication we know is Commutative. But you have to keep in mind that `Addition`, `Multiplication` are abstract operations, and `0` and `1` are abstract members of the set `R`.

In the concrete case of numbers, they're the operations we grew up with. And `0` and `1` are actual numbers. But in the abstract sense, `0` and `1` are just Additive and Multiplicative Identities. And Addition and Multiplication are arbitrary functions.

8.17. Semiring Typeclass

In PureScript, `Semiring` is a Typeclass defined as:

```
class Semiring a where
  add :: a -> a -> a
  zero :: a
  mul :: a -> a -> a
  one :: a

infixl 6 add as + ❶
infixl 7 mul as *
```

❶ Order of operations makes `*` higher Precedence than `+` as is common in Arithmetic with Numbers.

We can see from the Type Signature of `add` and `mul` that they are Closed operations since they take 2 `a's` and return an `a`. And there exists an Identity, `zero`, to be used with `add` and another Identity, `one`, to be used with `mul`.

The archetypical implementation is for `Int`:

```
instance semiringInt :: Semiring Int where
  add = intAdd
  zero = 0
  mul = intMul
  one = 1

foreign import intAdd :: Int -> Int -> Int ❶
foreign import intMul :: Int -> Int -> Int
```

❶ Actual implementations are in Javascript.

Here's the trivial implementation for `Unit`:

```
instance semiringUnit :: Semiring Unit where
  add _ _ = unit
  zero = unit
  mul _ _ = unit
  one = unit
```

Since `Unit` has a single inhabitant, `unit`, it's the only solution to the `add` and `mul` operations. So we just ignore the Input Parameters.

And since `zero` and `one` must be an element of the Set, we use the only Inhabitant we have, `unit`, in both cases.

8.18. Semiring for Functions

One of the benefits of using Abstractions for simple Arithmetic is that you can generalize Numbers to things that behave similarly. For example, Functions.

You might not immediately think to replace Numbers with Functions, but it actually can be done. Here's the `Semiring` instance for Functions:

```
instance semiringFn :: Semiring b => Semiring (a -> b) where
  add f g x = f x + g x
  zero = \_ -> zero
  mul f g x = f x * g x
  one = \_ -> one
```

This is a bit complicated and confusing so let's break down each line one at a time.

```
instance semiringFn :: Semiring b => Semiring (a -> b) where
```

This instance is for Functions of Type `a -> b`, where `b` is also a `Semiring`. This will prove to be important once we look at `add` and `mul`.

But looking at this, we may ask, what about functions of multiple Parameters. Well, remember, thanks to Currying, all functions technically only have 1 Parameter. They instead return a function to support multiple Parameters. So `b` here COULD be a function.

We can best see this by replacing `b` with `(b -> (c -> d))`. We can see, once again, how functions of multiple Parameters are equivalent to a function that returns another function and so on until we've exhausted all of the Parameters.

And since functions of multiple Parameters are just functions that take 1 Parameter and return a function, that function fulfills the `Semiring b` Constraint.

Let's look at the next line:

```
instance semiringFn :: Semiring b => Semiring (a -> b) where
  add f g x = f x + g x
```

The `add` function seems a bit strange when we consider its Type Signature:

```
add :: a -> a -> a ❶
```

❶ Takes 2 Parameters NOT 3.

Yet, we have 3 Parameters in our `add` function. If you remember from earlier, we can move variables across the equal sign. Let's do that and see how we really only have 2 Parameters:

```
instance semiringFn :: Semiring b => Semiring (a -> b) where
  add f g = \x -> f x + g x ❶
```

❶ Equivalent to the original `add` function above but now with only 2 Parameters on the left-hand side.

Now we only have 2 Parameters. The original way of coding this is far cleaner, but writing the Lambda version was just to remind us how it really works.

And the next line of code:

```
instance semiringFn :: Semiring b => Semiring (a -> b) where
  add f g x = f x + g x
  zero = \_ -> zero
```

The Identity for `add` is `zero` and it appears to be an Infinite Loop since it has `zero` on both sides of the equal sign. But it's not.

The left-hand side `zero` is the `zero` for `Semiring (a -> b)`, i.e. `zero` for functions.

The right-hand side `zero` is the `zero` for `Semiring b`, i.e. `zero` for `b`, whatever Type `b` happens to be.

This may seem confusing at first, but remember, the whole right-hand side of `zero` is of Type `a` (see the Typeclass definition), which here is `(a -> b)`. This means that the Expression, `\_ -> zero`, takes an `a` (which it ignores) and returns a `b`, in this case `zero`. This shows that the right-hand `zero` is `b's zero`. And so there's no Infinite Loop here.

And the remaining lines of code:

```
instance semiringFn :: Semiring b => Semiring (a -> b) where
  add f g x = f x + g x
  zero = \_ -> zero
  mul f g x = f x * g x
  one = \_ -> one
```

`mul` and `one` are analogous to `add` and `zero`, whereas `one` is the Identity for `mul`.

8.19. Semiring Laws for Functions

We should verify the laws of our `Semiring` instance. Here they are again:

The laws for Commutative Monoid (a Monoid where the Binary Operator is Commutative) and Addition:

```
(R, +) is a Commutative Monoid with Identity of 0:

(a + b) + c = a + (b + c)   [Associativity] (Semigroup)
0 + a = a + 0 = a           [Identity] (Monoid)
a + b = b + a               [Commutative] (Commutative Monoid)
```

Next, the laws for Monoid and Multiplication:

```
(R, ·) is a Monoid with identity of 1:

(a · b) · c = a · (b · c)   [Associativity] (Semigroup)
1 · a = a · 1 = a           [Idenity] (Monoid)
```

Then the laws for Multiplication Distribution on Left and Right across Addition:

```
a · (b + c) = (a · b) + (a · c)       [Left Distributivity]
(a + b) · c = (a · c) + (b · c)       [Right Distributivity]
```

And finally the law for Multiplication with `0` Annihilates `R`:

```
0 · a = a · 0 = 0          [Annihilation]
```

As you can see, this will be a lengthy process, so we're only going to do one, but the process would be similar.

Let's prove the Right Identity for Addition:

```
a + 0 = a ❶
```

❶ Adding `0` on the right-hand side has no effect on the original value (or function in our case).

Here's our Semiring Instance for functions again:

```
instance semiringFn :: Semiring b => Semiring (a -> b) where
  add f g x = f x + g x
  zero = \_ -> zero
  mul f g x = f x * g x
  one = \_ -> one
```

And here's our proof:

```
b is a Semiring                          [Given]
add h zero = \x -> h x + zero x          [Definition] ❶
= \x -> h x + (\_ -> zero) x             [Substitution] ❷
= \x -> h x + zero                       [Function Application]
= \x -> h x                              [Semiring Right Identity for b] ❸
= h                                      [ETA Reduction] ❹
```

❶ We moved the `x` over to the right-hand side of the equal sign.

❷ We substitute using the definition for `zero`.

❸ Since we can assume `b` is a `Semiring`, we are using the Right Identity Law for `b` to make this reduction.

❹ ETA Reduction or Cancel on the Right as I like to call it.

So `add h zero` is equal to `h` proving that our implementation abides by the Right Identity Law. You can imagine how easy proving the Left Identity Law would be now that we have this proof.

Proving the other Laws are something that we programmers must do since PureScript will not help us here. But the proofs are pretty simple, if not tedious. And that's why we're not going to do them all. I encourage you to go through the exercise of proving all of them for Functions.

8.20. Ring

A Ring adds 1 more requirement to Semiring.

In a Ring, the Set has an Additive Inverse:

```
a + (-a) = 0       [Additive Inverse] ❶
```

❶ Ring has all the same laws of Semiring plus this one.

This makes `(R, +)` a Commutative Group whereas it's a Commutative Monoid for Semirings.

And by having an Additive Inverse, we can finally Subtract since subtracting is just adding the Inverse.

8.21. Ring Typeclass

In PureScript, we have a Typeclass for Ring:

```
class Semiring a <= Ring a where
  sub :: a -> a -> a

infixl 6 sub as -  ❶
```

❶ Notice that – has the same Precedence as + does in Semiring.

Here's the archetypical implementation of Ring for Int:

```
instance ringInt :: Ring Int where
  sub = intSub

foreign import intSub :: Int -> Int -> Int ❶
```

❶ Implemented in Javascript.

Here's the trivial implementation, for Unit:

```
instance ringUnit :: Ring Unit where
  sub _ _ = unit
```

And Ring for Functions:

```
instance ringFn :: Ring b => Ring (a -> b) where
  sub f g x = f x - g x
```

No real surprises. sub is analogous to add here.

8.22. Commutative Ring Typeclass

Rings are Commutative for Addition. But ***Commutative Rings*** are Commutative for Addition AND Multiplication.

The Typeclass is very simple since we cannot enforce Commutativity in PureScript:

```
class Ring a <= CommutativeRing a
```

The canonical implementations are trivial:

```
instance commutativeRingInt :: CommutativeRing Int
instance commutativeRingUnit :: CommutativeRing Unit
instance commutativeRingFn :: CommutativeRing b => CommutativeRing (a -> b)
```

Since there are no Methods in the `CommutativeRing` Typeclass, their instances are simple declarations.

8.23. Euclidean Ring

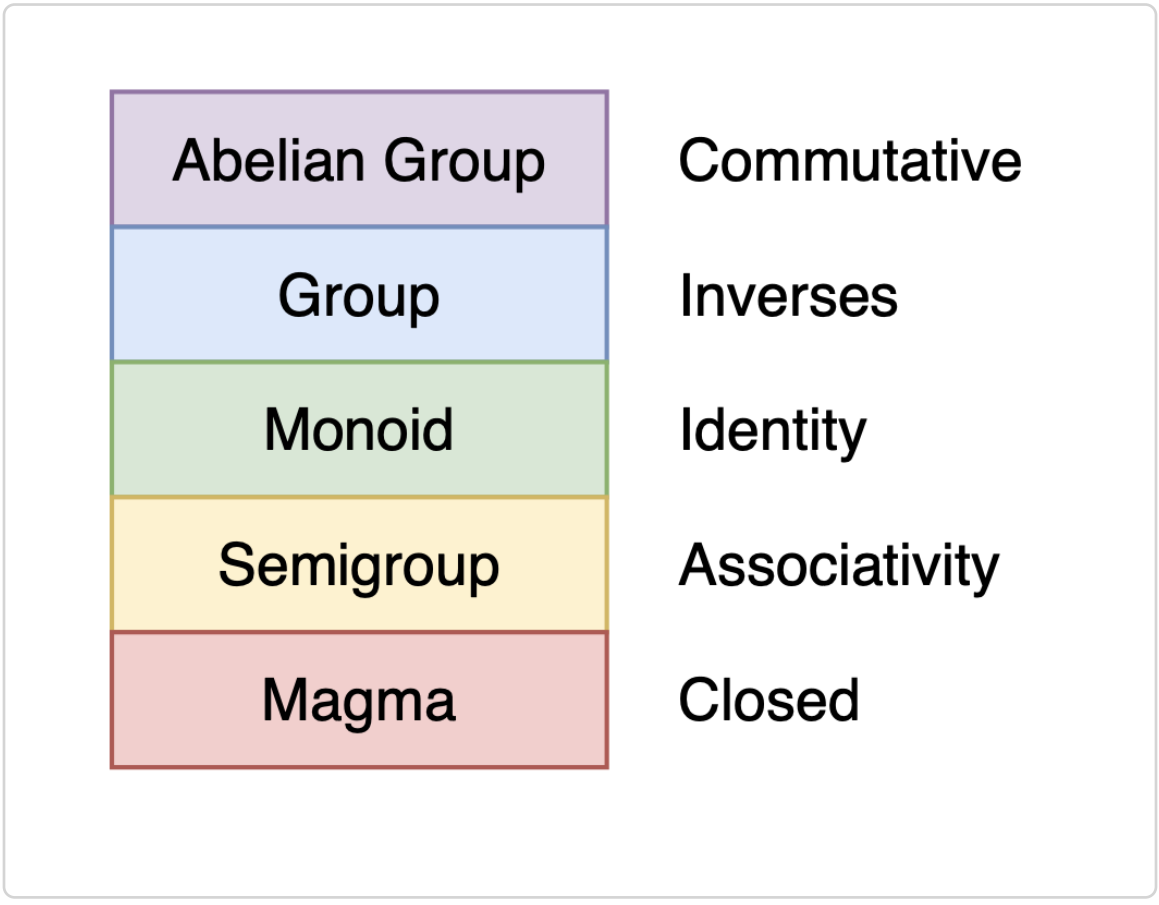
A ***Euclidean Ring*** is a Commutative Ring with 1 more requirement. Euclidean Rings support **Division**.

It has some pretty involved Laws, but we won't go into them here.

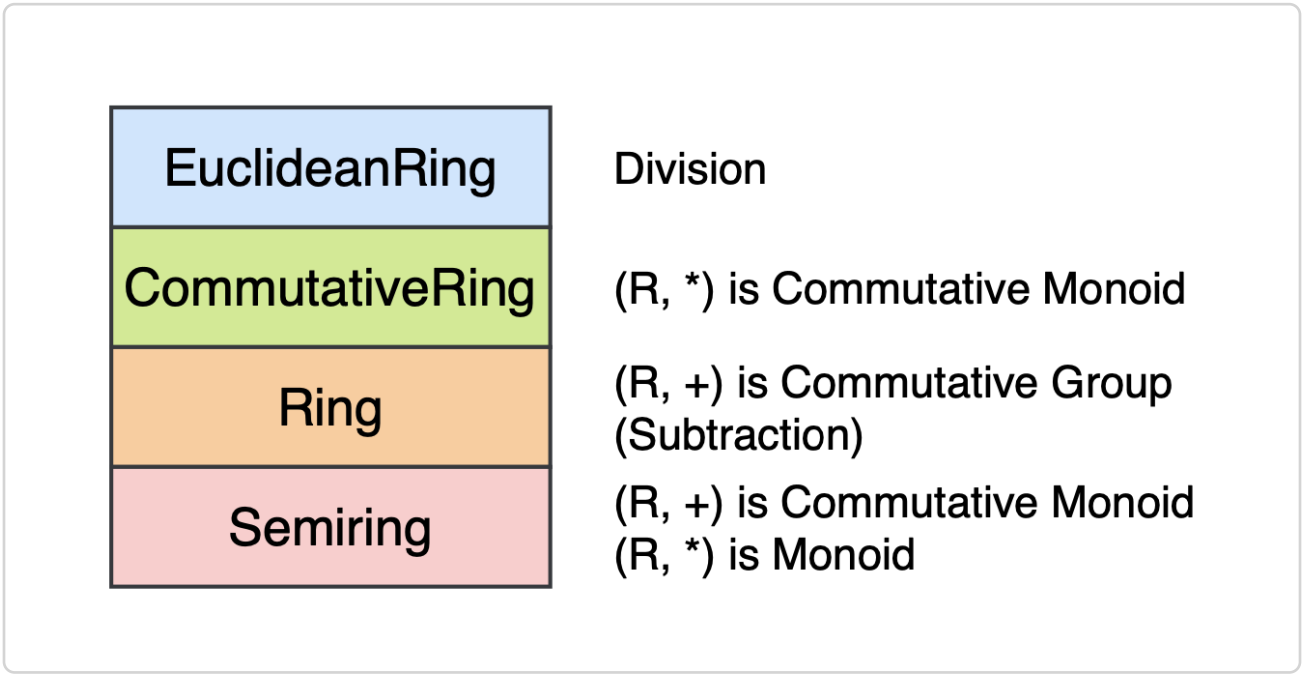
8.24. Summary

We can now see why the Types for simple Arithmetic Operators are so strange to us programmers. Hopefully, this Chapter has made them less so. In your day to day Functional Programming, you will use `Semigroup` and `Monoid` Typeclasses most of all.

But it's good to see where they fit into the hierarchy:



And it's also good to understand the Arithmetic Operator hierarchy so that we don't run scared from the complex names used in the Constraints:



Chapter 9. Coding Abstract Algebra

To help solidify what we learned in the last section, we're going to work with some of the Abstract Algebra concepts.

9.1. Writing Semigroup Typeclass

Return to your project that you've been coding in and create a file called Ch9.purs and add the following:

```
module Ch9 where

import Prelude (Unit) ❶

import Effect (Effect)
import Effect.Console (log)

test :: Effect Unit
test = do
  log "placeholder" ❷
```

❶ Just a reminder that we'd normally just include all of Prelude, but since we'll be overriding things from Prelude, we'll be very specific to only include things we need.

❷ This is here to appease the compiler. We'll remove it soon enough.

Now, without consulting Pursuit or looking back, try to write the Typeclass for Semigroup.

And for the Operator Alias, it should be Right-Associative with a Precedence of 5.

If you cannot remember the syntax for Typeclasses, look to the code you wrote in Ch7b.purs to refresh your memory. And after, if you're still stuck, see Hint section.

9.2. Hint for Semigroup Typeclass

A Semigroup's Binary Operator is called append in PureScript (and mappend in Haskell). So go ahead and try again before going on.

9.3. Code for Semigroup Typeclass

```
class Semigroup a where
  append :: a -> a -> a ❶

infixr 5 append as <>
```

❶ Combines 2 a's and produces an a. This meets the Magma requirement of being Closed under the Binary Operator, <>.

9.4. Writing Monoid Typeclass

Next, try to code the Typeclass for Monoid without consulting any other documentation.

Remember that a Monoid must also be a Semigroup.

9.5. Code for Monoid Typeclass

```
class Semigroup a <= Monoid a where ❶
  mempty :: a
```

❶ Semigroup is a Superclass of Monoid, i.e. every Monoid instance must also have an instance of Semigroup.

9.6. Writing Semigroup for AndBool

Let's start with the following Data Type:

```
import Data.Generic.Rep (class Generic) ❶
import Data.Show.Generic (genericShow)

data AndBool = AFalse | ATrue ❷
derive instance eqAndBool :: Eq AndBool ❸
derive instance genericAndBool :: Generic AndBool _ ❹

instance showAndBool :: Show AndBool where ❺
  show = genericShow
```

❶ We need to add these imports to use genericShow.

❷ Here is a Boolean-like Data Type whose Binary Operator will be a Logical And.

❸ We'll need this for test code later.

❹ We derive the Generic instance of our type so we can use genericShow.

❺ We implement Show to be used in our test code.

Remove the placeholder test code and replace it with:

```
log $ show $ ATrue <> ATrue ❶
log $ show $ ATrue <> AFalse ❷
log $ show $ AFalse <> AFalse ❸
```

❶ Prints ATrue.

❷ Prints AFalse.

❸ Prints AFalse.

You're also going to need to change your Prelude import to:

```
import Prelude (Unit, class Show, class Eq, ($), discard, show)
```

And finally, don't forget to change your Main.purs to call Ch9.test:

```
module Main where

import Prelude
import Effect (Effect)

import Ch9 as Ch9 ❶

main :: Effect Unit
main = Ch9.test ❶
```

❶ Change from Ch7 to Ch9.

We have a compiler error that we're expecting since we're not done.

Now, you're ready to write the Semigroup for AndBool with the Binary Operator of a Logical And. Please give it a try before reading on.

9.7. Hint for Semigroup for AndBool

You may be tempted to write the following:

```
instance semigroupAndBool :: Semigroup AndBool where
  append = (&&) -- COMPILER ERROR!! ❶ ❷
```

❶ The first compiler error will be because && was not imported from Prelude.

❷ Once you fix the import error, you'll get an error that there's no instance of HeytingAlgebra (whatever that is).

The && operator only works on instances of HeytingAlgebra, which is basically Boolean-like things. Since we haven't studied Heyting Algebras and we're trying to work on Semigroups at the moment, let's simply enumerating the different cases.

Give it another go before returning here.

9.8. Code for Semigroup for AndBool

```
instance semigroupAndBool :: Semigroup AndBool where
  append ATrue ATrue  = ATrue ❶
  append _ _          = AFalse
```

❶ The only time Logical And is True.

Running the test produces:

```
ATrue
AFalse
AFalse
```

9.9. Writing Monoid for AndBool

For the Monoid Instance we need to pick the appropriate Identity Value such that:

```
∀ a ∈ G ⇒ a • e = a
```

where • is our Binary Operator, which is <> in PureScript.

Add (==) to your Prelude imports and then the following test code to test the above Monoid requirement:

```
log $ show $ mempty <> ATrue == ATrue ❶
log $ show $ mempty <> AFalse == ATrue ❷
```

❶ Prints true.

❷ Prints false.

9.10. Code for Monoid for AndBool

```
instance monoidAndBool :: Monoid AndBool where
  mempty = ATrue
```

We picked ATrue to be our Identity Element since Logical And with ATrue will not affect the other value.

9.11. Verify Semigroup Laws for AndBool

Let's verify the Semigroup Laws which are:

```
∀ a, b, c ∈ S  ⇒  a • (b • c) = (a • b) • c
```

Essentially, we want to make sure that our Binary Operator is Associative. So, here's a function that'll get you most of the way there:

```
verifyAndBoolSemigroup :: Effect Unit
verifyAndBoolSemigroup = do
  log "Verifying AndBool Semigroup Laws (1 test)"
  log $ show $ ??? ❶
```

❶ Replace ??? with a Boolean expression that will evaluate to true if a • (b • c) = (a • b) • c.

Add the following to your test code:

```
verifyAndBoolSemigroup
```

Give it a try first, then read further.

9.12. Writing verifyAndBoolSemigroup

We want to print true when our Binary Operator is Associative. There a many variations of the following code, but this will do:

```
verifyAndBoolSemigroup :: Effect Unit
verifyAndBoolSemigroup = do
  log "verifying AndBool Semigroup Laws (1)"
  log $ show $ (AFalse <> ATrue) <> ATrue == AFalse <> (ATrue <> ATrue) ❶
```

❶ Prints `true`.

If our Binary Operator is Associative, then this should work. Granted a more exhaustive test of either random values or some exhaustive number of tests is ideal. But since we know && is Associative and our implementation of the `Semigroup` instance mirrors &&, one simple test should do. Feel free to others.

9.13. Verify Monoid Laws for `AndBool`

Next, let's verify the Monoid Laws which are:

```
e ∈ M, ∀ a ∈ M  ⇒  a • e = e • a = a
```

We have to make sure our `mempty` follows `a • e = e • a = a`, where `mempty` is `e`.

Here's the template for that function:

```
verifyAndBoolMonoid :: Effect Unit
verifyAndBoolMonoid = do
  log "Verifying AndBool Monoid Laws (2 tests)"
  log $ show $ ??? ❶
  log $ show $ ??? ❷
```

❶ The `???` are to be replaced with a Boolean that evaluates to `true` if `a • e = e • a = a` holds for `ATrue`.

❷ The same as the above `???` except for `AFalse`.

Add the following to your test code:

```
verifyAndBoolMonoid
```

9.14. Writing `verifyAndBoolMonoid`

We want to show that our `mempty` works with both `ATrue` and `AFalse`:

```
verifyAndBoolMonoid :: Effect Unit
verifyAndBoolMonoid = do
  log "Verifying AndBool Monoid Laws (2 tests)"
  log $ show $ mempty <> ATrue == ATrue <> mempty && ATrue <> mempty == ATrue
❶ ❷
  log $ show $ mempty <> AFalse == AFalse <> mempty && AFalse <> mempty ==
AFalse ❸
```

❶ You'll need to add `(&&)` to the `Prelude` import list.

❷ Prints `true`.

❸ Prints `true`.

We have shown that `e` can be applied on either the right or left-hand side to any `a` from the Set, `{ ATrue, AFalse }`, and it has no affect on the value.

9.15. Writing `Semigroup` and `Monoid` for `OrBool`

Starting with the Data Type:

```
data OrBool = OFalse | OTrue
```

where the Binary Operator is Logical Or.

Look back at what you did with `AndBool` and implement `Eq` and `Show`.

You should be able to easily implement `Semigroup` and `Monoid` instances before moving on.

9.16. Code for `Semigroup` and `Monoid` for `OrBool`

The code for `OrBool` will look nearly identical to `AndBool`:

```
data OrBool = OFalse | OTrue
derive instance eqOrBool :: Eq OrBool
derive instance genericOrBool :: Generic OrBool _

instance showOrBool :: Show OrBool where
  show = genericShow
```

The `Semigroup` instance will only differ from our `AndBool` implementation how Logical And differs from Logical Or:

```
instance semigroupOrBool :: Semigroup OrBool where
  append OFalse OFalse  = OFalse ❶
  append _ _            = OTrue
```

❶ Logical Or is False only if both values are False.

And the Identity value for Logical Or is the False value, which is `OFalse` for our Type:

```
instance monoidOrBool :: Monoid OrBool where
  mempty = OFalse ❶
```

❶ When `OFalse` is Logically Or'd with anything value, that value is unchanged.

9.17. Verify Semigroup and Monoid Laws for `OrBool`

We want to write similar routines for `OrBool` to verify the Laws. Look to what you did with `AndBool` and give it a try before venturing ahead.

Don't forget to add them to your test code.

9.18. Code for Semigroup and Monoid Laws for `OrBool`

Here is the function for verifying the Semigroup Laws:

```
verifyOrBoolSemigroup :: Effect Unit
verifyOrBoolSemigroup = do
  log "Verifying OrBool Semigroup Laws (1 test)"
  log $ show $ (OFalse <> OTrue) <> OTrue == OFalse <> (OTrue <> OTrue) ❶ ❷
```

❶ We used nearly the same test as we used in `verifyAndBoolSemigroup`.

❷ Prints `true`.

Many different tests can be written here. Yours may differ, but as long as it changes Associativity on the same expression, your test should be good.

And here is the Monoid Laws verification routine:

```
verifyOrBoolMonoid :: Effect Unit
verifyOrBoolMonoid = do
  log "Verifying OrBool Monoid Laws (2 tests)"
  log $ show $ mempty <> OTrue == OTrue <> mempty && OTrue <> mempty == OTrue
❶ ❷
  log $ show $ mempty <> OFalse == OFalse <> mempty && OFalse <> mempty ==
OFalse ❸
```

❶ We used nearly the same test as we used in `verifyAndBoolMonoid`.

❷ Prints `true`.

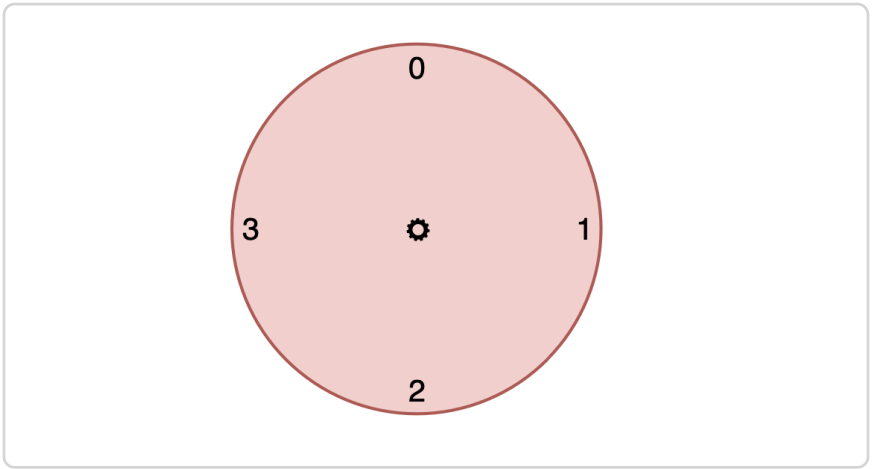
❸ Prints `true`.

Add the following to your test code:

```
verifyOrBoolSemigroup
verifyOrBoolMonoid
```

9.19. Writing Mod4 Data Type

Remember our clock with only 4 positions:



We can represent this as a Coproduct (Sum Type). Let's call it Mod4. Give it a shot before reading on.

9.20. Code for Mod4 Data Type

```
data Mod4 = Zero | One | Two | Three
```

We call our Data Type Mod4 because it represents Integers Mod 4, $\mathbb{Z}_4$. We represent the numbers with the

Data Constructors, Zero for 0, One for 1, etc.

9.21. Writing Semigroup for Mod4

In the previous Chapter, we proved that, $\mathbb{Z}_4$ was a Group. Now we're going write the code for that.

To do so, we must start with a Semigroup instance. We know the Set of values, { 0, 1, 2, 3 }, which we represented in PureScript with { Zero, One, Two, Three }.

Next, we have to determine the Binary Operator. When we were doing our Clock Arithmetic, we were doing Addition. So our operator is +.

We've decided everything we need to implement the Semigroup instance for Mod4. Please try it before reading ahead to the Hint section.

9.22. Hint for Semigroup for Mod4

We shouldn't expect to leverage the normal Addition operation here. We'll have to manually specify each possible operation.

This will involve using Pattern Matching. Once you get it coded, take a good look at the Zero case and see if you can simplify it.

Take another swing and see how you did by moving on to the next section.

9.23. Code for Semigroup for Mod4

We're going to write this one section at a time. First, Zero case:

```
instance semigroupMod4 :: Semigroup Mod4 where
  append Zero x       = x
  append x Zero       = x
```

We know that Zero added to anything, x, is that same value, x. And it doesn't matter which side we add on.

Remember, we are Pattern Matching, so we have to match with Zero on both sides of the <> operator.

Next we add the One case:

```
instance semigroupMod4 :: Semigroup Mod4 where
  append Zero x       = x
  append x Zero       = x

  append One One      = Two
  append One Two      = Three
  append One Three    = Zero
```

Here we make sure to wrap around when we get past Three. If this doesn't make sense, refer back to our clock face and add by counting as you move Clockwise.

Next, we add the Two Case:

```
instance semigroupMod4 :: Semigroup Mod4 where
  append Zero x       = x
  append x Zero       = x

  append One One      = Two
  append One Two      = Three
  append One Three    = Zero

  append Two One      = Three
  append Two Two      = Zero
  append Two Three    = One
```

And finally, we add the Three case:

```
instance semigroupMod4 :: Semigroup Mod4 where
  append Zero x       = x
  append x Zero       = x

  append One One      = Two
  append One Two      = Three
  append One Three    = Zero

  append Two One      = Three
  append Two Two      = Zero
  append Two Three    = One

  append Three One    = Zero
  append Three Two    = One
  append Three Three  = Two
```

9.24. Writing Monoid for Mod4

The Monoid instance simply defines the Identity Value.

Take a stab before moving forward.

9.25. Code for Monoid for Mod4

```
instance monoidMod4 :: Monoid Mod4 where
  mempty = Zero
```

Zero is the Additive Identity, i.e. adding Zero to anything element of our Set will return that value.

9.26. Writing Group Typeclass

Try to write the code for the Group Typeclass without referring to any reference materials.

The only hint I'll give is that the one and only Method is called ginverse. Give it a try before moving on.

9.27. Code for Group Typeclass

```
class Monoid a <= Group a where ❶
  ginverse :: a -> a
```

❶ Monoid is a Superclass of Group.

Here ginverse takes a value of Type a and returns its Inverse value also of Type a. That's because a Group is also a Magma.

9.28. Writing Group for Mod4

When writing the Group instance, you have to determine the Inverse values are for every value in the Set. Remember that the Identity Element is its own Inverse.

Also, keep in mind, the Binary Operator is Addition, which means to determine a value's Inverse, we must ask ourselves, "What can I add to this value to get our Identity Value?"

Refer to our clock face if that helps.

Give this a try and see how you do.

9.29. Code for Group for Mod4

Let's start with Zero:

```
instance groupMod4 :: Group Mod4 where
  ginverse Zero   = Zero
```

Since Zero is our Identity Value, it is its own Inverse.

Next we code One:

```
instance groupMod4 :: Group Mod4 where
  ginverse Zero   = Zero
  ginverse One    = Three
```

We have to add Three before we get back around to Zero.

Next, we add Two:

```
instance groupMod4 :: Group Mod4 where
  ginverse Zero   = Zero
  ginverse One    = Three
  ginverse Two    = Two
```

We have to add one less than the One case to get back to Zero.

And finally, the last case:

```
instance groupMod4 :: Group Mod4 where
  ginverse Zero  = Zero
  ginverse One   = Three
  ginverse Two   = Two
  ginverse Three = One
```

Notice that the Addition of `One` and `Three` is the same as `Three` and `One`. Our operator doesn't have to be Commutative, but, as it turns out, it is. So, if we wanted to, we could create a `Commutative` instance for our Type.

```
class Semigroup g <= Commutative g

instance commutativeMod4 :: Commutative Mod4
```

Remember that our Type only need be a `Semigroup` to make it `Commutative`.

9.30. Writing Eq and Show for Mod4

Before we can write test code for our new Data Type, we'll need some helpful instances, viz. `Eq` and `Show`.

We've done this a bunch of times now, so it should be fairly easy. Write them yourself before continuing.

9.31. Code for Eq and Show for Mod4

The code is exactly like `AndBool` and `OrBool`:

```
data Mod4 = Zero | One | Two | Three
derive instance eqMod4 :: Eq Mod4
derive instance genericMod4 :: Generic Mod4 _

instance showMod4 :: Show Mod4 where
  show = genericShow
```

9.32. Writing verifyMod4Semigroup for Mod4

We should make sure that we've coded a law-abiding Semigroup instance.

We can look to `AndBool` and `OrBool` to see how we might create a verification function.

Don't forget to add this function call to your test code.

9.33. Code for verifyMod4Semigroup for Mod4

```
verifyMod4Semigroup :: Effect Unit
verifyMod4Semigroup = do
  log "Verifying Mod4 Semigroup Laws (1 test)"
  log $ show $ (One <> Two) <> Three == One <> (Two <> Three) ❶
```

❶ Prints true.

Once again, there are lots of variations of this test code. Yours can vary and still be correct. The main thing is to make sure that the right and left-hand sides only differ in their Parentheses.

9.34. Writing verifyMod4Monoid for Mod4

Next, we'll want to check that our Monoid is good citizen.

Look to AndBool and OrBool for help, add verifyMod4Monoid to your test code and see how it goes before you move on to the next section.

9.35. Code for verifyMod4Monoid for Mod4

```
verifyMod4Monoid :: Effect Unit ❶
verifyMod4Monoid = do
  log "Verifying Mod4 Monoid Laws (1 test)"
  log $ show $ mempty <> One == One <> mempty && One <> mempty == One ❷
```

❶ Add a call to this Function in your test code.

❷ Prints true.

Notice that we only have 1 test here. We only need this test to show that mempty doesn't change values and since this test has Zero on the left-hand side and Zero on the right, it covers both cases.

9.36. Writing Semigroup for Maybe

Sometimes, as we have seen, there can be more than one way to combine things. For example, with Ints, we can combine them by adding or multiplying. And we accomplished this by creating a newtype for each cases, viz. AddInt and MultInt.

In writing, Semigroup for Maybe, there are two possible situations. Let's take an example:

```
Nothing <> Just 10 <> Nothing <> Just 20
```

Looking at this should conjure up a few questions. What does `<>` mean in this context? What should this evaluate to? And how do we choose? And what's our Identity?

The only reasonable value for Identity is `Nothing`. So that means that `Nothings` have NO effect on our outcome. So the example above would reduce to:

```
Just 10 <> Just 20
```

Now there are 2 possibilities here. We could require that our `a` be a Semigroup, i.e. `Ints` would have to be a Semigroup, which they are not. `Strings`, however, are and we could force that requirement giving us:

```
Just "abc" <> Just "123" = Just "abc123"
```

But that's a bit limiting and there are other ways of accomplishing this result. So let's choose, for now, to not add that Constraint.

That means there's only 2 possible answers to our original problem:

```
Just 10 <> Just 20 = Just 10 ❶
Just 10 <> Just 20 = Just 20 ❷
```

❶ Take the first `Just`.

❷ Take the second `Just`.

That's great, but what about multiple:

```
Just 10 <> Just 20 <> Just 30 = ???
```

Default Associativity here can help answer that. Go look up the Associativity of `<>` on `Pursuit` and rewrite the above with the implied Parentheses before moving on.

`<>` is Right-Associative meaning Parentheses will collect on the right:

```
(Just 10 <> (Just 20 <> Just 30)) = ???
```

Now there are 3 possibilities:

```
(Just 10 <> (Just 20 <> Just 30)) = Just 10 ❶
(Just 10 <> (Just 20 <> Just 30)) = Just 20 ❷
(Just 10 <> (Just 20 <> Just 30)) = Just 30 ❸
```

❶ Take the first `Just`.

❷ Sometimes take first `Just` then take second `Just`. This case doesn't make sense.

❸ Take the last `Just`.

To get a result of `Just 20`, we'd first have to pick the `second` value, i.e. the Parenthetical. Next, to evaluate what's inside the Parentheses, we'd have to pick the `first`. This is just absurd since we can't just randomly pick. So we'll disregard this case.

But, we're no better off with this new example, since we still have 2 possibilities. So just like with `AddInt` and `MultInt`, let's create 2 `newtypes` for both situations:

```
newtype First a = First (Maybe a)
newtype Last a = Last (Maybe a)
```

We're calling this `First` and `Last` to signify that in a long line of `appends`, we take the first `non-Nothing` value or the last.

9.37. Writing `Semigroup` and `Monoid` for `First`

Go ahead and write the Semigroup and Monoid instances for `First`, keeping in mind that `Nothing` is our Identity and therefore has no impact on any value when applied with `<>`.

Try it yourself before reading the solution.

9.38. Code for `Semigroup` and `Monoid` for `First`

First, the definition:

```
newtype First a = First (Maybe a)
```

Next, the `Semigroup` instance:

```
import Data.Maybe (Maybe (..)) ❶

instance semigroupFirst :: Semigroup (First a) where
  append (First Nothing) last = last ❷
  append first _              = first ❸
```

❶ Added this import.

❷ If the first is `Nothing` then keep the `last`.

❸ Otherwise, always keep the `first`.

We keep the first value UNLESS the first is a `Nothing`. Notice that once we get our first `non–Nothing` that we keep it indefinitely since the whole right-hand side is ignored. This will effectively short-circuit the computation once we find our first `non–Nothing`.

Finally, we write the `Monoid` instance:

```
instance monoidFirst :: Monoid (First a) where
  mempty = First Nothing
```

Notice that it's not just `Nothing` (no pun intended), but it's `First Nothing`.

9.39. Writing `Semigroup` and `Monoid` for `Last`

Having done `First`, writing `Last` should be easy. Give it try before proceeding on to the solution.

9.40. Code for `Semigroup` and `Monoid` for `First`

First, the definition:

```
newtype Last a = Last (Maybe a)
```

Next, the `Semigroup` instance:

```
instance semigroupLast :: Semigroup (Last a) where
  append first (Last Nothing) = first ❶
  append _ last               = last ❷
```

❶ If last is `Nothing`, then keep `first`.

❷ Otherwise, keep `last`.

Once again, we keep the last value unless it's a Nothing. Unlike, First, we have no opportunity to ignore the right-hand side which means that we will NOT short-circuit with Last. This makes combining Last less efficient than First since we have to evaluate every value of the expression:

```
First (Just 1) <> (First (Just 2) <> First Nothing) ❶
Last  (Just 1) <> (Last  (Just 2) <> Last  Nothing) ❷
```

❶ We don't need to evaluate (First (Just 2) <> First Nothing) since we found our first non-Nothing with First (Just 1).

❷ We have to evaluate (Last (Just 2) <> Last Nothing) because we have to keep looking for the LAST non-Nothing.

And finally, the Monoid instance:

```
instance monoidLast :: Monoid (Last a) where
  mempty = Last Nothing
```

Once again, it's not just Nothing, but Last Nothing.

Let's write test code:

```
  -- COMPILER ERROR!!
  log $ show $ First Nothing <> First (Just 77) ❶
  log $ show $ Last (Just 1) <> Last (Just 99) ❷
```

❶ Prints (First (Just 77)).

❷ Prints (Last (Just 99)).

Write Show instances for both First and Last to remove the compiler error.

Here are those changes:

```
newtype First a = First (Maybe a)
derive instance genericFirst :: Generic (First a) _ ❶

instance showFirst :: Show a => Show (First a) where ❷
  show = genericShow

newtype Last a = Last (Maybe a)
derive instance genericLast :: Generic (Last a) _ ❶

instance showLast :: Show a => Show (Last a) where ❷
  show = genericShow
```

❶ Make both instances of `Generic` so we can use `genericShow`.

❷ Implement `Show` instance for both.

Chapter 10. Folds

When we look back at the recursive functions we wrote, there are some common features that we may notice.

First, is that many times we pass State to our recursive call:

```
index :: ∀ a. List a -> Int -> Maybe a
index Nil _ = Nothing
index l i = go 0 l where ❶
  go _ Nil     = Nothing
  go ci (x : xs) = if ci == i then Just x else go (ci + 1) xs ❷ ❸

infixl 8 index as !!
```

❶ We pass the Initial State here which is the index of `0`.

❷ When we recurse, we add `1` to the current index, `ci` and pass it to `go`.

❸ We also Short-circuit here.

In the above example, our `go` function exists for the sole purpose of passing State without burdening the caller of the original function with having to pass the Initial State. In this case, our State is the current index we are visiting.

Another feature of our recursive functions is Short-circuiting as we can see in the above example and in the following:

```
findIndex :: ∀ a. (a -> Boolean) -> List a -> Maybe Int
findIndex _ Nil  = Nothing
findIndex pred l = go 0 l where
  go _ Nil      = Nothing
  go i (x : xs) = if pred x then Just i else go (i + 1) xs ❶
```

❶ Short-circuit when we find what we're looking for.

We will terminate the recursion when we either exhaust the `List` or find what we're looking for. When we find what we're searching for, then we'll Short-circuit the recursion, i.e. we will not finish looking through the whole `List`.

And finally, some recursive functions always exhaust the `List`, i.e. they recurse once per element:

```
length :: ∀ a. List a -> Int
length Nil       = 0 ❶
length (x : xs) = 1 + length xs
```

❶ Terminate when have exhausted the `List`.

These are the types of functions that we're going to look at more closely.

Let's start with `length`:

```
length :: ∀ a. List a -> Int
length Nil       = 0
length (x : xs) = 1 + length xs ❶
```

❶ Not Tail Recursive.

Notice how we have no State to pass here. The State is maintained in the Stack. We need to return to add `1`. This is how we count.

But in the Tail Recursive version of `length`, we explicitly pass State, which is evidenced by the existence of a `go` function:

```
length :: ∀ a. List a -> Int
length l = go 0 l where
  go :: ∀ a. Int -> List a -> Int ❶
  go acc Nil       = acc
  go acc (x : xs) = go (acc + 1) xs
```

❶ The first Parameter is the accumulated count of the length.

These types of functions are perfect for a process called `Fold`.



Fold is generalized way to iterate with State.

A major limitation of Folds is that they do NOT Short-circuit. This means that we can rewrite `length` using a Fold, but not `index` or `findIndex`.

10.1. Fold by other Names

In many non-Functional Languages, `fold` is also called `reduce`. In the case of our `length` function, we ***reduce*** our `List` to a single `Int`. But this is NOT always the case. We can use `fold` to break down one structure and build up another, e.g. process a `List` and generate a `Map` (a Key/Value store).

And in Category Theory, you'll hear the term ***Catamorphism*** (***Cata*** means ***down*** and ***morph*** means ***shape***), essentially meaning ***to break down structure***.

This is opposed to ***Anamorphism*** (***Ana*** means ***up*** and ***morph*** means ***shape***), essentially meaning ***to build up structure***.

If you forget these terms, you can remember Anabolic Steroids which builds muscle or Catabolic Exercise which breaks down muscle tissue and burns fat.

The point here is that `folds` are called `Catamorphisms` because they break down the original structure. There's also another concept called `unfold`, which are used far less often, and are called `Anamorphisms` since they build up structure from a single start value.

We are going to concern ourselves with just `folds` here though.

10.2. Foldable Typeclass

In PureScript, a Type can be `Foldable` if it has an instance of `Foldable`. Here is that Typeclass:

```
class Foldable f where
  foldr :: ∀ a b. (a -> b -> b) -> b -> f a -> b
  foldl :: ∀ a b. (b -> a -> b) -> b -> f a -> b
  foldMap :: ∀ a m. Monoid m => (a -> m) -> f a -> m
```

Let's break this down one function at a time. First `foldr`:

```
foldr :: ∀ a b. (a -> b -> b) -> b -> f a -> b
```

Let's break down each of its Parameters:

```
(a -> b -> b)     -- function to process the Foldable element
b                 -- initial value
f a               -- Foldable type `f`, e.g. `List` or `Array`
```

`foldr` returns a `b` as the final result of the fold operation.

The name of the function ends in `r` meaning that the operations are `Right-Associative`. We'll see a good example of this soon.

Next, we'll look at `foldl`:

```
foldl :: ∀ a b. (b -> a -> b) -> b -> f a -> b
```

It is identical to `foldr` except for the First Parameter. The First Parameter is a function whose Parameters are `flipped` when compared with `foldr`. We'll see why this is important soon.

And finally, `foldMap`:

```
foldMap :: ∀ a m. Monoid m => (a -> m) -> f a -> m
```

Let's break down each of its Parameters:

```
(a -> m)    -- function to convert the element to a Monoid
f a         -- Foldable type `f`, e.g. `List` or `Array`
```

This function returns an `m`, which is a `Monoid`.

Can you think of why it's a `Monoid` and not a `Semigroup`? It could've been a `Semigroup` but then we'd burden the user with having to pass an Initial Value, which if is possible, means that the Type is probably already a `Monoid`.

Notice how `foldMap` doesn't require an Initial Value like `foldl` and `foldr`. That's because `foldMap` works with `Monoids` and can use the Type's `mempty` as its Initial Value.

Also, notice that the variable `f` is being used here as the collection or structure that we're folding over. Once again, that's because it stands for `Functor` since most `Functors` are also `Foldable`. Don't worry, we will spend plenty of time on `Functors` soon enough.

10.3. Fold Associativity

`foldr` is Right-Associative and `foldl` is Left-Associative meaning:

```
foldr (-) 99 [1, 2, 3] == 1 - (2 - (3 - 99)) == -97
foldl (-) 99 [1, 2, 3] == ((99 - 1) - 2) - 3 == 93
```

Notice where the Initial Value, `99`, is. It's on the Far Right for `foldr` and the Far Left for `foldl`.

To better understand the differences, let's rewrite the `(-)` to show the moving parts:

```
foldr (\n acc -> n - acc) 99 [1, 2, 3] == -97
foldl (\acc n -> acc - n) 99 [1, 2, 3] == 93
```

Notice, in `foldr`, our Accumulator, `acc`, is on the Right of the Parameter list and on the Right of the Binary Operator, `–`.

Also, notice, in `foldl`, `acc` is on the Left of the Parameter list and on the Left side of `–`.

This is how we accomplish the appropriate Associativity. And this is also why the function that is passed to `foldr` is the flipped version of the function that's passed to `foldl`:

```
foldr :: ∀ a b. (a -> b -> b) -> b -> f a -> b
foldl :: ∀ a b. (b -> a -> b) -> b -> f a -> b
```

Remember, `flip` swaps the first 2 Parameters of a function.

10.4. Foldable for `List`

Let's look at the details for the implementation of `Foldable` for `List`.

First, we'll look at the instance definition:

```
instance foldableList :: Foldable ??? where
```

What we've been putting in place of `???` in other Typeclasses is the Type with all of its Parameters, e.g. `List a` or `Maybe a` as in:

```
instance eqMaybe :: Eq a => Eq (Maybe a) where
  ... ❶
```

❶ Implementation details are not important here.

And that's because if we look at the `Eq` Typeclass definition, we'll find:

```
class Eq a where
  eq :: a -> a -> Boolean
```

The Type Parameter to the `Eq` class is `a` and its ***Kind*** is `Type`.



Types categorize Values in the same way as Kinds categorize Types, i.e. a Value has a Type and a Type has a Kind. In other words, the type of a Value is called a Type, whereas the type of a Type is called a Kind.

Compare `Eq's` definition above with that of `Foldable`:

```
class Foldable f where
  foldr :: ∀ a b. (a -> b -> b) -> b -> f a -> b
  foldl :: ∀ a b. (b -> a -> b) -> b -> f a -> b
  foldMap :: ∀ a m. Monoid m => (a -> m) -> f a -> m
```

The Type Parameter to the `Foldable` class is `f` and when we look at how `f` is used in the Type Signatures of the Methods, we can see that `f` takes another Type, viz `a`.

That means that `f's` Kind is `Type -> Type`, i.e. `f` is a ***Type Constructor***. Just like Data Constructors, we can think of Type Constructors like functions except their Parameters are Types.

Here `f` is a Type Constructor that is expecting a single Parameter of a `Type` and it will return a `Type`.

To better understand the similarities between Type and Data Constructors, let's look at the Type Signatures for some common Data Constructors:

```
Just :: ∀ a. a -> Maybe a
Left :: ∀ a. a -> Either a b
Right :: ∀ b. b -> Either a b
```

Notice how Data Constructors take a Value and return a Value.

And now some examples of ***Kind Signatures*** for Type Constructors:

```
Maybe :: Type -> Type
Either :: Type -> Type -> Type
```

Notice how Type Constructors take Type and return a Type.

Now that we understand these difference, it should be clear that we need to specify the class definition differently than we've done so far:

```
instance foldableList :: Foldable List where
```

`List's` Kind is `Type -> Type` which matches the Type Signature of `f`. Notice that we didn't mention `a` anywhere.

When `f` is replaced by `List`, the Type Signatures of the Methods specialize to:

```
instance foldableList :: Foldable List where
  foldr :: ∀ a b. (a -> b -> b) -> b -> List a -> b
  foldl :: ∀ a b. (b -> a -> b) -> b -> List a -> b
  foldMap :: ∀ a m. Monoid m => (a -> m) -> List a -> m
```

Notice that all we did, was replace `f` with `List`. And we can do this because `f` takes a single Type Parameter `a` as so does `List`. We could've just as easily replaced `f` with `Array` or `Maybe`.

But we could not replace it with `Either` since it's Kind Signature is `Type -> Type -> Type`. Instead we'd have to fix the first Parameter and replace `f` with `Either a`. That works because we're partially applying the Type Constructor with its first Parameter. `Either a` has a Kind Signature `Type -> Type` which now matches `List`, `Array`, `Maybe` and `f`.

Now on to writing `foldr`. First, the base-case:

```
instance foldableList :: Foldable List where
  foldr _ acc Nil = acc
```

Here we just provide the accumulated result. This is either our Initial State, if we started with an empty `List`, or the one we've been passed when we've exhausted our `List`.

Next, we write the recursive case:

```
instance foldableList :: Foldable List where
  foldr _ acc Nil       = acc
  foldr f acc (x : xs)  = ???
```

Before we can determine what to write on the right-hand side of the equal sign, we have to keep in mind that we must write this so it's Right-Associative. Let's see what that looks like again:

```
foldr (-) 99 [1, 2, 3] == 1 - (2 - (3 - 99)) == -97
```

Notice that our Initial Value, `99`, isn't used in our computation until we reach the end of the `List`. This means that we need to keep passing this Initial Value recursively until we exhaust the `List`. We can see that the base-case will simply returns this Initial Value back to us and then we can start doing our computations.

This is good because we want the LAST element to be used in the FIRST computation we perform. That's what it means to be Right-Associative. If you were to calculate the above example, the first thing you would do is `3 - 99`. Our function needs to perform the same way.

So our Strategy here, is to keep recursing, passing our Initial Value and only when we get to the end, will we

perform the function (in our example, Subtraction) and return the result to the previous call, which in turn will perform the function with the previous value and so on and so on:

```
instance foldableList :: Foldable List where
  foldr _ acc Nil       = acc ❶
  foldr f acc (x : xs)  = f x (foldr f acc xs) ❷
```

❶ Base case returns the Initial Value.

❷ Here we can see that we defer the function call `f` until `(foldr f acc xs)` is computed. And each subsequent recursion will do the same until the last element is computed with the Initial Value (the value returned from the base-case).

To get a different perspective on how this works, let's look at executing this step by step by doing Algebraic Substitution:

```
foldr (-) 99 [1, 2, 3] =                  -- initial call
(-) 1 (foldr (-) 99 [2, 3]) =             -- substitution into ❷
(-) 1 ((-) 2 (foldr (-) 99 [3])) =        -- substitution into ❷
(-) 1 ((-) 2 ((-) 3 (foldr (-) 99 []))) = -- substitution into ❷
(-) 1 ((-) 2 ((-) 3 99)) =                -- substitution into ❶
(-) 1 ((-) 2 (3 - 99)) =                  -- rewrite prefixed as infixed
(-) 1 (2 - (3 - 99)) =                    -- rewrite prefixed as infixed
1 - (2 - (3 - 99)) =                      -- rewrite prefixed as infixed
```

Notice how 99, our Initial Value, got passed all the way down until it was needed in the computation with our last element, 3.

Next, we write the base-case for `foldl`:

```
instance foldableList :: Foldable List where
  foldr _ acc Nil       = acc
  foldr f acc (x : xs)  = f x (foldr f acc xs)
  foldl _ acc Nil       = acc
```

And then the recursive case:

```
instance foldableList :: Foldable List where
  foldr _ acc Nil       = acc
  foldr f acc (x : xs)  = f x (foldr f acc xs)
  foldl _ acc Nil       = acc
  foldl f acc (x : xs)  = ???
```

Before we can determine what to write on the right-hand side of the equal sign, we have to keep in mind that we must write this so it's Left-Associative. Let's see what that looks like again:

```
foldl (-) 99 [1, 2, 3] == ((99 - 1) - 2) - 3 == 93
```

In this case, we use the Initial Value right away and pass that result to the recursive call as the NEW accumulator value:

```
instance foldableList :: Foldable List where
  foldr _ acc Nil       = acc
  foldr f acc (x : xs)  = f x (foldr f acc xs)
  foldl _ acc Nil       = acc ❶
  foldl f acc (x : xs)  = foldl f (f acc x) xs ❷
```

Let's look at executing this step by step by doing Algebraic Substitution:

```
foldl (-) 99 [1, 2, 3] =                  -- initial call
foldl (-) ((-) 99 1) [2, 3] =             -- substitution into ❷
foldl (-) ((-) ((-) 99 1) 2) [3] =        -- substitution into ❷
foldl (-) ((-) ((-) ((-) 99 1) 2) 3) [] = -- substitution into ❷
(-) ((-) ((-) 99 1) 2) 3 =                -- substitution into ❶
(-) ((-) (99 - 1) 2) 3 =                  -- rewrite prefixed as infixed
(-) ((99 - 1) - 2) 3 =                    -- rewrite prefixed as infixed
((99 - 1) - 2) - 3 =                      -- rewrite prefixed as infixed
```

Before moving on to `foldMap`, we can make a few observations. When we use `foldl` or `foldr`, we are passing State using the `acc` variable. We've been referring to it as our Accumulator, but it's really our State.

Since there is no Global State in Functional Programming, the best we can do here is pass it from call to call:

```
instance foldableList :: Foldable List where
  foldr _ acc Nil       = acc ❶
  foldr f acc (x : xs)  = f x (foldr f acc xs) ❷
  foldl _ acc Nil       = acc ❸
  foldl f acc (x : xs)  = foldl f (f acc x) xs ❹
```

❶ Here `acc` is our Initial State.

❷ Our Initial State gets passed.

❸ Here `acc` is our Final State.

❹ The New State gets passed.

In the case of `foldl`, we PASS our New State to the Recursive Call, whereas in `foldr`, we RETURN our New State from the Recursive Call.

It's important to take a minute to let this sink in. In Functional Programming, we can only pass in State changes or return State changes. This is going to be a concept we revisit when we look at the ***State Monad***.

Now, we're ready to write `foldMap`.

First, we add the base-case:

```
instance foldableList :: Foldable List where
  foldr _ acc Nil       = acc
  foldr f acc (x : xs)  = f x (foldr f acc xs)
  foldl _ acc Nil       = acc
  foldl f acc (x : xs)  = foldl f (f acc x) xs
  foldMap _ Nil         = mempty ❶
```

❶ Base-case.

Luckily, Monoids has an Empty Element. So when we are given an empty `List`, we have a Value to return, viz. the Monoid's Empty Element, `mempty`.

And finally, we add the recursive case:

```
instance foldableList :: Foldable List where
  foldr _ acc Nil       = acc
  foldr f acc (x : xs)  = f x (foldr f acc xs)
  foldl _ acc Nil       = acc
  foldl f acc (x : xs)  = foldl f (f acc x) xs
  foldMap _ Nil         = mempty
  foldMap f (x : xs)    = ??? ❶
```

❶ Recursive case.

There are two possible approaches we can take when filling in the `???` is. We could associate Right or Left:

```
foldMapL f = foldl (\acc x -> acc <> f x) mempty ❶ ❷
foldMapR f = foldr (\x acc -> f x <> acc) mempty ❶ ❸
```

❶ Written in Point-free notation where the `List` isn't explicitly defined.

❷ The accumulator is on left-hand side of `<>`, hence, Left-Associative.

❸ The accumulator is on right-hand side of `<>`, hence, Right-Associative.

In the PureScript library, there are 2 functions, `foldMapDefaultR` and `foldMapDefaultL` which effectively is the above code.

And as it turns out, the authors of the `Array Foldable` Typeclass chose to be Right-Associative, while the authors of `List` chose Left-Associative.

While this is unfortunate, it's possible to convert between a List and an Array, so we can choose which one we want. We also can explicitly call `foldMapDefaultR` on `Lists`. But these non-default solutions cannot benefit from Typeclasses.

So, let's pick Left-Associative to match the library author's choice:

```
instance foldableList :: Foldable List where
  foldr _ acc Nil       = acc
  foldr f acc (x : xs)  = f x (foldr f acc xs)
  foldl _ acc Nil       = acc
  foldl f acc (x : xs)  = foldl f (f acc x) xs
  foldMap _ Nil         = mempty
  foldMap f l           = foldl (\acc x -> acc <> f x) mempty l ❶
```

❶ We cannot write in Point-free when using Pattern Matching.

10.5. Rewriting `length` with Folds

Since `foldl` and `foldr` pass State, we can recode the Tail Recursive version of `length` using Fold. Before we do, let's revisit that version of `length`:

```
length :: ∀ a. List a -> Int
length l = go 0 l where ❶
  go :: ∀ a. Int -> List a -> Int ❶
  go acc Nil      = acc
  go acc (x : xs) = go (acc + 1) xs ❷
```

❶ The first Parameter is our State, which is the current count and it starts at `0`.

❷ Each iteration we add `1` to our count.

We have to determine 2 things to use `foldl` or `foldr`. First, what's the State and its initial value. In this case, it's a count of the number of elements in the `List` and its Initial Value is `0`.

And the second thing to determine is what operation will we do each iteration. In this case, we'll simply add `1` to our count.

The last thing to determine is if Associativity matters to us. Since our operation is `+`, which is `Commutative`, i.e. order does NOT matter, we do not care about Associativity.

So we could choose either `foldl` or `foldr` and our results would be the same:

```
(((1 + 1) + 1) + 1) = (1 + (1 + (1 + 1)))
```

But there is one more factor to consider. Here is our implementation of `Foldable` for `List`:

```
instance foldableList :: Foldable List where
  foldr _ acc Nil       = acc
  foldr f acc (x : xs)  = f x (foldr f acc xs)
  foldl _ acc Nil       = acc
  foldl f acc (x : xs)  = foldl f (f acc x) xs
  foldMap _ Nil         = mempty
  foldMap f l           = foldl (\acc x -> acc <> f x) mempty l
```

Compare the implementation of `foldr` to `foldl` and see if you find a reason to use one over the other.

If you cannot see why one would be preferred over the other, see if one is more efficient than the other.

It turns out that `foldl` is more efficient than `foldr` since `foldl` is Tail Recursive, whereas `folr` is not. So we'll choose that to rewrite `length` using `foldl`:

```
length :: ∀ a. List a -> Int
length = foldl (\count _ -> count + 1) 0 ❶
```

❶ Since we don't care what the value is from the `List`, we ignore it with `_`.

Chapter 11. Coding Folds

We're going to practice writing with Folds by rewriting a small number of Recursive functions that we wrote earlier.

Return to your project that you've been coding in and create a file called `Ch11.purs` and add the following:

```
module Ch11 where

import Prelude (Unit, show, type (~>), ($)) ❶

import Data.List (List(..), (:))
import Effect (Effect)
import Effect.Console (log)

test :: Effect Unit
test = do
  log $ show $ reverse (10 : 20 : 30 : Nil) ❷ ❸
```

❶ Just a reminder that we'd normally just include all of `Prelude`, but since we'll be overriding things from `Prelude`, we'll be very specific to only include things we need.

❷ Prints `(30 : 20 : 10 : Nil)`

❸ Won't compile until we write `reverse`.

Next, change your `Main` module to import `Ch11` and call its `test` function:

```
module Main where

import Prelude
import Effect (Effect)

import Ch11 as Ch11

main :: Effect Unit
main = Ch11.test
```

11.1. Writing `reverse`

Here's the code we originally wrote for `reverse`:

```
reverse :: List ~> List
reverse Nil = Nil
reverse ol = go Nil ol where
  go rl Nil      = rl
  go rl (x : xs) = go (x : rl) xs
```

Go ahead and rewrite this by using either `foldl` or `foldr`, which ever makes the most sense. Then continue reading.

11.2. Code for `reverse`

First thing to decide before we begin coding is what is the State for `reverse`. Looking at our Tail Recursive version, we see that we're passing `rl`, i.e the reversed `List`, to our `go` function. We can also see that the Initial State is `Nil`, i.e. it's the value we initially pass to `go`.

We should use `foldl` since it's Tail Recursive and therefore more efficient, not to mention that we're rewriting a Tail Recursive function and we'd like to keep it that way:

```
import Data.List (List(..), (:), foldl) ❶

reverse :: List ~> List
reverse = foldl (\rl x -> x : rl) Nil ❷ ❸
```

❶ We could've imported `foldl` from `Data.Foldable` as well. `Data.List` re-exports `foldl` from `Data.Foldable`.

❷ The Initial State is `Nil` and at each iteration we add to the head of `rl` and return that as our modified State.

❸ This is written in Point-free form where the `List` is unnamed.

11.3. Writing `max`

Let's take a side tour and code up the `max` function. It takes 2 values and returns the maximum of the two.

Add the following to your test code:

```
log $ show $ max (-1) 99 ❶
log $ show $ max "aa" "z" ❷
```

❶ Prints `99`.

❷ Prints `"z"`.

Also, add discard and negate to your Prelude import list. We need discard since we're adding multiple lines to our test code using the do syntax (more on this later, I promise) and we need negate for our -1.

Remember, we normally do not need to worry about these imports when writing normal code but since we're recoding functions from Prelude, we're being very careful about our imports.

Another possible way we could handle imports from Prelude is to use hiding to EXCLUDE things that we're overriding, e.g.:

```
import Prelude hiding (max)
```

But if we did that, you'd learn a lot less about the different functions and where they come from. Keep in mind, you'll simply import Prelude with NO import list in your production code.

Go ahead and code up max before moving on the Hint section.

11.4. Hint for max

You probably got stumped by the test code. You have to write a function that will work with Ints and Strings. But how?

I'm going to show you a little trick. Write the function WITHOUT the Type Signature. Then once your function compiles, you'll get a compiler warning that will suggest what Type Signature you should use. That suggested Type Signature will tell you how.

Give it another try and see what you get.

11.5. Code for max

We're going to follow the approach suggested in the Hint section.

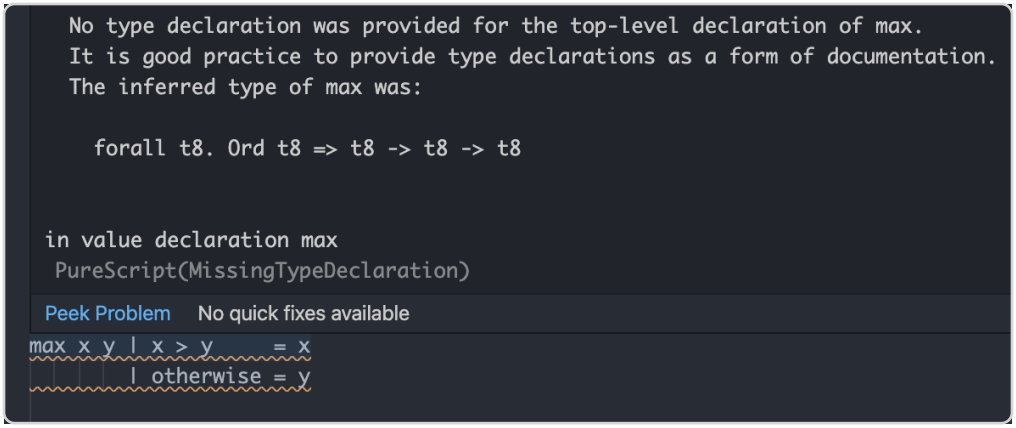
We can code this using an if expression or with Guards. Let's do it with Guards:

```
import Prelude (Unit, show, negate, discard, otherwise, type (~>), ($),
                (>)) ❶

max x y | x > y     = x   -- COMPILER WARNING!!
        | otherwise = y
```

❶ Had to add > and otherwise. Remember, otherwise is just defined to be true.

Now we get the following compiler warning:



We can see that we need constrain `a` to have an `Ord` instance. That's because the operator `>`, which is an Operator Alias for `greaterThan`, calls `compare` from the `Ord` Typeclass.

This is how `max` can work for `Ints` and `Strings`:

```
import Prelude (class Ord, Unit, show, negate, discard, otherwise,
                type (~>), ($), (>)) ❶

max :: ∀ a. Ord a => a -> a -> a ❷
max x y | x > y     = x
        | otherwise = y
```

❶ We had to import the `Ord` class. Note the `class` keyword.

❷ The Type Signature as suggested by the compiler warning message.

11.6. Writing `findMax`

Armed with our `max` function, we can now write a function to find the maximum value of a `List`. We've not written this before, so we'll first write it as regular Recursive function.

Let's make the first Parameter to `findMax` a default value so we don't have to return a `Maybe`.

Add the following line to your test code:

```
log $ show $ findMax 0 (37 : 311 : -1 : 2 : 84 : Nil) ❶
log $ show $ findMax "" ("a" : "bbb" : "c" : Nil) ❷
```

❶ Prints `311`.

❷ Prints `"c"`.

Try to code `findMax` before moving forward.

11.7. Code for findMax

Since we wrote max to help us write findMax, we can look to max's Type Signature for clues on what Constraints we'll need. We can see that if max requires values to have an Ord instance, then we'll need the same Constraint, i.e. if we want to call max with elements from our List:

```
findMax :: ∀ a. Ord a => a -> List a -> a ❶
```

❶ The first Parameter is the default value, just in case our List is empty.

Next, we write the base-case:

```
findMax :: ∀ a. Ord a => a -> List a -> a
findMax mx Nil = mx
```

We return either the default value if the initial List is empty or whatever the current maximum value is if we've just exhausted our List.

Finally, we code the recursive case:

```
findMax :: ∀ a. Ord a => a -> List a -> a
findMax mx Nil = mx
findMax mx (x : xs) = findMax (max x mx) xs ❶
```

❶ Tail Recursive call.

We recurse with the tail of the List and the max of our current maximum (which is initially our default value) and the head of our List.

11.8. Improving findMax

Let's examine our test code for findMax:

```
log $ show $ findMax 0 (37 : 311 : -1 : 2 : 84 : Nil)
```

What would happen if we didn't use 0 as our default value? What if we change the default value to 999 and run our tests. What did you think you'd get? Go ahead and try it.

If you changed your test and ran it, you'd see that you get 999 since it's larger than anything in the List. Not ideal.

We cannot trust our caller to pick the right default value. This function's design is flawed. How can we fix

it?

We could require our `a` to be a `Monoid` then we wouldn't need a default value. We could use `mempty`. But then `mempty` would have to be smaller than any other other value in our `List` and we cannot guarantee that.

Also, `Ints` aren't Monoids since we cannot definitively determine the Binary Operator. It could be Addition or Multiplication.

This is a can of worms. At first, not having to deal with `Maybe` seemed like a good idea, but it's actually caused more problems than it solved. So, let's redesign our function to make it return a `Maybe`:

```
import Data.Maybe (Maybe(..)) ❶

findMax :: ∀ a. Ord a => List a -> Maybe a
```

❶ We need to import `Maybe` with its Data Constructors. We will see the need for `Just` and `Nothing` as we continue.

Go ahead and write the rest of the code before moving on. Don't forget to remove the default value from your test code.

11.9. Improved `findMax`

So far we have:

```
findMax :: ∀ a. Ord a => List a -> Maybe a
```

Now for the base-case:

```
findMax :: ∀ a. Ord a => List a -> Maybe a
findMax Nil = Nothing ❶
```

❶ The failure case since we cannot find the max of an empty `List`.

And finally, we code the recursive case:

```
findMax :: ∀ a. Ord a => List a -> Maybe a
findMax Nil = Nothing
findMax (x : xs) = ???
```

We know we don't have an empty `List` here and we still need to keep track of our maximum value as we

traverse the `List`. But we cannot recurse with `findMax` because we removed the maximum value Parameter.

Therefore, we'll have to introduce a `go` function that has an extra variable to keep track of the current maximum.

```
findMax :: ∀ a. Ord a => List a -> Maybe a
findMax Nil = Nothing
findMax l@(x : _) = Just $ go x l where ❶ ❷ ❸
  ???
```

❶ We don't use the tail, so we renamed `xs` to `_`.

❷ We use Pattern Matching to remove the head of the `List`. The head will be our starting maximum value.

❸ We're using the `@` syntax to assign a name to the FULL `List`. That name is `l`.

Now, we need to handle the base-case for `go`:

```
findMax :: ∀ a. Ord a => List a -> Maybe a
findMax Nil = Nothing ❶
findMax l@(x : _) = Just $ go x l where
  go mx Nil = mx ❷
```

❶ This is the base-case for `findMax` when we are ***initially*** given an empty `List`.

❷ This is the base-case for `go` when we've exhausted our `List`. We return the current maximum value, `mx`, once we've gotten to the end of the `List`.

And finally, we code the recursive case for `go`:

```
findMax :: ∀ a. Ord a => List a -> Maybe a
findMax Nil = Nothing
findMax l@(first : _) = Just $ go first l where ❶
  go mx Nil       = mx
  go mx (x : xs)  = go (max x mx) xs ❷
```

❶ Renamed `x` to `first` so it wouldn't shadow the `x` on the last line.

❷ Tail Recursive.

And we need to change our test code:

```
  log $ show $ findMax (37 : 311 : -1 : 2 : 84 : Nil) ❶
  log $ show $ findMax ("a" : "bbb" : "c" : Nil) ❶
```

❶ Removed the first Parameter.

We now are ready to convert our final version of `findMax` to use fold. Give it a try before looking ahead at the solution.

11.10. Code for `findMax` using Fold

First, we know that we want to use `foldl` to keep `findMax` Tail Recursive.

We also know that our Initial State is `first`, our initial maximum and that we will modify our State with our `max` function. This means that the function that `foldl` calls will return the maximum between the `List` element and the current maximum (our State):

```
findMax :: ∀ a. Ord a => List a -> Maybe a
findMax Nil = Nothing
findMax l@(first : _) = Just $ foldl (\curMax e -> max curMax e) first l
```

But the function `foldl` calls, `(\curMax e -> max curMax e)`, can be ***ETA Reduced***, i.e. cancel on the right:

```
findMax :: ∀ a. Ord a => List a -> Maybe a
findMax Nil = Nothing
findMax l@(first : _) = Just $ foldl (\curMax -> max curMax) first l ❶
```

❶ ETA Reduced by canceling `e`.

And we can do it again:

```
findMax :: ∀ a. Ord a => List a -> Maybe a
findMax Nil = Nothing
findMax l@(first : _) = Just $ foldl max first l ❶
```

❶ ETA Reduced by canceling `curMax`.

Now we have a much easier to read function that leverages code that's already been written and debugged, `foldl`.

Also, it handles the error case far better than our initial design with the slight downside of forcing callers to deal with the `Maybe`. We could fix this by forcing the `List` to be `NonEmpty` as we've done before.

Let's do that. We're going to call our function `findMaxNE` and add the following lines to the test code:

```
log $ show $ findMaxNE (NonEmptyList $ 37 :| (311 : -1 : 2 : 84 : Nil)) ❶
log $ show $ findMaxNE (NonEmptyList $ "a" :| ("bbb" : "c" : Nil)) ❷
```

❶ Prints `311`.

❷ Prints `"c"`.

Notice, this is nearly identical to the `findMax` test code, with the exception of creating a `NonEmptyList`. We use the constructor for the `NonEmptyList` which takes a `NonEmpty List`. We use the `:|` operator, which is an Operator Alias for `NonEmpty` Data Constructor.

Remember that `NonEmptyList` requires at least 1 element and a `List`. That `List` can be empty since we always have a head value. In our test code, that would be `37` in the `NonEmptyList Int` case and `"a"` for `NonEmptyList String`.

And the following import:

```
import Data.NonEmpty (NonEmpty(..), (:|))
```

Look back in the book for `NonEmptyList` if you need a reminder on what we did last time. Take a stab at writing `findMaxNE` before reading on.

11.11. Code for `findMax` using `NonEmptyList`

First we write the Type Signature:

```
import Data.List.Types (NonEmptyList(..)) ❶

findMaxNE :: ∀ a. Ord a => NonEmptyList a -> a ❷
```

❶ We need to import `NonEmptyList`. Don't forget to keep your imports alphabetized. This would come right after the `Data.List` import.

❷ We just changed `List` to `NonEmptyList` in our old Type Signature and removed the `Maybe` since we can no longer fail.

Next, we write the base-case:

```
findMaxNE :: ∀ a. Ord a => NonEmptyList a -> a
findMaxNE ???
```

Wait, there is NO base-case for `findMaxNE`. We will always have a `NonEmptyList`.

That means we can just write the code using foldl:

```
findMaxNE :: ∀ a. Ord a => NonEmptyList a -> a
findMaxNE ??? = foldl max ???
```

We just have to figure out how to breakdown a NonEmptyList to get at it's components. Since we know that NonEmptyList is a newtype, let's start by unwrapping it:

```
findMaxNE :: ∀ a. Ord a => NonEmptyList a -> a
findMaxNE (NonEmptyList ???) = foldl max ???
```

Since, NonEmptyList is a newtype, we need to look up what it wraps. Pursuit is our friend here and after a brief consultation, we remember that NonEmptyList wraps NonEmpty List a. Now we can Pattern Match that part:

```
findMaxNE :: ∀ a. Ord a => NonEmptyList a -> a
findMaxNE (NonEmptyList (NonEmpty ???)) = foldl max ???
```

Now we need to look at the NonEmpty Data Constructor. Once again Pursuit to the rescue:

```
newtype NonEmpty f a = NonEmpty a (f a)
```

Remember that f is the collection (or Functor... more on this later). So that means we can pattern match with the head of a NonEmptyList, which is a (we'll call it first), and the tail of a NonEmptyList, which is f a or in our case List a (we'll call it l):

```
findMaxNE :: ∀ a. Ord a => NonEmptyList a -> a
findMaxNE (NonEmptyList (NonEmpty first l)) = foldl max ???
```

And finally, we can use first as our Initial State, essentially our starting maximum. And we'll use the tail, l, as the List for foldl, yielding:

```
findMaxNE :: ∀ a. Ord a => NonEmptyList a -> a
findMaxNE (NonEmptyList (NonEmpty first l)) = foldl max first l
```

Run your test code to make sure we did this right.

Did you get an error:

```
[error] Some of your project files import modules from packages that are not
in the direct dependencies of your project.
To fix this error add the following packages to the list of dependencies in
your config:
- nonempty
```

Add this package to `spago.dhall` and run it again.

11.12. Postmortem of `findMax` and `findMaxNE`

`findMax` was complicated by the fact that it could fail. The complexity became apparent when we had to code for both the failure and success cases.

By requiring that the `List` be non-empty, we got a simpler interface, i.e. `findMaxNE` does NOT return a `Maybe`. But our Pattern Match was a nightmare.

So, is there a way to have our cake and eat it too?

You know there is.

11.13. Writing `findMaxNE` using `foldl1`

When you know that your collection will NEVER be empty, you also know that you'll always have at least 1 element to pull from that collection. In our case, we have a `NonEmptyList` and we used Pattern Matching to pull out `first`:

```
findMaxNE :: ∀ a. Ord a => NonEmptyList a -> a
findMaxNE (NonEmptyList (NonEmpty first l)) = foldl max first l
```

We then passed `first` to `foldl`. But this work is going to be the same for any `NonEmpty`. And there's a helper function called `foldl1` that does some of the heavy lifting:

```
foldl1 :: ∀ f a. Foldable f => (a -> a -> a) -> NonEmpty f a -> a
```

Notice that there no Initial State. `foldl1` gets its Initial State from the head of the `NonEmpty` collection.

Also, notice how the function we pass to `foldl1` doesn't have a `b` in it like `foldl`. That's because our State is of the SAME Type as the elements of our non-empty collection. This isn't always what we want, but in the case of `findMaxNE`, it's exactly what we need.

By the way, once `foldl1` has its Initial State, it just calls our old friend `foldl` just like we did.

Rewrite `findMaxNE` to use `foldl1` before reading on.

Keep in mind that you only have to Pattern Match `NonEmptyList` to get at the `NonEmpty` and then you can call `foldl1` with it.

11.14. Code for `findMaxNE` using `foldl1`

The code is pretty straightforward:

```
import Data.NonEmpty ((:|)) ❶
import Data.Semigroup.Foldable (foldl1) ❷

findMaxNE :: ∀ a. Ord a => NonEmptyList a -> a
findMaxNE (NonEmptyList ne) = foldl1 max ne
```

❶ We no longer need to import `NonEmpty(..)` so it's been removed.

❷ Import `foldl`.

This version is the most terse of all versions. It also doesn't burden the caller with having to check for `Nothing`.

Run it to make sure it works. When you do you should get the following error:

```
[error] Some of your project files import modules from packages that are not
in the direct dependencies of your project.
To fix this error add the following packages to the list of dependencies in
your config:
- foldable-traversable
```

Add `foldable-traversable` package to `spago.dhall` and try again. It should run now.

11.15. Writing `foldl1`

`foldl1` sure made our work easier and it could seem a lot like magic, that is, right up to the point that you write it yourself. So let's do that.

Go ahead and remove `foldl1` from you imports and add:

```
import Data.Foldable (class Foldable)
```

Also, add back the `NonEmpty` import:

```
import Data.NonEmpty (NonEmpty, (:|))
```

Now give it a try, here's the Type Signature again:

```
foldl1 :: ∀ f a. Foldable f => (a -> a -> a) -> NonEmpty f a -> a
```

Please don't move on before you try to code `foldl1`.

11.16. Hint for `foldl1`

We want to leverage `foldl` to do the heavy lifting. And the only part `foldl1` is missing compared with `foldl` is the Initial State. That value we can get from our `NonEmpty`. Since it's non-empty, we can always get the head Element.

Remember, we're not just working with `NonEmptyLists` here, but instead ANY Type that's `Foldable`.

The hardest part of coding this function is the Pattern Matching part. So, make that easier by looking to `Pursuit` to see how `NonEmpty` is constructed. Remember, we Pattern Match against the Data Constructor NOT the Type Constructor. The Data Constructor is on the right-hand side of the equal sign in a `data` definition.

Go ahead and give another try before moving on.

11.17. Code for `foldl1`

There are 2 ways to write this. First, with the Data Constructor:

```
import Data.NonEmpty (NonEmpty(..), (:|)) ❶

foldl1 :: ∀ f a. Foldable f => (a -> a -> a) -> NonEmpty f a -> a
foldl1 f (NonEmpty x xs) = foldl f x xs ❷ ❸
```

❶ Had to put back the `(..)` to import the `NonEmpty` Data Constructors.

❷ Match against the `NonEmpty` Data Constructor.

❸ The head, `x`, is the Initial State we give to `foldl` and we use the tail as `foldl's Foldable` collection. This could be a `List` or an `Array` or whatever has a `Foldable` Instance.

The second way to write it, my favorite, is to use the Operator:

```
import Data.NonEmpty (NonEmpty, (:|)) ❶ ❷

foldl1 :: ∀ f a. Foldable f => (a -> a -> a) -> NonEmpty f a -> a
foldl1 f (x :| xs) = foldl f x xs
```

❶ Notice how we didn't have to change our imports since we already import our Operator.

❷ Removed `(..)` since we're not using Pattern Matching in this version.

The point of this exercise was do demystify `foldl1`. I hate when things appear to be magic. Nothing is magic once you understand it. And coding the function yourself is the best way to understand it.

11.18. Writing `sum`

Let's take a break and write an easy function, `sum` that takes a `List` of `Ints` and sums them up. First we'll write it as a recursive function and then we'll write it using `foldl`.

But why not `foldr`? That's because `foldl` is Tail Recursive and therefore makes more efficient use of the Stack AND Addition is Associative which means that it produces the same results for Right or Left-Associative computations:

```
1 + (2 + 3) == (1 + 2) + 3
```

Here's the test code that we'll use:

```
log $ show $ sum (1 : 2 : 3 : Nil) ❶
```

❶ Prints 6.

So first write `sum` as a regular old recursive function. Then continue on.

11.19. Code for `sum` as a Recursive Function

First we write the Type Signature:

```
sum :: List Int -> Int
```

Next, we write the base-case:

```
sum :: List Int -> Int
sum Nil = 0
```

And finally, we write the recursive case:

```
sum :: List Int -> Int
sum Nil = 0 ❶
sum (x : xs) = x + sum xs ❷ ❸
```

❶ Note that `0` is the Additive Identity.

❷ We have to add `(+)` to our `Prelude` imports.

❸ This is NOT Tail Recursive since we have to return from the `sum` call to add `x`.

Before moving on, rewrite `sum` to be Tail Recursive using a `go` function where we pass State as an extra parameter to `go`.

11.20. Code for `sum` as Tail Recursive

As usual, we start with the Type Signature:

```
sum :: List Int -> Int
```

Next we write the base-case:

```
sum :: List Int -> Int
sum Nil = 0 ❶
```

❶ Summing the values of an empty list returns 0, the Additive Identity for `Int`s.

Next we write the recursive case:

```
sum :: List Int -> Int
sum Nil = 0
sum l = go 0 l where ❶
  ???
```

❶ We set our Initial State to `0` because it's the Additive Identity for `Int`s, which means that we can add it to anything without affecting the result.

Now, we can write the base-case, but for the `go` function:

```
sum :: List Int -> Int
sum Nil = 0 ❶
sum l = go 0 l where
  go acc Nil = acc ❷
```

❶ sum's base-case.

❷ Once we exhaust our List, we return the accumulated value, acc.

At this point, you may have noticed that if go was ever INITIALLY passed an empty List, acc would be 0, which it would return. This is exactly the same behavior as sum's base-case.

That tells us that sum's base-case is redundant. Let's remove it now:

```
sum :: List Int -> Int
sum = go 0 where ❶
  go acc Nil = acc
```

❶ We can now write this Point-free where the List is unspecified.

And now we can finally code the recursive case for the go function:

```
sum :: List Int -> Int
sum = go 0 where
  go acc Nil      = acc
  go acc (x : xs) = go (x + acc) xs ❶
```

❶ We add the head of our List, x, to our accumulator, acc and recurse with the rest of the List, xs.

Now it's Tail Recursive.

But this was a lot of work to code and debug. This work is worth it when we're learning but when we're writing Production Code we want to be more efficient.

So let's rewrite this using foldl, the Tail Recursive fold. Do so then move on.

11.21. Code for sum using foldl

Looking back at our Tail Recursive version, we can see that our Initial State was 0, the Additive Identity, and our function at every iteration was simple Addition.

That means we can write the following:

```
sum :: List Int -> Int
sum = foldl (\acc x -> acc + x) 0
```

When we run our tests using this function, we see that it prints out 6 meaning that it works. But just because something works doesn't mean we're done.

We should really look to see if we can improve on what we've done. Can you see any room for improvement?

Looking at the function passed to foldl, we can see that we've just rewritten the addition function. Take a look and see if you can it now.

The best way to see this is to rewrite that function using the Operator in Prefixed Notation:

```
sum :: List Int -> Int
sum = foldl (\acc x -> (+) acc x) 0 ❶
```

❶ The Binary Operator is in Prefixed Notation.

Now, we can ETA reduce first the x and then acc leaving us with (+).

So, let's replace it with the addition function:

```
sum :: List Int -> Int
sum = foldl (+) 0
```

That looks great. It's leveraging foldl and + making this very terse and easy to read.

But can we do better? This only works with Lists of Ints. Can we make it work with Numbers, which is PureScript's floating point numbers?

Try to change this version of our function to work with Ints and Numbers before reading on to the Hint section.

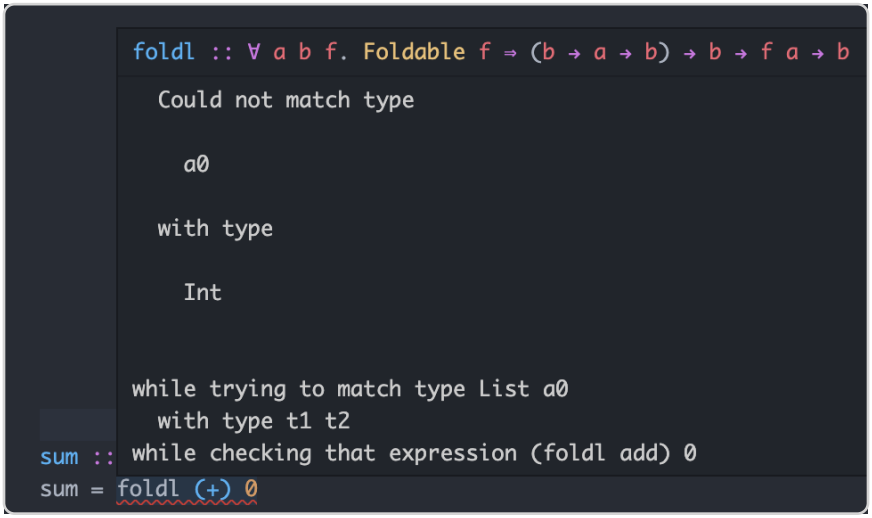
11.22. Hint for sum for Ints and Numbers

We know from Grade School that we can add Integers and Decimal numbers using the + operator. But how do we tell the compiler that. We could write:

```
sum :: ∀ a. List a -> a ❶
sum = foldl (+) 0 -- COMPILER ERROR!!
```

❶ We've changed `Int` to `a`.

This gives us the following compiler error:



The salient portion of this error is:

> Could not match type `a0` with type `Int`

That's because we're using an `Int` as our Initial State. We could change it to `0.0`:

```
sum :: ∀ a. List a -> a
sum = foldl (+) 0.0 -- COMPILER ERROR!! (AGAIN)
```

But now the error is:

> Could not match type `a0` with type `Number`

We need an Abstraction that represents all numbers, i.e. `Ints` and `Numbers`, and the Addition Operator. Can you think of something that does this? Maybe something from Abstract Algebra.

If you cannot, run the REPL and check the Type for the addition operator:

```
npx spago repl
> import Prelude ❶
> :t (+) ❷
??? ❸
```

❶ We need this to import `(+)`.

❷ Don't forget to type Control-D to exit

❸ Run the REPL yourself to see the answer.

Armed with this information, see if you can modify `sum` to work with both `Ints` and `Numbers`. And to help test that change, add the following to the test code:

```
log $ show $ sum (1.0 : 2.0 : 3.0 : Nil) ❶
```

❶ Prints `6.0`.

Look back to the Abstract Algebra section and the Abstract Algebra code you wrote to help you write this version of `sum` before moving forward.

11.23. Code for `sum` for `Ints` and `Numbers`

If you looked at the Type of `(+)` in the REPL, you got:

```
> import Prelude
> :t (+)
forall a. Semiring a => a -> a -> a ❶
> ^D
See ya!
()
```

❶ N.B. `Semiring`.

That should remind us that we need our `a` to have a `Semiring` instance. If that's the case, then we can just add that Constraint to our function to hopefully remove our compiler error:

```
import Data.Semiring (class Semiring) ❶

sum :: ∀ a. Semiring a => List a -> a
sum = foldl (+) 0.0 -- COMPILER ERROR!! (STILL)
```

❶ Import `Semiring` Type class, hence the `class` keyword.

We get the same compiler error as before:

```
Could not match type a0 with type Number
```

But why? See if you can catch the mistake.

The problem is that we're still using a hardcoded Initial State, which is the Additive Identity for `Numbers`. And we know that if we change it to `0`, the Additive Identity for `Ints`, we'll still get a similar error.

What we need is the Additive Identity for any `Semiring`. Luckily, the `Semiring` Typeclass has the very thing we need. It's called `zero`.

The compiler will pick the appropriate `zero` for `Ints` or `Numbers` or any Type that has a `Semiring` instance.

Let's review the `Semiring` definition:

```
class Semiring a where
  add :: a -> a -> a ❶
  zero :: a ❷
  mul :: a -> a -> a
  one :: a

infixl 6 add as + ❸
infixl 7 mul as *
```

❶ Here's the `add` function.

❷ This is the Additive Identity.

❸ The Operator Alias for `add` is `+`.

Armed with this refresher, we can fix our compiler error by replacing the `0.0` with `zero`:

```
import Data.Semiring (class Semiring, zero) ❶

sum :: ∀ a. Semiring a => List a -> a
sum = foldl (+) zero
```

❶ Added `zero` to the import list.

The compiler will chose the appropriate `zero` for the `Semiring`. When the `Semiring` is `Int`, it'll use `0` and when it's `Number`, it'll use `0.0`.

And this will work for ANY `Semiring`.

The compiler will also choose the appropriate `add` for the `Semiring`. Remember, each `Semiring` has its own implementation for `add`.

Now we've greatly improved our `sum` function because it isn't limited to just `Ints`.

But... We're not finished. We can do better. Can you see what the next improvement should be?

I'll give you a hint, we replaced Int with Semiring to make it work for ANY Element. How about now? Can you see any other hardcoded Type?

That's right. The collection, i.e. the List, can also be Abstracted. What constraint can we put on the collection to make more general?

This Chapter is about Foldable, and we've been restricting our code to List, but there are many other Foldable Data Structures, e.g. Array or trees, etc.

Add the following lines to your test code in anticipation of being able to sum over Arrays:

```
log $ show $ sum [1, 2, 3] ❶
log $ show $ sum [1.0, 2.0, 3.0] ❷
```

❶ Prints 6.

❷ Prints 6.0.

So try to modify our Function to work with ANY Foldable Type. Then see how you did by advancing to the next section.

11.24. Code for sum using Foldable

```
sum :: ∀ f a. Foldable f => Semiring a => f a -> a ❶
sum = foldl (+) zero
```

❶ We added the Foldable f Constraint where f is our foldable Type, e.g. List, Array, etc.

This final version of sum is as flexible as we can make it. We've abstracted both the Data Structure AND the Elements.

We could create a Tree and make it Foldable and then if the Tree holds a Semiring, we could call sum on it and it would just work with no additional effort on our part. This is the real power of Typeclasses.

And if we can, maybe we should.

11.25. Using sum with Tree

Let's make a simple Binary Tree Type:

```
data Tree a = Leaf a | Node (Tree a) (Tree a)
```

A Tree can be just a Leaf OR a Node with a Left Tree and a Right Tree, each of which can be a Leaf or another Node and so on.

This is a Recursive Definition because the `Tree` Type is on both the left and right-hand side of the equal sign.

`Leaf` is the base-case. It's what terminates the Recursion.

`Node` is the recursive case.

These mirror the way base-cases and recursive cases work in Recursive Functions. Our base-case, terminates the Recursion.

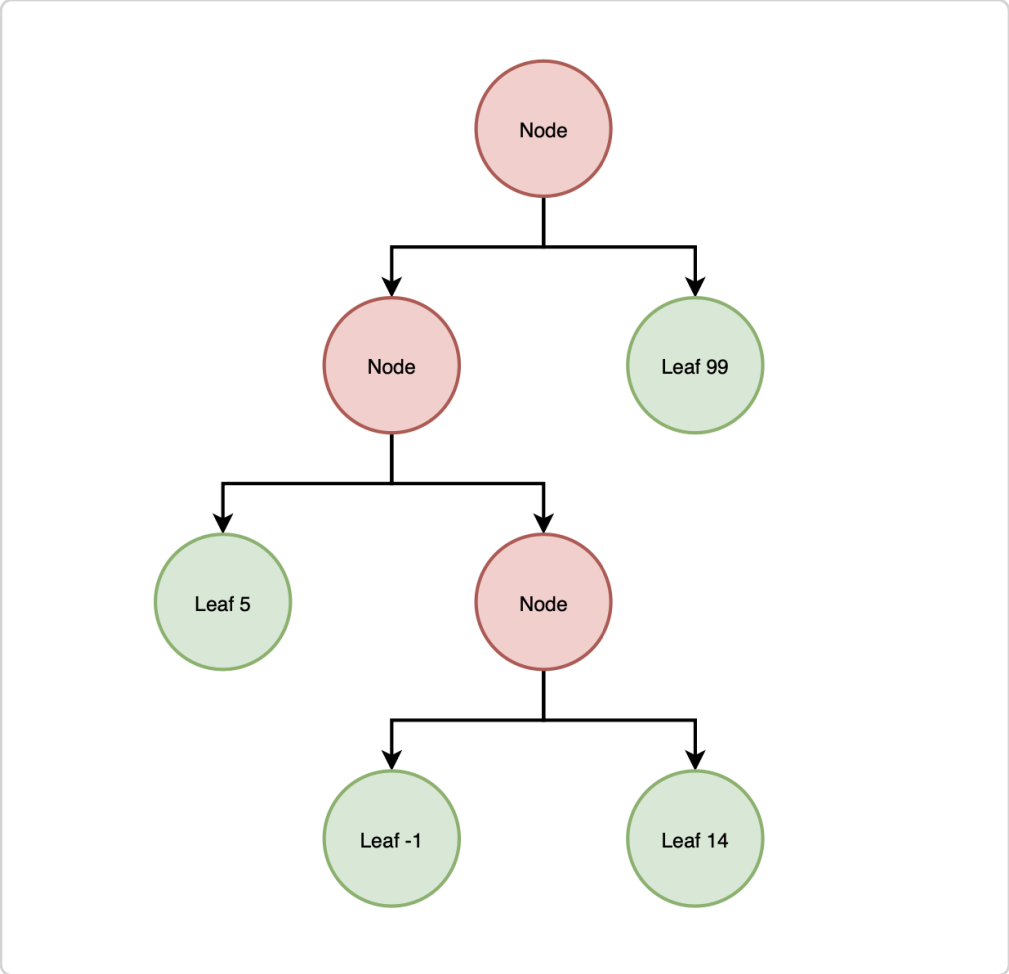
Now, all we have to do is make our Tree `Foldable`. Easier said than done though.

There are a number of things we must first decide before we make it `Foldable`. Are we going to favor, the left or right side when we encounter a `Node`? And, are we going to traverse the `Tree`, Depth-first or Breath-first.

We didn't have to make these decisions for `List` since there's only one way to traverse a `List`, i.e. from head to tail. But a `Tree` is a structure that can be traversed a few different ways.

To make our code easier, let's make an executive decision and choose ***Depth-first***.

Often times, it's worth stopping and creating a simple example of your Data Structure to help you develop your code. Here's a simple `Tree`:



Let's see how our `sum` function is affected by favoring the ***Left***`:

```
0 + 5 + (-1) + 14 + 99 = 117
```

If now if we favor the ***Right*** we'll get:

```
0 + 99 + 14 + (-1) + 5 = 117
```

Notice the result is the same. So using `sum` to help us decide which side to favor doesn't help us much.

We also need to remember that we're writing the `Foldable` instance for all functions, not just `sum`. So, we need to consider the best way to traverse it so it works well for all `Foldable` operations, e.g. Subtraction. With Subtraction, the results will NOT be the same.

But, we didn't specify anything special about this Binary Tree. It's just a Binary Tree. It's not balanced or sorted. So we cannot make any decisions based on that information.

So, let's make another executive decision and just choose ***Left***.

Now that we've decided on how to traverse, we're ready to code the `Foldable` instance for `Tree`. Consult

with Pursuit or look back in the book to get the Foldable class definition.

Give it try before checking out the Hint section.

11.26. Hint for Foldable for Tree

If you started to write foldr and then foldl (or if you imagined the work involved), you may have come to the conclusion that these functions both need to have the SAME traversal logic.

Here's an idea. First, write a function called toList that traverses the Tree Depth-first favoring the Left side.

Then take that list and call the appropriate Fold on it.

Use the following test code to test toList:

```
log $ show $ toList
  (Node (Node (Leaf 5) (Node (Leaf (-1)) (Leaf 14))) (Leaf 99)) ❶ ❷
```

❶ This Tree is the same as the example diagram above.

❷ Prints (5 : -1 : 14 : 99 : Nil).

11.27. Code for toList Instance for Tree

First, we write the Type Signature:

```
toList :: ∀ a. Tree a -> List a
```

Notice that I wasn't able to write Tree ~> List since our Tree Type isn't a Functor. Not yet, but soon. So for now, we use the more verbose Type Signature.

Next, we write the base-case:

```
import Data.List (List(..), (:), foldl, singleton) ❶

toList :: ∀ a. Tree a -> List a
toList (Leaf x) = singleton x ❷
???
```

❶ Added singleton to import list.

❷ A Tree with just one Leaf is just a Singleton List with the same value as the Leaf.

And finally, we write the recursive case:

```
toList :: ∀ a. Tree a -> List a
toList (Leaf x) = singleton x
toList (Node lt rt) = toList lt <> toList rt ❶ ❷
```

❶ (<>) needs to be added to Prelude imports.

❷ We favor the Left side. If we swap the lt and rt on the left side of the equal sign that would favor the Right side.

Now, we're ready to write the Foldable instance for Tree. Give it a try and make sure to leverage toList.

Use the following test code:

```
log $ show $ sum
  (Node (Node (Leaf 5) (Node (Leaf (-1)) (Leaf 14))) (Leaf 99)) ❶
```

❶ Prints 117.

Try to not skip ahead until you've tried coding it yourself.

11.28. Code for Foldable for Tree

We can simply convert our Tree to a List and then call List's Fold functions for all Methods:

```
import Data.Foldable (class Foldable, foldl, foldr, foldMap) ❶
import Data.List (List(..), (:), singleton) ❷

instance foldableTree :: Foldable Tree where ❸
  foldr f acc = foldr f acc <<< toList ❹ ❺
  foldl f acc = foldl f acc <<< toList
  foldMap f = foldMap f <<< toList
```

❶ We need to import foldl, foldr and foldMap, to write these functions.

❷ Removed foldl from List since it simply re-exports foldl from Data.Foldable and now we're importing it directly ourselves.

❸ Remember, that Foldable takes a Type Constructor that is of Kind Type -> Type. This is why we don't specify the a parameter for Tree.

❹ All 3 functions are Point-free. The Tree is the unnamed Parameter.

❺ You'll need to add (<<<) to your Prelude imports.

In all 3 cases, we pass all Parameters except the final one. That allows us to simply compose these functions since they're both waiting for a single parameter.

Now that our Tree is Foldable, we can call sum on it when it's a is a Semiring like our test Tree:

```
(Node (Node (Leaf 5) (Node (Leaf (-1)) (Leaf 14))) (Leaf 99))
```

11.29. Improving on Foldable for Tree

If we wanted to support favoring the Right side and/or Breath-first traversals, then we could use newtype to make new Types that have the appropriate behaviors and their own toList and Foldable instances.

We could even make our own Typeclass, called ToList to make that process easier.

We won't delve into this too deeply at this point, but for grins here's the implementation for Left and Right favoring for Depth-first ONLY, since I'm too lazy to write the Breadth-first traversal code.

Here is the full implementation for those interested:

```
data Tree a = Leaf a | Node (Tree a) (Tree a)

class ToList f where ❶
  toList :: ∀ a. f a -> List a

newtype RightFirstTree a = RightFirstTree (Tree a) ❷
newtype LeftFirstTree a = LeftFirstTree (Tree a)

instance toListRightFirstTree :: ToList RightFirstTree where
  toList (RightFirstTree (Leaf x)) = singleton x
  toList (RightFirstTree (Node lt rt)) =
    toList (RightFirstTree rt) <> toList (RightFirstTree lt) ❸ ❺

instance toListLeftFirstTree :: ToList LeftFirstTree where
  toList (LeftFirstTree (Leaf x)) = singleton x
  toList (LeftFirstTree (Node lt rt)) =
    toList (LeftFirstTree lt) <> toList (LeftFirstTree rt) ❹ ❺

instance foldableRightFirstTree :: Foldable RightFirstTree where
  foldr f acc = foldr f acc <<< toList
  foldl f acc = foldl f acc <<< toList
  foldMap f = foldMap f <<< toList

instance foldableLeftFirstTree :: Foldable LeftFirstTree where
  foldr f acc = foldr f acc <<< toList
  foldl f acc = foldl f acc <<< toList
  foldMap f = foldMap f <<< toList
```

❶ The `ToList` class to distinguish the traversal over a non-List structure.

❷ We wrap `Tree` in these speciality Types that have definitive traversals.

❸ `rt` is prepended to `lt` for a Right-favoring traversal.

❹ `lt` is prepended to `rt` for a Left-favoring traversal.

❺ We have to rewrap the Left and Right `Trees` in the `newtype` before we can Recurse. We could've derived a `Newtype` instance and used `wrap` instead.

Notice that we no longer need a `Foldable` instance for `Tree` or our standalone `toList` helper function. Also, our test code should change slightly:

```
log $ show $ toList $ LeftFirstTree
  (Node (Node (Leaf 5) (Node (Leaf (-1)) (Leaf 14))) (Leaf 99)) ❶
log $ show $ sum $ LeftFirstTree
  (Node (Node (Leaf 5) (Node (Leaf (-1)) (Leaf 14))) (Leaf 99)) ❷
log $ show $ toList $ RightFirstTree
  (Node (Node (Leaf 5) (Node (Leaf (-1)) (Leaf 14))) (Leaf 99)) ❸
log $ show $ sum $ RightFirstTree
  (Node (Node (Leaf 5) (Node (Leaf (-1)) (Leaf 14))) (Leaf 99)) ❹
```

❶ Prints `(5 : -1 : 14 : 99 : Nil)`.

❷ Prints `117`.

❸ Prints `(99 : 14 : -1 : 5 : Nil)`.

❹ Prints `117`.

Chapter 12. Functors

We're going to finally delve into Functors a very powerful Abstraction pilfered from Category Theory in Mathematics. We won't delve too much into the Category Theory here, but will primarily concentrate on the motivations for using this Abstraction to help solve Programming problems.

But before we do, I want to lay down a bit of groundwork that will hopefully take us all the way through Applicatives and Monads.

12.1. Defining ADTs

When we define an Algebraic Data Type (ADT) such as `Maybe`, we have an idea of what that Type represents. We've been talking about `Maybe` being the Type that holds the result a computation that can potentially fail:

```
data Maybe a = Nothing | Just a
```

But at this point, all we have is our ***Intention***. We haven't specified how it's to be used. Once we start adding Typeclass Instances, our intention becomes more explicit:

```
data Maybe a = Nothing | Just a
derive instance eqMaybe :: Eq (Maybe a)
derive instance ordMaybe :: Ord (Maybe a) ❶
```

❶ This derivation along with the initial Type definition will determine how `Maybes` will sort. `Nothings` will be smaller than `Justs` since they are defined first.

Every time we implement another Typeclass instance, we are cementing the Type's behaviors, choosing one possible behavior over all others. That doesn't mean we couldn't write code to sort `Maybe` where `Just` is sorted before `Nothing` despite the `Ord` derivation. But the whole point of `Ord` is to dictate the behavior when comparing values of a Type.

We could also decide not to derive `Ord` for our Type burdening those who use it to write a lot of boilerplate code to do common functionality:

```
compareMaybe :: ∀ a. Maybe a -> Maybe a -> Boolean
```

But this would be short-sighted and difficult to maintain. It might work for small codebases, but it certainly doesn't scale or work well with multiple projects, especially, if `compareMaybe` is different between projects.

Now, imagine we want to compute the following Mathematical equation:

```
f(x) = (83 ÷ x) × 10
```

We're going to first write a function to divide where a division by zero will result in `Nothing`:

```
import Prelude

import Data.EuclideanRing (class EuclideanRing)
import Data.Maybe (Maybe(..))

divide :: ∀ a. Eq a => EuclideanRing a => a -> a -> Maybe a ❶
divide x y
  | y == zero = Nothing ❷
  | otherwise = Just $ x / y
```

❶ Division is supported in a `EuclideanRing`.

❷ We are checking for a divide by abstract `zero`. We can use this since all `EuclideanRings` are also `Semirings`.

Now, let's compute our earlier formula:

```
f :: Number -> Maybe Number
f x = case divide 83.0 x of
  Nothing     -> Nothing ❶
  Just result -> Just $ result * 10.0 ❷
```

❶ We cannot continue our computation because the division failed.

❷ We continue our computation.

We didn't explicitly specify that failure will stop all computation in its tracks. That was an Implicit Behavior based on the structure of the Type. `Nothing` contains no values to continue with. All we can do is return `Nothing`.

This is not the case with `Either`, which makes our intension far more ambiguous:

```
data Either a b = Left a | Right b
```

What is our Intention when we define this Type?

There could be 2 different possibilities. `Either` could be a ***Choice***, i.e. a Logical Or. It could also be a ***Failure with Reason*** where `Left` is the failure case and `Right` is the success case.

With only this definition though, we have no way to tell. Let's look at both Intentions. First, ***Failure with***

Reason by repurposing our `Maybe` example:

```
import Data.Either (Either(..))

divide :: ∀ b. Eq b => EuclideanRing b => b -> b -> Either String b ❶
divide x y
  | y == zero = Left "Divide by zero" ❷
  | otherwise = Right $ x / y

f :: Number -> Either String Number ❸
f x = case divide 83.0 x of
  Left reason -> Left reason ❹
  Right result -> Right $ result * 10.0 ❺
```

❶ We changed our return Type to contain an error string if a failure occurs.

❷ Here we return a reason for the error.

❸ We changed our return Type here too.

❹ We cannot continue our computation since an error occurred. The best we can do is just return the reason for our failure.

❺ We continue our computation.

This Intention is very closely aligned with `Maybe's`, hence the use of the same example. The major difference is that we explicitly short-circuit `Left` with `Either`, whereas with `Maybe`, that choice was implicit due to the fact that `Nothing` has no value.

With the ***Choice*** Intention, we need to imagine a different use case:

```
convert :: ∀ a b c d. (a -> c) -> (b -> d) -> Either a b -> Either c d
convert f g = case _ of
  Left x -> Left $ f x
  Right y -> Right $ g y
```

Here we have 2 valid values and we can convert the `Left` from an `a` to a `b` with `f` or the `Right` from a `b` to a `d` using `g`.

We can operate on both sides since our Intention here is to use Either as a Choice between 2 different values of potentially 2 different Types.

It would be better to capture the Semantics of our Intent by creating a different Type that's Isomorphic to `Either` called `Choice`:

```
data Choice a b = PickA a | PickB b
```

Now, our name better reflects the Semantics of the Type.

Here's what our `convert` function would look like with our superiorly named Type:

```
convert :: ∀ a b c d. (a -> c) -> (b -> d) -> Choice a b -> Choice c d
convert f g = case _ of
  PickA x -> PickA $ f x
  PickB y -> PickB $ g y
```

This is far clearer than `convert` with `Either`.

12.2. Semantics

When we create a new Type, we intend it to be used in a very specific way. This is the Semantics of the Type and we have a few ways of defining those Semantics:

- Name
- Implicit Behaviors
- Explicit Behaviors

12.2.1. Name

The name we pick does a lot to communicate our Intention and provides a hint to the Semantics of the Type. When we chose to create the `Choice` Type to represent a Logical Or, the name helped immensely to make things clearer.

This is far better than reusing `Either` in limited ways to achieve the Semantics of `Choice`. Since they're Isomorphic, you can choose to use `Either` and your code will work, but you run the risk of muddying the Semantics of `Either`.

It's best to create a specialized Type for your situation. For example, if we can get a `String` or an `Int`, better to create the following Data Type than to use `Either Int String`:

```
data IntOrString = GotInt Int | GotString String
```

The approach affords us the opportunity to dictate unique Semantics that are perfectly tailored for our needs. Granted if we needed something that shared the exact same Semantics as `Either`, then it would make sense to use it instead since it saves us a ton of work defining duplicate Semantics.

12.2.2. Implicit Behavior

When we construct a Type, we are implicitly defining some of its behavior. In the case of `Maybe`, we have two possibilities, a `Just a` or `Nothing`. In the case of `Nothing`, there are no values to work with. So by definition, we must terminate (or Short-circuit) all future computations with this value.

We didn't have to make a decision to short-circuit or not. the Data Structure imposed this on us. Short-circuiting was implicit in the definition of `Nothing`.

In the case of a `Tree`:

```
data Tree a = Leaf a | Node (Tree a) (Tree a)
```

we have a recursive definition. Any Data Constructor that doesn't have `Tree a` as a Type Parameter will terminate our recursion, e.g. `Leaf a`. This behavior is implicit in the definition. With `List`, we have a very similar case:

```
data List a = Nil | Cons a (List a)
```

Here `Nil` is analogous to `Leaf a` with the notable difference of `Nil` short-circuiting any further computations since it contains no other values. In contrast, `Leaf a` contains an `a`, which means we can compute with that `a` and there is no Implicit short-circuit unless we Explicitly choose to do so.

12.2.3. Explicit Behavior

We have seen the explicit behavior of Short-circuiting in the code we wrote for `Either`:

```
f :: Number -> Either String Number
f x = case divide 83.0 x of
  Left reason -> Left reason ❶
  Right result -> Right $ result * 10.0
```

❶ Explicit Short-circuit, i.e. we're not required to short-circuit like with `Nothing`.

Unlike `Maybe`, this was a choice we made base on our Intent for `Either`. We could've decided to treat divide-by-zero as equal to `0`:

```
f :: Number -> Number ❶
f x = 10.0 * case divide 83.0 x of
  Left reason -> 0.0 ❷
  Right result -> result
```

❶ Doesn't fail anymore so we can always return `Number`.

❷ Doesn't short-circuit here but instead returns `0` as the result of the division allowing the computation to continue.

But then we wouldn't be returning an `Either`. But we still made a choice based on our Semantics of `Either`.

We made a very different choice for `Choice` (no pun intended):

```
convert :: ∀ a b c d. (a -> c) -> (b -> d) -> Choice a b -> Choice c d
convert f g = case _ of
  PickA x -> PickA $ f x ❶
  PickB y -> PickB $ g y
```

❶ No short-circuit. We apply the conversion function.

The decisions we make when implementing Typeclasses are also Explicit Behaviors, i.e. they aren't dictated by the Data Structure of the Type. We'll this soon enough when we implement Functors, in particular, for `Either`.

12.3. Contexts

A value that gets wrapped up in a `Maybe` is a very different situation than that same value wrapped in an `Either`:

```
Left "ABCDEFGHI" ❶
Just "ABCDEFGHI" ❷
```

❶ Short-circuiting since the Semantics of `Left` is an error.

❷ A successful value.

We refer to this difference as a ***Context***. We can start out with a Pure Value that's just a raw value in no context (or possibly in an Identity context) and then we can place that Pure value into a Context using the Data Constructors.

For `List`, we use `Cons`. For `Maybe` we use `Just`. For `Either` we use `Right` or `Left`.

You can think of the Context as a container of sorts, but that idea will limit your thinking as we move to more abstract Types. A Context is more nuanced encompassing not only the data it contains but also the Behaviors of the Type.

We've already seen that Contexts can have varying flow-controls. And when we get to Monads, we'll see that Contexts can have their own computations that are run along side our very own.

12.4. Higher-kinded Types

We talked about how values are grouped by Type and how Types are grouped by Kinds, e.g. `Maybe's` Kind Signature is:

```
Maybe :: Type -> Type
```

`Maybe` takes a `Type` and returns a `Type`. Unlike `Int`:

```
Int :: Type
```

`Int` is just a `Type`. And `Either` takes 2 Types:

```
Either :: Type -> Type -> Type
```

If you like, you can get the Kinds of Types in the PureScript REPL:

```
$ npx spago repl

> import Data.Either
> :k Either ❶
Type -> Type -> Type
> ^D ❷
See ya!
()
$
```

❶ `:k` is short for `:kind`.

❷ Control-D to exit the REPL.

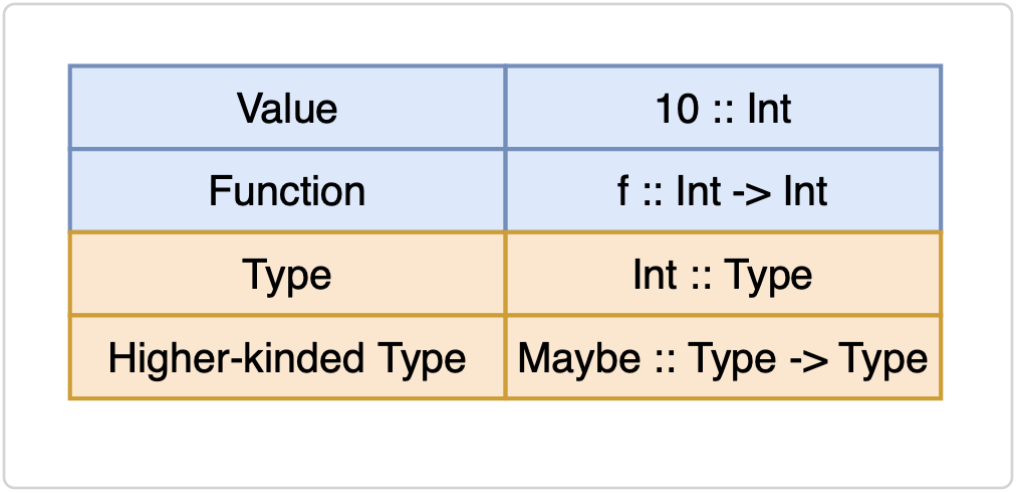
`Maybe` and `Either` look like functions and we can conceptualize them as such. They are in fact ***Type Constructors***, which is analogous to ***Data Constructors***, which are functions.



Any Type Constructor that takes 1 or more Parameters is a High-kinded Type.

`Int` is not a Higher-kinded Type like `Maybe` and `Either` since it takes NO Type Parameters.

We can draw analogies between the ***Value Space*** and ***Type Space*** in our programs:



Value	10 :: Int
Function	f :: Int -> Int
Type	Int :: Type
Higher-kinded Type	Maybe :: Type -> Type

Notice the parallelism between `Value` and `Type`. Also, the parallelism between `Function` and `Higher-kinded Type`.

12.5. Functions with Simple Types

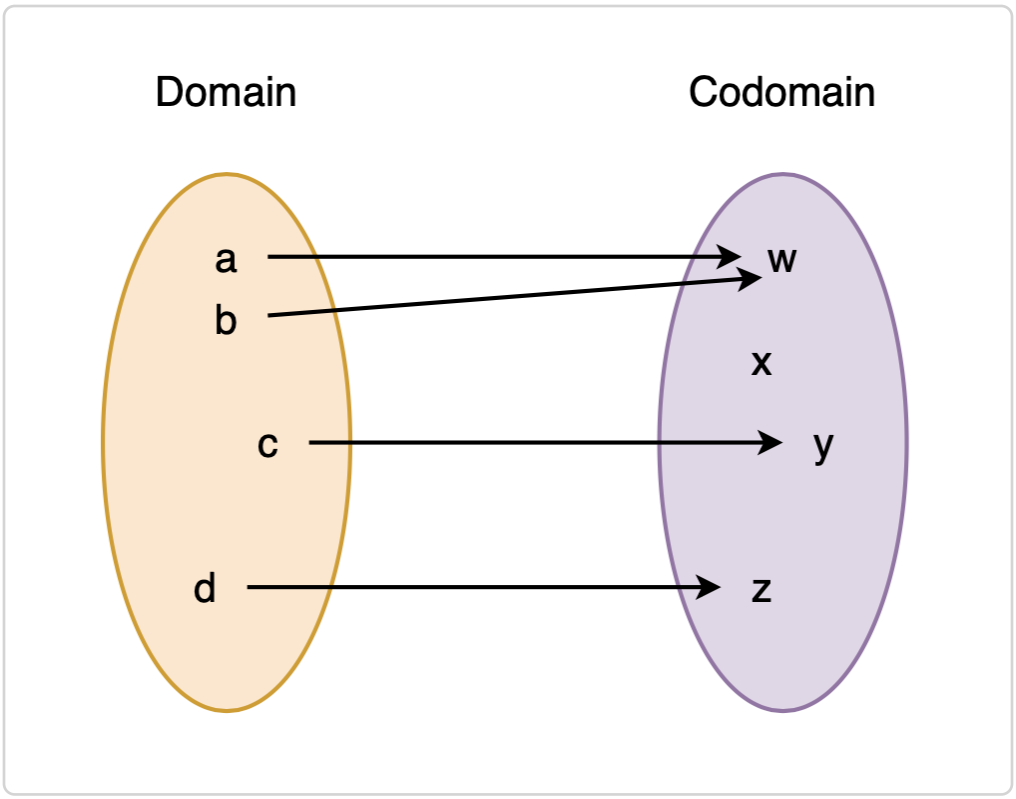
When we work with Simple Types, i.e. not Higher-kinded, we can write functions like:

```
f :: ∀ a b. a -> b
g :: ∀ b c. b -> c
h :: ∀ c d. c -> d
```

and then compose them:

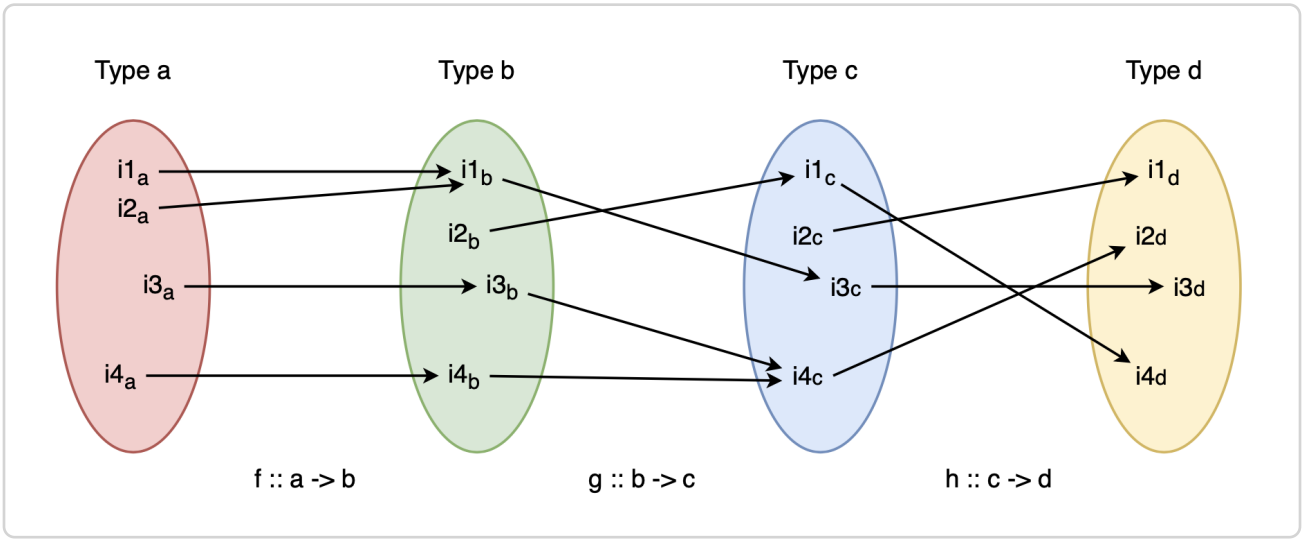
```
hgf :: ∀ a d. a -> d
hgf = h <<< g <<< f
```

In Mathematics, functions map from the Domain to the Codomain. The Domain and Codomain are Sets whose elements are values:



We say that a function maps elements in the Domain to elements in the Codomain.

But in Programming, our Domain and Codomain are Types and the elements are the Inhabitants of those Types:



Here we show the composition of `f`, `g` and `h` and their mappings from the Set of `Type a` to `Type b` and so forth. The inhabitants of `Type a` are the elements of the `Type a` Set.

And this image also holds true for repeated Function Application:

```
fgh :: ∀ a d. a -> d
fgh x = h $ g $ f x
```

Function Composition and Function Application are the bread of butter of Functional Programming. We are able to leverage smaller things to solve bigger problems, e.g. use Atoms to make Molecules.

Notice how all of our Types so far are Simple Types like `Int`, `String`, `Number`, etc. But what happens when we consider the case of failure, e.g. what if `g` could fail:

```
g :: ∀ b c. b -> Maybe c -- returns a Higher-kinded Type
```

We can compose it with `f` without issue:

```
gf :: ∀ a c. a -> Maybe c
gf = f <<< g
```

And we can do Function Application with it too:

```
fg :: ∀ a c. a -> Maybe c
fg x = g $ f x
```

But what happens when we want to Compose or Apply `h`?

The problem is `h` takes a `c` not a `Maybe c`. We could always rewrite `h` or make a specialized version of `h` that takes a `Maybe`, but that just opens up a can of worms. Our `h` function works with Simple Types but not with Higher-kinded Types.

Functors and Monads solve these problems. We will cover the solutions to this seemingly simple problem in great detail in this and the next few Chapters. And along the way, we will discover the real power of these concepts.

These concepts are too often skimmed over in other texts or introduced far too early. We've taken hundreds of pages to get here with the hope that you're comfortable with the underlying concepts before delving into these Abstractions. And, hopefully, you are now able to concentrate on just the concepts at hand.

These Abstractions will make up a large percentage of your Functional Programming experience and so they are worth taking slowly to make sure you get a good, solid understanding.

12.6. Functors to the Rescue

We introduced the notion of Contexts earlier and in our recent example, `c` is in the Context of `Maybe`:

```
g :: ∀ b c. b -> Maybe c
```

Our conundrum was how to apply `h` to the output of `g`. Let's write a function to help us apply `h` to a `Maybe c`.

First, we'll start with the Type Signature:

```
applyH :: (c -> d) -> Maybe c -> ???
```

What do we put for the final Type of our function? We may lean toward making it `d` to match what we did when `g` could not fail. But when we try that, we get stuck:

```
applyH :: (c -> d) -> Maybe c -> d
applyH h Nothing = ???
```

We cannot magically produce a value for the `Nothing` case. We need to short-circuit the computation and return `Nothing`. That means our final result must be `Maybe d`:

```
applyH :: (c -> d) -> Maybe c -> Maybe d
applyH _ Nothing = Nothing
```

Next, we can write the `Just` case:

```
applyH :: (c -> d) -> Maybe c -> Maybe d
applyH _ Nothing = Nothing
applyH h (Just x) = ???
```

We could replace the `???` with 2 possible values. First is `Nothing`, but then our function wouldn't provide the promised functionality of applying `h` to the value in the `Maybe` Context. It simply would always throw away all data and always return `Nothing`. We might as well just replace `h` with `const Nothing`.

So, there's only one reasonable implementation:

```
applyH :: (c -> d) -> Maybe c -> Maybe d
applyH _ Nothing = Nothing
applyH h (Just x) = Just $ h x
```

This is great. So now we can rewrite our Function Application example from the previous section:

```
fgh :: ∀ a d. a -> Maybe d
fgh x = applyH h $ g $ f x
```

But what if we wanted `g` to also return a reason for the failure:

```
g :: ∀ b c. b -> Either String c
```

Now we have to write another function to apply `h` that works with `Either String`. Bells should be going off in your head at this point. This is going to be a problem for ALL Higher-kinded Types.

We would like to solve this problem in a more general fashion and, as it turns out, Category Theory gives us just the right solution, the ***Functor***.

Here's the definition of a Functor in PureScript:

```
class Functor f where ❶
  map :: ∀ a b. (a -> b) -> f a -> f b ❷

infixl 4 map as <$> ❸
```

❶ When we used `NonEmpty`, it used `f` in its Type Signature. That stood for `Functor` and we use the same variable name here. This is common in PureScript and Haskell. Another use of `f` as a Type Parameter is for `Foldable`. But typically when you see `f` it represents a `Functor`.

❷ Notice how `f` is a Higher-kinded Type here since it takes a Single Type Parameter, which means it has the Kind `Type -> Type`.

❸ Mapping over Functors is so common that there's an Operator Alias for it. Notice how it mirrors the Function Application Operator, `$`. Also, realize our `applyH` is just the `map` Method for `Maybe` where `f` is `Maybe`.

The `map` Method allows us to Apply the function `a -> b` to the `a` within the Context of `f` and modify it WITHOUT changing the Context. This is an important point. When we examine the Functor Laws, we'll see why this is true.

It's unfortunate that the function is called `map` since this is an overloaded term, e.g. `Maps` are key/value stores and functions map from the Domain to the Codomain. So while the function is called `map`, many times we say ***map over***, e.g. we want to map `f` over the `List`.

Let's write the Functor instance for `Maybe`:

```
instance functorMaybe :: Functor Maybe where
  map _ Nothing = Nothing
  map f (Just x) = Just $ f x
```

Does that look familiar? `map` is a generalized solution to our specific one that we tried to solve with `applyH`.

Now let's use this to rewrite our Function Application example:

```
fgh :: ∀ a d. a -> Maybe d
fgh x = h <$> g $ f x
```

Contrast that with the pure version, i.e. no Higher-kinded Types:

```
fgh :: ∀ a d. a -> d
fgh x = h $ g $ f x
```

The only difference is that `$` is replaced with `<$>`.

12.7. Two Perspectives of `map`

Look carefully at the `map` Type Signature:

```
class Functor f where
  map :: ∀ a b. (a -> b) -> f a -> f b
```

We can interpret this function as taking a function from `a` to `b` and a value `a` in some Context `f` and returning a `b` in that same Context.

Another way to state this is that the function `a -> b` is ***lifted*** into the Context and applied. You will hear the term ***lift*** a lot when talking about Functors.

Now, let's take that same Type Signature and add the implied Parentheses as we've done before:

```
class Functor f where
  map :: ∀ a b. (a -> b) -> (f a -> f b)
```

Now, how might we interpret this?

We can say that `map` ***lifts*** the function into the Functor, i.e. we take a function that operates on Simple Types (also referred to as Pure Values) and lifts it to work on Functor Types.

So, `map` can morph BOTH functions and values. Keep this in mind as we see how Functors work in Category Theory.

12.8. The Power of Abstractions

Abstract models come about when we see common patterns among many different concrete examples. For example, we build an abstract model of a dog when we are young. We notice that dogs have tails and they bark. This pattern was built up over repeated encounters with dogs. In the beginning our models are naïve.

A five-year-old child might tell us that all dogs bark and all dogs have tails. But as we grow in experience with dogs, we realize that some dogs don't have tails and there are some dogs that do not bark. But most do.

We augment our model of dogs and when we talk about a particular breed, we may qualify it with the fact that this particular breed doesn't bark.

Similar processes occur in Mathematics. We first learn about Groups. They are abstractions that were developed over many years of observation of common patterns in Math. But as we lived with Groups, we saw minor differences in certain concrete examples.

Once such difference is when the Binary Operator is Commutative. This is not a requirement to be a Group, but many operations that we encounter are Commutative and therefore we have specialized the Abelian Group to capture that fact.

These abstractions are great in Math and in Programming. We want to think abstractly once and then just recognize that we've encountered a new example of our abstraction.

Typeclasses are one such abstraction that we recently learned about. Let's look at a particular example:

```
class Eq a <= Ord a where
  compare :: a -> a -> Ordering

data Ordering = LT | GT | EQ
```

Here we have abstracted what it means to have an Order. Elements of Sets can potentially be Ordered. Not all things have Order, but when they do, we know we can use everything we've learned about Order-able things with our new example.

For example, if some Type is Order-able, then we can sort `Lists` or `Arrays` of them. We don't have to think about it, since we have an abstract concept of Ordering and Sorting that tells us that it's possible.

If you study Math long enough, you will start to see the same patterns over and over again. This is because the same ideas are expressed in different disciplines slightly differently, but they are essentially Isomorphic ideas.

There are many examples of this. One such pertinent example are Church Numerals in Lambda Calculus.

This is a way to represent Numbers using Lambda Notation:

```
0 = λf.λx.x
1 = λfλx.f x
2 = λf.λx.f (f x)
3 = λf.λx.f (f (f x))
```

Basically, the number is represented by how many times a function `f` is applied to `x`. In the zero case, it's applied 0 times. In the one case, it's applied 1 time and so on.

This may be easier to see using PureScript Notation:

```
0 = \f x -> x
1 = \f x -> f x
2 = \f x -> f (f x)
3 = \f x -> f (f (f x))
```

We can also represent this same idea with a Data Type in PureScript:

```
data Nat = Zero | Succ Nat ❶
```

❶ `Nat` stands for Natural Numbers

Notice that this is a Recursive definition since `Nat` is on both sides of the equal sign. `Zero` is our starting point and `Succ` stands for the Successor of the previous Natural Number.

Here's how we can represent numbers:

```
0 = Zero
1 = Succ Zero
2 = Succ (Succ Zero)
3 = Succ (Succ (Succ Zero))
```

The definition for `3` says that `3` is the Successor of the Successor of the Successor of `0`. Another way we could say this is that `3` is the Successor of `2`, which is the Successor of `1`, which is the Successor of `0`.

Notice that the function `f` from the Lambda Calculus's encoding is our `Succ` Data Constructor and the `x` in Church Numerals is our `Zero`.

And now let's see how numbers are represented in Set Theory:

```
0 = {}          = Ø
1 = { 0 }       = { Ø }
2 = { 0, 1 }    = { Ø, { Ø } }
3 = { 0, 1, 2 } = { Ø, { Ø }, { Ø, { Ø } } } }
```

Here `Ø` represents the Empty Set.

We start with an Empty Set, `Ø`, for `0` and then add `0` to that Set to get `1`.

To get `2`, we add `1` to one's set and so on.

The point here is that when you're creating a model in Mathematics, you don't want to introduce numbers if you can show that there is a mapping from elements in your model to numbers.

In Set Theory, everything is a Set. When creating a new area of Mathematics, you want to define the smallest number of elements and then build the whole universe from those primitive elements. Same with Lambda Calculus. The whole world are Lambda Expressions.

This barely scratching the surface of commonality that can be found between different areas of Mathematics.

Hopefully, from this simple example, you can see how it might be possible that we could abstract over many areas of Mathematics based on their similarities.

And that's exactly what Category Theory does. It's an abstraction over Mathematics. It allows us to learn about this super abstraction and then apply new areas of research to this abstraction and if we can make it fit, we instantly gain insight into the nature of our research.

Functional Programming can be modeled using Category Theory and in doing so, it has informed us immensely and has empowered us with powerful abstractions that we are just now embarking upon in this book, e.g Functors, Applicatives and Monads.

12.9. A Little Category Theory

Category Theory is extremely abstract and because of that Abstraction it can Abstract over many areas of Mathematics. The area we're concerned with is Programming and that's were we'll concentrate.

We're just barely going to scratch the surface here but the Functional landscape is fraught with Category Theory references and so a little basic knowledge will go a long way in helping to navigate beyond this book.

Let's start with a definition of a Category.



A Category consists of a Set of ***Objects*** and a Set of ***Morphisms*** between the Objects.

A pretty simple definition as one would expect with something so Abstract. Now the devil is in the details.

We get to pick what our Objects and Morphisms are, which depends on what we're trying to model. Since our interest is Programming, we'll pick our Objects as `Types` and our Morphisms as `Functions`. This Category is traditionally called `Hask` after Haskell. But works just fine for PureScript.

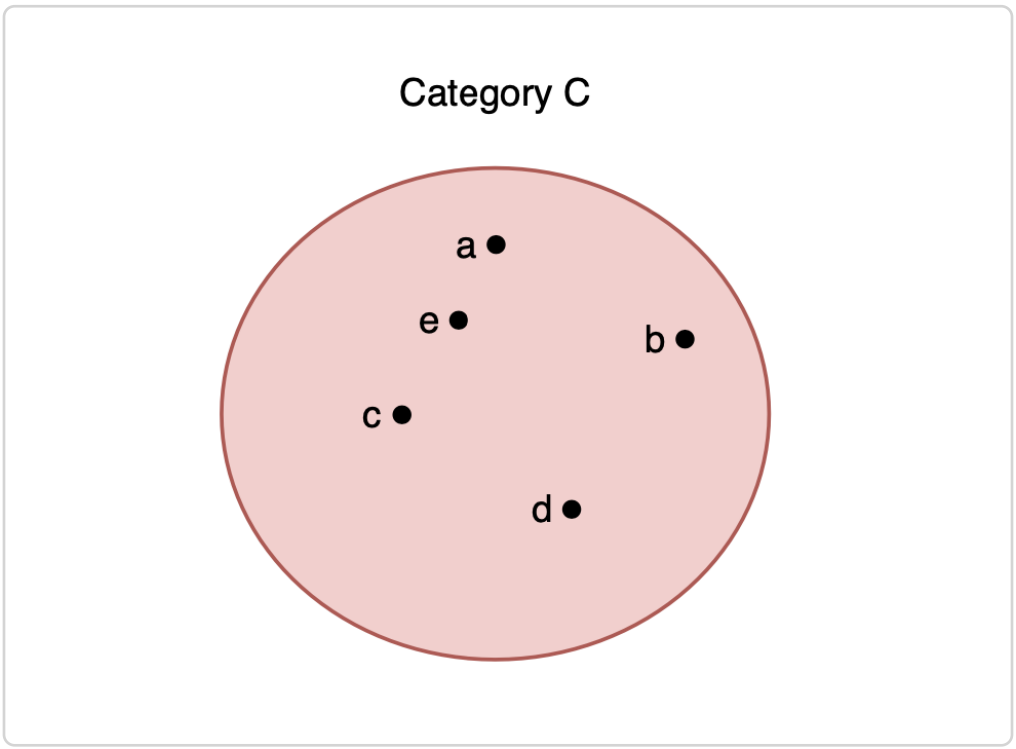
And as always in Math, there are laws:

- Each Morphism `f` has a Source Object `a` and Target Object `b` and is written like `f: a → b` and we say "f is a morphism from a to b"
- Morphisms compose
- There is an Identity Morphism for every Object, i.e. 1_x`: x → x`
- Composition is Associative: if `f: a -> b`, `g: b -> c`, and `h: c -> d` then `h ∘ (g ∘ f) = (h ∘ g) ∘ f`

This is a lot to digest at once, but, hopefully, you can see a lot of similarities with what we've been doing in PureScript so far.

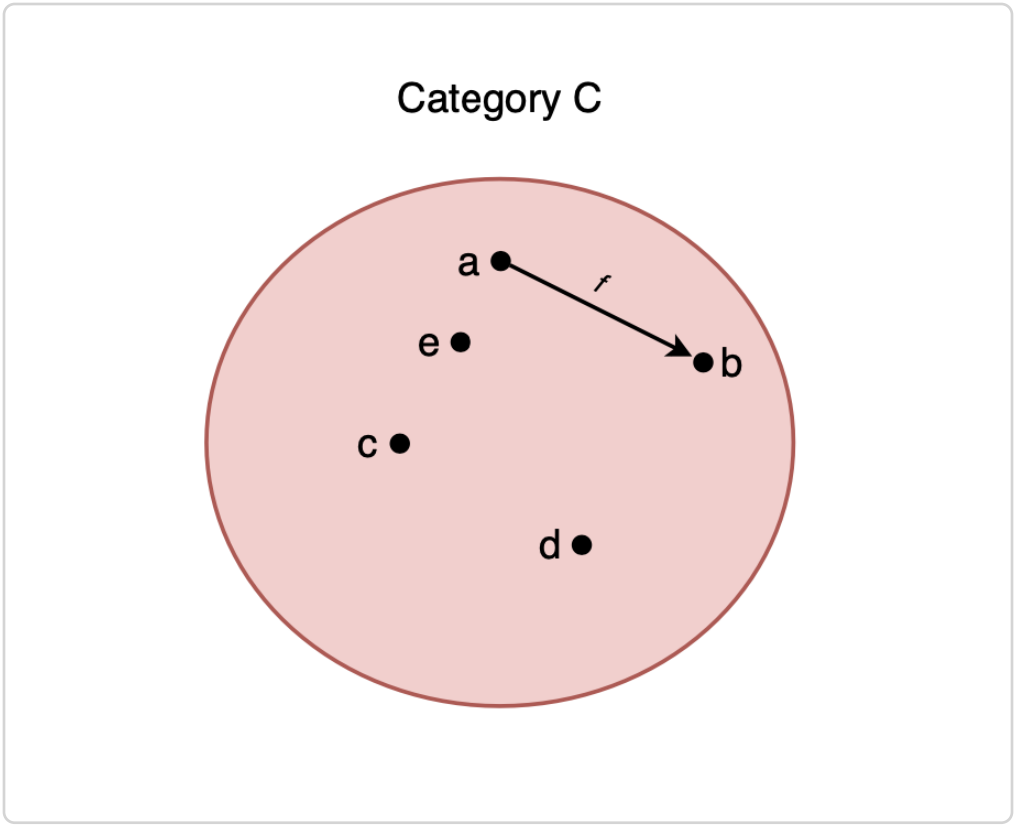
Let's take all of this one step at a time but this time with pictures:

Here's a Category, `C` with a Set of Objects, `{ a, b, c, d, e }`:



In PureScript, `a`, `b`, etc. are Types.

And here is a Morphism, `f`, from `a` to `b`, `f: a → b`:

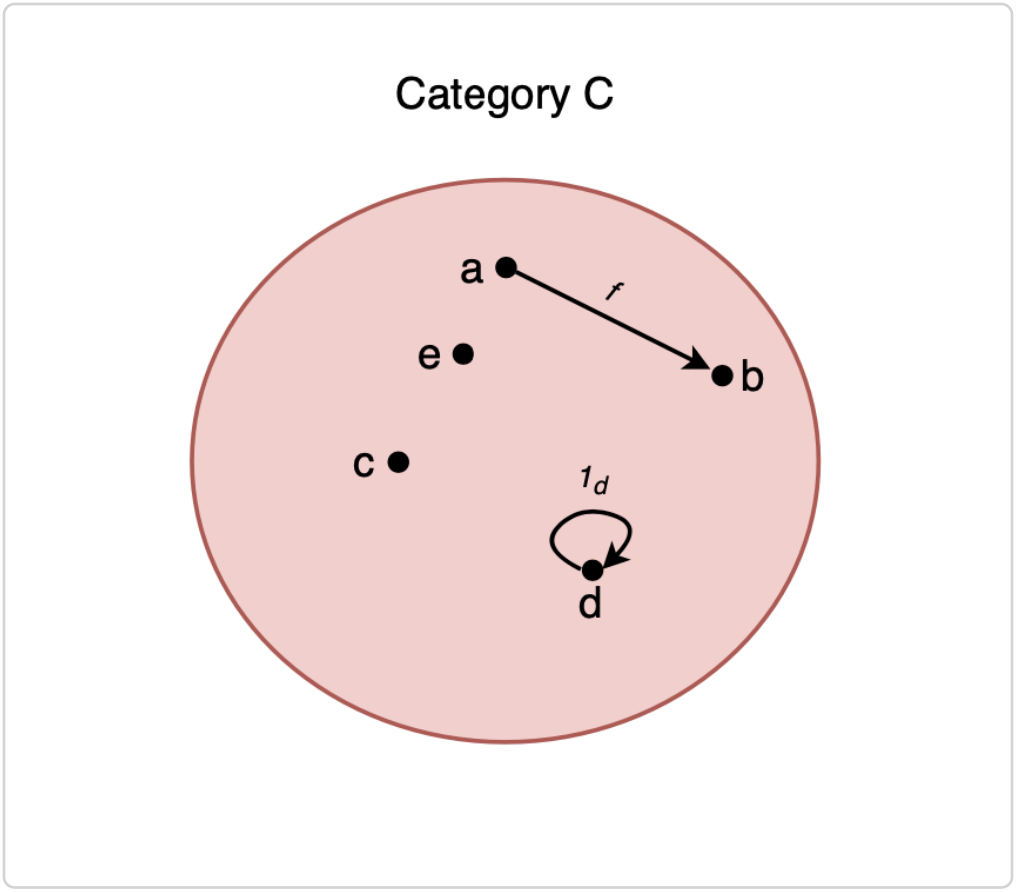


The Morphism `f` morphs Object `a` to Object `b`.

But since Objects are Types in `Hask`, when we write a function in PureScript, it morphs Values of Type `a` into Values of Type `b`:

```
f :: ∀ a b. a -> b
```

Next, we'll add an Identity Morphism for Object `d`, 1_d`: d → d` (all objects have Identity Morphisms but we're just showing for `d`):

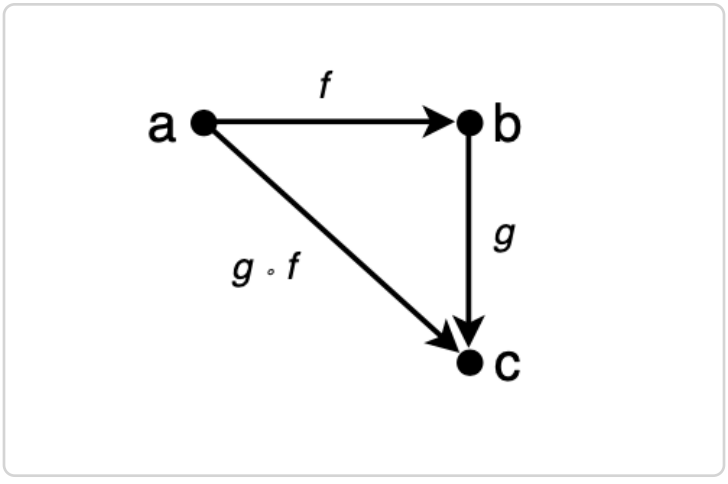


The Identity Morphism, 1_d, morphs Object `d` to Object `d`.

In PureScript, the Identity Morphism is called `identity`:

```
identity :: ∀ a. a -> a
identity x = x
```

Let's look at Functional Composition pictorially:



We can get from Object `a` to Object `c` taking 1 of 2 paths. It is said that this diagram ***Commutes***.

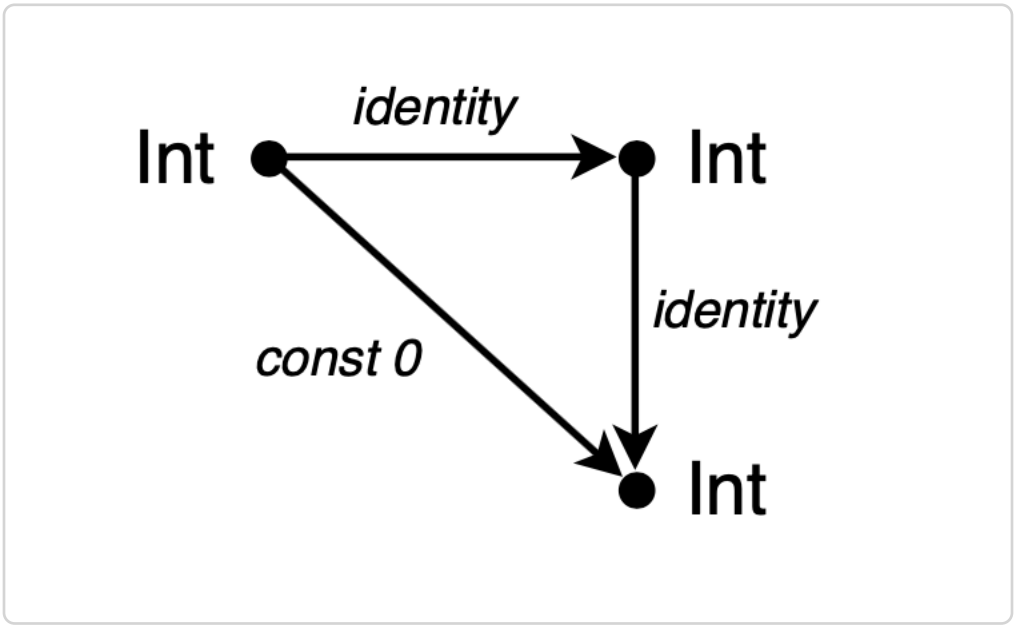


A Commutative Diagram is a diagram where all directed paths (arrows) with the same *start* and *endpoints* lead to the same result.

If we start at `a` with ANY value of Type `a`, we can go along the horizontal path through `f` and get a value of Type `b` and then take that value through `g` to a value of Type `c`.

This MUST produce the same value as starting back at `a` and taking the SAME value through `g ∘ f`.

Thanks to the definition of Composition, it does. But NOT all diagrams commute:



If we start at the upper left corner `Int` with any value and go through `identity` horizontally and then vertically through `identity` again, we end up in the lower right corner with that same value.

But if we start back at the upper left corner with the same value and go diagonally through `const 0`, we end up in the lower right corner with `0`.

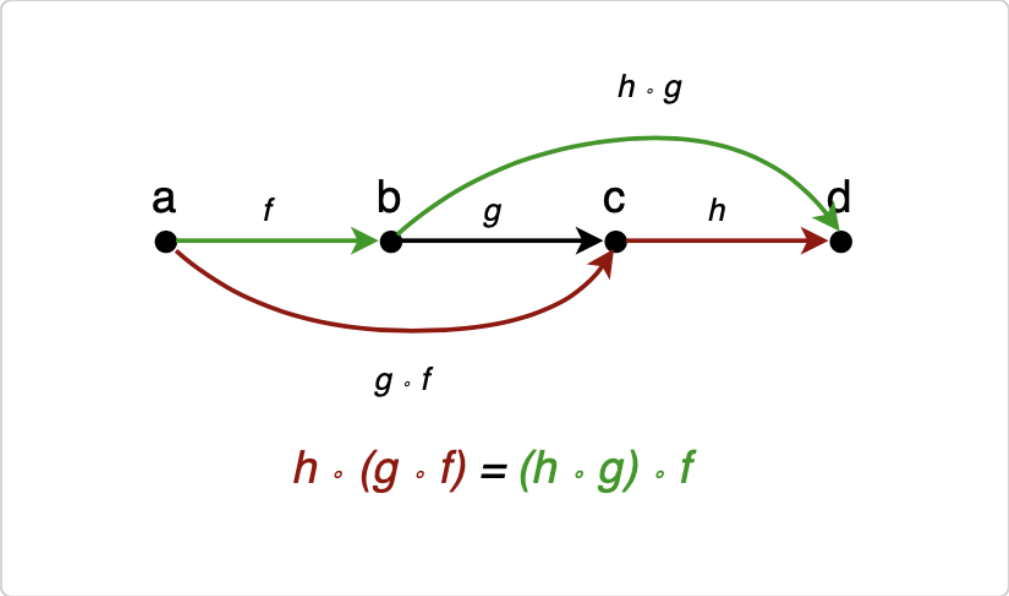
Taking these two paths from upper left corner to lower right corner doesn't produce the same result and therefore does NOT commute.

In PureScript, we'd write a function to compose like:

```
compose :: ∀ a b c. (b -> c) -> (a -> b) -> (a -> c)
compose g f x = g (f x)
```

Note, this is not how it's implemented in PureScript. It's actually implemented in a Typeclass called `Semigroupoid` which is like a `Category` without the Identity requirement. But for our purposes, this definition works.

Now we can examine the Associativity of Composition:



In PureScript, we can show that Composition is Associative:

```
compose :: ∀ a b c. (b -> c) -> (a -> b) -> (a -> c)
compose g f x = g (f x) ❶
```

❶ This equation will be used in the following proof.

```
h ∘ (g ∘ f)              = (h ∘ g) ∘ f              -- Given (Math)
compose h (compose g f) = compose (compose h g) f -- Given (PureScript)
h ((compose g f) x)      = compose (compose h g) f -- Subst left using ❶
h ((compose g f) x)      = (compose h g) (f x)     -- Subst right using ❶
h (compose g f x)        = (compose h g) (f x)     -- Remove Parentheses
h (g (f x))              = (compose h g) (f x)     -- Subst left using ❶
h (g (f x))              = h (g (f x))             -- Subst right using ❶
```

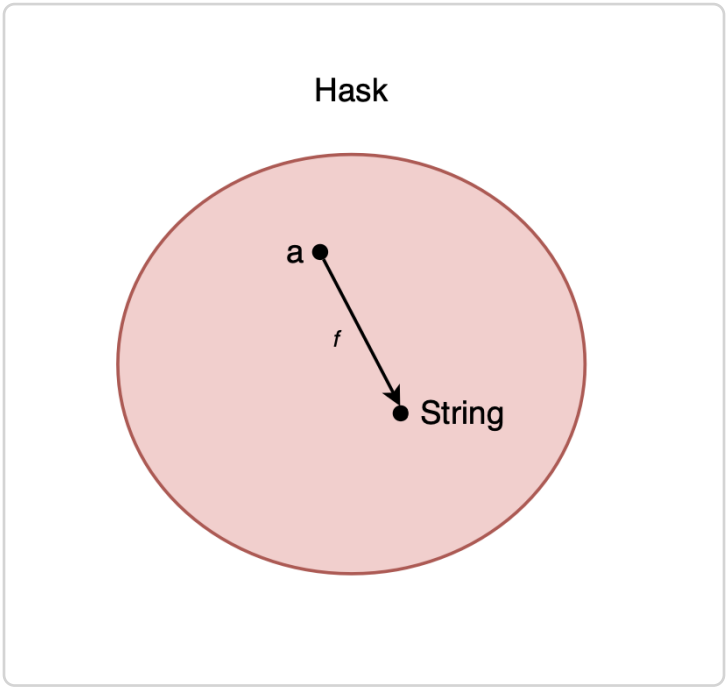
We have now shown how Types as Objects and Functions as Morphisms form a law-abiding Category. And after this very brief introduction to Category Theory, I hope you can see how Functional Language designers were influenced by this relatively new area of Mathematics (circa 1945).

12.10. Concrete Examples in Hask

Let's look at some examples in Hask starting with the Show Method, show:

```
class Show a where
  show :: a -> String
```

We can pictorially represent this as:



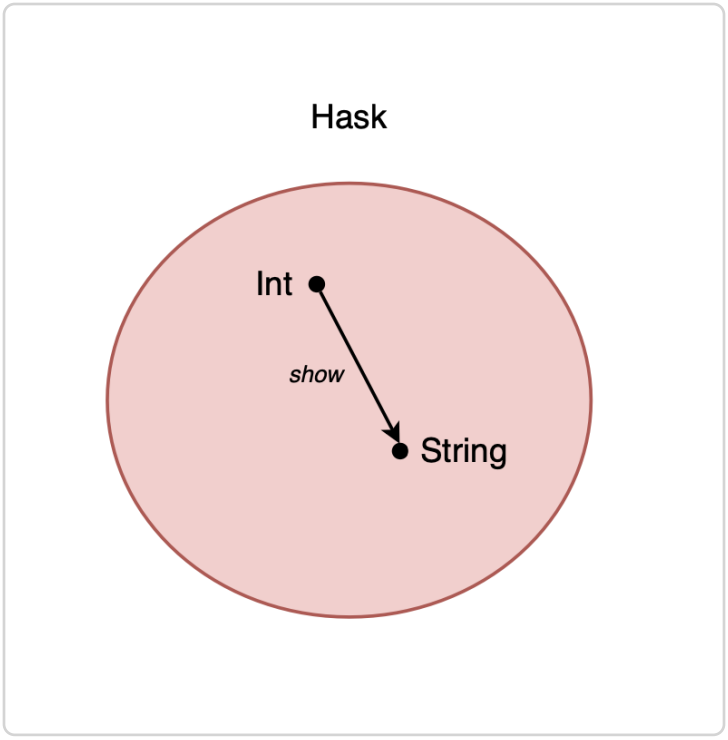
And for the `Int` instance of `Show`:

```
instance showInt :: Show Int where
  show = showIntImpl

foreign import showIntImpl :: Int -> String ❶
```

❶ Actual implementation in Javascript.

We can show this as:



Hopefully, these simple examples make sense based on what we've done so far. But those examples seems too simplistic. What about functions of multiple Parameters?

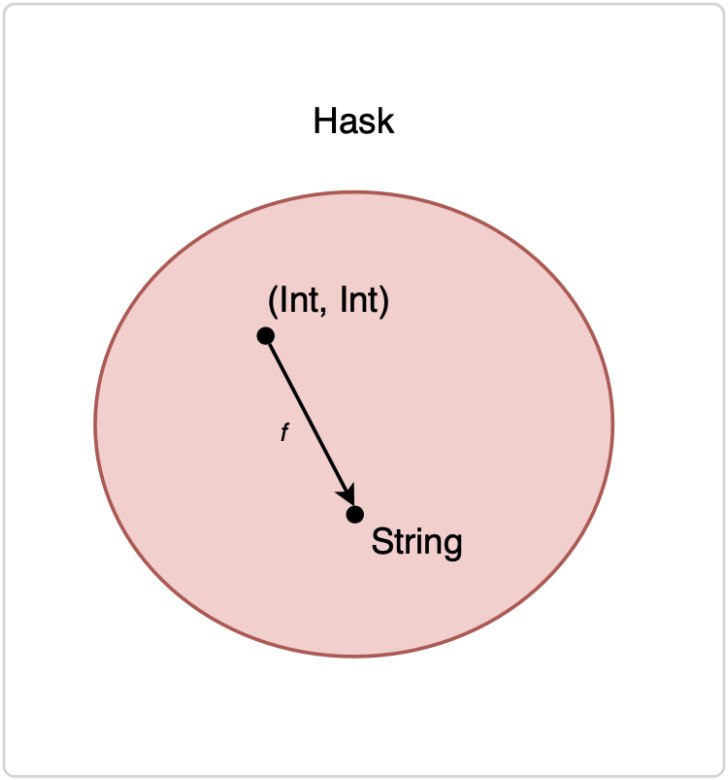
```
f :: Int -> Int -> String
```

Remember that all functions are Curried. And it possible to rewrite this function in its Uncurried form:

```
f :: Tuple Int Int -> String ❶
```

❶ Technically speaking, we have ***uncurried*** this function. (More on this later)

Now that it takes a single Parameter, we can draw it in Category Hask:



Hopefully, you can imagine how we'd do this with a function of 3 Parameters:

```
g :: Int -> Boolean -> String -> String

g' :: Tuple Int (Tuple Boolean String) -> String
```

And so on.

After this brief introducing, we should now be able to see how Functors work in Category Theory and compare it with Functors in Programming.

12.11. Functors in Category Theory

So far, we've only looked at a single Category and ways to morph Objects in that Category into other Objects in the SAME Category. This is what Functions do in PureScript:

```
f :: ∀ a b. a -> b ❶
```

❶ Maps Types to Types.

While Morphisms are mappings between Objects, Functors are mappings between Categories. And in PureScript, as we have seen, Functors are just Functions, viz. `map`.

But in Programming, we only have one Category since our Functions can only map Types to Types. Functors

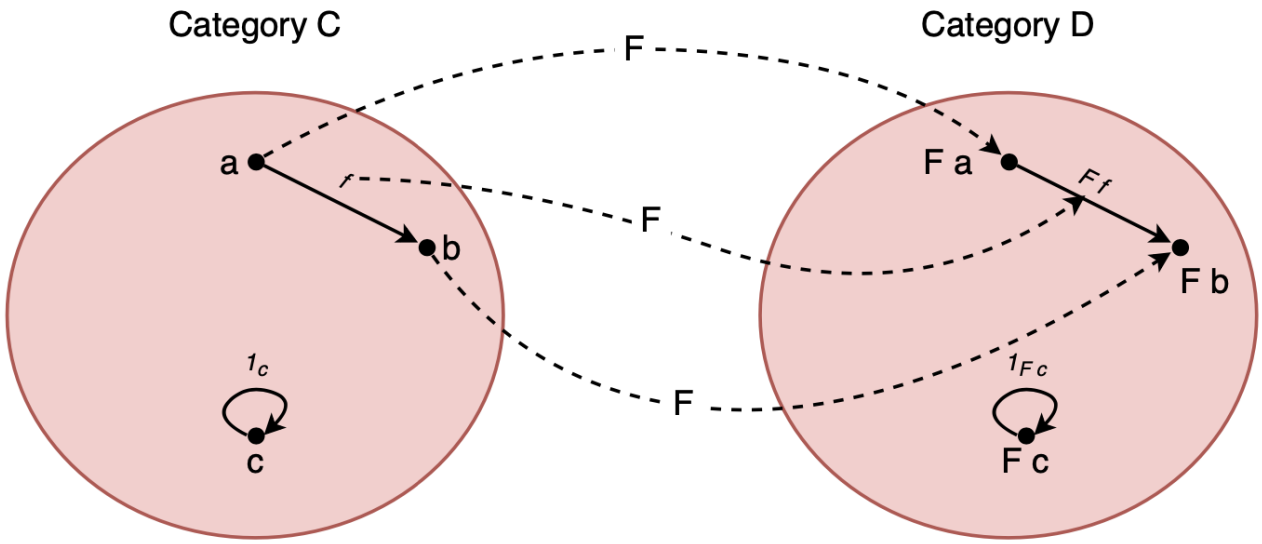
that map back to the SAME Category are known as ***EndoFunctors***. So all `Functors` in PureScript are really EndoFunctors but everyone just says ***Functors***.

There's another really important requirement of Functors.



Functors preserve Categorical Structure. In Abstract Algebra, functions that preserves the Algebraic Structure of say a Group are called ***Homomorphism*** (***Homo*** means ***same*** and ***morph*** means ***shape***). Here Functors are like Homomorphisms since they too preserve the Categorical Structure.

If a Object `a` has a Morphism to Object `b` then a Functor must preserve that same Morphism between the Objects when mapping them into the new Category. It also, cannot drop any Objects or Morphisms when mapping nor can it create any new Objects or Morphisms that weren't in the original Category:



Here the Functor `F` maps both Objects and Morphisms from Category `C` to `D`.

In this diagram, `F` maps `a` to `F a` and `b` to `F b`, i.e. maps all Objects (not all mappings shown).

It also shows that `F` maps `f` to `F f`, i.e. maps all Morphisms (not all mappings shown).

When we look at `map` in PureScript with the implied Parentheses:

```
class Functor f where
  map :: ∀ a b. (a -> b) -> (f a -> f b) ❶

infixl 4 map as <$>
```

❶ We added the implied Parentheses.

we can see the mapping of Morphisms, i.e. `a -> b` is mapped to `f a -> f b`.

But where is the mapping of `a` to `f a` and `b` to `f b` in PureScript. For that, we're going to have to wait for Applicative Functors. But here's a preview:

```
class Apply f <= Applicative f where ❶
  pure :: ∀ a. a -> f a
```

❶ `Functor` is a Superclass of `Apply`, which means that `Applicative` is also a `Functor`. Therefore, Applicatives has both `map` and `pure`, which cover the Categorical Functor mappings of Objects (`pure`) and Morphisms (`map`).

We'll encounter Applicative Functors soon enough, but for now there's so much to learn and do with Functors.

12.12. Functor Laws

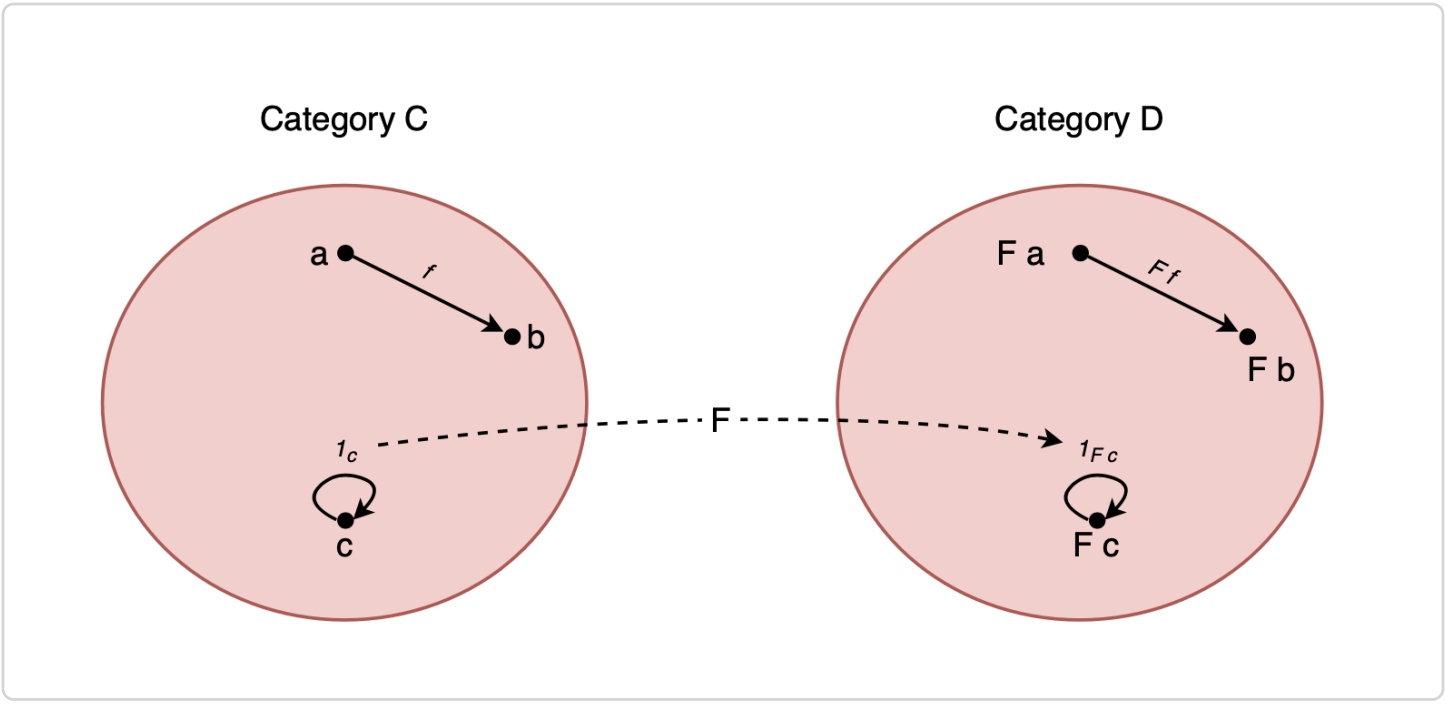
There is no way to enforce the `Functor` laws in PureScript. The burden is on the author of the Typeclass to verify that their implementation abides by these laws:

```
map identity = identity            [Identity]
map (g <<< f) = map g <<< map f    [Composition]
```

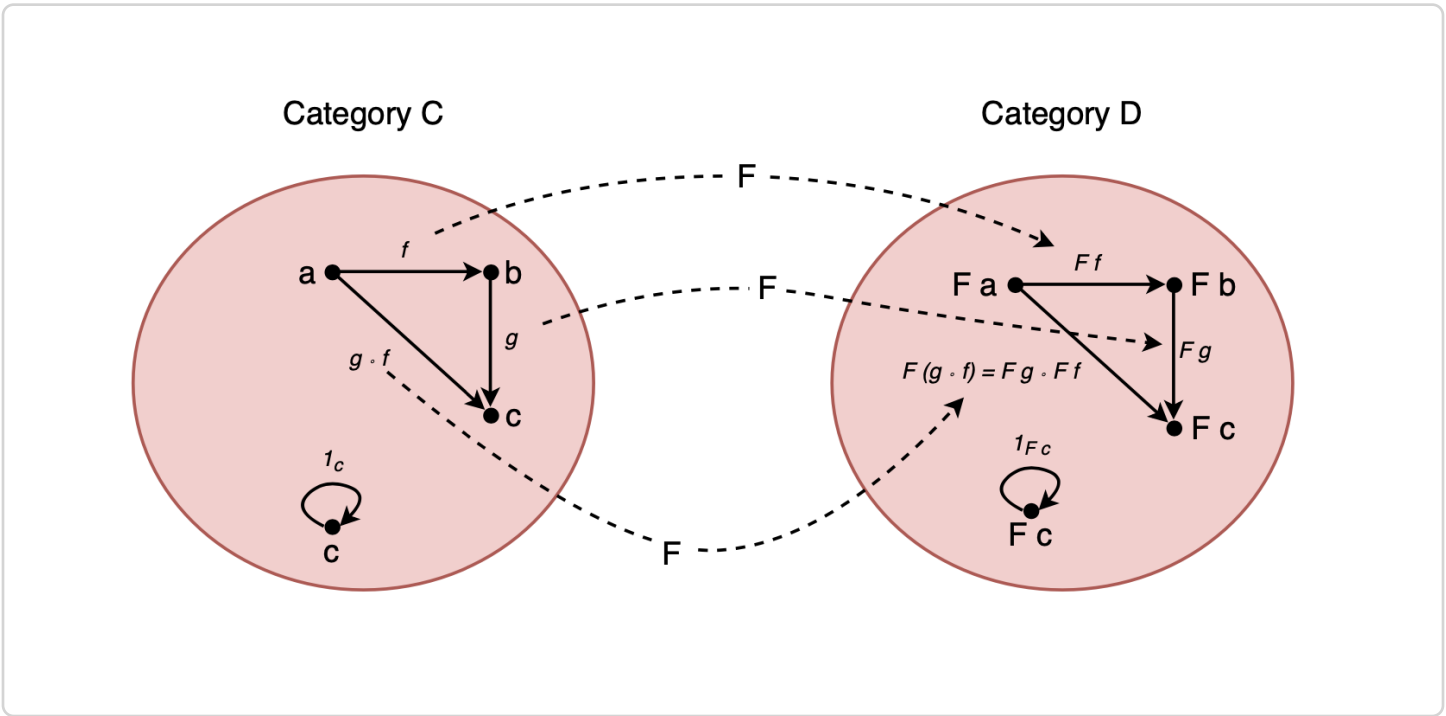
In short, these laws dictate that `Functors` must preserve Identity and Composition.

The `Identity` law says that if we `map` the `identity` function, then it's equivalent to just calling `identity` on the `Functor`. Notice that this law means that `map` can NEVER change the structure of the `Functor`. So a `Just` can never become a `Nothing` and a `Left` cannot change to a `Right`. In other words, mapping `identity` leaves the `Functor` intact.

Categorically speaking, a Functor maps the Identity Morphism in Category `C` to the Identity Morphism in Category `D`:



The `Composition` law says that mapping the Composition of 2 functions is equivalent to mapping each and Composing the results. We can see this pictorially using Categories:



Mapping `g ∘ f` to `F (g ∘ f)` is equivalent to mapping `f` to `F f` and `g` to `F g` and then composing them to get `F g ∘ F f`. This makes sense since Functors must preserve the Categorical Structure and rules, one of which is that morphism compose.

Leaving Category Theory for the moment and returning to Programming, we can see that from the `Identity` law that mapping the identity function (or morphism) has no effect.

Let's take a look at what that means for us using Maybe:

```
instance functorMaybe :: Functor Maybe where
  map _ Nothing   = Nothing
  map f (Just x)  = Just $ f x
```

Here Nothings stay Nothings and Justs stay Justs. If they didn't then we'd be breaking the Identity law.

As do Lefts and Rights for Either:

```
instance functorEither :: Functor (Either a) where
  map _ (Left x)  = Left x
  map f (Right y) = Right $ f y
```

12.13. Verifying the Functor Laws for Maybe

Let's start with:

```
data Maybe a = Nothing | Just a

instance functorMaybe :: Functor Maybe where
  map _ Nothing   = Nothing ❶
  map f (Just x)  = Just $ f x ❷
```

❶ This equation will be used in the following proof.

❷ This equation will be used in the following proof.

And let's verify the Laws. Our approach will be to start with the Law and then using our implementation, reduce the left side of the equation and then the right. If the results are identical then they're obviously equal.

First, Identity for Nothing:

```
map identity Nothing  = identity Nothing        -- Identity Law
Nothing               = identity Nothing        -- Subst left using ❶
Nothing               = Nothing                 -- Function Application
```

Next, for Just:

```
map identity (Just x) = identity (Just x)       -- Identity Law
Just (identity x)     = identity (Just x)       -- Subst left using ❷
Just x                = identity (Just x)       -- Function Application
Just x                = Just x                  -- Function Application
```

Next, we'll verify the Composition Law for `Nothing`:

```
map (g <<< f) Nothing = (map g <<< map f) Nothing -- Composition Law
Nothing               = (map g <<< map f) Nothing -- Subst left using ❶
Nothing               = map g (map f Nothing)     -- Function Composition
Nothing               = map g Nothing            -- Subst right using ❶
Nothing               = Nothing                  -- Subst right using ❶
```

And finally, using `Just`:

```
map (g <<< f) (Just x) = (map g <<< map f) (Just x) -- Composition Law
Just ((g <<< f) x)     = (map g <<< map f) (Just x) -- Subst left ❷
Just (g (f x))         = (map g <<< map f) (Just x) -- Func Composition
Just (g (f x))         = map g (map f (Just x))     -- Func Composition
Just (g (f x))         = map g (Just (f x))         -- Subst right ❷
Just (g (f x))         = Just (g (f x))             -- Subst right ❷
```

12.14. Functor Instance for `Either`

Let's write the `Functor` Instance for `Either` following what we did for `Maybe` while keeping in mind that we're short-circuiting for `Left`:

```
data Either a b = Left a | Right b

instance functorEither :: Functor Either where -- COMPILE ERROR!!
  map _ (Left x)  = Left x ❶
  map f (Right y) = Right $ f y ❷
```

❶ Short-circuiting.

❷ Applying `f` to the value in the Context, i.e. `y`.

Let's look at the compiler error:



Where the salient portion is:

```
Could not match kind Type with kind Type -> Type
```

Seems like we have a Kind mismatch. Let's look at the Kind for `Either`:

```
Either :: Type -> Type -> Type
```

And let's look at the Kind for `Functor`:

```
???
```

Wait. How do we get the Kind for a Typeclass? We can use the REPL to get the Kind for a Type like `Either`, but there is no way to do that for a Typeclass.

We're just going to have to look at the definition and figure it out:

```
class Functor f where
  map :: ∀ a b. (a -> b) -> (f a -> f b)
```

Now that we're looking at the `class` definition, it's clear that `Functor` takes a single Type Parameter `f`. But what is it's Kind?

We have to look further into the definition to see how it's used. When we do we can see `f a` and `f b`. From the `∀ a b`, we know `a` and `b` are Types, which tells us that `f` has Kind:

```
 f :: Type -> Type
```

So this is where the Kind mismatch is. We put `Either` in place of `f`. But `Either's` Kind doesn't match `f's`:

```
f :: Type -> Type
Either :: Type -> Type -> Type
```

But we can fix this problem by Partially Applying Types to Either. Just like functions, we can Partial Apply Type Constructors which are function-like:

```
f :: Type -> Type
Either a :: Type -> Type
```

Now the Kinds match. Let's apply this fix to our instance:

```
data Either a b = Left a | Right b

instance functorEither :: Functor (Either a) where ❶
  map _ (Left x)  = Left x
  map f (Right y) = Right $ f y
```

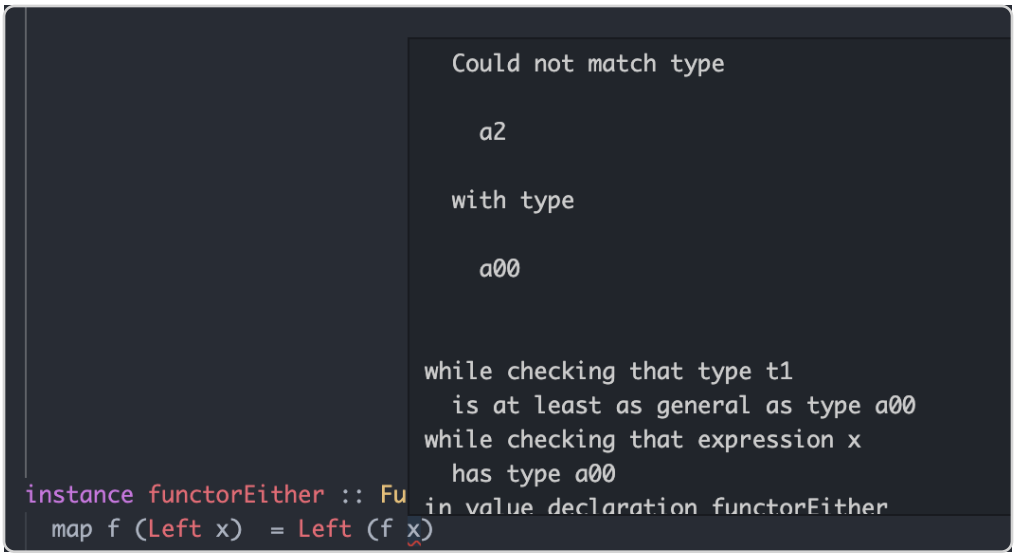
❶ Replaced `Either` with `Either a`.

We need to take note that `a` in `Either a` is implied to be `∀ a` and doesn't need to be explicitly defined here like we do in functions.

But there's a bigger issue going on here. Because a `Functor's` Type Parameter, `f`, only takes 1 Type Parameter, `a`, we cannot map `Left` even if we wanted to. Let's see what happens if we try:

```
instance functorEither :: Functor (Either a) where
  map f (Left x)  = Left $ f x  -- COMPILER ERROR!!
  map f (Right y) = Right $ f y
```

Here's the compiler error:



The Type names in the error are a bit confusing and the rest of the error isn't much help here either. The best I can surmise from this is that the compiler cannot match 2 different `a's`.

But if we are patient and take the time to think it through, we can understand why this won't work.

Let's take a radical approach. We're going to replace `f` with `Either a` in the Typeclass Definition for `Functor`. This won't compile, but it'll help us understand what's going on here.

Start with `Functor`:

```
class Functor f where
  map :: ∀ a b. (a -> b) -> (f a -> f b)
```

Replace `f` with `Either a'` (we're using `a'` so it doesn't collide with the `a` in `map's` Type Signature):

```
-- WON'T COMPILE - ONLY FOR ILLUSTRATIVE PURPOSES
class Functor (Either a') where
  map :: ∀ a b. (a -> b) -> ((Either a') a -> (Either a') b)
```

From this, we can see that the function, `f :: a -> b`, passed to `map` morphs the SECOND Type Parameter of `Either a' a`.

The FIRST Type Parameter stay constant and certainly doesn't match the `a` that `map's` function expects.

This is why we cannot use `f` on the `Left` value. `Left's` Type is `a'`, which must stays constant.

Now, the compiler error makes more sense. It cannot match `a` to `a'`. That's the 2 different `a's`.

Let's remind ourselves of the PROPER `Functor` for `Either`:

```
instance functorEither :: Functor (Either a) where
  map _ (Left x)  = Left x
  map f (Right y) = Right $ f y
```

What we've learned here is that when trying to create a Functor instance for a Higher-kinded Type that has multiple Type Parameters, then all of them are held constant except for the RIGHTMOST Type Parameter.

Keep this in mind when we write the Functor for Tuple a b in the Exercises.

12.15. Functor Instance for Choice

It didn't matter that we decided to Short-circuit Either, since Functor dictates that for us.

But we have real problem with our Choice Data Type:

```
data Choice a b = PickA a | PickB b
```

We decided that this would NOT Short-circuit. But we just saw that Functor forces our hand on all but the last Type Parameter.

If we make Choice a Functor, we have to short-circuit for PickA:

```
instance functorChoice :: Functor (Choice a) where
  map _ (PickA x)  = PickA x ❶
  map f (PickB y) = PickB $ f y
```

❶ Short-circuit that we do NOT want.

So is there a Functor that will let us map both?

By now, you know I wouldn't ask the question if there wasn't a way.

12.16. Bifunctor Typeclass

We've seen that Functor can only map 1 of the Type Parameters. To map 2 Type Parameters we need to use something called a Bifunctor (***Bi*** means two).

Here's the Bifunctor in PureScript:

```
class Bifunctor f where
  bimap :: ∀ a b c d. (a -> c) -> (b -> d) -> f a b -> f c d ❶
```

❶ The Type Constructor, `f`, takes 2 arguments, unlike `Functor` where it only took 1.

`bimap` takes 2 functions, one to map the first Type, `a`, and one to map the second Type, `b`.

Now we can create our `Bifunctor` Instance for `Choice` without being forced to short-circuit:

```
instance bifunctorChoice :: Bifunctor Choice where ❶
  bimap f _ (PickA x) = PickA $ f x
  bimap _ g (PickB y) = PickB $ g y
```

❶ We no longer have to use `Choice a` like we did with `Functor` that's because `Bifunctor's f` takes 2 Parameters and so does `Choice`. In other words, their Kinds match, i.e. `Choice :: Type -> Type` matches `f :: Type -> Type`.

Now we map both cases of `Choice` since we now have a tailor-made function for each case.

`Choice` is a Sum Type (Coproduct), but what about a Product Type? Can you think of a Product Type we can make a `Bifunctor`?

What about this one:

```
data Tuple a b = Tuple a b

instance bifunctorTuple :: Bifunctor Tuple where
  bimap f g (Tuple x y) = Tuple (f x) (g y)
```

Here we map both sides of the `Tuple` every time. But what if we only wanted to map the left side of the `Tuple`?

We could write the following function:

```
lmap :: ∀ a b c f. Bifunctor f => (a -> b) -> f a c -> f b c
lmap f = bimap f identity ❶
```

❶ We provide `identity` as the right side mapping, effectively mapping the right side back to itself.

Turns out that this function and `rmap` are in the PureScript `Bifunctor` library, `package-bifunctors`, in the module, `Data.Bifunctor`.

`rmap` maps the right side of the `Bifunctor`, which is basically just `map` from `Functor`.

12.17. Bifunctors in Category Theory

Let's return the `Hask` Category where Objects are Types and Morphisms are functions.

When we looked at `Functors` in `Hask`, we saw how it mapped Types to other Types. But we only had a single Type to work with and `Bifunctors` map 2 Types to 2 Types.

So, for `Bifunctors`, our Objects must be 2 Types. So we need a Type that's 2 Types. If this sounds impossible or paradoxical, it's actually not.

Let's look at few things before we cross that bridge. First, let's refresh our memories regarding ***Cartesian Products***. Here's an example:

```
A = { 1, 2, 3 }
B = { a, b }

A × B = { (1, a), (1, b), (2, a), (2, b), (3, a), (3, b) }
```

The Cartesian Product is a `Set of Pairs` created by taking every element of `Set A` and combining it with every element in `Set B`.

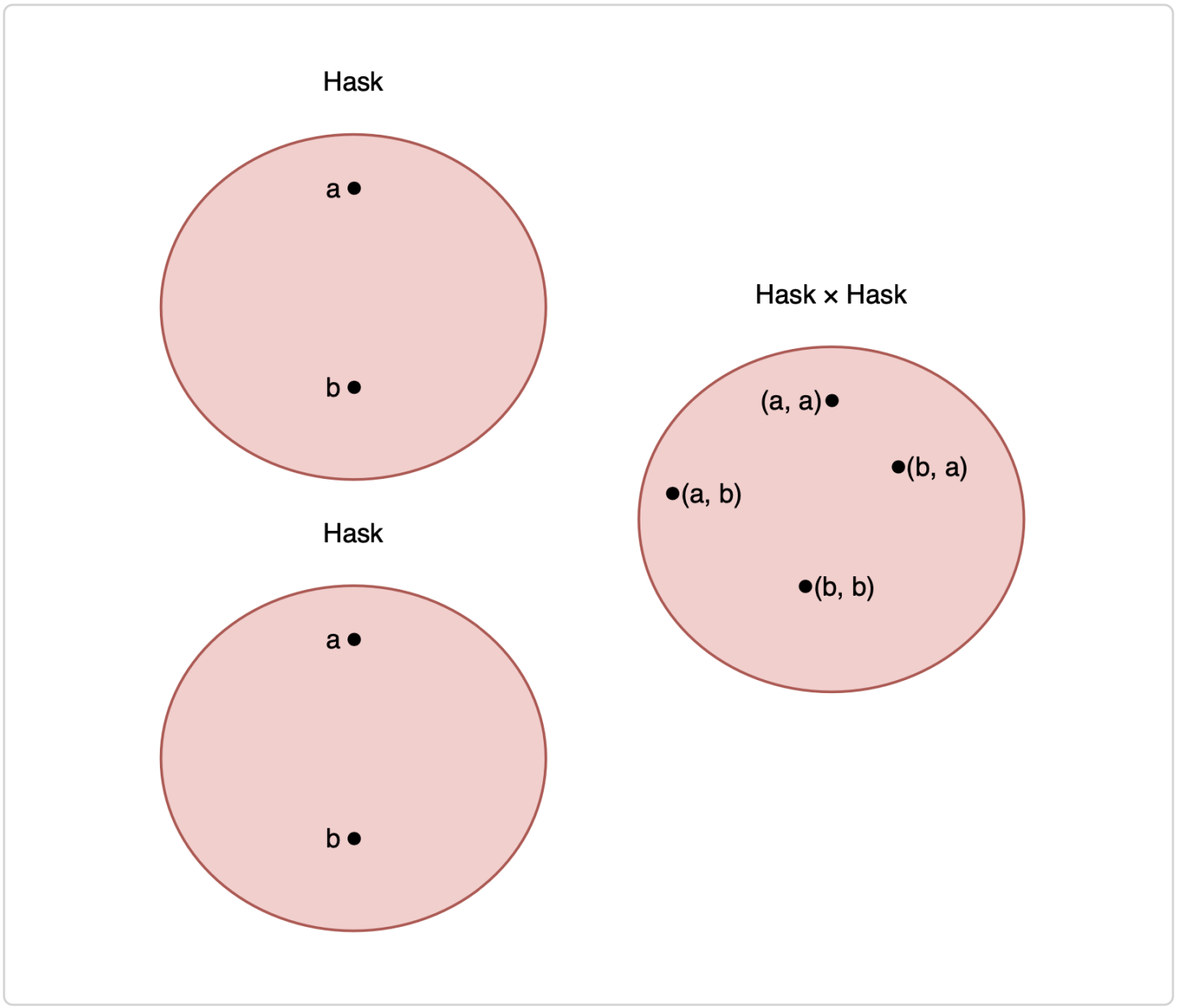
And here's the formal definition:

```
A × B = { (a, b): a ∈ A and b ∈ B }
```

Once again, let's parse out the Math:

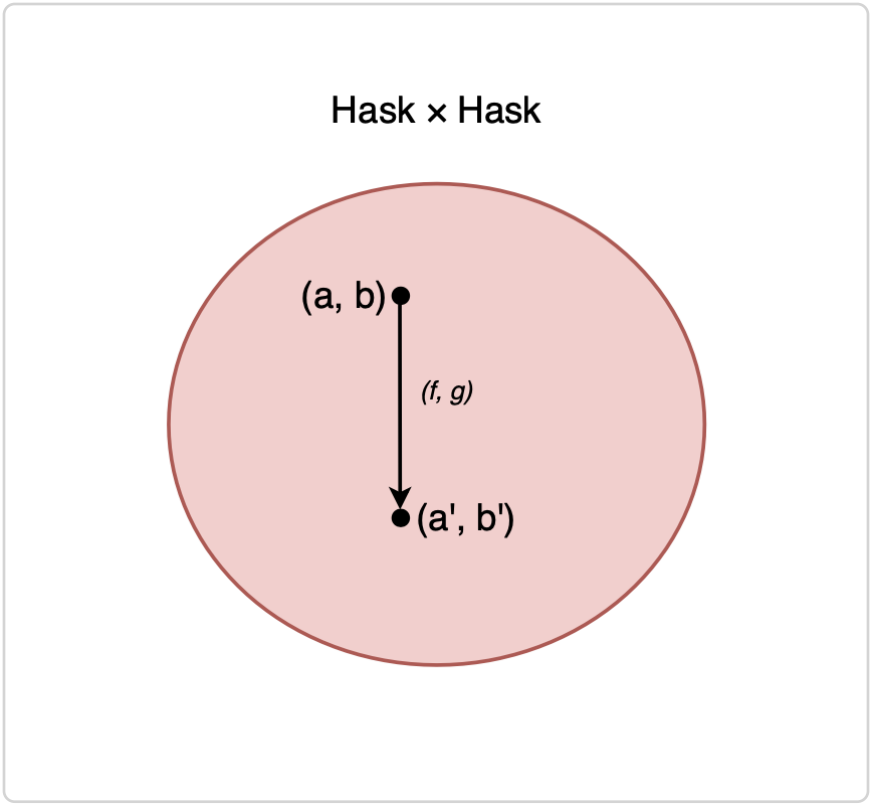
```
A × B                 (Cartesian Product between Sets A and B)
{ ... }               (encloses a Set)
(a, b)                (pair of elements)
a ∈ A                 (a is an element of Set A)
b ∈ B                 (b is an element of Set B)
```

In Categories, Objects and Morphisms are Sets. In particular, in `Hask`, we have a `Set of Types`. And for `Bifunctor` we need a Category where Objects are 2 Types. We can create such a Category by taking the Cartesian Product of `Hask` with `Hask`:



Our new Category `Hask × Hask` was conjured up with some Black Magic by just taking the Cartesian Product of our Objects. But is our new Category a valid Category?

Let's make sure that Morphisms compose. But wait, what's a Morphism in our new Category? In `Hask`, it's just a function from Type to Type, but we have a Pair of Types, which means we're going to need a Pair of functions.



We next need to show that our Morphisms compose:

```
(f, g) ∘ (f', g') = (f ∘ f', g ∘ g')
```

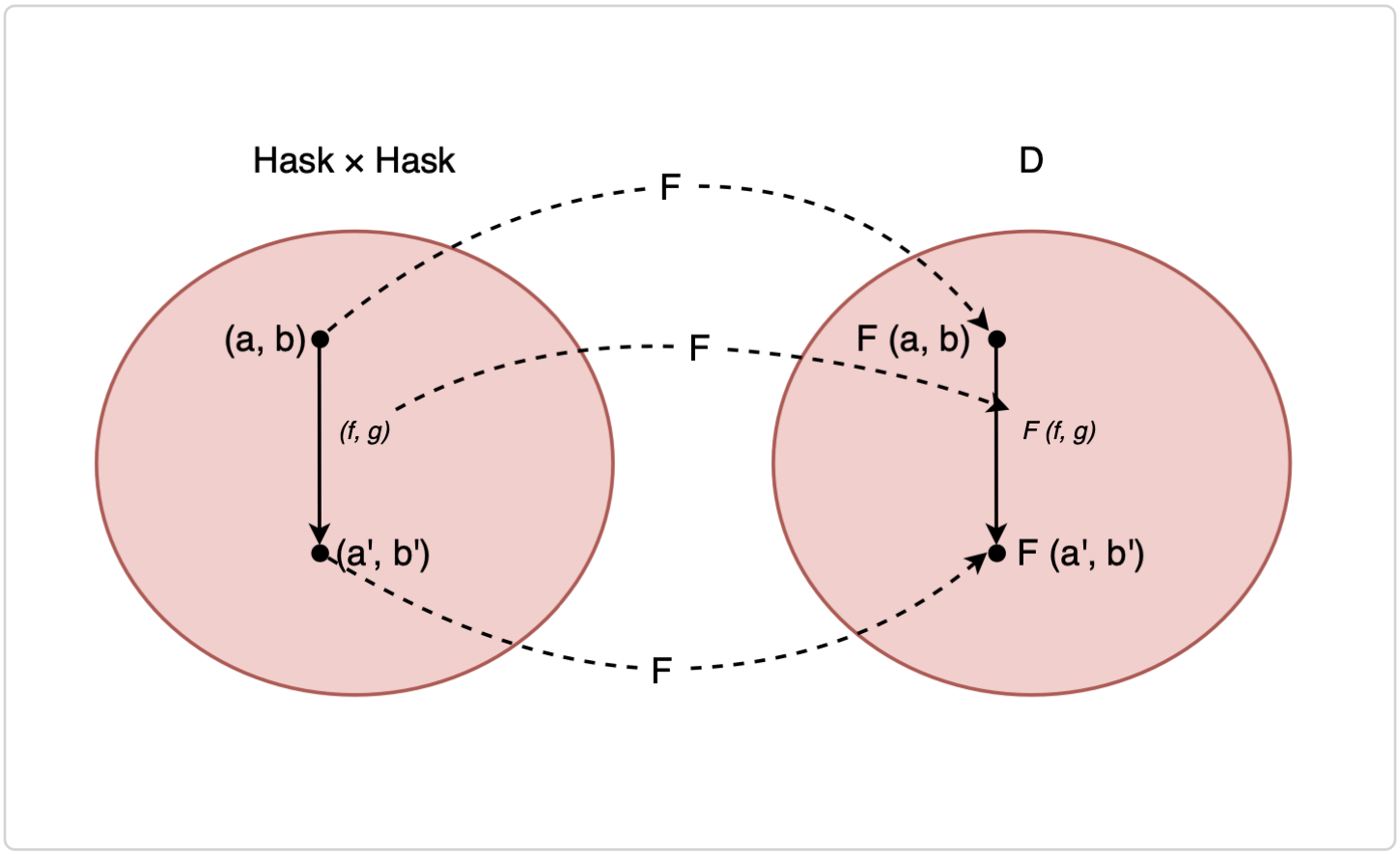
And we need to define the Identity Morphism:

```
(identity, identity) ❶
```

❶ Just a Pair of `identity` functions.

So, this Category seems valid. I say seems because we haven't exhaustively proven everything, e.g. Associativity. But it holds.

Now, we can look at Functors in `Hask × Hask`:



This is pretty unremarkable. It looks just like any Functor between Categories with the slight exception of operating on a Pair of Types, which is basically the Definition of ***Bifunctor***, which is as follows:

```
For Categories, C, D and E, the Functor

        F: C × D → E

is also called a Bifunctor from C and D to E.
```

`C x D` in this definition is our Category `Hask x Hask` in the above diagram. And E in this definition is our Category `D` in the above diagram.

Finally, we're ready to see how `Either` is a `Bifunctor`. But first let's take a short detour.

First thing to note is that we represent Pairs in PureScript using `Tuple`, therefore we code the ordered pair `(a, b)` as:

```
Tuple a b
```

Next thing to note are these helpful functions that form an Isomorphism for functions:

```
uncurry :: ∀ a b c. (a -> b -> c) -> (Tuple a b -> c)
uncurry f = \(Tuple x y) -> f x y ❶

curry :: ∀ a b c. (Tuple a b -> c) -> (a -> b -> c)
curry f = \x y -> f (Tuple x y) ❷
```

❶ `uncurry` takes a normal function of 2 Parameters and returns a function that takes 1 Parameter, i.e. both parameter values in a `Tuple`.

❷ `curry` takes a function of 1 Parameter that consists of a `Tuple` and returns a function of 2 Parameters.

These functions form an Isomorphism for functions since they are inverses of each other:

```
curry <<< uncurry = identity
```

This means that any function can be rewritten in an "uncurried way":

```
f :: ∀ a b c. a -> b -> c       -- normal function

g :: ∀ a b c. Tuple a b -> c    -- uncurried version of f
```

These two Type Signatures are Isomorphic since we can use `uncurry` to convert `f` to `g` and `curry` to convert `g` back to `f`.

Now we're ready to examine `Either's` Kind Signature:

```
Either :: Type -> Type -> Type
```

We can rewrite this Kind Signature ***uncurried***:

```
Either :: Tuple Type Type -> Type
```

This has the Mathematical equivalent to:

```
(Type, Type) -> Type
```

which looks just like a Bifunctor. The left side is our Pair of Types like we saw in `Hask × Hask`, e.g. `(a, b)` and produces a single Type `F (a, b)`.

Referring to the uncurried Kind Signature for `Either`, `(a, b)` or `Tuple a b` is the first Type Parameter to

`Either` and `F (a, b)` is just `Either a b`, which is what the `Either` Type Constructor returns. In other words, `Either` maps 2 Types, `a` and `b` to the Type `Either a b`.

This means that ANY Type that has at least 2 Type Parameters is a Bifunctor and therefore a `Bifunctor` Instance can be written for it.

We saw this with `Choice`:

```
data Choice a b = PickA a | PickB b

instance bifunctorChoice :: Bifunctor Choice where
  bimap f _ (PickA x) = PickA $ f x
  bimap _ g (PickB y) = PickB $ g y
```

And with `Tuple`:

```
data Tuple a b = Tuple a b

instance bifunctorTuple :: Bifunctor Tuple where
  bimap f g (Tuple x y) = Tuple (f x) (g y)
```

They both have 2 Type Parameters. But what about this Type:

```
data Threeple a b c = Threeple a b c
```

We can only map over the last 2 Parameters, `b` and `c`, which means we must "fix" Type `a`:

```
data Threeple a b c = Threeple a b c

instance bifunctorTuple :: Bifunctor (Threeple a) where ❶
  bimap f g (Threeple x y z) = Threeple x (f y) (g z) ❷
```

❶ Here we "fix" `a`.

❷ `x`, which is of Type `a`, is unaffected by the `bimap` function.

12.18. Bifunctor Laws

You didn't think we'd get away without looking at the Bifunctor Laws did you?

They're actually pretty straightforward:

```
bimap identity identity = identity       [Identity]
bimap (g1 <<< f1) (g2 <<< f2) =
           bimap g1 g2 <<< bimap f1 f2   [Composition]
```

These are analogous to the Functor Laws:

```
map identity = identity           [Identity]
map (g <<< f) = map g <<< map f   [Composition]
```

See if you can see how they parallel each other.

12.19. Functors in Perspective

Normal function application is done as follows:

```
f $ x ❶
```

❶ Here we're redundantly using the `$` operator.

We can take Pure Values and apply them to functions that expect those Pure Values. We can see this when we look at the Type Signature for the `$` operator:

```
apply :: ∀ a b. (a -> b) -> a -> b ❶

infixr 0 apply as $
```

❶ All values are Pure.

Rewriting `apply` with the implied Parentheses:

```
apply :: ∀ a b. (a -> b) -> (a -> b)
```

we can see that this is just the `identity` function specialized for functions, i.e. takes an `a -> b` and returns and `a -> b`.

On the surface, this can seem useless and it is. `apply` is rarely called directly. The reason it exists is so we can have the parenthetical benefits of its operator, `$`.

But in the case of `map`, it's more than a convenience:

```
map :: ∀ a b. (a -> b) -> f a -> f b ❶
```

❶ Remember that this is a Class Method and `f` is defined in the `class` definition. This is why we don't have to put `f` after the `∀`.

Without `map`, we have no way to apply a Value in a Context to a function that takes Pure Values. Imagine we have a Pure Value in a `Maybe`:

```
Just 10
```

Let's also imagine we obtained this value from a function that could potentially have failed. And, when it doesn't fail, we would like to divide the result of that function by 2.

The following will NOT work:

```
div2 :: Int -> Int
div2 = (_ / 2)

x :: Maybe Int
x = div2 $ Just 10 -- COMPILER ERROR
```

The compiler complains that it cannot match `Maybe Int` with `Int`.

This is where `map` from the `Functor` class comes in. With a very small modification to our code, we get the working program:

```
div2 :: Int -> Int
div2 = (_ / 2)

x :: Maybe Int
x = div2 <$> Just 10 ❶
```

❶ Remember that `<$>` is the operator for map.

So `Functors` let us apply Contextual Values, i.e. a Value in a Context, e.g. `Just 10`, to function that expect Pure Values.

We've written a lot of Functional code without ever needing this. You can imagine that everything was going along just fine with simple functions right up until the point where people starting creating these complex contexts. Once that happened, there was a desire to use the myriad of functions they had at their disposal on these new Contextual Values.

The Context is merely what we call a `Functor`. The `map` function allows us to access the encapsulated Value

or Values and manipulate them IN PLACE:

```
map :: ∀ a b. (a -> b) -> f a -> f b
```

And looking at the other side of that same coin, `map` takes functions that take and return Pure Values and it ***lifts*** them into functions that take and return Contextual Values:

```
map :: ∀ a b. (a -> b) -> (f a -> f b) ❶
```

❶ Only difference here is that we added in the implied Parentheses to emphasize the lifting effect.

With this perspective, we see that `map` takes a function expecting Pure Values and turns it into a function that works with Contextual Values.

Chapter 13. Coding Functors

Before we move on to more advanced `Functors`, we're going to first write `Functor` instances for some Types we're already familiar with. Then we'll create our own Types and make them `Functors`.

Return to your project that you've been coding in and create a file called `Ch13.purs` and add the following:

```
module Ch13 where

import Prelude (Unit) ❶

import Effect (Effect)
import Effect.Console (log)

test :: Effect Unit
test = do
  log "placeholder" ❷
```

❶ Just a reminder that we'd normally just include all of `Prelude`, but since we'll be overriding things from `Prelude`, we'll be very specific to only include things we need.

❷ We'll delete this soon enough. It's just here to keep the compiler happy.

Next, change your `Main` module to import `Ch13` and call its `test` function:

```
module Main where

import Prelude
import Effect (Effect)

import Ch13 as Ch13

main :: Effect Unit
main = Ch13.test
```

13.1. Writing Functor Instance for `Maybe`

`Maybe` is one of the first Type that we've learned about. `Maybe` has a Contextual Value in the case of non-failure, i.e. `Just`.

We would like to be able to manipulate this value while leaving the Context in tact and so we're going to do that by making `Maybe` a `Functor`. To do this, we merely need to create a law-abiding Instance of `Functor` for `Maybe`.

Remove the placeholder line from out test code and replace it with the following lines:

```
log $ show $ (_ / 2) <$> Just 10 ❶
log $ show $ (_ / 2) <$> Nothing ❷
```

❶ Prints `(Just 5)`.

❷ Prints `Nothing`.

Also, edit the Prelude import to include a few more of the functions we need to perform these tests:

```
import Prelude (Unit, show, discard, ($), (/), (<$>)) ❶
```

❶ Remember `discard` is for the `do` syntax (more on this when we get to Monads).

Now first write the Type for `Maybe. We've done this many times now and so this should be easy.

Next, write the `Functor` Typeclass definition. Try to do your best without cheating and see how much you've remembered. At this stage, it not uncommon to have to look parts of this up.

And finally, write the `Functor` Instance for `Maybe`. This should also be easy since all you need is the Type definition for `Maybe` and the Typeclass definition for `Functor`.

Give it a try before continuing.

13.2. Code for Functor Instance of `Maybe`

First let's write the `Maybe` Type definition:

```
data Maybe a = Nothing | Just a ❶
```

❶ `Nothing` defined first so that it sorts before `Just`.

Next, let's write `Functor` Typeclass on line at a time:

```
class Functor f where
```

Here `f` is the `Functor` Type. In our case, it'll stand for `Maybe`.

Now let's add the `Functor` Methods:

```
class Functor f where
  map :: ∀ a b. (a -> b) -> f a -> f b ❶
```

❶ Notice how `f` is not part of the `∀`. That's because it's defined already as part of the Typeclass definition which has the implication of `∀ f`.

The `map` function takes a function of Pure Values, `a -> b` and a Contextual Value and it applies that function to that Value and returns the result in the SAME Context. The same context requirement is dictated by the Identity Functor Law.

Finally, we'll write the `Functor` instance for `Maybe`, one line at a time:

```
instance functorMaybe :: Functor Maybe where
```

We've used canonical naming with `functorMaybe`. Also, we didn't specify an `a` after the `Maybe`. Do you remember why?

It's because `f's` Kind is `Type -> Type` and so is `Maybe's`. Since `Maybe` takes a Type and returns a Type, it matches how `f` is used in the Typeclass. So we just replace `f` directly with `Maybe`. HINT: This will not be the case for `Either` whose Kind is `Type -> Type -> Type`.

Next we write the `map` Method, first for the `Nothing` case:

```
instance functorMaybe :: Functor Maybe where
  map _ Nothing = Nothing ❶
```

❶ The first Parameter is unused.

If we have `Nothing`, then we have no Value to apply the function to. The only thing we can do is return `Nothing`.

And finally, we write the `Just` case:

```
instance functorMaybe :: Functor Maybe where
  map _ Nothing   = Nothing
  map f (Just x)  = Just $ f x ❶
```

❶ We unwrap the `Just` to expose its Value, then apply the function to that Value and immediately wrap it back up in the `Just`.

Now, at this point your code has the following compiler error:

```
No type class instance was found for

  Data.Show.Show (Maybe Int)
```

That's because we didn't make Maybe showable. Stop here and create a Show instance for our Maybe. We've done this before and this will be a good review of that process. Try not to cheat and look at what you did before, i.e. unless you've tried and get completely stuck.

The first step to adding a Show instance for Maybe is to derive a Generic instance for Maybe so we can leverage genericShow.

```
import Data.Generic.Rep (class Generic) ❶

data Maybe a = Nothing | Just a
derive instance genericMaybe :: Generic (Maybe a) _ ❷
```

❶ Add import of the Generic Typeclass.

❷ Derive Generic leaving the second Parameter unspecified, i.e. _.

Then we delegate Maybe's show to genericShow:

```
import Prelude (class Show, Unit, show, discard, ($), (/), (<$>)) ❶

import Data.Show.Generic (genericShow) ❷

data Maybe a = Nothing | Just a
derive instance genericMaybe :: Generic (Maybe a) _

instance showMaybe :: Show (Maybe a) where ❸
  show = genericShow
```

❶ Add the Show Typeclass from Prelude.

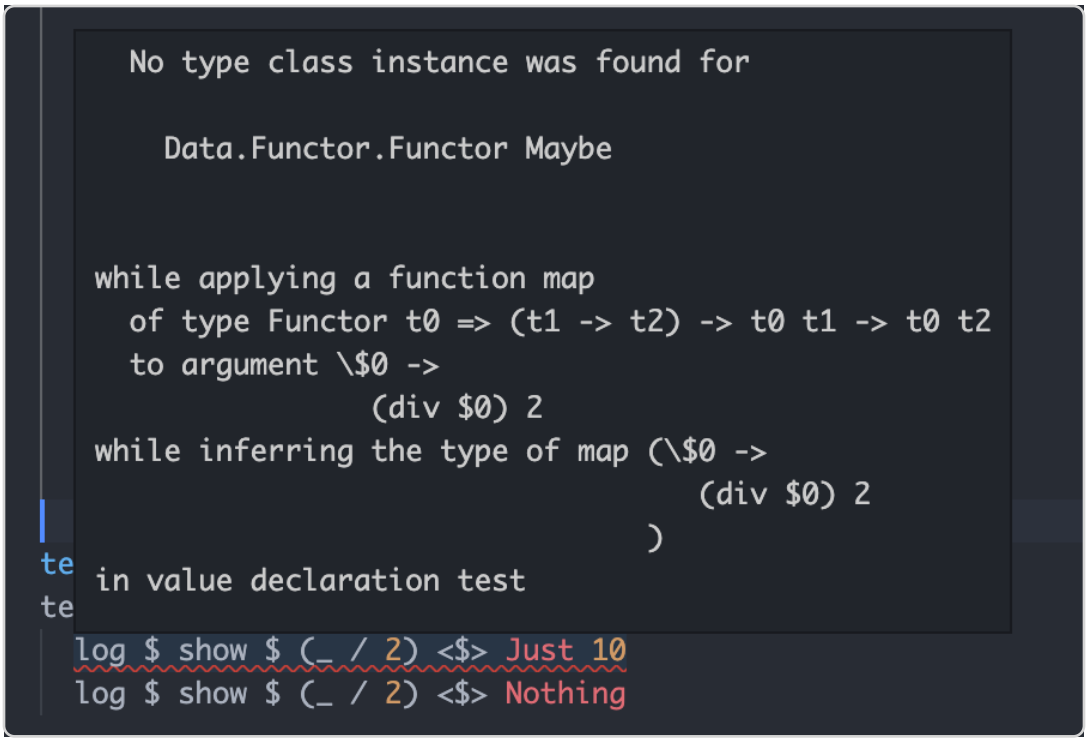
❷ Add the import for genericShow.

❸ Add the Show instance for Maybe delegating to genericShow (since Maybe has a Generic instance).

And now we have a COMPILER ERROR!!

Why?

Let's look at the error carefully:



The salient portion:

```
No type class instance was found for

  Data.Functor.Functor Maybe
```

How can this be possible? We wrote a `Functor` instance for `Maybe`:

```
instance functorMaybe :: Functor Maybe where
  map _ Nothing   = Nothing
  map f (Just x)  = Just $ f x
```

Take a moment to see if you can figure out what we did wrong. HINT: Look very carefully at the error message.

Notice that the error message says we don't have an instance of `Data.Functor.Functor Maybe`. We have our OWN `Functor` class and we made an instance for `Maybe` for OUR Typeclass.

What we didn't do is provide an instance for the `Functor` class that's in the module `Data.Functor`.

Looking at WHERE the error occurred we can see that it's on the line where we call `map`. Actually, we use the `<$>` operator. But the `<$>` operator was imported from Prelude:

```
import Prelude (class Show, Unit, show, discard, ($), (/), (<$>))
```

And that operator doesn't know about our `Functor`. It does, however, know about the `Data.Functor.Functor` Typeclass. In fact, the operator is in the same module.

So we need to write our own `<$>` operator and NOT use the one from Prelude:

```
import Prelude (class Show, Unit, show, discard, ($), (/)) ❶

class Functor f where
  map :: ∀ a b. (a -> b) -> f a -> f b

infixl 4 map as <$> ❷
```

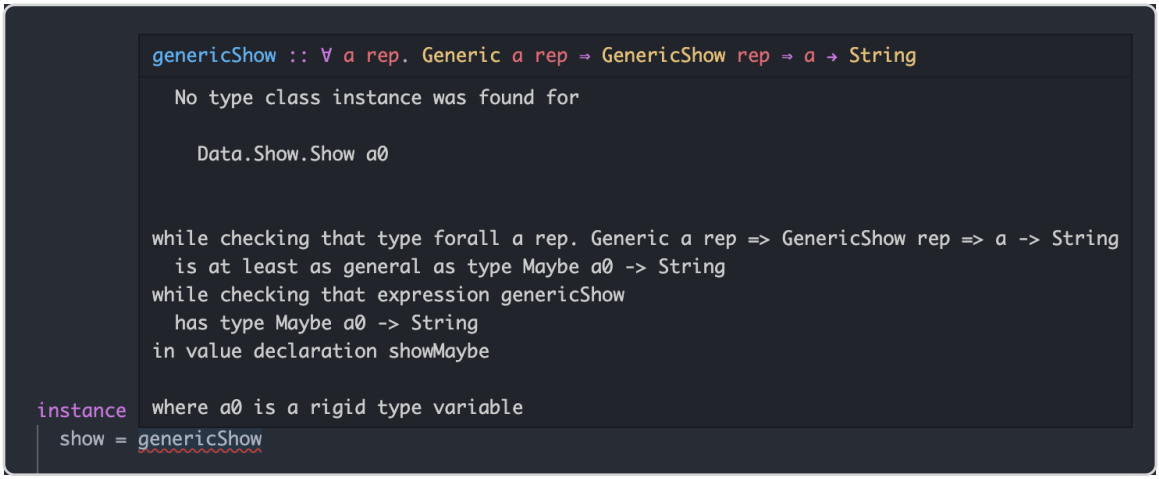
❶ Remove `(<$>)` from the import list.

❷ Add our own operator.

And everything is... NOT GOOD. We still have a COMPILER ERROR!!

Why?

Once again, we look carefully at the error message:



The salient portion:

```
No type class instance was found for

  Data.Show.Show a0
```

Can you see why this is problematic? Let's look at our instance definition to hopefully see why:

```
instance showMaybe :: Show (Maybe a) where
  show = genericShow -- COMPILER ERROR!!
```

Looking at the error message it says that we don't have a `Data.Show.Show` instance for `a0`. Remember the compiler types don't always match our types. But since the only type we have is `a` and the compiler is referring to `a0`, we can safely deduce that they are the same.

The compiler is saying that it cannot call `genericShow` since `a` doesn't have a `Show` instance. That's because `a` can be ANY Type. Not just showable ones.

Let's fix this by assuring the compiler that we'll only accept showable Types for `a`:

```
instance showMaybe :: Show a => Show (Maybe a) where ❶
  show = genericShow
```

❶ Constrain `a` to have a `Show` instance.

And FINALLY, our code compiles. But does it work? I hope so. Run the tests and see.

13.3. Writing Functor Instance for `Either`

Now that we've made it through writing the `Functor` Instance for `Maybe`, writing it for `Either` should be pretty easy. Keep in mind that `Either` has a different Kind Signature than `Maybe`. Review the last chapter on this point if you've forgot this nuance.

And once your memory has been refreshed, add the following lines to your test code and repeat what we just did for `Maybe` but for `Either`:

```
log $ show $ (_ / 2) <$> Right 10 ❶
log $ show $ (_ / 2) <$> Left "error reason" ❷
```

❶ Prints `(Right 5)`.

❷ Prints `(Left "error reason")`.

13.4. Code for Functor Instance of `Either`

First we write the Type definition:

```
data Either a b = Left a | Right b
```

Next, we make Either showable for our tests to compile:

```
data Either a b = Left a | Right b
derive instance genericEither :: Generic (Either a b) _ ❶

instance showEither :: Show a => Show (Either a b) where ❷
  show = genericShow -- COMPILER ERROR!!
```

❶ Derive Generic so we can use genericShow.

❷ Added the Show instance like we did in Maybe.

Essentially, this code just mimics what we did for Maybe. But there's a COMPILER ERROR!! Why?

Here's the salient part of the error:

```
No type class instance was found for

  Data.Show.Show b1
```

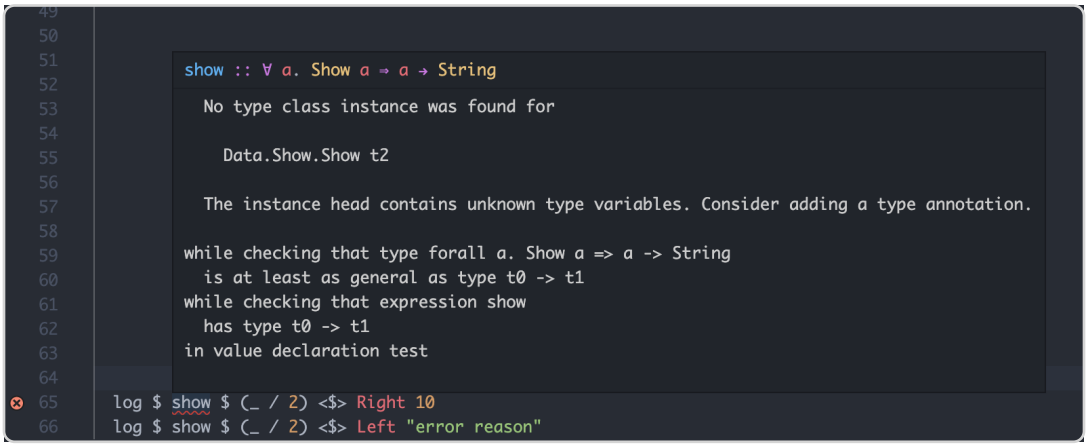
We've seen this before when we forgot the Show a for the Maybe instance of Show. This is telling us that we also need to constrain b:

```
data Either a b = Left a | Right b
derive instance genericEither :: Generic (Either a b) _

instance showEither :: (Show a, Show b) => Show (Either a b) where ❶
  show = genericShow
```

❶ Added Show b constraint.

While that fixes this compiler error, we have a brand new one to contend with in our test code:



Here's the salient part of the error:

```
No type class instance was found for

  Data.Show.Show t2
```

What's `t2` here? Well, since we're trying to show an `Either`, `t2` must be `a` or `b` in our `Either a b`. And since we have a `Right Int` on this line, `b` is an `Int`. But what is `a`?

`a` is an unknown Type. We cannot answer that question anymore than the compiler can. So `t2` must unify with our `a`.

We've seen this problem before. We had to give the compiler an answer to what `a` is but not just any answer. Looking back at our instance definition for showing an `Either`:

```
instance showEither :: (Show a, Show b) => Show (Either a b) where
  show = genericShow
```

We can see that we told the compiler that `a` isn't ANY `a`, but ONLY showable `a's`. Now the compiler is holding us to that promise, but it cannot verify that fact since neither of us know what type `a` is. So it's up to us to give it a Type but not just any old Type.

We have to provide a Type that has a Show instance. We could just decide to say it's an `Int` also. But why not say it's a `List Int` or `Maybe String`. There are a lot of possible Types that will make the compiler happy.

But we want to pick the simplest showable Type here. And if you remember from before, that Type is `Unit`.

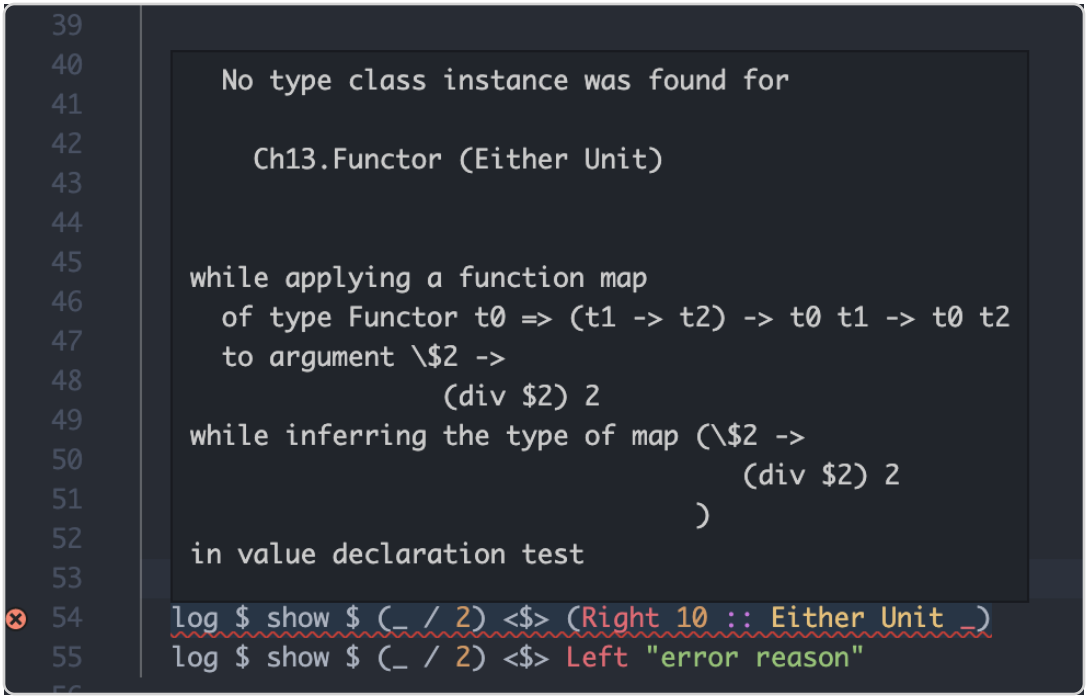
```
log $ show $ (_ / 2) <$> (Right 10 :: Either Unit _) ❶
```

❶ Clearly, the compiler knows what Type the right is, so we can use _ here to let the compiler fill that in for us. All that's critical here is that we provide the left Type.

Notice how we put Parentheses around the `Right 10` part. That's because if we didn't, then we'd be specifying the Type for the whole line of code, which would be `Effect Unit` and that doesn't help the compiler out at all.

But now the same line has a different COMPILER ERROR!! Why?

Here's that error:



The salient portion is:

```
No type class instance was found for

  Ch13.Functor (Either Unit)
```

This is saying that our `Either` doesn't have a `Functor` instance. So let's write one.

We've already written the `Functor` Typeclass when we were working on `Maybe`, so we can skip that step and write the `Functor` instance for `Either` one line at a time:

```
instance functorEither :: Functor Either where
```

Next, the `Left` case:

```
instance functorEither :: Functor Either where
  map _ (Left err) = Left err ❶
```

❶ This seems like a waste of effort. We're unwrapping the error, doing nothing with it and then rewrapping it.

Let's return to this minor issue AFTER we write the `Right` case:

```
-- COMPILER ERROR!!
instance functorEither :: Functor Either where
  map _ (Left err) = Left err
  map f (Right x) = Right $ f x ❶
```

❶ This is exactly like we did with Just, we apply the function to the contained Value.

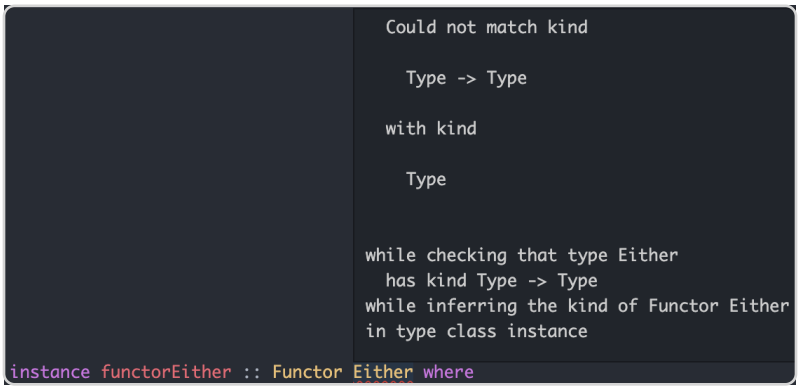
Now, we can simplify the process for Left, by doing it last and not wasting time wrapping and unwrapping it:

```
-- COMPILER ERROR!!
instance functorEither :: Functor Either where
  map f (Right x) = Right $ f x
  map _ left = left ❶
```

❶ Return the Left value unchanged.

Here we have to check Right first so we KNOW that the next case must be Left. The second case is the catch-all case since it's has NO Patterns to match, i.e. none of the Parameters involve Data Constructors.

Now we have an compiler error to contend with:



Can you see the problem? Stop and try to fix this before reading on.

Okay, so you either figured it out or gave up. The important part is that you spent time thinking about it before reading the solution.

The problem is that we simply just parroted what we did in the Maybe instance definition. So if we ask ourselves what's the difference between the Type Parameters for Maybe and Either, we should see immediately that Either has 1 more Type Parameter than Maybe.

Maybe's Type Signature matches f's Type Signature in the Functor Typeclass definition:

```
class Functor f where ❶
  map :: ∀ a b. (a -> b) -> f a -> f b
```

❶ `f's` Kind Signature is `Type -> Type`.

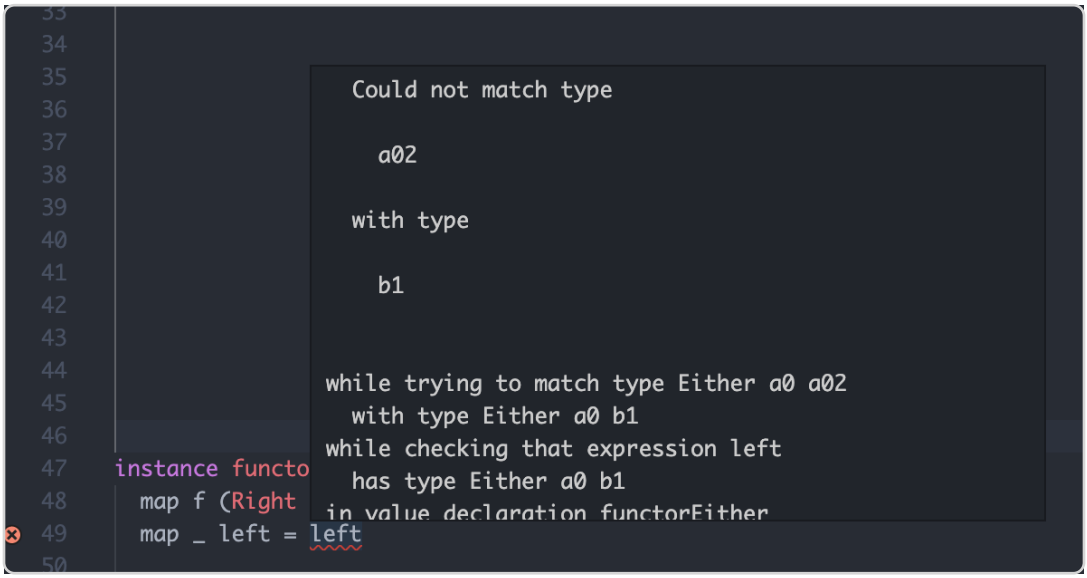
Remember, `Either's` Kind Signature is `Type -> Type -> Type`. It has 1 more than `f's`.

To make `Either` match `f's` Kind Signature, we must give `Either` a Type Parameter:

```
instance functorEither :: Functor (Either a) where ❶
  map f (Right x) = Right $ f x
  map _ left = left
```

❶ Partially apply Type Parameters to the Type Constructor so its Kind matches.

And now that we fixed that error, we are presented with a brand new one:



And I usually don't dig in too deep into these error messages since they rarely require it. Usually, you can just look at the top part and your code to figure out what's wrong.

This error, however, is different.

It requires a little more digging. So let's do that. Here's the top part of the error that's usually sufficient:

```
Could not match type

  a02

with type

  b1
```

But this is too vague. So looking to the next portion we find:

```
while trying to match type Either a0 a02
  with type Either a0 b1
```

And now I can see that the a02 and b1 are the Types regarding the Right in our Either. Notice that the Left Type Parameter, a0, matches fine.

Looking at the offending line doesn't offer much help:

```
  map _ left = left
               ~~~~
```

But the underlines give us a clue, but not quite enough yet. Let's dig deeper into the next portion of the error message:

```
while checking that expression left
  has type Either a0 b1
```

This tells us more. It's underlining the right-hand left (wow that's not confusing) and it's trying to make sure that it's of the Type Either a0 b1, but from the previous part of the error, it found that it's of Type Either a0 a02.

Okay, so this is quite the conundrum. Let's try an experiment. Let's not be so clever and let's write the map function for the Left analogous to how we did it with Right:

```
instance functorEither :: Functor (Either a) where
  map f (Right x)   = Right $ f x
  map _ (Left err)  = Left err ❶
```

❶ This is analogous to how we coded the Right case.

This compiles fine. Okay, so we fixed it. But wait a minute. What was wrong with our initial approach? Many times when we're coding at work, we won't be able to take the time to dig into why. But this is a book and we're learning here, so let's take the time.

When I first encountered this error, I got this far and was stumped. The original code made sense to me. I couldn't understand why the compiler was confused. So I reached out to the PureScript community in the Functional Programming organization. There are channels for lots of languages including Elm and Haskell there.

The community is great and very helpful and I was able to get help immediately regarding this issue. There are 2 channels that I frequent, `purescript-beginners` and `purescript`. If you're asking a beginner type question, I'd suggest asking on `purescript-beginners`. The people who answer it will expect that you're starting out in PureScript and, hopefully, answer accordingly.

For more advanced questions or if you just want a quick to the point answer without much preamble, then I'd suggest the `purescript` channel. This is where I posed my problem and was promptly answered by `natefaubion`, `kritzcreek` and `hdgarrood`. All gave great responses, but `kritzcreek` said something that just clicked and the problem became obvious.

Let's look back at the two different ways of coding the `Left` case:

```
  map _ left = left -- COMPILER ERROR!!
  map _ (Left err)  = Left err
```

Now, let's ask ourselves questions about the different types. First, let's start with the one that broke the compilation.

What type is `left`? Can you answer this? If not, let's look at the Type Signature for `map` to help, but let's specialize it for `Either`:

```
map :: ∀ b c. (b -> c) -> Either a b -> Either a c ❶
```

❶ I renamed some of the Type Parameters from the traditional definition to avoid confusion.

Now, we can see that `left` is Type `Either a b` because it's on the left-hand side of the equal sign an it's the second Parameter.

But wait. It's also on the right-hand side. But the `map` function expects the right-hand side to have a DIFFERENT Type, viz. `Either a c`. And that's our problem!

`left` cannot have both Types. And the mystery is solved. Right?

Well, no. That doesn't explain why the other approach works. It only explains why our original approach failed. Let's look at the working code.

What type is the `Left err` that's on the left-hand side of the equal sign?

The left-hand side `Left err` has Type `Either a b`.

What about the right-hand side's Type? It's `Either a c`. But how do we know that. Well, we know that because it compiles, but it's only clear that it's `Either a c` because it compiles. We could've argued that it's the same just like the code that doesn't compile.

Let's look at the Data Constructor, `Left` and it's Type Signature to see if we can figure out why:

```
Left :: ∀ a b. a -> Either a b
```

Notice the `b`, the second Type Parameter. It can be ANYTHING. `Left` doesn't care. It only cares about its Type, viz. `a`.

And this is what `kritzcreek` said to me that made it click:

```
...when you call the Left constructor you get to pick a new type for the
second type variable of Either
```

The second Type Parameter on the left-hand side of the equal sign has Type `Either a b` but when we extract `err`, all we have is something of Type `a`. It's no longer part of an `Either'.

When we wrap it back up in `Left`, the second Type Parameter can be ANYTHING. So the right-hand side of the equal sign can be `Either a c` which matches the return Type for `map`.

When we incorrectly used `left` on the left-hand side, it brought along with it 2 Types, `a` AND `b`. That's because `left` is an `Either a b`. When we tried to use it on the right-hand side, it still had those 2 Types. But `b` doesn't match with `c`. `map` returns a Value of Type `Either a c` NOT `Either a b`. This is why the compiler complained.

I know this is a bit of a mind bender. Hopefully, you've gained a better understanding and I'd suggest revisiting this at a later date if you didn't fully understand everything. Chances are it'll make more sense the second time through. Maybe even reread it now.

Go ahead, I'll wait.

Okay, now that you've reread and hopefully fully understood where we went wrong (or more accurately where I led you astray), we can fix our `Functor` instance for `Either`:

```
instance functorEither :: Functor (Either a) where
  map _ (Left err)  = Left err ❶
  map f (Right x)   = Right $ f x
```

❶ For some reason, I like `Left` before `Right`.

And FINALLY our code compiles! Go ahead and run the tests and see if it works. Spoiler, it does.

13.5. Writing Functor Instance for `Tuple`

Writing the `Functor` instance for `Tuple` should be very easy now that we've got `Either` working and that's because both `Tuple` and `Either` have 2 Type Parameters.

Add the following to your test code:

```
log $ show $ (_ / 2) <$> Tuple 10 20 ❶
```

❶ Prints `(Tuple 10 10)`.

Go ahead and give it a try before you continue reading.

13.6. Code for Functor Instance of `Tuple`

First we code the Type definition:

```
data Tuple a b = Tuple a b
```

Next, we make it showable for our test code:

```
data Tuple a b = Tuple a b
derive instance genericTuple :: Generic (Tuple a b) _ ❶

instance showTuple :: (Show a, Show b) => Show (Tuple a b) where ❷
  show = genericShow
```

❶ Derived `Generic` instance so we can use `genericShow`.

❷ Leverage `genericShow`.

And finally, we add the `Functor` instance:

```
instance functorTuple :: Functor (Tuple a) where ❶
  map f (Tuple x y) = Tuple x $ f y ❷
```

❶ Just like `Either` we fix the `a` Type Parameter, which means we can only `map` over the second value of the `Tuple`. We'll need a `Bifunctor` instance to map over the first.

❷ Here we only map over the y.

And that's it. Run it and see if it works.

13.7. Writing Functor Instance for Threeple

Threeple? What's that?

It's a name I made up for a Tuple that has 3 Type Parameters:

```
data Threeple a b c = Threeple a b c
```

Add this definition and the following line to your test code:

```
log $ show $ (_ / 2) <$> Threeple 10 20 40 ❶
```

❶ Prints (Threeple 10 20 20).

So go ahead and write this. It'll be just as fast as Tuple was with the only wrinkle being that you have 3 Type Parameters to deal with.

Don't read ahead until you've given it a try.

13.8. Writing Functor Instance for Threeple

First we wrote the definition:

```
data Threeple a b c = Threeple a b c
```

Then we write the Show Instance so our test code will work:

```
data Threeple a b c = Threeple a b c
derive instance genericThreeple :: Generic (Threeple a b c) _ ❶

instance showThreeple :: (Show a, Show b, Show c) => Show (Threeple a b c)
where ❶
  show = genericShow
```

❶ We had to add one more Type Parameter c in 2 places.

And finally, we write the Functor Instance:

```
instance functorThreeple :: Functor (Threeple a b) where ❶
  map f (Threeple x y z) = Threeple x y $ f z ❷
```

❶ We had to add the first 2 Type Parameters so that the Kinds would match. This means that the only thing we can map is the `c` Type Parameter.

❷ Here we only map our `z` Parameter which is of Type `c`.

13.9. Validating the Functor Laws

We've been coding `Functors` without validating the laws. There are 2 different ways we could do this. First, is to write test code to verify them. The second approach is Algebraic Substitution. For `Functors`, we're going to write code. Later, when we get to `Bifunctor`, we'll use Algebraic Substitution.

The first way is easy. We're just going write test code for them. Here's the test code for `Maybe`:

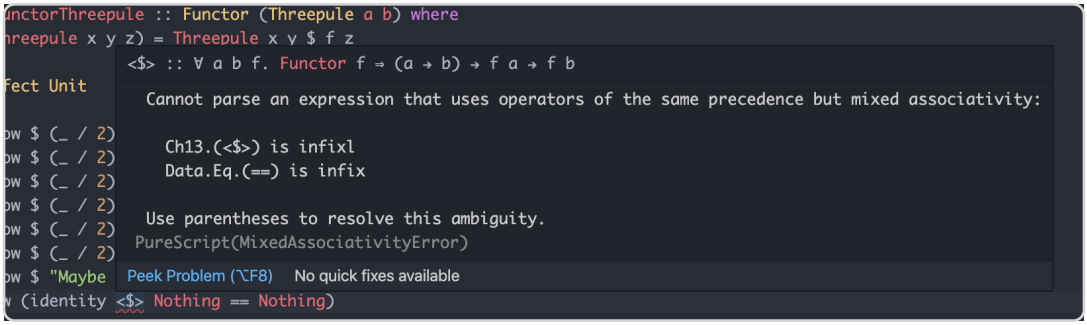
```
import Prelude (class Show, Unit, show, discard, identity, ($), (/), (<>), (
==)) ❶

log $ show $ "Maybe Identity for Nothing: "
  <> show (identity <$> Nothing == Nothing) -- COMPILER ERROR!! ❷
```

❶ Added `identity`, `<>` and `==` from Prelude.

❷ Prints `Maybe Identity for Nothing: true`.

This is testing to see if the Identity Law holds for `Nothing`, but we're getting the following compiler error:



The error message tells us everything. We have used `<$>` and `==` on the same line of code and they both have the same precedence with different associativity (`<$>` is Left-Associative and `==` is Right). So we need to resolve the ambiguity manually by adding Parentheses:

```
log $ show $ "Maybe Identity for Nothing: "
  <> show ((identity <$> Nothing) == Nothing) -- NEW COMPILER ERROR!!
```

And now a new compiler error that tells us that we cannot compare `Maybe` for equality. But that's easy enough to fix and I'll let you fix that one.

Go ahead and make it so `Maybes` can be compared for equality and run this test before proceeding.

First step is to add `Eq` Instance for `Maybe`:

```
import Data.Eq (class Eq) ❶

derive instance eqMaybe :: Eq (Maybe a) -- COMPILER ERROR!!
```

❶ We could've also imported this from Prelude.

The compiler error should look familiar. See if you can solve it before reading on.

Once again, if we want to compare `Maybe a` for equality, we have to be able to compare `a` for equality, which means we need an `Eq` Constraint:

```
import Data.Eq (class Eq)

derive instance eqMaybe :: Eq a => Eq (Maybe a)
```

But now we STILL have trouble:

```
log $ show $ "Maybe Identity for Nothing: "
  <> show ((identity <$> Nothing) == Nothing) -- NEWER COMPILER ERROR!!
```

This time the error is telling:

```
No type class instance was found for

  Data.Eq.Eq t1

The instance head contains unknown type variables. Consider adding a type
annotation
```

And we've seen this before. The compiler cannot figure out if `a` has an `Eq` Instance since it cannot figure out what `a` is. So, like so many times before, we're going to give it an `a`, viz. `Unit`, the simplest Type:

```
log $ show $ "Maybe Identity for Nothing: "
  <> show ((identity <$> Nothing) == (Nothing :: Maybe Unit)) ❶ ❷
```

❶ Important to Parenthesize this otherwise the compiler thinks we're defining the Type for the whole equality test and not just `Nothing's` Type.

❷ We could have just as easily put this Type Annotation on the first `Nothing` but that would have cluttered the code up.

Now run your tests and they should work.

Go ahead and add a test for `Just 10` or `Just "abc"` or some other `Just a` before moving on.

Here's my `Just a` test:

```
log $ show $ "Maybe Identity for Just: "
  <> show ((identity <$> Just [1, 2]) == Just [1, 2]) ❶
```

❶ Prints `Maybe Identity for Just: true`.

Next, write the code to validate the Composition Law before reading on. Here's the Functor Laws to refresh your memory:

```
map identity = identity            [Identity]
map (g <<< f) = map g <<< map f    [Composition]
```

Now give it a try before seeing my solution below.

Here's an example of how you can write a validation for the Composition Law. Your code may vary:

```
import Prelude (class Show, Unit, show, discard, identity, ($), (/), (<>), (
==), (<<<), (*)) ❶

let g x = x * 2 ❷
    f x = x * 3
log $ show $ "Maybe Composition for Nothing: "
  <> show ((map (g <<< f) Nothing) == (map f <<< map g) Nothing) ❸ ❹
log $ show $ "Maybe Composition for Just: "
  <> show ((map (g <<< f) (Just 60)) == (map f <<< map g) (Just 60)) ❺
```

❶ Had to import `<<<` and `*` operators from Prelude.

❷ Define local Functions to `test` that multiplies `Ints` by a constant.

❸ We didn't need to explicitly annotate the Type of `a` in the `Maybe a` here since the compiler could infer the Type since both `f` and `g` work on `Ints`. Therefore, `a` is `Int`.

❹ Prints `Maybe Composition for Nothing: true`.

❺ Prints `Maybe Composition for Just: true`.

When we run our test code, we can see that they all print true. This means that our implementation for Maybe is a law-abiding Functor.

This can be done for all other types that we've created Functors for. We won't go through this step by step, because it's nearly identical to what we've done for Maybe. In fact, here's the tests for Tuple:

```
log $ show $ "Tuple Identity: "
  <> show ((identity <$> Tuple 10 20) == Tuple 10 20) ❶
log $ show $ "Tuple Composition : "
  <> show ((map (g <<< f) (Tuple 10 20)) == (map f <<< map g) (Tuple 10 20))
❷
```

❶ Prints "Tuple Identity: true".

❷ Prints "Tuple Composition : true".

One thing that was needed by above code was an Eq instance for Tuple. Here's how you can derive it:

```
derive instance eqTuple :: (Eq a, Eq b) => Eq (Tuple a b)
```

Notice I had to constrain both a and b to have Eq instances too.

Run the test and verify it works.

13.10. Writing Bifunctor Instance for Either

We cannot write a Bifunctor Instance for Maybe since it only has 1 Type Parameter a:

```
data Maybe a = Nothing | Just a
```

But Either has 2:

```
data Either a b = Left a | Right b
```

So we can write an instance for Either.

But first add the following lines of code to you test code:

```
import Data.String.Common (toUpper) ❶

log $ show $ rmap (_ * 2) $ Left "error reason" ❷
log $ show $ rmap (_ * 2) $ (Right 10 :: Either Unit _) ❸ ❹
log $ show $ lmap toUpper $ (Left "error reason" :: Either _ Unit) ❺ ❻
log $ show $ lmap toUpper $ Right 10 ❼
```

❶ Import `toUpper` from `Data.String.Common`. This was determined by searching Pursuit.

❷ Prints `(Left "error resason")`.

❸ Prints `(Right 20)`.

❹ We need to add a Type annotation here. More on this below.

❺ Prints `(Left "ERROR REASON")`.

❻ We need to add a Type annotation here. More on this below.

❼ Prints `(Right 10)`.

In the above test code, we had to specify Type annotations for `Right` on `rmap`, but not `Left`. And on `Left` for `lmap`, but not `Right`. It's easy to see that the right map, `rmap`, wants a Type annotation for `Right` and that `lmap` wants it for `Left`.

But it's not clear as to why this is the case and why it wasn't needed for `rmap` and `Left` or for `lmap` and `Right`. To get a better idea for why this is, search `Pursuit` to determine what the Type Signatures are for `rmap` and `lmap` before moving on and see if you can figure out why this is so.

Please take the time to think about this before reading on.

Here are the Type Signatures for `rmap` and `lmap`:

```
rmap :: ∀ f a b c. Bifunctor f => (b -> c) -> f a b -> f a c ❶
lmap :: ∀ f a b c. Bifunctor f => (a -> b) -> f a c -> f b c ❷
```

❶ `rmap` morphs the second Type Parameter.

❷ `lmap` morphs the first Type Parameter.

Let's specialize these Type Signatures for `Either a b`:

```
rmap :: ∀ a b c. (b -> c) -> Either a b -> Either a c
lmap :: ∀ a b c. (a -> b) -> Either a c -> Either b c
```

Let's start with `rmap`:

```
log $ show $ rmap (_ * 2) $ Left "error reason"
```

`rmap (_ * 2) $ Left "error reason"` has Type `Either a b`. The compiler needs to figure out if `a` and `b` are both showable before it can safely call `show` on it.

Since `rmap` takes a function that morphs the second Type Parameter, the compiler can infer `b` from that function. In this case, `b` unifies with `Int` since `(_ * 2)` return an `Int`.

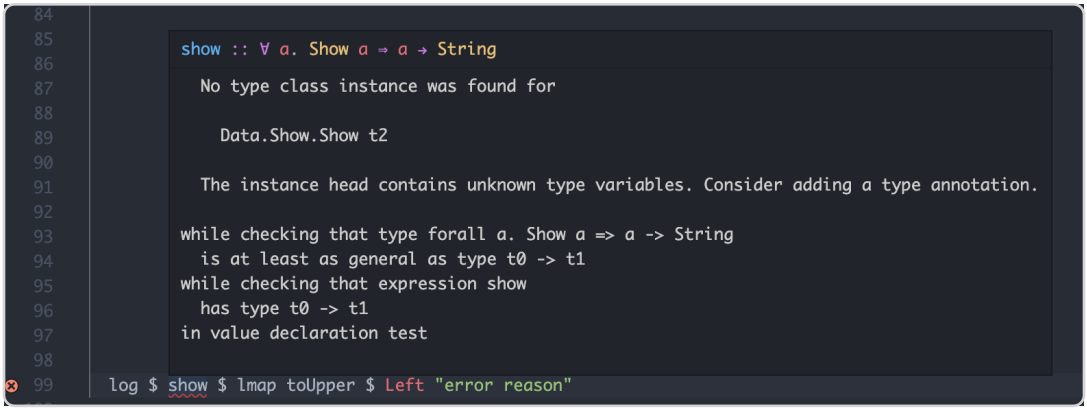
We've also given `rmap` a Value which tells the compiler what the first Type Parameter is. `Left "error reason"` has Type `Either String b`.

Therefore, the compiler can infer that `rmap (_ * 2) $ Left "error reason"` has Type `Either String Int`. And since both `String` and `Int` have `Show` Instances, this line of code compiles.

But when `lmap` is called on `Left "error message"`:

```
log $ show $ lmap toUpper $ Left "error reason" -- COMPILER ERROR!!
```

the compiler can only infer that `a` is a `String`, but there's no function passed to `lmap` that involves the second Type Parameter. So the compiler cannot verify the `Show` constraint on `b`. The compiler error here says exactly that:



The salient portion:

```
No type class instance was found for

  Data.Show.Show t2

The instance head contains unknown type variables. Consider adding a type
annotation.
```

So we are forced to provide an explicit Type Signature to tell it what Type `b` is. We did this by specifying

`Either _ Unit`.

The same argument can be made for `rmap` with `Right`. The compiler comes up short and we're forced to define `a` and we specified `Either Unit _`.

In both, cases we simply used _ in place of the Type that the compiler could figure out on its own. This isn't just for typing efficiency. If we make a change in our program in the future and the Type that the compiler can infer changes, then the compiler will fill in the _ with this new Type.

Had we explicitly put in `String` for example, and in the future the Type changes to `Int`, then our explicit Type annotation will also need to be edited so as not to cause a compiler error. This is why it's best to not specify a Type unnecessarily.

With all of the done, you're ready to write the `Bifunctor` Typeclass definition and then the `Bifunctor` Instance for `Either`. Proceed after you've given it a try.

13.11. Code for Bifunctor Instance for `Either`

First, we write the Typeclass definition for `Bifunctor` step by step:

```
class Bifunctor f where ❶
```

❶ We need to think about the Type Parameter `f`'s Kind Signature here.

The Kind Signature for `f` is different than it was for `Functor`. The whole point of `Bifunctor` is to allow us to map over 2 Type Parameters. This means that `f` has the Kind `Type -> Type -> Type`. We should take note that this is the same Kind Signature for `Either`.

Next, we write the Typeclass Method Type Signatures:

```
class Bifunctor f where
  bimap :: ∀ a b c d. (a -> c) -> (b -> d) -> f a b -> f c d
```

We need to write the helper functions, `rmap` and `lmap` since we used them in our test code. First, we'll write `rmap`'s Type Signature:

```
rmap :: ∀ f a b c. Bifunctor f => (b -> c) -> f a b -> f a c ❶ ❷
```

❶ `f` is constrained to have a `Bifunctor` instance.

❷ Notice the function `b -> c` maps over the RIGHT Type Parameter and takes a `f a b` and turns it into a `f a c`.

Now we can write `rmap`'s implementation, but since we constrained `f` to be a `Bifunctor`, we can leverage

bimap:

```
rmap :: ∀ f a b c. Bifunctor f => (b -> c) -> f a b -> f a c
rmap = bimap identity ❶
```

❶ Written in Point-free notation where the function is unnamed.

Now we can write lmap's Type Signature:

```
lmap :: ∀ f a b c. Bifunctor f => (a -> c) -> f a b -> f c b ❶
```

❶ Notice the function a -> c maps over the LEFT Type Parameter and takes a f a b and turns it into a f c b.

And finally, we write the implementation for lmap leveraging bimap:

```
lmap :: ∀ f a b c. Bifunctor f => (a -> c) -> f a b -> f c b
lmap f = bimap f identity ❶
```

❶ This could've been written Point-free by using flip, but in this case, it would make the code more verbose. For the curious, the Point-free version is: lmap = flip bimap identity.

Now we're ready to start writing the instance for Either. We'll start by writing its Type definition:

```
data Either a b = Left a | Right b ❶
```

❶ Either's Kind is Type -> Type -> Type

Next, let's write the instance definition:

```
instance bifunctorEither :: Bifunctor Either where ❶
```

❶ Either's Kind Signature matches f's in the Bifunctor Typeclass, so we can just specify it without any Type Parameters.

And now, let's add the implementation for the Left case:

```
instance bifunctorEither :: Bifunctor Either where
  bimap f _ (Left err)  = Left $ f err ❶
```

❶ In this case, we don't care about the RIGHT mapping function. This is because Either is a Sum Type (Coproduct). We will care when we write bimap for Product Types.

And finally, we code the Right case:

```
instance biFunctorEither :: Bifunctor Either where
  bimap f _ (Left err)  = Left $ f err
  bimap _ g (Right x)   = Right $ g x ❶
```

❶ In this case, we don't care about the LEFT mapping function.

13.12. Writing Bifunctor Instance for Tuple

Unlike Either, Tuple is a Product Type, i.e. both of its Type Parameters are present at the same time. So keep this in mind while coding its Bifunctor Instance.

Here's the test code that you should add:

```
log $ show $ rmap (_ * 2) $ Tuple 80 40 ❶
log $ show $ lmap (_ / 2) $ Tuple 80 40 ❷
log $ show $ bimap (_ / 2) (_ * 2) $ Tuple 80 40 ❸
```

❶ Prints (Tuple 80 80).

❷ Prints (Tuple 40 40).

❸ Prints (Tuple 40 80).

Notice that we didn't run into any of the problems that we did with our Either test code. That's because Tuple is a Product Type and both of its Type Parameters are present in all of its Inhabitants. The compiler can know full well that a and b are both Ints.

Now based on what we've done earlier, writing Bifunctor for Tuple should go pretty smoothly. After you've written it, continue on.

13.13. Code for Bifunctor Instance for Tuple

First we write the instance definition:

```
instance biFunctorTuple :: Bifunctor Tuple where
```

And finally, we write the bimap Method:

```
instance bifunctorTuple :: Bifunctor Tuple where
  bimap f g (Tuple x y) = Tuple (f x) (g y) ❶
```

❶ Here we need to apply BOTH the RIGHT and LEFT mapping functions.

13.14. Writing Bifunctor Instance for Threeple

Our made up triplet, Threeple, has 3 Type Parameters, hence its name. So keep the Kind Signatures in mind when writing the Bifunctor instance. Think back to how we wrote the Functor Instance for Either.

Add the following test code:

```
log $ show $ rmap (_ * 2) $ Threeple 99 80 40 ❶
log $ show $ lmap (_ / 2) $ Threeple 99 80 40 ❷
log $ show $ bimap (_ / 2) (_ * 2) $ Threeple 99 80 40 ❸
```

❶ Prints (Threeple 99 80 80).

❷ Prints (Threeple 99 40 40).

❸ Prints (Threeple 99 40 80).

Give it a try before looking at the answer.

13.15. Code for Bifunctor Instance for Threeple

Let's refresh our memories regarding Threeple's definition:

```
data Threeple a b c = Threeple a b c
```

The Kind Signature for the Type Threeple is Type -> Type -> Type -> Type. That's one more Type than the f in the Bifunctor Typeclass definition.

So we're going to have to keep the first Type Parameter a constant when we write the instance definition:

```
instance bifunctorThreeple :: Bifunctor (Threeple a) where
```

And now we can write the bimap Method:

```
instance bifunctorThreeple :: Bifunctor (Threeple a) where
  bimap f g (Threeple x y z) = Threeple x (f y) (g z) ❶
```

❶ We leave the first Parameter to the Data Constructor alone, viz. x. This should make sense since we kept the Type Parameter a constant and x is of Type a.

13.16. Validating the Bifunctor Laws

Now, we're ready to validate the `Bifunctor` laws. When we did this for `Functor`, we used code. This time we're going to use Algebraic Substitution.

Let's review the laws:

```
bimap identity identity = identity        [Identity]
bimap (g1 <<< f1) (g2 <<< f2) =
           bimap g1 g2 <<< bimap f1 f2    [Composition]
```

Put into words, the `Identity` law states that if you `bimap` using `identity` on both the right and the left, it has the same affect as just using the `identity` function. And the `Composition` law states that composing 2 functions on both sides is equivalent to `bimapping` with one function set and then composing with `bimapping` with the other.

Let's validate that our `Tuple` implementation is a law-abiding `Bifunctor`:

```
instance bifunctorTuple :: Bifunctor Tuple where
  bimap f g (Tuple x y) = Tuple (f x) (g y) ❶
```

We're going to substitute starting on the left-hand side and then the right until the two sides are equal, which proves that we're law-abiding.

Let's start with the `Identity` law:

```
bimap identity identity (Tuple x y)
                        = identity (Tuple x y) -- Given
Tuple (identity x) (identity y)
                        = identity (Tuple x y) -- Subst into ❶
Tuple x y = identity (Tuple x y)               -- Function application
Tuple x y = Tuple x y                          -- Function application
```

Next the `Composition` law:

```
bimap (g1 <<< f1) (g2 <<< f2) (Tuple x y)
      = bimap g1 g2 <<< bimap f1 f2 (Tuple x y) -- Given
Tuple (g1 <<< f1 $ x) (g2 <<< f2 $ y)
      = bimap g1 g2 <<< bimap f1 f2 (Tuple x y) -- Subst into ❶
Tuple (g1 (f1 x)) (g2 (f2 y))
      = bimap g1 g2 <<< bimap f1 f2 (Tuple x y) -- Compose equivalence
Tuple (g1 (f1 x)) (g2 (f2 y))
        = bimap g1 g2 (bimap f1 f2 (Tuple x y)) -- Compose equivalence
Tuple (g1 (f1 x)) (g2 (f2 y))
            = bimap g1 g2 (Tuple (f1 x) (f2 y)) -- Subst into ❶
Tuple (g1 (f1 x)) (g2 (f2 y))
               = Tuple (g1 (f1 x)) (g2 (f2 y)) -- Subst into ❶
```

Chapter 14. More Functors

So far, we've only considered `Functors` that contained Values. But technically, a Function is a Value. So can a `Functor` contain a Function?

Of course, it can and these sorts of `Functors` are some of the most powerful and useful `Functors` we will use. But before we delve into `Functors` of `Functions`, let's do a quick recap of `Functors` of Values.

14.1. Functors of Values

Here are some generic Product Types with 1 and 2 Parameters:

```
newtype PV1 a = PV1 a
data PV2 a b = PV2 a b
```

Writing their `Functor` Instances is pretty mechanical:

```
instance functorPV1 :: Functor PV1 where
  map f (PV1 x) = PV1 (f x)

instance functorPV2 :: Functor (PV2 a) where
  map f (PV2 x y) = PV2 x (f y)
```

Notice how we use the function, `f`. We immediately apply it to the Values to produce new Values within the Context. This is an important distinction that will be more apparent when those Values are Functions.

Here are some generic Sum Types with 1 or 2 Parameters:

```
data SV1 a = Nought | SV1 a
data SV2 a b = Nada | SV2A a | SV2B b
```

And writing the `Functor` Instances for these is equally mechanical:

```
instance functorSV1 :: Functor SV1 where
  map _ Nought  = Nought
  map f (SV1 x) = SV1 (f x)

instance functorSV2 :: Functor (SV2 a) where
  map _ Nada      = Nada
  map _ (SV2A x)  = SV2A x
  map f (SV2B y)  = SV2B (f y)
```

Once again, we apply the function `f` immediately to the Values.

14.2. Functors of Functions

But what happens when we have the following Type:

```
newtype F1 a = F1 (Int -> a)
```

How do we write the `Functor` Instance for `F1`? Let's give it a try:

```
instance functorF1 :: Functor F1 where
  map f (F1 g) = F1 (???) ❶
```

❶ We need to combine `f` and `g` somehow.

How do we combine functions? Well, usually through Composition, but it really depends. We also have to make sure that Types match. So, let's look at the Types in detail.

First, our mapping function's Type Signature:

```
f :: a -> b
```

Nothing special here. This is the standard Type Signature for any `Functor's` mapping function.

Next, let's add `F1` Function's Type, `g`:

```
f :: a -> b
g :: Int -> a
```

And finally, let's add the composition of `f` and `g`:

```
f :: a -> b
g :: Int -> a
g <<< f --- COMPILER ERROR!! ❶
f <<< g :: Int -> b ❷
```

❶ Cannot compose this way since the Types don't match.

❷ This is a clue that composition will probably give us what we want since `Int -> b` is the resulting Type Signature that we're expecting after we call `map` on `F1`.

Looks like `f <<< g` is our answer since it gives us the correct Type Signature. Just for fun, let's return to our attempt to the write `F1's Functor` and look at this from another perspective:

```
instance functorF1 :: Functor F1 where
  map f (F1 g) = F1 (???)
```

In place of the `???`, we need a function that takes `Int`s and returns `b`s. If we start with `g` which takes `Int`s and returns `a`'s, we can use `f` to convert those `a`'s into a `b`'s, which is exactly the return Type we're looking for.

This also shows us that we can compose the two functions using `f` to convert the output of `g` to get the desired Type:

```
instance functorF1 :: Functor F1 where
  map f (F1 g) = F1 (f <<< g)
```

Notice that when we map over Types that have Functions instead of Values, we don't apply the Function to the Value. Instead, we compose the Functions. This is an important distinction that we should keep in mind moving forward.

Let's try another:

```
newtype C1 a = C1 (a -> Int)
```

This is pretty similar to our `F1` with the only difference being that `C1` contains a Function whose Type Signature is backwards from `F1`.

But no matter, we should write the `Functor` next:

```
instance functorC1 :: Functor C1 where
  map f (C1 g) = C1 (f <<< g) -- COMPILER ERROR!!
```

Looks like our Types don't match up. Let's look at them:

```
f :: (a -> b)
g :: (a -> Int)
f <<< g -- Type mismatch
g <<< f -- Type mismatch
```

`f <<< g` isn't valid since `g's` output is `Int`, which is NOT `f's` input. And in the same vein, `g <<< f` is not valid since `f's` output is `b`, which is not `g's` input.

So what gives? How do we combine `f` and `g` so that we can match their Types?

Let's look more closely at the differences between `F1` and `C1` to see if we can glean any insights to help us solve this dilemma:

```
newtype F1 a = F1 (Int -> a)
newtype C1 a = C1 (a -> Int)
```

The main difference here is that `F1` produces `a's` whereas `C1` consumes `a's`.

The reason we could easily write a `Functor` instance for `F1` was because we could convert the Functor's output, `a` to `b` by using `f`, which takes `a's` and turns them into `b's`.

When we `map` over `F1 a` we're turning it into an `F1 b`, which means the underlying function will now PRODUCE `b's`.

But when we `map` over `C1 a`, we want to turn it into `C1 b`, which means that the underlying function will now CONSUME `b's`.

When we have a Function that PRODUCES an `a` and we want it to PRODUCE a `b`, all we have to do is convert it's OUTPUT, `a`, to a `b`. This is what `Functors` do best.

But when we want to convert a Function's INPUT, we need a different beast. Remember that we're starting with a Function that consumes `a's`. And we want to make it consume `b's`. So that means we need a function that takes `b's` from the outside world and converts them to `a's` which our original function expects, so that we can call it.

Let's look at this in detail:

```
f' :: b -> a ❶
g :: a -> Int ❷
g <<< f' :: b -> Int ❸
```

❶ Our flipped mapping function, which takes `b's` and converts them to `a's` that can be safely passed to

our original function.

❷ Our original function.

❸ These functions compose.

Notice how our `f'` converts INPUTS unlike `f` from `Functor`, which converts OUTPUTS.

It should be clear that we need a Functor-like thing that maps INPUTS instead of OUTPUTS like `Functors` do.

14.3. Contravariant Functor

The full name for `Functor` is Covariant Functor. In math, Covariance is the norm and Contravariance is the opposite.

A Contravariant Functor's mapping function has the arrow reversed from the Covariant Functor:

```
f :: a -> b ❶
f' :: b -> a ❷
```

❶ Arrow goes from `a` to `b`.

❷ Arrow goes from `b` to `a`.

In PureScript, the Contravariant Functor is represented by the `Contravariant` Typeclass:

```
class Contravariant f where
  cmap :: ∀ a b. (b -> a) -> f a -> f b

infixl 4 cmap as >$<
```

There are only 2 differences here, besides the Typeclass name. First, `Contravariant's` Method is called `cmap` and the Type signature of the mapping function has the arrows reversed from `Functor`.

Second, `Contravariant's` Operator Alias, has the greater-than/less-than symbols flipped.

We failed to make a `Functor` instance for `C1` but now with `Contravariant`, we can at least make it an instance of `Contravariant`:

```
instance contravariantC1 :: Contravariant C1 where
  cmap f (C1 g) = C1 (g <<< f) ❶
```

❶ Here `f` converts the INPUT before calling the original function `g`.

Let's compare that with `Functor` Instance for `F1`:

```
instance functorF1 :: Functor F1 where
  map f (F1 g) = F1 (f <<< g) ❶
```

❶ Here we call our original function FIRST, then use `f` to convert it's OUTPUT.

14.4. Contravariant Laws

Of course, there are laws for `Contravariant`:

```
cmap identity = identity             [Identity]
cmap (f <<< g) = cmap f <<< cmap g   [Composition]
```

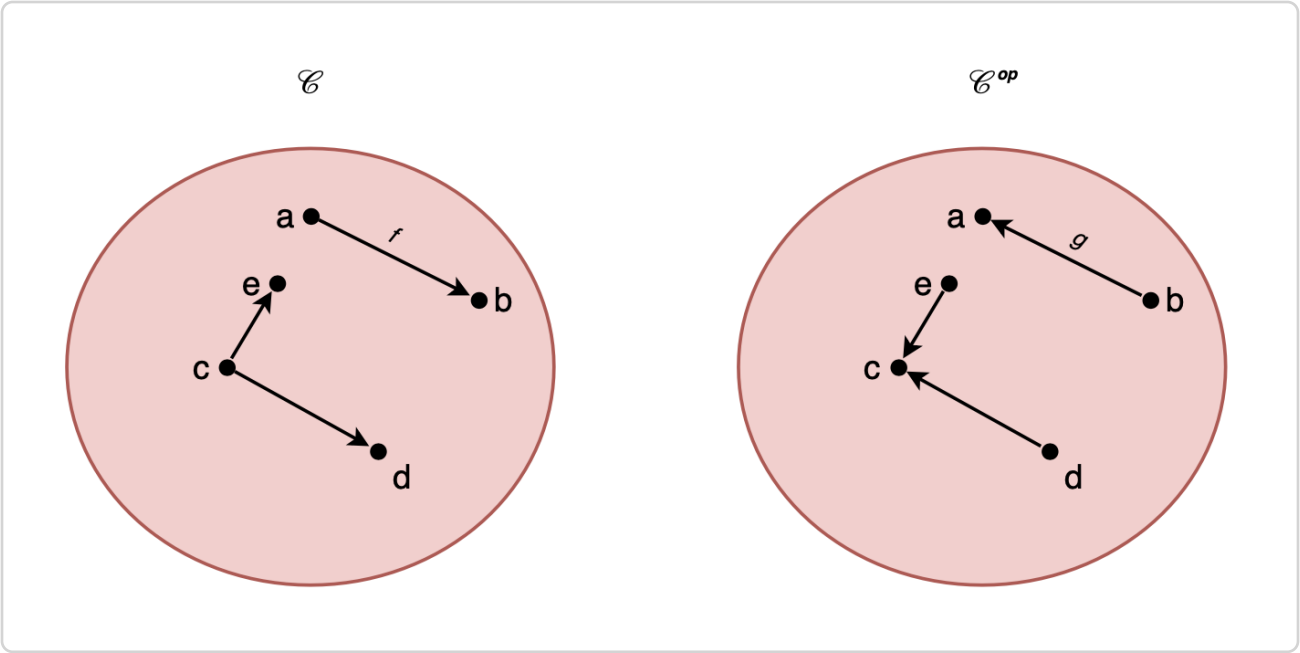
These mirror the `Functor` laws, with the only difference being the Method name:

```
map identity = identity              [Identity]
map (f <<< g) = map f <<< map g      [Composition]
```

Basically, these laws tell us that mapping `identity` has no effect `[Identity]`. And that we can map the composition of 2 functions which is equivalent to mapping them individually and then composing the resulting functions `[Composition]`.

14.5. Contravariant in Category Theory

In Category Theory, there's this idea of an Opposite Category, denoted as $\mathscr{C}^{op}$. If the original Category, denoted by, $\mathscr{C}$, has an arrow from `a` to `b`, then $\mathscr{C}^{op}$ has an arrow from `b` to `a`. All arrows in the Opposite Category are reversed:



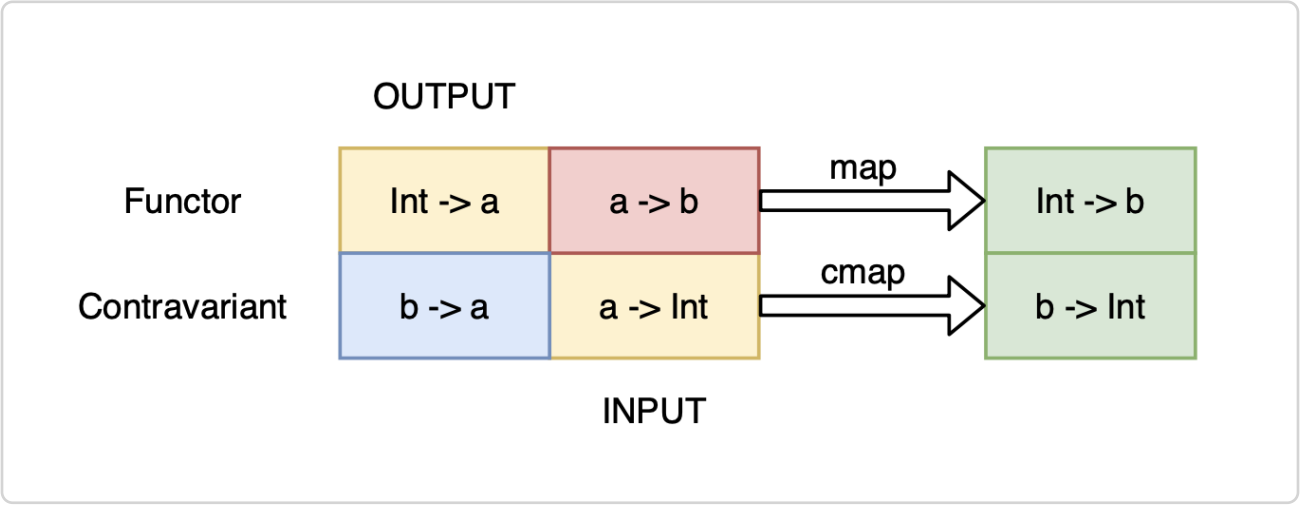
Functions `f` and `g` in the diagram perfectly mirror the mapping functions for `Functor` and `Contravariant`:

```
f :: a -> b   -- Functor mapping function
g :: b -> a   -- Contravariant mapping function
```

A Contravariant Functor from Category $\mathscr{C}$ to Category $\mathscr{D}$ is just a Covariant Functor from the Opposite Category $\mathscr{C}^{op}$ to Category $\mathscr{D}$.

14.6. Choosing `Functor` or `Contravariant`

Stepping back for a second, we have learned that Types that contain functions where the Polymorphic Type Parameter is an OUTPUT, can ONLY have a `Functor` Instance. But when that Polymorphic Type Parameter is an INPUT, then we can ONLY have a `Contravariant`.



In the above diagram, the mapping functions map the OUTPUT of a Functor and the INPUT of a Contravariant.

But this is for 2 Parameters. What about more than 2 Parameters?

Consider the following:

```
newtype F2 a = F2 (a -> Int -> Int)
```

What about this case? What do you think? The Polymorphic Parameter is an INPUT so we'd expect to write a Contravariant Instance. Does that seem right to you?

To help us decide, let's add in the implied Parentheses:

```
newtype F2 a = F2 (a -> (Int -> Int))
```

Hopefully, you can now see how a is clearly an INPUT and so a Contravariant Instance can be created for F2.

But what about this:

```
newtype F3 a = F3 (Int -> Int -> a)
```

a is clearly an OUTPUT, but with the extra Parameter, our implementation cannot simply compose 2 Functions. Let's add the implied Parentheses:

```
newtype F3 a = F3 (Int -> (Int -> a))
```

Now we can see that once we receive the first Parameter, we're back to Int -> a where we can compose f and g. So we need to accept an extra parameter and THEN pass that to g before composing with f.

And since a is an OUTPUT, it's possible to write a Functor Instance:

```
instance functorF3 :: Functor F3 where
  map f (F3 g) = F3 \x -> f <<< g x ❶
```

❶ We first pass x to g making the resulting Type Signature Int -> a which can now be composed with f.

So far so good, but what about the following:

```
newtype F4 a = F4 ((a -> Int) -> Int)
```

It's clear that our Polymorphic Type Parameter, a, is an INPUT, which would normally mean a Contravariant Instance can be created. But, before we run off and try to write that Instance, we also notice that a is an INPUT to a Function that's an INPUT to the larger function. We may begin to wonder if this nuance is pertinent.

To see if this difference matters, let's examine the salient types:

```
newtype F4 a = F4 ((a -> Int) -> Int)

f  :: b -> a              -- mapping function for Contravariant
g  :: (a -> Int) -> Int   -- original function
g' :: (b -> Int) -> Int   -- g AFTER it's been mapped
```

Unfortunately, there is no way to compose f and g to produce g'. That's a clue that Contravariant MAY not be right. Do we need another Functor-like thing? Before we worry about that, let's try to write a Functor Instance:

```
newtype F4 a = F4 ((a -> Int) -> Int)

f  :: a -> b              -- mapping function for Functor
g  :: (a -> Int) -> Int   -- original function
g' :: (b -> Int) -> Int   -- g AFTER it's been mapped
```

We still cannot compose f and g to produce g'. The Types don't line up.

So, this is clearly more complex than our previous examples. We'll need to take a different tact.

We need to map (a -> Int) -> Int to (b -> Int) -> Int, which will be the signature of the function AFTER it's been mapped.

So, let's write the final function after it's been mapped:

```
\h -> ??? :: (b -> Int) -> Int  -- final function AFTER mapping
h :: b -> Int                   -- 1st parameter to final function
```

Notice that our function takes a function h which is of Type b -> Int. And now, all we have to figure out what to replace ??? with.

To help us do that, let's add g to the mix:

```
\h -> ??? :: (b -> Int) -> Int  -- function AFTER mapping
h :: (b -> Int)                 -- 1st parameter to final function
g :: (a -> Int) -> Int          -- original function
```

We know we're going to have to call g eventually. So let's add that:

```
\h -> g ??? :: (b -> Int) -> Int  -- function AFTER mapping
h :: (b -> Int)                   -- 1st parameter to final function
g :: (a -> Int) -> Int            -- original function
```

But we cannot just pass h directly to g since g wants an a -> Int but h is b -> Int.

We could, however, pass h to g if we could just change h to accept a's instead of b's. We can do this by composing h with a conversion function, f, like:

```
f :: a -> b
h <<< f :: a -> Int
```

Here we're going to convert the a we're given to a b so that it can be passed to h. Now, it should be pretty clear that f is the mapping function from Functor. We can see that by just looking at f's Type Signature.

Let's add these to the mix:

```
\h -> g ??? :: (b -> Int) -> Int  -- function AFTER mapping
h :: b -> Int                     -- 1st parameter to final function
g :: (a -> Int) -> Int            -- original function
f :: a -> b                       -- Functor mapping function
h <<< f :: a -> Int               -- a function that g will accept
```

Notice that h <<< f is EXACTLY what g needs. And now we can replace the ??? in our final function with that:

```
\h -> g (h <<< f) :: (b -> Int) -> Int  -- function AFTER mapping
h :: b -> Int                           -- 1st param to final function
g :: (a -> Int) -> Int                  -- original function
f :: a -> b                             -- Functor mapping function
h <<< f :: a -> Int                     -- a function that g will accept
```

The mapping function's Type Signature, a -> b, informs us that we need to create a Functor Instance:

```
newtype F4 a = F4 ((a -> Int) -> Int)

instance functorF4 :: Functor F4 where
  map f (F4 g) = F4 \h -> g (h <<< f)
```

Notice the types on the right-hand and left-hand side of the map equation:

```
F4 g :: F4 ((a -> Int) -> Int)            -- left-hand side
F4 \h -> g (h <<< f) :: (b -> Int) -> Int -- right-hand side
```

This is exactly what we want map to do, convert the function, which takes an a, into a function that takes a b.

But wait...

We successfully wrote a Functor Instance where the Polymorphic Type Parameter was an INPUT. But, our rules said that we cannot do that. According to our rules, we can only create a Contravariant for an INPUT. That was the whole point of Contravariant.

So what gives?

To better understand this dilemma, let's experiment a bit by reversing the function Parameters for F4:

```
newtype F5 a = F5 ((Int -> a) -> Int)
```

Based on our previous experience, we would expect this to also defy our rules. Even though our Polymorphic Type Parameter, a, is an OUTPUT, which normally would mean we could make a Functor Instance, we're going to try to write a Contravariant instead.

Let's start like before by writing the function we want AFTER mapping:

```
\h -> ??? :: (Int -> b) -> Int  -- function AFTER mapping
h :: (Int -> b)                 -- 1st parameter to final function
```

Next, we'll add the original function, which we're going to eventually call in our final function. We'll also add the mapping function from Contravariant:

```
\h -> g ??? :: (Int -> b) -> Int  -- function AFTER mapping
h :: (Int -> b)                   -- 1st parameter to final function
g :: (Int -> a) -> Int            -- original function
f :: b -> a                       -- Contravariant mapping function
```

Like before, we can use `f` to convert `h` into something we can pass to `g`. The only difference is that we call `h` first and then convert its OUTPUT of Type `b` into an `a`. To do this, we'll compose them in the REVERSE order from before:

```
\h -> g ??? :: (Int -> b) -> Int  -- function AFTER mapping
h :: (Int -> b)                   -- 1st parameter to final function
g :: (Int -> a) -> Int            -- original function
f :: b -> a                       -- Contravariant mapping function
f <<< h :: Int -> a               -- a function that g will accept
```

Now we can simply replace `???` with `f <<< h` when we write the `Contravariant` Instance for `F5`:

```
newtype F5 a = F5 ((Int -> a) -> Int)

instance contravariantF5 :: Contravariant F5 where
  cmap f (F5 g) = F5 \h -> g (f <<< h)
```

Our simplistic rules for INPUTS and OUTPUTS didn't work as soon as we looked at Functions that were INPUTS to other functions.

In fact, it was like the rules were polar opposites in this new scenario.

14.7. Polarity

We need a better model for determining whether we can write a `Functor` or a `Contravariant`.

Let's define something called ***Polarity*** as:

```
Positive Position: Any Parameter that is an OUTPUT of a Function
Negative Position: Any Parameter that is an INPUT of a Function
```

Let's look at the Polarity of the Types of this simple function:

```
foo :: a -> b
foo :: - -> +
```

Here it is said that `foo` is Covariant in `b` (POSITIVE) and Contravariant in `a` (NEGATIVE).

If the Polymorphic Parameter is in Positive Position (OUTPUT) then create a `Functor` instance.

If the Polymorphic Parameter is in Negative Position (INPUT) then create a `Contravariant` instance.

Returning to our first `Functor` Instance:

```
newtype F1 a = F1 (Int -> a) ❶

instance functorF1 :: Functor F1 where
  map f (F1 g) = F1 (f <<< g)
```

❶ `a` is in Positive Position.

Since the Polymorphic Parameter `a` is in Positive Position (OUTPUT), we can make a `Functor` instance.

And revisiting our first `Contravariant` Instance:

```
newtype C1 a = C1 (a -> Int) ❶

instance contravariantC1 :: Contravariant C1 where
  cmap f (C1 g) = C1 (g <<< f)
```

❶ `a` is in Negative Position.

Since the Polymorphic variable `a` is in Negative Position (INPUT), we can make a `Contravariant` instance.

But this isn't any different than our old rules except for the added complexity of Positive and Negative positioning.

How is this helping?

Let's look at our problem cases to see how Positive and Negative positioning can help us:

```
newtype F4 a = F4 ((a -> Int) -> Int) ❶ ❷

instance functorF4 :: Functor F4 where
  map f (F4 g) = F4 \h -> g (h <<< f)
```

❶ `a` is in Negative Position.

❷ `a -> Int` is also in Negative Position.

Even though `a` is in Negative Position, we still successfully created a `Functor`. But there's more going on here than just that. The function that `a` is part of, is also in Negative Position. This was not the case with our simple examples.

Finally, let's look at our second problem case:

```
newtype F5 a = F5 ((Int -> a) -> Int) ❶ ❷

instance contravariantF5 :: Contravariant F5 where
  cmap f (F5 g) = F5 \h -> g (f <<< h)
```

❶ `a` is in Positive Position.

❷ `a -> Int` is in Negative Position.

The only real difference between this case and the previous is that `a` is now in a Negative Position.

Summarizing this:

```
newtype  a      function      result    instance
=======  =      ========      ======    ========
   F4    -         -             +      Functor
   F5    +         -             -      Contravariant
```

Can you see why we talk about Polarity and use the terms Positive and Negative?

With `F4`, we have 2 Negatives and they resulted in a Positive.

With `F5`, we had a Negative and a Positive that resulted in a Negative.

This seems to follow the same rules as Multiplication with Positive and Negative Numbers. Multiply 2 Negatives results in a Positive (`F4`). And a Negative multiplied by a Positive is Negative (`F5`).

Here is a cheatsheet for Polymorphic Type Parameters and Polarity:

```
a      function      result    multplication
=      ========      ======    =============
+         +             +      + • + = +      ❶
+         -             -      + • - = -      ❷
-         +             -      - • + = -      ❸
-         -             +      - • - = +      ❹
```

❶ `a` in Positive Position in a Function that's also in Positive Position results in Positive Position.

❷ a in Negative Position in a Function that's in Positive Position results in Negative Position.

❸ a in Positive Position in a Function that's in Negative Position results in Negative Position.

❹ a in Negative Position in a Function that's also in Negative Position results in Positive Position.

Let's apply this new model to our problem cases:

```
-- Negative • Negative = Positive (Functor)
newtype F4 a = F4 ((a -> Int) -> Int)
                  (   -    )         -- `a -> Int` (Negative Position)
                  (-     + )         -- `a`        (Negative Position)

-- Negative • Positive = Negative (Contravariant)
newtype F5 a = F5 ((Int -> a) -> Int)
                  (   -    )         -- `Int -> a` (Negative Position)
                  ( -     +)         -- `a`        (Positive Position)
```

And what if we swap the position of the function:

```
-- Positive • Negative = Negative (Contravariant)
newtype X a = X (Int -> (a -> Int))
                       (    +   )  -- `a -> Int` (Positive Position)
                       (-     + )  -- `a`        (Negative Position)

-- Positive • Positive = Positive (Functor)
newtype Y a = Y (Int -> (Int -> a))
                       (    +   )  -- `a -> Int` (Positive Position)
                       ( -     +)  -- `a`        (Positive Position)
```

Polarity works with more complex Type Signatures in exactly the same way:

```
-- Negative • Negative • Negative = Negative (Contravariant)
newtype Z a = Z (Int -> ((a -> Int) -> Int) -> Int)
                        (         -         )         -- (Negative)
                        ((   -     )        )         -- (Negative)
                        ((-      +)        )         -- (Negative)
```

As a mnemonic for remembering which is Positive and which is Negative, I always say, "Positive Outcome" which reminds me that OUTPUTS are Positive.

14.8. Invariant Functor

Polarity is a useful model for helping us determine if our `a` is in Positive or Negative Position. But what if `a` is in both?

```
newtype Both a = Both (a -> a)
```

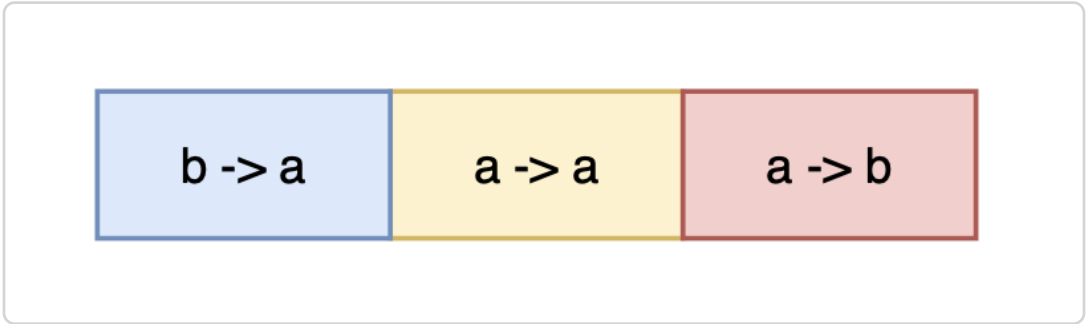
We cannot write a `Functor` Instance because `a` is in Negative Position and we cannot write a `Contravariant` Instance since `a` is also in Positive Position. What we need is something that can handle both.

And that something is called an ***Invariant Functor***. In PureScript, the Typeclass definition is as follows:

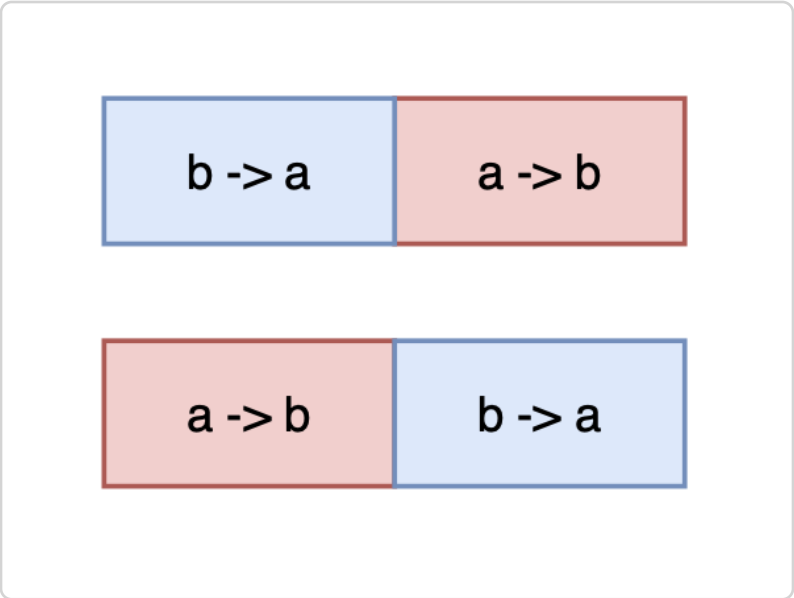
```
class Invariant f where
  imap :: ∀ a b. (a -> b) -> (b -> a) -> f a -> f b
```

The first mapping function, `a -> b` is the same as the one in `Functor`, which maps the OUTPUT value. The second mapping function `b -> a` is the same as `Contravariant`, which maps the INPUT.

We need both functions to map `a -> a` to `b -> b`:



Notice how the 2 functions CAN form an Isomorphism (there's no requirement that the MUST):



If you start with a `b` you can turn it into an `a` and then right back into a `b` again. Same with starting with an `a`.

However, thinking of concrete examples is much harder than you might imagine. For example, you cannot map `Int -> Int` to `Number -> Number`. That's because `Int` and `Number` are not Isomorphic since `1.0`, `1.1` and `1.2` can map to `1` but cannot map back and produce the original numbers.

Sure `1` can map to `1.0`, but it also needs to map to `1.1` and `1.2` etc. But functions cannot do this since they can only have 1 output for 1 input.

Same goes with `Int` and `String`. Converting from `Int` to `String` is easy enough, but going back is problematic since there is no way to map `"abc"` to `Int`. So any function that goes from a `String` to a `Int`, will have a Type Signature of `String -> Maybe Int` since failures are possible.

We'll see some real world examples of Isomorphisms used with Invariants soon enough.

14.9. Invariant Laws

And as we might expect, `Invariant` has a set of laws:

```
imap identity identity = identity    [Identity]
imap (f1 <<< g1) (f2 <<< g2) =
         imap f1 g1 <<< imap f2 g2   [Composition]
```

Not surprisingly, these also mirror the `Functor` laws, with the only difference being the Method name:

```
map identity = identity               [Identity]
map (f <<< g) = map f <<< map g       [Composition]
```

14.10. Homomorphisms

Homo means ***same*** and ***morph*** means ***shape***. And a ***Homomorphism*** is a morphism that preserves shape or structure.

Functors are Homomorphisms since they preserve the Categorical Structure. But what does it mean to "preserve Categorical Structure"? The question really is what is meant by "Categorical Structure".

Recall that a Category has Objects and potentially zero or more Morphisms between those Objects.

Also, remember that every Object has at least one ***Endomorphism*** (***Endo*** means ***internal***) called the Identity Morphism.

And the other thing a Category has is Composition. This too must be preserved after mapping to the other Category.

So "Preserving Categorical Structure" means that the Functor must map all Objects and Morphisms to the new Category while maintaining the SAME Morphisms between those Objects.

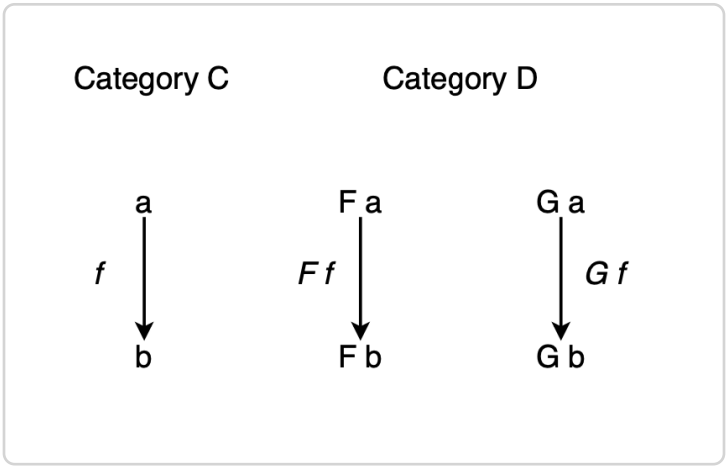
For example, if there's a Morphism between Objects `a` and `b` in the Source Category, the Functor must maintain that Morphism between their counterparts in the Target Category.

14.11. Natural Transformations

Natural Transformations are simply morphisms between Functors. They too are Homomorphisms.

The symbol, `η` is used to represent Natural Transformations since it sort of looks like an `N`, I guess. It's actually the lowercase of `H` (called Eta) in the Greek alphabet.

Let's use the following diagram to help us understand Natural Transformations:



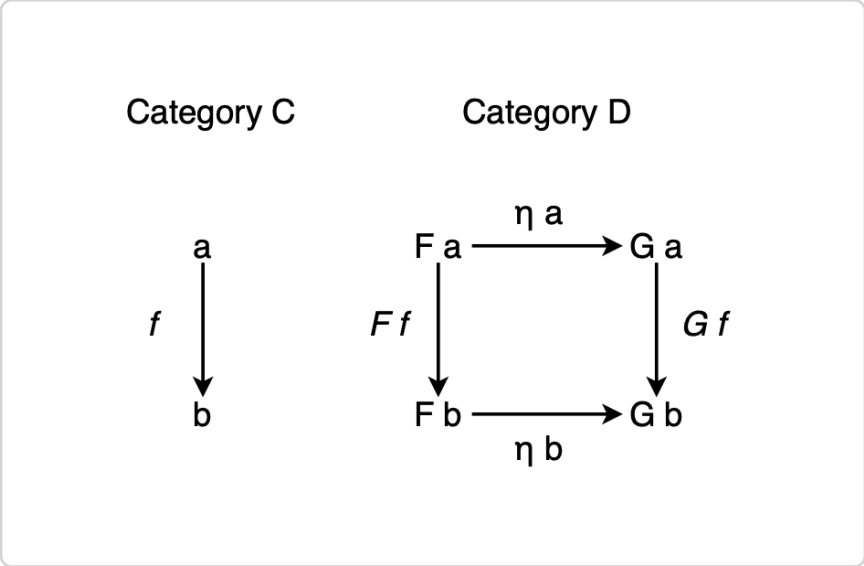
Let's look at this diagram on piece at a time.

On the left, we have Category `C`. The Morphism, `f`, maps from `a` to `b`.

On the right, we have Category `D`. Functor `F` maps `a`, `b` and `f` from Category `C` to Category `D`.

We also have Functor `G`, which maps the same.

Now, let's add Morphisms between our Functors, `F` and `G`:



These Morphisms between Functors are Natural Transformations if and only if, the following is true:

1. The Morphism, `η x : F x → G x` exists in Category `D` for every Object, `x`, in Category `C`.
2. For every Morphism, `f : a → b` in Category `C`, `η b ∘ F f = G f ∘ η a` in Category `D`.

Notice how requirement #1 deals with the mapping of Objects from `C` to `D` and #2 deals with mapping of Morphisms. Not surprising since Functors map BOTH.

Another thing to note here is that the above diagram in Category `D` commutes.



A Commutative Diagram is a diagram where all directed paths (arrows) with the same *start* and *endpoints* lead to the same result.

It's important to note that the above diagram is for Covariant Functors (called `Functors` in PureScript).

If we wanted to represent Contravariant Functors, we would need to reverse all Vertical Arrows in our diagram. The Natural Transformation requirements for Contravariant are:

1. The Morphism, `η x : F x → G x` exists in Category `D` for every Object, `x`, in Category `C`. ***(Same as Covariant)***
2. For every Morphism, `f : b → a` in Category `C`, `η a ∘ F f = G f ∘ η b` in Category `D`. ***(a and b have swapped)***

Let's look at a real-world example, a Homomorphism from `Maybe` to `Either Unit`:

```
hom :: Maybe ~> Either Unit ❶
hom Nothing  = Left unit
hom (Just x) = Right x
```

❶ `~>` is the PureScript symbol for Natural Transformation.

We could've just as easily written the Type Signature as:

```
hom :: ∀ a. Maybe a -> Either Unit a ❶
```

❶ Equivalent to `η a : F a → G a` where `Maybe` is `F` and `Either Unit` is `G`.

We don't have to prove that requirement #1 for Natural Transformation holds since our function compiles and our Type Signature is basically that requirement.

For requirement #2 (for Covariant):

```
f : a -> b
hom ∘ (mMap f) = (eMap f) ∘ hom ❶ ❷
```

❶ `mMap` is map from `Maybe`, i.e. Functor `F`.

❷ `eMap` is map from `Either Unit`, i.e. Functor `G`.

Starting on the left side of the 2nd equation, if we first use `Maybe's map` (`mMap f`) and then transform to an `Either` (`hom`), it's the same as if we first transform to an `Either` (`hom`) and then map using `Either's map` (`eMap f`).

We can confidently reason that the equation holds true thanks to the Functor laws.

If you are still in doubt, check out the `Functor` implementations for `Maybe` and `Either a` (where `a` is `Unit` for our example):

```
instance functorMaybe :: Functor Maybe where
  map _ Nothing = Nothing ❶
  map f (Just x) = Just $ f x ❷

instance functorEither :: Functor (Either a) where
  map _ (Left x)  = Left x ❸
  map f (Right y) = Right $ f y ❹
```

Can you see the similarities?

If you're still not convinced, simply do Algebraic Substitution. Here's `hom` again:

```
hom :: Maybe ~> Either Unit
hom Nothing  = Left unit ❺
hom (Just x) = Right x ❻
```

First, the `Nothing` case:

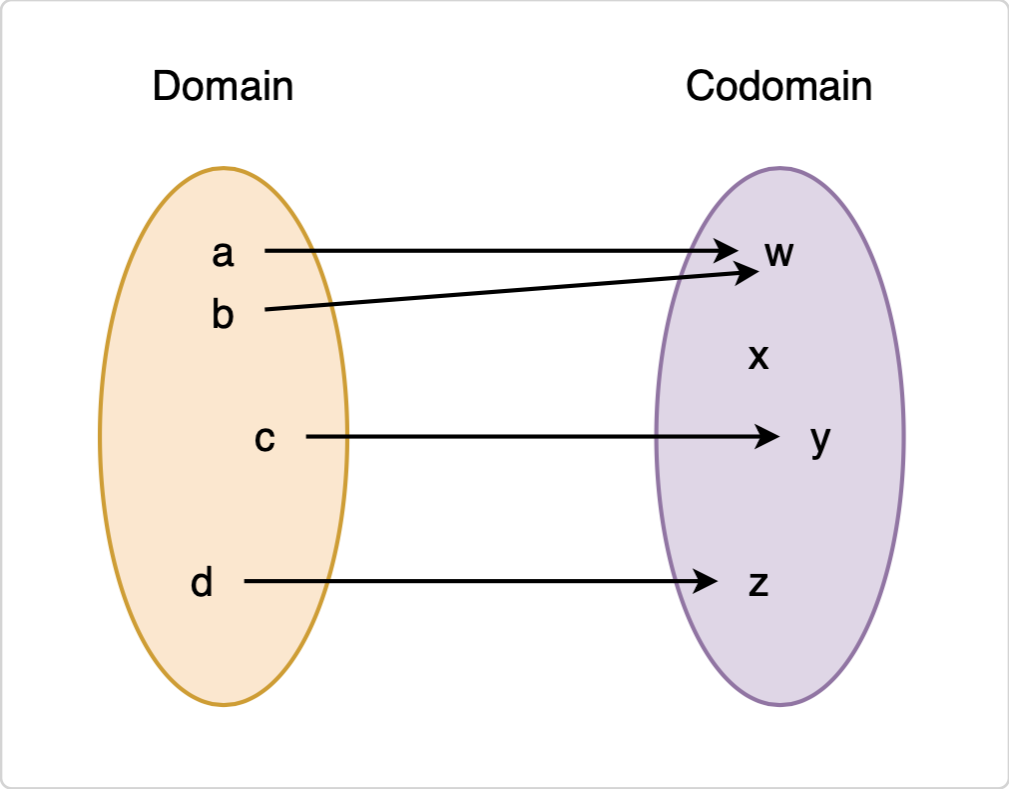
```
(hom ∘ (mMap f)) Nothing = ((eMap f) ∘ hom) Nothing
                                              -- Given
hom Nothing = ((eMap f) ∘ hom) Nothing        -- Substitute into ❶
Left unit = ((eMap f) ∘ hom) Nothing          -- Apply on left side ❺
Left unit = (eMap f) (Left unit)              -- Apply on right side ❺
Left unit = Left unit                         -- Substitute into ❸
```

Now the `Just x` case:

```
(hom ∘ (mMap f)) (Just x) = ((eMap f) ∘ hom) (Just x)
                                              -- Given
hom (Just $ f x) = ((eMap f) ∘ hom) (Just x) -- Substitute into ❷
Right $ f x = ((eMap f) ∘ hom) (Just x)       -- Apply on left side ❻
Right $ f x = (eMap f) (Right x)              -- Apply on right side ❻
Right $ f x = Right $ f x                     -- Substitute into ❹
```

14.12. Monoid Homomorphisms

Functions in Math map elements from their input Set called the Domain into their output Set called the Codomain:



And Monoids are Sets with Binary Operators and certain restrictions:

1. Binary Operator must be Closed over the Set. (Magma)
2. Binary Operator must be Associative. (Semigroup)
3. Set must have an Identity Element. (Monoid)

It's possible to have a Function whose input Set (Domain) is from a Monoid, `M`. But there's no guarantee that the output Set (Codomain) will also be a Monoid. It depends on the function.

Here we're using the equivalent of the `identity` function:

```
idHom :: ∀ m. Monoid m => m -> m
idHom x = x
```

It should be obvious that this preserves the Monoidal Structure, i.e. this function had no effect on the input and therefore, it's result is still a Monoid.

And here's another example, which is just the equivalent of `const mempty`:

```
constHom :: ∀ m. Monoid m => m -> m
constHom _ = mempty
```

This too preserves the Monoidial Structure since it's a Set that contains only `mempty`, which meets all the

requirements of a Monoid:

```
mempty <> mempty = mempty            (Closed)      [Magma]
(mempty <> mempty) <> mempty =
      mempty <> (mempty <> mempty)   (Associative) [Semigroup]
mempty <> mempty = mempty            (Idenity)     [Monoid]
```

We can also write a function that destroys that structure:

```
always123 :: String -> String
always123 _ = "123"
```

This function maps every `String` in our `String Monoid` to `"123"`. The resulting Set, the Codomain, is NOT a Monoid. In fact, it's not even a Magma:

```
"123" <> "123" = "123123"          (NOT CLOSED)
```

But so far, we've only seen contrived, simple examples. There is, however, a common function that we've all used many times that is a Monoid Homomorphism, `length`, the `String` length function. It maps `Strings` to `Ints`.

But before we verify that `length` is a Monoid Homomorphism, it might be good to give the formal definition for Monoid Homomorphism:



A function, `h : M → N`, which maps elements from Monoid `M` to Monoid `N`, is a Homomorphism if:

```
h (m1 • m2) = h(m1) ∗ h(m2)
```

In other words, first combining `m1` and `m2` in `M` and then mapping to `N` is equal to first mapping `m1` and `m2` to `N` and then combining in `N`.

Notice that combining in `M` uses `M's` Binary Operator `•`, while combining in `N` uses `N's`, `∗`. These Binary Operators are different in the case of the `length` Homomorphism, but are the same when using `idHom` or `constHom`.

Here we can see that in action:

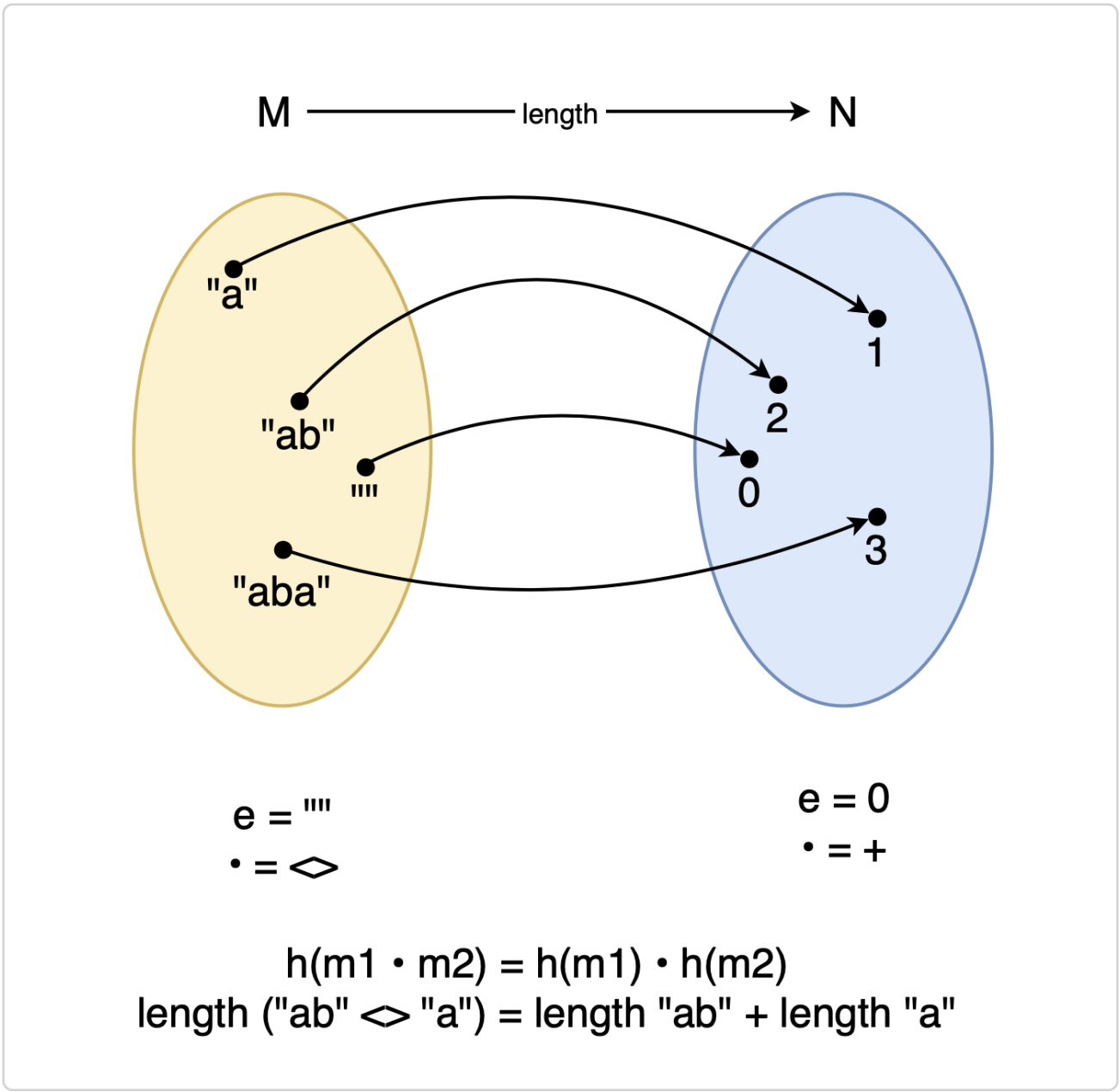
```haskell
length :: String -> Int
length s = ...

length "" = 0 ❶
length (xs <> ys) = length xs + length ys ❷
```

❶ Maps `String's mempty` to `Int's mempty`.

❷ `<>` is String's Binary Operator and `+` is Int's

We also see this visually:



Here `""` is the Empty Element of `String` and `length` produces the Empty Element for `Int`, `0`.

And since the `length` of concatenated `Strings` is equal to the addition of their individual lengths, `length` preserves the Monoidal Structure.

The `length` function from `Data.Array` and `Data.List` are also Monoid Homomorphisms:

```
import Data.Array as A
import Data.List as L

A.length [] = 0
A.length (xs <> ys) = A.length xs + A.length ys

L.length Nil = 0
length (xs <> ys) = length xs + length ys
```

One thing to note is that a Homomorphism is a one-way mapping. There is no requirement that an Inverse Function exists to map back. In fact, it's impossible to write an inverse function for `length` that maps from `Int's` back to `String's`:

```
length "abc" = 3
length "xyz" = 3
```

Here we have `3` for both `"abc"` and `"xyz"`. If we start with `3` we CANNOT map back to BOTH `Strings`. Functions cannot do that.

But what about cases where can have an Inverse Homomorphism? Well, those are called ***Isomorphisms***.

14.13. Isomorphisms

As we have seen, `Maybe` and `Either Unit` are Isomorphic. Here are a pair of functions that form an Isomorphism between `Maybe` and `Either Unit`:

```
maybeToEither :: Maybe ~> Either Unit
maybeToEither Nothing  = Left unit
maybeToEither (Just x) = Right x

maybeFromEither :: Either Unit ~> Maybe
maybeFromEither (Left _)   = Nothing
maybeFromEither (Right x)  = Just x
```

We can write a helper Type to model an Isomorphism:

```
data Iso a b = Iso (a -> b) (b -> a)
```

The first Parameter is a function FROM `a` to `b` and the second Parameter is a function back TO `a`.

Let's write the Type Signature for the Data Constructor:

```
Iso :: ∀ a b. (a -> b) -> (b -> a) -> Iso a b
```

If we look carefully at the functions that are the Parameters to the Data Constructor, `Iso`, we may recognize it as being very similar to our old friend `imap` from `Invariant`:

```
class Invariant f where
  imap :: ∀ a b. (a -> b) -> (b -> a) -> f a -> f b
```

When we initially looked at `Invariants`, we stated that these 2 functions CAN be Isomorphic. They don't have to be, but if they're not, it's not very useful since we're unable to represent all `a's` as `b's` or vice versa. It really depends on the application.

For the cases where they are Isomorphic, we can write a helper function to use `Iso's` with `Invariants`:

```
iso ::  ∀ f a b. Invariant f => Iso a b -> f a -> f b ❶
iso (Iso to from) = imap to from
```

❶ We constraint `f` to be an Invariant Functor.

Next, let's create a Type that's an Invariant Functor. We're going to create a Type for an ***Endomorphism***. An Endomorphism is a morphism back to itself. The Identity Function, `identity`, is an EndoMorphism:

```
identity :: ∀ a. a -> a
identity x = x
```

Notice the use of the prefix ***Endo*** like we saw with Endofunctors, i.e. Functors that map back to the same Category, viz. all Functors in PureScript and Haskell. From the perspective of Category Theory, Endomorphisms are Morphisms that map back to the original Object. In programming, this is a function that maps a Type back to that same Type.

Here's our Type to model Endomorphisms:

```
newtype Endo a = Endo (a -> a)
```

Since `a` is both in POSITIVE and NEGATIVE position, we can write an `Invariant` Instance for `Endo`:

```
instance invariantEndo :: Invariant Endo where
  imap ab ba (Endo aa) = Endo (ab <<< aa <<< ba) ❶ ❷
```

❶ `aa` means a function from `a -> a`. `ab` means a functions from `a -> b`. `ba` means a function from `b -> a`.

❷ After calling `imap` we have mapped an `Endo a` to `Endo b`. `ba` maps INPUT to `aa`, where `ab` maps the OUTPUT of `aa`.

So far, we've been laying the groundwork to be able to see how we can leverage functions written in an Isomorphic Type to the Type we'd like it to be in. Let's look at an example where we have a function written using `Maybe` and we'd like a function that uses `Either String` instead, but we don't want to have to rewrite our original function:

```
import Data.Either (note, hush) ❶ ❷

data Iso a b = Iso (a -> b) (b -> a)

iso :: ∀ a b f. Invariant f => Iso a b -> f a -> f b
iso (Iso to from) = imap to from

newtype Endo a = Endo (a -> a)

instance invariantEndo :: Invariant Endo where
  imap ab ba (Endo aa) = Endo (ab <<< aa <<< ba)

positive :: Maybe Int -> Maybe Int ❸
positive x = if x >= Just 0 then x else Nothing

isoMaybeEither :: String -> Iso (Maybe Int) (Either String Int) ❹
isoMaybeEither err = Iso (note err) hush

positiveEither :: Either String Int -> Either String Int ❺
positiveEither = f where (Endo f) =
  iso (isoMaybeEither "Not a positive integer") (Endo positive)
```

❶ `note` converts a `Maybe` to an `Either a`. If the `Maybe` is `Nothing`, it'll use a specified default value to create a `Left`.

❷ `hush` converts an `Either a` to `Maybe`. It "hushes" the error information, i.e. throws it away.

❸ If `Just x` is not Positive or it's a `Nothing`, then this function fails with a `Nothing`.

❹ We model the Isomorphism between `note` and `hush`.

❺ We have leveraged `iso` to convert `positive` to `positiveEither`, by using the Isomorphism, `isoMaybeEither`. In detail, we wrapped `positive` in an `Endo` since it has an `Invariant` Instance. Then we map it and unwrap it to get the mapped function back out.

By recognizing Isomorphisms, we are able to simply map between Isomorphic Types instead of being forced to rewrite functions that don't return the exact Type we'd like to use.

14.14. Functor Instances for Function

We can think of Functions 2 ways:

```
a -> b     -- infixed
(->) a b   -- prefixed
```

When we consider the prefixed way of writing a Function, we can see that `(->)` is a just Type Constructor of 2 Parameters, `a` and `b`. It's no different than `Either a b`. To be completely pedantic, it's a Type Operator since it contains Symbols.

`->` is built in to PureScript, but we can make our own Type Operators and then use them in our Type Signatures:

```
infixr 6 type Tuple as & ❶

tup :: ∀ a b. a -> b -> a & b ❷
tup x y = Tuple x y
```

❶ We've defined `&` to be an infix operator for Tuple.

❷ Here's where we're using our Type Operator.

And since `->` is no different than `Either`, we should be able to write a `Functor` Instance for functions:

```
instance functorFn :: Functor ((->) r) where ❶ ❷ ❸
  map f g = ??? ❹
```

❶ It's tradition to use `r` as the first Type Parameter to `(->)`.

❷ Can't follow the normal naming convention since this is for `(->)` which is built-in to the language and so there's no function that it's an alias for. So we replaced the Operator with `Fn` in the name.

❸ `((->) r)` holds the Input Parameter Type fixed, i.e. we cannot map over the INPUT. This is consistent with `Functors` which map OUTPUTS.

❹ We need to apply our mapping function `f` to the result of `g`.

We should replace `???` with code that maps `f` over the Output Value of `g`, our function. We can do this easily by using compose:

```
instance functorFn :: Functor ((->) r) where
  map f g = f <<< g ❶
```

❶ Call our function and then map the Output Value.

That's great. Functions are also `Functors`. But Functions have 2 Type Parameters. Can we write a `Bifunctor` Instance?

Should be able to. But before we do, let's remind ourselves what `Bifunctor` looks like:

```
class Bifunctor f where
  bimap :: ∀ a b c d. (a -> b) -> (c -> d) -> f a c -> f b d
```

Remember that the `Bifunctor's` Type Parameter, `f`, is `(->)`, i.e. the Constructor for Functions. So lets rewrite the `bimap` Type Signature specialized for Functions by replacing `f` with `(->)`:

```
  bimap :: ∀ a b c d. (a -> b) -> (c -> d) -> (->) a c -> (->) b d
```

Now let's rewrite the Function Type Operator as INFIXED:

```
  bimap :: ∀ a b c d. (a -> b) -> (c -> d) -> (a -> c) -> (b -> d)
```

Now, we're ready to write the `Bifunctor` Instance for Function:

```
instance bifunctorFn : Bifunctor (->) where
  bimap f g h = ???
```

At this point, we have 2 mapping functions, `f` and `g`. Our original function is `h`. Notice that `h` has a Type Signature of `a -> c`. That means that we can take the output of `h :: a -> c` and pass it to `g :: c -> d`.

Let's capture that part:

```
 h >>> g :: a -> d
```

Now we just have to figure out how to use `f`:

```
h >>> g :: a -> d
f :: a -> b
```

We cannot compose `f` with `h >>> g` since the Types don't line up. `f` is backwards from what we want. We've seen this before with `Contravariant`. It looks like we CANNOT write a `Bifunctor` Instance for Functions. But why?

To understand why this is, let's first look at `Functor`:

```
class Functor f where
  map f :: ∀ a b. (a -> b) -> f a -> f b
```

This is a Covariant Functor. It has a mapping function from `a -> b`, which means it maps OUTPUTS. The mapping function is Covariant.

Now let's look at `Contravariant`:

```
class Contravariant f where
  cmap :: ∀ a b. (b -> a) -> f a -> f b
```

This is Contravariant. It has a mapping function that maps INPUTS, meaning it's Contravariant.

Let's revisit Bifunctor with Covariance and Contravariance in mind:

```
class Bifunctor f where
  bimap :: ∀ a b c d. (a -> b) -> (c -> d) -> f a c -> f b d
```

Notice that the 2 mapping functions are both Covariant. So `a` and `c` must be Covariant to have a `Bifunctor` Instance, since both function must map INPUTS.

But that's impossible with Functions. By definition a Function has 1 INPUT and 1 OUTPUT:

```
f :: a -> b
f :: (->) a b
```

That means that `(->)` is Contravariant in `a` and Covariant in `b`. So this is why we cannot write a `Bifunctor` Instance for `(->)`.

But what about `Invariant`? It is has 2 mapping functions. The first is Covariant (maps INPUTS) and the second is Contravariant (maps OUTPUTS).

Problem is that `f` in `Invariant` has the Kind Signature, `Type -> Type`, but `(->)` has `Type -> Type -> Type`, i.e. `Invariant` takes 1 Parameter, whereas `(->)` takes 2:

```
class Invariant f where ❶
  imap :: ∀ a b. (a -> b) -> (b -> a) -> f a -> f b
```

❶ Here's `f` has Kind `Type -> Type`.

Couldn't we just fix the INPUT Parameter like we did to create a `Functor` Instance?

```
instance invarianFn :: Invariant ((->) r) where
  imap :: ∀ a b. (a -> b) -> (b -> a) -> (r -> a) -> (r -> b) ❶
  imap f g h = ??? ❷ ❸
```

❶ Here is the `imap` Type Signature specialized for `(->)`.

❷ `f` and `g` are our mapping functions. `f` is Covariant and `g` is Contravariant.

❸ `h` is our Function.

We can combine `h` and `f` like:

```
h >>> f :: r -> b
```

Yielding us:

```
instance invarianFn :: Invariant ((->) r) where
  imap :: ∀ a b. (a -> b) -> (b -> a) -> (r -> a) -> (r -> b)
  imap f g h = h >>> f ❶
```

❶ Didn't use `g`.

And we're done. But we never used `g`, which is basically just a `Functor`:

```
instance functorFn :: Functor ((->) r) where
  map f h = h >>> f ❶
```

❶ This was written earlier as `f <<< g`, which is equivalent. I renamed `g` to `h` and flipped the compose operator and the order of Parameters.

There's no reason to write the `Invariant` since it's the same as `Functor`. So, clearly Functions aren't `Invariant` Functors.

What we really need is some kind of Functor that's Contravariant in the first Parameter and Covariant in the second.

14.15. Profunctor

To be able to map both INPUT and OUTPUT Types of a Function, we need to use what's called a ***Profunctor***.

```
class Profunctor p where
  dimap :: ∀ a b c d. (b -> a) -> (c -> d) -> p a c -> p b d
```

This looks a lot like `Bifunctor` with the notable difference of the reversal of the Types in the Type Signature of the first mapping function, viz. `b -> a`.

That means that a `Profunctor` is Contravariant in the first Parameter (maps INPUTS) and Covariant in the second (maps OUTPUTS).

Let's write a `Profunctor` Instance for `(->)`:

```
instance profunctorFn :: Profunctor (->) where
  dimap :: ∀ a b c d. (b -> a) -> (c -> d) -> (a -> c) -> (b -> d) ❶
  dimap f g h = ??? ❷
```

❶ The `dimap` Type Signature specialized for `(->)` in INFIXED form, i.e. `p a c` first becomes `(->) a c` and then becomes `a -> c`. And through the same process, `p b d` becomes `b -> d`.

❷ `f` and `g` are our mapping functions. And `h` is our original function.

Since `f` is Contravariant (maps INPUTS), we should expect to call it BEFORE our function, `h`. In the same vein, since `g` is Covariant (maps OUTPUTS), we should expect to call it AFTER `h`. So let's just try to compose them in the that order to see if the Types line up:

```
f :: b -> a
g :: c -> d
h :: a -> c
g <<< h <<< f :: b -> d ❶
```

❶ This is exactly the return Type of `dimap`.

Since the composition of `f`, `g` and `h` is the correct Type, we can simply replace `???` with that:

```
instance profunctorFn :: Profunctor (->) where
  dimap :: ∀ a b c d. (b -> a) -> (c -> d) -> (a -> c) -> (b -> d)
  dimap f g h = g <<< h <<< f
```

And like with `Bifunctor`, there are `Profunctor` helpers if you want to map only one side or the other:

```
lcmap :: ∀ a b c p. Profunctor p => (b -> a) -> p a c -> p b c

rmap :: ∀ a c d p. Profunctor p => (c -> d) -> p a c -> p a d
```

Be careful here. `rmap` from `Data.Profunctor` will name collide with `rmap` from `Data.Bifunctor`.

`lcmap` probably stands for `Left Contravariant Map` so there's no name collision with `lmap` from `Data.Bifunctor`.

14.16. Profunctor Laws

I wouldn't want to disappoint by leaving out the Profunctor Laws:

```
dimap identity identity = identity       [Identity]
dimap (f1 <<< g1) (g2 <<< f2) = ❶
          dimap g1 g2 <<< dimap f1 f2    [Composition]
```

❶ `f1 <<< g1` is swapped when compared to the `Bifunctor` Law.

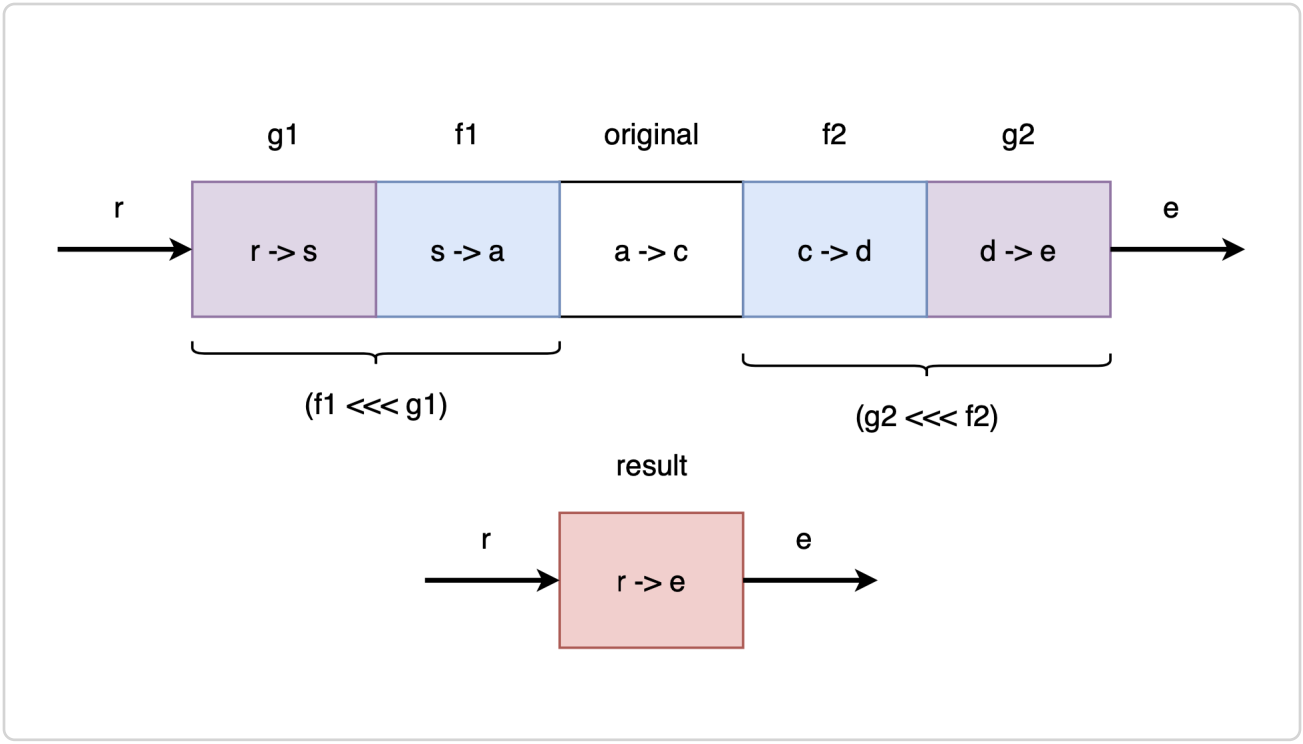
Notice how this is analogous to `Bifunctor`:

```
bimap identity identity = identity       [Identity]
bimap (g1 <<< f1) (g2 <<< f2) =
          bimap g1 g2 <<< bimap f1 f2    [Composition]
```

There are 2 notable differences. First, the mapping function is called `dimap` for `Profunctor`.

Second, in `Profunctor`, the Contravariant composition is `f1 <<< g1`, whereas in `Bifunctor` the Covariant composition is `g1 <<< f1`. They are swapped because we're mapping INPUTS in the `Profunctor`.

Here's a diagram showing the mappings of the INPUT and the OUTPUT to help us understand why:



Remember, `dimap f1 f2` is called first (see Profunctor Composition Law). This is why we see `f1` and `f2` next to the original function in the diagram.

`dimap g1 g2` is called second (see same Law). This is why we see `g1` and `g2` outside of `f1` and `f2`, respectively.

`g1` get's the INPUT from the outside world, maps it and passes the value to `f1` who maps it and passes it to the original function, in other words `f1` after `g1` or `f1 <<< g1`.

But on the OUTPUT side, `f2` gets the OUTPUT from the original function, maps it and passes the value to `g2` who maps it and returns that to the outside world, in other words `g2` after `f2` or `g2 <<< f2`.

14.17. Profunctor and Isomorphisms

A `Profunctor` is very similar to `Invariant` with the most notable exception being that `Profunctor` has 2 Type Parameters and `Invariant` has 1.

That means that we should be able to create a similar Isomorphism helper like we did for `Invariant`. Before we do, let's review the Invariant helper:

```
data Iso a b = Iso (a -> b) (b -> a) ❶

iso :: ∀ a b f. Invariant f => Iso a b -> f a -> f b
iso (Iso to from) = imap to from
```

❶ A Data Type that models an Isomorphism with 2 functions that are inverses of each other.

Now we want to write helper `isoP` to perform the analogous function as `iso` but for `Profunctors`:

```
isoP :: ∀ a b p. Profunctor p => Iso a b -> p a a -> p b b ❶
isoP (Iso to from) = dimap from to ❷
```

❶ The Type Parameters, `c` and `d`, we're used to seeing in the second position with `Profunctor` are missing because we are working with `Iso`, which only has `a` and `b`.

❷ Notice how we reverse `from` and `to`.

Since `Profunctor` is Contravariant in the first Parameter, we have to use `from` as the first mapping function for `dimap`. And since `Profunctor` is Covariant in the second, `to` is our second mapping function.

As usual, `Profunctor` Type Signatures can look pretty abstract. The best way to gain an intuition for them is to specialize them for Functions, i.e. `(->)`:

```
isoPFn :: ∀ a b. Iso a b -> (a -> a) -> (b -> b)
```

When we were working with `Invariant`, we created `Endo` as our `Invariant`:

```
newtype Endo a = Endo (a -> a)

instance invariantEndo :: Invariant Endo where
  imap ab ba (Endo aa) = Endo (ab <<< aa <<< ba)
```

We don't need to do this for `Profunctor` since we can just use `Function`, which is a `Profunctor` and it's already built into the language.

At this point, we can rewrite the example from `Invariant`:

```
import Data.Either (note, hush)

data Iso a b = Iso (a -> b) (b -> a)

isoPFn :: ∀ a b. Iso a b -> (a -> a) -> (b -> b)
isoPFn (Iso to from) = dimap from to

positive :: Maybe Int -> Maybe Int ❶
positive x = if x >= Just 0 then x else Nothing

isoMaybeEither :: String -> Iso (Maybe Int) (Either String Int) ❶
isoMaybeEither err = Iso (note err) hush

positiveEither :: Either String Int -> Either String Int ❷
positiveEither = isoPFn (isoMaybeEither "Not a positive integer") positive
```

❶ These 2 functions didn't change from `Invariant`.

❷ There is no need for the `where` section like we had in the `Invariant` version where we had to wrap and unwrap our functions in `Endo`. Since Functions are `Profunctors`, we can use them directly.

14.18. Profunctor Perspective

`Profunctors` can be thought of as generalized Function Application:

```
class Profunctor p where
  dimap :: ∀ a b c d. (b -> a) -> (c -> d) -> p a c -> p b d

instance profunctorFn :: Profunctor (->) where
  dimap :: ∀ a b c d. (b -> a) -> (c -> d) -> (a -> c) -> (b -> d)
  dimap f g h = g <<< h <<< f
```

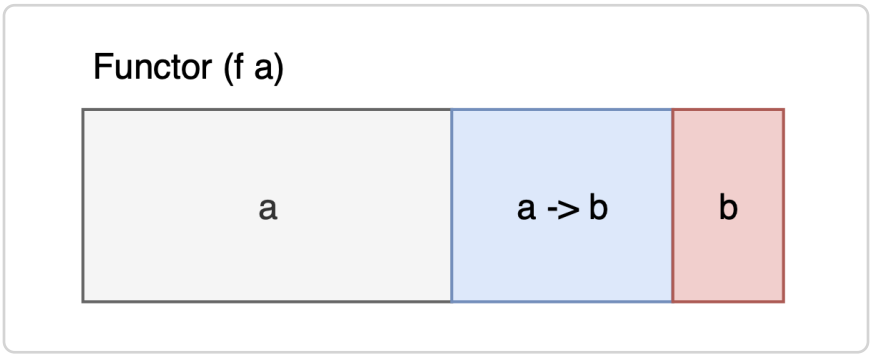
When the abstractness of `Profunctor` seems confusing, you can simply think of `p a b` as `a -> b`. It helps us to have a real world example of this abstraction, especially one that we're very familiar with.

This perspective will come in handy when you use ***Lenses***. Lenses allow us to traverse complex Data Structures to either retrieve or modify values. Lenses, like functions, are composable.

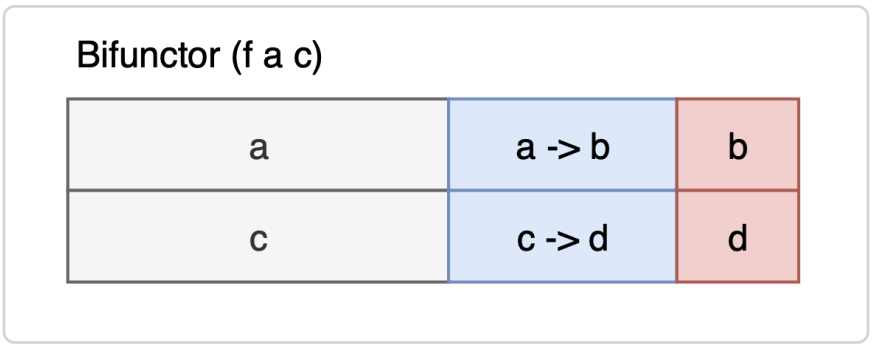
When you study or use Lenses, understanding `Profunctors` will be critical since Lenses in PureScript (and some libraries in Haskell) are built on top of `Profunctors`.

14.19. Functor Summary

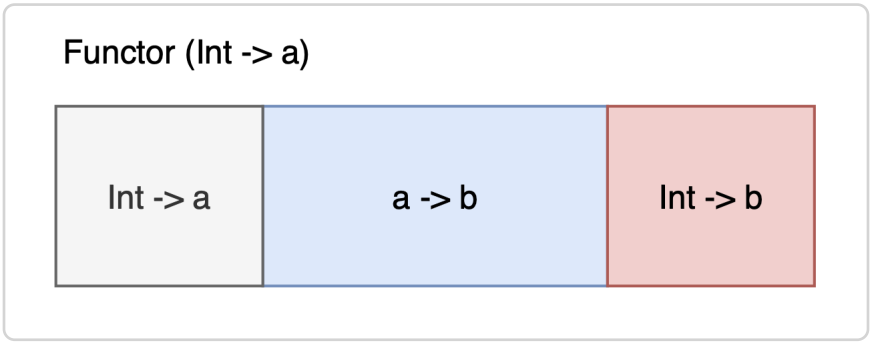
We've looked at a bunch of Functors and after a while, they can all blend in to one another. Here are some diagrams to help distinguish them:



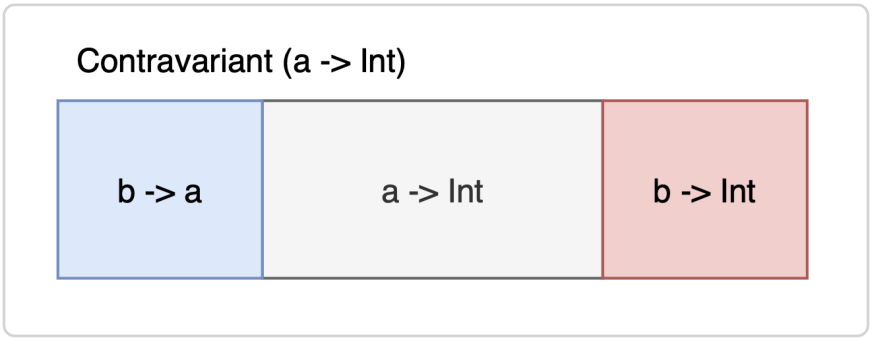
`Functor` containing an `a`, which is transformed by `a -> b` producing `b`.



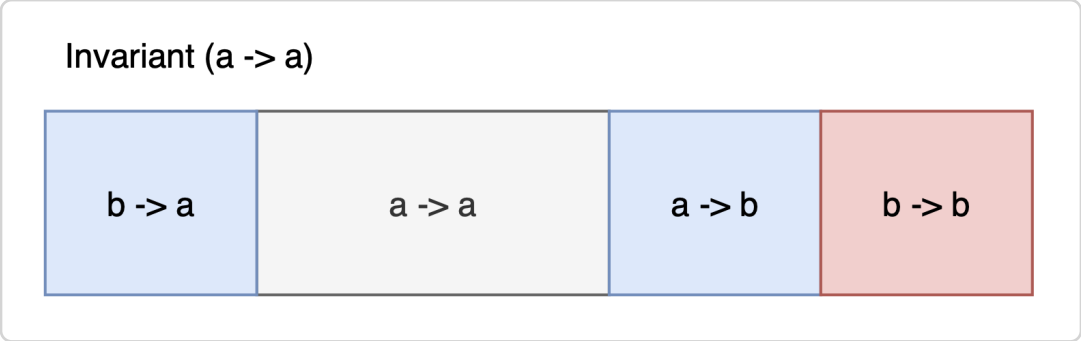
`Bifunctor` containing `a` and `c`, which are transformed by `a -> b` and `c -> d` producing `b` and `d`.



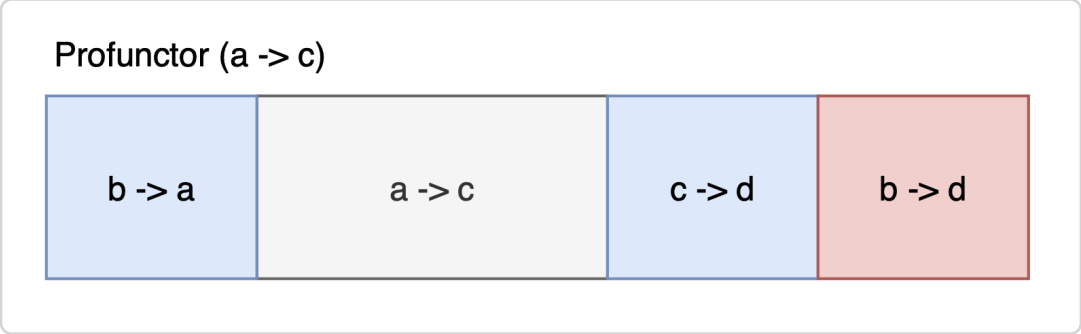
Covariant `Functor` containing `Int -> a`, which is composed with `a -> b` producing `Int -> b`.



`Contravariant` containing `a -> Int`, which is composed with `b -> a` producing `b -> Int`.



`Invariant` containing `a -> a`, which is composed with `b -> a` and `a -> b` producing `b -> b`.



Profunctor containing `a -> c`, which is composed with `b -> a` and `c -> d` producing `b -> d`.

Here's a quick lookup table for Functors:

```
- Functors with 1 mapping function (f)
  - Functor       (Covariant)                 (a -> b) ❶
  - Contravariant (Contravariant)             (b -> a) ❶

- Functors with 2 mapping functions (f, g),
  - Bifunctor     (Covariant, Covariant)      (a -> b, c -> d) ❷
  - Invariant     (Covariant, Contravariant)  (a -> b, b -> a) ❶
  - Profunctor    (Contravariant, Covariant)  (b -> a, c -> d) ❷
```

❶ Has 1 Type Parameter.

❷ Has 2 Type Parameters.

14.20. Functor Intuition

To help us get some intuition regarding Functors, let's look at the different Functor Data Types:

```
Functor a
Bifunctor a b
Contravariant a
Invariant a
Profunctor a b
```

We can think of these Functors in the following way:

- a `Functor` is something that CONTAINS `a's.
- a `Bifunctor` is something that CONTAINS `a's AND `b's.
- a `Contravariant` is something that CONSUMES `a's.
- an `Invariant` is something that CONSUMES `a's and PRODUCES `a's.
- a `Profunctor` is something that CONSUMES `a's and PRODUCES `b's.

Keep in mind that when we say, e.g. `Functor` contains an `a`, `a` can be ANY Type including a function.

Also, try to not over generalize Functors by thinking they're simply containers. As we progress, we'll see that is NOT always the case. It's better to think of them as Contexts.

Chapter 15. Coding More Functors

As always, it's good to take the time to code the things we've just learned for ourselves. It's easy to fool ourselves that we understand something when we first learn it. But trying to implement what we've just learned is a great way to both let us know at what level we actually did learn it and to help deepen that understanding.

Return to the project that you've been coding in and create a file called `Ch15.purs` and add the following:

```
module Ch15 where

import Prelude ❶

import Effect (Effect)
import Effect.Console (log)

test :: Effect Unit
test = do
  log "placeholder" ❷
```

❶ We're not rewriting anything from Prelude in this Chapter, so we can just import everything from Prelude.

❷ We'll delete this soon enough. It's just here to keep the compiler happy.

Next, change your `Main` module to import `Ch15` and call its `test` function:

```
module Main where

import Prelude
import Effect (Effect)

import Ch15 as Ch15

main :: Effect Unit
main = Ch15.test
```

If you run:

```
npx spago run
```

you should see it build and print:

```
[info] Build succeeded.
placeholder
```

15.1. Writing the Predicate odd

Let's revisit the concept of a Predicate. A Predicate is a function that takes a value and determines if it meets some predefined criteria and returns a `Boolean`. An example of that is `even` from `Data.Int`:

```
import Data.Int.Bits ((.&.)) ❶

even :: Int -> Boolean
even x = x .&. 1 == 0 ❷
```

❶ Import bit level AND operator.

❷ Mask the Least Significant bit and if it's 0 then we have an even number.

If you're not too good with Binary Arithmetic, here's a quick primmer. Here is a table of numbers in both Decimal and their Binary Equivalent:

```
0 = 0000
1 = 0001
2 = 0010
3 = 0011
4 = 0100
5 = 0101
6 = 0110
7 = 0111
```

Notice that all odd numbers have a `1` in the Least Significant position, i.e. the rightmost bit. Also remember that an AND operation is only `true`, or, in the Binary world, equal to `1`, when both values are `true`:

```
0 .&. 0 = 0
0 .&. 1 = 0
1 .&. 0 = 0
1 .&. 1 = 1
```

With that Truth Table in mind, here are a few numbers ANDed with `1` to see what's going on inside the function `even`:

```
    0000 = 0
.&. 0001 = 1 (this is the 1 from `even`)
    ----
    0000 = 0

    0001 = 1
.&. 0001 = 1 (this is the 1 from `even`)
    ----
    0000 = 1

    0010 = 2
.&. 0001 = 1 (this is the 1 from `even`)
    ----
    0000 = 0

    0011 = 3
.&. 0001 = 1 (this is the 1 from `even`)
    ----
    0001 = 1
```

Notice how every time we AND an odd number with `1` we get `1` and every time we AND an even number we get `0`. That's because odd numbers have a `1` in the Least Significant position, which when ANDed with `1`, which also has a `1` in it's Least Significant position, results in `1` (see above Truth Table).

Here is the general case:

```
    xxx0 = some even number (x is a don't care value)
.&. 0001 = 1 (this is the 1 from `even`)
    ----
    0000 = 0

    xxx1 = some odd number (x is a don't care value)
.&. 0001 = 1 (this is the 1 from `even`)
    ----
    0001 = 1
```

Here `x` is a ***don't-care*** value since it doesn't matter whether it's a `0` or a `1` since it's going to be ANDed with `0`, which is always `0` (`false`).

Hopefully, this has helped you see why `even` is ANDing with `1` and checking for a `0` result.

Now, just to drive this point home, write `odd`, but first add the following to your test code:

```
log $ show $ odd 0 ❶
log $ show $ odd 1 ❷
```

❶ Prints `false`.

❷ Prints `true`.

Now, you can remove the placeholder line your test code. Give it a try before moving on, using the above Truth Table to help you if necessary.

15.2. Code for the Predicate odd

Here's the Truth Table again:

```
0 .&. 0 = 0
0 .&. 1 = 0
1 .&. 0 = 0 ❶
1 .&. 1 = 1 ❷
```

❶ The fact that `1` ANDed with `0` is `0` can be used to determine if a number is even.

❷ The fact that `1` ANDed with `1` is `1` can be used to determine if a number is odd.

Since, in Binary, odd numbers end in `1`, we should do exactly the same thing as `even`, with one slight difference, we'll check to see if the result is `1`:

```
import Data.Int.Bits ((.&.)) ❶

odd :: Int -> Boolean
odd x = x .&. 1 == 1 ❷
```

❶ Need to import the bitwise AND operator.

❷ Unlike even, we check for `1`.

Another way we could've written `odd` would be to use the Modulo Operator, i.e. the remainder of Division:

```
odd' :: Int -> Boolean
odd' x = x `mod` 2 == 1 ❶
```

❶ Using `mod` in infixed notation.

If dividing by `2` has a remainder of `1`, then we have an odd number. This approach is good, but not as efficient as the bitwise operation. But in reality, no one's going to tell the difference in your program unless

you're doing millions and millions of checks for odd and maybe not even then.

15.3. Writing a `Predicate` Type

Let's look at the Type Signatures of some concrete examples of Predicates to see if we can see a pattern that will allow us to write a general Predicate Type:

```
even :: Int -> Boolean
odd :: Int -> Boolean
null :: String -> Boolean
null :: ∀ a. List a -> Boolean
null :: ∀ a. Array a -> Boolean
member :: ∀ a. Ord a => a -> Set a -> Boolean
member :: ∀ k v. Ord k => k -> Map k v -> Boolean
```

Take a look at these Type Signatures and see if you can see a pattern. One thing you should notice is that each function is asking a question, for example, "Is this number even".

Next, you may notice that the answer is always a yes/no or in our particular case true/false. So the return Type is always `Boolean`.

And, you may also notice, that ALMOST all of the functions take only 1 Parameter and return a Boolean. Only 2 do not:

```
member :: ∀ a. Ord a => a -> Set a -> Boolean
member :: ∀ k v. Ord k => k -> Map k v -> Boolean
```

Is there anyway we can make these only take 1 Parameter so they match our pattern?

Remember, the Isomorphism formed by `curry` and `uncurry`?

If you don't remember, here are their Type Signatures to help jar your memory:

```
curry :: ∀ a b c. (Tuple a b -> c) -> a -> b -> c
uncurry :: ∀ a b c. (a -> b -> c) -> Tuple a b -> c
```

We can take any function of 2 Parameters and `uncurry` it to produce a function of 1 Parameter. And then we can then take it back to it's original form using `curry`. `curry` and `uncurry` are inverse functions and hence form an Isomorphism.

By the way, functions of 3 or more Parameters can be `uncurried` multiple times until they take only a single Parameter. Like in the this made up function:

```
isExactMember :: ∀ k v. Ord k => k -> v -> Map k v -> Boolean
isExactMember k v m = ... ❶

un :: ∀ v k. Ord k => Tuple k v -> Map k v -> Boolean ❷
un = uncurry isExactMember

unun :: ∀ k v. Ord k => Tuple (Tuple k v) (Map k v) -> Boolean ❸
unun = uncurry un
```

❶ Details don't matter here.

❷ The result of uncurrying once. We now have 2 Parameters. Notice that the first 2 Parameters of `isExactMember` are now Tuple-ized.

❸ The result of uncurrying a second time. We now have only 1 input Parameter. Notice that the first 2 Parameters of `un` are now Tuple-ized.

Let's return to our 2 `member` functions in their `uncurried` form:

```
member :: ∀ a. Ord a => Tuple a (Set a) -> Boolean
member :: ∀ k v. Ord k => Tuple k (Map k v) -> Boolean
```

Now ALL of our example functions take 1 Parameter and produce a Boolean result.

Based on this, create a Type that embodies the idea of a Predicate. HINT: the Type will contain a function of 1 Parameter that returns a Boolean.

Try it before reading the solution.

15.4. Code for a `Predicate` Type

While it might not be intuitive just yet, writing this Type is fairly simple:

```
data Predicate a = Predicate (a -> Boolean)
```

Since we've been studying Functors and Polarity, we might happen to look at the function contained in the Predicate Type in that light. The Monomorphic Parameter, `Boolean`, is in Positive Position since it's an OUTPUT.

The Polymorphic Parameter, `a`, is in Negative Position, meaning that we cannot write a `Functor` Instance, but instead can write a `Contravariant` Instance.

But before we write the `Contravariant`, let's first write a helper function called `runPredicate` that will run the embeded function for us. You'll need to extract the function from `Predicate` then call it.

Use the following to test your code:

```
log "-----------------------------------" ❶
log $ show $ runPredicate (Predicate odd) $ 10 ❷
log $ show $ runPredicate (Predicate odd) $ 11 ❸
```

❶ Helpful for delimiting our output since they're all `Booleans`.

❷ Prints `false`.

❸ Prints `true`.

Give it a try before reading on.

15.5. Code for `runPredicate`

Let's write the Type Signature first:

```
runPredicate :: ∀ a. Predicate a -> a -> Boolean
```

We take a `Predicate a` and a value `a` and return the result of that Predicate.

Notice that `runPredicate` is general, i.e. it will work with any `Predicate`. This is why we use the Polymorphic Parameter `a`.

Next, we write the function:

```
runPredicate :: ∀ a. Predicate a -> a -> Boolean
runPredicate (Predicate f) x = ??? ❶
```

❶ We destructure `Predicate` using Pattern Matching to extract or unwrap the function.

Next, we simply just call that function:

```
runPredicate :: ∀ a. Predicate a -> a -> Boolean
runPredicate (Predicate f) x = f x
```

15.6. Writing `Contravariant` Instance for `Predicate`

We are now ready to write the `Contravariant` Instance.

First, add the following code to your test code:

```
log "----------------------------------"  ❶
log $ show $ runPredicate (cmap (_ + 1) (Predicate odd)) 10  ❷
log $ show $ runPredicate (cmap (_ + 2) (Predicate odd)) 10  ❸
log $ show $ runPredicate ((_ + 1) >$< (Predicate odd)) 10  ❹
log $ show $ runPredicate ((_ + 2) >$< (Predicate odd)) 10  ❺
```

❶ Helpful for delimiting our output since they're all Booleans.

❷ Prints true.

❸ Prints false.

❹ Prints true.

❺ Prints false.

Remember that (>$<) is the Binary Operator alias for cmap.

Notice that these tests cmap the function over the Predicate to augment the INPUT value BEFORE the Predicate function is executed.

You'll also need to add these imports to keep the compiler happy:

```
import Data.Functor.Contravariant (cmap, (>$<))
```

Now write the Contravariant Instance for Predicate. Consult Pursuit to refresh your memory on Contravariant and look back at the previous Chapter for examples of writing such instances before moving on to the next section.

Also, keep in mind that cmap's mapping function maps INPUTS.

15.7. Code for Contravariant Instance for Predicate

The instance code is very simple for Predicate. All we have to do is use cmap's mapping function to map the INPUT to Predicate's function. This can be accomplished using Function Composition:

```
import Data.Functor.Contravariant (class Contravariant, cmap, (>$<))  ❶

instance contravariantPredicate :: Contravariant Predicate where
  cmap f (Predicate g) = Predicate (g <<< f)  ❷
```

❶ Import the Type Class Contrvariant.

❷ We call f first to map the INPUT to something g expects.

Now run the test code and see what you get.

You should get:

```
[error] Some of your project files import modules from packages that are not
in the direct dependencies of your project.
To fix this error add the following packages to the list of dependencies in
your config:
- contravariant
```

Add `contravariant` to `spago.dhall` and try again.

15.8. Folds and Moore Machines

Before we code Profunctors, let's first model Folds in a Type, much like we did with `Predicates`.

The first step to modeling anything is to make sure we fully understand the thing we're trying to model. Many times we can get a good overview by first look at the salient attributes of the thing we're trying to model. So let's list out the attributes of a Fold-Left Operation:

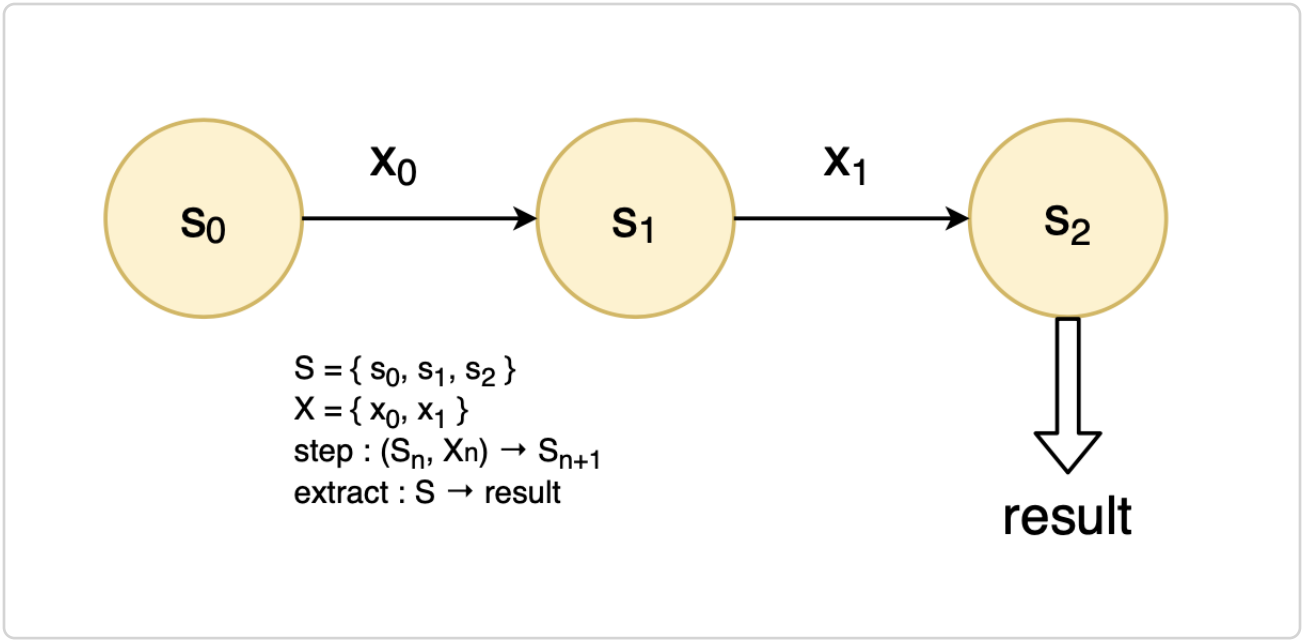
```
s0 :: s              -- initial State
x :: a               -- input value
step :: s -> a -> s -- function called at each step
```

When we're finished with a `foldl` or `foldr`, we are left with the Final State of the operation. We don't always want the full State but maybe a portion of it, e.g. we may have a Tuple in the State but we only want the `fst` of that Tuple since `snd` was just a working value.

Let's add a function called `extract` to extract the final result we're interested in. Realize that this function can always be `identity` when we want the Full State:

```
s0 :: s              -- initial State
x :: a               -- input value
step :: s -> a -> s -- function called at each step
extract :: s -> r    -- function to extract result from Final State
```

Here's a diagram that shows the execution of a Fold:



Let's breakdown the diagram:

- `S` is the set of States
- `X` is the set of Inputs
- `step` takes a current State, S_n and current Input, X_n and produces next State, S_{n+1}
- `extract` takes a State and extracts the result we're interested in

It turns out that this model is very close to something called a ***Moore Machine***. Moore Machine is a definition for a ***Finite State Machine***.

State Machines are commonly used when writing code dealing with Protocols, Machine Control, Process Control, in fact, nearly anything we code can be modeled as a State Machine. However, some applications lend themselves to State Machine more than others.

These State Machines get their name from ***Edward F. Moore***, whose 1956 paper, "*Gedanken-experiments on Sequential Machines*", introduced the idea.

Here's the definition of a Moore Machine from Wikipedia:

```
A Moore machine can be defined as a 6-tuple (S, S₀, Σ, Λ, T, G) consisting of
the following:

A finite set of states S
A start state (also called initial state) S₀ which is an element of S
A finite set called the input alphabet Σ
A finite set called the output alphabet Λ
A transition function T : S × Σ → S mapping a state and the input alphabet to
the next state
An output function G : S → Λ mapping each state to the output alphabet
```

Although the use of Greek letters can feel a bit foreign, if we look carefully we can see the parallels with our Fold definition:

```
s0 :: s              -- initial State
x :: a               -- input value
step :: s -> a -> s  -- function called at each step
extract :: s -> r    -- function to extract result from Final State
```

```
s = A finite set of states S
s0 = A start state (also called initial state) S₀ which is an element of S
a = A finite set called the input alphabet Σ
r = A finite set called the output alphabet Λ
step = A transition function T : S × Σ → S
extract = An output function G : S → Λ
```

Notice how Types in a program are Sets in the Mathematical definition. That's because Types have Inhabitants which are Sets of Values of that Type.

Let's look at the Type Signature for `foldl` to see how Moore Machines relate:

```
foldl :: ∀ a s f. Foldable f => (s -> a -> s) -> s -> f a -> s
```

The first Parameter to `foldl` is the `step` function in our model (Moore: transition function, $T : S \times \Sigma \to S$). Notice how Moore's definition is just a Cartesian Product of S and Σ, which is just `foldl's` function `uncurried`, `Tuple s a -> s`:

```
s = A finite set of states S
a = A finite set called the input alphabet Σ
Tuple s a = (s, a) ∈ S × Σ
```

The second Parameter to `foldl` is the initial State (Moore: start state, S_0).

`foldl's` third Parameter is a `Foldable`, which contains the input values (Moore: input alphabet, Σ)

And the final Parameter to `foldl` is the Final State, which doesn't have a direct parallelism to a Moore Machine. Our `foldl` simply returns the full Final State, but Moore has an output function $G : S \rightarrow \Lambda$ that extracts out the result. This corresponds to our `extract` function in our Fold Model from earlier.

Since a Moore Machine is slightly more general than Folds, we're going to model a Moore Machine:

```
data Moore s a b = Moore ???
```

Here `s` is the State Type, `a` is the Input Type and `b` is the Output Type.

Let's add the Initial State to our definition:

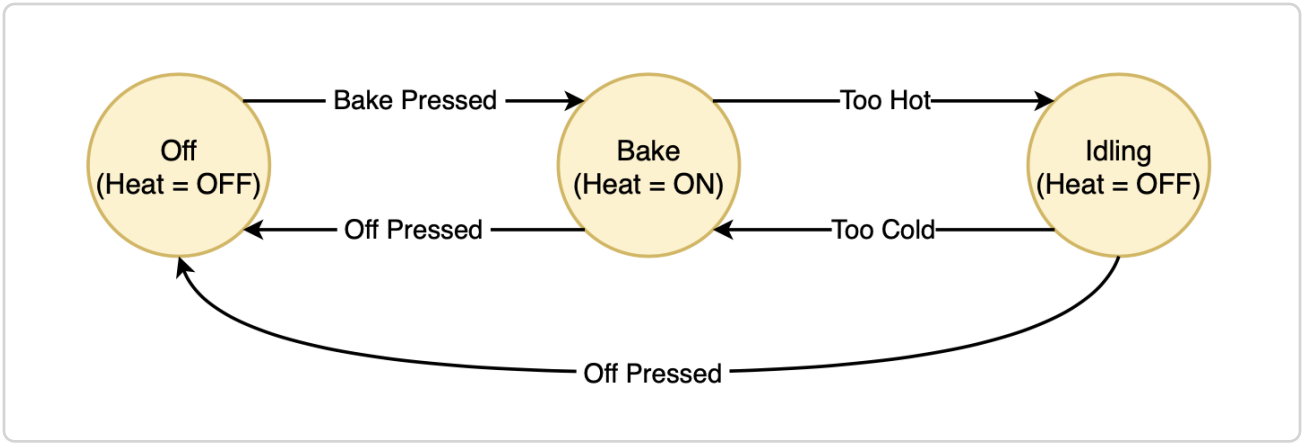
```
data Moore s a b = Moore s ???
```

Looking back at the definition for a Moore Machine, we're only missing the Output Function and the Transition Function. So let's first add the Output Function, i.e. a function that takes a State and produces an Output:

```
data Moore s a b = Moore s (s -> b) ???
```

Remember, `s` is the State Type and `b` is the Output Type. When folding, we'll use this function to produce our final result. But in a Moore Machine, each State has an Output value.

This can be seen in the following State Diagram, which is an overly simplistic control of an Oven:



In this diagram, our Output is whether Heat is ON or OFF. We can imagine what the code for the Output Function would look like:

```
data OvenState = Off | Bake | Idling

data Heat = HeatOn | HeatOff

outputFn :: OvenState -> Heat ❶
outputFn Off = HeatOff
outputFn Bake = HeatOn
outputFn Idling = HeatOff
```

❶ In this concrete example, `s`, the State, is `OvenState` and `b`, the Output, is `Heat`.

Now, we can model the Transition Function:

```
data Moore s a b = Moore s (s -> b) (s -> a -> s)
```

Add this definition to your code.

And for our Oven, the Transition Function would like:

```
data InputSignal = BakePressed | OffPressed | TooHot | TooCold

transitionFn :: OvenState -> InputSignal -> OvenState ❶
transitionFn Off BakePressed = Bake
transitionFn Bake OffPressed = Off
transitionFn Bake TooHot = Idling
transitionFn Idling TooCold = Bake
transitionFn Idling OffPressed = Off
transitionFn s _ = s ❷
```

❶ Once again `s`, the State, is `OvenState` and `a`, the Input, is `InputSignal`.

❷ Any `InputSignals` received in a State where there's no Transition in the State Diagram, will NOT Transition and simply maintain the current state, `s`.

Compare this Function to our Oven State Machine above before moving on. It's important to see how a State Machine diagram translates easily to code.

Now, let's return to our definition of a Moore Machine:

```
data Moore s a b = Moore s (s -> b) (s -> a -> s)
```

We can see that in the Output Function, `s -> b`, `b` is in Positive Position (OUTPUT) meaning that `Moore s` is Covariant in `b`.

But, looking at the Transition Function is a bit harder, so let's map it out:

```
s -> (a -> s) ❶
-    (  +   ) ❷
     (-    +) ❸
```

❶ Added implied Parentheses.

❷ The function `a -> s` is an OUTPUT and therefore in Positive Position.

❸ `a` is an INPUT so it's in Negative Position.

Overall, `a` is Negative Position TIMES Positive Position for the function resulting in Negative Position. This means that `Moore s` is Contravariant in `a`.

You may be wondering why I keep referring to `Moore s` instead of `Moore`. That's because `Moore s` has 2 Parameters `a` and `b`, which we've determined is Contravariant and Covariant respectfully, meaning we can write a `Profunctor` Instance for `Moore s`.

Remember `Profunctors` only have 2 Type Parameters, but `Moore` has 3 Type Parameters. This is why we need to look at `Moore s a b` as:

```
(Moore s) a b
```

where the `p` in our `Profunctor` Instance will be `Moore s`:

```
instance profunctorMoore :: Profunctor (Moore s) where
```

Finally. After all of that, we can write a `Profunctor` Instance. So please give it a try. Refer to the previous

Chapter and Pursuit to refresh your memory on how to write `Profunctor` instances.

But before you do, you'll have to add the package `purescript-profunctor` to `spago.dhall`. Your file should now look like:

```
{-
Welcome to a Spago project!
You can edit this file as you like.
-}
{ name = "my-project"
, dependencies =
  [ "console"
  , "contravariant"
  , "effect"
  , "foldable-traversable"
  , "integers"
  , "lists"
  , "maybe"
  , "newtype"
  , "nonempty"
  , "prelude"
  , "profunctor" ❶
  , "psci-support"
  , "strings"
  , "tuples"
  ]
, packages = ./packages.dhall
, sources = [ "src/**/*.purs", "test/**/*.purs" ]
}
```

❶ This was added MINUS the `purescript-` prefix.

Don't forget to rebuild using:

```
npx spago build
```

And now when you add the following import, the compiler won't complain:

```
import Data.Profunctor (class Profunctor)
```

Try writing the `Profunctor` instance for `Moore` before continuing.

15.9. Hint for `Profunctor` for `Moore`

If you're struggling to code this, it may be because you don't realize that there is a vast difference between the Type Parameters and the Data Constructor Parameters. For example, we are mapping between Types:

```
Moore s a b   -- before mapping
Moore s c d   -- after mapping
```

But when we destructure and reconstruct the `Moore` data structure, we're looking at the Data Constructor mapping:

```
Moore s (s -> b) (s -> a -> s)    -- before mapping
Moore s (s -> d) (s -> c -> s)    -- after mapping
```

Please do not confuse the Type Parameters with the Data Constructor Parameters. Here's the Type with its Parameters:

```
Moore s a b
```

The Type Parameter `a` is in the second position of the Type, which is VERY different from the Parameter passed to the Data Constructor in the second position:

```
Moore s (s -> b) (s -> a -> a)
```

The second position of the Data Constructor takes a function of Type, `s -> b`. It's easy to confuse these two when coding this. Notice how, `a` is in the second position of the Type Constructor but is used in the third position of the Data Constructor:

```
data Moore s a b = Moore s (s -> b) (s -> a -> s)
```

It's possible to have any Type Constructor Parameter show up in any position of the Data Constructor:

```
data SomeType a b c = SomeType a (c -> a) (a -> b)
```

Also, keep in mind that `Moore` is the NAME of a Type Constructor AND the NAME of a Data Constructor. This can add to the confusion but is so prevalent in codebases in both Haskell and PureScript, it's worth getting use to.

Remember, when we write the specialized Type Signature for `dimap`, we are ONLY concerned with the Type

Constructor Parameters:

```
instance profunctorMoore :: Profunctor (Moore s) where
  dimap :: ∀ a b c d. (c -> a) -> (b -> d) -> Moore s a b -> Moore s c d
```

When we are coding the `dimap` function, we are now ONLY concerned with the Data Constructor Parameters.

To summarize, Type Constructors take Types as Parameters, while Data Constructors take Values, which have a specific Types.

Hopefully, this can help get you over this hump. Give the code another try.

15.10. Code for `Profunctor` for `Moore`

Here's the definition for `Moore` again:

```
data Moore s a b = Moore s (s -> b) (s -> a -> s)
```

And the start of its `Profunctor` Instance:

```
instance profunctorMoore :: Profunctor (Moore s) where ❶
  dimap ::
    ∀ a b c d. (c -> a) -> (b -> d) -> Moore s a b -> Moore s c d ❷
  dimap ??? = ???
```

❶ Once again, we need a Type that only has 2 Parameters. This is why we use `Moore s` here since there are 2 remaining Parameters, i.e. `a` and `b`.

❷ It's not a requirement to write the specialized Type Signature when writing an Instance Method, but many times this is helpful as a reference when writing the code.

Next, let's work on the left-hand `???`:

```
instance profunctorMoore :: Profunctor (Moore s) where
  dimap :: ∀ a b c d. (c -> a) -> (b -> d) -> Moore s a b -> Moore s c d
  dimap f g (Moore s0 output transition) = ??? ❶
```

❶ `s0` is the Initial State, `output` is the Output Function and `transition` is the Transition Function.

All we've done here is name our mapping functions and destructure our `Moore`.

Notice that in the Type Signature for `dimap`, we're going from an `a` (in `Moore s a b`) to a `c` (in `Moore s b`

d), yet f is c -> a. It's backwards, which means that f is Contravariant.

In contrast, we're going from a b (in Moore s a b) to a d (in Moore s b d) and g is b -> d, which matches meaning that g is Covariant.

Next step is to reconstruct a new Moore:

```
instance profunctorMoore :: Profunctor (Moore s) where
  dimap :: ∀ a b c d. (c -> a) -> (b -> d) -> Moore s a b -> Moore s c d
  dimap f g (Moore s0 output transition) = Moore s0 ??? ❶
```

❶ The Initial State is not altered by dimap. This is reenforced by the fact that we're writing a Profunctor Instance for Moore s.

The easiest way to figure out what to do next, is to start writing down our types. We have 2 mapping functions f and g and in this phase, we're trying to map output. So let's write down those types:

```
f :: c -> a
g :: b -> d
output :: ???
```

So what's the Type of output here? If you said a, please go back and reread the Hint section. To determine the Type, we look to the Data Constructor:

```
Moore s (s -> b) (s -> a -> s)    -- 1st Parameter is Initial State
                                  -- 2nd Parameter is the Output Function
                                  -- 3rd Parameter is the Transition Function
```

So we can now add that type to our list:

```
f :: c -> a
g :: b -> d
output :: s -> b
```

At this point it should be clear that we cannot use f to map output since they share no common Types. But g looks quite promising. We can take the output of output, which is of Type b, and pass it to g which takes a b:

```
f :: c -> a
g :: b -> d
output :: s -> b
g <<< output :: s -> d ❶
```

❶ Remember, we're calling `output` first and then passing it's return value of Type `b` to `g`.

The composition of the mapping function `g` with `output` gives us the Type we expect, i.e. `b` has been mapped to `d` and now we can update our code:

```
instance profunctorMoore :: Profunctor (Moore s) where
  dimap :: ∀ a b c d. (c -> a) -> (b -> d) -> Moore s a b -> Moore s c d
  dimap f g (Moore s0 output transition) = Moore s0 (g <<< output) ??? ❶
```

❶ Call the original `output` function then transform the `b` to a `d` using `g :: b -> d`.

Finally, we're ready to transform the Transition Function, `s -> a -> s`. But the `a` isn't the first Parameter like we saw in the previous Chapter. We can't simply compose `f` with `transition` since the Types won't line up.

If `transition` had a Type Signature of `a -> s`, then we'd be able to compose it with `f`. To fix this, we'll just have to pass an `s` to `transition`. By partially applying `s` to `transition` we get:

```
transition s :: a -> s
```

Let's next add the mapping functions and their Types to this list:

```
transition s :: a -> s
f :: c -> a
g :: b -> d
```

It's clear that `g` cannot be used to map `transition s` since they have no Types in common. But, `f` looks promising. So let's compose them and add it to our list:

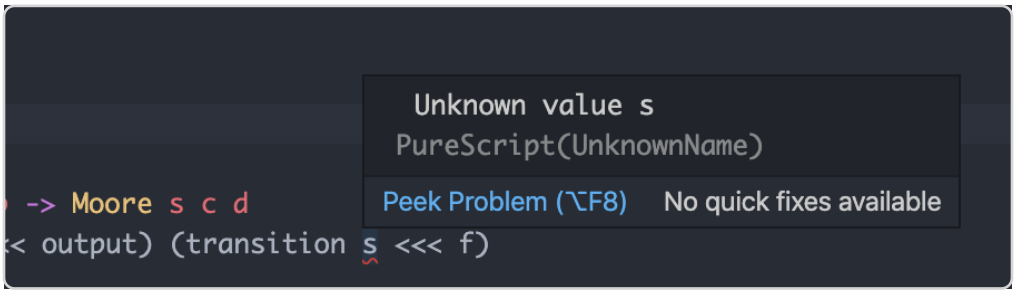
```
transition s :: a -> s
f :: c -> a
g :: b -> d
transition s <<< f :: c -> s ❶
```

❶ Unlike the mapping of `output`, we call `f` FIRST, which is because `f` is Contravariant, i.e. maps INPUTS.

Even though we don't have signature we want, i.e. `s → c → s` just yet, let's see what happens if we just add it to our function:

```
instance profunctorMoore :: Profunctor (Moore s) where
  dimap :: ∀ a b c d. (c -> a) -> (b -> d) -> Moore s a b -> Moore s c d
  dimap f g (Moore s0 output transition) =
    Moore s0 (g <<< output) (transition s <<< f) -- COMPILER ERROR!!
```

Can you see what we're missing? No? Well, maybe the compiler error can help:



The problem is that there is no `s`, so let's give it one:

```
instance profunctorMoore :: Profunctor (Moore s) where
  dimap :: ∀ a b c d. (c -> a) -> (b -> d) -> Moore s a b -> Moore s c d
  dimap f g (Moore s0 output transition) =
    Moore s0 (g <<< output) (\s -> transition s <<< f) ❶
```

❶ Our new `transition` function takes an `s` and then transforms the INPUT using `f` and then calls the old `tranistion` function.

Let's breakdown why this works by looking at the Types. First, here's the `transition` function for both versions of `Moore`:

```
transition :: s -> a -> s  -- Moore s a b ❶
transition :: s -> c -> s  -- Moore s c d ❷
```

❶ Before mapping.

❷ After mapping.

So we need to map from the first one to the second one. Let's look at the Types involved in mapping the `transition` function:

```
transition               :: s -> a -> s  -- Moore s a b
s                        :: s
transition s             :: a -> s
f                        :: c -> a
transition s <<< f       :: c -> s
\s -> transition s <<< f :: s -> c -> s  -- Moore s c d
```

If that last step is confusing, maybe seeing it with the implied Parentheses will help:

```
transition s <<< f       :: c -> s
\s -> transition s <<< f :: s -> (c -> s)
```

This was a tough one since we had to dig down into the second Parameter. Then we had to put that Parameter into a Lambda function. This technique is so common that you should keep in mind.

Had the Type Signature been `s -> s -> a -> s` we would require 2 Parameters in the Lambda and we'd pass them both to `transition`:

```
\s1 s2 -> transition s1 s2 <<< f
```

Notice we maintain the order of `s1` and `s2`.

15.11. Modeling Folds with Moore Machines

We started modeling Folds and wound up taking a detour into Moore Machines since they were nearly identical. The only real difference was the `output` function. While that's not officially part of the `foldl` or `foldr` signature, we tend to extract the result from the Final State on the very next line of our code in the case where we don't want the full Final State.

In the case of a Fold where we want the entire State to be our result, we can simply use `identity` as our `output` function.

There are a couple approaches we could take to model Folds now that we have `Moore`. Let's explore them, but first here's `Moore` again:

```
data Moore s a b = Moore s (s -> b) (s -> a -> s)
```

The first option is to just use this Type. That's not a bad solution but it could be confusing for someone who reads our code and doesn't know what a Moore Machine is. We could fix that by educating them with a well placed comment in our code.

This isn't a bad solution and we'll use it as our "back-pocket solution". This is what I call solutions that I'll use if all other possible solutions fail. Many times, I'm looking for a better solution than my back-pocket one. But at least I know this will work.

In our search for a better solution, we're looking for something that says Fold directly, if possible.

One possible solution is to use a Type Alias:

```
type FoldL = Moore
```

This would simply make them equivalent. Not bad, but it's not fully modeling a `foldl`. It goes beyond that with the `output` function, which may or may not be something we want. We could have 2 flavors:

```
type FoldL' = Moore ❶
type FoldL s a = Moore s (s -> s) (s -> a -> s)  ❷
```

❶ Full-featured Fold.

❷ Modeling Fold exactly with the `output` function constrained to `identity`. Remember, there's only 1 function that can be written with the Type Signature, `s -> s` and that's `identity`.

This isn't bad. But we cannot write a `Contravariant` Instance for it since it's a Type Alias and not a Type.

We could make it a `newtype` to fix that:

```
newtype FoldL s a = FoldL (Moore s a b) -- COMPILER ERROR!!
```

But now we don't have a way to define `b`. Trust me when I say, we don't want to go down this path. Here be dragons.

Another approach is to wrap `Moore`:

```
newtype FoldL s a b = FoldL (Moore s a b)
```

But now we have to wrap and unwrap to get access to `Moore` with no possible benefit except for naming.

It's clear that all of these solutions are terrible. A Moore Machine is just slightly more than a Fold and therefore all of these solutions are just avoiding the name `Moore`.

So we're going to forego this path and simply go with `Moore` for the same reason we call a `Functor` a Functor and not Mappable. We benefit by thinking of it as a Moore Machine and all that comes with that even if we're just going to use it for Folds.

15.12. Writing a Moore Machine that Folds

Instead of writing a Function that takes an `Array` of `Ints` and adds them up, simply write a Function that models or defines this operation using `Moore`. Eventually, we'll write a Function to execute this definition.

Call the Function, `addr` and give it a try before reading ahead.

15.13. Code for `addr`

As usual, we start with the Type Signature:

```
--      Moore  s   a   b
addr :: Moore Int Int Int
```

Here the first Parameter to `Moore`, our State, is an `Int` since it's the Accumulator of our sum.

The second Parameter is an `Int`, i.e. each element of our `Foldable` is an `Int`.

The third Parameter is also an `Int`, i.e. after adding up all the `Ints` we expect an `Int` as our result.

Before, we add the code, let's look at the `Moore` Data Constructor:

```
data Moore s a b = Moore s (s -> b) (s -> a -> s)
```

So we'll have to write something like:

```
addr :: Moore Int Int Int
addr = Moore ??? ??? ???
```

The first Parameter to the Data Constructor is the Initial State. That will be `0` for our `addr`.

```
addr :: Moore Int Int Int
addr = Moore 0 ??? ???
```

The second Parameter of the Data Constructor is the `output` function which converts the Final State to our result. But the Final State is the sum, which is exactly what we want. So we don't need to process it, which means we can use `identity` here:

```
addr :: Moore Int Int Int
addr = Moore 0 identity ???
```

The final Parameter is the `transition` function, i.e. the function that's called upon receiving an Input to the State Machine to compute the next State.

In a Fold, the Input is a set of Elements from a `Foldable`, which, in our case, are `Int` values we want to sum up. We'll do this by adding each Element to the current State to produce the next State. In other words, our State is just an Accumulator.

We can accomplish this by using the addition function, `(+)`. This is the function that we will ultimately pass to `foldl` in our Interpreter:

```
addr :: Moore Int Int Int
addr = Moore 0 identity (+)
```

Realize that this function just produces a `Moore`, which is just a definition or instructions on how to Fold `Ints`. We need an Interpreter of this Value to actually perform the operation.

But before we write that, it would be nice if `addr` also worked for `Numbers`. In fact, it would be nice if it worked for anything that supports `(+)`.

Use either Pursuit or the REPL to determine the Type Signature for `(+)`. This should help you make `addr` more general.

Then comment out this version of `addr` in your project and rewrite it to be as general as possible. This version should work with at least `Ints` and `Numbers`. Give it a try before moving on to the next section.

15.14. Code for a general `addr`

If you consulted Pursuit, you'll find that `(+)` is defined in the `Data.Semiring` module. This might clue you into the fact that we want our function to work with all `Semirings`. If you didn't jump to that conclusion but kept looking, you would find that `(+)` is an alias for `Data.Semiring.add` which is a `Semiring` Method.

This also tells you that you want to work with `Semirings`. So let's first rewrite the Type Signature for `addr` remembering to constrain our Type to `Semiring`:

```
addr :: ∀ a. Semiring a => Moore a a a ❶
```

❶ It's easy to forget the `∀ a.` when you first code this. Luckily, the compiler reminds us.

And next we write the code

```
addr :: ∀ a. Semiring a => Moore a a a
addr = Moore 0 identity (+) -- COMPILER ERROR!!
```

Can you see where we went wrong? Turns out, we cannot use `0` since that's the Additive Identity for `Int`. But this Function needs to work with all `Semirings` not just `Int`.

This is why `Semiring` has the value `zero`. A `zero` for `Int` is `0`, but a `zero` for `Number` is `0.0`. Let's fix this by using `a's zero` value:

```
addr :: ∀ a. Semiring a => Moore a a a
addr = Moore zero identity (+) ❶
```

❶ The compiler will use the appropriate `zero` depending on what Type `a` unifies to. If `a` unifies with `Int`, then `zero` will be `0`. If `a` unifies with `Number`, then `zero` will be `0.0`.

Now, we have a general `addr` Function. Next, we need to write the interpreter that will take this Fold Definition and actually perform the `foldl` operation. We're going to call this Function, `runFoldL`.

15.15. Writing `runFoldL`

We can finally add some test code to test the code we've been writing:

```
log "-------------------------------------"
log $ show $ runFoldL addr [1, 2, 3] ❶
log $ show $ runFoldL addr (1.0 : 2.0 : 3.0 : Nil) ❷
```

❶ Prints `6`.

❷ Prints `6.0`.

You're also going to need the following imports:

```
import Data.Foldable (class Foldable, foldl)
import Data.List (List(..), (:))
```

Next, write a Function called `runFoldL` that takes a `Moore`, i.e. a Fold Definition, and a `Foldable` and performs the operation. Take a look at the test code to see how it's used.

Take a swing at this before reading the solution. Even if all you get is the Type Signature, that's better than not trying at all.

15.16. Code for `runFoldL`

First, as always, we write the Type Signature. `runFoldL` will take a `Moore` and a `Foldable` of Type `a` and perform the `foldl` over that Data Structure to produce a final result, `b`:

```
runFoldL :: ∀ s a b f. Foldable f => Moore s a b -> f a -> b
```

Next, let's write half of the function body to destructure the `Moore`. But before we do, it's useful to consult the Data Constructor:

```
data Moore s a b = Moore s (s -> b) (s -> a -> s)
```

That helps us write the following:

```
runFoldL :: ∀ s a b f. Foldable f => Moore s a b -> f a -> b
runFoldL (Moore s0 output transition) xs = ??? ❶ ❷ ❸ ❹
```

❶ `s0` is our Initial State.

❷ `output` takes the Final State and produces our result, `b`.

❸ `transition` is the `foldl` step function.

❹ `xs` is our `Foldable`.

Now, we can write the rightside of the equal sign:

```
runFoldL :: ∀ s a b f. Foldable f => Moore s a b -> f a -> b
runFoldL (Moore s0 output transition) xs =
  output $ foldl transition s0 xs ❶
```

❶ Perform the `foldl` and then use `output` to extract the result from the Final State.

This function can be Eta Reduced by canceling `xs` on both sides of the equal sign:

```
runFoldL :: ∀ s a b f. Foldable f => Moore s a b -> f a -> b
runFoldL (Moore s0 output transition) = output <<< foldl transition s0 ❶
```

❶ Notice how Function Application changes to Function Composition. That's because `foldl transition s0` is short 1 Parameter.

15.17. Leveraging `Moore's Profunctor` Instance

Our `addr` function is pretty useful, but to appreciate the full power of the Profunctor, let's leverage `addr` to write a function called `sizer` that takes a `Foldable` of `String` and takes the length of each `String` and adds them up.

And when it's done, we want it to print out a message telling us the length. Here's some test code to add that

illustrates how `sizer` will be used and its expected output:

```
log $ show $ runFoldL sizer [ "This", "is", "the", "test" ] ❶
```

❶ Prints `"Size is 13"`.

Remember, `sizer` is just a Definition of a Fold. `runFoldL` interprets that Definition and performs the actual Fold.

Also, keep in mind, we want to leverage `addr` and the goal of this section is to experience the power of the `Profunctor` Instance, so chances are we're going to use `dimap` or perhaps just `rmap` or `lcmap`.

Keeping all of this in mind, write `sizer` by leveraging `addr` without looking ahead.

15.18. Hint for `sizer`

When planning to write `sizer` we should first consult the Type Signature for `Moore`:

```
data Moore s a b = Moore s (s -> b) (s -> a -> s) ❶
```

❶ `s` is the State Type, `a` is the Element Type and `b` is the Output Type.

Here's the start of our `sizer` Type Signature:

```
sizer :: Moore ? ? ?
```

The first `?` is the State Type. Since we're leveraging `addr`, we'd expect `sizer's` State Type to match `addr's`. But `addr's` State Type is the Polymorphic Parameter `a`:

```
addr :: ∀ a. Semiring a => Moore a a a
```

But we know that `sizer` will be adding lengths of `Strings`. So `sizer's` State isn't just any `Semiring` but a very specific one, `Int`:

```
sizer :: Moore Int ? ?
```

The next `?` is the Element Type. Looking at our test code:

```
log $ show $ runFoldL sizer [ "This", "is", "the", "test" ] ❶
```

❶ Prints "Size is 13".

we see that we have a Foldable of String. So this Type Parameter is String, to indicate we have String Elements:

```
sizer :: Moore Int String ?
```

The final ? is the Output Type. We know by looking at the expected output of our test code, "Size is 13", that this is also a String.

```
sizer :: Moore Int String String
```

Now that we have the Type Signature figured out, go ahead and write the sizer function remembering the important fact that we're going to do some sort of mapping over the addr.

So we're going to need to map addr to sizer:

```
                        s     a      b
Moore a a a ---> Moore Int String String
```

We know that a unifies with Int giving us:

```
       s   a   b              s     a      b
Moore Int Int Int ---> Moore Int String String
```

This tells us we need to map addr's Output, which is an Int, to sizer's Output, which is a String.

This also tells us that we need to map sizer's Element, which is an String, to addr's Element, which is an Int. Notice how the previous sentence is BACKWARDS from the one before it. That's because this is mapping an INPUT (Contravariant). We want to leverage addr, so we have to map from String to Int before we can safely call addr.

We have specified 2 mappings. The second specification is Contrvariant. Does that sound familiar?

If it doesn't, it should. It's our old friend Profunctor.

Use this new perspective to finish writing the function before reading on.

15.19. Code for sizer

Starting with the Type Signature from the Hint section:

```
sizer :: Moore Int String String
```

We know we're going to use dimap over addr, which will do the summing for us:

```
import Data.Profunctor (class Profunctor, dimap) ❶

sizer :: Moore Int String String
sizer = dimap ??? ??? addr
```

❶ Added dimap to the import list.

Now all we have to do is figure out what our mapping functions are.

Let's look back at the dimap Type Signature:

```
class Profunctor p where
  dimap :: ∀ a b c d. (b -> a) -> (c -> d) -> p a c -> p b d
```

We can see that the first Function is the Contravariant function. We can use it to map the Element (see Hint section).

Let's do that. So, how do we map String to Int? Lots of ways, but sizer is for adding up LENGTHS of Strings. So we'll use the function, length :: String -> Int, to do that mapping:

```
import Data.String (length) ❶

sizer :: Moore Int String String
sizer = dimap length ??? addr ❷
```

❶ Added this import.

❷ Convert String elements that sizer expects into Int elements that addr expects.

Now looking back at dimap, we can see that the second Function is Covariant, i.e. maps OUTPUTS. This is what we'll use to map addr's Output, the sum total, from Int to String.

But how to do that?

Let's look back at the our test code:

```
log $ show $ runFoldL sizer [ "This", "is", "the", "test" ] ❶
```

❶ Prints "Size is 13".

From the annotation, we can see that we need a function that takes the result of adding up all the lengths, which is an `Int`, and displays a message like, `"Size is 13"`. So let's add such a function:

```
sizer :: Moore Int String String
sizer = dimap length (\n -> "Size is " <> show n) addr ❶
```

❶ We use `show` to convert the `Int`, `n`, to a `String`.

Test out your code to see if it works.

We have now successfully leveraged `addr` to create `sizer` using `Profunctor's dimap`.

15.20. Power of the Functor

Functors of Functions let us leverage existing functions to create new ones. With Profunctors, specifically `Moore`, we were able to map INPUTS and OUTPUTS to leverage the existing function `addr` to produce `sizer`.

If we didn't have this power, we'd have to write:

```
log $ show $ (\n -> "Size is " <> show n) $ ❶
  runFoldL addr (length <$> [ "This", "is", "the", "test" ]) ❷
```

❶ We take the result of running the Fold and convert it to a `String`. This is analogous to mapping OUTPUTS.

❷ We use `Functor` to map `length` over the `List` of `Strings`. This is analogous to mapping the INPUTS.

instead of:

```
log $ show $ runFoldL sizer [ "This", "is", "the", "test" ]
```

And we'd have to do this extra work every single time we wanted to leverage `addr`.

With the power of Functors, and in particular Profunctors, we can make our lives so much easier by both leveraging existing functions and by making the use of those new functions easy.

Part III: Advanced

Chapter 16. Applicative Functors, Traversables and Alternatives

So far we've seen a few flavors of Functors to perform the following operations:

- mapping a Function over a Value that's in a Context (Functor, BiFunctor)
- mapping a Function over another Function that's in a Context (Functor, Contravariant, Invariant, Profunctor)

In both cases, we have a Function on the outside of the Context applied to either a Value or Function on the inside of the Context.

The one case we've not examined yet is when the Value is on the outside and the Function is on the inside. What does that look like?

Let's start with a Function in a Context:

```
Just (_ * 10) :: Maybe (Int -> Int)
```

Now, we'd like to take some Value, say `20` and apply it to this function and wind up with the result inside of our Context, which here is a `Maybe`. We could write:

```
(\f -> f 20) <$> Just (_ * 10) :: Maybe Int
```

Here our Lambda Expression will be applied to the "value" in the Context, which happens to be a Function. Then `f` will be bound to `(_ * 10)` and then `20` will be applied to it.

So it looks like we can already do this with a `Functor`. But what happens if we have `Just 20` instead of `20`?

Let's try a similar approach:

```
(\f -> f <$> Just 20) <$> Just (_ * 10) :: Maybe (Maybe Int) ❶
```

❶ Yuck. Not what we were hoping for.

We had to `map` twice here. First time was when applying `f` to `Just 20` and then we apply the whole Lambda to `Just (_ * 10)`. This is why we get a `Maybe` in a `Maybe`.

So mapping isn't a great solution here.

Another possibility is to write a special function for `Maybe`:

```
applyToMaybe :: ∀ a b. Maybe (a -> b) -> Maybe a -> Maybe b
applyToMaybe Nothing _  = Nothing
applyToMaybe (Just f) x = f <$> x
```

That works, but when `Either` is our Context, we'd have to write a specially named function for that one too. We've seen this pattern before and it's best handled with a Typeclass.

And PureScript has just the Typeclass for this, called `Apply`:

```
class Functor f <= Apply f where
  apply :: ∀ a b. f (a -> b) -> f a -> f b ❶

infixl 4 apply as <*> ❷
```

❶ Apply the Function in Context, `f (a -> b)`, to a Value in Context `f a`.

❷ The `apply` Binary Operator alias, `<*>`, is infixed and Left-Associative.

Notice that `Apply` is a `Functor`.

It turns out that `Maybe` has an `Apply` instance. So, we can rewrite our code where we had to map twice to use `apply` instead:

```
Just (_ * 10) <*> Just 20 :: Maybe Int ❶ ❷
```

❶ Now we only have 1 `Maybe`.

❷ Results in `Just 200`

Let's look at another way that we can use `apply's` Operator Alias, `<*>`:

```
data Tuple a b = Tuple a b

x :: ∀ a b. Maybe a -> Maybe b -> Maybe (Tuple a b)
x ma mb = Tuple <$> ma <*> mb ❶
```

❶ Huh? The first time I saw this, I was confused. Don't worry, we'll go through it slowly.

So let's take this one step at a time.

First, `ma` is a `Maybe a`. That means we can `map` the Data Constructor, `Tuple`, over it:

```
ma :: Maybe a
Tuple <$> ma :: Maybe (b -> Tuple a b) ❶
```

❶ We've only Partially Applied `Tuple`. It's expecting 2 Parameters but we only gave it 1.

We are now in the situation we spoke of earlier. We have a Function, `b -> Tuple a b`, in a Context, `Maybe`, and we want to apply `mb` to it. Realize that `mb` is just `b` in the SAME Context.

Now we can use `apply`:

```
ma :: Maybe a
Tuple <$> ma :: Maybe (b -> Tuple a b)
mb :: Maybe b
Tuple <$> ma <*> mb :: Maybe (Tuple a b) ❶
```

❶ This time we're using `<*>` instead of `<$>`, so we don't wind up with a `Maybe` wrapped in another `Maybe`.

This is a common example of what using `apply` looks like. First, we `map` the first Parameter with `<$>` and then all the remaining Parameters are applied using `<*>`.

Here's another example using our old friend `Threeple`:

```
data Threeple a b c = Threeple a b c

threeple :: ∀ a b c.
  Maybe a -> Maybe b -> Maybe c -> Maybe (Threeple a b c)
threeple mx my mz = Threeple <$> mx <*> my <*> mz ❶
```

❶ As soon as we have our function in a Context, viz. `Maybe`, we can start using `<*>`.

If `Threeple` was already in the Context, i.e. `Maybe Threeple` then we could use `<*>` right away:

```
data Threeple a b c = Threeple a b c

threeple :: ∀ a b c.
  Maybe a -> Maybe b -> Maybe c -> Maybe (Threeple a b c)
threeple mx my mz = Just Threeple <*> mx <*> my <*> mz ❶
```

❶ We don't use `<$>` since our function, `Threeple`, which is a Data Constructor, is already in a Context.

Compare this version with the previous. The difference is subtle.

Now let's take the last version of `threeple` and make the Context a Parameter, i.e. let's generalize it:

```
data Threeple a b c = Threeple a b c

threeple :: ∀ a b c f. f a -> f b -> f c -> f (Threeple a b c) ❶
threeple mx my mz = ??? Threeple <*> mx <*> my <*> mz ❷
```

❶ We've replaced `Maybe` with the Type Parameter, `f`.

❷ `???` is where the Data Constructor for Type `f` goes.

The problem here is we don't know how to wrap `Threeple` in the Context, `f`. We know `f`, but `f` is a Type not a Data Constructor. We need a Data Constructor, e.g. `Just`, to wrap `Threeple`, but all we know is the Type, e.g. `Maybe`.

And once again, the way to do this without knowing the Type up front is to use a Typeclass.

It turns out that we can use the `Applicative` Typeclass in PureScript to accomplish this:

```
class Apply f <= Applicative f where
  pure :: ∀ a. a -> f a
```

Here `pure` takes a Pure Value and wraps it into a Context. Notice that `Applicative` is an `Apply` which means it's also a `Functor`.

Now, let's finish writing `threeple`:

```
data Threeple a b c = Threeple a b c

threeple :: ∀ a b c f. Applicative f => ❶
  f a -> f b -> f c -> f (Threeple a b c)
threeple mx my mz = pure Threeple <*> mx <*> my <*> mz ❷
```

❶ We've constrained `f` to be an `Applicative`.

❷ We replaced `???` from before with `pure`.

So as long as `f` is an `Applicative`, we can wrap it in the Context, `f`, by using `pure` and then use `apply` or it's Binary Operator, `<*>`.

16.1. Applicative in Haskell

In Haskell, `Applicative` is defined as:

```haskell
-- HASKELL Appicative
class Functor f => Applicative f where ❶
  pure :: a -> f a ❷
  (<*>) :: f (a -> b) -> f a -> f b ❸
```

❶ In Haskell, Constraints on classes have the same forward arrow as Constraints on functions.

❷ In Haskell, `∀ a` is implied.

❸ In Haskell, Binary Operators do not have to be aliased.

Notice how Haskell's `Applicative` has the same Typeclass Methods as PureScript's `Apply` and `Applicative`.

This is a minor difference, but we'll see that in PureScript, the decision was made to be a bit more granular with both `Applicative` and `Monads`.

Hopefully, this example is more proof that moving from PureScript to Haskell will be pretty easy compared to learning Functional Programming in the first place. Sort of like moving from Java to C#.

16.2. Applicative Functors in Category Theory

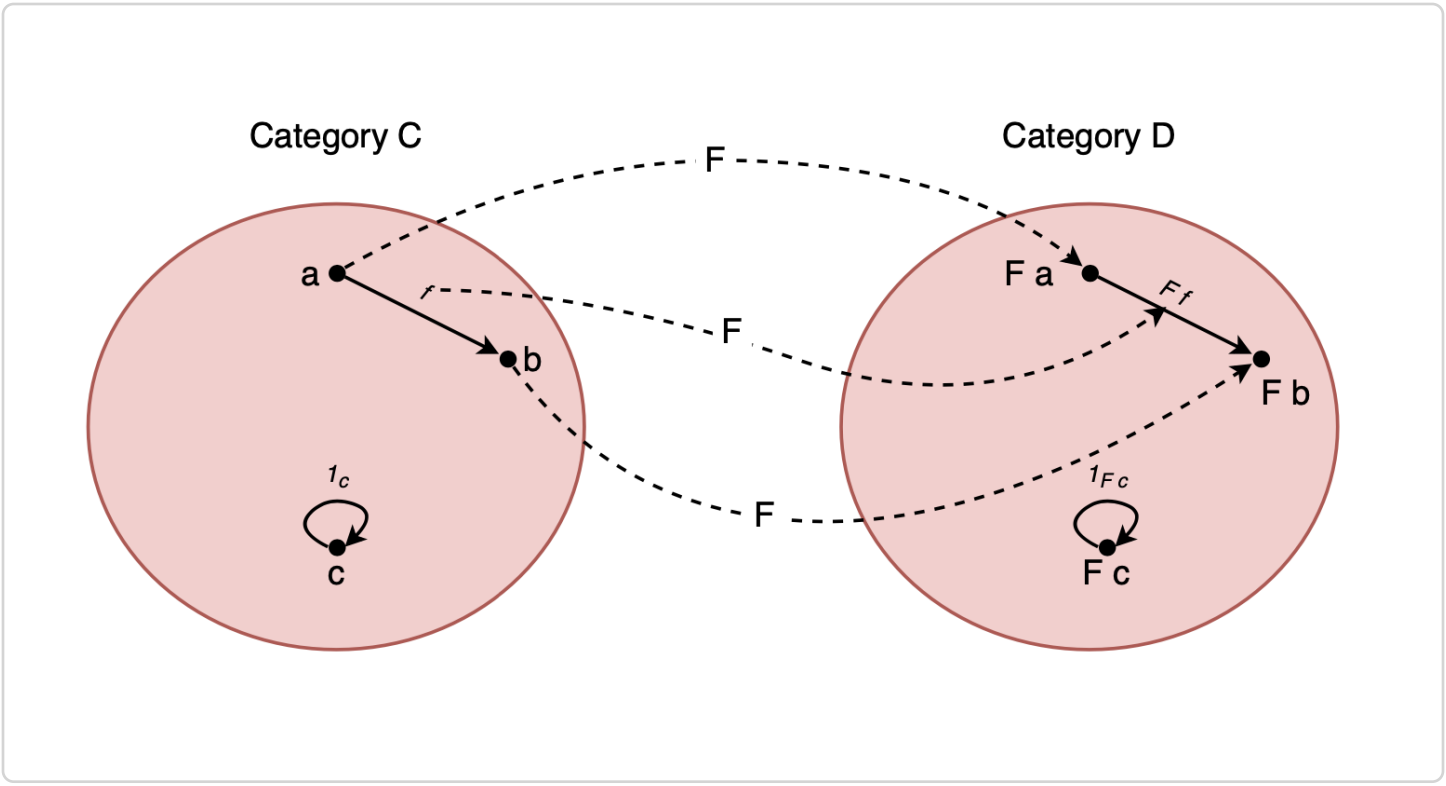
Unlike Functor, the Applicative Functor has it's origins in programming. But it does round out the Categorical Functor.

If you remember, `Functor` only maps Morphisms:

```haskell
class Functor f where
  map :: ∀ a b. (a -> b) -> (f a -> f b) ❶
```

❶ We've added the implied Parentheses here.

But in Category Theory, Functors map BOTH Morphisms and Objects:



In the diagram, Functor `F` maps `a` to `F a`.

And now with `pure` from `Applicative` we finally have a way to map Objects:

```
class Apply f <= Applicative f where
  pure :: ∀ a. a -> f a
```

`pure` maps `a` into the Context, `f a`, which mirrors the way Functors map Objects between Categories.

Since an `Applicative` is also a `Functor`, it also maps Morphisms which makes it the Programming equivalent of Functors in Category Theory.

16.3. Apply Laws

Having the definition of `Apply` close to hand, will make understanding its single, yet complex, law much easier:

```
class Functor f <= Apply f where
  apply :: ∀ a b. f (a -> b) -> f a -> f b
```

Here's the one and only law for `Apply`:

```
(<<<) <$> u <*> v <*> w = u <*> (v <*> w)     [Associative Composition]
```

Here `u` and `v` are Functions and `w` is a Value. But what's not obvious is that `u`, `v` and `w` are in some Context, i.e. in an `Apply`, which is a `Functor`.

So let's rewrite it so we can keep track of the fact that everything is in a Context.

Let's replace `u` with `C f` where `C` is the Context (e.g. the Functor `Maybe`) and `f` is the contained Function.

And let's replace `v` with `C g` and `w` with `C x`:

```
(<<<) <$> C f <*> C g <*> C x = C f <*> (C g <*> C x)
```

Now, `f` and `g` are Pure Functions and `x` is a Pure Value. All, of which, are in Context `C`.

On the left, we apply the Composition Operator to `C f` and `C g` before apply it to `C x`.

On the right, we apply `C g` to `C x` then apply `C f` to that.

This isn't very different from the Composition Law for functions:

```
(f <<< g) x = f (g x)
```

Let's keep writing it until it looks more like the law above:

```
((<<<) f g) x         = f (g x)
(((<<<) $ f) g) x     = f (g x)
(((<<<) $ f) $ g) x   = f (g x)
(((<<<) $ f) $ g) $ x = f (g x)
(((<<<) $ f) $ g) $ x = f $ (g x)
(((<<<) $ f) $ g) $ x = f $ (g $ x)
```

And now the resemblance should be obvious:

```
((  (<<<)  $    f)  $    g)  $    x =   f  $  (  g  $    x)
((C (<<<) <*> C f) <*> C g) <*> C x = C f <*> (C g <*> C x)
```

Notice that `<*>` is equivalent to `$` for Pure Functions.

To prove this law, let's start by first reducing the left-hand side and then the right until they are equal:

```
(<<<) <$> C f <*> C g <*> C x = C f <*> (C g <*> C x)   -- Given
C (f <<< _) <*> C g <*> C x = C f <*> (C g <*> C x)     -- Applying <$>
C (f <<< g) <*> C x = C f <*> (C g <*> C x)             -- Applying <*>
C ((f <<< g) x) = C f <*> (C g <*> C x)                 -- Applying <*>
C (f (g x)) = C f <*> (C g <*> C x)                     -- Composition
C (f (g x)) = C f <*> (C (g x))                         -- Applying <*>
C (f (g x)) = C (f (g x))                               -- Applying <*>
```

16.4. Applicative Laws

Here is the `Applicative` definition for reference:

```
class Apply f <= Applicative f where
  pure :: ∀ a. a -> f a
```

Here are the `Applicative` laws:

```
pure identity <*> x = x                                 [Identity]
pure (f x) = pure f <*> pure x                          [Homomorphism]
u <*> pure x = pure (_ $ x) <*> u                       [Interchange]
```

These laws are an eyeful. So let's look at them one at a time. First, Identity:

```
pure identity <*> x = x                                 [Identity]
```

This simply says that wrapping `identity` in a Context and then applying it to `x`, which is also in a Context, has no effect on `x`.

This should make sense since:

```
pure identity <*> ≡ identity <$>
```

giving us:

```
identity <$> x = x                                      [Identity]
```

which says that mapping `identity` into a Context doesn't affect the Context. This is just the Functor Law for `Identity`.

Next, let's look at the Homomorphism Law. But before we do, let's remind ourselves of what a Homomorphism is:

```
h (x • y) = h(x) ✱ h(y)
```

Here `h` is the Homomorphic Function that preserves structure. With Monoid Homomorphisms, we saw how it preserved the Monoidal Structure, i.e. after we apply `h`, we expect the new Binary Operator to behave like a proper Monoid.

With `Applicative`, we expect the Function Application Operator, `$`, to behave similarly.

So, let's rewrite the law to see that better:

```
pure (f x) = pure f <*> pure x                    [Homomorphism]
pure (f $ x) = pure f <*> pure x                  -- Added redundant $
```

Hopefully, you can now see how this matches the Homomorphism formula:

```
pure (f $ x) = pure f <*> pure x
 h   (x • y) =  h  (x) ✱   h  (y) ❶
```

❶ Remember that the Binary Operators do not have to be the same on both sides of the equal sign. And in this case, they are not. On the left side, the Function Application Operator is `$` and it's `<*>` on the right.

Here `pure` is our Homomorphism that preserves the operation of Function Application.

We can apply the function, `f` to `x` using `$` and then `pure` the result, and that's equivalent to using `pure` on both `f` and `x` first, then applying them using the `Applicative` apply function, `<*>`.

Finally, let's look at the Interchange Law:

```
u <*> pure x = pure (_ $ x) <*> u                 [Interchange]
```

By looking at this law, it should be clear that `<*>` is NOT Commutative. But we can interchange the function and value if we do a tiny bit of magic. That magic is the function `(_ $ x)`.

Looking at just the magic bit, we can see that it's just higher-order function that expects a function and will pass `x` to it.

This isn't special since this trick works with Pure Functions and Pure Values too:

```
f $ x = (_ $ x) $ f ❶ ❷
```

❶ The last `$` is redundant here, but will be helpful for comparing with `Applicative` (see below).

❷ In your mind, simply replace `_` with `f` to see how these are equal.

The only complexity is that in the `Applicative`, everything is wrapped in a Context. Let's see the Pure version next to the `Applicative` one:

```
f  $       x =      (_ $ x)  $  f
u <*> pure x = pure (_ $ x) <*> u
```

Let's look at each pure part with it's corresponding Contextual part, one at a time.

- `f` is a function while `u` is a function in a Context.
- `$` is the Pure function application operator while `<*>` is function application within Contexts.
- `x` is a value while `pure x` is a value in a Context.
- `(_ $ x)` is a higher-order function while `pure (_ $ x)` is that same function in a Context.

Notice how `<*>` is analogous to `$` again. Just like we saw in the `Homomorphism Law`.

This law pretty much states that function application, while not Commutative, can be written reversed using `(_ $ x)`, which is no different for Pure function application, i.e. functions and values NOT in a Context.

16.5. Applicative Instance for `Maybe`

We can write an `Apply` and `Applicative` instance for `Maybe`:

```
data Maybe a = Nothing | Just a

instance functorMaybe :: Functor Maybe where
  map f (Just x) = Just (f x)
  map _ Nothing  = Nothing

instance applyMaybe :: Apply Maybe where
  apply (Just f) x = f <$> x ❶
  apply Nothing _  = Nothing

instance applicativeMaybe :: Applicative Maybe where
  pure = Just ❷ ❸
```

❶ Once we've unwrapped `f`, we can map it over `x`, which is wrapped.

❷ `Just` is the only logical choice here. If we had used `Nothing`, we would throw away the Pure Value. Plus it would break the `Identity` law.

❸ Written in Point-free Notation where the Pure Value is not named.

Notice that we wrote the `Apply` instance in terms of `map`. That because:

```
pure f <*> x = f <$> x ❶
```

❶ `f` is Pure, whereas `x` is in a Context.

The equality, which only applies to `Applicatives`, says that we can write `apply` in terms of `map`, which is what we did.

But it also says, we can write `map` in terms of `apply`:

```
data Maybe a = Nothing | Just a

instance functorMaybe :: Functor Maybe where
  map f x = pure f <*> x ❶

instance applyMaybe :: Apply Maybe where
  apply (Just f) x = f <$> x ❷
  apply Nothing _  = Nothing

instance applicativeMaybe :: Applicative Maybe where
  pure = Just
```

❶ Changed to leverage the `Applicative` instance. We used `<*>` from `Apply` and `pure` from `Applicative`.

❷ BEWARE OF INFINITE LOOPS!

Just changing our implementation to `map` is not sufficient. It caused an infinite loop. `map` is defined in terms of `apply` and `apply` is defined in terms of `map`.

This is fine for Math, but for Software it means it'll infinitely recurse and never terminate.

So, let's fix that by writing a proper `apply`:

```
data Maybe a = Nothing | Just a

instance functorMaybe :: Functor Maybe where
  map f x = liftA1 -- equivalent to pure f <*> x ❶

instance applyMaybe :: Apply Maybe where
  apply (Just f) (Just x) = Just (f x) ❷
  apply _        Nothing  = Nothing ❷
  apply Nothing  _        = Nothing

instance applicativeMaybe :: Applicative Maybe where
  pure = Just
```

❶ Here's we've changed our previous implementation to use the helper function `liftA1` from the `Control.Applicative` module. It's equivalent to `pure f <*> x`.

❷ We had to add these 2 lines to handle the 2 `Just` cases.

Notice that there is no free lunch. We have to explicitly write the `map` function or the `apply`. We can write one in terms of the other, but not both.

When looking at these 2 versions, I think I like writing `apply` in terms of `map` since it's few lines and less complex.

My personal experience is that writing `apply` is usually more difficult than writing `map`. So, I usually leverage `map` in the `apply` function. Your mileage may vary.

16.6. Applicative Instance for Product Types

`Tuple` is a Product Type, which slightly complicates the process of writing an `Applicative` Instance. To see why, let's imagine that `Tuple` already has an `Apply` Instance.

So what would its `pure` implementation look like? Something like the following:

```
instance applicativeTuple :: Applicative (Tuple a) where
  pure x = Tuple ??? x
```

It's clear what to do with the `x` since `Applicative` has Kind `Type -> Type`, i.e. it has 1 Type Parameter.

But what do we put in the first position of the `Tuple`, which is a value of Type `a`?

It's impossible for us to manufacture an `a` without knowing more about it. This means that we'll need to constrain `a` so we can safely construct one. But how?

If it's not obvious how to do this, maybe we can glean some clues by considering the `Apply` Instance for

`Tuple`:

```
instance applyTuple :: Apply (Tuple a) where
  apply (Tuple x f) (Tuple y z) = Tuple ??? (f z)
```

Once again, we know what to do with the second part of the Tuple. We APPLY `f` to `z`, i.e. `f :: a -> b` applied to `z :: a` produces `f z :: b`.

But, we have NO idea what to do with the `x` and `y`, i.e. the values in the first position. Seems wrong to keep one and toss the other. And even if that was the right thing to do, which takes precedence?

Instead, we're looking for a solution that combines these values somehow.

```
instance applyTuple :: Apply (Tuple a) where
  apply (Tuple x f) (Tuple y z) = Tuple (x ??? y) (f z) ❶
```

❶ Here `???` is some kind of Binary Operator that combines `x` and `y` somehow.

And for `pure`, we're looking for a solution that creates an EMPTY value for the first position:

```
instance applicativeTuple :: Applicative (Tuple a) where
  pure x = Tuple ??? x ❶
```

❶ `???` is some EMPTY value of Type `a`.

Can you think of something we've seen before that would fit these requirements?

We need something that has an EMPTY value and a Binary Operator to combine 2 values of Type `a` to produce a result of Type `a`.

At this point, I hope you're screaming out `Monoid`. It has everything that we need. So let's add a `Semigroup` Constraint to `Apply` and a `Monoid` Constraint to `Applicative`:

```
instance applyTuple :: Semigroup a => Apply (Tuple a) where ❶
  apply (Tuple x f) (Tuple y z) = Tuple (x <> y) (f z) ❷

instance applicativeTuple :: Monoid a => Applicative (Tuple a) where ❸
  pure x = Tuple mempty x ❹
```

❶ We constrain `a` to be a `Semigroup` since we're only using `<>` in this instance.

❷ We combine the values in the first position using the `append` operator.

❸ We constrain `a` to be a `Monoid` since we're using `mempty`.

❹ We use the `Monoid's` EMPTY value for the value in the first position.

Now, let's see our newly minted `Applicative` in action:

```
Tuple "abc" (_ * 10) <*> Tuple "123" 42 -- Tuple "abc123" 420
pure (_ * 10)        <*> Tuple "abc" 42 -- Tuple "abc"    420 ❶
Tuple "abc" (_ * 10) <*> pure 42        -- Tuple "abc"    420 ❶
```

❶ Notice that when we apply with a Pure Function or a Pure Value, that the first value of the `Tuple` remains untouched. That's because of the `mempty` that `pure` uses.

Now that we've solved the problem with `Tuple`, we should be able to use that experience to extrapolate to `Threeple`:

```
data Threeple a b c = Threeple a b c
```

We now have to worry about combining 2 Types, `a` and `b`. Can you think of the solution?

More Monoid Constraints:

```
instance applyThreeple ::
  (Monoid a, Monoid b) => Apply (Threeple a b) where ❶
    apply (Threeple v w f) (Threeple x y z) =
      Threeple (v <> x) (w <> y) (f z) ❷

instance applicativeThreeple ::
  (Monoid a, Monoid b) => Applicative (Threeple a b) where ❶
    pure x = Threeple mempty mempty x ❸
```

❶ We hold `a` and `b` constant.

❷ We combine the `a` values and the `b` values.

❸ We have 2 empty values.

16.7. Applicative Instance for Sum Types

With Sum Types, we had none of the problems we encountered writing `Applicative` Instances for Product Types.

Here's our implementation for `Maybe` again:

```
data Maybe a = Nothing | Just a

instance functorMaybe :: Functor Maybe where
  map f (Just x) = Just (f x)
  map _ Nothing  = Nothing

instance applyMaybe :: Apply Maybe where
  apply (Just f) x = f <$> x
  apply Nothing _  = Nothing

instance applicativeMaybe :: Applicative Maybe where
  pure = Just
```

It's pretty straightforward. No need for Constraints or `Monoids`.

Here's another Sum Type and its `Applicative` Instance:

```
data Somethings a b c
  = Something1 a
  | Something2 b
  | Something3 c

instance functorSomethings :: Functor (Somethings a b) where
  map f (Something3 x) = Something3 $ f x
  map _ x              = x ❶

instance applySomethings :: Apply (Somethings a b) where
  apply (Something3 f) x = f <$> x
  apply _              x = x ❷

instance applicativeSomethings :: Applicative (Somethings a b) where
  pure x = Something3 x ❸
```

❶ Mapping over `Something1` or `Something2` has no effect.

❷ Applying with a `Something1` or `Something2` has no effect.

❸ We use `Something3` to wrap our Pure Value so that we can use it to `apply` to another `Something`. If we had put it in `Something1` or `Something2` then it would have no effect in `apply` since we don't use the contextual function in those cases.

While slightly more complex than `Maybe`, this implementation is fairly simple. But that's not always the case.

Here's a Sum Type that illustrates that unfortunate fact:

```
data Things a b c
  = Thing1 a
  | Thing2 b c -- PRODUCT
```

Now, even though `Things` is a Sum Type, `Thing2` constructs a Product.

Looking at the `Things` Type, we can see that we're going to need to hold `a` and `b` constant. We're also going to have to constrain `b` to be a `Monoid` thanks to the Data Constructor, `Thing2`:

```
instance functorThings :: Functor (Things a b) where ❶
  map _ (Thing1 x)   = Thing1 x
  map f (Thing2 x y) = Thing2 x $ f y

instance applyThings :: Monoid b => Apply (Things a b) where ❶ ❷
  apply (Thing2 x f) (Thing2 y z) = Thing2 (x <> y) (f z)
  apply _            (Thing1 x)   = Thing1 x
  apply (Thing1 f)   _            = Thing1 f

instance applicativeThings :: Monoid b => Applicative (Things a b) where ❶ ❸
  pure x = Thing2 mempty x
```

❶ `a` and `b` held constant.

❷ `b` constrained to be a `Monoid` for the `Apply` Instance.

❸ Since `Applicative` is also an `Apply`, it must abide by the same constraints as its Superclass. So we must constrain `b` here too.

Just because we start with a Sum Type, doesn't always mean it'll be smooth sailing. We have to be careful to examine the underlying Data Constructors, which may construct Products.

16.8. An Applicative Example

Let's imagine that we want to have a function that builds a Full Name out of name components. So we write the following:

```
fullName :: String -> String -> String -> String
fullName first middle last = first <> " " <> middle <> " " <> last
```

But, then we later realize that this doesn't quite workout since our database can have NULL values for the first, middle and last names. So the Parameters aren't `Strings` but instead they're `Maybe Strings`.

So then we modify our function:

```
fullName :: Maybe String -> Maybe String -> Maybe String -> Maybe String ❶
fullName (Just first) (Just middle) (Just last) =
            Just $ first <> " " <> middle <> " " <> last
fullName Nothing      _             _           = Nothing ❷
fullName _            Nothing       _           = Nothing ❷
fullName _            _             Nothing     = Nothing ❷
```

❶ To accommodate missing values, this takes `Maybe Strings` and return `Maybe String`.

❷ Any time we're missing a value, this function fails.

And while this is a bit verbose, it does what we want... sort of. It would be nice to know WHICH element is missing.

Let's change the return value from a `Maybe` to an `Either String`:

```
fullName :: Maybe String -> Maybe String -> Maybe String
            -> Either String String ❶
fullName (Just first) (Just middle) (Just last) =
               Right $ first <> " " <> middle <> " " <> last
fullName Nothing _       _       = Left "first name missing" ❷
fullName _       Nothing _       = Left "middle name missing" ❷
fullName _       _       Nothing = Left "last name missing" ❷
```

❶ Now it returns an `Either String String`.

❷ Returns an error with a message telling us which field is missing.

Okay, we got what we wanted, but we had to brute force it and we ventured far afield from the simplicity of our original function.

Let's at least write a function that takes `Strings` like the one we had before and have it call or error checking function, `fullName`:

```
fullName' :: String -> String -> String -> String
fullName' first middle last =
  case fullName (Just first) (Just middle) (Just last) of
    Left err -> err ❶
    Right name -> name
```

❶ Returning the error `String` as the full name is clunky, but since we'll never pass `Nothing` to `fullName`, we know that this case can never happen. Still, not the best solution.

We accomplished what we set out to do, but it was a lot of work, produced verbose complex functions and the result is inflexible.

The reason this solution is unsatisfactory is because we wrote the first thing that popped into our heads. This is very common when you're working under a deadline and you don't have time to think. But, this isn't that case. We have time to think.

So let's try a bit more thinking. Looking back on our first attempt, we can learn some things. For example, we lost the simple function in our bid to have error checking.

Also, when we wrote a version of that simple function, it was burdened with the complexity of the error checking version, i.e. we now had to check for `Left` and `Right`.

Let's try this again, but this time we'd like to keep our original, simple, function. So let's start there:

```
fullName :: String -> String -> String -> String
fullName first middle last = first <> " " <> middle <> " " <> last
```

Next, we'd also like to have a function that will be the error checking version:

```
fullNameEither :: Maybe String -> Maybe String -> Maybe String
                  -> Either String String
fullNameEither = ???
```

But this time, we want this function to call our simple version.

Instead of writing code inside `fullNameEither` to check for missing fields, let's factor that out and put it in its own function. This is good for many reasons:

- It decomposes the problem.
- It means we can use the code elsewhere in our program.
- It means that we can later change out the validation for another validation function.

We're going to call our validation function `errIfMissing`:

```
errIfMissing :: Maybe String -> String -> Either String String ❶
errIfMissing Nothing err = Left err
errIfMissing (Just s) _  = Right s ❷
```

❶ The ordering of the Parameters is so that we can use `errIfMissing` in infixed notation as you will soon see.

❷ Disregard the error reason `String`, since it's NOT missing.

This function takes a `Maybe String` and if it's `Nothing`, then returns the second Parameter (wrapped in `Left`), which is the error reason. Otherwise, it returns `Right` of the `String`.

Now, we're going to call `fullName` with one Parameter at a time but only AFTER that Parameter has been passed through our validation function, `errIfMissing`.

Unfortunately, `errIfMissing` returns an `Either String String` but `fullName` expects `Strings` as Parameters:

```
fullName :: String -> String -> String -> String
```

But we can map `fullName` over `Either String String` since the second Parameter of `Either String String` is `String`.

So, let's at least do that for the first parameter to see where that takes us:

```
fullNameEither :: Maybe String -> Maybe String -> Maybe String -> Either
String String
fullNameEither first middle last =
  fullName <$> (first `errIfMissing` "First name must exist") ❶
  ???
```

❶ We map `fullName` over `Either String String`, which will result in a Partially Applied Function in a Context, viz. `Either`.

We're not done yet. This doesn't compile and for good reason. The Types don't match:

```
 fullName <$> (first `errIfMissing` "First name must exist")
  :: Either String (String -> String -> String)
```

At this point, we now have a Partially Applied function in a Context, viz. `Either`. Notice that it's waiting for 2 more `Strings`, i.e. `middle` and `last`.

This is where `Applicatives` shine.

So let's use `<*>` to pass `middle` and `last` to our enclosed function, but first we'll pass those values through our validation function, `errIfMissing`:

```
fullNameEither :: Maybe String -> Maybe String -> Maybe String
               -> Either String String
fullNameEither first middle last =
  fullName <$> (first `errIfMissing` "First name must exist")
           <*> (middle `errIfMissing` "Middle name must exist")
           <*> (last `errIfMissing` "Last name must exist")
```

This approach is far better than our previous one. We kept our simple function simple and leveraged it from our validating version. Our validation logic is also factored out into a helper function.

Here's our final implementation:

```
errIfMissing :: Maybe String -> String -> Either String String
errIfMissing Nothing err = Left err
errIfMissing (Just s) _  = Right s

fullName :: String -> String -> String -> String
fullName first middle last = first <> " " <> middle <> " " <> last

fullNameEither :: Maybe String -> Maybe String -> Maybe String
               -> Either String String
fullNameEither first middle last =
  fullName <$> (first `errIfMissing` "First name must exist")
           <*> (middle `errIfMissing` "Middle name must exist")
           <*> (last `errIfMissing` "Last name must exist")
```

16.9. Applicative Effects

In an earlier Chapter, Semantics was discussed regarding Types, in particular, what exactly does it mean to use `Maybe` or `Either`. We concluded that `Maybe's` Implicit Behavior was to short-circuit. That's because it was dictated by the Type's structure.

In contrast, `Either` also short-circuits but the Type didn't dictate that behavior, but instead was a conscious choice, which is why we called it an Explicit Behavior.

While I prefer the more nuanced look at these attributes of our Types, they tend to get lumped into a single term, ***Effects***. The trouble with this term is that it's poorly defined and overloaded. I suspect this is for historical reasons.

In the early days of Haskell, for example, they had no way of doing Effects. They literally could only do pure function calls. Effects were solved by using Monads, which strangely enough existed in the language before Applicatives.

And since all Monads are Applicatives, and many Functors can be made into both Applicatives and Monads, there became a need to identify the "effect" of the `Maybe` Monad, which is to short-circuit.

So be careful when it's used to make sure to understand whether they're talking about the behavior of a Type or a real-world side-effect, e.g. writing data to a database.

Keeping this in mind, let's look at another example of using Applicatives to combine a list of Effects. Let's imagine we map a function over a `List` of values and that function returns an Applicative, e.g. a `Maybe`.

Since the function returns a `Maybe`, we can safely conclude that that function could fail. And now we want to get a final result from that `List` by combining the results into a single result of success or failure. We'd also like to be able to do the same with `Either`.

But instead of writing it just for success and failure Applicatives, we're going to write a general combining function called `combineList`.

Here's its Type Signature:

```
combineList :: ∀ a f. Applicative f => List (f a) -> f (List a)
```

In the case where `f` is `Maybe`, we'll get a `Maybe (List a)`.

And if `f` is `Either a` then we'll get an `Either a(List b)`.

Realize that the "effect" of `Maybe` and `Either` are to short-circuit. Other `Applicatives` could have different "effects".

But with `Maybe`, we'd expect that if there are any errors in our `List`, i.e. `Nothing`, that we'll get a final result of an error, i.e. `Nothing`.

And the same with `Either`. If there's a single error in the list, i.e. `Left`, we'll get a `Left`.

Let's remind ourselves of `Maybe's Apply` Instance:

```
instance applyMaybe :: Apply Maybe where
  apply (Just f) x = f <$> x
  apply Nothing _  = Nothing ❶
```

❶ Applying a `Nothing` to anything will result in `Nothing`.

From this code, it should be easy to see that if we combine a `List` of `Maybes` with `<*>`, we'll get a final result of `Nothing` if any of the elements of that `List` are `Nothing`. This is `Maybe's` short-circuit "effect" in action.

Now we'll finish writing `combineList`. First, the base case:

```
combineList :: ∀ a f. Applicative f => List (f a) -> f (List a)
combineList Nil       = pure Nil
```

If given an empty `List`, we simply wrap it up in the Context, using `pure`.

Next, we write the recursive case:

```
combineList :: ∀ a f. Applicative f => List (f a) -> f (List a)
combineList Nil       = pure Nil
combineList (x : xs)  = Cons <$> x <*> combineList xs ❶
```

❶ `Cons` is a Data Constructor that takes a value and a `List` and produces a `List`.

Remember, we have a `List` of values of Type `a` that are in some Context, `f`. So we cannot pass them directly to `Cons` but must use `<$>` and `<*>`.

If this is still a brain bender, let's look at the recursive case one step at a time. Pay close attention to the Types:

```
Cons :: ∀ a. a -> List a -> List a
```

Here `Cons` takes a value and a `List` and produces a `List`, after placing that value at the head.

Next step:

```
x :: f a
Cons <$> x :: f (List a -> List a)
```

Once we mapped `Cons` over the Applicative, `x`, we're stuck with a function in a Context. That means it's time to break out apply, `<*>`:

```
combineList xs :: f (List a)
Cons <$> x <*> combineList xs :: f (List a)
```

And finally, we're applying `combineList xs` to the function in the Context resulting in a `List a` in Context `f`.

Let's see how our new function works. First with `Maybe`:

```
import Data.List (fromFoldable) ❶

combineList (fromFoldable [Just 1, Just 2, Just 3])
  -- (Just (1 : 2 : 3 : Nil))
combineList (fromFoldable [Just 1, Nothing, Just 3])
  -- Nothing
```

❶ We import a helper function to convert any `Foldable` to a `List`. This is convenient since the syntax for an `Array` is simpler than for `List`.

Notice that a single `Nothing` in the `List`, short-circuits and produces `Nothing`.

Just in case this still seems like Black Magic, we're going to look at a simpler version of this example one step at a time.

Here's our function:

```
combineList :: ∀ a f. Applicative f => List (f a) -> f (List a)
combineList Nil       = pure Nil ❶
combineList (x : xs)  = Cons <$> x <*> combineList xs ❷
```

We're going to do Algebraic Substitution and Function Application to see how each step of the computation process works:

```
combineList (fromFoldable [Just 1, Just 2])       -- Given
combineList (Just 1 : Just 2 : Nil)               -- Apply fromFoldable
Cons <$> Just 1 <*> combineList (Just 2 : Nil)    -- Substitute into ❷
Just (Cons 1) <*> combineList (Just 2 : Nil)      -- Apply <$>
Just (Cons 1) <*> (Cons <$> Just 2 <*> combineList Nil)
                                                  -- Substitute into ❷
Just (Cons 1) <*> (Just (Cons 2) <*> combineList Nil)
                                                  -- Apply <$>
Just (Cons 1) <*> (Just (Cons 2) <*> pure Nil)    -- Substitute into ❶
Just (Cons 1) <*> (Just (Cons 2 Nil))             -- Apply <*>
Just (Cons 1 (Cons 2 Nil))                        -- Apply <*>
Just (1 : 2 : Nil)                                -- Rewrite with :
```

Now let's look at `Either`:

```
import Data.List (fromFoldable)

combineList (fromFoldable [Right 1, Right 2, Right 3])
  -- (Right (1 : 2 : 3 : Nil))
combineList (fromFoldable [Right 1, Left "err", Right 3, Left "err2"])
  -- (Left "err")
```

And we see that the FIRST `Left` short-circuited to produce `Left "err"`. The rest of the `List` was effectively ignored.

It's important to understand that it was still processed since PureScript is a Strict language. In Haskell, a Lazy language, this would NOT be the true making Haskell more efficient in this case.

16.10. Functors vs Applicatives with Effects

Remember, that based on the Functor Laws, mapping over a `Just` or a `Right` will always produce a `Just` or a `Right`, respectively:

```
(5 * _) <$> Just 10  = Just 50
(5 * _) <$> Right 10 = Right 50
```

This is NOT the case with `Applicative`:

```
Just (5 * _) <*> Nothing     = Nothing
Right (5 * _) <*> Left "err" = Left "err"
```

Here we started with a `Just` and applied a `Nothing` resulting in `Nothing`. And we also turned a `Right` into a `Left`.

From this, we can say the following:

- Functors CANNOT by themselves have Effects, i.e. once a `Just` always a `Just`. Once a `Nothing`, always a `Nothing`.
- Applicatives CAN have Effects, i.e. a `Just` can become a `Nothing` and a `Right` a `Left`.

Both `map` and `apply` are applying a Function to the Value in the Context. The difference being that with `apply`, the Function is also in a Context. So, there is an opportunity to have 2 different Data Structures, e.g. a Function in a `Just` and instead of a Value we have `Nothing`, or a Function in a `Right` and a Value in a `Left`.

These disparate Data Structures must be combined somehow to produce a SINGLE Data Structure. We have no choice but to choose one over the other in some logic and useful fashion. And in this case, since the Implicit Behavior of `Maybe`, i.e. its "effect", is to short-circuit, we'll prioritize `Nothings` over `Justs` to maintain this behavior:

```
Just (5 * _) <*> Nothing = Nothing ❶
Nothing      <*> Just 2  = Nothing ❶
Just (5 * _) <*> Just 2  = Just 10 ❷
Nothing      <*> Nothing = Nothing ❸
```

❶ We prioritized `Nothing` over `Just`.

❷ No need to choose since both are `Just`.

❸ No need to choose since both are `Nothing`.

16.11. Examples of Applicative Effects

Like `Maybe`, `Either's` "effect" is Short-circuit:

```
instance applyEither :: Apply (Either a) where
  apply (Left f)      _     = (Left f) ❶
  apply     _     (Left x)  = (Left x)
  apply (Right f) (Right x) = Right (f x)
```

❶ Notice that we match the left side first. This means that the FIRST error will be kept. We saw this behavior earlier with our `combineList` example.

If `Left` is combined in any way, it will result in a `Left` favoring the left side of the Binary Operator, i.e. the FIRST `Left`.

The "effect" for an `Array` and `List` is Multiple Values:

```
Tuple <$> [10,20] <*> [3, 4]
  -- [(Tuple 10 3),(Tuple 10 4),(Tuple 20 3),(Tuple 20 4)]
```

This example returns all Combinations of the 2 `Arrays`, i.e. a Cartesian Product. Notice that it combines from left to right in each `Array`, e.g. the first element of the resulting `Array` is `Tuple (10, 3)`, which is the leftmost value of both `Arrays`. And the last element of the resulting `Array` is `Tuple (20, 4)`, which is the rightmost value of both `Arrays`.

16.12. Applicative Effects and Commutativity

An `Applicative` is Commutative IFF (if and only iff):

```
f <$> x <*> y = flip f <$> y <*> x
```

This says that if we apply Parameters `x` and `y` to `f` and it is equal to flipping the Parameters to the function and then applying `x` and `y` in reverse, then `<*>` is Commutative.

Notice how this test works for `Maybe`:

```
(+) <$> Just 3 <*> Just 9        -- Just 12
flip (+) <$> Just 9 <*> Just 3   -- Just 12

(-) <$> Just 3 <*> Just 9        -- Just -6
flip (-) <$> Just 9 <*> Just 3   -- Just -6 ❶

(+) <$> Just 3 <*> Nothing       -- Nothing
flip (+) <$> Nothing <*> Just 3  -- Nothing

(+) <$> Nothing <*> Just 4       -- Nothing
flip (+) <$> Just 4 <*> Nothing  -- Nothing
```

❶ Notice that `<*>` is Commutative even though `(-)` is not. This is because we flipped BOTH the Parameters and the Function.

But this test fails for `Either`:

```
(+) <$> Left "err1" <*> Left "err2"       -- Left "err1"
flip (+) <$> Left "err2" <*> Left "err1"  -- Left "err2"
```

And it fails for `Array` (or `List`) too:

```
(+) <$> [2,3] <*> [7, 13]        -- [9,15,10,16]
flip (+) <$> [7, 13] <*> [2,3]   -- [9,10,15,16]
```

16.13. Traversables

Let's consider a common scenario that we will encounter from time to time. First, we have an `Array` of `Ints` that we want to map over with what is called an ***Effectful*** function, i.e. a function that does NOT produce a Pure Value, but a Value in a Context:

```
half :: Int -> Maybe Int ❶
half x = if even x then Just (x `div` 2) else Nothing

map half [2, 4, 6]      -- [Just 1, Just 2, Just 3]

map half [2, 3, 4]      -- [Just 1, Nothing, Just 2]
```

❶ A non-effectful version of `half` would simply return `Int`.

But we don't want an `Array` of `Maybes`. We want the `Ints` but ONLY if we had no failures. Sound familiar?

We can use our old friend `combineLists` to accomplish this:

```
import Data.List (fromFoldable)

half :: Int -> Maybe Int
half x = if even x then Just (x `div` 2) else Nothing

combine = combineLists . fromFoldable ❶

map half [2, 4, 6]            -- [Just 1, Just 2, Just 3]
combine $ map half [2, 4, 6]  -- Just [1, 2, 3] ❷

map half [2, 3, 4]            -- [Just 1, Nothing, Just 2]
combine $ map half [2, 3, 6]  -- Nothing ❸
```

❶ A little helper function to reduce code clutter.

❷ Here we get an `Array` since we had zero failures.

❸ Here we get `Nothing` since a computation failed. The "effect" here is that we short-circuited on the error and returned `Nothing`.

Our `combineList` is pretty convenient for this common scenario. In fact, it's so common, that it's been codified and generalized in a Typeclass called ***Traversable***.



A ***Traversable*** is a `Foldable` `Functor` that can be traversed while accumulating results and effects in an `Applicative`.

Here's the PureScript TypeClass for `Traversable`:

```
class (Functor t, Foldable t) <= Traversable t where
  traverse :: ∀ a b m. Applicative m => (a -> m b) -> t a -> m (t b)
  sequence :: ∀ a m. Applicative m => t (m a) -> m (t a)
```

This is pretty abstract, so let's parse it one step at a time.

First, the class definition:

```
class (Functor t, Foldable t) <= Traversable t where
```

The `Traversable` Type, `t`, is both a `Functor` and `Foldable`. In the combineList example, we had an `Array` of values (an `Array is a` `Foldable`) that we mapped over (to `map` we need a `Functor`).

Next, let's look at `traverse`:

```
class (Functor t, Foldable t) <= Traversable t where
  traverse :: ∀ a b m. Applicative m => (a -> m b) -> t a -> m (t b) ❶
```

❶ The variable `m` is used here because `Monads` are `Applicatives` and many `Applicatives` also are `Monads`, but if you ask me, it should be called `f` (it is in Haskell).

`a -> m b` is our effectful mapping function, e.g. `half :: Int -> Maybe Int`.

`t a` is our `Foldable` data structure, e.g. `Array Int`.

`m (t b)` is our final result, e.g. `Maybe (Array Int)`.

But sometimes, we start with a `Foldable` structure with Values already in a Context. That's when `sequence` comes in:

```
class (Functor t, Foldable t) <= Traversable t where
  traverse :: ∀ a b m. Applicative m => (a -> m b) -> t a -> m (t b)
  sequence :: ∀ a m. Applicative m => t (m a) -> m (t a)
```

`t (m a)` is our `Foldable` of Contextual Values, e.g. `Array (Maybe Int)`.

`m (t a)` is our final result, e.g. `Maybe (Array Int)`.

`sequence` is just `traverse` without the `mapping` function, therefore:

```
sequence = traverse identity
```

In other words, calling `sequence` is identical to calling `traverse` where the mapping function is `identity`, which leaves all values intact.

Look carefully at `sequence's` Type Signature:

```
sequence :: ∀ a m. Applicative m => t (m a) -> m (t a)
```

Notice how the `t` and the `m` swap places. They do in `traverse` but it's harder to see since there's a function involved. This swapping seems like magic, but it's not. It's really cool and useful.

So imagine you have the following scenario where `Maybe` is our `Applicative`, `m`, and `Array` is our `Foldable` and `Functor`, `t`, and `a` is `Int`:

```
x          :: Array (Maybe Int) -- t (m a)
sequence x :: Maybe (Array Int) -- m (t a)
```

In this case, if any of the Maybe's in the Array are Nothing, then the result will be Nothing. Same goes for:

```
x          :: Array (Either String Int) -- t (m a)
sequence x :: Either String (Array Int) -- m (t a)
```

If any of the Either's in the Array are Left, then the result will be the FIRST Left.

But how does that work?

It works this way for Maybe and Either because of how their apply instances are coded to short-circuit in those cases. At this, point is not really clear how the code for <*> affects sequence or traverse.

But it will be once we write the Traversable Instance for List. It also will become clear to see why there's an Applicative Constraint on m.

16.14. Traversable for List

Writing this instance is hard. If you look at the source code in the PureScript library for help, it won't be very helpful. In fact here it is:

```
instance traversableList :: Traversable List where
  traverse f = map (foldl (flip (:)) Nil)
    <<< foldl (\acc -> lift2 (flip (:)) acc <<< f) (pure Nil) ❶
  sequence = traverse identity
```

❶ lift2 is a helper function that lifts a Pure Function into an Applicative. It does a map and an apply internally. It can be imported from Control.Apply.

This is difficult to understand because it was written using foldl to take advantage of its Tail Recursive nature, which makes it efficient in space but not time since it takes 2 passes over the list.

It needs 2 passes since using foldl produces the List in the reverse order. The map (foldl (flip (:)) Nil) part is to reverse the List INSIDE the Applicative. This reverse step is the second pass over the List.

Instead we're going to write ours using foldr, which will be time efficient but not space. But the real reason is because the foldr implementation is far simpler to understand.

The next area of difficulty is that traversable is too abstract. So, to help us think more clearly about this, we're going to create a simple concrete example to strategize with.

We'll start with a `List` of `x's`:

```
(x1 : x2 : Nil) :: List a
```

The first step is to use our function, `f :: a -> m b`, to convert our Pure Values into Applicatives:

```
(x1 : x2 : Nil)       :: List a     -- starting point
(A(y1) : A(y2) : Nil) :: List (m b) -- 1st step ❶ ❷
```

❶ This is the starting point for `sequence`, i.e. when we already have a bunch of `Applicatives`.

❷ `x1` becomes `y1` to signify that our function changed Types from `a` to `b`.

The next step is the hard part to imagine. We're going to want our take all of these `Applicatives` and apply them to each other resulting in a `List` INSIDE the `Applicative`. To accomplish this, we're going to need `map` the `Cons` operator, `(:)` over the `Applicatives`. Since `(:)` takes 2 Parameters, it'll be Partially Applied.

Then when we `apply` them to each other, it'll build a `List`, but we'll need to do so in a Right-Associative way since `(:)` is Right-Associative. `foldr` will make sure we compute with the correct Associativity.

Let's see a very simple example of how this will work. First, we start with a Value, `y1` in an `Applicative`, `A`, that's been Partially Applied to the `Cons` operator, and we'll `apply` it to an empty `List` in that same `Applicative`:

```
A(y1 : _) <*> pure Nil
```

Now, when we execute the apply, i.e. `<*>`, we get:

```
A(y1 : Nil)
```

This is what we wanted, a `List` in an `Applicative`.

Now let's do a slightly more complex example:

```
A(y1 : _) <*> (A(y2 : _) <*> pure Nil) ❶
```

❶ The Parentheses shows that this computation is Right-Associative. That's because `(:)` is Right-Associative. Good thing we're using `foldr`, since it too is Right-Associative.

Applying the inner Parentheses first yields:

```
A(y1 : _) <*> A(y2 : Nil)
```

And applying one more time yields:

```
A(y1 : y2 : Nil)
```

Once again, we have a `List` in an `Applicative`.

So now let's go back to our strategy with these final steps added:

```
(x1 : x2 : Nil) ❶
(A(y1) : A(y2) : Nil) ❷
(A(y1 : _) : A(y2 : _) : Nil) ❸
A(y1 : _) <*> (A(y2 : _) <*> pure Nil) ❹
A(y1 : y2 : Nil) ❺
```

❶ We start with a `List`.

❷ We apply to each element our function, `f :: a -> m b`, resulting in a `List` of `Applicatives`.

❸ We partially apply `(:)` to each value in the `List`.

❹ We apply each element of the `List` to each other in a Right-Associative way starting with an empty `List` in the same `Applicative`, i.e. `pure Nil`.

❺ This results in a `List` INSIDE the `Applicative`.

Now that we have a strategy, we're ready to write the `Traversable` instance for `List`. First the instance definition:

```
instance traversableList :: Traversable List where
```

Next, we'll write `traversable's` Type Signature specialized for `List` to help us write the code (we can always delete this line later):

```
instance traversableList :: Traversable List where
  traverse :: ∀ a b m. Applicative m => (a -> m b) -> List a -> m (List b)
```

Now, let's use our strategy one step at a time as we code. But, since we know we're going to use `foldr` and that the initial value is `pure Nil`, we should add that first as our skeleton:

```
instance traversableList :: Traversable List where
  traverse :: ∀ a b m. Applicative m => (a -> m b) -> List a -> m (List b)
  traverse f = foldr (\x acc -> ???) (pure Nil) ❶ ❷
```

❶ Written Point-free. The unnamed variable is the List.

❷ I always remember that the Accumulator, acc, is on the right-hand side of the Parameters for the foldr function and on the left-hand side for foldl.

First step from our strategy is to apply the function, f, to each element:

```
instance traversableList :: Traversable List where
  traverse :: ∀ a b m. Applicative m => (a -> m b) -> List a -> m (List b)
  traverse f = foldr (\x acc -> f x) (pure Nil) -- NOT COMPLETE
```

The next step is to partially apply (:) to each element:

```
instance traversableList :: Traversable List where
  traverse :: ∀ a b m. Applicative m => (a -> m b) -> List a -> m (List b)
  traverse f = foldr (\x acc -> (:) <$> f x) (pure Nil) -- NOT COMPLETE ❶
```

❶ We map (:) over f x since f x is an Applicative of Type m b.

The final step of our strategy is to apply each element to each other in a Right-Associative way. We already get the correct Associativity by using foldr.

But in the Fold Function, we only get 1 element from our Foldable at a time and that element needs to be placed at the head of a List. The only List we have is our Accumulator, acc, which is inside of an Applicative.

What we've coded so far is (:) <$> f x, which is a Partially Applied Cons function that already has the head of the List. It too is inside of an Applicative.

We can combine the two by simply using <*>:

```
instance traversableList :: Traversable List where
  traverse :: ∀ a b m. Applicative m => (a -> m b) -> List a -> m (List b)
  traverse f = foldr (\x acc -> (:) <$> f x <*> acc) (pure Nil) ❶
```

❶ The acc is a List, which is the second Parameter for the Cons operator, (:). This is why it's applied AFTER (:) is mapped into the Applicative, f x.

Now that we have our traverse function complete, we're going to write sequence in terms of traverse:

```
import Data.Traversable (class Traversable, sequence, traverse) ❶

instance traversableList :: Traversable List where
  traverse :: ∀ a b m. Applicative m => (a -> m b) -> List a -> m (List b)
  traverse f = foldr (\x acc -> (:) <$> f x <*> acc) (pure Nil)

  sequence :: ∀ a m. Applicative m => List (m a) -> m (List a)
  sequence = traverse identity ❷
```

❶ We need these imports to compile.

❷ Remember that `sequence` is `traverse` with `identity` mapped over the `Functor`.

And we have a complete implementation for `Traversable` for `List`.

Notice that we did the following operations to complete this process:

- `map` from `Functor`
- `foldr` from `Foldable`
- `apply` from `Applicative`

This is why we have the `Foldable` and `Functor` constraints in the `Traversable` Typeclass definition and the `Applicative` constraint in the Method definitions.

16.15. A Few Words on Context

When we have a Pure Value, our computations are Pure, i.e. there's nothing coloring our computations. But when we have a Contextual Value, e.g. `Maybe Int`, computations are going to be affected by the Context that they're performed in.

Let's compare and contrast `Int` with `Either Int Int`:

```
(*) 5 10                    = 50       -- Pure computation
(*) <$> Right 5 <*> Left 10 = Left 10  -- Short-circuited computation
```

The Context, here `Either`, imposes its own rules on our computation, i.e. `Left` will short-circuit all computations.

There is a Context that has ZERO affect on computations, it's appropriately called `Identity`:

```
newtype Identity a = Identity a

instance functorIdentity :: Identity a where
  map f (Identity x) = Identity $ f x

instance applyIdentity ::  Identity a where
  apply (Identity f) x = f <$> x

instance applicativeIdenity :: Applicative Identity where
  pure = Identity
```

Notice how `Idenity` is nothing more than a simple wrapper, with no additional behavior, i.e. it doesn't affect the computation in any way.

Now we can compare it with our earlier examples:

```
(*) 5 10                          = 50          -- Pure computation
(*) <$> Right 5 <*> Left 10       = Left 10     -- Short-circuited
(*) <$> Identity 5 <*> Identity 10 = Identity 50 -- Pure computation
```

In contrast to `Identity`, `Array` will impose the behavior of computing all possible outcomes:

```
(*) <$> [10, 20] <*> [3, 4] = [30, 40, 60, 80]
```

As we talked about before, many people will talk about these as Effects, e.g. the effect of `Array` is non-determinism, i.e. we cannot determine a single answer but instead get all of them. This is sloppy nomenclature and can be quite confusing.

I'd rather talk about the Context and the behaviors imposed on computations by the Context. It is, however, important you understand they are basically the same idea.

16.16. Alt

Remember that Values have Types, e.g. the value `10` has Type `Int`, and the value `identity` has the Type `∀ a. a -> a` (Functions are Values).

Also, remember, that Types have Kind, e.g. the Type, `Maybe`, has Kind `Type -> Type` since it takes 1 Type Parameter, whereas the Type, `Either`, has Kind `Type -> Type -> Type` since it takes 2 Type Parameters.

We've seen that `Functor` takes a Type Parameter, `f`, that has Kind `Type -> Type`, since `f` takes 1 Type Parameter:

```
class Functor f where
  map :: ∀ a b. (a -> b) -> f a -> f b ❶
```

❶ Here `f a` and `f b` show us that `f` takes only 1 Type Parameter.

And `Bifunctor` takes a Type Parameter, `f` that has Kind `Type -> Type -> Type` since `f` takes 2 Type Parameters:

```
class Bifunctor f where
  bimap :: ∀ a b c d. (a -> c) (b -> d) -> f a b -> f c d ❶
```

❶ Here `f a b` and `f c d` show us that `f` takes 2 Type Parameters.

When we learned about `Semigroups`, we learned that it takes a Type Parameter `a` that has Kind `Type` since `a` takes no Type Parameters:

```
class Semigroup a where
  append :: a -> a -> a ❶
```

❶ `a` takes no Type Parameters.

This means that we cannot create a `Semigroup` instance for `List`, since `List` is of Kind `Type -> Type`, i.e. `List` is expecting 1 more Type Parameter to be a complete Type, e.g. `List Int` or `List String`.

But just like we have `Bifunctor` to solve this problem for `Functor`, we have a solution for `Semigroup`. It's called `Alt`:

```
class Functor f <= Alt f where ❶
  alt :: f a -> f a -> f a ❷

infixl 3 alt as <|> ❸
```

❶ `Functor` is a Superclass of `Alt`, i.e. all `Alts` are `Functors`.

❷ `f a` shows us that `f` takes 1 Type Parameter.

❸ The Binary Operator, `<|>`, is very similar to `Semigroup's` operator, `<>`.

Here `Alt's` Type Parameter `f` has Kind `Type -> Type` which means we can write an instance for `List`:

```
instance altList :: Alt List where
  alt = ???
```

But what implementation should we write?

Well, once again, we have to ask ourselves, what are the Semantics of `Alt`. It can't just be the same as `Semigroup` since we already have an instance for `List a`:

```
instance semigroupList Semigroup (List a) where
  append xs ys = foldr (:) ys xs ❶
```

❶ If `xs` is `{x1, x2, x3}`, then `foldr` gives us `(x1 : (x2 : (x3 : ys)))`.

So what does `Alt` mean? We can think of `Alt` as short for ***alternative***. So, if it means anything, it means Choosing. But for `List`, how do we choose?

Let's look at how we choose between any 2 things:

```
a pony or a turtle
a bicycle or a car
the red pill or the blue pill
```

In each case, we see the word `or`. This kind of makes sense when we look at the Binary Operator for `Alt`, `<|>`, which has the vertical bar so commonly associated with a Logical Or.

When picking between 2 things, the concept of `or` makes sense. But what does `or` mean when we have 2 sets of things?

```
a number between 1 and 10, or a number between 50 and 100
green cars or blue cars
dogs or cats
```

Let's look to what we learned as children in Set Theory to help us decide. When I say to you that there are 2 Sets `A` and `B` and the value you are looking for is either in `A` or in `B`, how would you express this fact?

If the answer isn't obvious, then let's look at some concrete examples to help us:

```
A = { 1, 2, 3 }
B = { 10, 11, 12, 13 }
```

Now when I say that I'm thinking of a number that in Set `A` or Set `B`, what would be all the possible guesses that you could make? I ask that very question because the Semantics of `List` and `Array` are non-determinism, i.e. all possible values. So, what are all the possible guesses?

It would be anything from Set `A` ***or*** from Set `B`, i.e the UNION of `A` and `B`:

```
A ∪ B = { 1, 2, 3, 10, 11, 12, 13 }
```

Notice how ***or*** in Set Theory corresponds to the UNION operator.

It turns out that simply appending 2 `Lists` or 2 `Arrays` stays consistent with their Semantics, of all possible values, and with Set Theory:

```
import Control.Alt (class Alt) ❶

instance altList :: Alt List where
  alt = (<>)
```

❶ Notice that the module for `Alt` does NOT start with `Data`. This is most likely because `Alt` is a control flow operation.

This implementation means that the following is true:

```
(1 : 2 : 3 : Nil) <|> (4 : 5 : 6 : Nil) = (1 : 2 : 3 : 4 : 5 : 6 : Nil)
[1, 2, 3] <|> [3, 2, 1] = [1, 2, 3, 3, 2, 1]
```

One thing to note is that `List` and `Array` are NOT Sets and therefore duplicates are NOT removed.

That's great but what about other `Functors`?

Well, let's look at `Maybe` in light of the Semantics of `Alt`, which is a Choice. I'm now going to give you a choice to pick between:

```
Nothing <|> Just 10
```

Which would you choose?

That's probably not the best question I could ask. A better question is what SHOULD you choose? And why?

If we think about it, we could choose `Nothing` but then our choice would ALWAYS be `Nothing` no matter how many "somethings" we could choose from, for example:

```
Just 10 <|> Nothing <|> Just 20 = Nothing ???
```

This MAY be what we want. We use `Nothing` to short-circuit all the time, which is the behavior of computations in the `Maybe` Context.

But we also have to consider the fact that the Semantics of `Alt` is very much like a Logical Or. Let's take a

second look at our original choice in that light:

```
Nothing <|> Just 10
```

Here the first computation failed to produce a value, but the second did not. We could think of `Maybe` as the answer to the question, "did our computation succeed". In that light, a failure as `false` and success as `true`. And if `Alt` is similar to a Logical Or, then a `Nothing` (`false`) ***or*** `Just 10` (`true`) should be `true`, i.e. `Just 10`.

That interpretation seems much more useful than always picking `Nothing`, but how would we handle the following case:

```
Nothing <|> Just 10 <|> Just 11
```

Well, looking at the Associativity of `<|>` we see it's Left-Associative. So let's add in the implied Parentheses:

```
((Nothing <|> Just 10) <|> Just 11)
```

And now we reduce `Nothing <|> Just 10` to `Just 10`:

```
Just 10 <|> Just 11
```

And since Logical Or short-circuits, maybe `<|>` should also. That would mean when choosing between multiple `Maybes`, we pick the first, i.e. leftmost, non-failure value.

This approach seems to be the most useful, while maintaining the Semantics of `Alt`, i.e. Logical Or. It will also be a good approach for `Either`.

Here's `Maybe` with our Semantics:

```
instance altMaybe :: Alt Maybe where
  alt Nothing r = r ❶
  alt l       _ = l ❷
```

❶ If the left side is `Nothing` then we choose the right.

❷ If the left side is `Just a` then we choose it.

Essentially, we're NOT choosing the failure case. Unless of course, `r` is also `Nothing`, but that's okay, because there is no non-failure case to choose from.

And now `Either`:

```
instance altEither :: Alt (Either a) where ❶
  alt (Left _) r = r
  alt l        _ = l
```

❶ Note that `l` and `r` represent the left and right side of the Binary Operator NOT `Left` and `Right` in `Either`.

This is analogous to `Maybe` in that we actively avoid choosing the failure case unless both cases are failure.

Let's see `Alt` in action:

```
import Data.Int.Bits ((.&.))

assertEven :: Int -> Maybe Int
assertEven v = if v .&. 1 == 0 then Just v else Nothing

assertion :: Maybe Int
assertion = assertEven 1 <|> assertEven 2 <|> assertEven 4  -- Just 2
```

We chose the first computation that does not fail, i.e. the first even.

Here `<|>` can be thought of as a Logical Or of sorts:

```
x = (1 == 2) || (3 == 3) || (4 == 100) -- true
```

`4 == 100` never gets evaluated. There is no reason for the evaluation to continue once we get a `true` value, i.e. `3 == 3`.

In this regard, `||` short-circuits on `true` and `<|>` short-circuits on non-failure for `Maybe` and `Either`.

16.17. Alt Laws

When writing `Alt` Instances, we need to keep in mind these laws:

```
(x <|> y) <|> z = x <|> (y <|> z)          [Associativity]
f <$> (x <|> y) = f <$> x <|> f <$> y      [Distributivity]
```

The Associativity Law says that the `Alt` Binary Operator is Associative, which is no different from the `Semigroup` operator, `<>`, which is also Associative.

The Distributivity Law says that mapping over the result of a choice is the same as mapping over each possibility and then choosing.

Let's make sure the Associativity Laws works for `Maybe`:

```
instance altMaybe :: Alt Maybe where
  alt Nothing r = r ❶
  alt l       _ = l ❷
```

We'll do this by using a simple example and by first reducing the left side and then the right using Algebraic Substitution until they are equal:

```
(Nothing <|> Just 10) <|> Just 11 = Nothing <|> (Just 10 <|> Just 11)
                                                                -- Given
Just 10 <|> Just 11 = Nothing <|> (Just 10 <|> Just 11)         -- Subst ❶
Just 10 = Nothing <|> (Just 10 <|> Just 11)                     -- Subst ❷
Just 10 = Nothing <|> Just 10                                   -- Subst ❷
Just 10 = Just 10                                               -- Subst ❶
```

Let's make sure that Distributivity works for `Maybe` with a simple example:

```
(_ * 10) <$> (Nothing <|> Just 2) =
               (_ * 10) <$> Nothing <|> (_ * 10) <$> Just 2  -- Given
(_ * 10) <$> Just 2 =
               (_ * 10) <$> Nothing <|> (_ * 10) <$> Just 2  -- Subst ❶
Just 20 = (_ * 10) <$> Nothing <|> (_ * 10) <$> Just 2       -- Apply <$>
Just 20 = Nothing <|> (_ * 10) <$> Just 2                    -- Apply <$>
Just 20 = Nothing <|> Just 20                                -- Apply <$>
Just 20 = Just 20                                            -- Subst ❶
```

We could just as easily show `Either` using the same process. The only difference is `Nothing` would be replaced with `Left a`.

16.18. Plus

If `Alt` is like `Semigroup` but for a Type of Kind `Type -> Type`, is there a similar Typeclass to `Monoid`?

The answer to that is `Plus`:

```
class Alt f <= Plus f where ❶
  empty :: ∀ a. f a ❷
```

❶ `Alt` is the Superclass of `Plus`, i.e. all `Plus` instances are also `Alt` instances.

❷ Here `empty` is the analogy of `mempty` in `Monoid`.

Let's write a few instances of `Plus`:

```
instance plusList :: Plus List where
  empty = Nil

instance plusArray :: Plus Array where
  empty = []

instance plusMaybe :: Plus Maybe where
  empty = Nothing
```

With `List` and `Array`, these `empty` values are like the Empty Set, which doesn't change the original Set when applying a UNION operation, i.e. `<|>` from `Alt`.

And with `Maybe`, `Nothing` doesn't change the choice result since we don't choose `Nothing` unless we have no other choice.

These act exactly like `mempty` does from `Monoid`, i.e. they are Identity Values.

But what about `Either`?

```
instance plusEither :: Plus (Either a) where
  empty = ??? ❶
```

❶ We need some empty value here that will work with ALL `Either a's`.

Here we are stuck. You might think that we could simply constrain `b` to be a `Monoid` and then write the following:

```
-- COMPILER ERROR!!
instance plusEither :: Monoid b => Plus (Either a) where
  empty = Right mempty
```

We cannot write an empty value for every possible `Either a` since the second Type Parameter of `Either`, which is `b`, cannot be constrained since it's not part of the `instance` definition.

You might think that we could just define it:

```
-- COMPILER ERROR!!
instance plusEither :: ∀ b. Monoid b => Plus (Either a) where
  empty = Right mempty
```

But unfortunately, `∀ b` is not allowed here and even if it was, there's no way to tell the compiler that the `b` we're referring to here is the second Type Parameter of `Either`.

So it is IMPOSSIBLE to write an instance of `Plus` for `Either`.

This is the first time we've encountered such a limitation. But as we move on to higher and higher levels of abstraction, we will encounter situations where writing an instance for our Types either doesn't make sense or break laws or produces undesirable results or simply cannot be done.

16.19. Plus Laws

And of course, there are laws:

```
empty <|> x = x                [Left Identity]
x <|> empty = x                [Right Identity]
f <$> empty = empty            [Annihilation]
```

The 2 Identity Laws are identical to the `Monoid` laws.

And the Annihilation Law tells us that mapping has NO affect on the Empty Value, which is evidenced by `Maybe's empty` being `Nothing`, which when mapped over, produces `Nothing`.

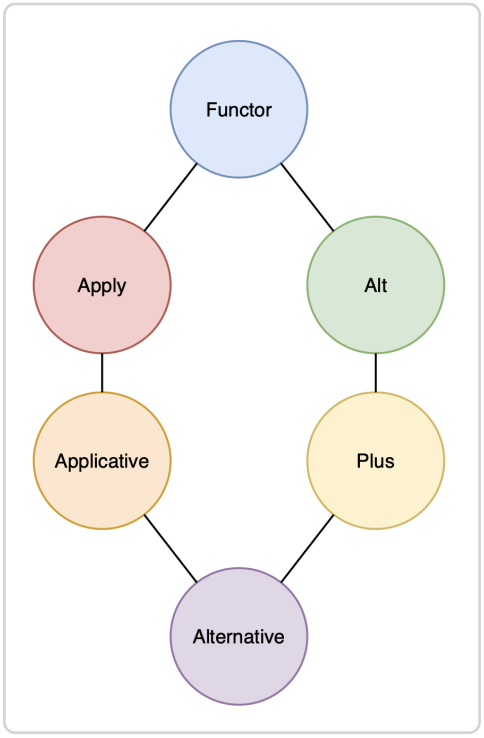
16.20. Alternative Functor

We combine `Applicative` with `Plus` (a Subclass of `Alt`) to produce the ***Alternative Functor***:

```
class (Applicative f, Plus f) <= Alternative f
```

Notice that there are no Typeclass Methods. An `Alternative` has all the methods of `Applicative` AND `Plus` and NO Methods unique to itself.

`Alternative` has a complex hierarchy. So here's a diagram to help cement it in our minds:



A canonical use of `Alternative` is in Parsers. We're going to write an Applicative Parser in the next Chapter and we'll then expand that Parser to be Monadic one once we learn about Monads.

But for now, imagine that the functions `num` and `words` are Parsers and that Parser is an Applicative Parser that has an `Alternative` instance.

Then here's how we could use `Alternative`:

```
data Token
  = Word String
  | Num Int
  | WordNum String Int

wordOrNum :: Parser Token
wordOrNum = Num <$> num <|> Word <$> word ❶

wordAndNum :: Parser Token
wordAndNum = WordNum <$> word <*> num ❷
```

❶ Since `num` and `word` are Applicative Functors, we need to map our Data Constructors over them.

❷ We use `<*>` since `WordNum` takes 2 Parameters.

`wordOrNum` will first try to parse for a number and if and only if that fails, it will then try to parse a word.

`wordAndNum` will parse a word and then immediately parse a number, e.g. `abc123`.

Don't worry if these Parser functions are a bit obtuse. One of the exercises in the next Chapter is to build an

Applicative Parser and hopefully by then, this will make a bit more sense. It'll definitely make more sense once we upgrade our Applicative Parser to a Monadic one.

Chapter 17. Coding Applicatives

Let's start off our coding session like all of the others with creating a module to work in.

Return to the project that you've been coding in and create a file called `Ch17.purs` and add the following:

```
module Ch17 where

import Prelude ❶

import Effect (Effect)
import Effect.Console (log)

test :: Effect Unit
test = do
  log "placeholder" ❷
```

❶ Once again we are not writing anything from Prelude so we'll just import all of it.

❷ We'll delete this soon enough. It's just here to keep the compiler happy.

Next, change your `Main` module to import `Ch17` and call its `test` Function:

```
module Main where

import Prelude
import Effect (Effect)

import Ch17 as Ch17

main :: Effect Unit
main = Ch17.test
```

If you run:

```
npx spago run
```

you should see it build and print:

```
[info] Build succeeded.
placeholder
```

This coding Chapter is going to differ from earlier ones. You're going to code like all the exercise Chapters, but sometimes we'll code together when the going gets rough.

17.1. Writing Applicative Instance for Maybe

Create the Maybe Type from scratch providing all of the prerequisites for Applicative:

- Define the Type
- Create a Show Instance using genericShow (for our test code)
- Create a Functor Instance
- Create an Apply Instance
- Create an Applicative Instance

We've already done some of this before, but do yourself a favor and don't look back at your old code. Write this from scratch for practice. The more times you write this the better.

Here's the test code that you can now replace the placeholder test code with:

```
log $ show $ (+) <$> Just 21 <*> Just 21 ❶
log $ show $ (*) <$> pure 2 <*> (pure 21 :: Maybe Int) ❷ ❸
log $ show $ pure (+) <*> Just 17 <*> Just 25 ❹
```

❶ Prints (Just 42).

❷ Prints (Just 42).

❸ The compiler has no way to know which pure we're using here. So we let it know by telling it that one of the pure calls is a Maybe. It'll be able to figure out the other from that.

❹ Prints (Just 42).

So go ahead and write the code using the above 5-step plan before reading on.

17.2. Code for Applicative Instance for Maybe

We start out by creating the Type:

```
data Maybe a = Nothing | Just a
```

Then deriving Generic so we can use genericShow:

```
import Data.Generic.Rep (class Generic) ❶

data Maybe a = Nothing | Just a
derive instance genericMaybe :: Generic (Maybe a) _ ❷
```

❶ We need to import the Generic class, hence the class keyword.

❷ Derive Generic.

Next, we write the Show instance:

```
import Data.Show.Generic (genericShow) ❶

instance showMaybe :: Show a => Show (Maybe a) where
  show = genericShow
```

❶ We need to import genericShow.

And then the Functor instance:

```
instance functorMaybe :: Functor Maybe where
  map f (Just x) = Just $ f x
  map _ Nothing  = Nothing
```

At this point, we are embarking on new territory. So let's write the Apply, one line at a time:

```
instance applyMaybe :: Apply Maybe where
```

Remember, that Apply's Type Parameter is of Kind Type -> Type, which is why we specify Maybe instead of Maybe a.

Next, we'll write the Just case:

```
instance applyMaybe :: Apply Maybe where
  apply (Just f) x = f <$> x
```

We only unwrap (or destructure) the first Parameter to gain access the Function, f so we can apply it. Once we do, we leverage map's code, which will handle 2 cases for us. The first case <$> handles is where the second Parameter, x is Just x. And the other case is when x is Nothing.

By leveraging map, we don't have worry about those cases here. That means we only have 1 more case to consider and that's when the first parameter is Nothing:

```
instance applyMaybe :: Apply Maybe where
  apply (Just f) x        = f <$> x
  apply Nothing  _        = Nothing
```

We cannot apply a Function we don't have, so we all we can do is return Nothing. This is in keeping with the Semantics of Maybe.

Next, we'll write the Applicative Instance one line at a time:

```
instance applicativeMaybe :: Applicative Maybe where
```

We use Maybe here and not Maybe a for the same reasons we did for Apply.

And since Applicative only has a single Method, pure, we only need to write that. But before we do, let's think about what pure should be.

Remember, that pure takes a Pure Value or a Pure Function (which technically is a Value) and wraps it in the Context. This is helpful when we want to use it with Functions that expect values in a particular Context, e.g. <*> expects this.

So we have to options for implementing pure. We could make pure return Nothing or Just x (where x is our Pure Value). If we were to pick Nothing, we would NEVER be able to use Pure Values with Contextual ones since we would annihilate the value. That's a hugely undesirable limitation.

Also, if we use Nothing, we'd break the Identity Laws for Applicative, which if we think about, is in place so we don't do something destructive like use Nothing.

So the only reasonable, law-abiding and useful option is to use Just:

```
instance applicativeMaybe :: Applicative Maybe where
  pure = Just ❶
```

❶ Written in Point-free Notation.

Test your code to see if it prints out the expected results.

17.3. Writing Applicative Instance for Either

We're going to write the Applicative Instance for Either in exactly the same way we did with Maybe except our test code is going to verify the Applicative Laws (which include the Apply Laws). So we'll need to add the Eq instance to Either.

But instead of writing Eq and Functor, we're going to derive it and let the compiler do some work for us.

So here's our plan:

- Define the Type
- Derive Eq Instance
- Derive Ord Instance (we're going to need this later on)
- Derive Functor Instance
- Create a Show Instance using genericShow (for our test code)
- Create a Bifunctor Instance (we're going to need this later on)
- Create an Apply Instance
- Create an Applicative Instance

And our test code will validate the Applicative Laws:

```
-- LAW: Associative Composition
-- (<<<) <$> u <*> v <*> w = u <*> (v <*> w)
log $ show $ ((<<<) <$> pure identity <*> pure identity <*> pure 1) ==
  (pure identity <*> (pure identity <*> pure 1) :: Either Unit Int)
-- LAW: Identity
-- pure identity <*> x = x
log $ show $ (pure identity <*> pure 1) == (pure 1 :: Either Unit Int)
-- LAW: Homomorphism
-- pure (f x) = pure f <*> pure x
log $ show $ pure (negate 1) == (pure negate <*> pure 1 :: Either Unit Int)
-- LAW: Interchange
-- u <*> pure x = pure (_ $ x) <*> u
log $ show $ (pure negate <*> pure 1) == (pure (_ $ 1) <*> pure negate ::
Either Unit Int)
```

These should all print out true.

Notice that we had to tell the compiler which pure to use. That's because there just isn't enough evidence in the line of code to infer which. By the way, we could've just as easily specified Maybe Int. But we didn't, since we're testing our Either code.

Please take a minute to look at the test code and see how it corresponds to the individual laws. And ask yourself, if I were only given the laws, could I have written this test code.

Once you're done with that, follow the aforementioned plan and run the test code to make sure you're coded up a law-abiding Applicative for Either before advancing on.

17.4. Code for Applicative Instance for Either

As always the Type definition comes first:

```
data Either a b = Left a | Right b
```

Then we'll derive Eq:

```
data Either a b = Left a | Right b
derive instance eqEither :: (Eq a, Eq b) => Eq (Either a b)
```

Notice how we constrained both a and b to have Eq instances.

Then Ord:

```
data Either a b = Left a | Right b
derive instance eqEither :: (Eq a, Eq b) => Eq (Either a b)
derive instance ordEither :: (Ord a, Ord b) => Ord (Either a b)
```

Once again, notice the constraints.

Next we'll derive Functor:

```
data Either a b = Left a | Right b
derive instance eqEither :: (Eq a, Eq b) => Eq (Either a b)
derive instance ordEither :: (Ord a, Ord b) => Ord (Either a b)
derive instance functorEither :: Functor (Either a)
```

Notice how we Partially Applied the Type, a, to the Type Constructor Either. That's because Functor's Type Parameter has Kind Type -> Type but Either has Kind Type -> Type -> Type. This means we can only map over the b Type Parameter, i.e. the Right of Either.

Next, we'll derive Generic:

```
data Either a b = Left a | Right b
derive instance eqEither :: (Eq a, Eq b) => Eq (Either a b)
derive instance ordEither :: (Ord a, Ord b) => Ord (Either a b)
derive instance functorEither :: Functor (Either a)
derive instance genericEither :: Generic (Either a b) _
```

Now we can write the instance for show:

```
instance showEither :: (Show a, Show b) => Show (Either a b) where
  show = genericShow
```

Notice how both a and b are constrained to have Show Instances.

Next, we write the Bifunctor Instance:

```
import Data.Bifunctor (class Bifunctor) ❶

instance bifunctorEither :: Bifunctor Either where
  bimap f _ (Left x)  = Left $ f x
  bimap _ g (Right y) = Right $ g y
```

❶ We need to import the Bifunctor class.

And now we'll write the Apply Instance a line at a time. First, the instance definition:

```
instance applyEither :: Apply (Either a) where
```

Notice how we're specifying Either a instead of Either for the same reason we do in Functor.

Next we will write the Right case:

```
instance applyEither :: Apply (Either a) where
  apply (Right f) x = f <$> x
```

Just like we did with Maybe, we're only unwrapping the first parameter and then leveraging the map from the Functor Instance.

And finally, we write the Left case:

```
instance applyEither :: Apply (Either a) where
  apply (Right f) x = f <$> x
  apply (Left y)  _ = Left y
```

Here's where short-circuiting is applied. We ignore the second Parameter when we have a Left. This has the effect of ignoring all subsequent Rights and Lefts once we encounter our first Left. This is the same short-circuiting behavior we implemented with Maybe when we encountered a Nothing.

The difference to note here is that it really doesn't matter which `Nothing` we favor since they are all the same. But with `Either`, that's not the case and so we favor the FIRST `Left`.

Technically, we favor the FIRST `Nothing` too, but we just didn't think about that with `Maybe`, but we need to think about with `Either`.

And finally, let's write the Applicative one line at a time:

```
instance applicativeEither :: Applicative (Either a) where
```

Notice the `Either a` again for all the same reasons.

And we're going to use the same reasoning we did when we used `Just` as the value for `pure` in `Maybe` by using `Right` since `Left` would annihilate the Pure Value:

```
instance applicativeEither :: Applicative (Either a) where
  pure = Right
```

Run the tests to see how our new code performs.

And it doesn't:

```
[error] Some of your project files import modules from packages that are not
in the direct dependencies of your project.
To fix this error add the following packages to the list of dependencies in
your config:
- bifunctors
```

Add `bifunctors` to you `spago.dhall` and try again.

17.5. Validation

In the previous Chapter, we saw we could use `<*>` to call a Function to construct a Full Name while doing validation:

```
fullNameEither :: Maybe String -> Maybe String -> Maybe String
                  -> Either String String
fullNameEither first middle last =
  fullName <$> (first `errIfMissing` "First name must exist")
           <*> (middle `errIfMissing` "Middle name must exist")
           <*> (last `errIfMissing` "Last name must exist")
```

Since this returns an `Either`, which has an Explicit Behavior (or Effect) of Short-circuiting, we only get the first error. In the above example, if both `first` and `last` are missing, we'd never know that.

We'd only get the error regarding the fact that `first` is `Nothing`. But sometimes, we want to accumulate ALL errors.

To do this, we're going to need something like `Either` without the Short-circuiting behavior.

We're going to call this Type, `Validation`:

```
newtype Validation err result = Validation (Either err result)
```

Notice how we are simply just wrapping `Either` in a `newtype`. This will allow us to leverage `Either's` implementation when it suits us, i.e. when `Validation's` behavior is identical to `Either`, we'll tell the compiler to derive an instance using `newtype`.

With a new Type we also have the option of overriding certain things we don't like about `Either`, e.g. how it short-circuits in `apply`.

So, go ahead and add this definition to your code and derive the `Functor` and `Bifunctor` instances using `derive newtype instance`. But don't forget, to leverage `derive newtype instance`, you first have to derive `Newtype`.

Give it go before reading on.

Okay, so that was a tall order. Let's start with the `Newtype` derivation:

```
import Data.Newtype (class Newtype) ❶

newtype Validation err result = Validation (Either err result)
derive instance newtypeValidation :: Newtype (Validation err result) _
```

❶ Import the `Newtype` class.

Now we're free to derive `Functor` and `Bifunctor` by leveraging the underlying implementations of `Either` by using `derive newtype instance`:

```
newtype Validation err result = Validation (Either err result)
derive instance newtypeValidation :: Newtype (Validation err result) _
derive newtype instance functorValidation :: Functor (Validation err)
derive newtype instance bifunctorValidation :: Bifunctor Validation
```

Let's round out our derivations by deriving `Eq` and `Ord`. Give it a try before continuing on.

There are 2 ways to do the derivation. With or without the `newtype`:

```
derive newtype instance eqValidation ::
  (Eq err, Eq result) => Eq (Validation err result) ❶
derive instance eqValidation ::
  (Eq err, Eq result) => Eq (Validation err result) ❶
```

❶ Each line of code is valid just not both.

So, since they're equivalent, no sense typing the extra word:

```
derive instance eqValidation ::
  (Eq err, Eq result) => Eq (Validation err result)
derive instance ordValidation ::
  (Ord err, Ord result) => Ord (Validation err result)
```

Our new Type is pretty capable for most common operations at this point and not different from `Either`, yet.

The next step is the whole reason for this Type. We want to write an `Apply` Instance that collects all of the errors in some sort of `Semigroup` or `Monoid`.

When faced with this choice, it's best to choose the simplest constraint to start, i.e. `Semigroup`. The compiler will let us know if we need a `Monoid`, e.g. if we use `mempty` and didn't realize it.

But if we constrain our Type to `Monoid` right out of the gate, then the compiler will NOT complain if we don't use `mempty`. It's up to us to remember to ask the question, "Did we constraint our Type too much".

Stop reading here and code the `Apply` Instance to collect the errors in a `Semigroup` as opposed to short-circuiting like `Either`. Try not to read ahead to the answer.

To code the `Apply` instance, let's start with the instance definition:

```
instance applyValidation :: Semigroup err => Apply (Validation err) where
```

Notice that we constrain the error Type to be a `Semigroup` so we can collect the errors up in it. In practice, `Array a` is the most likely candidate for this Type, where `a` is some error Type. The simplest example being `String`.

Next, we code the error case where we combine errors:

```
instance applyValidation :: Semigroup err => Apply (Validation err) where
  apply (Validation (Left err1)) (Validation (Left err2))
                                  = Validation $ Left (err1 <> err2) ❶
```

❶ We combine the errors here. Notice how we didn't need to use `mempty`. That means our choice of `Semigroup` was a good one and will stay.

Next, we code the other error case, i.e. when the left side is an error:

```
instance applyValidation :: Semigroup err => Apply (Validation err) where
  apply (Validation (Left err1)) (Validation (Left err2))
                                  = Validation $ Left (err1 <> err2)
  apply (Validation (Left err))  _ = Validation $ Left err ❶ ❷
```

❶ We simply use the error.

❷ We know that `_` will only be matched to a `Right` otherwise the first Pattern would've matched.

And finally, the successful case where we have a result on both sides:

```
instance applyValidation :: Semigroup err => Apply (Validation err) where
  apply (Validation (Left err1)) (Validation (Left err2))
                                  = Validation $ Left (err1 <> err2)
  apply (Validation (Left err))  _ = Validation $ Left err
  apply (Validation (Right f)) x   = f <$> x ❶
```

❶ And like we've done so many times before when implementing an `Apply` Instance, we only unwrap the Function, `f` and then use `map` from the `Functor` Instance, which, under the covers, leverages `map` from `Either`.

And now we can write the `Applicative` Instance. Give it a try before proceeding.

The code for `Applicative` should be pretty straight forward using the same logic as we did when we wrote `Either's`:

```
instance applicativeValidation ::
  Semigroup err => Applicative (Validation err) where ❶
    pure = Validation <<< Right ❷
```

❶ We still need to constrain the error Type to `Semigroup` since an `Applicative` is also an `Apply`, which has this constraint. If you forget this, the compiler will dutifully remind you.

❷ We take the Pure Value and wrap it up in a `Right`, which is an `Either` and then we wrap that up in `Validation`.

As a final step, and to aid in the next exercise, write the `Show` Instance for `Validation`. Code it up before reading further.

First step is to derive a `Generic` instance for `Validation` so we can use `genericShow`:

```
derive instance genericValidation :: Generic (Validation err result) _
```

Now we can write the instance:

```
instance showValidation ::
  (Show err, Show result) => Show (Validation err result) where
    show = genericShow
```

Notice the constraints on `err` and `result`. We cannot show `Validation` without being able to show them as well.

17.6. Using Validation

Most good examples used for learning are pretty simplistic. They're just complex enough to present the subject at hand. But in this case, I'd like to expand our example into something a bit more realistic at the risk of obscuring the overall goal, i.e. to see how we can validate data using `Validation`.

One side benefit of doing this, however, is that we'll get to see it in a more realistic example, even though it too will fall short of being a real-world one. And when we do a more realistic example, we'll get to see how we combine things we've learned along the way to solve problems that will more closely mirror the ones we'll encounter in the wild.

So, let's start with the idea. We have a family of a Father, Mother and Child. We want to keep track of their names and ages. When we construct their ages, we don't want any invalid ages, i.e. we want to set a reasonable range.

Another important feature is that we want to anonymize the family by working only with their ages. This means that we'll want the family names and ages sometimes but other times we'll only want their ages.

There are certainly dozens of good ways to model this problem, but I've chosen the one I'm going to guide you through to best show off how `Validation` can be used and as a review of some ancillary subjects we've learned along the way that you may have forgotten at this point.

Our first attempt at modeling this might be the following:

```
type FamilyAges = { fatherAge :: Int, motherAge :: Int, childAge :: Int }

type Family =
  { fatherName :: String
  , motherName :: String
  , childName :: String
  , fatherAge :: Int
  , motherAge :: Int
  , childAge :: Int
  }
```

Can you think of why this is undesirable?

The most obvious problem here is that we've MANUALLY duplicated the fields in `FamilyAges` inside `Family`. Can you think of a way we can eliminate this manually duplication process?

Do you remember when we talked about `Row Types`? Don't feel bad if you don't, there's been a lot of ground covered since then. Besides we didn't delve into them deeply. We'll dive a bit deeper here.

First, a review of what we did talk about. Here's the example where Row Types were introduced:

```
isCalifornia :: ∀ r. { address :: Address | r } -> Boolean
isCalifornia { address: { state } } = state == "CA"
```

The salient portion here is the `| r` portion. That tells us that we can have other fields in this Record. Here the only REQUIRED field is `address`.

What we didn't say back then was how to specify `r` syntactically in PureScript.

Let's do that using our family example:

```
type FamilyAgesRow
  = ( fatherAge :: Int, motherAge :: Int, childAge :: Int )
```

Notice the syntax for a Row is different than for a Record. We use Parentheses instead of Curly Brackets.

Now we can rewrite our definition of `Family` so that we can pass this Row as a Type Parameter, `r`. The following definition for `Family` is called an ***Extensible Record***:

```
type Family r = ❶
  { fatherName :: String
  , motherName :: String
  , childName :: String
  | r ❷
  }
```

❶ We have to pass `r` as the Row that will Extend this Record, hence the term Extensible Record.

❷ We specify that `r` is a Row Type that extends the Record using this syntax.

This is great since it removes the duplication, but it also gives us another huge benefit. It gives us some flexibility when creating a `Family`, e.g we could add other fields:

```
Family ( fatherOccupation :: Occupation, motherOccupation :: Occupation )
```

Or we could add NO fields:

```
Family ()
```

Notice that `()` is the syntax for an Empty Row, i.e. no fields. Be carful not to confuse this with the unit Type and Value from Haskell or Elm.

And now that we can see the benefits of using Rows, let's start with Rows to define our fields and then put them together to construct a Record called `Family`. First, add `FamilyAgesRow` to your code and then code up `FamilyNamesRow` on your own before continuing further.

Once you're done coding, you should have something like the following:

```
type FamilyAgesRow
  = ( fatherAge :: Int, motherAge :: Int, childAge :: Int )

type FamilyNamesRow
  = ( fatherName :: String, motherName :: String, childName :: String )
```

Now we're ready to use these to create `Family`. How would you create `Family` from these 2 rows?

You might think that it's possible to add them to Family like this:

```
-- NOT VALID PureScript SYNTAX!!!
newtype Family = Family { | FamilyNamesRow, FamilyAgesRow }
```

or maybe:

```
-- NOT VALID PureScript SYNTAX!!!
newtype Family = Family { | FamilyNamesRow | FamilyAgesRow }
```

Unfortunately, none of these work. The way to do accomplish this is to add a Row Parameter to our Row Types, `FamilyAgesRow` and `FamilyNamesRow`. Please do so before reading on.

If you struggled with this, don't worry. You just haven't seen the syntax enough. So here it is:

```
type FamilyAgesRow r = ❶
  ( fatherAge :: Int, motherAge :: Int, childAge :: Int | r ) ❷

type FamilyNamesRow r = ❶
  ( fatherName :: String
  , motherName :: String
  , childName :: String | r ) ❷
```

❶ Add the `r` Parameter to both Type Constructors.

❷ Extend the Row using the `| r` syntax.

Take the time to update your code.

It turns out that when I create Rows, I always create them like this. Then I can combine them to make more complex Rows like:

```
FamilyNamesRow (FamilyAgesRow ())
```

Here we extend the `FamilyNamesRow` with `FamilyAgesRow`, which is extended with the Empty Row, `()`.

Go ahead and create a `newtype` called `Family` that's a Record that contains both Row fields, then continue reading.

Here's the definition:

```
newtype Family = Family { | FamilyNamesRow (FamilyAgesRow ()) }
```

You can also write this like:

```
newtype Family = Family (Record (FamilyNamesRow (FamilyAgesRow ())))
```

Here we have the `newtype` called `Family` that has a Data Constructor that takes a Record which has all of the fields in `FamilyNamesRow` and `FamilyAgesRow`.

Let's look at what we have so far:

```
type FamilyAgesRow r ❶
  = ( fatherAge :: Int, motherAge :: Int, childAge :: Int | r )

type FamilyNamesRow r ❶
  = ( fatherName :: String, motherName :: String, childName :: String | r )

newtype Family = Family { | FamilyNamesRow (FamilyAgesRow ()) }
```

To make this a bit more realistic, let's not use `Int` for age or `String` for their full names.

Go ahead and create `newtypes` called `Age` and `FullName` and then update `FamilyAgesRow` and `FamilyNamesRow` to use them. Please don't read ahead at the solution until you've given it a try.

Here is the `newtype` for `Age` that simply wraps an `Int`:

```
newtype Age = Age Int
```

And `FullName` that wraps a `String`:

```
newtype FullName = FullName String
```

And our updated Rows:

```
type FamilyAgesRow r
  = ( fatherAge :: Age, motherAge :: Age, childAge :: Age | r )

type FamilyNamesRow r =
  ( fatherName :: FullName
  , motherName :: FullName
  , childName :: FullName
  | r
  )
```

While we're here, go ahead and make the `Show` instance for `Age`, `FullName` and `Family` before reading on.

We simply need to derive the `Generic` instance and we can use `genericShow`:

```
newtype Age = Age Int
derive instance genericAge :: Generic Age _

instance showAge :: Show Age where
  show = genericShow

newtype FullName = FullName String
derive instance genericFullName :: Generic FullName _

instance showFullName :: Show FullName where
  show = genericShow

newtype Family = Family { | FamilyNamesRow (FamilyAgesRow ()) }
derive instance genericFamily :: Generic Family _

instance showFamily :: Show Family where
  show = genericShow
```

We had to make both `Age` and `FullName` showable so that we could make `Family` showable since the fields in the Record are of those Types.

By the way, we could've done the following:

```
newtype Age = Age Int
derive newtype instance showAge :: Show Age

newtype FullName = FullName String
derive newtype instance showFullName :: Show FullName
```

The difference between using `derive newtype` and deriving `Generic` and leveraging `genericShow` is the format of the output:

```
newtype Age = Age Int
derive newtype instance showAge :: Show Age

age :: Age
age = Age 10

--
newtype Age' = Age' Int
derive instance genericAge' :: Generic Age' _

instance showAge' :: Show Age' where
  show = genericShow

age' :: Age'
age' = Age' 10
--

log $ show age  -- Prints 10
log $ show age' -- Prints (Age' 10)
```

With `derive newtype`, the compiler just delegates to the underlying Type, which in the case of `Age` is `Int`. This is why just the number `10` is shown, whereas, in contrast, using `genericShow`, we get the full Type, `(Age' 10)`.

For our purposes, we're going to leverage `genericShow` so there's no need to update your code.

To make our example simpler, we're only going to concentrate on validation of ages. So let's create a `newtype` called `FamilyAges` that mirrors `Family` but only contains ages. Don't forget to make it showable too. Code that before you read ahead to check your work.

The Type definition is nearly identical to `Family`:

```
newtype FamilyAges = FamilyAges { | FamilyAgesRow () } ❶
derive instance genericFamilyAges :: Generic FamilyAges _

instance showFamilyAges :: Show FamilyAges where
  show = genericShow
```

❶ The only difference here is that we only include fields from `FamilyAgesRow`, which is extended with an Empty Row, i.e. `()`.

We are now ready to create the Function we're going to use to validate `Age`. Write just the Type Signature of such a Function called `validateAge` that checks to make sure that the age is in a specified range and then proceed to see how yours compares with mine. Remember, we want this to use our `Validation` Type so we

can collect up all of the errors.

There are many ways to skin this cat, but the following is the one I came up with:

```
validateAge :: Int -> Int -> Age -> String
  -> Validation (Array String) Age
```

Let's look each portion of the signature. First, tell me what this part means:

```
validateAge :: Int -> Int -- partial Type Signature
```

Which `Int` is the upper and lower bounds? You might guess based on convention that lower is first then upper, but I've done that before and been wrong.

To fix this, we're going to create `newtypes` for `LowerAge` and `UpperAge`:

```
newtype UpperAge = UpperAge Int
newtype LowerAge = LowerAge Int
```

Please add this to your code and change you Type Signature. I'll change mine:

```
validateAge :: LowerAge -> UpperAge -> Age -> String
  -> Validation (Array String) Age
```

Okay, now it's really clear and Type safe. At this point, the first 3 Parameters are clear. What's the `String` Parameter?

That's a `String` that tells which person is invalid, i.e. `"Father"`, `"Mother"` and `"Child"`. That's not obvious from the Type Signature. Should we create another type here? We could, but it would only be used here and since there are no other `String` Parameters, I'm going to leave it. I may regret this decision in the future, however. If I do, I'll change it then.

The return Parameter is the most important part here. It's not an `Either` since we want all errors, not just the first. This is why I'm using our `Validation` Type. Also, I'm using `Array String` as the `Semigroup` error Type. If I used `String`, which is also a `Semigroup`, then it would concatenate the error messages making it really hard to read since there'd be nothing to delimit one error from another.

And finally, the result Type is `Age` since that's what we're validating.

Your solution may be Isomorphic to this one, but so that we can stay on the same page, replace yours with this Type Signature. Do so before reading on.

Now, we're ready to finish writing the Function. It turns out that this Function is the perfect case for using Guards, so look back and review Guards if you've forgotten the syntax.

Another thing to keep in mind is that we want our `LowerAge` and `UpperAge` range to be inclusive, i.e. if they are `LowerAge 1` and `UpperAge 18` then both `Age 1` and `Age 18` are VALID.

Go ahead and code the body of the Function. Please don't just skip ahead. You'll benefit more than you realize by trying as many exercises as you can.

There are 3 cases in our Guard:

- The age is too old, i.e. greater than upper (INVALID)
- The age is too young, i.e. less than lower (INVALID)
- Otherwise, the age must be within range (VALID)

Let's code those cases:

```
validateAge
  :: LowerAge
  -> UpperAge
  -> Age
  -> String
  -> Validation (Array String) Age
validateAge (LowerAge lower) (UpperAge upper) (Age age) who ❶
  | age > upper = Validation $ Left [ who <> " is too old" ]
  | age < lower = Validation $ Left [ who <> " is too young" ]
  | otherwise   = Validation $ Right $ Age age ❷
```

❶ Destructure the `newtypes` as soon as possible now that they've outlived their usefulness, i.e. Type safety and making the Type Signature self documenting.

❷ Remember, `otherwise` is defined in `Prelude` as `true`. This is the catch-all case, i.e. the final `else` in a if-then-else chain.

And finally, we can write a Function to create `FamilyAges`. Let's call it `createFamilyAges`. Write the Type Signature for the Function before continuing.

There are many ways we can write the Type Signature. What follows are a few examples from worst to best:

```
createFamilyAges
  :: Int
  -> Int
  -> Int
  -> Validation (Array String) FamilyAges ❶
--
createFamilyAges
  :: Age
  -> Age
  -> Age
  -> Validation (Array String) FamilyAges ❷
--
createFamilyAges
  :: { fatherAge :: Age, motherAge :: Age, childAge }
  -> Validation (Array String) FamilyAges ❸
--
createFamilyAges
  :: { | FamilyAgesRow () } -> Validation (Array String) FamilyAges ❹
```

❶ Uses `Int` making it easy to accidentally swap ages or use any number, e.g. a Temperature value, i.e. no Type Safety.

❷ Hard to tell which Parameter is the Mother's age.

❸ Duplicates a Record Definition.

❹ Leverage the definition of `FamilyAgesRow` to avoid duplication.

It's here we see the benefit of defining the Row independent of the Record. The first Parameter is NOT a `FamilyAges`. Instead, it's any Record that has ages in it that are labeled similarly, but these are not identical Types. `FamilyAges` will, however, unify with this Type meaning you can pass `FamilyAges` to this Function. But you can also pass an infinite number of varying Records, just as long as they have the same labels as is in `FamilyAgesRow`.

`createFamilyAges` is what is often called a ***Smart Constructor***. You create Smart Constructors when you want to make sure that users of your code cannot accidentally create an invalid version of your Type. It's VERY important to NOT export the Data Constructor, `FamilyAges`, in this case. You only want to export the Type Constructor, `FamilyAges`. If that seems confusing, it's because both the Data Constructor and Type Constructor have the same name.

This leaves the client of our module no other choice but to call `createFamilyAges`, which, as we'll soon see, will only return valid `FamilyAges`.

The way to do this is as follows:

```
module Ch17

( Age (..) ❶
, FamilyAges ❷
, FamilyAgesRow ❸
, Validation ❹
, createFamilyAges ❺
, test ❻
) where ❼
```

❶ Without this we'll get a compiler error telling us that to export `FamilyAgesRow`, we also need to export `Age`. We export both the Type Constructor and Data Constructor using `(..)`.

❷ We ONLY export out the Type Constructor, which means you cannot construct the Type directly.

❸ We export the Type Alias `FamilyAgesRow`.

❹ Without this, the compiler will complain that we also need to export `Validation`. This is because no one could use `createFamilyAges` if they cannot refer to the `Validation` Type since that's what `createFamilyAges` returns. Notice we've chosen to NOT export the `Validation` Data Constructor. We may want to do this later, but for now we'll leave this.

❺ We export our Smart Constructor so that we can "safely" construct `FamilyAges`.

❻ We export `test` so our `main` can still call the test code.

❼ Notice the location of the `where` keyword which, so far, directly followed the module name. This is because we haven't explicitly exported anything yet. In those cases, everything is exported from our module. In this case, only what's listed is exported.

Add this to your module definition so you have it as an example of how to do this.

Then write the remainder of the Function. Remember, we're practicing `Applicatives`. So go look back at the `fullNameEither` example in the previous Chapter to refresh you memory. Then, finish off this Function using the same techniques we used there. Stop reading here and give it a try.

Let's start, as always, with the Type Signature:

```
createFamilyAges
  :: { | FamilyAgesRow () } -> Validation (Array String) FamilyAges
```

We're going to take a Record of ages and return `FamilyAges` or an `Array` of error `Strings`.

Next, let's add the Function Parameters:

```
createFamilyAges
  :: { | FamilyAgesRow () } -> Validation (Array String) FamilyAges
createFamilyAges { fatherAge, motherAge, childAge } = ???
```

Notice that we destructure the Record as soon as possible now that it's outlived its usefulness.

Next, we start with the Data Constructor for the `FamilyAges` Type, also called `FamilyAges`:

```
createFamilyAges
  :: { | FamilyAgesRow () } -> Validation (Array String) FamilyAges
createFamilyAges { fatherAge, motherAge, childAge } = FamilyAges ???
```

At this point, it might be good to remind ourselves what the Data Constructor, `FamilyAges`, expects as Parameters:

```
type FamilyAgesRow r = ( fatherAge :: Age, motherAge :: Age, childAge :: Age |
r )

newtype FamilyAges = FamilyAges { | FamilyAgesRow () }
```

So, it wants a record. We need to build a record with the values in it. But not just any values. We want them to be VALIDATED values.

If you reviewed `fullNameEither` then you know that we're going to use `<$>` and `<*>`. And the first part of this Applicative sequence always starts with a Function either in Context or not. If it is in Context, then we use `<*>` right away, otherwise we use `<$>` and then `<*>` for all subsequent Parameters.

Many times, we can use the Data Constructor as that Function, but in this case, `FamilyAges` has only 1 Parameter and that's a Record, which has 3 fields, each of which must be validated. What we need is some Function that takes 3 Parameters and returns a validated Record.

Let's add that magical Function to our code calling it `magic` for now as just a placeholder:

```
createFamilyAges
  :: { | FamilyAgesRow () } -> Validation (Array String) FamilyAges
createFamilyAges { fatherAge, motherAge, childAge } =
  FamilyAges <$> magic ???
```

Notice, how I used `<$>`. That's because whatever our magical Function is, it'll return a `Validation`, which will contain our validated Record that the Data Constructor, `FamilyAges`, needs. We use `map` to apply the Data Constructor to the `Validation` Context.

Now what does our `magic` Function look like:

```
\fatherAge motherAge childAge ->
  { fatherAge: fatherAge, motherAge: motherAge, childAge: childAge }
```

This takes 3 Parameters and produces a Record with the ages.

But this Function can be written more tersely using Wildcards:

```
{ fatherAge: _, motherAge: _, childAge: _ }
```

Remember, the first `_` is the first Parameter to this Anonymous Function. The second `_` is the second Parameter, etc.

So let's replace `magic` with this terser version of our Function:

```
createFamilyAges
  :: { | FamilyAgesRow () } -> Validation (Array String) FamilyAges
createFamilyAges { fatherAge, motherAge, childAge } =
  FamilyAges <$> { fatherAge: _, motherAge: _, childAge: _ } ???
```

Now all we have to do is provide our new Function with VALIDATED age values. Do you know of any way to validate ages?

Exactly, our `validateAge` Function from before.

At this point, I think it would be beneficial to give this Function another try. Remember, we're going to do `Applicative` Function Application with `Validation` using `<$>` and `<*>` just like we did in `fullNameEither`.

Take the above code and write the rest, i.e. the `???` part before reading on.

Coding one Parameter at a time, we apply a VALIDATED `fatherAge` to our Function:

```
createFamilyAges
  :: { | FamilyAgesRow () } -> Validation (Array String) FamilyAges
createFamilyAges { fatherAge, motherAge, childAge } =
  FamilyAges <$> { fatherAge: _, motherAge: _, childAge: _ }
    <$> (validateAge (LowerAge 18) (UpperAge 100) fatherAge "Father") ❶
    ???
```

❶ `validateAge` returns a `Validation` and we map our Function into that Context applying the first

Parameter, `fatherAge`.

Now the next Parameter is the Mother's:

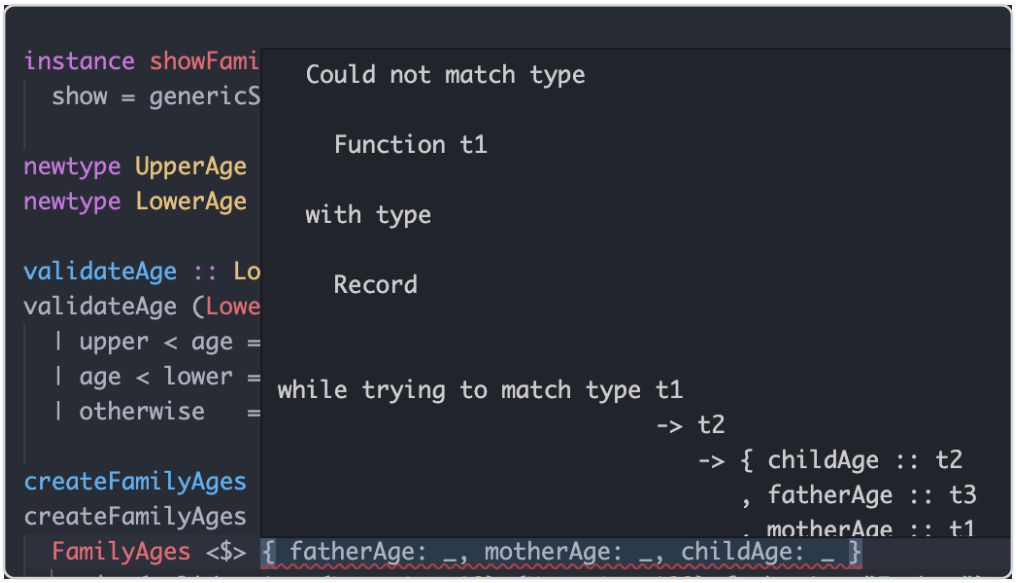
```
createFamilyAges
  :: { | FamilyAgesRow () } -> Validation (Array String) FamilyAges
createFamilyAges { fatherAge, motherAge, childAge } =
  FamilyAges <$> { fatherAge: _, motherAge: _, childAge: _ }
    <$> (validateAge (LowerAge 18) (UpperAge 100) fatherAge "Father")
    <*> (validateAge (LowerAge 18) (UpperAge 100) motherAge "Mother") ❶
    ???
```

❶ We can no longer use map, `<$>`, to apply to a Function in our Context, so we reach for `<*>`.

And our final Parameter is the Child's:

```
createFamilyAges
  :: { | FamilyAgesRow () } -> Validation (Array String) FamilyAges
createFamilyAges { fatherAge, motherAge, childAge } =
 -- COMPILER ERROR!!
  FamilyAges <$> { fatherAge: _, motherAge: _, childAge: _ }
    <$> (validateAge (LowerAge 18) (UpperAge 100) fatherAge "Father")
    <*> (validateAge (LowerAge 18) (UpperAge 100) motherAge "Mother")
    <*> (validateAge (LowerAge 1) (UpperAge 18) childAge "Child")
```

Here's the compiler error we get:



This says that it cannot match `Function t1` with `Record`. This means it's expecting `Record` but got `Function t1` instead, which makes sense since `FamilyAges` wants a Record.

When we look carefully at what we wrote, we can see that it's trying to apply `FamilyAges` to the Function, `{ fatherAge: _, motherAge: _, childAge: _ }`, first. But we want that to happen AFTER we apply all of the Parameters to our Function.

By adding Parentheses, we can control the order:

```
createFamilyAges
  :: { | FamilyAgesRow () } -> Validation (Array String) FamilyAges
createFamilyAges { fatherAge, motherAge, childAge } =
  FamilyAges <$> ({ fatherAge: _, motherAge: _, childAge: _ }
    <$> (validateAge (LowerAge 18) (UpperAge 100) fatherAge "Father")
    <*> (validateAge (LowerAge 18) (UpperAge 100) motherAge "Mother")
    <*> (validateAge (LowerAge 1) (UpperAge 18) childAge "Child"))
```

Once your code compiles you should get the following compiler warning:

```
module Ch17

( Age (..)
, FamilyAges
, FamilyAgesRow
, Validation
, createFamilyAges
, test

  An export for Validation hides data constructors but the type declares an instance of Data.Newtype.Newtype.
  Such instance allows to match and construct values of this type, effectively making the constructors public.
PureScript(HiddenConstructors)

  An export for Validation hides data constructors but the type declares an instance of Data.Generic.Rep.Generic.
  Such instance allows to match and construct values of this type, effectively making the constructors public.
PureScript(HiddenConstructors)

  An export for FamilyAges hides data constructors but the type declares an instance of Data.Generic.Rep.Generic.
  Such instance allows to match and construct values of this type, effectively making the constructors public.
PureScript(HiddenConstructors)
```

The first 2 paragraphs of the compiler warning is telling that we have made `Newtype` and `Generic` instances of `Validation` which allow Data Construction via those instances even though we did NOT export the Data Contructor explicitly.

The third paragraph is letting us know that we have a `Generic` instance of `FamilyAges` which allows it to be constructed even though we didn't export the Data Constructor.

So, we have a few options:

- Do nothing and live with the warning.
- Remove those instances and suffer the consequences of the loss of functionality.
- Export the Data Constructors and hope developers don't call them but use our Smart Constructors instead.

None of these solutions are ideal, but when writing library code, the best solution in my opinon is to not limit the power and functionality and to rely on developers following the rules to construct only using the Smart Constructors. This isn't ideal, but leaving the warning will make us quickly start to ignore ALL warnings and removing functionality will burn us down the line and cause us to have to write boilerplate code because we don't have a helpful Type Class Instance.

So, to implement our solution, make the following changes:

```
module Ch17
-- COMPILER ERROR!!
( Age (..)
, FamilyAges(..) ❶
, FamilyAgesRow
, Validation(..) ❶
, createFamilyAges
, test
) where
```

❶ Added `(..)` to these lines.

And now we have a Compiler Error:



And to fix this is easy:

```
module Ch17

( Age (..)
, Either(..) ❶
, FamilyAges(..)
, FamilyAgesRow
, Validation(..)
, createFamilyAges
, test
) where
```

❶ Export `Either` with its Data Constructors since `Either` has a `Generic` instance and if we don't do this, we'll get the same warning we got earlier with `Validation` and `FamilyAges`.

Now you can use the following code to test your work:

```
log $ show $ createFamilyAges
  { fatherAge: Age 40,  motherAge: Age 30,  childAge: Age 10  }
log $ show $ createFamilyAges
  { fatherAge: Age 400, motherAge: Age 300, childAge: Age 0   }
log $ show $ createFamilyAges
  { fatherAge: Age 4,   motherAge: Age 3,   childAge: Age 10  }
log $ show $ createFamilyAges
  { fatherAge: Age 40,  motherAge: Age 30,  childAge: Age 100 }
log $ show $ createFamilyAges
  { fatherAge: Age 40,  motherAge: Age 3,   childAge: Age 0   }
```

The output should be:

```
(Validation (Right (FamilyAges { childAge: (Age 10), fatherAge: (Age 40),
motherAge: (Age 30) })))
(Validation (Left ["Father is too old","Mother is too old","Child is too
young"]))
(Validation (Left ["Father is too young","Mother is too young"]))
(Validation (Left ["Child is too old"]))
(Validation (Left ["Mother is too young","Child is too young"]))
```

Notice how the errors are collected in our `Semigroup`, `Array String`. Let's look back at the code that does that:

```
instance applyValidation :: Semigroup err => Apply (Validation err) where
  apply (Validation (Left err1)) (Validation (Left err2))
                                  = Validation $ Left (err1 <> err2) ❶
  apply (Validation (Left err)) _ = Validation $ Left err
  apply (Validation (Right f)) x  = f <$> x
```

❶ Here's is where the combining is done when we have an error and we encounter yet another.

Had we used `Either`, we would NOT have gotten all of the errors since it short-circuits after the first one.

17.7. Applicative Parsers

Parsing `Strings` is commonly done with Regular Expressions, but, except for the simplest cases, Regular Expression are hard to read, debug and maintain.

In many non-Functional languages, there are whole libraries to do Parsing but they require a lot of time to learn and therefore do not get used for everyday tasks.

What would be nice is to have something in between. Something that makes Parsing easy, flexible, debuggable, and maintainable. Something that we can reach for when Regular Expressions won't do or when we don't want the burden of reading Regular Expressions, which often resemble hieroglyphics.

Monadic Parsers are one such animal. There are numerous libraries that provide Monadic Parsers in Haskell, e.g. `parsec`, `megaparsec`, `autoparsec`, etc., and in PureScript there is `purescript-parsing` and `purescript-string-parsers`.

We're going to build a simple Parser using what we've learned so far, which means we won't be able to make a Monadic Parser just yet, but we can make an ***Applicative Parser***. Haskell has `optparse-applicative` and PureScript has a port of that library called `purescript-optparse`.

These Parsers are typically used for Command Line Argument parsing. But, as we will soon learn, all `Monads` are also `Applicatives`, which means that we'll be able to use our Applicative Parser as the basis for the Monadic one we'll build in the `Monad` exercises.

Parsers can get very complex and you could spend your whole career just on Parser design and implementations, but we're just going to create a Parser that's fairly simple, yet very powerful. One that we can use at the drop of hat, i.e. something we'll reach for as readily as Regular Expressions to do our parsing.

17.8. Parsers in General

Let's look at a simple Parsing example to help us understand how we want to design our Parser:

```
"10/2/1962"
```

Here is a date from the past. We've all needed to parse a date before and we're all too familiar with the Regular Expression to do so:

```
\d{1,2}\/\d{1,2}\/\d{4}
```

If you not familiar with Regular Expressions this will be confusing and appear to be gibberish. Even if you are, you have to think about it far too long and when I initially wrote this expression, I wrote it wrong. I forgot to escape the forward slash characters.

So we'd like to have something better than this, but parsing is parsing and the same issues arise in all parsers, which means we can use this example to help us think about ours.

Let's look at this Regular Expression in detail:

```
\d{1,2}   -- digits 1 or 2
\/        -- forward slash (escaped since it's a special character)
\d{1,2}   -- digits 1 or 2
\/        -- forward slash (escaped since it's a special character)
\d{4}     -- digits 4
```

We have 3 unique "mini-parsers":

```
\d{1,2}   -- digits 1 or 2
\/        -- forward slash (escaped since it's a special character)
\d{4}     -- digits 4
```

In typical Functional fashion, it would be nice to build Parsers for each of these and then combine them somehow to finish the whole job. In Functional Programming, we combine Functions by composing them where the output of one Function is the input to another.

So if we're going to combine Parsers to build larger Parsers, each Parser will need to whittle away at our `String`, doing their part, until we fail or success to parse.

That means that the Parsing State is going to need to be passed from Parser to Parser, i.e. when the current Parser is done, it passes what's left of the `String` to the next Parser who takes a stab at parsing what's left.

Also, if a single Parser in the chain were to fail, we want to short-circuit the parsing and return the error, hopefully with some useful information as to what went wrong.

17.9. Writing an Applicative Parser

Now that we have a Strategy for how to implement Parsers that can be combined to produce more complex

Parsers, let's start modeling our Types.

First, we talked about each Parser accepting a State and returning that State. Without worrying how this is going to work, let's simply design our Parser State. We know that we are going to Parser characters from strings and return something. In the case of our date example, we were going to need to parse the `String` and return an `Int`.

We'll want that Parser to return an `Int`. But not every Parser is going to parse numbers into an `Int`. So, we'll want our Parser to return the Polymorphic Parameter, `a`. And there are no constraints on `a`. `a` can be any Type.

The other thing we'll need is the `String` to parse. We will consume what we need from that `String` and return what's left over.

Before you start coding, create a new Module called `Parser.purs` and add the following:

```
module Parser where

import Prelude

import Effect (Effect)
import Effect.Console (log)

test :: Effect Unit
test = do
  log "placeholder"
```

And change your `Main` module to:

```
module Main where

import Prelude
import Effect (Effect)

import Parser as P

main :: Effect Unit
main = P.test
```

And run it with:

```
npx spago run
```

This should display:

```
Compiling Main
[info] Build succeeded.
placeholder
```

Go ahead and model a Type or Type Alias called `ParserState` that contains both the parsed value of Type `a` and a `String`. Do this before reading on.

So, modeling this Type isn't straightforward since there are lots of ways to do it. I'll display a few:

```
type ParserState a = Tuple String a

data ParserState a = ParserState String a

newtype ParserState a = ParserState (Tuple String a)
```

While the `data` and `newtype` are proper types, I don't see the need for anything more than a simple Type Alias, at least at this point. I may find that I'll regret this decision, but for now, I think it's good enough. It will let us leverage the code written for `Tuple`, e.g. `Functor`, `Bifunctor`, `Applicative`, etc.

Notice that we have the Polymorphic Parameter, `a` second. That's so we can have a leverage `Tuple's Functor` Instance. Not sure if we're going to need it, but there's no skin off our noses at this point, so we might as well put it second.

So change your code to use the Type Alias:

```
import Data.Tuple (Tuple) ❶

type ParserState a = Tuple String a ❷
```

❶ We'll need to import `Tuple`.

❷ We've placed the `a` Type Parameter second just in case we want to map over it someday.

Next, step in modeling our Parser is to model the error Type. There are 3 possibilities for modeling it. First, is to just make it a human readable `String` using a Type Alias:

```
type ParserError = String
```

The second approach is to make it computer readable using a Sum Type or a Coproduct:

```
data ParserError
  = EOF
  | SomeOtherError
  | AnotherErrorWithReason String
```

And finally, we could just make it a Parameter to our Parser. This will require us to make a Typeclass to allow our general Parser to set errors. This is the most complex but the also most flexible. This allows the Application to make its own error Type with its own errors.

The `Parser` we're building would normally be in a library. There's no way for the library developer to know what the Application errors are. But, the Typeclass can dictate the errors that the library will return from its base `Parsers`. And the Application can write its own `Parsers` on top of the library ones and return its Application-specific errors from those `Parsers`.

We're going to go with this for no other reason than I think you'll learn the most from it. Keep in mind, depending on your needs, a Sum Type or even a `String` can work. But, a Parser is a pretty general tool and so we'll get more out of it the more flexibility we build into it.

We're not exactly ready to model our Parser. However, we can write the following:

```
newtype Parser a e = Parser ??? ❶ ❷
```

❶ `a` is our return Type.

❷ `e` is our error Type. It's going to be specified by the user of our `Parser`. It represents the Application-specific error Type.

The real question is what does out Parser contain. Well, from what we know, it could contain a Value or a Function. Which do you think it should contain?

Well, if it contains just a Value, e.g. `10` or `"abc"`, that doesn't seem very useful. But if it contains a Function, we can call it to do something useful, e.g. parse a `String`.

So, now we know we want a Function. But what should this Function's do? It should PARSE. Since we're parsing `Strings`, then it's going to need a `String` and since it's going to be composed with other Parsers, it should return the `ParserState` which should, at least, contain the remaining `String`.

In the ParserState, we have 2 useful pieces of data. First, the parsed Value and second, the part of the `String` that wasn't used by the parse Function. The left over `String` can then be passed to the next Parser.

You're now ready to write a definition that is called `ParseFunction`. This definition should be the Type Signature of the Function in the Parser. Please give it try without looking ahead.

Our Type Alias is simply the parser Function's Type Signature:

```
type ParseFunction a = String -> ParserState a
```

What's missing from this Type Signature?

This Function cannot fail. So go ahead and change it to allow failure with an Error Type Parameter called `e` before continuing.

First thought you might have is to use `Maybe`, but we cannot since `Maybe` doesn't have an error Parameter. All it has is `Nothing` for an error and `Nothing` only tells us that it failed not why it failed.

Next thought you might have is to use our fancy new `Validation` Type, but we're not going to since we don't want to deal with the complexity of multiple Parser errors, which is harder than you might think.

To do this, we're going to use `Either`:

```
import Data.Either (Either) ❶

type ParseFunction a e = String -> Either e (ParserState a) ❷
```

❶ We needed to import `Either`.

❷ We added the `e` Type Parameter.

That's better, but we'll need one more thing. We need `e` to be constrained by a Typeclass that has Methods to set parser errors. This requirement is not obvious, but after writing a few parsers, I know we're going to need to specify errors in our `Parser` Instances. Trust me on this.

Stop reading and write a Typeclass called `ParserError` with NO Methods for now. We'll add them as we need them.

The Typeclass with no Methods is simply:

```
class ParserError (e :: Type) ❶ ❷
```

❶ We omit the `where` keyword since we have no Methods.

❷ We explicitly tell the compiler that `e` has Kind `Type` since there's nothing on the righthand side of this definition for it to know that it's not just any Kind, e.g a `Symbol`, `Row Type` or `Constraint`.

Now constrain the `ParseFunction` to return errors of Type `ParserError` before reading on.

To constrain the `ParseFunction`, we simply add the Constraint to the right side of the equation:

```
type ParseFunction a e =
  ParserError e => String -> Either e (ParserState a)
```

When we cannot parse the `String`, we'll return `Left e`, otherwise, we'll return `Right (ParserState a)`.

For example, let's say we have a Parser that parses one or more Numbers. And we have the following `String`:

```
"123abc"
```

After calling `ParseFunction`, we'll have the following return value:

```
Right (Tuple "abc" 123)
```

Now we can finish writing our `Parser` Type:

```
newtype Parser a e = Parser (ParseFunction a e)
```

Add this definition to your code.

At this point, your code should look like:

```
module Parser where

import Prelude

import Data.Either (Either)
import Data.Tuple (Tuple)
import Effect (Effect)
import Effect.Console (log)

type ParserState a = Tuple String a

class ParserError (e :: Type)

type ParseFunction a e =
  ParserError e => String -> Either e (ParserState a)

newtype Parser a e = Parser (ParseFunction a e)

test :: Effect Unit
test = do
  log "placeholder"
```

We haven't written a lot yet. We've spent a lot of time thinking about what we want and compare alternative solutions. Once we are satisfied with our initial choices, we're ready to move on.

I say ***initial choices*** for a reason. Many times, you think you've modeled something correctly, but learn later that your initial assessment was wrong. In that case, you'll need to modify your model. We may have to do that later, but for now, this looks good.

So what do you think we should do next? If you're not sure, then ask yourself, what is our goal.

The answer is to create an Applicative Parser. So if that's true, then we'd better write an `Applicative` Instance for our Parser. Let's look at the definition of `Applicative` to see how to do that:

```
class Apply f <= Applicative f where
  pure :: ∀ a. a -> f a
```

Now, that we're looking at `Applicative`, we can see that we need to have an `Apply` Instance first. So let's look at that definition:

```
class Functor f <= Apply f where
  apply :: ∀ a b. f (a -> b) -> f a -> f b

infixl 4 apply as <*>
```

Again, we can see that we'll first need a `Functor` Instance:

```
class Functor f where
  map :: ∀ a b. (a -> b) -> f a -> f b
```

We finally have something that doesn't have a Superclass. So our plan is to first write the `Functor`, then the `Apply` and finally the `Applicative`, which is the main goal of this Chapter.

Go ahead and write the `Functor` Instance for our Parser before reading the solution.

If you got stuck, I'd suggest writing the Type Signature for the `map` Function specialized for `Parser`. Try that and see if it doesn't help a bit. Once you've tried that, get as far as you can and then read on.

Let's write the `Functor` Instance one line at a time and see how far we can get:

```
instance functorParser :: Functor (Parser a) where
```

Remember, that `Parser's` Kind is `Type -> Type -> Type`. That means we'll need to hold a Type Parameter constant.

But we've held the WRONG parameter constant. We want to map over the Parsed Value not the Error. We want to hold the Error Type constant like we've done with `Either's` Functor instance:

```
instance functorEither :: Functor (Either a) where
  map _ (Left x)  = Left x
  map f (Right y) = Right $ f y
```

So we're going to need to SWAP our Type Parameters to our `Parser` Type:

```
newtype Parser e a = Parser (ParseFunction e a) ❶
```

❶ Swapped `a` and `e` from our earlier definition.

Go ahead and change your code to swap these Type Parameters.

You know, it might be a good idea to swap `a` and `e` in `ParseFunction` too. This will keep us from getting the

order confused:

```
type ParseFunction e a =
  ParserError e => String -> Either e (ParserState a) ❶

newtype Parser e a = Parser (ParseFunction e a) ❷
```

❶ Swapped `a` and `e` on the left-hand side of the equal sign.

❷ Swapped `a` and `e` Type Parameters to `ParseFunction`,

Go ahead and update your code with these changes.

Now we can hold the correct Type Parameter constant:

```
instance functorParser :: Functor (Parser e) where
```

And we'll add the specialized Type Signature for `map`:

```
instance functorParser :: Functor (Parser e) where
  map :: ∀ a b. (a -> b) -> Parser e a -> Parser e b ❶
```

❶ Instead of `f a` and `f b`, we have `Parser e a` and `Parser e b`. This is what is meant by specializing for `Parser`.

You should have at least this. If not add it. Then add as much of the Function implementation as you can before reading on.

Here's how far you may have gotten before it got difficult:

```
instance functorParser :: Functor (Parser e) where
  map :: ∀ a b. (a -> b) -> Parser e a -> Parser e b
  map f (Parser g) = Parser \s -> ???
```

We know that `map` returns a `Parser`, which takes a Function that accepts a `String`, we're calling `s` here. And that's as far as we can go for now without doing some more thinking.

We know that `Functors` map OUTPUTS (Covariant). So, we'll need to call the mapping Function AFTER calling the original Function.

We also know that our original Function, `g`, expects a `String` and it turns out that we have one, `s`. So write that much even if you get a compiler error. We'll look at the error together and use it to help us write the rest of this Function. Give it go.

I'm going to first add the part where we call the original Function and pass it `s` (ignoring compiler errors):

```
instance functorParser :: Functor (Parser e) where
  map :: ∀ a b. (a -> b) -> Parser e a -> Parser e b
  map f (Parser g) = Parser \s -> g s          -- COMPILER ERROR!!
```

And next, I'll call our mapping Function `f` AFTER we've called `g`:

```
instance functorParser :: Functor (Parser e) where
  map :: ∀ a b. (a -> b) -> Parser e a -> Parser e b
  map f (Parser g) = Parser \s -> f $ g s          -- COMPILER ERROR!!
```

We still have a compiler error so let's look at it in detail:

```
import Data.Tuple (Tuple)
import Effect (Effect)
import Effect.Console (log)

type ParserState a = Tuple String a

class ParserError e

type ParserFunction a e = ParserError

data Parser e a = Parser (ParserFuncti

instance functorParser :: Functor (Par
  map :: ∀ a b. (a -> b) -> Parser e a
  map f (Parser g) = Parser \s -> f $ g s
```

```
  Could not match type

    Either e1 (Tuple String a0)

  with type

    a0


while checking that type Either e1 (Tuple String a0)
  is at least as general as type a0
while checking that expression g s
  has type a0
in value declaration functorParser
```

We can see that `g s` is the problem since it's underlined.

We can also see that we have a type mismatch, `Either e1 (Tuple String a0)` with `a0`.

This is because `f` wants an `a` or `a0` as the compiler calls it and we've given `f` an `Either e1 (Tuple String a0)`, which is what `g s` returns.

So, `f` wants to operate on `a0` here but `a0` is buried inside first an `Either` and then a `Tuple`. So it looks like we're going to have to do some fancy mapping to get `f` to make 2 hops.

But that may be a bit difficult to do in one fell swoop. So go ahead and add map `f` into `g s` and see what new compiler error you get before reading on.

To map `f` over `g s`, we simply change `$` to `<$>`:

```
instance functorParser :: Functor (Parser e) where
  map :: ∀ a b. (a -> b) -> Parser e a -> Parser e b
  map f (Parser g) = Parser \s -> f <$> g s       -- COMPILER ERROR!!
```

And our new compiler error is:

```
import Data.Tuple (Tuple)
import Effect (Effect)
import Effect.Console (log)

type ParserState a = Tuple String a

class ParserError e

type ParserFunction a e = ParserError e

data Parser e a = Parser (ParserFunction

instance functorParser :: Functor (Parse
  map :: ∀ a b. (a -> b) -> Parser e a -
  map f (Parser g) = Parser \s -> f <$> g s
```

```
  Could not match type

    Tuple String a1

  with type

    a1


while trying to match type Either e2 (Tuple String a1)
  with type t0 a1
while checking that expression g s
  has type t0 a1
in value declaration functorParser
```

From this error we can see that we've made 1 hop, i.e. into `Either`, but we need to map again so we map the `a0` inside the `Tuple`.

Good thing we made `a` the second Parameter of our Tuple. If the Type Parameters were swapped, we wouldn't be able to use `map` to map `a`, We'd have to use `lmap` from `Bifunctor` instead, which isn't terrible but I'd rather stick with `map` when possible.

So how do we do this `map`? We can't just add another `<$>`:

```
f <$> <$> g s      -- COMPILER ERROR!!
```

That's not a valid use of a Binary Operator. Can you think of a way?

We have:

```
f <$> g s
```

We want to map `f` again. If you have any ideas, then give them a try and see how you do. If not, keep reading.

Let's rewrite this using `map` instead of `<$>` to see if that gives us some insight:

```
map f (g s)
```

Okay, now let's add another `map`:

```
map map f (g s)     -- COMPILER ERROR!!
```

Clearly, we need to some Parentheses, but where? We can randomly just start putting in Parentheses until we make the compiler happy, but this is just flailing and may work as a strategy in this simple case, but it's really a terrible general strategy.

Instead of flailing, let reason it out. Looking at the FIRST `map`, we know it wants a Function as it's first Parameter. But of what Type?

Looking at the Type Signature:

```
map :: ∀ a b. (a -> b) -> f a -> f b
```

So the Function `map` wants as it's first Parameter is of `a -> b`, but we've passed it ANOTHER `map` which isn't a Function of 2 Parameters.

With only 1 `map`, it was happy when we passed `f` because it's a Function of 2 Parameters. So do we have a Function of 2 Parameters other than `f`?

Yes we do. It's `map f`. Here's its Type Signature:

```
map f :: ∀ a b. f a -> f b
```

So, this should be the FIRST Parameter to the FIRST `map`. Go ahead and add the Parentheses to make `map f` the first Parameter before reading on.

Here's the change:

```
instance functorParser :: Functor (Parser e) where
  map :: ∀ a b. (a -> b) -> Parser e a -> Parser e b
  map f (Parser g) = Parser \s -> map (map f) (g s) ❶
```

❶ We're mapping `map f` over `g s`, which once it's inside of the `Either` maps `f` over the `Tuple`.

This compiles.

Now I'm not a fan of this syntax since it has too many `maps` for my taste. So I'm going to change the first `map` to `(<$>)` in Prefixed Notation first:

```
instance functorParser :: Functor (Parser e) where
  map :: ∀ a b. (a -> b) -> Parser e a -> Parser e b
  map f (Parser g) = Parser \s -> (<$>) (map f) (g s)
```

And now I'm going to rewrite it in Infixed Notation:

```
instance functorParser :: Functor (Parser e) where
  map :: ∀ a b. (a -> b) -> Parser e a -> Parser e b
  map f (Parser g) = Parser \s -> map f <$> g s ❶
```

❶ Once again, we're mapping `map f` over `g s`, which once it's inside of the `Either` maps `f` over the `Tuple`.

This syntax is my personal favorite and you'll probably find this style more common. But if you like the previous way, feel free to use it.

Whew. That was some work. Maybe you'll want to take a break and continue after a good 20 or 30 minutes.

Hopefully, you've taken that break and are ready to work on `Apply`. Go ahead and write `Apply` or get as far as you can before reading more.

As usual, we'll write the Type Signature first:

```
instance applyParser :: Apply (Parser e) where ❶
```

❶ Once again, we hold the `e` constant.

Next, we'll write the specialized Type Signature just to help us keep the Types straight in our heads. I usually delete this line when it compiles because it's not required:

```
instance applyParser :: Apply (Parser e) where
  apply :: ∀ a b. Parser e (a -> b) -> Parser e a -> Parser e b
```

If you got this far, you're doing great. If not, that's okay, simply type this code into your Module.

Next step, write the Function but only left-hand side of the equal sign before reading on.

Here's the Function's left-hand side:

```
instance applyParser :: Apply (Parser e) where
  apply :: ∀ a b. Parser e (a -> b) -> Parser e a -> Parser e b
  apply (Parser f) (Parser g) = ???
```

Try to write as much of the right-hand side as you can before reading on.

Just like with our `Functor` Instance, we know we're going to return a `Parser` which has a Function that takes a `String`, which we'll call `s`. So let's write that much:

```
instance applyParser :: Apply (Parser e) where
  apply :: ∀ a b. Parser e (a -> b) -> Parser e a -> Parser e b
  apply (Parser f) (Parser g) = Parser \s -> ???
```

If we stop here and determine the Types for `f` and `g`, it's going to help us immensely when writing the code on the right-hand side of the equal sign.

If you'd like to try your hand out this before reading my approach, feel free to do so now.

Let's start with `Parser f` and `Parser g` Types:

```
Parser f :: Parser e (a -> b)
Parser g :: Parser e a
```

`g` looks to be easier to determine, so let's work on `g` first. We know that `Parsers` contain `ParseFunctions`, so let's add both the `Parser` definition and the `ParseFunction` definition to the mix:

```
type ParseFunction e a =
  ParserError e => String -> Either e (ParserState a)
newtype Parser e a = Parser (ParseFunction e a)

Parser f :: Parser e (a -> b)
Parser g :: Parser e a
```

So let's just write `g's` Type using `Parser` and `ParseFunction` definitions as reference:

```
type ParseFunction e a =
  ParserError e => String -> Either e (ParserState a)
newtype Parser e a = Parser (ParseFunction e a)

Parser f :: Parser e (a -> b)
Parser g :: Parser e a

g :: String -> Either e (ParserState a)
```

Now we can simply replace the `a` in `g's` Type Signature with `a -> b` to get `f's` Type Signature since that

the only difference between the Type Signatures of `Parser g` and `Parser f`:

```
Parser f :: Parser e (a -> b)
Parser g :: Parser e a

f :: String -> Either e (ParserState (a -> b))
g :: String -> Either e (ParserState a)
```

We can now return to our code:

```
instance applyParser :: Apply (Parser e) where
  apply :: ∀ a b. Parser e (a -> b) -> Parser e a -> Parser e b
  apply (Parser f) (Parser g) = Parser \s -> ???
```

At this point, let's step back and look carefully what we have. We have 2 `Parsers`. We're clearly going to want to build a final Parser that calls both of them. But what order? Should the leftmost `Parser` get called first or last?

Remember the definition of `apply's` Binary Operator:

```
infixl 4 apply as <*>
```

So, `apply` is going to be used like:

```
 parser1 <*> parser2 <*> parser3
```

And, since `<*>` is Left-Associative, the above is equivalent to:

```
((parser1 <*> parser2) <*> parser3) ❶
```

❶ Added implied Parentheses.

It should be clear that the Parsers are applied from left to right. In other words, we want to execute the leftmost first.

Go ahead and call the first parser Function, `f`, which returns an `Either`. Then check its return value using a `case` expression. Go as far as you can before reading any further.

Here's we call `f`:

```
instance applyParser :: Apply (Parser e) where
  apply :: ∀ a b. Parser e (a -> b) -> Parser e a -> Parser e b
  apply (Parser f) (Parser g) = Parser \s -> f s      -- COMPILER ERROR!!
```

And now adding the `case` expression:

```
import Data.Either (Either(..)) ❶
import Data.Tuple (Tuple(..)) ❶

instance applyParser :: Apply (Parser e) where
  apply :: ∀ a b. Parser e (a -> b) -> Parser e a -> Parser e b
  apply (Parser f) (Parser g) = Parser \s -> case f s of
    Left err -> Left err ❷
    Right (Tuple s1 h) -> ??? ❸ ❹
```

❶ Add `(..)` to import Data Constructors, `Left` and `Right`.

❷ We simply return an error if the first `Parser` fails.

❸ Remember, that `ParserState` is a `Tuple`.

❹ If it succeeds, `s1` is the `String` that's left over after the first `Parser` was done. Also, `h` is the Function that's returned from `f :: String -> Either e (ParserState (a -> b))`.

We pretty much just need to mechanically repeat what we did with the first `Parser` with the notable exception that `g` will return a Value instead of a Function. It is this value that we'll map over using `h`.

Do this before reading on to the solution.

Here is the solution:

```
instance applyParser :: Apply (Parser e) where
  apply :: ∀ a b. Parser e (a -> b) -> Parser e a -> Parser e b
  apply (Parser f) (Parser g) = Parser \s -> case f s of
    Left err -> Left err
    Right (Tuple s1 h) -> case g s1 of
      Left err -> Left err ❶
      Right (Tuple s2 x) -> ??? ❷ ❸
```

❶ We return an error if our second `Parser` fails.

❷ Notice how we've named the variable `x` here to remind us that `g` returns a Value in the `Tuple`.

❸ Here `s2` is what's left of the `String` after the second `Parser` is through with it.

All that's left to do is apply `h` to `x` and wrap everything up in a `Tuple` and then wrap that up in an `Either`.

Code this final step yourself first, then read on.

The final step:

```
instance applyParser :: Apply (Parser e) where
  apply :: ∀ a b. Parser e (a -> b) -> Parser e a -> Parser e b
  apply (Parser f) (Parser g) = Parser \s -> case f s of
    Left err -> Left err
    Right (Tuple s1 h) -> case g s1 of
      Left err -> Left err
      Right (Tuple s2 x) -> Right $ Tuple s2 (h x) ❶
```

❶ We apply `h :: a -> b` to `x :: a` producing a Value of Type `b`. This is the whole point of the `apply` Method, to change a `Parser e a` to a `Parser e b`.

Notice that we had to deal with the error cases, twice. If that code looks verbose to you, that's because it is. We'll be able to come back and simplify this once we learn about Monads. Turns out that `Either` is a `Monad` and there are Methods in the `Monad` Typeclass that will make this code prettier. But for now, this code works.

Notice, for a moment, how much code we wrote. It's not a lot. But it's dense. There was a lot of thinking involved to figure out our Types and make sure we did things in the right order.

While the thinking was difficult, especially for your first time, the coding wasn't really that hard. We've experienced this before and will continue to do so. This is the situation I refer to when I joke that in Haskell (or PureScript), you think for 2 hours and code for 2 minutes.

And finally, we should make our `Parser` an `Applicative`. To help in this process, look up `Applicative` on `Pursuit`. Go ahead and write the `Applicative` Instance before reading on.

As always, the instance definition is first:

```
instance applicativeParser :: Applicative (Parser e) where
```

Then the Methods, which here is only `pure`:

```
instance applicativeParser :: Applicative (Parser e) where
  pure x = Parser \s -> ???
```

So next we need to just return a successful parse of `x` WITHOUT consuming any characters from the `String`, `s`. If you got stuck here, try to apply this strategy and see if you can finish it. Then read on.

Here's the completed class definition:

```
instance applicativeParser :: Applicative (Parser e) where
  pure x = Parser \s -> pure $ Tuple s x ❶
```

❶ Note, that rightmost `pure` is `pure` for `Either`.

We consume ZERO characters from `s` and simply return it. We also return `x` as if we've actually parsed something. This is sort of like `const` but for a `Parser`.

And we now have ourselves an ***Applicative Parser***.

17.10. Looking Deeper at our Parser

After having written the `Apply` Instance for our `Parser`, you may or may not be wondering how we can have a `Parser` that has a Function that takes a `String` and returns a Function, `a -> b`. Seems ludicrous. We call a parsing Function, which may or may not consumes some or all of a `String`, and it returns a Function. Returning an `Int` or a `Boolean` or some other Value makes sense, but a Function?

While this sounds odd, it actually isn't. Let's look at the 2 cases where a Function can be returned from a `Parser`.

First is when we use `pure`:

```
instance applicativeParser :: Applicative (Parser e) where
  pure x = Parser \s -> pure $ Tuple s x
```

We can see that the `String`, `s`, is not consumed at all, but instead is simply returned unchanged. If `x` is a Function, then we can see how a `Parser` can take a `String` and return a Function. It doesn't consume any part of the `String` but instead returns it along with the Function. So this case makes sense.

The second case is when we call `map`:

```
instance functorParser :: Functor (Parser e) where
  map f (Parser g) = Parser \s -> map f <$> g s
```

And in this case, the `String` is consumed by `g` our Original Parser and then we map over the output value of `g s`. But how does this case return a Function?

Remember that `f` is of Type `a -> b`. But `b` can be anything and thanks to Currying, `b` can be a Function, e.g. `c -> d -> e`. So if we map over a `Parser` with a Function that only gets Partially Applied, then that resulting `Parser` will return a Function.

Now, let's examine `apply`, where we expect the left-hand side to be a `Parser` that returns a Function:

```
instance applyParser :: Apply (Parser e) where
  apply (Parser f) (Parser g) = Parser \s -> case f s of ❶
    Left err -> Left err
    Right (Tuple s1 h) -> case g s1 of ❷
      Left err -> Left err
      Right (Tuple s2 x) -> Right $ Tuple s2 (h x)
```

❶ Here is where we call the `ParseFunction` from the `Parser` that returns the Function, `h`.

❷ Here we bind `h` to the Function returned by the left-hand side `Parser`.

Notice that we pass the original `String`, `s`, to the `ParseFunction` of the `Parser` that returns a Function. The `String` will either be consumed by another `Parser`, if the `Parser` was the result of a `map`, or it'll be unchanged, if this `Parser` was the result of a call to `pure`. In the last case, `s1` will equal `s`.

So, while on the surface it may seem strange that we can have a `Parser` consume a `String` and produce a Function, it's not so crazy when we look at the details of our implementation.

17.11. A Common Pattern with Our Parser

One thing worth pointing out regarding our Parser, is the following Pattern:

```
Parser \s -> ...
```

When we have a `Functor` that contains a Function, we'll use this pattern over and over again.

Let's look at the Functions we written so far:

```
instance functorParser :: Functor (Parser e) where
  map f (Parser g) = Parser \s -> map f <$> g s ❶

instance applyParser :: Apply (Parser e) where
  apply (Parser f) (Parser g) = Parser \s -> case f s of ❶
    Left err -> Left err
    Right (Tuple s1 h) -> case g s1 of
      Left err -> Left err
      Right (Tuple s2 x) -> Right $ Tuple s2 (h x)

instance applicativeParser :: Applicative (Parser e) where
  pure x = Parser \s -> Right $ Tuple s x ❶
```

❶ In all 3 Functions, we see the pattern `Parser \s -> ...`.

I'm taking just a moment to point this out because we'll see this pattern repeat itself when we write some common Monads. And once you've seen this pattern enough times, you begin to start coding with it and if you do, you'll, more times than not, be on the right track.

At this point you haven't seen enough examples to recognize this pattern. But trust me, it's there.

17.12. Using Our Parser

Looking at the code you have in `Parser.purs`, you may wonder, how we're going to use our `Parser`. So far, we've only modeled a Parser. And the only operations we can do is manipulate the output of a `Parser` with `map`, combine two or more with `apply` and create trivial ones with `pure`.

But how do we actually parse something?

Well, just like `runFoldL` that we wrote in a previous Chapter, we need a Function to crack open a `Parser` and call it's `ParseFunction`.

So, write a Function called `parse` that will call the Function contained in a `Parser`. Try not to read ahead at the solution.

Let's review the `ParseFunction` Type Signature:

```
type ParseFunction e a =
  ParserError e => String -> Either e (ParserState a)
```

This Function takes a `String` and returns `Either e (ParserState a)`. So that return Type is going be the same for our `parse` Function since we're executing the `Parser's` Function.

So let's write the Type Signature first:

```
parse :: ∀ e a. Parser e a -> String -> Either e (ParserState a) ❶
```

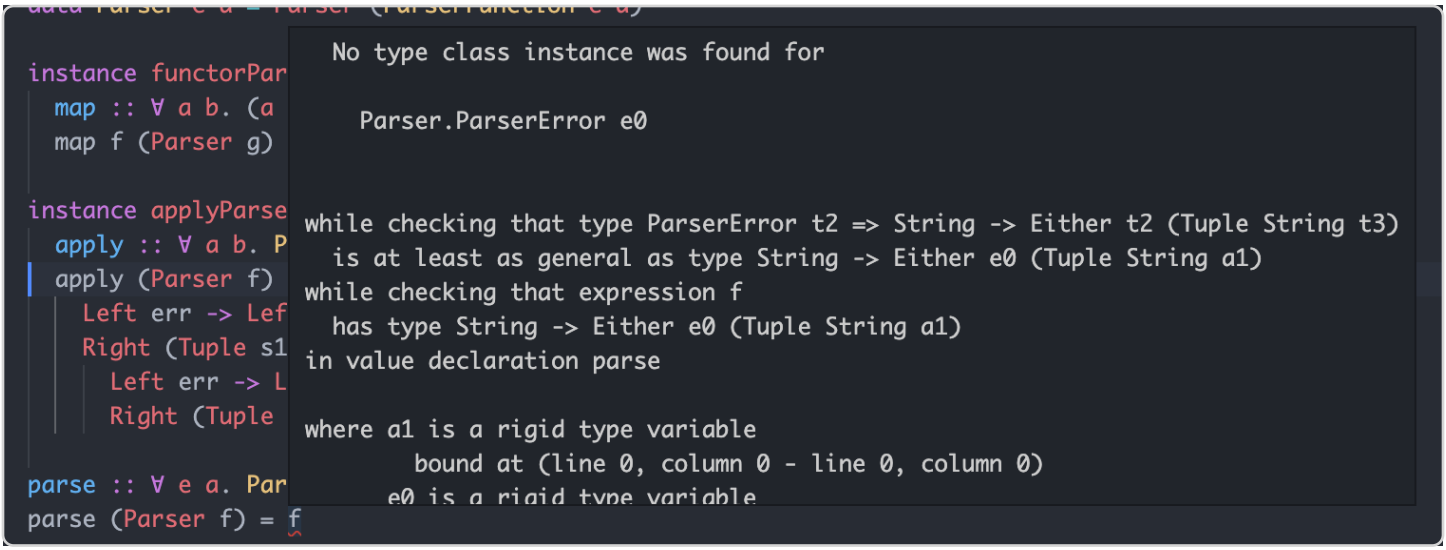
❶ Don't forget the `∀`.

And now we can write the Function:

```
parse :: ∀ e a. Parser e a -> String -> Either e (ParserState a)
parse (Parser f) = f    -- COMPILER ERROR!! ❶
```

❶ Destructure `Parser` to get at the Function and simply return it.

The code looks good but there an unexpected compiler error:



It's saying that `e0` or our `e` doesn't have a instance for `ParserError`. Can you see what we forgot?

Looking back at the `ParseFunction` definition, we can see why:

```
type ParseFunction e a =
  ParserError e => String -> Either e (ParserState a)
```

The `ParseFunction` definition constrains `e`. We should do the same in our Function:

```
parse :: ∀ e a. ParserError e => ❶
  Parser e a -> String -> Either e (ParserState a)
parse (Parser f) = f
```

❶ Added constraint.

That's better. But we could make this Type Signature simpler if we just replace the return value with `ParseFunction`. That's basically what our code is doing. We're just returning `f`. Make the following improvement:

```
parse :: ∀ e a. ParserError e => Parser e a -> ParseFunction e a
parse (Parser f) = f
```

But now since we're using `ParseFunction`, which already has the constraint on `ParserError`, we can safely remove it:

```
parse :: ∀ e a. Parser e a -> ParseFunction e a
parse (Parser f) = f
```

This Function could be used to make our `map` and `apply` Functions a bit easier to understand:

```
instance functorParser :: Functor (Parser e) where
  map :: ∀ a b. (a -> b) -> Parser e a -> Parser e b
  map f p = Parser \s -> map f <$> parse p s ❶

instance applyParser :: Apply (Parser e) where
  apply :: ∀ a b. Parser e (a -> b) -> Parser e a -> Parser e b
  apply p1 p2 = Parser \s -> case parse p1 s of ❷ ❸
    Left err -> Left err
    Right (Tuple s1 h) -> case parse p2 s1 of ❹
      Left err -> Left err
      Right (Tuple s2 x) -> Right $ Tuple s2 (h x)
```

❶ We don't destructure our `Parser`, but just refer to it as `p`.

❷ We don't destructure our `Parsers`, but just refer to them as `p1` and `p2`.

❸ We run the `Parser` p1 here using `parse` and pass it the `String`, `s`.

❹ We run the `Parser` p2 here using `parse` and pass it the `String`, `s1`.

This is a better. Not as good as when we can use `Monads`, but better. It's definitely clearer that we're parsing. Before we just had `f s` and now we've replaced it with `parse p1 s`.

Go ahead and update your code to reflect this.

The next thing we need is a way to parse a single character from the `String`. This `Parser` will be the base `Parser` that all other `Parsers` will be built upon.

Here's the Type Signature to that Function:

```
char :: ∀ e. Parser e Char
```

Go ahead and write the Function. It should construct a `Parser` that contains a Function that will remove the head of the `String` and return it along with the remainder of the `String`, i.e. the tail. Keep in mind, what the return Type is for `ParseFunction` to make sure you construct the proper Type.

Here's a hint, look up `uncons` in `Pursuit` in `Data.String.CodeUnits`. Give it a try and get as far as you can before reading on.

Let's add one line at a time to our Type Signature:

```
import Data.String.CodeUnits (uncons) ❶

char :: ∀ e. Parser e Char
char = Parser \s -> case uncons s of ❷
  ???
```

❶ Import `uncons`.

❷ `uncons` can fail if the `String` is empty, so we'll have to handle the `Maybe`, hence the `case` expression.

Let's handle the error case first:

```
import Data.Maybe (Maybe(..)) ❶

char :: ∀ e. Parser e Char
char = Parser \s -> case uncons s of
  Nothing -> Left ??? ❷
  ???
```

❶ Added this import.

❷ The `???` is supposed to be a Value of a Type that has a Typeclass Instance of `ParserError`. Let's put this off for now.

We'll return to the error case at the end. So, let's move ahead with the success case:

```
char :: ∀ e. Parser e Char
char = Parser \s -> case uncons s of
  Nothing              -> Left ???
  Just { head, tail } -> Right $ Tuple tail head
```

Notice that `head` is what this `Parser` parsed and `tail` is the unused portion of the `String`.

But what about our error case. Remember when we created `ParserError` we didn't add any Methods? Well, now we'll need to add one. We need a Method that will return an error of Type `e` when we run out of `String`.

Let's call the Method `eof` for End Of File, even though it's the end of the `String`. `EOF` is a common acronym and so we're going to leverage that common understanding. Besides, this `String` could come from a file or stream where `EOF` makes sense.

Add `eof` to the `ParserError` Typeclass before continuing.

Here is the expanded definition of `ParserError`:

```
class ParserError (e :: Type) where ❶
  eof :: e ❷
```

❶ Don't forget to add the `where` keyword. We didn't need this before since we had no Methods.

❷ Here `e` is our Application-specific error Type.

Now, any error Type that the user of our general `Parser` creates must at least support `eof` for the `char` parser.

The following is an imaginary error Type for some imaginary Application that uses our `Parser` and meets this requirement:

```
data MyError ❶
  = EOF ❷
  | InvalidThing ❸

instance parserErrorMyError :: ParserError MyError where
  eof = EOF
```

❶ Made up error type.

❷ `EOF` for `char`.

❸ `InvalidThing` is an Application-specific error. This is most likely an error from a higher-level `Parser` that the Application Developer wrote.

Here's another imaginary error Type for a different imaginary Application:

```
data ParsingError ❶
  = NoMoreData ❷
  | MissingStuff String
  | IntOutOfRange Int

instance parserErrorParsingError :: ParserError ParsingError where
  eof = NoMoreData ❸
```

❶ Another made up error Type.

❷ The designer of this error Type decided to signify running out of `String` by the value `NoMoreData`.

❸ When `char` runs out of `String`, it'll use this instance's `eof`, which returns `NoMoreData`.

Remember, the approach we chose allows the user of our `Parser` the flexibility to name their errors whatever they want as we've done in these 2 preceding examples.

Now, we're going to create our own error Type and call it `PError` with a single Data Constructor called `EOF`.

We're also going need to create a `ParserError` Instance for it. Use the above examples to help you code it.

Also, add a `Show` Instance for `PError` since we'll need it later when writing test code.

Please do so before moving forward.

Here's our error Type:

```
import Data.Generic.Rep (class Generic)
import Data.Show.Generic (genericShow) ❶

data PError
  = EOF
derive instance genericPError :: Generic PError _ ❷

instance showPError :: Show PError where ❸
  show = genericShow

instance parserErrorPError :: ParserError PError where ❹
  eof = EOF
```

❶ We need to import `genericShow` since we're going to leverage it in our `Show` Instance.

❷ We need to derive `Generic` so we can use `genericShow`.

❸ Our `Show` Instance.

❹ We make `PError` and Instance of `ParserError` so we can use it with the `Parsers` we're building.

We'll add more Methods to this as the need arises.

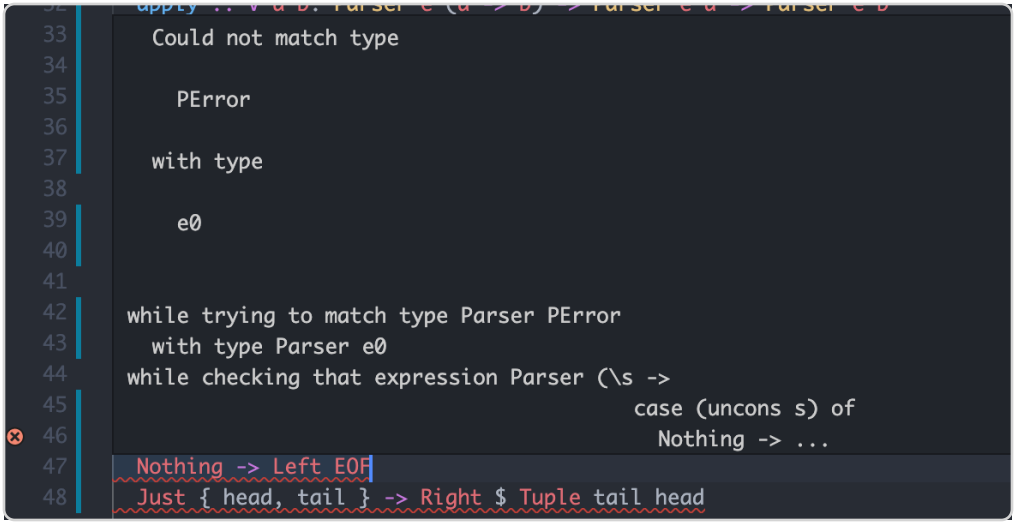
Now, we can get back to the work at hand, finishing the error case for `char`. Please give it try before you read on.

You may have been tempted to write the following:

```
char :: ∀ e. Parser e Char
char = Parser \s -> case uncons s of
  Nothing              -> Left EOF     -- COMPILER ERROR!! ❶
  Just { head, tail } -> Right $ Tuple tail head
```

❶ Compiler error since it cannot match `e0` to `ParserError`.

Here's the error:



Even if this did compile it still would be wrong. We're writing all of our code in a single Module since we're just learning here. So, the compiler knows about `EOF`.

But if we were to remove our `Parser` implementation into its own library, as we would in production code, we would NOT move `PError`. That's because `PError` is an Application-specific error Type. That was the whole point of creating the `ParserError` Typeclass.

`char` would be a base `Parser` in our library, which would have no way of knowing what the user of the library is going to call the error that represents the end of data from the `String`. So, the above code could never be written in that library.

Instead, we need to use `eof`, i.e. the `ParserError` Typeclass Method, to get the correct error Value for the Application-specific error Type, `e`:

```
char :: ∀ e. Parser e Char
char = Parser \s -> case uncons s of
  Nothing             -> Left eof
  Just { head, tail } -> Right $ Tuple tail head
```

Make sure your code uses `eof` instead of `EOF`.

Next, write a `Parser` called `twoChars` that returns a `Tuple Char Char`. Here's a hint before you do. You're going to make use of `<$>` and `<*>`. Give it a try before moving forward.

The Type Signature should be:

```
twoChars :: ∀ e. Parser e (Tuple Char Char)
```

By the way, we don't have to constrain `e` with `ParserError`. Look at the definition to `Parser` and `ParseFunction` to see why:

```
type ParseFunction e a =
  ParserError e => String -> Either e (ParserState a)

newtype Parser e a = Parser e (ParseFunction e a)
```

Can you see why now?

It's because when you replace `ParseFunction e a` in the Type Signature for `twoChars` with it's definition, you get:

```
twoChars :: ∀ e. Parser e (ParserError e =>
                     String -> Either e (ParserState (Tuple Char Char)))
```

which has `e` constrained in the `ParseFunction` definition, which is why we don't have to specify it.

And now we'll continue coding, starting with the `char`, which parses a single `Char`:

```
twoChars :: ∀ e. Parser e (Tuple Char Char)
twoChars = char          -- COMPILER ERROR!! ❶
```

❶ Types don't match since we're not done.

Now we have a single `char` parser that would return a single `Char`. But we want it to return this `Char` as the first part of a `Tuple`. So let's map `Tuple` over the `Parser`:

```
twoChars :: ∀ e. Parser e (Tuple Char Char)
twoChars = Tuple <$> char          -- COMPILER ERROR!! ❶
```

❶ Types don't match since we're still not done.

Now we have our `Tuple` Data Constructor, which is a Function, Partially Applied in a Context, i.e. inside of `Parser`. Luckily for us, `Parser` is an `Applicative`, which means it's also an `Apply`. So we can use `<*>` to apply the Function in one `Parser` to result of another `Parser`.

So let's apply another `Parser`, viz. `char`:

```
twoChars :: ∀ e. Parser e (Tuple Char Char)
twoChars = Tuple <$> char <*> char
```

This probably feels like magic when you just look at the final solution. But it's not.

The code for `<*>` does all the heavy lifting for us. It manages passing the `Strings` from the first `Parser` to the second, handling the `Tuple` return values and error checking:

```
apply :: ∀ a b. Parser e (a -> b) -> Parser e a -> Parser e b
apply p1 p2 = Parser \s -> case parse p1 s of ❶
  Left err -> Left err
  Right (Tuple s1 h) -> case parse p2 s1 of ❷
    Left err -> Left err
    Right (Tuple s2 x) -> Right $ Tuple s2 (h x) ❸
```

❶ Pass the first `String`, `s`.

❷ Pass the remaining `String`, `s1`, from the first `Parser` to the second.

❸ Return the remaining `String`, `s2`, from the second `Parser`.

We get a lot of Functionality out of our tiny little Binary Operator, `<*>`. This is why the code may seem to be magic, since we can easily forget all of the work that being done behind the scenes.

And while we're here, let's take a moment to realize that the `Apply` instance implementation combines `Parsers`, `p1` and `p2`. And each of those `Parsers` could be simple `Parsers` like we saw in `twoChars` or a combinations of other `Parsers` as in the following example:

```
threeChars :: ∀ e. Parser e (Tuple Char (Tuple Char Char))
threeChars = Tuple <$> char <*> twoChars
```

Here `<*>` combines 2 `Parsers` where `twoChars` is, itself, a combination of 2 `Parsers`.

By the way, here is another way to write a `Parser` of 3 characters:

```
data Threeple a b c = Threeple a b c ❶

threeChars' :: ∀ e. Parser e (Threeple Char Char Char)
threeChars' = Threeple <$> char <*> char <*> char
```

❶ Our badly named Type for 3-tuples.

But what if we want our `threeChars` to return a `String` instead? That would be much better than having to pull the characters out of a `Threeple` every time.

Here's a hint, use a Function that takes 3 characters and produces a `String` instead of using `Threeple`.

Now, when you start to wonder how to take 3 characters and put them together to make a `String`, do yourself a favor and look up `fromCharArray` on `Pursuit`.

Rewrite `threeChars` to return a `String` before reading further.

Our Type Signature needs to change:

```
threeChars :: ∀ e. Parser e String
```

Now we should write a Function to accept 3 characters and produce a `String` and then replace the `Threeple` Data Constructor in our previous example, with that Function:

```
import Data.String.CodeUnits (uncons, fromCharArray) ❶

threeChars :: ∀ e. Parser e String
threeChars =
  (\c1 c2 c3 -> fromCharArray [c1, c2, c3]) <$> char <*> char <*> char
```

❶ Added `fromCharArray`.

We haven't tested any of our code yet. So let's remove our placeholder test code and add the following to make sure we don't have bugs:

```
log $ show $ parse char "ABC" ❶
log $ show $ parse twoChars "ABC" ❷
log $ show $ parse threeChars "ABC" ❸
```

❶ Prints `(Right (Tuple "BC" 'A'))`.

❷ Prints `(Right (Tuple "C" (Tuple 'A' 'B')))`.

❸ Prints `(Right (Tuple "" "ABC"))`.

But as soon as you add this, you'll get an error and this is because the compiler cannot figure out what the error Type is. We can fix this like we have in the past by adding explicit Types:

```
log $ show $ (parse char "ABC" :: Either PError _)
log $ show $ (parse twoChars "ABC" :: Either PError _)
log $ show $ (parse threeChars "ABC" :: Either PError _)
```

But this is ugly and cumbersome. So instead, let's simply write a new `parse` Function that specifies the error Type:

```
parse' :: ∀ a. Parser PError a -> ParseFunction PError a ❶
parse' = parse
```

❶ We've replaced the `e` in `parse's` Type Signature with `PError`.

This Function can ONLY return a `PError`, whereas `parse` can return any `e`.

Now we call `parse'` instead in our test code:

```
log $ show $ parse' char "ABC"
log $ show $ parse' twoChars "ABC"
log $ show $ parse' threeChars "ABC"
```

Run your test code by running the command:

```
npx spago run
```

And you'll get the following `spago` error:

```
[error] Some of your project files import modules from packages that are not
in the direct dependencies of your project.
To fix this error add the following packages to the list of dependencies in
your config:
- either
```

Add `either` to `spago.dhall` and run it again.

This should output:

```
(Right (Tuple "BC" 'A'))
(Right (Tuple "C" (Tuple 'A' 'B')))
(Right (Tuple "" "ABC"))
```

The fact that they all return `Rights` tells us that we've successfully parsed in each case.

The `Tuple` has, as its first element, what's left over after the `Parser` completes. In the first result, we can see that it consumed 1 character and returned 2 characters in the remaining `String`, i.e. the second element of the `Tuple`.

The second result consumed 2 characters and left 1 character behind in the `String`. And the final case consumed all 3 characters leaving us with an empty `String`.

We should probably test a failure case. So add the following test case:

```
log $ show $ parse' threeChars "A" ❶
```

❶ Prints `(Left EOF)`.

Go ahead and run that. You should see how it returns our `EOF` error. That's because it ran out of data when processing the `String`. The `Parser` would consume 3 characters but the `String` only had 1.

Now what if I asked you to write a Function that consumes 10 characters? Would you write the following:

```
tenChars :: ∀ e. Parser e String
tenChars = (\c1 c2 c3 c4 c5 c6 c7 c8 c9 c10 ->
  fromCharArray [c1, c2, c3, c4, c5, c6, c7, c8, c9, c10])
  <$> char <*> char <*> char <*> char <*> char <*> char
  <*> char <*> char <*> char <*> char
```

I should hope not or at least you'd question your sanity after you had. I know I am questioning mine as I type this.

What we really need is a better mechanism for taking our `Parser`, `char`, and making it consume a specific number of characters.

But how? First of all, let's look at my insane implementation for `tenChars`. Notice we have 10 instances of `char` and we keep `applying` them to each other. Can you remember what we've seen that does this?

If you said `Traversable`, then kudos to you. If not, don't worry, I didn't think of using `Traversal` either. I cheated and looked at what was done in Haskell's `parsec` library.

So, the first thing that we'll need is a way to duplicate the `Parser` in an `Array`. This can be accomplished with a Function called `replicate` from `Data.Array`.

The next thing we'll need to do is to run `sequence` over it to `apply` the parser over and over again.

Let's refresh our memories regarding the `Traversable` Typeclass:

```
class (Functor t, Foldable t) <= Traversable t where
  traverse :: ∀ a b m. Applicative m => (a -> m b) -> t a -> m (t b)
  sequence :: ∀ a m. Applicative m => t (m a) -> m (t a)
```

In our usage case, `t` is `Array` and `m` is our `Parser e`, which is an `Applicative`.

So we want a `Parser`, which we'll call `count`, that takes a count and a `Parser` that parses an `a` and produces a new `Parser` that produces `Array a`, i.e a bunch of `a's`.

Go ahead and just write the Type Signature for `count` before reading on.

The Type Signature can be written directly from the above description:

```
count :: ∀ e a. Int -> Parser e a -> Parser e (Array a)
```

Notice that the new Parser returns not just 1 `a` but an `Array` of `a`'s. In the case where `Parser` is `char`, we'll need to convert the returned `Array` to a `String` using `fromCharArray` from `Data.String.CodeUnits`.

Next, let's write the rest of the code together. First, we'll write the error case when we're passed a negative count or a zero count. We'll do this using Guards:

```
count :: ∀ e a. Int -> Parser e a -> Parser e (Array a)
count n
  | n <= 0 = pure [] ❶
  ???
```

❶ Remember, the `pure` implementation for `Parser` is a `Parser` that consumes NO characters from the `String` and returns the Pure Value. In this case, an empty `Array`.

And finally, let's work on the valid case:

```
count :: ∀ e a. Int -> Parser e a -> Parser e (Array a)
count n p
  | n <= 0     = pure []
  | otherwise = ???
```

Now, we decided up above that we need to replicate the parser into an `Array` and then `sequence` it. So, let's add that:

```
import Data.Array (replicate) ❶
import Data.Traversable (sequence)

count :: ∀ e a. Int -> Parser e a -> Parser e (Array a)
count n p
  | n < 0     = pure []
  | otherwise = sequence (replicate n p)
```

❶ Don't forget to add these imports.

Now we have a `Parser`, `count`, that will take a count and another `Parser` and produce a `Parser` that will parse the specified number of times and return an `Array` of parsed Values.

So this may seem a bit magical, but let me remind you that `sequence` doesn't actually parse. All it does is

`apply` a bunch of `Applicatives` together, and in our case, a bunch of `Parsers`.

Each time we `apply` 2 Parsers, we wind up with a `Parser` that is a combination of both. We can think of this as building a ever increasingly complex Parser, which ultimate will parse the specified number of times.

It isn't until we call `parse` that the actual parsing happens. It isn't until then that our massively complex Function gets called.

Let's actually call that massive parse Function by adding the following to our test code:

```
log $ show $ parse' (fromCharArray <$> (count 3 char)) "xyz" ❶ ❷
```

❶ Prints `(Right (Tuple "" "xyz"))`. `"xyz"` is the returned `String` and `""` is what's left over after parsing.

❷ Notice that we're mapping `fromCharArray` into our `Parser`. This way, it'll return a `String` and not an `Array Char`. It's so much easy to work with `Strings`.

Go ahead and run the code to see it in action.

And when you do, `spago` complains:

```
[error] Some of your project files import modules from packages that are not
in the direct dependencies of your project.
To fix this error add the following packages to the list of dependencies in
your config:
- arrays
```

Add `arrays` to `spago.dhall` and rerun to see if it prints out the expect result.

Now that our `count` Function is done and working, we should examine it for areas of improvement:

```
count :: ∀ e a. Int -> Parser e a -> Parser e (Array a)
count n p
  | n <= 0     = pure []
  | otherwise = sequence (replicate n p)
```

Can you see anything? Can we make it more general?

Well, as it turns out, we can make it more general. It might be nice if we could parse into a `List` or an `Array` or any `Traversable`. Let's do that.

First, we're going to need to abstract `Array`:

```
import Data.Traversable (class Traversable, sequence) ❶

count :: ∀ e a f. Traversable f => Int -> Parser e a -> Parser e (f a) ❷
```

❶ We added `class Traversable` to the import list.

❷ We've abstracted `Array` with our Type Parameter `f`. Now `f` can stand for `List` or `Array`, and even `MyTree` (assuming your tree has a `Traversable` Instance).

Next, we should recode the Function. First the error case:

```
count :: ∀ e a f. Traversable f => Int -> Parser e a -> Parser e (f a)
count n p
  | n <= 0 = ???
```

Now we're stuck. We need an empty value here. How can we get an empty value when we don't know what type `f` is?

If you're screaming out `Monoid`, you're on the right track. But I'm going to save you the compiler errors by mentioning that `f's` Kind is `Type -> Type`, but `Monoid's` Type is just `Type`. So that won't work.

We need a `Monoid` for `Type -> Type`. Do you remember what that's called?

Did you say `Plus`? I thought so.

So let's add that constraint and use `empty` from `Plus`:

```
import Control.Plus (class Plus, empty) ❶

count
  :: ∀ e a f
  .  Plus f
  => Traversable f
  => Int
  -> Parser e a
  -> Parser e (f a)
count n p
  | n <= 0 = pure empty ❷
  ???
```

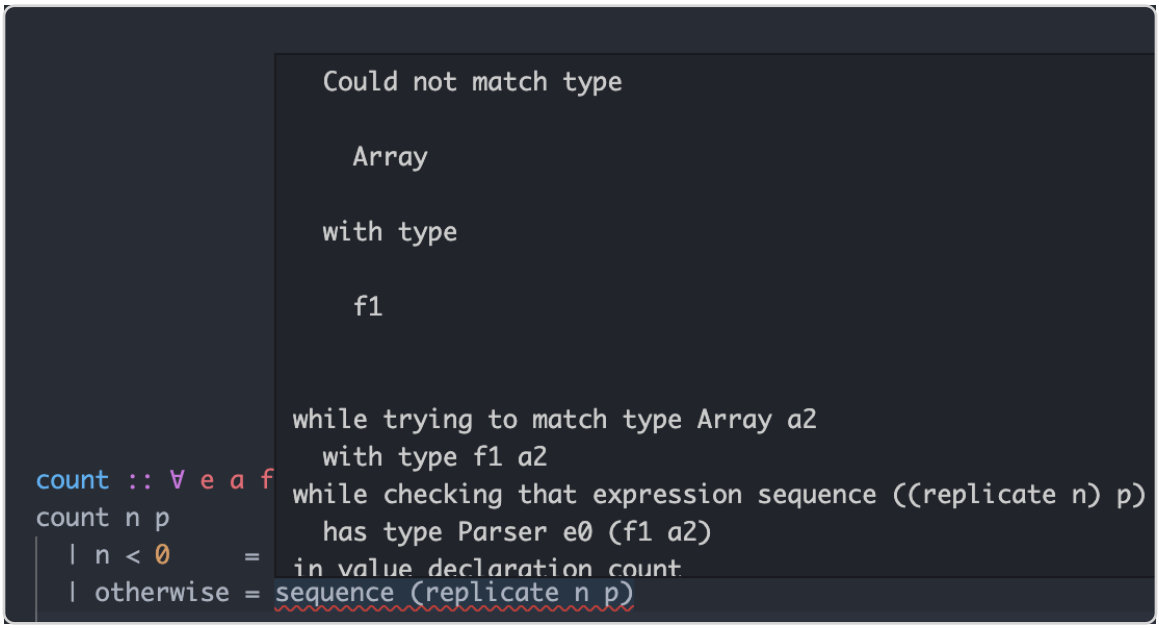
❶ We're going to need to explicitly import these.

❷ We return an `empty` value here, e.g for `List` it's `Nil` and for `Array` it's `[]`.

Now we're ready to write the valid case:

```
count
  :: ∀ e a f
  .  Plus f
  => Traversable f
  => Int
  -> Parser e a
  -> Parser e (f a)
count n p
  | n <= 0    = pure empty
  | otherwise = sequence (replicate n p) -- COMPILER ERROR!!
```

Unfortunately, we get the following compiler error:



It cannot match `Array` with `f1`, that's our `f`. Why would the compiler be complaining about an `Array` when we removed that from the Type Signature? Can you see why? What does `replicate` return?

Go look up `replicate` in `Pursuit` and see.

What did you find? There are lots of `replicate` Functions and we imported the one from `Data.Array`. That version returns an `Array`. We need to remove that.

So do that now.

But now we need a more general version of `replicate`. Consulting `Pursuit`, we find one in `Data.Unfoldable` that looks really general:

```
replicate :: ∀ f a. Unfoldable f => Int -> a -> f a
```

But what's `Unfoldable`?

Back when we talked about Folds, I mentioned Unfolds and how they were ***Anamorphisms***, i.e. Functions that build up structure as opposed to Folds which are ***Catamorphisms*** that break down structure. Well, `replicate` starts with a seed, in our case a `Parser`, and it produces a bunch of them in an `Array` or `List` or some another `Unfoldable` from that one `Parser`.

So, we probably can use this since `Array` and `List` have `Unfoldable` instances. You can see this fact in `Pursuit`, but you'll have look in different places for each. Since `Array` is a base Type in PureScript, the `Unfoldable` Typeclass code implements it. In the case of `List`, which is not a base Type, it's implemented in the same Module as `List`.

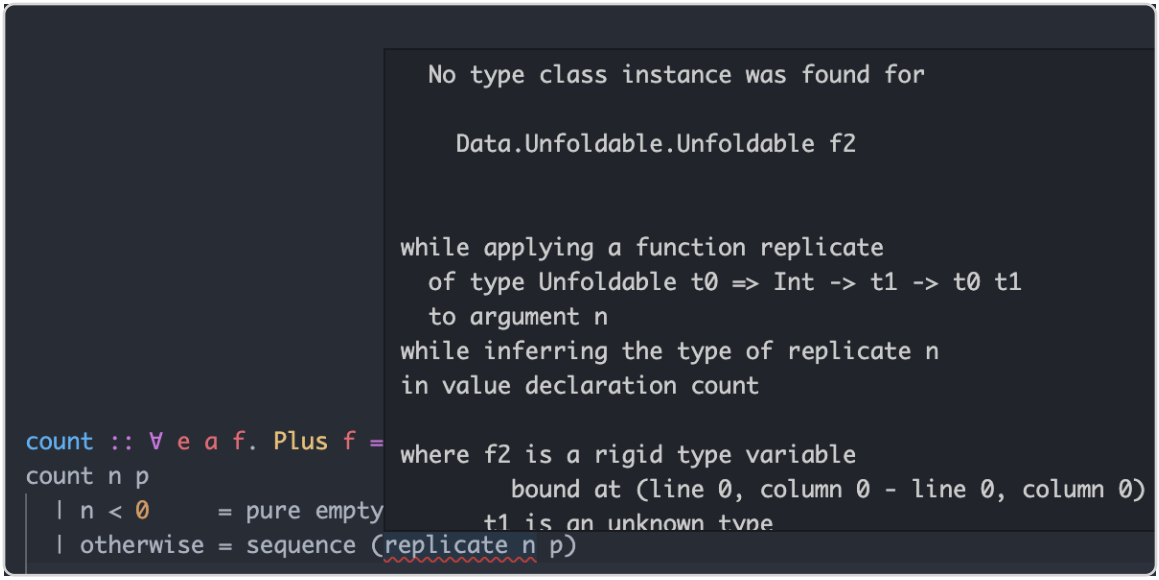
Remember Orphaned Instances? Well, that's why you'll find these in those 2 different places. Those are the only valid places one can put instances.

So let's add the import for `replicate`, the `Unfoldable` version:

```
import Data.Unfoldable (replicate)

count
  :: ∀ e a f
  .  Plus f
  => Traversable f
  => Int
  -> Parser e a
  -> Parser e (f a)
count n p
  | n <= 0     = pure empty
  | otherwise = sequence (replicate n p) -- COMPILER ERROR!!
```

Once again, the compiler is complaining:



Can you see what we forgot?

We're missing the `Unfoldable` constraint on `f`. We need it since we're using the `Unfoldable` version of `replicate`, which also has that constraint in it's Type Signature. So, let's add it:

```
import Data.Unfoldable (class Unfoldable, replicate) ❶

count
  :: ∀ e a f
  .  Plus f
  => Traversable f
  => Unfoldable f ❷
  => Int
  -> Parser e a
  -> Parser e (f a)
count n p
  | n <= 0     = pure empty
  | otherwise = sequence (replicate n p)
```

❶ Have to import the class before we can use it in a Constraint.

❷ Constrain `f` to be `Unfoldable`.

And hurray! No compiler errors!

Let's run it just in case... Good thing:

```
[error] Some of your project files import modules from packages that are not
in the direct dependencies of your project.
To fix this error add the following packages to the list of dependencies in
your config:
- control
- unfoldable
```

Add `control` and `unfoldable` to `spago.dhall`.

And after rerunning it, our test prints out `(Right (Tuple "" "xyz"))`, which is correct!

But we get the following warning:

```
[warn] None of your project files import modules from some projects that are
in the direct dependencies of your project.
These dependencies are unused. To fix this warning, remove the following
packages from the list of dependencies in your config:
- arrays
```

We can safely remove this from `spago.dhall` now that `count` doesn't need `Data.Array`.

There's one more minor change we should make. In `Data.Unfoldable`, there's a Function called `none` that returns an empty `Unfoldable` collection. This means, that we can remove the `Plus` constraint if we use `none` instead of `empty`. The fewer constraints we have, the better:

```
import Data.Unfoldable (class Unfoldable, replicate, none) ❶ ❷

count
  :: ∀ e a f
  .  Traversable f ❸
  => Unfoldable f
  => Int
  -> Parser e a
  -> Parser e (f a)
count n p
  | n <= 0     = pure none ❹
  | otherwise = sequence (replicate n p)
```

❶ Add `none` to the imports.

❷ We deleted the `Control.Plus` import.

❸ We removed the `Plus` constraint

❹ Changed `empty` to `none`.

Running this new version, we get:

```
[warn] None of your project files import modules from some projects that are
in the direct dependencies of your project.
These dependencies are unused. To fix this warning, remove the following
packages from the list of dependencies in your config:
- control
```

Remove `control` from `spago.dhall` since we're not using `Control.Plus` anymore.

Changing `count` to be more general was a lot of extra work but it paid off in the end. We could've lived the rest of our lives with this Function returning an `Array`, and many times generalizing a Function isn't worth it.

But, in this case, we're trying to write a general parsing library and we don't know who in the world might use it and under what conditions. So making this change is well worth it. Besides, the more you generalize, the easier it gets. So this is good practice.

We're going to leave this parser as is for now. We'll return to it after we learn about `Monads` to make it a ***Monadic Parser***, which can parse things an Applicative Parser cannot. We'll also see how `Alternative` helps us write more complex parsers.

Chapter 18. Monads

Monads are the most poorly explained area of Functional Programming. This history of bad explanations comes from people who have just learned them and try to explain them in online forms or in blog posts. But the problem is that they don't fully understanding them yet. Or they only partially understand them.

Another problem with many explanations is that they try to use analogies to make it "easy to understand". This only serves to obscure the subject further and in the worst cases, it misleads.

Unfortunately, because of this, Monads have gotten a bad reputation for being difficult to understand. In reality, they aren't any more difficult to understand than Functors or Applicatives. They are, however, much more powerful. And when we combine them, their powers multiply.

Monads are really the meat-and-potatoes of Functional Programming since we interface with the outside world in real-world applications and Monads help us to do that in a purely Functional way.

18.1. Side-effects

When you interface with the outside world, you're dealing in the realm of ***Side-effects***. For example, when you read from a file or call a Web API, the value that's returned from that Function will NOT be deterministic, i.e. it could be different for 2 different consecutive calls with the same inputs.

These operations are very different from Pure Functions, which always return the same results for the same inputs. This is why Global Variables are forbidden in Functional Languages because they would introduce Side-effects into every single Pure Function that accessed them.

Let's first look at an example of how Global Variables can be used in Javascript to maintain a log of calls made to Functions:

```
var log = '';
const f = x => { x + 10; log += 'added 10\n'; };
const g = y => y * 100; log += 'multiplied by 100\n'; };
const h = z => g(f(z));
const h2 = z => g(f(z)) + g(f(z));
```

Here the Global Variable `log` is updated each time `f` or `g` is called, i.e. these Functions produce a Side-effect.

This means that `f` and `g` are NOT Pure. Looking at the Function, `h2`, we can see that we can't simply evaluate `g(f(z))` once and then double its Value like we can in a Purely Functional Language. These Functions need to be called again and again so that their Side-effects can occur.

But we're learning Functional Programming using PureScript, which is Pure. So, how do we implement this sort of Functionality in a Pure way using PureScript (or Haskell)?

With Pure Functions, all we can do is pass values in and get values out. So let's write `f` and `g` in PureScript using that approach:

```
f :: Int -> Tuple String Int
f x = Tuple "added 10\n" (x + 10)

g :: Int -> Tuple String Int
g y = Tuple "multiplied by 100\n" (y + 100)
```

Notice how each Function returns both the computation and the Side-effect. Our result is no longer a Pure Value. Instead, it's a Pure Value in a Context. That Context contains the Side-effect of that computation.

So, we haven't broken any laws yet since we kept our Functions Pure. Before reading further, think about how we can compose these 2 Functions to produce `h` like we saw up above.

To see how we can compose `f` and `g` to produce `h`, let's look at versions of `f` and `g` without Side-effects:

```
f' :: Int -> Int
f' x = x + 10

g' :: Int -> Int
g' y = y + 100
```

With these Pure Functions, we can easily compose them to get `h'`:

```
h' = g' <<< f'
```

But with `f` and `g`, we cannot do this, since `f` returns `Tuple String Int` and `g` takes an `Int`:

```
f :: Int -> Tuple String Int
f x = Tuple "added 10\n" (x + 10)

g :: Int -> Tuple String Int
g y = Tuple "multiplied by 100\n" (y + 100)
```

So, we'll need to write a more complicated Function to compose them. Let's do that one line at a time:

```
h :: Int -> Tuple String Int
```

`h` must return the Side-effects it gets back from calling `f` and `g`, which will be returned in the first position

of the `Tuple`.

Next, we add the `z` parameter (see the Javascript version of `h`):

```
h :: Int -> Tuple String Int
h z = ???
```

The next step calls `f` and captures both the computational result and the Side-effect:

```
h :: Int -> Tuple String Int
h z =
  let Tuple s r = f z
  ???
```

Here the computational Result is `r` and the Side-effect is `s`.

Next, we need to call `g` with the result of `f`, which is `r`. Then we'll capture `g's` result and Side-effect:

```
h :: Int -> Tuple String Int
h z =
  let Tuple s r  = f z
      Tuple s' t = g r
  ???
```

We capture `g's` computational Result in `t` and the Side-effect in `s'`.

And finally, we have to return the final computational Result and the overall Side-effect. The final computational result is easy. It's just `t`.

But what's the overall Side-effect?

Looking back out our Javascript implementation, we can see that when we call `f`, the `log` will first contain the `String "added 10\n"` and then when we call `g`, `"multiplied by 100\n"` will appended to it. We should do the same here by simply concatenating `f's` Side-effect with `g's`:

```
h :: Int -> Tuple String Int
h z =
  let Tuple s r  = f z
      Tuple s' t = g r in
  Tuple (s <> s') t
```

18.1.1. Composing Side-effect Functions

Now that we have a working Function, we should step back and ask ourselves can we make this better?

It's not obvious how we can. And that's too bad because the Pure version, `h'`, was so simple:

```
h' = g' <<< f'
```

It's clear that `h'` is superior to `h`. The code for `h` is painful to write and the process is error-prone due to its complexities. With `h'`, we can just inline the solution, but with `h`, we're forced to write a Function to compose `f` and `g`.

This is hardly ideal. Maybe we can write some library Functions that will make things better.

Let's try that by writing a compose Function that will let us compose Side-effect Functions. How might we do that?

That's a hard question to answer in one step. Instead, let's attempt something a bit more attainable. We'll write a Function that lets us compose Functions that return, as their Side-effect, a `String` stating that they were called and what they did.

Let's call it `composeDebuggable` since it'll compose any Function that returns debugging information:

```
composeDebuggable :: ???
```

Writing the Type Signature here isn't obvious. Let's look at the Type Signature for the regular `compose` Function to see if we can gain any insight. Looking to `Pursuit` we find:

```
class Semigroupoid a where
  compose :: ∀ b c d. a c d -> a b c -> a b d

infixr 9 compose as <<<
```

Wow. That's not confusing at all. Okay, so first, what's a `Semigroupoid`?

A `Semigroupoid` is a `Category` minus the identity morphism requirement.

Remember, a Category consists of Objects and Morphisms. Each Object has an Identity Morphism (this part is NOT part of a `Semigroupoid`). And Morphisms are composable (this part IS part of a `Semigroupoid`).

We can generalize `Semigroupoids` as composable things. And if that's true, then we can make `Function` an instance of `Semigroupoid` since `Functions` can be composed:

```
instance semigroupoidFn :: Semigroupoid (->) where ❶
  compose :: ∀ b c d. (c -> d) -> (b -> c) -> (b -> d) ❷
  compose f g x = f (g x)
```

❶ Remember, `(->)`, is the Type Operator for the Type Constructor, `Function`.

❷ Unlike the Type Signature for compose from `Semigroupoid`, we're writing the specialized Type Signature for `Function` using infixed notation. So everywhere you see `->` think of `a` but prefixed to see how this Type Signature relates to the `Semigroupoid` one.

Let's pull out just the Type Signature and see if we can write an analogous one for `composeDebuggable`:

```
compose
  :: ∀ b c d
  .  (c -> d)
  -> (b -> c)
  -> (b -> d)

composeDebuggable
  :: ∀ b c d
  .  (c -> Tuple String d)
  -> (b -> Tuple String c)
  -> (b -> Tuple String d)
```

Notice that the output of every Function in `composeDebuggable` is prefixed with `Tuple String` when compared with `compose`.

Here's the full implementation, which is nearly identical to what we wrote for `h` above:

```
composeDebuggable
  :: ∀ b c d
  .  (c -> Tuple String d)
  -> (b -> Tuple String c)
  -> (b -> Tuple String d)
composeDebuggable g f x =
  let Tuple s r  = f x
      Tuple s' t = g r in
  Tuple (s <> s') t
```

Notice how `composeDebuggable` manages the Side-effect for us. It squirrels them away into Variables, `s` and `s'`, until it's time to combine them and return the results.

We now can rewrite `h` to use `composeDebuggable`:

```
h :: Int -> Tuple String Int
h = g `composeDebuggable` f
```

This looks very close to our Pure version, `h'`:

```
h' :: Int -> Int
h' = g `compose` f ❶
```

❶ I chose not use `<<<` to make the similarity even more obvious.

`composeDebuggable` now lets us compose these Side-effect Functions, but it's only limited to ones that return `Tuple String Int`. At this point, writing a more general compose Function is very difficult and maybe impossible. Or it's possible, but so complex that it's inconvenient to use.

But no matter, we've got a good solution for these kinds of Functions, which means we can compose them.

But our world is filled with lots and lots of Pure Functions. And we cannot use `compose` or `composeDebuggable` to compose a Pure Function with our Side-effect one. It sure would be nice if we could compose Pure Functions with our Side-effect ones.

18.1.2. Composing Side-effect Function with Pure Ones

Let's write a Function that converts a Pure Function into a Side-effect one. This will let us compose all the Pure Functions in the world with our Side-effect ones.

We'll call it `makeFuncDebuggable`. First the Type Signature:

```
makeFuncDebuggable :: ∀ a b. (a -> b) -> (a -> Tuple String b) ❶
```

❶ Implicit, Right-Associative Parentheses added.

`makeFuncDebuggable` will ***lift*** our Function into one that has a Side-effect, i.e. it takes a Function that normally returns a Pure Value, `b`, and makes it return a Side-effect Value, `Tuple String b`.

Next, we'll write the implementation:

```
makeFuncDebuggable :: ∀ a b. (a -> b) -> a -> Tuple String b ❶
makeFuncDebuggable f x = Tuple "" (f x)
```

❶ Removed redundant Parentheses.

Notice how we use an empty `String` so that NOTHING is added to the log. This should make you think of `mempty` from `Monoid`, since `composeDebuggable` uses the `Semigroup` Operator, `<>`, to combine the

`Strings`.

Here's an example of how we might use `makeFuncDebuggable`:

```
nse :: Int -> Int
nse x = x + 42

f :: Int -> Tuple String Int
f x = Tuple "added 10\n" (x + 10)

c :: Int -> Tuple String Int
c = makeFuncDebuggable nse `composeDebuggable` f (1)
```

(1) The non-side-effect Function, `nse`, is lifted into one that produces Side-effects using `makeFuncDebuggable`, which means that we can now compose it with our Side-effect Function, `f`, using `composeDebuggable`.

18.1.3. Function Application with Side-effect Functions

Composing Functions is great but what about Function Application? We don't always want to compose Functions. Sometimes we'd like to just call them.

Once again, we'll start by looking at how Function Application works with Pure Functions. Let's see what happens if we try to do Function Application with our Pure Functions, `f'` and `g'`:

```
y' :: Tuple String Int
y' = 12345 # f' # g'
```

Here `12345` is first passed to `f'` and the `Int` result is then passed to `g'`.

But what happens if we try this same approach with our Side-effect versions, `f` and `g`:

```
y :: Tuple String Int
y = 12345 # f
```

This is as far as we could go. We cannot apply the results to `g` since it expects an `Int` as its input and applying `12345` to `f` returns a `Tuple String Int`.

Maybe we can solve this problem, like we did for composing, by writing a Side-effect version of `#`. Let's call it `applyDebuggable`.

Let's first look at the Type Signature for `#`, i.e. `applyFlipped`:

```
applyFlipped
  :: ∀ a b
  .  a
  -> (a -> b)
  -> b
```

Notice that we take a Pure Value, `a`, and a Pure Function, `a -> b`, and apply them to get a Pure Value, `b`.

Now, we'll make the equivalent for our Side-effect Functions:

```
applyDebuggable
  :: ∀ a b
  .  Tuple String a
  -> (a -> Tuple String b)
  -> Tuple String b
```

Notice that we take a Side-effect Value, `Tuple String a`, and a Side-effect Function, `(a -> Tuple String b)`, and apply them to get a Side-effect Value, `Tuple String b`.

And now, here's the implementation.

```
applyDebuggable
  :: ∀ a b
  .  Tuple String a
  -> (a -> Tuple String b)
  -> Tuple String b
applyDebuggable (Tuple s x) f = ❶
  let Tuple s' r = f x in ❷
  Tuple (s <> s') r ❸
```

❶ We destructure the Side-effect Value to get at its Pure Value, `x`, and its Side-effect Value, `s`.

❷ We apply `f` to the Pure Value, `x`, which results in a Side-effect Value, `Tuple s' r` since `f` is a Side-effect Function.

❸ We manage the Side-effects, i.e. we combine them and return both the Side-effect Value and the Pure Value.

Just like `composeDebuggable`, this manages the Side-effects behind the scenes. In fact, `applyDebuggable` looks a lot like `composeDebuggable`. We should keep this in mind. We may be able to write one of them in terms of the other.

For now though, let's see how we can use `applyDebuggable` in our earlier example:

```
y :: Tuple String Int
y = (12345 # f) `applyDebuggable` g
```

That works nicely. But it turns out that we can eliminate the Pure Function apply, #, by have a Function that makes Pure Values debuggable like we made Pure Functions debuggable with `makeFuncDebuggable`:

```
makeDebuggable :: ∀ a. a -> Tuple String a
makeDebuggable x = Tuple "" x
```

This Function looks a lot like `makeFuncDebuggable`. We should also keep this in mind. We may want to consolidate all these Functions that we're inventing.

But, for now, we can use `makeDebuggable`, which lets us drop # and use `applyDebuggable` in its place:

```
y :: Tuple String Int
y = makeDebuggable 12345 `applyDebuggable` f   `applyDebuggable` g
--  (                12345          #        f ) `applyDebuggable` g
--  (                12345          #        f')         #         g'
```

18.2. `Debuggable` Type

We've been talking about our Functions as being ***debuggable*** and about the Side-effect Type that they return as being a `Tuple String Int`.

At this point, it's probably time to create a Type that captures these facts. We're going to make a Type Alias that's a bit more general though:

```
type Debuggable a = Tuple String a ❶
```

❶ Instead of `Int`, we've made this Type Alias polymorphic by adding the Polymorphic Type Parameter, `a`.

Now, let's rewrite the Type Signatures of the Side-effect Functions we've created to use our new Type Alias:

```
composeDebuggable
  :: ∀ b c d
  .  (c -> Debuggable d)
  -> (b -> Debuggable c)
  -> (b -> Debuggable d)
composeDebuggable g f x =
  let Tuple s r  = f x
      Tuple s' t = g r in
  Tuple (s <> s') t

applyDebuggable
  :: ∀ a b
  .  Debuggable a
  -> (a -> Debuggable b)
  -> Debuggable b
applyDebuggable (Tuple s x) f =
  let Tuple s' r = f x in
  Tuple (s <> s') r

makeFuncDebuggable :: ∀ a b. (a -> b) -> a -> Debuggable b
makeFuncDebuggable f x = Tuple "" (f x)

makeDebuggable :: ∀ a. a -> Debuggable a
makeDebuggable x = Tuple "" x
```

See them all together, we can see that `applyDebuggable` does almost the same things as `composeDebuggable`.

Let's write one in terms of the other:

```
composeDebuggable
  :: ∀ b c d
  .  (c -> Debuggable d)
  -> (b -> Debuggable c)
  -> (b -> Debuggable d)
composeDebuggable g f x = f x `applyDebuggable` g ❶

applyDebuggable
  :: ∀ a b
  .  Debuggable a
  -> (a -> Debuggable b)
  -> Debuggable b
applyDebuggable (Tuple s x) f =
  let Tuple s' r = f x in
  Tuple (s <> s') r
```

❶ `f x` returns a Value of Type `Debuggable c`, which can be applied to `g`.

We've reduced duplication that we recognized earlier in 2 of our Functions.

Now, turning our attention back to the other 2 Functions that we recognized as being very similar, reveals that `makeFuncDebuggable` and `makeDebuggable` are practically the same Function.

So, once again, let's write one in terms of the other:

```
makeFuncDebuggable :: ∀ a b. (a -> b) -> a -> Debuggable b
makeFuncDebuggable f x = makeDebuggable (f x)

makeDebuggable :: ∀ a. a -> Debuggable a
makeDebuggable x = Tuple "" x
```

So we have the following generalized helper Functions for `Debuggable`:

```
composeDebuggable
  :: ∀ b c d
  .  (c -> Debuggable d)
  -> (b -> Debuggable c)
  -> (b -> Debuggable d)
composeDebuggable g f x = f x `applyDebuggable` g

makeFuncDebuggable :: ∀ a b. (a -> b) -> a -> Debuggable b
makeFuncDebuggable f x = makeDebuggable (f x)

makeDebuggable :: ∀ a. a -> Debuggable a
makeDebuggable x = Tuple "" x

applyDebuggable
  :: ∀ a b
  .  Debuggable a
  -> (a -> Debuggable b)
  -> Debuggable b
applyDebuggable (Tuple s x) f =
  let Tuple s' r = f x in
  Tuple (s <> s') r
```

18.3. Generalizing `Debuggable`

Let's imagine a Side-effect Type like `Debuggable` but slightly different. Let's call it `Countable` with the following definition:

```
type Countable a = Tuple Int a
```

With `Countable`, we're counting things and collecting them in an `Int`. The Polymorphic Parameter, `a`, is the computational result like it was before.

Now, we could write a whole new set of Functions for `Countable` that matches the same Functions we wrote for `Debuggable`. But then we'd have to do that same process again and again for other Type Aliases.

We've seen this pattern before. Multiple Types and a common set of Functions. Can you think of what that could be?

If you said Typeclass, then you're thinking what I'm thinking. So, let's make one. We're going to call our new Typeclass `SideEffect`:

```
class SideEffect s where
  ???
```

At this point, we should be wondering which of our helper Functions should be Methods in our new Typeclass. Since we wrote `composeDebuggable` and `makeFuncDebuggable` in terms of other Functions, we can write those as helper Functions with Constraints, which means we can exclude them from our Methods.

The other 2 Functions need to be written specifically for the Type, so we'll add them with more general names:

```
class SideEffect s where ❶
  makeSideEffect :: ∀ a. a -> s a
  applySideEffect :: ∀ a b. s a -> (a -> s b) -> s b
```

❶ `s`, our Side-effect Type, would be replaced with `Debuggable` or `Countable` or whatever.

But wait. Look carefully at `makeSideEffect's` Type Signature. Does that look familiar?

If not, then how about after I change the Type Parameter `s` to `f`:

```
class SideEffect f where
  makeSideEffect :: ∀ a. a -> f a
  applySideEffect :: ∀ a b. f a -> (a -> f b) -> f b
```

How about now?

If you said `pure`, then you get a gold star. It's exactly the same as `pure`, which means that `SideEffect` is at least an `Applicative`. So, let's add that fact and remove `makeSideEffect` since we'll just use `pure` instead:

```
class Applicative f <= SideEffect f where ❶
  applySideEffect :: ∀ a b. f a -> (a -> f b) -> f b
```

❶ `Applicative` is a Superclass of `SideEffect`.

Now we'll have to make our `Debuggable` and `Countable` real Types, not just Aliases, so that we can make them instances of `SideEffect`:

```
newtype Debuggable a = Debuggable (Tuple String a)
newtype Countable a = Countable (Tuple Int a)
```

And now we'll make them both instances of `SideEffect`, starting with `Debuggable`:

```
newtype Debuggable a = Debuggable (Tuple String a)
derive newtype instance functorDebuggable :: Functor Debuggable
derive newtype instance applyDebuggable :: Apply Debuggable
derive newtype instance applicativeDebuggable :: Applicative Debuggable
```

Here we derive via `newtype` all the way up to `Applicative` since we need an `Applicative` Instance to implement `SideEffect`. We start with `Functor` then `Apply` and finish with `Applicative`.

Next, we'll write the `SideEffect` Instance, but not before consulting our `applyDebuggable` Function:

```
applyDebuggable
  :: ∀ a b
  .  Debuggable a
  -> (a -> Debuggable b)
  -> Debuggable b ❶
applyDebuggable (Tuple s x) f =
  let Tuple s' r = f x in
  Tuple (s <> s') r
```

❶ Remember, when we originally wrote this, `Debugable` was a Type Alias for `Tuple String a`.

Now all we have to do is add the `Debuggable` Data Constructor in the appropriate places, since `Debuggable` is now a `newtype`:

```
instance sideEffectDebuggable :: SideEffect Debuggable where
  applySideEffect (Debuggable (Tuple s x)) f = ❶
    let Debuggable (Tuple s' r) = f x in ❶
    Debuggable $ Tuple (s <> s') r ❷
```

❶ In these 2 locations, the Data Constructor, `Debuggable`, is used for destructuring via Pattern Matching.

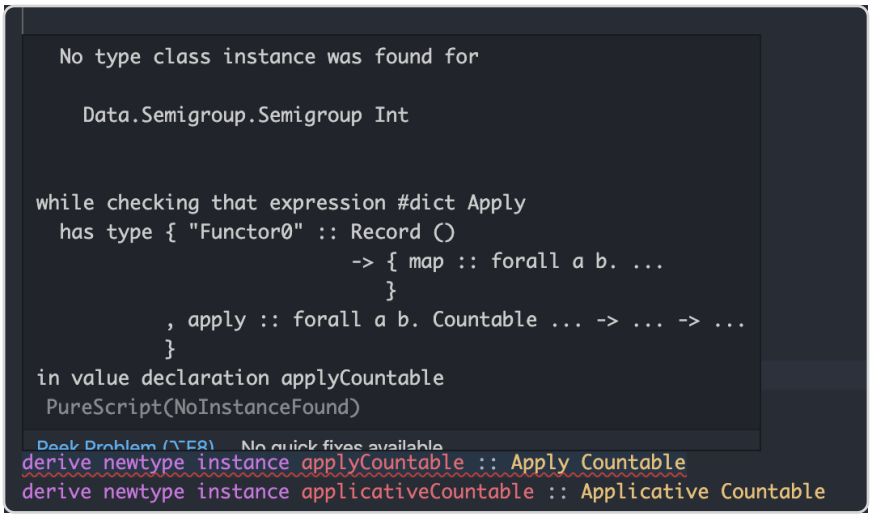
❷ Here we construct using the Data Constructor.

Now we can write the same for `Countable`:

```
newtype Countable a = Countable (Tuple Int a)
derive newtype instance functorCountable :: Functor Countable
derive newtype instance applyCountable :: Apply Countable ❶
derive newtype instance applicativeCountable :: Applicative Countable
```

❶ COMPILER ERROR!!

The compile is complaining. Here's the error:



Well, it's clear from the error that the compiler cannot derive `Apply` without `Int` having a `Semigroup` Instance.

But why?

It turns out that when we use `derive newtype`, we're deriving `Apply` by delegating to the underlying Type's `Apply` Instance. In this case, that underlying Type is `Tuple`, which if you remember correctly, has a `Semigroup` Constraint:

```
instance applyTuple :: Semigroup a => Apply (Tuple a) where ❶
  apply (Tuple x f) (Tuple y z) = Tuple (x <> y) (f z)
```

❶ Here's the `Semigroup` Constraint.

Remember, there are multiple ways to combine `Int`s, e.g. addition or multiplication. So that's why `Int` doesn't have a `Semigroup` Instance. But, we can solve this problem by creating a `newtype` that wraps an `Int`, and then we'll write a `Semigroup` Instance for that `newtype`.

Let's call it `Count`:

```
newtype Count = Count Int
```

Next, we'll write `Semigroup` Instance for `Count` where we use Addition to combine Counts:

```
instance semigroupCount :: Semigroup Count where
  append = (+) ❶
```

❶ `(+)` is the Operator Alias for `add`, a `Semiring` Method.

And since we're using `(+)` from `Semiring`, we should derive `Semiring`:

```
newtype Count = Count Int
derive newtype instance semiringCount :: Semiring Count ❶ ❷
```

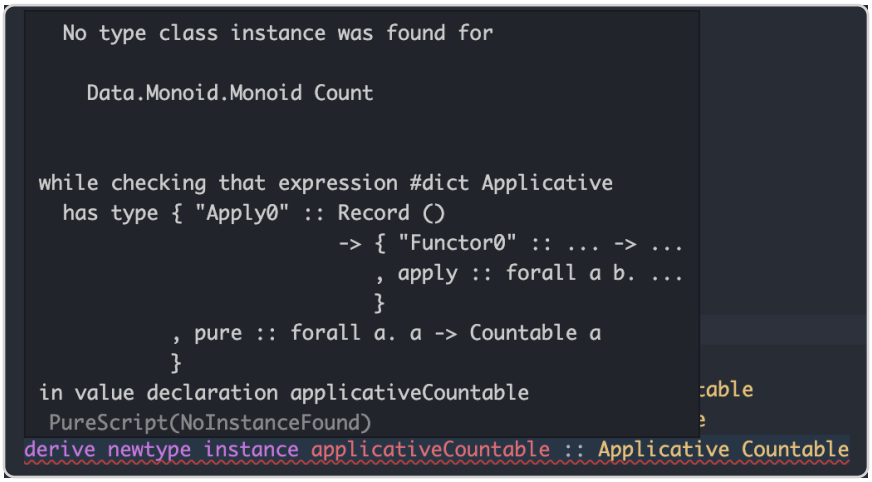
❶ This will allow us to use `(+)`.

❷ There's no reason to write this since `Int` is a `Semiring`.

Next, we'll make `Countable` wrap `Count` instead of `Int`:

```
newtype Countable a = Countable (Tuple Count a)
```

But now we get a brand new compiler error on the derive for `Applicative`:



Once again, Tuple is dictating that `Count` must have a `Monoid` Instance. This is because `Tuple's` Instance implementation for `Applicative` constrains `a`, which in our case is `Count`, to be a `Monoid`:

```
instance applicativeTuple :: Monoid a => Applicative (Tuple a) where ❶
  pure x = Tuple mempty x
```

❶ Here's the `Monoid` Constraint.

We'll fix the error by writing a `Monoid` Instance for `Count`:

```
instance monoidCount :: Monoid Count where
  mempty = zero ❶
```

❶ `zero` is from `Semiring`, which is `0`, which is the Empty Element or Identity Element for `Int` and `(+)`.

And finally, we can write the `SideEffect` Instance for `Countable`:

```
instance sideEffectCountable :: SideEffect Countable where
  applySideEffect (Countable (Tuple c x)) f =
    let Countable (Tuple c' r) = f x in
    Countable $ Tuple (c + c') r ❶
```

❶ We combine by adding the counts. We could've used `<>` since the underlying implementation simply calls `(+)`.

And our final implementation is:

```
class Applicative f <= SideEffect f where
  applySideEffect :: ∀ a b. f a -> (a -> f b) -> f b

newtype Debuggable a = Debuggable (Tuple String a)
derive newtype instance functorDebuggable :: Functor Debuggable
derive newtype instance applyDebuggable :: Apply Debuggable
derive newtype instance applicativeDebuggable :: Applicative Debuggable

instance sideEffectDebuggable :: SideEffect Debuggable where
  applySideEffect (Debuggable (Tuple s x)) f =
    let Debuggable (Tuple s' r) = f x in
    Debuggable $ Tuple (s <> s') r

newtype Count = Count Int
derive newtype instance semiringCount :: Semiring Count

instance semigroupCount :: Semigroup Count where
  append = (+)

instance monoidCount :: Monoid Count where
  mempty = zero

newtype Countable a = Countable (Tuple Count a)
derive newtype instance functorCountable :: Functor Countable
derive newtype instance applyCountable :: Apply Countable
derive newtype instance applicativeCountable :: Applicative Countable

instance sideEffectCountable :: SideEffect Countable where
  applySideEffect (Countable (Tuple c x)) f =
    let Countable (Tuple c' r) = f x in
    Countable $ Tuple (c + c') r
```

Our `SideEffect` works pretty well. It fits nicely into our Hierarchy of Typeclasses, i.e. `Functor`, `Apply`, and

Applicative.

But this is a Chapter on Monads. And so you might be wondering what does our SideEffect Typeclass have to do with Monads?

18.4. We Created a Monad and More

It turns out that our SideEffect class is nearly the Monad class from PureScript. In PureScript, the Monad has been factored out into 2 Typeclasses, Bind and Monad, whereas in Haskell it's in a single Typeclass.

Here's our SideEffect class for reference:

```
class Applicative f <= SideEffect f where
  applySideEffect :: ∀ a b. f a -> (a -> f b) -> f b
```

And here are Bind and Monad:

```
class Apply m <= Bind m where
  bind :: ∀ a b. m a -> (a -> m b) -> m b ❶

infixl 1 bind as >>= ❷

class (Applicative m, Bind m) <= Monad m ❸
```

❶ bind is equivalent to applySideEffect. This should be obvious when you change the f Type Parameter to m in SideEffect.

❷ The >>= Operator Alias is for bind.

❸ No methods for Monad. Just implied laws that the programmer must verify.

Notice the use of the Polymorphic Parameter, m. That's because we're usually thinking in the context of Monad, as opposed to Functor, when we use these.

One thing to note here is that Bind has Apply as its Superclass. It doesn't need pure to do it's job. But Monad, on the other hand, absolutely needs it for the same reason we created makeDebuggable. It lets us work with Pure Functions. This is why Monad has Applicative as its Superclass, since Applicative implements pure.

Next, let's step back and think about what Monad gives us. It basically allows us to compose Functions of the following form:

```
a -> m b
```

Functions of this type are commonly referred to as ***Monadic Functions*** or ***Effectful Functions***. Basically, Function with Side-effects.

But how does `bind` accomplish this kind of composition, especially since it's more akin to Function Application than Function Composition?

Well, remember when we rewrote `composeDebuggable` in terms of `applyDebuggable`?

```
composeDebuggable
  :: ∀ b c d
  .  (c -> Debuggable d)
  -> (b -> Debuggable c)
  -> (b -> Debuggable d)
composeDebuggable g f x = f x `applyDebuggable` g
```

But `applyDebuggable` is just `bind` specialized for `Debuggable`.

So let's write the general version of `composeDebuggable` that leverages `bind` to allow us to compose Monadic Functions.

We're going to call it `composeKleisli` for reasons that will become apparent once we talk about the ***Kleisli Category***:

```
composeKleisli
  :: ∀ a b c m
  .  Monad m
  => (b -> m c)
  -> (a -> m b)
  -> (a -> m c)
composeKleisli g f x = f x >>= g

infixr 5 composeKleisli as >=> ❶
```

❶ This is the Operator Alias for `composeKleisli` and is commonly referred to as the ***Fish Operator***.

So `bind`, Monadic Function Application, lets us write `composeKleisli` for `Monads`, just like `$`, Pure Function Application, lets us write `compose` for `Functions`:

```
compose
  :: ∀ a b c
  .  (b -> c)
  -> (a -> b)
  -> (a -> c)
compose g f x           = g  $  f x ❶

composeKleisli
  :: ∀ a b c m
  .  Monad m
  => (b -> m c)
  -> (a -> m b)
  -> (a -> m c)
composeKleisli g f x    = g =<< f x ❷
```

❶ This is normally written as `g (f x)`. By using `$`, it's easier to see the similarities between the 2 Functions.

❷ I'm using the flipped `bind` operator to help show how `$` and `bind` work similarly in implementing composition.

The implementations are analogous where `g $ f x` is the analogy of `g =<< f x`.

To sum things up, with `Bind` and `Monad`, we have a way of composing and calling Monadic Functions, i.e. Functions that return Side-effects along with their Pure results. And with `pure` from `Applicative`, we have a way to lift a Pure Function into a Monadic one, so it can operate along side of other Monadic Functions.

At its most basic level, the Monad lets us do Function Application and indirectly Function Composition of Monadic Functions.

That's it. Nothing magical. Nothing complicated. Just very useful and very powerful, as we shall soon see.

18.5. An Alternative `Monad` Implementation

We're going to look at another equally valid way to implement `Monads`. It's more in line with the way Mathematician's see it. And in some languages, `Monad` is implemented in this way. This implementation is Isomorphic to what's done in PureScript and Haskell.

Let's look back at `Bind`:

```
class Apply m <= Bind m where
  bind :: ∀ a b. m a -> (a -> m b) -> m b
```

Since `Apply` is the Superclass of `Bind`, that means that `Bind` is a `Functor`, which means that `Monad` is also a

Functor since Applicative is one of its Superclasses:

```
class (Applicative m, Bind m) <= Monad m
```

Let's pretend, for a moment, that the Bind Typeclass does NOT exist. We'd still like to have a bind Function but not as a Method of a Typeclass.

We know that a Monad is Functor, so let's try to write bind as a helper Function, that leverages map:

```
bind :: ∀ a b m. Functor m => m a -> (a -> m b) -> m b ❶
bind x f = map f x -- COMPILER ERROR!!
```

❶ m must at least be a Functor since we're using map.

Attempting to write bind in terms of map gives us the following compiler error:

```
    Could not match type

      m0 b1

    with type

      b1


  while trying to match type t2 (m0 b1)
    with type m0 b1
  while checking that expression (map f) x
    has type m0 b1
in value declaration bind'
bind' :: ∀ a
bind' x f = map f x
```

The error talks about m0 b1 and b1 not matching. If it's not clear what the problem is, then it's time to write out our Types:

```
map :: (a -> b) -> m a -> m b ❶
f :: a -> m b
map f :: m a -> m (m b) ❷
map f x :: m (m b) ❸
```

❶ Our Functor, usually signified with f, is our Monad, m. So this Type Signature has m in it to make things easier.

❷ Here's where things go wrong. map expects a Function with Type a -> b but we gave it a Function with Type a -> m b.

❸ This final Type doesn't match our bind Type Signature.

Here we wound up with an m b where we only wanted a b. Look back at the error message. That's exactly what the compiler was complaining about.

So now, what do we need to fix this?

We need a Function, which I'll call join, that does the following:

```
join :: ∀ a m. m (m a) -> m a
```

This basically flattens out one of the Monad Contexts. In fact, in other Functional Languages, e.g. Scala, this Function is called flatten. But in PureScript and Haskell it's called join.

So, instead of implementing a Bind Typeclass, we could implement a Typeclass that has join as its Member:

```
class Apply m <= Join m where
  join :: ∀ a. m (m a) -> m a ❶
```

❶ I've dropped m from ∀ in the Type Signature since it can now be found in the Typeclass definition.

Our alternative Monad definition uses Join instead of Bind:

```
class (Applicative m, Join m) <= Monad ❶
```

❶ Join instead of Bind.

And now we can fix our bind Function:

```
bind :: ∀ a b m. Join m => m a -> (a -> m b) -> m b ❶
bind x f = join $ map f x ❷
```

❶ Now it has to at least be a Join since we're using join.

❷ We use join after map has left us with doubly nested m's. It will flatten out one of the m's.

When this approach is taken for Monad, bind is typically called flatMap, which should be apparent from its implementation:

```
flatMap :: ∀ a b m. Monad m => m a -> (a -> m b) -> m b ❶
flatMap x f = flatten $ map f x ❷
```

❶ `bind` has been renamed to `flatMap`.

❷ `join` has been renamed to `flatten`.

I've included this alternative view on `Monad` in case you take what you've learned here to languages that use `flatMap` or if you watch Functional Programming talks in Scala.

18.6. Back to PureScript's `Monad` Implementation

We saw in the previous section a Function `join` that was used to implement an alternative, but Isomorphic, version of `Monad`. And if it's Isomorphic, we should be able to write `join` in terms of things from our `Monad` implementation, i.e. the PureScript one.

Let's look at the Type Signature for `join`:

```
join :: ∀ a m. Bind m => m (m a) -> m a ❶
```

❶ In the PureScript implementation, `m` needs to be at least a `Bind`.

Now, let's write the code:

```
join :: ∀ a m. Bind m => m (m a) -> m a
join x = ???
```

What things from our `Bind` or `Monad` implementation can we use to accomplish `join's` task?

We know that `m` is a `Bind`, which means we COULD use `bind` or `>>=`. To see if `bind` will work for us, let's look to it's definition:

```
class Apply m <= Bind m where
  bind :: ∀ a b. m a -> (a -> m b) -> m b
```

Notice how `bind` MUST break into its first Parameter, `m a`, to get at the `a` before it can call the Function, `a -> m b`.

This will get rid of one layer of Context, i.e. `m`. So far this looks promising since we want `join` to remove a layer of Context.

So, let's try using `bind` to see if it works for us:

```
join :: ∀ a m. Bind m => m (m a) -> m a
join x = x >>= ??? ❶
```

❶ We're taking `x` and binding it to something. We don't know what yet.

Now all we have to do is pick the right Function. It must have the Type Signature `a -> m b`, which seems like it'll just take our newly unwrapped Value, `a`, and rewrap it right back up into a Context giving us back an `m b`.

But that Type Signature is deceiving. Let's write it more accurately to reflect what we're dealing with in `join`. First, the `a` is already in a Context, i.e. it's really `m a`. So, the following is more accurate:

```
??? :: (m a) -> m (m b)
```

And second, `join` doesn't change an `a` to a `b`, so we can unify `a` with `b` and write the following:

```
??? :: (m a) -> m (m a)
```

And finally, we don't want this function to wrap up its input into the Context, so we'll rewrite the Type Signature to reflect that:

```
??? :: (m a) -> (m a)
```

Does that Type Signature look familiar? How about:

```
??? :: a -> a
```

It's just `identity`. So let's replace `???` in `join` with `identity`:

```
join :: ∀ a m. Bind m => m (m a) -> m a
join x = x >>= identity
```

So, in the alternative implementation of `Monad`, we wrote `bind` in terms of `join` and in PureScript and Haskell's implementation, we write `join` in terms of `bind`.

In both worlds, we have the same Functionality just slightly different, but Isomorphic, implementations.

18.7. Recap `Bind`, `Monad` and Supporting Functions

We've covered a lot of ground so far. So it's probably a good time to take inventory.

Here are the classes we've discovered:

```
class Apply m <= Bind m where
  bind :: ∀ a b. m a -> (a -> m b) -> m b

infixl 1 bind as >>=

class (Applicative m, Bind m) <= Monad m
```

And here are the helper Functions:

```
composeKleisli
  :: ∀ a b c m
  .  Monad m
  => (b -> m c)
  -> (a -> m b)
  -> (a -> m c)
composeKleisli g f x = f x >>= g

infixr 5 composeKleisli as >=>

join :: ∀ a m. Bind m => m (m a) -> m a
join x = x >>= identity
```

And here are the flipped version of the operators:

```
composeKleisliFlipped
  :: ∀ a b c m
  .  Monad m
  => (a -> m b)
  -> (b -> m c)
  -> (a -> m c)
composeKleisliFlipped = flip composeKleisli

infixr 1 composeKleisliFlipped as <=< ❶

bindFlipped :: ∀ a b m. Bind m => (a -> m b) -> m a -> m b
bindFlipped = flip bind

infixr 1 bindFlipped as =<< ❷
```

❶ Flipped Fish Operator.

❷ Flipped `bind`.

18.8. Haskell's Monad Implementation

Haskell's implementation is less factored than PureScript's since the PureScript developers had the luxury of stealing from Haskell and improving when they could.

One thing to keep in mind when looking at Haskell's `Monad` class is that `Monads` were in the language BEFORE `Applicatives`. So there are 2 Functions in Haskell that are absent from `Bind` because they are implemented in other classes in PureScript. Once Haskell added `Applicative`, these 2 Functions became redundant.

Another thing we've not seen in Haskell is Default Implementations. PureScript doesn't have these, but in a Typeclass definition in Haskell, you can provide a default implementation for a Method.

Here's Haskell's `Monad` Typeclass definition:

```
class Applicative m => Monad m where ❶
    (>>=)       :: forall a b. m a -> (a -> m b) -> m b ❷

    (>>)        :: forall a b. m a -> m b -> m b ❸ ❹
    m >> k      = m >>= \_ -> k ❺

    return      :: a -> m a ❻ ❼
    return      = pure ❽
```

❶ In Haskell, Typeclasses are constrained the same way Functions are, therefore the fat-arrow is pointing forward.

❷ Same as PureScript's `bind` Function. Remember, in Haskell, you can define operators as Methods without having to create an Operator Alias.

❸ This Function is like `bind` except that the result of the `m a` computation is thrown away, which can be seen in the Lambda by the unnamed Parameter.

❹ This Function is now redundant and is called `*>` in both Haskell and PureScript and can be found in the `Control.Apply` module in PureScript and `Control.Applicative` in Haskell. More on `*>` later.

❺ In Haskell, you can provide Default Implementations in the Typeclass Methods.

❻ This Function has a terrible name. It was called `return` to aid Imperative Programmers when using the `do` notation, which takes `>>=` and makes it APPEAR more Imperative. More on `do` notation later.

❼ This Function was made redundant once `Applicatives` were added to the language. This does exactly what `pure` does.

❽ The Default Implementation is just to call `pure`.

In PureScript, factoring out `Bind` is helpful when something acts like a `Monad`, but breaks one of the Monad Laws. In that case, a developer SHOULD only implement a `Bind` Instance. When a Type has a `Monad` Instance, it's impling that it obeys the Monad Laws.

There is no such flexibility in Haskell.

18.9. Monad Laws

Like everything else, `Monads` have Laws:

```
pure >=> g = g                          [Left Identity]
f >=> pure = f                          [Right Identity]
(f >=> g) >=> h = f >=> (g >=> h)       [Associativity]
```

Keep in mind that `f`, `g` and `h` have Type Signatures of the form `a -> m b`, i.e. they are Monadic Functions.

The Left and Right Identity Laws says that `pure` must act like an `identity` for Kleisli Composition.

The Associativity Law says that Kleisli Composition is Associative just like Pure Function Composition.

Compare the Monad Laws with these for Pure Functions:

```
identity >>> g = g                          [Left Identity]
f >>> identity = f                          [Right Identity]
(f >>> g) >>> h = f >>> (g >>> h)           [Associativity]
```

The similarities should make sense since Kleisli Composition allows us to compose Monadic Functions. So, we'd expect them to behave in the very same manner as Pure Function Composition.

And the Associativity Law is particularly important when we look at the Kleisli Category where the arrows are not `a -> b` like they are in the Hask Category, but are `a -> m b`. For the Kleisli Category to be a legal Category it's morphisms must compose and be Associative.

Let's look at our old friend `Debuggable`. Remember him? It was way back before we know about `Monads`. We made it an instance of `SideEffect`, which we later learned was just `Bind`. It may seem that `Debuggable` is a `Monad` but we've got to check the laws first before we rush to code its instance.

First, we'll convert `Debuggable` to be a `Bind`:

```
newtype Debuggable a = Debuggable (Tuple String a)
derive newtype instance functorDebuggable :: Functor Debuggable
derive newtype instance applyDebuggable :: Apply Debuggable
derive newtype instance applicativeDebuggable :: Applicative Debuggable ❶

instance bindDebuggable :: Bind Debuggable where
  bind (Debuggable (Tuple s x)) f =
    let (Debuggable (Tuple s' r)) = f x in
    Debuggable $ Tuple (s <> s') r ❷
```

❶ We have a law abiding `pure` for `Debuggable` because we delegate to `Tuple's` implementation.

❷ Take note of this step where we combine the first Value of the `Tuple` using the `Semigroup` operator.

We know that `pure` meets the requirements for the Left and Right Identity Laws since we're using the `pure` implementation from `Tuple`, which is a law abiding `Monad`.

In `bind`, we combine the first Value of the Tuple using `<>` from `Semigroup`. Since `String` is a law abiding `Semigroup`, and `<>` must be Associative, we know that the first Value of our underlying `Tuple` is Associative.

The second Value of the underlying `Tuple` is just normal Function Application:

```
instance bindDebuggable :: Bind Debuggable where
  bind (Debuggable (Tuple s x)) f =
    let (Debuggable (Tuple s' r)) = f x in ❶
    Debuggable $ Tuple (s <> s') r ❷
```

❶ Function Application, i.e. `f x`.

❷ `r` is the computational result of `f x`.

We could just wave our hands and argue that our `bind` is also Associative in the second Value of the `Tuple` because Function Application is Associative. But, we've not proven that Function Application is Associative. And even if we did, that doesn't necessarily follow that our `bind` will be.

We're going to have to put our bind through the Associativity Law to make sure it holds. But that law is written using Klesli Composition not `bind`. What we'll need is a version of the Associativity Law written using `bind`.

Since Kleisli Composition is written using `bind`, we should be able to start with the Associativity Law and convert it to a version that uses `bind`.

To do that, we'll start by rewriting Kleisli Composition to remove the 3rd Parameter. This will make the equation easier to use:

```
composeKleisli
  :: ∀ a b c m
  .  Monad m
  => (b -> m c)
  -> (a -> m b)
  -> (a -> m c)
composeKleisli g f = \x -> f x >>= g ❶
```

❶ `x` moved from the right side of the equation to the left as the first argument of the Lambda.

Let's write `composeKleisli` in infixed notation and change the Function names so they don't collide with the Associativity Law to make it easier to work with:

```
composeKleisli s r   = \x -> r x >>= s
s `composeKleisli` r = \x -> r x >>= s
s       >=>        r = \x -> r x >>= s
```

Our strategy here is to substitute the above implementation for `composeKleisli` into the Associativity Laws to get the equivalent law for `bind`:

```
(f >=> g) >=> h = f >=> (g >=> h)          [Associativity]
```

We'll do that by working from left to right:

```
(f >=> g) >=> h                 = f >=> (g >=> h)                  ❶
(\x -> g x >>= f) >=> h         = f >=> (g >=> h)                  ❷
\y -> h y >>= (\x -> g x >>= f) = f >=> (g >=> h)                  ❸
\y -> h y >>= (\x -> g x >>= f) = f >=> (\x -> h x >>= g)          ❹
\y -> h y >>= (\x -> g x >>= f) = \y -> (\x -> h x >>= g) y >>= f  ❺
```

❶ The Associativity Law using Kleisli Composition.

❷ Substitute `f >=> g` into `s >=> r = \x -> r x >>= s`, where `f` binds to `s` and `g` binds to `r`.

❸ Substitute `(\x -> g x >>= f) >=> h` into `s >=> r = \y -> r y >>= s`, where `(\x -> g x >>= f)` binds to `s` and `h` binds to `r`. To avoid name confusion, `x` was renamed to `y` in the Kleisli Composition formula.

❹ Now on the right side, substitute `g >=> h` into `s >=> r = \x -> r x >>= s`, where `g` binds to `s` and `h` binds to `r`.

❺ Substitute `f >=> (\x -> h x >>= g)` into `s >=> r = \y -> r y >>= s`, where `f` binds to `s` and `(\x -> h x >>= g)` binds to `r`. To avoid name confusion, `x` was renamed to `y` in the Kleisli Composition formula.

At this point, we simply reduce the equation at each step:

```
\y -> h y >>= (\x -> g x >>= f) = \y -> (\x -> h x >>= g) y >>= f ❶
 h y >>= (\x -> g x >>= f)       = (\x -> h x >>= g) y >>= f      ❷
 h y >>= (\x -> g x >>= f)       = (h y >>= g) >>= f              ❸
 m >>= (\x -> g x >>= f)         = (m >>= g) >>= f                ❹
```

❶ The last substitution step for reference.

❷ Eliminate `\y ->` on both sides.

❸ Apply `y` to the Lambda on the right side binding `y` to `x`.

❹ Substitute `m` for `h y`. This will be the Monadic Value we start with when doing our `binds`.

The final equation is the Associativity Law for `Monads` but written with `bind` instead of Kleisli Composition:

```
(m >>= g) >>= f = m >>= (\x -> g x >>= f)     [Associativity] ❶
```

❶ Flipped the equation from above.

Now, we can put our implementation for `bind` through this formula to prove it's Associative. But, we only care about the second Value of the `Tuple`. We've already determined that our `Strings`, the first Value of the `Tuple`, will be combined in an Associative way thanks to the Associativity of `<>`.

So, let's write a degenerate version of our `bind` to isolate just the second Value of the `Tuple`. This will help reduce the noise:

```
-- ORIGINAL
instance bindDebuggable :: Bind Debuggable where
  bind (Debuggable (Tuple s x)) f =
    let (Debuggable (Tuple s' r)) = f x
    Debuggable $ Tuple (s <> s') r

-- DEGENERATE VERSION to isolate the second Value of the Tuple
instance bindDebuggable :: Bind Debuggable where
  bind (Debuggable x) f = f x
```

If the degenerate version is Associative then our original version will be too.

Here are the assumptions that we'll be working with to put our degenerate `bind` through the Associativity Law:

```
Debuggable y >>= h = h x ❶
v :: a ❷
w :: b
z :: c
m = Debuggable v :: Debuggable a ❸
g :: a -> Debuggable b ❹
f :: b -> Debuggable c ❺
```

❶ Our degenerate version of `bind` written infixed and with the variables renamed to avoid confusion.

❷ `v`, `w` and `z` are just some working Values specified here with their Types.

❸ We're going to start with `Debuggable v` as our starting Monadic Value for the `bind`.

❹ `g` is a Monadic Function of the form `a -> m b`, where the Type `m` is the Monad, `Debuggable`, not to be confused with our Monadic Value, `m`.

❺ `f` is also a Monadic Function of the form `b -> m c`.

Now we start with the left side of the law:

```
(m >>= g) >>= f                    ❶
(Debuggable v >>= g) >>= f         ❷
g v >>= f                          ❸
Debuggable w >>= f                 ❹
f w                                ❺
Debuggable z                       ❻
```

❶ Left side of Associativity Law.

❷ Substitute using `m = Debuggable v`.

❸ Substitute `Debuggable v >>= g` into `Debuggable y >>= h = h y`, where `v` binds to `y` and `g` binds to `h`.

❹ Apply `g` to `v`. Remember that `w` is of Type `b`.

❺ Substitute `Debuggable w >>= f` into `Debuggable y >>= h = h y`, where `w` binds to `y` and `f` binds to `h`.

❻ Apply `f` to `w`. Remember that `z` is of Type `c`.

And we do the same with the right side of the law:

```
m >>= (\x -> g x >>= f)              ❶
Debuggable v >>= (\x -> g x >>= f)   ❷
(\x -> g x >>= f) v                  ❸
g v >>= f                            ❹
Debuggable w >>= f                   ❺
f w                                  ❻
Debuggable z                         ❼
```

❶ Right side of Associativity Law.

❷ Substitute using `m = Debuggable v`.

❸ Substitute `Debuggable v >>= (\x -> g x >>= f)` into `Debuggable y >>= h = h y`, where `v` binds to `y` and `(\x -> g x >>= f)` binds to `h`.

❹ Apply `(\x -> g x >>= f)` to `v`, where `v` binds to `x`.

❺ Apply `g` to `v`. Remember that `w` is of Type `b`.

❻ Substitute `Debuggable w >>= f` into `Debuggable y >>= h = h y`, where `w` binds to `y` and `f` binds to `h`.

❼ Apply `f` to `w`. Remember that `z` is of Type `c`.

Since substituting our `bind` implementation into both sides of the Associativity Laws resulted in `Debuggable z`, we know that our degenerate implementation of `bind` is Associative, which means our original version of `bind` is also Associative.

18.10. Monad Instance for Maybe

Now that we've discovered the Monad, let's use it. And, as always, `Maybe` is a great place to start.

For reference, here's the `Maybe` implementation up to and including `Applicative`:

```
data Maybe a = Nothing | Just a

instance functorMaybe :: Functor Maybe where
  map _ Nothing  = Nothing
  map f (Just x) = Just $ f x

instance applyMaybe :: Apply Maybe where
  apply Nothing  _ = Nothing
  apply (Just f) x = f <$> x

instance applicativeMaybe :: Applicative Maybe where
  pure = Just
```

Also, for reference, here's the Bind and Monad Typeclass definitions:

```
class Apply m <= Bind m where
  bind :: ∀ a b. m a -> (a -> m b) -> m b

infixl 1 bind as >>=

class (Applicative m, Bind m) <= Monad m
```

And now we can write the Monad Instance for Maybe. But first we have to write the Bind Instance. We'll do that one line at a time:

```
instance bindMaybe :: Maybe where
```

Nothing out of the ordinary in this line of code. We've done this dozens of times now.

And just to help us code, we'll add the specialized Type Signature for Maybe. As always, we can delete this when we're done:

```
instance bindMaybe :: Maybe where
  bind :: ∀ a b. Maybe a -> (a -> Maybe b) -> Maybe b
  ???
```

Next, we'll add the Nothing case:

```
instance bindMaybe :: Maybe where
  bind :: ∀ a b. Maybe a -> (a -> Maybe b) -> Maybe b
  bind Nothing _ = Nothing
  ???
```

Since there is no Value in the Nothing, we cannot call the Monadic Function. So, once again the Data Structure dictates that we must Short-circuit. This is great because Maybe Short-circuits with Nothing just about everywhere else and this instance doesn't break the Semantics of Maybe.

And now the Just case:

```
instance bindMaybe :: Bind Maybe where
  bind :: ∀ a b. Maybe a -> (a -> Maybe b) -> Maybe b
  bind Nothing _ = Nothing
  bind (Just x) f = f x ❶
```

❶ Don't forget that `f` returns a `Maybe b`. That's why this code looks so simple.

That may seem too easy. In fact, it is that easy. And, as it turns out, it was easier to write this than it was to write the `Applicative` Instance. This is a common pattern. And once we learn about `ap`, writing `Applicative` Instances will get easy too.

And now for the easiest part, coding the `Monad` Instance:

```
instance monadMaybe :: Monad Maybe
```

While that was super simple, it comes at a cost. By creating the `Monad` Instance, we're promising users of our Type that we've giving them a law abiding `Monad` Instance, which means we need to do the work of proving that we have.

First, we start with the Monad Laws:

```
pure >=> g = g                                [Left Identity (Kleisli)]
f >=> pure = f                                [Right Identity (Kleisli)]
(f >=> g) >=> h = f >=> (g >=> h)             [Associativity (Kleisli)]
(m >>= g) >>= f = m >>= (\x -> g x >>= f)     [Associativity (Bind)]
```

I've include both the Kleisli Composition version along with the Bind version of the Associativity Law, where `>=>` is Monadic Function Composition and `>>=` is Monadic Function Application.

Remember, a Monadic Function (or Effectful Function) has the Type Signature of `a -> m b`, where a Pure Function has `a -> b`.

To prove we have a proper `Monad`, we'll want to use the `Bind` versions of the laws, since our code is for `bind`. But we'll first need to convert the Left and Right Identity Laws from Kleisli Composition to `bind`.

To do this, we'll need the implementation of `composeKleisli`, which is written using `bind`, i.e. `>>=`:

```
composeKleisli
  :: ∀ a b c m
  .  Monad m
  => (b -> m c)
  -> (a -> m b)
  -> (a -> m c)
composeKleisli g f = \x -> f x >>= g ❶
```

❶ Written using a Lambda to keep it easier to convert our laws to use `bind`.

Once again, we'll rewrite `composeKleisli` implementation using infixed notation and with renamed

variables to avoid name collision with the variables in the laws:

```
s >=> r = \x -> r x >>= s ❶
```

❶ Replaced `f` with `r` and `g` with `s`.

First, we'll convert the Left Identity Law:

```
pure >=> g            = g   ❶
\x -> g x >>= pure    = g   ❷
g x >>= pure          = g x ❸
```

❶ The Kleisli version of the Left Identity Law.

❷ Substitute `pure >=> g` into `s >=> r = \x -> r x >>= s`, where `pure` binds to `s` and `g` binds to `r`.

❸ Moved the `x` Parameter to the other side of the equal sign.

Next, we'll convert the Right Identity Law:

```
f >=> pure            = f   ❶
\x -> pure x >>= f    = f   ❷
pure x >>= f          = f x ❸
```

❶ The Kleisli version of the Right Identity Law.

❷ Substitute `f >=> pure` into `s >=> r = \x -> r x >>= s`, where `f` binds to `s` and `pure` binds to `r`.

❸ Moved the `x` Parameter to the other side of the equal sign.

And we now have all of the Monad Laws written using `bind`:

```
g x >>= pure = g x                          [Left Identity (Bind)]
pure x >>= f = f x                          [Right Identity (Bind)]
(m >>= g) >>= f = m >>= (\x -> g x >>= f)   [Associativity (Bind)]
```

Here are our `Applicative` and `Bind` Instance implementations for `Maybe` (we'll need both):

```
instance applicativeMaybe :: Applicative Maybe where
  pure = Just

instance bindMaybe :: Bind Maybe where
  bind Nothing _ = Nothing
  bind (Just x) f = f x
```

Now, we're ready to verify the laws. As always, we'll do Algebraic Substitution and Function Application until both sides of the Law's equation are identical.

Let's start with Left Identity where `g x = Nothing`:

```
g x >>= pure     = g x       ❶
Nothing >>= pure = Nothing   ❷
Nothing          = Nothing   ❸
```

❶ The Bind version of the Left Identity Law.

❷ Substitute `g x = Nothing` on both sides of the equal sign.

❸ Apply `Nothing` case from `bindMaybe`.

And now the Left Identity when `g x = Just y`:

```
g x >>= pure     = g x     ❶
Just y >>= pure  = Just y  ❷
pure y           = Just y  ❸
Just y           = Just y  ❹
```

❶ The Bind version of the Left Identity Law.

❷ Substitute `g x = Just y` on both sides of the equal sign.

❸ Apply `Just` case from `bindMaybe`.

❹ Substitute `pure` for `Just` from `applicativeMaybe`.

And now the Right Identity proof:

```
pure x >>= f = f x   ❶
Just x >>= f = f x   ❷
f x          = f x   ❸
```

❶ The Bind version of the Right Identity Law.

❷ Substitute `pure` for `Just` from `applicativeMaybe`.

❸ Apply `Just` case from `bindMaybe`.

Next, the Associativity Law for `m = Nothing`.

```
(m >>= g) >>= f       = m >>= (\x -> g x >>= f)          ❶
(Nothing >>= g) >>= f = Nothing >>= (\x -> g x >>= f)    ❷
Nothing >>= f         = Nothing >>= (\x -> g x >>= f)    ❸
Nothing               = Nothing >>= (\x -> g x >>= f)    ❹
Nothing               = Nothing                          ❺
```

❶ The Bind version of the Associativity Law.

❷ Substitute `m` for `Nothing` on both sides of the equal sign.

❸ Apply `Nothing` case from `bindMaybe` on left-hand side.

❹ Apply `Nothing` case from `bindMaybe` on left-hand side again.

❺ Apply `Nothing` case from `bindMaybe` on right-hand side.

And finally, the Associativity Law where `m = Just y`:

```
(m >>= g) >>= f       = m >>= (\x -> g x >>= f)          ❶
(Just y >>= g) >>= f  = Just y >>= (\x -> g x >>= f)     ❷
g y >>= f             = Just y >>= (\x -> g x >>= f)     ❸
g y >>= f             = (\x -> g x >>= f) y              ❹
g y >>= f             = g y >>= f                        ❺
```

❶ The Bind version of the Associativity Law.

❷ Substitute `m` for `Just y` on both sides of the equal sign.

❸ Apply `Just` case from `bindMaybe` on left-hand side.

❹ Apply `Just` case from `bindMaybe` on right-hand side.

❺ Apply `y` to Lambda, `\x -> g x >>= f`, where `y` binds to `x`.

18.11. Working with `Maybe` Monad

Now that we have the `Maybe` Monad, we should take advantage of it. Let's use it to put a number through a set of tests any of which could fail.

Let's write some tests:

```
import Data.Int.Bits ((.&.))

oddTest :: Int -> Maybe Int
oddTest x = if x .&. 1 == 1 then Just x else Nothing

greaterThanTest :: Int -> Int -> Maybe Int
greaterThanTest min x = if x > min then Just x else Nothing

lessThanTest :: Int -> Int -> Maybe Int
lessThanTest max x = if x < max then Just x else Nothing
```

Next, we'll write a Function that puts our number through a gauntlet of tests. First, we'll write it using Kleisli Composition:

```
gauntlet :: Int -> Maybe Int
gauntlet = oddTest >=> greaterThanTest 10 >=> lessThanTest 20
```

Now we can run some numbers through:

```
log $ show $ gauntlet 14    -- Nothing -- fails oddTest
log $ show $ gauntlet 1     -- Nothing -- fails greaterThanTest
log $ show $ gauntlet 93    -- Nothing -- fails lessThanTest
log $ show $ gauntlet 17    -- (Just 17)
```

Let's modify `guantlet` to do some computations along the way:

```
gauntlet :: Int -> Maybe Int
gauntlet =
  oddTest >=> pure <<< (_ + 1) >=> greaterThanTest 10 >=> lessThanTest 20
```

Here we first test for `odd` then add 1 before doing more tests. Notice how we compose `pure` with the Pure Function, `(_ + 1)`, to turn it into a Monadic Function. This is so we we can compose it using Kleisli Composition.

Let's rewrite this new version of `gauntlet` to use `bind`:

```
gauntlet :: Int -> Maybe Int
gauntlet x =
  pure x >>= oddTest ❶
    >>= \o -> pure (o + 1) ❷ ❸
      >>= \y -> greaterThanTest 10 y
        >>= \z -> lessThanTest 20 z
```

❶ We have to put our initial Pure Value, `x`, into a Context, in this case, a `Maybe`, before we can use `>>=`. We do that by using `pure`.

❷ Do NOT use `pure $ o + 1`. Notice how each line is indented. That's because this line is continued on multiple lines. If we use the `$` then it'll effectively put Parentheses starting at the `$` and ending at the end of the Function. This is NOT what we want.

❸ We use `pure` to put the Pure Value, `(o + 1)`, into a Context, in this case, a `Maybe`.

Referring to our latest version of `gauntlet`, what happens when we try to pass the number `1` through? Here are the steps:

1. It's placed into the proper Context, with `pure x`.
2. It passes the `oddTest`.
3. One is added to it making it `2` and then it's put into the proper context using `pure`
4. It FAILS the `greaterThanTest` since it's NOT greater than `10`.

Notice how the `lessThanTest` is never evaluated. Why?

Because of the implementation of the `Bind` Instance for `Maybe`:

```
instance bindMaybe :: Bind Maybe where
  bind :: ∀ a b. Maybe a -> (a -> Maybe b) -> Maybe b
  bind Nothing _ = Nothing ❶
  bind (Just x) f = f x
```

❶ Here's where the `lessThanTest` is ignored. See the `_`. That's the second Parameter to `>>=`, which is ignored if the first Parameter is `Nothing`. In the above example, this happens when `greaterThanTest` FAILS.

Looking back at our `gauntlet` Function:

```
gauntlet :: Int -> Maybe Int
gauntlet x =
  pure x >>= oddTest
    >>= \o -> pure (o + 1)
      >>= \y -> greaterThanTest 10 y
        >>= \z -> lessThanTest 20 z
```

Notice that we didn't write one bit of code to check for errors. We wrote the code as if there were no errors. This is commonly called the ***Happy Path***.

It turns out that `Maybe's bind` does all of the hard work of checking errors for us. The Pattern Matching that's done on the first Parameter of the `bind` implementation, i.e. the right-hand side of `>>=`, is where all of the if-logic resides. The compiler generates this `if` code for us. When we match the `Nothing` case, we Short-circuit the computation and return `Nothing`, which means we had an error.

So we can just write code as if errors never happen, since `bind` is doing the error checking for us. This is a huge win. We don't have to clutter our code with the usual litany of error checks. Instead, they're magically hidden in the `>>=` Operator, i.e. in the `bind` implementation for `Maybe`.

In most languages, we're forced to write massive amounts of `if-then-else` statements. So much so, that it clutters our code making it nearly impossible to understand what our Functions are supposed to do since the majority of the code is error checking.

Now the only real issue is the indenting. Imagine what would happen if we had a lot of tests:

```
lotsOfTests :: Int -> Maybe Int
lotsOfTests a =
  pure a >>= oddTest
    >>= \o -> pure (o + 1)
      >>= \b -> testB b ❶
        >>= \c -> testC c
          >>= \d -> testD d
            >>= \e -> testE e
              >>= \f -> testF f
                >>= \g -> testG g
                  >>= \h -> testH h
                    >>= \i -> testI i
```

❶ This line of code COULD be reduced to `>>= testB`, but I'm writing the code like this here and on every subsequent line to help us compare this with a later example.

This is what I like to call ***Indentation Hell***. Well, both PureScript and Haskell have a very nice feature to help us wrangle these indentations. It's called ***Do Notation***.

18.12. Do Notation

Do Notation is syntactical sugar for `binds`. It removes all of the indentations.

We've seen this notation in each of our `test` Functions that we've written.

Let's look at some simple examples of rewriting `bind` using `do`:

```
gauntlet' :: Int -> Maybe Int
gauntlet' x =
  pure x >>= oddTest
    >>= \o -> pure (o + 1) ❷
      >>= \y -> greaterThanTest 10 y ❸
        >>= \z -> lessThanTest 20 z

gauntlet :: Int -> Maybe Int
gauntlet x = do ❶
  o <- oddTest x ❷
  y <- pure (o + 1) ❸
  z <- greaterThanTest 10 y
  r <- lessThanTest 20 z
  pure r ❹
```

❶ We use the `do` keyword to start a block of this special syntax.

❷ `o` is the EXTRACTED result from `pure $ oddTest x` assuming it does NOT fail. This is what the Backward Arrow, `<-`, signifies. In the `bind` version above, `o` is also the EXTRACTED result but just written as a `Lambda`.

❸ Notice how `o` is in scope, i.e. it's a Value that can be seen by this line of code, just like it was in our `bind` version. Remember, this is just syntactical sugar.

❹ We return the final value.

Now we can make a few improvements:

```
gauntlet :: Int -> Maybe Int
gauntlet x = do
  o <- oddTest x
  let y = o + 1 ❶ ❷
  z <- greaterThanTest 10 y
  lessThanTest 20 z ❸
```

❶ We can use `let` here eliminating the need for `pure`.

❷ Notice there is NO `in` keyword. You do NOT use `in` when you're using `let` in a `do` block. This can be a bit

of pain to get used to but will happen over time. I'm making a big deal about it here to hopefully help accelerate that process. You will make this mistake. Be patient with yourself. It'll become second nature. So much so, that you'll start leaving it off outside of `do` blocks.

❸ We don't need to extract `r` and then rewrap it with `pure`. Those steps are redundant. We simply return the result of `lessThanTest`, which is already a `Maybe Int`. Notice how I used the word `return`. This is the kind of thinking that was in play when Haskell decided to call their `Monad` Method `return`. Remember that Haskell's `return` is the equivalent of `pure`.

Let's look at how the Do Notation makes our `lotsOfTests` Function far more readable:

```
lotsOfTests' :: Int -> Maybe Int
lotsOfTests' a =
  pure a >>= oddTest
    >>= \o -> pure (o + 1)
      >>= \b -> testB b
        >>= \c -> testC c
          >>= \d -> testD d
            >>= \e -> testE e
              >>= \f -> testF f
                >>= \g -> testG g
                  >>= \h -> testH h
                    >>= \i -> testI i

lotsOfTests :: Int -> Maybe Int
lotsOfTests a = do
  o <- oddTest a
  let b = o + 1
  c <- testB b
  d <- testC c
  e <- testD d
  f <- testE e
  g <- testF f
  h <- testG g
  i <- testH h
  testI i
```

To reiterate, once a Value is extracted using the backward arrow, `<-`, it remains in scope for the remainder of the `do` block. The scoping of Values is very intuitive this way.

There are a few other things you can do inside of a `do` block, e.g. you can have `if` expressions:

```
gauntlet :: Int -> Maybe Int
gauntlet x = do
  o <- oddTest x
  let y = o + 1
  if o .&. 1 == 0 then do ❶ ❷
    z <- greaterThanTest 10 y
    lessThanTest 20 z
  else
    Nothing
```

❶ I realize that this test is redundant. It's only here to demonstrate how `if` expressions work in a `do` block.

❷ Notice how we start a new `do` block under the `then` section. That's because we have multiple steps to complete in the section. Not so under the `else` section where we handle the error case.

You may have noticed that I've been using different variables to hold the result of our tests. But if you remember our tests all return the original value in a `Maybe` UNCHANGED.

This means I can write the following code:

```
gauntlet :: Int -> Maybe Int
gauntlet x = do
  o <- oddTest x
  let y = o + 1
  greaterThanTest 10 y -- COMPILER ERROR!!
  lessThanTest 20 y
```

But we get a compiler error:

```
    A result of type

      Int

    was implicitly discarded in a do notation block.
    You can use _ <- ... to explicitly discard the result.

  while applying a function discard
    of type Discard t0 => Bind t1 => t1 t0 -> (t0 -> t1 t2) -> t1 t2
    to argument (greaterThanTest 10) o
ga while inferring the type of discard ((greaterThanTest 10) o)
ga in value declaration gauntlet

  where t0 is an unknown type
  greaterThanTest 10 o
  lessThanTest 20 o
```

This is because we are discarding the result of a computation in a `do` block. You might remember in early Chapters we added the Function `discard` to our `Prelude` imports. The `discard` Function is automatically

used by the compiler when the Value that you're discarding has a `Discard` Instance.

You create a `Discard` Instance if and only if your Type can be safely discarded in a `do` block. `Unit` is one such Type. That's why the compiler uses it with our test code:

```
test :: Effect Unit
test = do
  log "placeholder" ❶
```

❶ `log` returns an `Effect Unit`, which means that if we were to extract the Value, we'd get `unit`. The compiler doesn't complain here since it can be safely discarded.

In the code that's causing the compiler error, we're discarding an `Int`, which does NOT have a `Discard` Instance. So we must manually discard it. We'll do as the error message suggests:

```
gauntlet :: Int -> Maybe Int
gauntlet x = do
  o <- oddTest x
  let y = o + 1
  _ <- greaterThanTest 10 y ❶
  lessThanTest 20 y
```

❶ The extracted value is discarded since the Variable isn't named, i.e. is `_`.

And now that fixes our compiler error, but, to be honest, I think this syntax looks a bit ugly and it's also hard to type. I may be nitpicking here, but there's another solution that I favor for those minor reasons.

It's using a Function called `void`:

```
gauntlet :: Int -> Maybe Int
gauntlet x = do
  o <- oddTest x
  let y = o + 1
  void $ greaterThanTest 10 y ❶
  lessThanTest 20 y
```

❶ I think this is better than extracting the Value into a unnamed Variable.

But how does `void` work it's magic?

Well, without looking at `void's` Type Signature, we could imagine that `void` didn't exist and we want to write it.

We'd need to take any `m a`, i.e. any Monadic Value and turn it into something that the compiler will discard

on our behalf. We know that `Unit` is such a Type, which means that we'll need a Function of the following Type Signature:

```
ourVoid :: ∀ a. Monad m => m a -> m Unit
```

In our example, this will turn our `Maybe Int` into a `Maybe Unit`, which can be safely discarded.

But looking at `ourVoid`, we might want to ask ourselves if we need the `Monad` constraint. It would work fine if we limited it to `Bind`. But can we loosen our requirements any further?

What about going all the way down to `Functor`:

```
ourVoid :: ∀ a. Functor f => f a -> f Unit
```

Is there any downside to this more relaxed Constraint?

I don't see any and evidently neither has anyone else since this is the same Type Signature as `void`.

By the way, we don't use `void` when using `bind`. Instead, we use `*>` :

```
gauntlet :: Int -> Maybe Int
gauntlet x =
  oddTest x
    >>= \o -> pure (o + 1)
      >>= \y -> greaterThanTest 10 y
        *> lessThanTest 20 y ❶ ❷
```

❶ Here we disregard the result of the left-hand side by using `*>`.

❷ In Haskell, you'll see the `>>` operator used more often than `*>`. That's because `>>` is a `Monad` Method and `*>` is an `Applicative` helper Function. But since `Applicatives` came later than `Monads` in Haskell, the use of `>>` became idiomatic. In PureScript, only `*>` exists and it's an `Apply` helper Function called `applySecond`.

To understand how `*>` works, viz. how it disposes of the first result, let's take a look at the implementation of its underlying Function, `applySecond`:

```
applySecond :: ∀ a b f. Apply f => f a -> f b -> f b
applySecond a b = const identity <$> a <*> b
```

Mapping `const identity` is what discards whatever Value happens to be inside of the Context `f` and replaces that Value, i.e. `a`, with the `identity` Function. Now that we have a Function in a Context, we can safely apply it to `b`. Applying `identity` to the Value in `b' will NOT change the Value. And so we wind up

with just the result of the second Parameter.

We do, however, get the Effect of a, which, in our case, is Short-circuiting:

```
\y -> greaterThanTest 10 y *> lessThanTest 20 y
```

If greaterThanTest fails, then the computation fails. That's the Effect of Maybe. But if it doesn't fail, it's return Value is discarded in favor of lessThanTest's.

Notice that *> is written NOT in terms of bind, >>=, but in terms of map, <$>, and apply, <*>. The Short-circuiting behavior is the same in both map and apply as it is in bind:

```
instance applyMaybe :: Apply Maybe where
  apply Nothing  _ = Nothing ❶
  apply (Just f) x = f <$> x

instance bindMaybe :: Bind Maybe where
  bind :: ∀ a b. Maybe a -> (a -> Maybe b) -> Maybe b
  bind Nothing _ = Nothing ❷
  bind (Just x) f = f x
```

❶ Short-circuit logic.

❷ Short-circuit logic AGAIN.

It's unfortunate that we had to write that logic twice.

If there were only a way to do it once.

18.13. Cheating with the Applicative Instance and ap

Let's compare our apply and bind implementations for Maybe:

```
instance applyMaybe :: Apply Maybe where
  apply Nothing  _ = Nothing
  apply (Just f) x = f <$> x

instance bindMaybe :: Bind Maybe where
  bind :: ∀ a b. Maybe a -> (a -> Maybe b) -> Maybe b
  bind Nothing _ = Nothing
  bind (Just x) f = f x
```

There isn't much difference here. When I see this, it makes me think that we could write one in terms of the

other. And it turns out that we can.

Let's write apply in terms of bind since apply is typically the most difficult to implement when compared to Functor and Bind.

First, we start with the instance definition:

```
instance applyMaybe :: Apply Maybe where
```

And now the right side of the equal sign:

```
instance applyMaybe :: Apply Maybe where
  apply mf mx = ??? ❶ ❷
```

❶ mf is the Function, f, in the Context, m, hence the name mf.

❷ mx is the Value, x, in the Context, m, hence the name mx.

At this point, our strategy should be to first extract the Function, f, from the Context, then call it with mx using bind.

Remember, when we first talked about apply, we decided that since its operator is Left-Associative, we should extract the Function first.

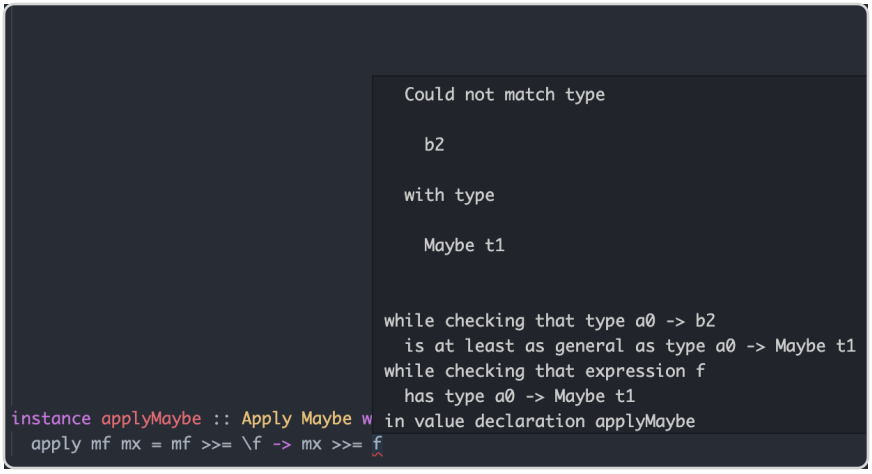
First, we'll extract the Function using >>=:

```
instance applyMaybe :: Apply Maybe where
  apply mf mx = mf >>= \f -> ???
```

Now we can call the Function with mx using bind again:

```
instance applyMaybe :: Apply Maybe where
  apply mf mx = mf >>= \f -> mx >>= f     -- COMPILER ERROR!!
```

Unfortunately, we have a problem. Here's the compiler error message:



It found a `b2` but wants a `Maybe t1`. Not super helpful other than telling us that the Types don't match. The red underline is a clue that the compiler is referring to the Function, `f`.

To figure out what's wrong, let's add in the Type Signature for `apply` specialized for `Maybe`:

```
instance applyMaybe :: Apply Maybe where
  apply :: ∀ a b. Maybe (a -> b) -> Maybe a -> Maybe b
  apply mf mx = mf >>= \f -> mx >>= f     -- COMPILER ERROR!!
```

This makes it easier for us to write down all of the Types of our variables paying particular attention to `f`:

```
mf :: Maybe (a -> b)
f :: a -> b
mx :: Maybe a
```

Can you see what's wrong now?

Maybe if we write the Type Signature for `bind`, `>>=`, specialized for `Maybe` it will become apparent:

```
bind :: ∀ a b. Maybe a -> (a -> Maybe b) -> Maybe b
```

Look at the Types of our variables and `mx >>= f` to see if you can spot the problem now. Try to figure out the issue before reading on.

We know that that `f` is the problem since the red underline was under the `f`. `f` is the second Parameter to `bind`, which means its Type Signature is supposed to be `a -> Maybe b` but it's not. Instead `f's` Type Signature is `a -> b`.

We can fix this by composing it with `pure` to wrap the `b` that `f` returns in a `Maybe`:

```
instance applyMaybe :: Apply Maybe where
  apply mf mx = mf >>= \f -> mx >>= pure <<< f
```

Okay, so this was a bit of work and not intuitive at all. It's also hard to read with all of the operators.

So let's rewrite it using do notation:

```
instance applyMaybe :: Apply Maybe where
  apply mf mx = do
    f <- mf ❶
    x <- mx ❷
    pure $ f x ❸
```

❶ First, extract the Function, f.

❷ Then extract the Value, x.

❸ Apply f to x and wrap it back up into the Context via pure.

This is so much easier to read and, therefore, easier to reason about.

Notice how this code is independent of the fact that we're working with Maybe. That's because we wrote it using pure instead of Just. That means we can make a helper Function. Let's call it ap:

```
ap :: ∀ a b m. Monad m => m (a -> b) -> m a -> m b ❶
ap mf mx = do
    f <- mf
    x <- mx
    pure $ f x
```

❶ We have to constrain the Context, m, to be a Monad since we use >>= from Bind and pure from Applicative. Remember, Monad is a Bind and an Applicative.

Now we can rewrite apply for Maybe using ap:

```
instance applyMaybe :: Apply Maybe where
  apply = ap
```

Wow, that's much simpler. By the way, the ap Function we wrote already exists in the Control.Monad module. Turns out that ap is considered so useful, that it's imported by Prelude.

We can use ap like this on any Type that is going be a Monad, but realize when you first code it up, the compiler may complain because you haven't written the Bind or Monad Instances yet.

18.14. Monad Instance for Either

Before we write the Monad Instance for Either, let's look at what we've written for Either so far:

```
data Either a b = Left a | Right b

instance functorEither :: Functor (Either a) where
  map _ (Left err) = Left err
  map f (Right x)  = Right $ f x

instance applyEither :: Apply (Either a) where
  apply (Left y)  _ = Left y
  apply (Right f) x = f <$> x

instance applicativeEither :: Applicative (Either a) where
  pure = Right
```

This implementation is very similar to Maybe, so writing the Bind Instance for Either should be equally simple.

First, the instance definition:

```
instance bindEither :: Bind (Either a) where ❶
```

❶ Holding a constant.

Next, the error case:

```
instance bindEither :: Bind (Either a) where
  bind (Left y) _ = Left y
  ???
```

And finally, the success case:

```
instance bindEither :: Bind (Either a) where
  bind (Left y) _ = Left y
  bind (Right x) f  = f x ❶
```

❶ Remember, f will return an Either so we don't have to worry about wrapping it back up.

And writing the Monad Instance is trivial:

```
instance monadEither :: Monad (Either a)
```

We technically should verify the Monad Laws, but since it's implementation is analogous to `Maybe` and `Either a` is Isomorphic to `Maybe`, we'll just assume it doesn't break any laws. But, if we were so inclined to prove the laws, we could simply substitute `Nothing` with `Left y` and `Just` with `Right` in our proof we did for `Maybe`.

One final thing we can do here is compare `apply` to `bind`:

```
instance applyEither :: Apply (Either a) where
  apply (Left y)  _ = Left y
  apply (Right f) x = f <$> x

instance bindEither :: Bind (Either a) where
  bind (Left y) _ = Left y
  bind (Right x) f  = f x
```

Once again, we can see that these implementations are more alike than not. They both Short-circuit when `Left` is encountered. We could write `apply` in terms of `bind` by using `ap`:

```
instance applyEither :: Apply (Either a) where
  apply = ap
```

You may be wondering if doing this always works. Well, let's think about it for a second.

Any `apply` implementation SHOULD have the same Semantics, i.e. behavior, as the `bind` implementation. If it does not, it will most likely break the `Apply` or `Monad` Laws. But even if it doesn't, it would be a terribly designed Type if the `Apply` had different Semantics from `Bind`.

In the case of `Maybe` and `Either`, they Short-circuit. And both the `Apply` and `Bind` Instances should exhibit this behavior. With `Maybe`, we have no choice since Short-circuiting is intrinsic to the Data Structure. But with `Either`, we made an explicit decision to Short-circuit, therefore, we'd want our `apply` and `bind` implementations to both Short-circuit.

So, by all reasonable measures, `ap` SHOULD be able to do the work of `Apply` in all cases. While it is possible to create a Type that has inconsistent Semantics and, therefore, `ap` could not be used to produce this nefarious Type, I would advise strongly against ever doing so.

When we return to our `Parser`, we'll see that its `Apply` Instance could easily be replaced with `ap`, that is after we write the `Monad` Instance.

18.15. Working with Either Monad

We can do computations within the Either Monad in nearly identical ways to the Maybe Monad with the only discernible difference being that Left gives us more information than Nothing.

Let's rewrite our tests from the Maybe Section:

```
import Data.Int.Bits ((.&.))

oddTest :: Int -> Either String Int
oddTest x = if x .&. 1 == 1 then Right x else Left "Number is not odd"

greaterThanTest :: Int -> Int -> Either String Int
greaterThanTest min x =
  if x > min then Right x
  else Left $ "Number is not greater than " <> show min

lessThanTest :: Int -> Int -> Either String Int
lessThanTest max x =
  if x < max then Right x
  else Left $ "Number is not less than " <> show max
```

Now we can rewrite gauntlet:

```
gauntlet :: Int -> Either String Int ❶
gauntlet x = do
  o <- oddTest x
  let y = o + 1
  void $ greaterThanTest 10 y
  lessThanTest 20 y
```

❶ Maybe was replaced with Either String.

What's interesting is that we only had to change the Type, i.e. once we changed our test Functions. That's because gauntlet was written using bind, which is a TypeClass Method.

The gauntlet example is pretty contrived. So, let's use Either to do an example that's a bit more realistic.

Let's revisit our fullName Function from an earlier Chapter:

```
fullNameEither :: Maybe String -> Maybe String -> Maybe String
                  -> Either String String
fullNameEither first middle last =
  fullName <$> (first `errIfMissing` "First name must exist")
           <*> (middle `errIfMissing` "Middle name must exist")
           <*> (last `errIfMissing` "Last name must exist")
```

Let's rewrite this using Monadic computation with do notation:

```
fullNameEither :: Maybe String -> Maybe String -> Maybe String
                  -> Either String String
fullNameEither first middle last = do
  f <- errIfMissing first "First name must exist"
  m <- errIfMissing middle "Middle name must exist"
  l <- errIfMissing last "Last name must exist"
  pure $ fullName f m l
```

Personally, I like the Applicative version better, but when the computation is more complex, I'll reach for the Monadic version. It really depends.

The Monadic version still suffers from the same problem as our Applicative one does and that is we only get the first error. We have an Applicative called Validation that we used to collect all of the errors.

I wonder if we can make that a Monad.

18.16. Adding Monad Instance for Validation

Let's start with what we've written so far for Validation:

```
newtype Validation err result = Validation (Either err result)
derive newtype instance functorValidation :: Functor (Validation err)

instance applyValidation :: Semigroup err => Apply (Validation err) where
  apply (Validation (Left err1)) (Validation (Left err2))
                                  = Validation $ Left (err1 <> err2)
  apply (Validation (Left err)) _ = Validation $ Left err
  apply (Validation (Right f)) x  = f <$> x
```

Now let's write the Bind Instance one line at a time:

```
instance bindValidation :: Semigroup err => Bind (Validation err) where
```

We use the same Constraint on `err` to be a `Semigroup` since `Bind` is an `Apply` and the `Apply` Instance has that Constraint.

We also are expecting to combine errors in the `bind` implementation, so we're going to need this anyway.

So now let's write the error case:

```
instance bindValidation :: Semigroup err => Bind (Validation err) where
  bind (Validation (Left err)) f = ???
```

And now we have a strange problem. We only have a single error at a time with the `bind` operation. There is no way for us to combine it with any other errors. Calling `f` has the potential of giving us a second error, but we have no Value to apply it to since we are in the case where the previous computation failed.

When we look at the `Apply` implementation, we can see that we have 2 things in Context, but with the `Monad` we only have 1.

Because of this important difference, we cannot make a `Bind` Instance for `Validation` and, therefore, we cannot make a `Monad` Instance.

We've encountered our first Type that is NOT a `Monad`. It's only an `Applicative`. There are plenty of Types that cannot be made into `Monads` and it's very useful for us to learn to recognize when we can or cannot do so.

The only way we can use `Validation` then is via `<$>` and `<*>`.

At this point, you may be wondering how one would go about collecting up errors in a `Monad`. Well, that's easy, i.e. once you've learned about the `State` Monad. But before we do, there are few `Monads` worth learning along the way that will make learning the `State` Monad much easier.

18.17. `Writer` Monad

When we first embarked on our Monadic Journey, we encountered a problem with logging in a Purely Functional Language. It was easy to accomplish in an Imperative Language using Global Variables with all of the caveats that comes with that approach.

But we had to jump through some hoops to do this with Pure Functions. And then we invented `Debuggable` and ultimately encapsulated the management of the logging Side-effect in `SideEffect`, which turned out to be Isomorphic to the `Monad`.

Well, we're now going to learn about the Writer Monad which codifies this very problem in the Type `Writer`. Here's its Type definition:

```
newtype Writer w a = Writer (Tuple a w) ❶
```

❶ The reason for `a` being in the first position of the `Tuple` is for historical reasons. If it were up to me, they would be swapped.

Not unlike our `Debuggable` it contains a `Tuple` of the computation, `a`, and a log `w`.

And now we can write the `Functor` Instance for `Writer`. As always, we'll do it one line at a time starting with the instance definition:

```
instance functorWriter :: Functor (Writer w) where
```

We hold the Type `w` constant, i.e. we can only `map` over the computation Type, `a`.

Next, we'll code the left-hand side of `map`:

```
instance functorWriter :: Functor (Writer w) where
  map f (Writer (Tuple x l)) = ???
```

We're destructuring the `Writer` and `Tuple` to dig out the Value, `x` and the log, `l`.

And finally, we'll apply `f` to `x` and wrap everything back up:

```
instance functorWriter :: Functor (Writer w) where
  map f (Writer (Tuple x l)) = Writer (Tuple (f x) l)
```

Next, we'll write the `Apply` Instance one line at a time starting with the instance definition:

```
instance applyWriter :: Apply (Writer w) where
```

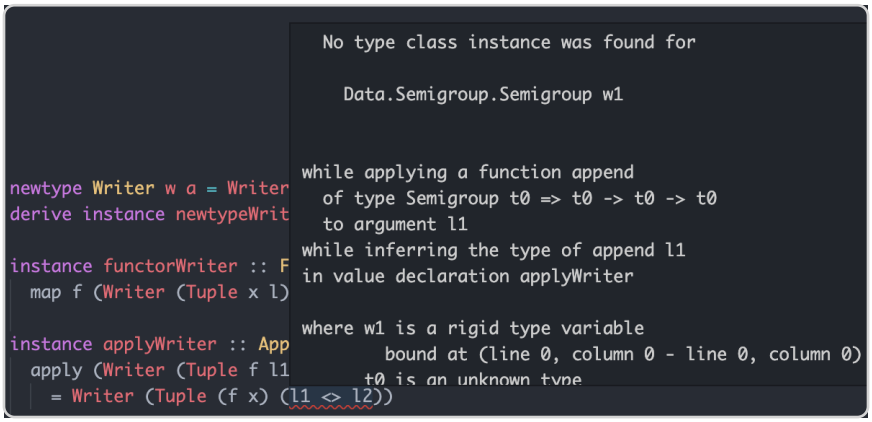
Next, we'll write the left-hand side of `apply` to destructure both Parameters to expose their components:

```
instance applyWriter :: Apply (Writer w) where
  apply (Writer (Tuple f l1)) (Writer (Tuple x l2)) = ???
```

We now can apply `f` to `x` and combine the logs:

```
instance applyWriter :: Apply (Writer w) where
  apply (Writer (Tuple f l1)) (Writer (Tuple x l2)) =
    Writer (Tuple (f x) (l1 <> l2))   -- COMPILER ERROR!!
```

But we get a compiler error:



This tells us that `w1` (or what we called `w`) doesn't have a `Semigroup` Instance. This is because we used `<>` on `l` which has Type `w`. We'll need to add a Constraint:

```
instance applyWriter :: Semigroup w => Apply (Writer w) where ❶
  apply (Writer (Tuple f l1)) (Writer (Tuple x l2)) =
    Writer (Tuple (f x) (l1 <> l2))
```

❶ `Semigroup` Constraint added.

And next, we'll write the `Applicative` Instance one step at a time:

```
instance applicativeWriter :: Semigroup w => Applicative (Writer w) where
```

We've proactively added the `Semigroup` Constraint since `Applicative` is also an `Apply` and it has that Constraint.

Now we can write the left-hand side of `pure`:

```
instance applicativeWriter :: Semigroup w => Applicative (Writer w) where
  pure x = ???
```

Now, we want to create a Writer with NO log, i.e. an empty log. There's that magical word, `empty`. That means that we're going to have to tighten our requirements. `w` can't just be a `Semigroup` anymore. Now it also has to be a `Monoid`:

```
instance applicativeWriter :: Monoid w => Applicative (Writer w) where
  pure x = Writer (Tuple x mempty) ❶
```

❶ We wrap `x` in a `Tuple` with an empty log and then wrap that in a `Writer`.

Now we can write the `Bind` Instance one line at a time:

```
instance bindWriter :: Semigroup w => Bind (Writer w) where
```

Again, we've added the `Semigroup` Constraint since `Bind` is also an `Apply` and it has that Constraint. Notice that we do NOT need to make it a `Monoid`. That's because a `Bind` is an `Apply` NOT an `Applicative` and `Apply` only requires `Semigroup`.

Next, we'll write the left-hand side of `bind`:

```
instance bindWriter :: Semigroup w => Bind (Writer w) where
  bind (Writer (Tuple x l1)) f = ???
```

We destructure the `Writer` to get at its components. And now we can apply `f` to `x`:

```
instance bindWriter :: Semigroup w => Bind (Writer w) where
  bind (Writer (Tuple x l1)) f = f x ???
```

We'll need to remember `f's` Type Signature is `a -> m b`, or specifically, `a -> Writer w b`, which means that we're going to get a second `Writer`. So, we'll pass it to a Lambda to destructure it so we can get access to its components:

```
instance bindWriter :: Semigroup w => Bind (Writer w) where
  bind (Writer (Tuple x l1)) f = f x # \(Writer (Tuple y l2)) -> ???
```

At this point `y` is the final result of the computation and we have 2 logs to combine. And we'll also need to wrap it back up in a `Writer`:

```
instance bindWriter :: Semigroup w => Bind (Writer w) where
  bind (Writer (Tuple x l1)) f =
    f x # \(Writer (Tuple y l2)) -> Writer (Tuple y (l1 <> l2))
```

Finally, we'll write the `Monad` Instance:

```
instance monadWriter :: Monoid w => Monad (Writer w)
```

We had to use the Monoid Constraint since a Monad is also an Applicative, which has a Monoid Constraint.

But before we're done, we should look at our apply and bind:

```
instance applyWriter :: Semigroup w => Apply (Writer w) where
  apply (Writer (Tuple f l1)) (Writer (Tuple x l2)) =
    Writer (Tuple (f x) (l1 <> l2))

instance bindWriter :: Semigroup w => Bind (Writer w) where
  bind (Writer (Tuple x l1)) f =
    f x # \(Writer (Tuple y l2)) -> Writer (Tuple y (l1 <> l2))
```

We can see that their general behavior is identical, i.e. they apply f and x to get the final result and combine the logs in a Left-Associative way.

So, let's use ap instead of our apply implementation:

```
instance applyWriter :: Semigroup w => Apply (Writer w) where
  apply = ap -- COMPILER ERROR!!
```

And, as usual, the compiler is unhappy with us. But why? Let's look at the error message:

```
newtype Wr   No type class instance was found for
derive ins
               Data.Monoid.Monoid w0
instance f
  map f (W
while checking that type forall m a b. Monad m => m (a -> b) -> m a -> m b
  is at least as general as type Writer w0 (a1 -> b2) -> Writer w0 a1 -> Writer w0 b2
instance a
  apply = ap
```

It says that w0, or w as we called it, is not a Monoid. Why do you think that, all of a sudden, the compiler expects w to be a Monoid instead of a Semigroup?

We'll remember that ap has a Constraint of Monad, which is both a Bind and an Applicative:

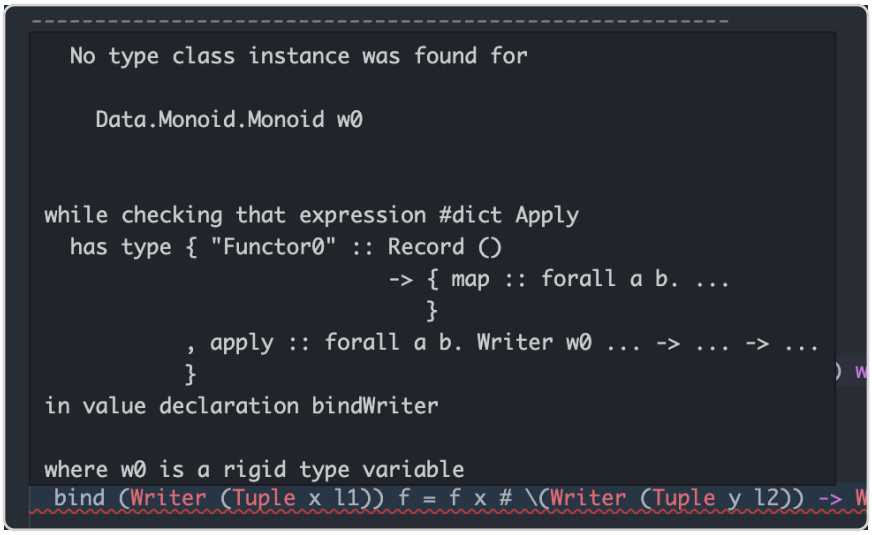
```
ap :: ∀ a b m. Monad m => m (a -> b) -> m a -> m b
ap mf mx = do
    f <- mf
    x <- mx
    pure $ f x
```

Our Bind Instance has a Constraint of Semigroup, but it's our Applicative Instance that has a Constraint of Monoid that's causing us grief.

For us to use ap, we'll have to tighten our requirements on Apply, by changing the Constraint from Semigroup to Monoid:

```
instance applyWriter :: Monoid w => Apply (Writer w) where
  apply = ap
```

And, to no one's surprise, a brand new compiler error. This time on our Bind Instance implementation:



Again, it now wants our w to have a Monoid Constraint, but this time on Bind. That's because Bind is an Apply and since we just changed Apply's Constraint to Monoid, we'll need to change Bind's Constraint to match:

```
instance bindWriter :: Monoid w => Bind (Writer w) where
  bind (Writer (Tuple x l1)) f =
    f x # \(Writer (Tuple y l2)) -> Writer (Tuple y (l1 <> l2))
```

And finally, the compiler is happy. But, notice that's there's no such thing as a free lunch. We cheated by not writing Apply and relied on our Monad implementation instead. This came at the cost of forcing the log to be a Monoid instead of a Semigroup.

But, this cost is fairly low, since we're probably going to use the Monad instance for Writer, which already had a Monoid Constraint. The use case of using Writer as ONLY an Apply, where we'd only require a Semigroup, is probably an edge case.

18.18. `Writer` Helper Functions

When a `Monad` has helper Functions that allow the computation to interact with the Side-effect, I like to refer to them collectively as the ***Monad's API***.

And `Writer` has such an API:

```
tell :: ∀ w. w -> Writer w Unit
listen :: ∀ a w. Writer w a -> Writer w (Tuple a w)
pass :: ∀ a w. Writer w (Tuple a (w -> w)) -> Writer w a
```

Of these API Functions, `tell` is the most used and so we'll take a closer look at that Function. What we learn there can apply to `listen` and `pass`, as well as any other Monad API Functions.

`tell` will let us specify something to be appended to our "log", for example:

```
doNothingWithLog :: Writer (Array String) Int
doNothingWithLog = do
  tell ["We did nothing"] ❶
  pure 0
```

❶ No need to discard the extracted Type here since it's `Unit`, which is discardable thanks to its `Discard` Instance.

Notice the Type of `tell's` Pure Computation. It's `Unit`. Why would we call a Function in a Purely Functional Programming Language that only returns a `Unit`. What would be the point of that Function? Couldn't we just use `unit` instead of that Function?

Think back to this Javascript Function:

```
console.log("Hello, world!");
```

It doesn't return anything either. Then why call it?

Because we want its Side-effect, i.e. to write on the console. And it's the same with `tell`. It's Side-effect is to append to the `Monoid`. To better understand how `tell` accomplishes this, let's look at its implementation:

```
tell :: ∀ w. w -> Writer w Unit
tell l = Writer (Tuple unit l)
```

Okay, so that tells us nothing (no pun intended). How does `tell` perform its Side-effect?

It doesn't.

Think back to when we wrote the `Bind` Instance:

```
instance bindWriter :: Monoid w => Bind (Writer w) where
  bind (Writer (Tuple x l1)) f =
    f x # \(Writer (Tuple y l2)) -> Writer (Tuple y (l1 <> l2))
```

It's the `bind` implementation where the Side-effects are handled, i.e. the logs are appended. All that `tell` does is provide `bind` with a `Writer` that has a log.

If this is a bit difficult to see, let's rewrite our `doNothingWithLog` without `do` notation and use `>>=` and `*>` instead. First, here's `doNothingWithLog` using `do` notation:

```
doNothingWithLog :: Writer (Array String) Int
doNothingWithLog = do
  tell ["We did nothing"]
  pure 0
```

And now with `>>=`:

```
doNothingWithLog :: Writer (Array String) Int
doNothingWithLog = tell ["We did nothing"] *> pure 0
```

So, technically, we didn't use `>>=`. Instead, we only used `*>`, which uses `<*>` under the covers. This is fine because in our original `apply` implementation, its Side-effect was to append the logs. But after we changed `apply` to use `ap`, it now uses `bind` and so, we're using `>>=` indirectly.

If you are still doubtful that we're using `>>=`, even though we are, we can rewrite it explicitly using `>>=`:

```
doNothingWithLog :: Writer (Array String) Int
doNothingWithLog = tell ["We did nothing"] >>= \_-> pure 0
```

Now, to see how the Side-effect is handled in gory detail, let's reduce this implementation. First, we'll use `bind` instead of `>>=`:

```
doNothingWithLog :: Writer (Array String) Int
doNothingWithLog = bind (tell ["We did nothing"]) (\_-> pure 0)
```

And now we'll substitute `tell`, with its definition, `tell l = Writer (Tuple unit l)`:

```
doNothingWithLog :: Writer (Array String) Int
doNothingWithLog = bind
  (Writer (Tuple unit ["We did nothing"])) (\_-> pure 0)
```

And next, we'll substitute `pure` with its definition, i.e. `pure x = Writer (Tuple x mempty)` and apply `0` to it:

```
doNothingWithLog :: Writer (Array String) Int
doNothingWithLog = bind
  (Writer (Tuple unit ["We did nothing"])) (\_-> Writer (Tuple 0 mempty))
```

Then we'll substitute `bind` with its definition and apply the 2 Parameters. First, here's bind's definition:

```
instance bindWriter :: Monoid w => Bind (Writer w) where
  bind (Writer (Tuple x l1)) f =
    f x # \(Writer (Tuple y l2)) -> Writer (Tuple y (l1 <> l2))
```

And now we'll substitute:

```
doNothingWithLog :: Writer (Array String) Int
doNothingWithLog = (\_-> Writer (Tuple 0 mempty)) unit
  # \(Writer (Tuple y l2)) -> Writer (Tuple y (["We did nothing"] <> l2))
```

And then we'll apply the first Function:

```
doNothingWithLog :: Writer (Array String) Int
doNothingWithLog = (Writer (Tuple 0 mempty))
  # \(Writer (Tuple y l2)) -> Writer (Tuple y (["We did nothing"] <> l2))
```

Then we'll apply using `#`, where `0` binds to `y` and `mempty` binds to `l2`.

```
doNothingWithLog :: Writer (Array String) Int
doNothingWithLog = Writer (Tuple 0 (["We did nothing"] <> mempty))
```

And finally, we'll apply `<>`:

```
doNothingWithLog :: Writer (Array String) Int
doNothingWithLog = Writer (Tuple 0 ["We did nothing"])
```

Sometimes, it's easy to forget that `bind` is happening when you're looking at code that uses `do` notation, which is good and bad. It's good because `do` notation greatly reduce cognitive overhead and code clutter. But, it's bad because the Side-effect can seem like ***Spooky Action at a Distance***, even though it's not.

When we look at the code written with `>>=`, it reminds us of the fact that the `bind` implementation is working its magic, which in the case of `Writer` is appending to the log.

We can also think of this happening BETWEEN lines in code written in `do` notation:

```
doNothingWithLog :: Writer (Array String) Int
doNothingWithLog = do
  tell ["We did nothing"]
    -- append the preceeding log, i.e. ["We did nothing"]
    -- to the following log, i.e. mempty
  pure 0
```

That's because in `do` notation there's an implied `>>=` between each line. And since `>>=` is the `bind` implementation, it's behavior is hidden from us and appears to be magic. But it's not. It's actually what I like to think of as a ***Parallel Computation***.

18.19. Parallel Computations

There are 2 Computations that are present in every Monad. First is the Pure Computation, i.e. the code that you've written in the computation, for example:

```
gauntlet :: Int -> Either String Int
gauntlet x = do
  o <- oddTest x
  let y = o + 1
  void $ greaterThanTest 10 y
  lessThanTest 20 y
```

Here the Pure Computation is the calls to `oddTest`, `greaterThanTest` and `lessThanTest`, plus the addition of `1` to `o`.

The second computation is the Monadic Computation, i.e what the `bind` implementation is doing. In the above example, it's checking for errors:

```
instance bindMaybe :: Bind Maybe where
  bind Nothing _ = Nothing ❶
  bind (Just x) f = f x
```

❶ The compiler is generating `if` logic to check Patterns, which, effectively, checks for errors in this implementation.

And since the Monad's `bind` implementation does this work for us, we're freed to code without thinking too much about it.

Here are some common Monads and their Monadic Computations:

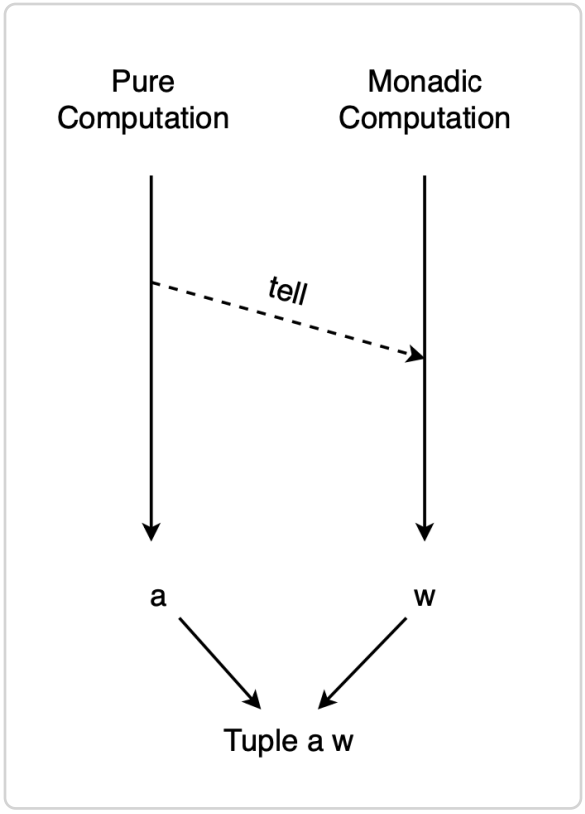
```
Identity  -> No Computation
Maybe     -> Error Checking
Either    -> Error Checking
Writer    -> Log Appending
Reader    -> Threading Read Only Value
State     -> Threading State
```

We haven't seen `Reader` or `State` just yet so their Monadic Computation may not make sense, but they will.

But look at `Maybe` or `Either` and notice what we've been calling a Side-effect is really a Parallel Computation, i.e. the Monadic Computation that's being performed every time we use `bind`. Visually, it's hidden from us, buried in the implementation of `>>=`, and even more so thanks to `do` notation.

But it's there all the same and the compiler certainly won't forget about it.

Another thing that's important to think about is that the Monad's API Functions help bridge the gap between the Parallel Computations. Here's a diagram to illustrate how `tell` does this:



The Pure Computation produces a Value of Type `a`, whereas the Monadic Computation produces a Value of Type `w`. And they are returned in a `Tuple` by `Writer`.

The Monadic API Function, `tell`, bridges the gap by allowing a WRITE from the Pure Computation to the Monadic Computation. This makes sense since a `Writer` is a Monad for writing to a log and the only way we can do that is from the Pure Computation.

But what about reading from the Monadic Computation?

18.20. `Reader` Monad

Imagine we've written an application that uses configuration information at runtime by reading it from a file or by calling a Web Server to download a JSON file with the appropriate configuration, or even simply reading the Command Line Parameters or Environment Variables.

In an Imperative Program, we might store this information in one or more Global Variables. Then every Function in our Program could consult them when needed. Even in Object-oriented Languages, we can accomplish Global Access by using a Singleton Classes or Static Class Variables depending on the language.

But, in Functional Languages, we've relegated the Global Variable to the same junk heap where we, long ago, tossed out the `goto` command. So, the only choice we have is to pass it to nearly EVERY Function. At the very least, we'll have to pass it to all Functions at the top of the call stack and typically, we'll have to pass it down much further.

This is a daunting task that pollutes ever Type Signature it encounters. But now that we know that Monads

can maintain their own Computation, maybe we can leverage them to do the tedious work of passing this configuration information around.

Turns out that someone long ago had this same brilliant idea and produced a Monad called Reader. Here's its definition:

```
newtype Reader r a = Reader (r -> a)
```

Looking at the Type Constructor, i.e. the lefthand Reader, we see it takes 2 Type Parameters. r is the Type that the Reader will maintain and pass to every Function for us and a is the result of the Pure Computation.

Turning our attention to the Data Constructor, i.e. the righthand Reader, we see it takes 1 Parameter and that's a Function with the Type Signature r -> a. This Function takes an r, i.e. some read-only Value and produces an a, i.e. its Pure Computation.

Let's look at how this will be used to give us a better understanding of Reader:

```
type Config = ❶
  { numDecimalPlaces :: Int
  , debug :: Boolean
  }

funcWithConfig :: Int -> Reader Config Int ❷
funcWithConfig v = do
  { numDecimalPlaces } <- ask ❸ ❹
  ... ❺

main :: Effect Unit
main = do
  value <- runReader (funcWithConfig 42) ❻
    { numDecimalPlaces = 2
    , debug = false
    }
  ... ❼
```

❶ This is our configuration Type.

❷ funcWithConfig takes an Int and returns a Reader. Keep in mind, like our Parser, we'll need to run this using a Run Function.

❸ ask is a Reader API Function that returns the whole read-only Value. Notice the backwards arrow. That's because ask returns a Reader r a and we want to extract the Value of Type a.

❹ We are destructuring the extracted Value to only get numDecimalPlaces from Config. This is because funcWithConfig doesn't need any other fields from the Config Record.

❺ The remainder of the code isn't important.

❻ `runReader` is used to run `funcWithConfig`. This is similar to `runFoldL` and `parse` from earlier Chapters. We need `runReader` because `Reader` contains a Function. When we wrote `bind` for our `Parser`, we noted that `bind` builds a complex composition of Functions and returns that composition in a new `Parser`, which is why it needs a Run Function. `Reader` is going to do the exact same thing. More on this later.

❼ The remainder of the code isn't important.

When `runReader` is called in `main`, `r` is passed in so it can be passed through our Functions. This is `Reader's` whole reason for existing, to pass `r` to the Functions that have been bound together via `bind` and `apply`. In our contrived example, `Config` is our `r` Type.

Let's write the `Functor` instance for `Reader` starting with the instance definition:

```
newtype Reader r a = Reader (r -> a) ❶

instance functorReader :: Functor (Reader r) where ❷
```

❶ The Type definition for reference.

❷ Notice we keep `r` constant, i.e. we can only map over `a`.

Now we'll code the left-hand side of `map`:

```
instance functorReader :: Functor (Reader r) where
  map f (Reader g) = ???
```

We've destructured the `Reader` to gain access to its Function, `g`.

The next step is the difficult part. So pay close attention because in the next Chapter, you're going to have to write some Monads just like this.

Do you remember the pattern we discovered when we were writing the `Parser` Instances?

If you don't remember, don't worry since we're going to rediscover it here.

What we do know is that we're going to need to return a `Reader` so we can start with that:

```
instance functorReader :: Functor (Reader r) where
  map f (Reader g) = Reader ???
```

And, we also know that the `Reader` Data Constructor takes a Function, which takes 1 Parameter, a read-only Value, that we'll call `r`:

```
instance functorReader :: Functor (Reader r) where
  map f (Reader g) = Reader \r -> ???
```

This is the pattern I was referring to. When Functors contain Functions, you almost always start coding like this. We'll also start with this as our template when we code apply and bind.

The final step is the next hurdle. What do we know about Functors and map's job?

We know that Functors map OUTPUTS. We also know that the Type Signature for g is r -> a. From these facts, we should be able to deduce that we need to first pass r to g, which will result in a Value of Type a, and then finally map f over that result:

```
instance functorReader :: Functor (Reader r) where
  map f (Reader g) = Reader \r -> f $ g r
```

Next up is the Apply Instance. Let's start with the instance definition:

```
instance applyReader :: Apply (Reader r) where
```

Again, we'll start with the left-hand side of apply:

```
instance applyReader :: Apply (Reader r) where
  apply (Reader ff) (Reader fx) = ???
```

We're destructuring to gain access to the underlying Functions, ff and fx. The prefixed f in the names is to remind us that these are Functions, i.e. ff is a Function that will return a Function, and fx is a Function that will return a Value.

And now, we'll start with the aforementioned pattern on the right-hand side:

```
instance applyReader :: Apply (Reader r) where
  apply (Reader ff) (Reader fx) = Reader \r -> ???
```

Calling ff with r will return a Function and calling fx with r will return a Value. We can then pass the resulting Value, fx r, to the resulting Function, ff r:

```
instance applyReader :: Apply (Reader r) where
  apply (Reader ff) (Reader fx) = Reader \r -> ff r $ fx r
```

Now for the `Applicative` Instance.

We'll start with the definition:

```
instance applicativeReader :: Applicative (Reader r) where
```

Next, the left-hand side of `pure`:

```
instance applicativeReader :: Applicative (Reader r) where
  pure x = ???
```

Now all we have to do is create a `Reader` that ultimately returns `x`:

```
instance applicativeReader :: Applicative (Reader r) where
  pure x = Reader \r -> x ❶
```

❶ Notice that same pattern again.

If we look at what we wrote, it should be obvious that `r` is never used, so let's eliminate it:

```
instance applicativeReader :: Applicative (Reader r) where
  pure x = Reader \_ -> x
```

Does `\_ -> x` look familiar? We've seen this before. It's a Function that throws away one of its Parameters and returns a constant value.

`\_ -> x` is equivalent to `const x`. So, let's make that change:

```
instance applicativeReader :: Applicative (Reader r) where
  pure x = Reader $ const x
```

And finally, we can Eta-reduce this by canceling the rightmost variable, i.e. `x`, on both sides of the equal sign:

```
instance applicativeReader :: Applicative (Reader r) where
  pure = Reader <<< const ❶
```

❶ Notice, how Function Application, `$`, gets replaced with Function Composition, `<<<`, during an Eta-reduction step.

Next, we'll write the Bind Instance.

We'll start with the definition:

```
instance bindReader :: Bind (Reader r) where
```

And then the left-hand side of bind:

```
instance bindReader :: Bind (Reader r) where
  bind (Reader fx) f = ???
```

We've destructured Reader to unwrap fx, which is a Function, which, when called with an r, will return a Value.

Now we add our, all too familiar, pattern:

```
instance bindReader :: Bind (Reader r) where
  bind (Reader fx) f = Reader \r -> ???
```

This is where things get difficult. So, let's go slowly through building a strategy for how to attack this problem.

First, we'll need to get the Value from fx by passing it r. Then we can apply that result to f. But unlike the mapping Functions in Functor or Apply, our f is a Monadic Function, i.e. it returns an Reader r b not b.

So, we're probably going to have a Type mismatch problem, i.e. a Reader inside of another Reader instead of just a Reader.

There are 2 approaches for moving forward. The first is to just code up what we know we need to do and see what complaints the compiler makes and figure out what to do afterwards. This approach is risky since the compiler won't always be helpful.

The second approach is to write out all the Types and figure out how to code it from that. This is less risky and will always work, but requires a bit more effort.

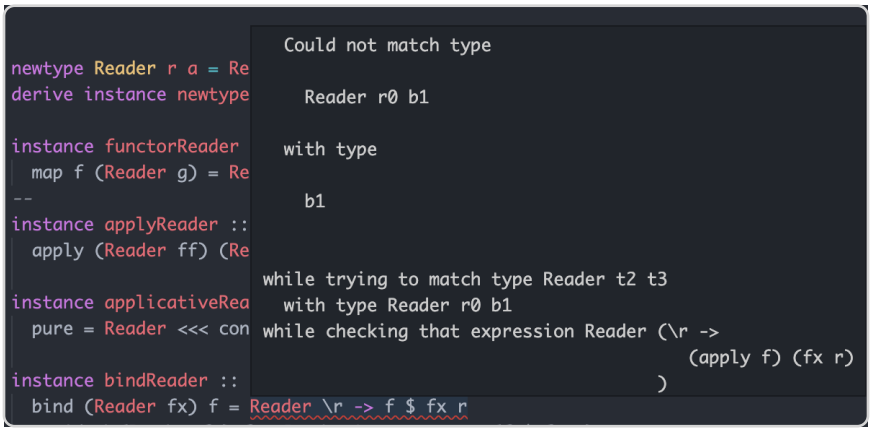
If the first approach fails us, then we'll be forced to use the second approach.

So, let's be lazy and start with the first approach and if we have to, we can always fallback to the second.

We know we have to call f with the Value fx r. So, we'll write just that and hope the compiler error can help us:

```
instance bindReader :: Bind (Reader r) where
  bind (Reader fx) f = Reader \r -> f $ fx r -- COMPILER ERROR!!
```

Here's the compiler error:



It cannot match the Type that it found, `Reader r0 b1`, with the Type it expects, `b1`. We should also look at the red underline for clues. It's underlining the whole right-hand side of `bind`.

Is this enough information to tell you what to do next?

I don't know about you, but I don't find this error super helpful. Clearly, we picked the wrong approach. If you ask me, this approach doesn't work often enough. Too bad because it would've been so much easier.

But writing out Types isn't really that hard. We were just being lazy and now we've spent more time than if we would have just started with the second approach first.

So now, we'll write out the pertinent Types:

```
f :: a -> Reader r b ❶
fx :: r -> a ❷
fx r :: a ❸
f $ fx r :: Reader r b ❹
Reader \r -> f $ fx r :: Reader r (Reader r b) ❺
whatWeWant :: Reader r b ❻
```

❶ The Monadic Function that `bind` gets as its second Parameter. How do we know this Type? Because `bind` always gets a Monadic Function as its second Parameter, i.e. a Function of Type `a -> m b`, where `m` is `Reader r`.

❷ The Function we unwrapped from the `Reader` in `bind's` first Parameter.

❸ Applying `r` to `fx` to get the Value out.

❹ Applying the Monadic Function to our extracted Value.

❺ The code we wrote that the compiler is complaining about.

❻ The actual Type we want.

What we want is `Reader r b` but we have `Reader r (Reader r b)`, i.e. the `b` part in what we want has a `Reader r b` in it instead. This is exactly what the compiler was trying to tell us earlier. It found a `Reader r0 b1` where a `b1` should be. It's obvious now, but only after we wrote out all of the Types. It figures.

How do we fix this? We have a `Reader` inside of another `Reader`, or more generally a `Monad` inside another `Monad`.

Do you remember `join`? What does join do?

Here's the definition for `join`:

```
join :: ∀ a m. Bind m => m (m a) -> m a
join x = x >>= identity
```

This may be a bit too abstract. So let's replace `m` with `Reader r` in the Type Signature:

```
join :: ∀ a. (Reader r) ((Reader r) a) -> (Reader r) a
```

And we'll remove the extra Parentheses and replace `a` with `b` to better match our situation:

```
join :: ∀ b. Reader r (Reader r b) -> Reader r b
```

This will take our `Reader r (Reader r b)`, i.e. the Type our code returns, and give us `Reader r b`, i.e. the Type we ultimately want. That should work.

So let's add join as the final step in our code:

```
instance bindReader :: Bind (Reader r) where
  bind (Reader fx) f = join $ Reader \r -> f $ fx r
```

And now the compiler is happy. But we won't be. It's probably not obvious why, but I've made this mistake way too many times and I want to save you the headache.

It's possible to write a Function that makes the compiler happy, but that doesn't mean that we'll be. For example, here's a Function that makes the compiler happy:

```
infiniteLoop :: AnyValidType -> AnyValidType
infiniteLoop x = infiniteLoop x ❶
```

❶ By the way, if you Eta-reduce this the compiler will catch this infinitely recursion definition.

`infiniteLoop's` return Type is met by calling itself. The compiler is happy with this because the Types match. But when we call this Function, we won't be happy one bit. Not unless we want our program to sit and spin.

And the same is true about our use of `join`. To see why, let's look at `join` and `bind` together:

```
join :: ∀ a m. Bind m => m (m a) -> m a
join x = x >>= identity

instance bindReader :: Bind (Reader r) where
  bind (Reader fx) f = join $ Reader \r -> f $ fx r
```

Can you see how this is like our `infiniteLoop` example?

Hopefully, you noticed how `join` is written in terms of `bind`, via `>>=`, and `bind` is written in terms of `join`.

This will cause an infinite loop when we run our program. `bind` will call `join` and `join` will call `bind`. They form an infinite recursion.

That means that we can't use `join` as our solution to our Type problem. We'll have to go back to where we started and try something else.

Here's where we left off, i.e. before adding `join`:

```
instance bindReader :: Bind (Reader r) where
  bind (Reader fx) f = Reader \r -> f $ fx r -- COMPILER ERROR!!
```

And here's the Types we came up with:

```
f :: a -> Reader r b
fx :: r -> a
fx r :: a
f $ fx r :: Reader r b
Reader \r -> f $ fx r :: Reader r (Reader r b)
whatWeWant :: Reader r b
```

The problem with what we wrote is that we have 2 `Readers`. The outer `Reader`, `Reader \r -> f $ fx r`,

and the inner Reader, f $ fx r. We'd rather have the inner Reader be something of Type b not Reader r b. But, if we run the inner Reader, we'll get a Value of Type b, which would solve our problem.

But how do we run a Reader?

Do you remember seeing runReader in our example on how to use Readers? Well, runReader's job is to run Readers. Except, we never wrote that Function. Once we do, then we can use it in our bind code to eliminate the inner Reader. So let's do that now.

The first step in writing runReader is to decide on its Type Signature. runReader will need 2 things, a Reader and a read-only Value to pass to all of the Functions.

There are 2 ways we could write the Type Signature, but the following is the simplest:

```
runReader :: ∀ a r. Reader r a -> r -> a ❶
```

❶ We could've written the Type Signature flipped, i.e. r -> Reader r a -> a.

When we add the implied Parentheses, we can see more clearly how to implement this:

```
runReader :: ∀ a r. Reader r a -> (r -> a)
```

With this version of the Type Signature, it should be clear that runReader simply unwraps the Reader's Function, which results in the following implementation:

```
runReader :: ∀ a r. Reader r a -> (r -> a)
runReader (Reader f) = f
```

By the way, we could've also leveraged unwrap from Newtype had we made Reader an instance of Newtype. But runReader is more traditional thanks to Reader's history from Haskell.

Now let's apply it to our problematic bind implementation.

Here's where we are so far:

```
instance bindReader :: Bind (Reader r) where
  bind (Reader fx) f = Reader \r -> f $ fx r -- COMPILER ERROR!!
```

Our plan is to run the Reader returned by f $ fx r, so it will give us a b. Let's write the Type for running the Reader:

```
runReader (f $ fx r) :: r -> b
```

Okay, so it doesn't exactly give us a b directly. We'll have to pass an r to the Function runReader to get a b:

```
instance bindReader :: Bind (Reader r) where
  bind (Reader fx) f = Reader \r -> runReader (f $ fx r) r
```

Not only does this compile, it's NOT an infinite loop.

Now that we have a valid bind implementation, the only instance left to write is the Monad Instance:

```
instance monadReader :: Monad (Reader r)
```

Technically, we should validate the Monad Laws BEFORE we write this line of code. Feel free to do so if you want to see for yourself, but, for expedience sake, I'm going to assume that someone has done this already for us since Reader has been in Haskell long before PureScript existed.

At this point, we have written the following:

```
newtype Reader r a = Reader (r -> a)

runReader :: ∀ a r. Reader r a -> (r -> a)
runReader (Reader f) = f

instance functorReader :: Functor (Reader r) where
  map f (Reader g) = Reader \r -> f $ g r

instance applyReader :: Apply (Reader r) where
  apply (Reader ff) (Reader fx) = Reader \r -> ff r $ fx r

instance applicativeReader :: Applicative (Reader r) where
  pure = Reader <<< const

instance bindReader :: Bind (Reader r) where
  bind (Reader fx) f = Reader \r -> runReader (f $ fx r) r
```

The amount of work it took us to write this is certainly NOT reflected in the number of lines of code. This is what I mean by 2 hours of thinking and 2 minutes of coding. This is certainly better than the alternative since it's better to have 10 lines of code to maintain than 100.

The last thing to write is the Reader API:

```
ask :: ∀ r. Reader r r
asks :: ∀ a r. (r -> a) -> Reader r a
```

`ask`, which we saw in the example for how to use `Readers`, returns the full read-only Value.

In contrast, `asks`, takes a Function that will accept the full read-only Value and return at least some portion of it. Technically speaking, this Function could take any or all of the read-only Value and return some computation on it, but typically, it only returns a field from a Record.

Let's start by examining the Type Signature for `ask`:

```
ask :: ∀ r. Reader r r
```

Notice that the second Type Parameter of `Reader r r` is an `r`, i.e. the Type of our read-only Value. The second Type Parameter is normally reserved for the computation Type. But for `ask`, the computation ***is*** the read-only Value. This is the only way we can get our hands on it.

Let me reiterate that because it's important. Returning the read-only Value as the Pure Computation is how `ask` bridges the gap between the Parallel Computations in the `Reader` Monad.

And now, we'll write the implementation starting with our familiar pattern:

```
ask :: ∀ r. Reader r r
ask = Reader \r -> ???
```

We know enough to write up to this point having seen this pattern over and over again, but now, we have to return something of Type `r`. The only thing we have that's Type `r` is the Value `r` and so we'll return that:

```
ask :: ∀ r. Reader r r
ask = Reader \r -> r
```

I hope you're dying to change `\r -> r` to `identity` like I am:

```
ask :: ∀ r. Reader r r
ask = Reader identity
```

That's all there is to `ask`. When the `Reader` returned by `ask` is run, it'll return the read-only Value.

And now we'll start with the Type Signature for `asks`:

```
asks :: ∀ a r. (r -> a) -> Reader r a ❶
```

❶ `asks` takes a Function that will take the full read-only Value, `r`, and return some or all of it, `a`.

And now the right-hand side of the equal sign:

```
asks :: ∀ a r. (r -> a) -> Reader r a
asks f = ???
```

Next, let's add our common pattern, since we're returning a `Reader`:

```
asks :: ∀ a r. (r -> a) -> Reader r a
asks f = Reader \r -> ???
```

Now the only thing that we can do is apply `f` to `r`:

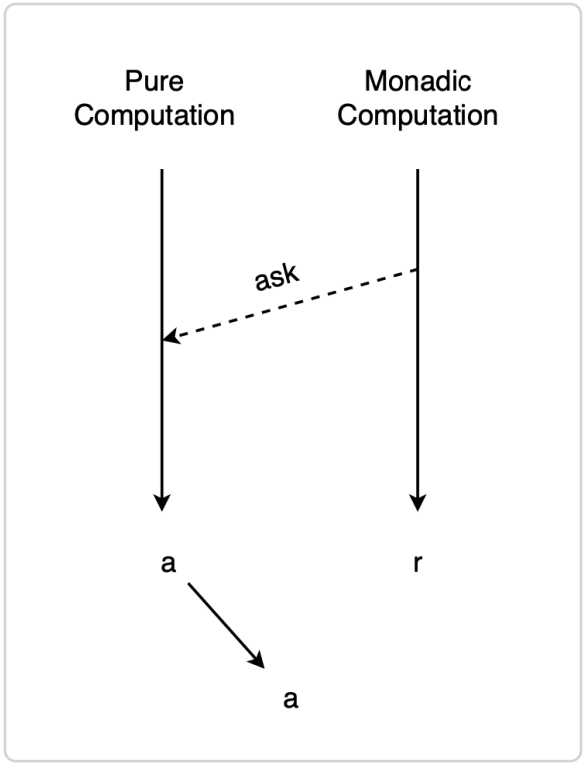
```
asks :: ∀ a r. (r -> a) -> Reader r a
asks f = Reader \r -> f r
```

And that concludes the API:

```
ask :: ∀ r. Reader r r
ask = Reader identity

asks :: ∀ a r. (r -> a) -> Reader r a
asks f = Reader \r -> f r
```

Here's a diagram that illustrates how `ask` bridges the gap between the Pure Computation and the Monadic one:



The Pure Computation produces a Value of Type `a`, whereas the Monadic Computation produces a Value of Type `r`. But only `a` is returned by `Reader`. There is no need to return `r` since it's a read-only Value and never changes. And since we have to provide this Value to `runReader`, we already have access to it, if we need it later.

Also, notice that `ask` goes in the opposite direction from `tell` (see earlier diagram). That's because `ask` is reading, whereas `tell` is writing.

But what if we want to read ***and*** write?

18.21. `State` Monad

We cannot use Global State in PureScript. Well, technically, there are mechanisms that we'll go into later in the Beyond Section of this book for multi-threaded programs. So, I'll restate that. We SHOULD NOT use Global State in PureScript, except when there is no other choice.

But, in a single thread of execution, we have a choice and that is to pass State around. We've seen this in `foldl` and `foldr`. What is often called the Accumulator is nothing more than the State of the iteration. When we fold, we're iterating with State where the new State is passed from the current iteration to the next.

But, what if we want to carry State around for longer than a fold? We've seen how `Reader's` Side-effect is to pass a read-only Value to every Function. That means we should be able to do the same thing with State except the Value won't be read-only.

Let's model the Type. But before we do, we should consider how the Type compares with `Reader`. Here's `Reader's` Type definition for reference:

```
newtype Reader r a = Reader (r -> a)
```

This would probably work for State with a few things renamed:

```
newtype State s a = State (s -> a) -- NOT IDEAL
```

But there's a downside to this definition. Can you see what it is?

With Reader, we didn't return the read-only value because it was never modified. But with State, that will not be the case. We should return the new State. While this may be nice to have at the end of the Computation, it's imperative to have if we hope to compose Monadic Functions.

A Function will change the State and return it along with its computation. We'll take that new State and pass it to the next Function and so on. This is the reason State's Function must also return the new State otherwise we can only make State a Functor.

Here's our proper version of State:

```
newtype State s a = State (s -> Tuple a s)
```

Now, when State is run, it'll return both the computation of Type a and the new State of Type s.

We'll start coding with Functor one line at a time:

```
instance functorState :: Functor (State s) where
```

Realize, we're holding s constant, i.e. we'll be mapping over a. If you haven't noticed, we always leave a as our last Type Parameter in these cases. Then we can hold all the other Type Parameters constant.

And now for the left-hand side of map:

```
instance functorState :: Functor (State s) where
  map f (State fx) = ???
```

Here the name fx reminds us that it's a Function that returns a Value.

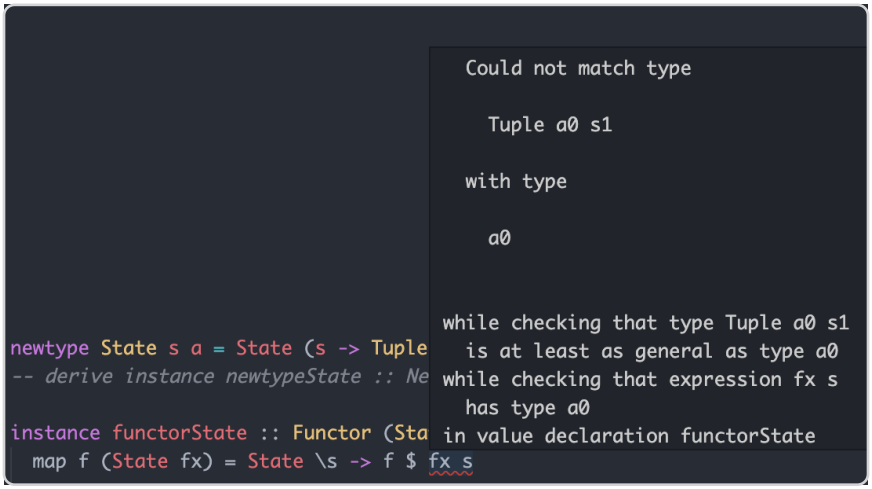
Let's start with our pattern on the right-hand side:

```
instance functorState :: Functor (State s) where
  map f (State fx) = State \s -> ???
```

Just like we had with `Reader` and `Writer` before, we need to apply `f` to `fx's` Value somehow. We've done this before by passing `s` to `fx` to extract the Value and then by applying `f` to that Value. Let's write that:

```
instance functorState :: Functor (State s) where
  map f (State fx) = State \s -> f $ fx s   -- COMPILER ERROR!!
```

Here's the compiler error:



The compiler encountered a `Tuple a0 s1` but expected `a0`. And there's a red underline under `fx s`, which makes me think that `fx s` is a returning a `Tuple`. Can you see what we did wrong?

The problem is that `State's` Function has the Type `s -> Tuple a s`. So, we were right. `fx s` does returns a `Tuple`. This means that we cannot call `f` just yet. We're going to need to get inside that `Tuple`.

But we can't just `map` over the `Tuple` because it's defined in the wrong order:

```
Tuple a s ❶
```

❶ We can ONLY map over the second Parameter of a `Tuple`, but it's an `s` NOT an `a`.

We could fix this by reversing the Types and that's what I would do. But, there's a long history in Haskell for this Type and the developers of the PureScript library that implements `State` followed in Haskell's footsteps. So, we're going to do the same, which means we cannot use `map` on the `Tuple`.

We could use `lmap` from `Bifunctor` to map over it's left Type, but, instead, we're just going to destructure it in a Lambda Function:

```
instance functorState :: Functor (State s) where
  map f (State fx) = State \s -> fx s # \(Tuple x s') -> ??? ❶ ❷
```

❶ We've removed the call to `f` for now.

❷ `s'` is our new State since `fx` could have modified it.

Now `x` is of Type `a`, which means that we can finally apply `f` to it. Then we'll wrap up that result along with the new State, `s'`, in a `Tuple`:

```
instance functorState :: Functor (State s) where
  map f (State fx) = State \s -> fx s # \(Tuple x s') -> Tuple (f x) s'
```

Next up is the `Apply` Instance. We'll start in the usual way:

```
instance applyState :: Apply (State s) where
```

And now the left-hand side of `apply`:

```
instance applyState :: Apply (State s) where
  apply (State ff) (State fx) = ???
```

We've named `ff` and `fx` to remind us that the first Function will return a Function and the second will return a Value.

And since we're returning a `State`, we'll start the right-hand side with the usually pattern:

```
instance applyState :: Apply (State s) where
  apply (State ff) (State fx) = State \s -> ???
```

Now before we move forward, we should be wary after our experience with the `Functor` Instance. We're going to get `Tuples` back from any `State` Functions we call. So, we'll need to destructure them like we did in `map`.

But, the most important thing to keep in mind is that we MUST call `ff` BEFORE `fx` since `apply` is Left-Associative and `ff` may modify the `State`. We want to call `ff` first to make sure that `State` is modified in the proper order:

```
instance applyState :: Apply (State s) where
  apply (State ff) (State fx) = State \s -> ff s # \(Tuple g s') -> ???
```

Here `g` is the Function that `ff` returns as the first part of the `Tuple` and `s'` is the new State.

Now we'll call `fx` with the new State, `s'`:

```
instance applyState :: Apply (State s) where
  apply (State ff) (State fx) = State \s -> ff s # \(Tuple g s') -> fx s'
    # \(Tuple x s'') -> ???
```

Here x is the Value that fx returns and s'' is the new, new State.

The only thing left for us to do is apply g to x and wrap things up in a Tuple:

```
instance applyState :: Apply (State s) where
  apply (State ff) (State fx) = State \s -> ff s # \(Tuple g s') -> fx s'
    # \(Tuple x s'') -> Tuple (g x) s''
```

This is a nasty mess, but it works. We could've just used our old friend ap since we know State is going to be a Monad. But, we wouldn't learn as much if we did that.

One important fact to note about using ap is that the order in ap is properly done for Left-Associativity:

```
ap :: ∀ a b m. Monad m => m (a -> b) -> m a -> m b
ap mf mx = do
    f <- ff ❶
    x <- fx ❷
    pure $ f x
```

❶ bind is called on ff first.

❷ bind is called on fx second.

Using ap will keep us from accidentally switching the order of ff and fx in our handwritten implementations. If we make the mistake of switching their order, we won't get a compiler error. Instead, our program will have subtle bugs that we may not catch until our code is in production.

Now we are ready to write the Applicative Instance:

```
instance applicativeState :: Applicative (State s) where
```

And now the right-hand side:

```
instance applicativeState :: Applicative (State s) where
  pure x = ???
```

We'll add our familiar pattern that has yet to fail us:

```
instance applicativeState :: Applicative (State s) where
  pure x = State \s -> ???
```

And all we have to do is put x into our Tuple with NO change to the State, s:

```
instance applicativeState :: Applicative (State s) where
  pure x = State \s -> Tuple x s
```

Gotta love the Applicative implementation. It's usually pretty easy.

Now we'll write the Bind Instance:

```
instance bindState :: Bind (State s) where
```

And the right-hand side:

```
instance bindState :: Bind (State s) where
  bind (State fx) f = ???
```

We're destructuring to get access to fx, i.e. a Function that will return a Value.

Before we rush off to code up the left-hand side, we should keep in mind what trips us up every time we code a Bind Instance. Can you remember what that is?

It's the fact that f is a Monadic Function, not a Pure one like in map and apply. So we'll need to unwrap its output like we did in Reader. Do you remember how we did that?

We used runReader to unwrap the result of calling f in Reader's bind. And we're going to need to do the same with State.

At this point, we should write runState. We're going to use runReader as a template:

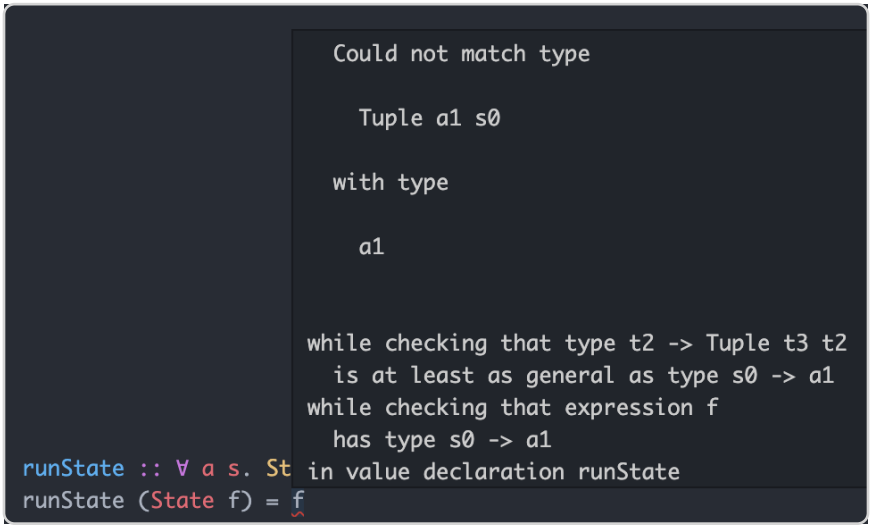
```
runReader :: ∀ a r. Reader r a -> (r -> a)
runReader (Reader f) = f
```

I don't see any reason runState can't just be exactly the same with a few of the names changed:

```
runState :: ∀ a s. State s a -> (s -> a)
runState (State f) = f -- COMPILER ERROR!!
```

Oops… I guess there are differences. But what?

Here's the compiler error:



Now, I see why these aren't the same. I forgot about the `Tuple`. That's an easy fix:

```
runState :: ∀ a s. State s a -> (s -> Tuple a s) ❶
runState (State f) = f
```

❶ Changed `s -> a` to `s -> Tuple a s`.

Now we can finish the code for `bind`. Here's where we left off:

```
instance bindState :: Bind (State s) where
  bind (State fx) f = ???
```

Let's start with our pattern on the right:

```
instance bindState :: Bind (State s) where
  bind (State fx) f = State \s -> ???
```

And now we'll extract `x` from `fx` by passing it the State, `s`:

```
instance bindState :: Bind (State s) where
  bind (State fx) f = State \s -> fx s ???
```

Then destructure the results, i.e. the `Tuple`:

```
instance bindState :: Bind (State s) where
  bind (State fx) f = State \s -> fx s # \(Tuple x s') -> ???
```

Now, we can call `f` on `x`, which is the whole point of `bind`:

```
instance bindState :: Bind (State s) where
  bind (State fx) f =
    State \s -> fx s # \(Tuple x s') -> f x -- COMPILER ERROR!!
```

We're not surprised by the compiler error because we remember that `f` is a Monadic Function, which means we'll need to call `runState` on `f x`. We also remember that we'll need to pass the new State, `s'`, to `runState` as its second Parameter:

```
instance bindState :: Bind (State s) where
  bind (State fx) f =
    State \s -> fx s # \(Tuple x s') -> runState (f x) s' ???
```

And now we destructure the `runState` result, which is a `Tuple`:

```
instance bindState :: Bind (State s) where
  bind (State fx) f =
    State \s -> fx s # \(Tuple x s') -> runState (f x) s'
      # \(Tuple y s'') -> ???
```

And now we can finish by putting it all back together:

```
instance bindState :: Bind (State s) where
  bind (State fx) f =
    State \s -> fx s # \(Tuple x s') -> runState (f x) s'
      # \(Tuple y s'') -> Tuple y s'' ❶
```

❶ What happened here?

Looking at the last Lambda, `\(Tuple y s'') -> Tuple y s''`, we can see that it's just `identity`.

Looks that I got a bit carried away here. We were done when we added `runState`. So, let's roll things back a bit:

```
instance bindState :: Bind (State s) where
  bind (State fx) f =
    State \s -> fx s # \(Tuple x s') -> runState (f x) s'
```

And last, but not least, we have the Monad Instance:

```
instance monadState :: Monad (State s)
```

Here's what we've written for State:

```
newtype State s a = State (s -> Tuple a s)

instance functorState :: Functor (State s) where
  map f (State fx) = State \s -> fx s # \(Tuple x s') -> Tuple (f x) s'

instance applyState :: Apply (State s) where
  apply (State ff) (State fx) = State \s -> ff s # \(Tuple g s') -> fx s'
    # \(Tuple x s'') -> Tuple (g x) s''

instance applicativeState :: Applicative (State s) where
  pure x = State \s -> Tuple x s

instance bindState :: Bind (State s) where
  bind (State fx) f =
    State \s -> fx s # \(Tuple x s') -> runState (f x) s'

instance monadState :: Monad (State s)
```

And, not surprisingly, there isn't much code here. A lot of thinking went into coding a little bit of code.

But we're not quite done yet. We still need to write the State API:

```
get :: ∀ s. State s s
put :: ∀ s. s -> State s Unit
modify :: ∀ s. (s -> s) -> State s s
modify_ :: ∀ s. (s -> s) -> State s Unit
```

Here get will retrieve the current State and put will replace the current State.

modify will modify the current State using a Function that takes the old State and returns the new State. modify also returns the new State, whereas, modify_ will do the same as modify but returns Unit instead.

In practice, modify_ is used more often since it doesn't require a void in a do block and we rarely need the new State back. But, when we do, modify will save us a call to get.

Let's start with get:

```
get :: ∀ s. State s s
get = ???
```

We need to return a State, so we'll use our pattern:

```
get :: ∀ s. State s s
get = State \s -> ???
```

And now, all we have to do is return the State UNCHANGED in both slots of the Tuple, since the resulting Pure Computation ***is*** the State:

```
get :: ∀ s. State s s
get = State \s -> Tuple s s
```

Here the first s of the Tuple is the Pure Computational result and the second s is the Monadic Computational result. This is just like ask in Reader.

Now, we'll code put:

```
put :: ∀ s. s -> State s Unit
put s = ???
```

This will be a bit tricker. put gets a new State, s, that's going to replace whatever State that is passed to it. So, we're going to start with the pattern we always use to return a State, but, in this case, we're going to throw away the Parameter, which we usually call s:

```
put :: ∀ s. s -> State s Unit
put s = State \_ -> ??? ❶
```

❶ _ is normally called s but we don't care about the value of the old State, since we're replacing it.

And now, we can return the new State, s, and unit wrapped up in a Tuple:

```
put :: ∀ s. s -> State s Unit
put s = State \_ -> Tuple unit s ❶
```

❶ unit is the Pure Computational result and s is the Monadic Computational result.

Can you think of why the return type is State s Unit? What else could we have used instead of Unit?

We could have used an Int, which means we'd returned Tuple 0 s instead of Tuple unit s. But there's no reason for the 0 or any Int Value for that matter.

Unit here works for 2 reasons. First, it's a ***don't-care*** Value and second, it has a Discard Instance, which means when we use put in a do block, we won't have to use void to discard the Value returned.

Now we'll write modify:

```
modify :: ∀ s. (s -> s) -> State s s
modify f = ???
```

And we need to return a State, so we'll add our pattern:

```
modify :: ∀ s. (s -> s) -> State s s
modify f = State \s -> ???
```

Remember that f is used to modify the State by applying it to s. So let's add that returning the new State as both the Pure Computation and the Monadic one:

```
modify :: ∀ s. (s -> s) -> State s s
modify f = State \s -> Tuple (f s) (f s)
```

Okay, so, I'm probably being a bit picky here, but I hate the duplication of code. You can stop here if you want but I want to go one step further:

```
modify :: ∀ s. (s -> s) -> State s s
modify f = State \s -> let ns = f s in Tuple ns ns ❶
```

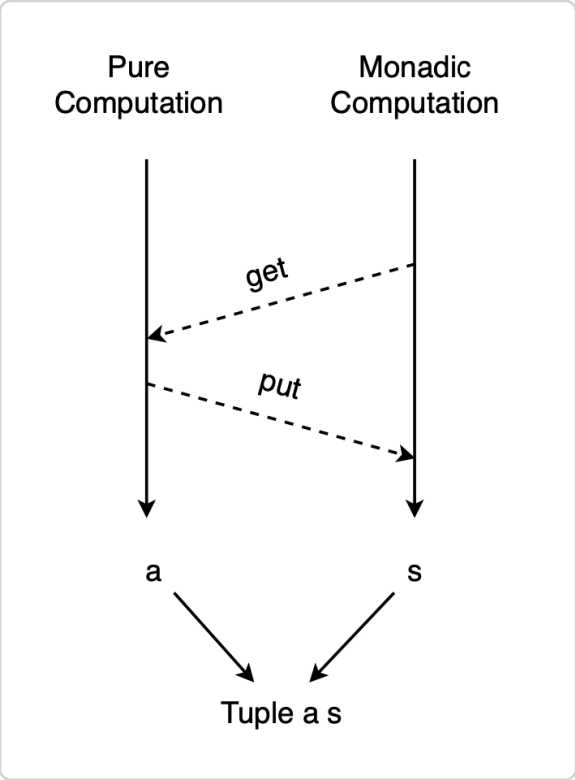
❶ ns is our new State.

Notice the use of the let expression to avoid duplication of code. I'm not sure if the PureScipt compiler would have optimized the duplication out in the previous step, so the safe bet is to not have it in the first place.

Now modify_ should be easy. We can copy modify's implementation making sure to change it to return unit as the Pure Computation:

```
modify_ :: ∀ s. (s -> s) -> State s Unit
modify_ f = State \s -> Tuple unit (f s)
```

Here's the parallel computation diagram for State:



In the diagram, get reads the current Monadic Computation's result and makes it available to the Pure Computation channel. put, on the other hand, takes a Value and replaces the Monadic Computation with it.

18.22. Using State as a Monadic Validation

Earlier we considered making Validation a Monad but ran into problems writing bind. But now that we have the State Monad, we should be able to use it to collect errors. So let's give it a try.

We'll start with our example before we starting using Validation example:

```
errIfMissing :: Maybe String -> String -> Either String String
errIfMissing Nothing err = Left err
errIfMissing (Just s) _  = Right s

fullName :: String -> String -> String -> String
fullName first middle last = first <> " " <> middle <> " " <> last

fullNameEither :: Maybe String -> Maybe String -> Maybe String
               -> Either String String
fullNameEither first middle last =
  fullName <$> (first `errIfMissing` "First name must exist")
           <*> (middle `errIfMissing` "Middle name must exist")
           <*> (last `errIfMissing` "Last name must exist")
```

Take a moment to review this.

And if you remember, we rewrote `fullNameEither` to use `do` notation:

```
fullNameEither :: Maybe String -> Maybe String -> Maybe String
                  -> Either String String
fullNameEither first middle last = do
  f <- errIfMissing first "First name must exist"
  m <- errIfMissing middle "Middle name must exist"
  l <- errIfMissing last "Last name must exist"
  pure $ fullName f m l
```

But both versions of `fullNameEither` stop at the first error since they're using `Either`, which Short-circuits. That's why we're going to use `State` instead of `Either`.

Let's rewrite `errIfMissing` so it uses `State` instead of `Either`. We'll start with its Type Signature:

```
errIfMissing :: Maybe String -> String -> State ???
```

Now we have to figure out what Type Parameters we're going to need for `State`. Let's pick `Array String` as our State, i.e. the first Type Parameter in the `State` Type Constructor. This way we can collect our errors in an `Array`:

```
errIfMissing :: Maybe String -> String -> State (Array String) ???
```

By looking at our `Either` implementation, we see that it returned a `String` in the success case. So let's try that for our Pure Computation result Type:

```
errIfMissing :: Maybe String -> String -> State (Array String) String
```

Now we're ready to code the Nothing case starting with the right-hand side:

```
errIfMissing :: Maybe String -> String -> State (Array String) String
errIfMissing Nothing err = ???
```

Now we need to add the error to the State.

```
errIfMissing :: Maybe String -> String -> State (Array String) String
errIfMissing Nothing err = do ❶
  modify_ (_ <> [err]) ❷
  ???
```

❶ We're using do notation.

❷ Appends the error, err, to the current State, i.e. an Array of errors. The _ in the wildcard Function is a placeholder for the current State.

Now, the big question here is what do we return as the Pure Computation in the error case. We don't have a String to return since we a have a Nothing.

But, we have to return some String for the Pure Computation. It really doesn't matter what that String is since we have errors, which means that we're not going to use the Pure Computational Value.

Since it doesn't matter, we might as well use the shortest String possible. So, let's use mempty since String is a Monoid.

```
errIfMissing :: Maybe String -> String -> State (Array String) String
errIfMissing Nothing err = do
  modify_ (_ <> [err])
  pure mempty
```

Notice how we use pure to put mempty into a Context, i.e. State. That's because the last line of a do block must return a Monadic Value, in this case, it returns a State.

Now for the Just case:

```
errIfMissing :: Maybe String -> String -> State (Array String) String
errIfMissing Nothing err = do
  modify_ (_ <> [err])
  pure mempty
errIfMissing (Just s) _ = ???
```

All we need to do is return the String that we've been given since this is NOT the error case:

```
errIfMissing :: Maybe String -> String -> State (Array String) String
errIfMissing Nothing err = do
  modify_ (_ <> [err])
  pure mempty
errIfMissing (Just s) _ = pure s ❶
```

❶ Once again, we put that String in a Context, i.e. State.

And now all we have to do is change fullNameEither. First, we'll change it's name since it'll return State. Second, we'll change it's return Type to match errIfMissing's return Type:

```
fullNameValid :: Maybe String -> Maybe String -> Maybe String ❶
                 -> State (Array String) String ❷
fullNameValid first middle last = do
  f <- errIfMissing first "First name must exist"
  m <- errIfMissing middle "Middle name must exist"
  l <- errIfMissing last "Last name must exist"
  pure $ fullName f m l
```

❶ Name changed.

❷ Returns a State instead of an Either.

Pretty nice that we didn't have to change much to fullNameValid when changing Monads, i.e. we went from Either to State.

Now, if we were to test this code, we could write something like:

```
test :: Effect Unit
test = do ❶
  log $ show $ runState (fullNameValid Nothing (Just "") Nothing) []
```

❶ This do is unnecessary since there's only 1 line in the do block. It can safely be removed. In Haskell, the linter will complain about this redundancy. Not so in PureScript. That's your job.

Running this test will display:

```
(Tuple "" ["First name must exist","Last name must exist"])
```

Notice that the Tuple contains both the Pure Computation, i.e. "", and the Monadic Computation, i.e. ["First name must exist","Last name must exist"].

If we were to use this approach, we'd want to check that the Monadic Computation, i.e. the Final State, is NOT []. That would indicate that there are errors.

Now that we've written this to work with State instead of Valiation, it's clear that we didn't need State. If we look at errIfMissing, we should be able to see how it's just appending using a Semigroup:

```
errIfMissing :: Maybe String -> String -> State (Array String) String
errIfMissing Nothing err = do
  modify_ (_ <> [err]) ❶
  pure mempty
errIfMissing (Just s) _ = pure s
```

❶ Here's where append is called.

But that's what Writer does. That means that we can rewrite this using Writer?

Let's try by first rewriting errIfMissing's definition by changing State to Writer:

```
errIfMissing :: Maybe String -> String -> Writer (Array String) String
```

And now the Nothing case:

```
errIfMissing :: Maybe String -> String -> Writer (Array String) String
errIfMissing Nothing err = do
  tell [err]
  pure mempty
```

The only thing that needed to change here was calling tell instead of modify_. We didn't have to change the pure mempty at all thanks to the way Typeclasses work. The pure being used now is Writer's instead of State's. The compiler takes care of that for us.

And now the Just case:

```
errIfMissing :: Maybe String -> String -> Writer (Array String) String
errIfMissing Nothing err = do
  tell [err]
  pure mempty
errIfMissing (Just s) _ = pure s
```

The success case didn't change at all, since the compiler will use Writer's pure instead of State's.

Now, all we have to do is change the Type Signature on fullNameValid. Since both State and Writer are Monads, all of our code inside the do block doesn't change. That's because the compiler will use Writer's Methods, i.e. bind and pure, instead of State's Methods:

```
fullNameValid :: Maybe String -> Maybe String -> Maybe String
                 -> Writer (Array String) String ❶
fullNameValid first middle last = do
  f <- errIfMissing first "First name must exist"
  m <- errIfMissing middle "Middle name must exist"
  l <- errIfMissing last "Last name must exist"
  pure $ fullName f m l
```

❶ State changed to Writer.

This implementation works equally well. All we have left to do is modify our test code, i.e. change runState to runWriter:

```
test :: Effect Unit
test = do
  log $ show $ runWriter
    (fullNameValid Nothing (Just "") Nothing) [] -- COMPILER ERROR!!
```

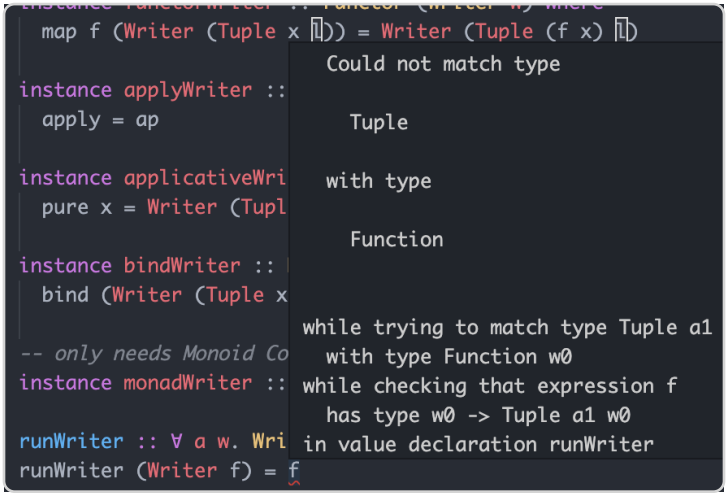
Oh, no. We never wrote runWriter. Let's quickly do that using runState as our template:

```
runState :: ∀ a s. State s a -> (s -> Tuple a s)
runState (State f) = f
```

Let's change State to Writer and we'll change s to w to match the Type Definition for Writer:

```
runWriter :: ∀ a w. Writer w a -> (w -> Tuple a w)
runWriter (Writer f) = f -- COMPILER ERROR!!
```

And, of course, a compiler error:



This tells us that the compiler encountered a `Tuple` but expected a `Function`. The first thing I ask myself when this happens is what's different between `State` and `Writer` since I copy-pasta'd that code.

And that difference is that `Writer` contains a `Tuple`, whereas `State` contains a Function that returns a `Tuple`. Let's fix that:

```
runWriter :: ∀ a w. Writer w a -> Tuple a w ❶
runWriter (Writer x) = x ❷
```

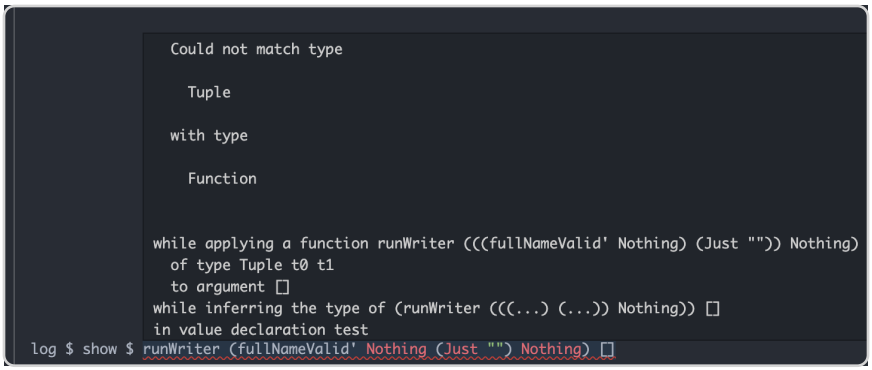
❶ Changed `(w -> Tuple a w)` to `Tuple a w`.

❷ Renamed `f` to `x` to reflect the fact that `Writer` contains a `Value` NOT a `Function` like `State` does.

Now we get a compiler error on this line of our test code:

```
test :: Effect Unit
test = do
  log $ show $ runWriter
    (fullNameValid Nothing (Just "") Nothing) [] -- COMPILER ERROR!!
```

Here's the compiler error:



Not sure if that error message if very helpful, but this is another copy-pasta from `State`'s test code. So, once again, I ask, what's different between `State` and `Writer`.

Well, the most important difference here is that `State` needs an Initial State, which is why we passed `[]` into `runState`. `Writer` starts with `mempty` as its initial Value. So, there's no need to pass `[]`, which happens to be the `mempty` Value for `Array a`.

Another way to look at this is to compare their Type Signatures. `runState` takes 2 Parameters, whereas `runWriter` takes only 1. Either way, we need to remove the `[]` Parameter:

```
test :: Effect Unit
test = do log $ show $ runWriter (fullNameValid Nothing (Just "") Nothing)
```

And now we can run this code, which will print:

```
(Tuple "" ["First name must exist","Last name must exist"])
```

This output is identical to our `State` implementation and it's far superior since `Writer` is simpler.

But stepping back and looking at the bigger picture, it's clear that there aren't any advantages gained by using `Writer` over `Validation`. Unless, for some odd reason, we need to use a `Monad`. In that rare situation, we now know how to do that. But, in practice, `Validation` should be preferable.

In general, I'd suggest starting with an `Applicative` before reaching for a `Monad` for the same reason we favored `Writer` over `State` earlier. And that reason is simplicity. Better to use the simplest tool that can get the job done.

18.23. The Kleisli Category

We talked earlier about ***Kleisli Composition*** and the ***Kleisli Category***. We saw how Kleisli Composition composes Monadic Functions, i.e. Functions of the form, `a -> m b`:

```
composeKleisli
  :: ∀ a b c m
  .  Monad m
  => (b -> m c)
  -> (a -> m b)
  -> (a -> m c)
composeKleisli g f x = f x >>= g

infixr 5 composeKleisli as >=>
```

The difference between Monadic Functions and Pure Functions can be seen by comparing `composeKleisli` with Pure Function Composition, `compose`. This is especially obvious when focusing on the `Function` Instance implementation for `Semigroupoid`:

```
class Semigroupoid a where
  compose :: ∀ b c d. a c d -> a b c -> a b d

infixr 9 compose as <<<

instance semigroupoidFn :: Semigroupoid (->) where
  compose :: ∀ b c d. (c -> d) -> (b -> c) -> (b -> d)
  compose f g x = f (g x)
```

We also talked about `Semigroupoid` is like a Category, but without the Identity requirement.

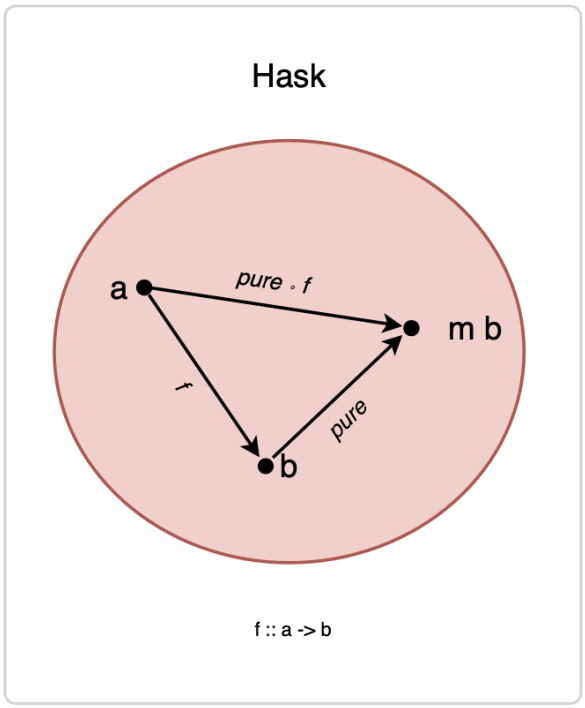
There is a Typeclass in PureScript called `Category`. It models Categories in Category Theory by adding the Identity Law to `Semigroupoid`:

```
class Semigroupoid a <= Category a where
  identity :: ∀ t. t -> t
```

Remember, that a Category has Objects, and Morphisms between those Objects. In the Category of `Hask`, the Objects are Types and the Morphisms are Functions.

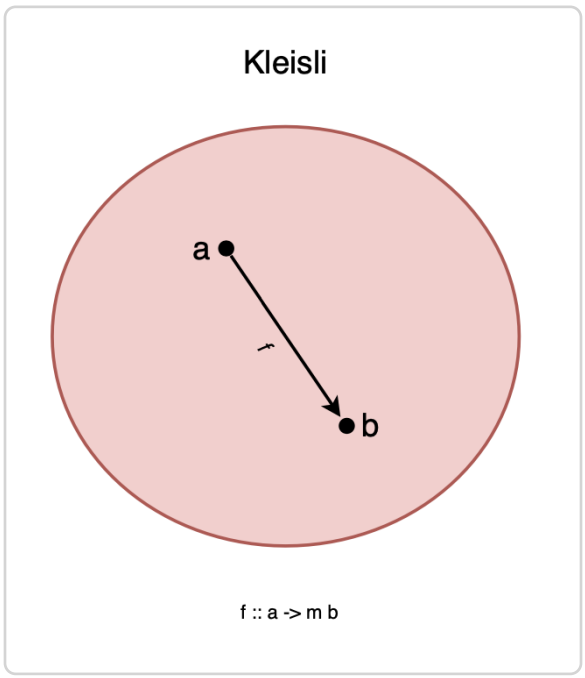
In the Kleisli Category, the Objects are also Types and but the Morphisms are Monadic Functions, i.e. Functions of the Type, `a -> m b`.

First, let's take a look at `Hask`:



Notice that Morphisms between Objects `a` and `b` are Pure Functions, i.e. `f :: a -> b`. We can compose Pure Functions as is shown in the diagram. Here we compose `pure` with `f` to product a Monadic Function, `pure <<< f :: a -> m b`.

Now let's look at `Kleisli`:



Unlike the Hask Category, in the Kleisli Category, a Morphism between `a` and `b` has the Type Signature of `a -> m b`. These Morphisms are formally known as ***Kleisli Arrows***.

We can show that Kleisli is a valid Cateogory if we can prove the Category Laws:

- Morphisms compose
- There is an Identity Morphism for every Object, i.e. 1_x: `x → x`
- Composition is Associative: if `f: a -> m b`, `g: b -> m c`, and `h: c -> m d` then `h ∘ (g ∘ f) = (h ∘ g) ∘ f`

Notice the last law is slightly modified from what we saw for `Hask` earlier in the book. Now Functions return `m b`, `m c` and `m d` instead of `b`, `c` and `d` respectively. That's because we're going to be using Kleisli Composition.

As a point of reference, we should also look at the `Semigroupoid` and `Category` Instances for `Function`:

```
instance semigroupoidFn :: Semigroupoid (->) where
  compose f g x = f (g x)

instance categoryFn :: Category (->) where
  identity x = x
```

We've proven that Functions compose by implementing `compose`. We also have shown that an Identity Morphism exists by creating `identity`. And we know that Function Composition in Functional Programming is Associative, so we know that the Associativity Law is upheld.

And so, PureScript Functions along with the Types they operate on form a Category known as `Hask`.

Let's do the same thing except for `Kleisli`.

To do this, we first need to model Kleisli Arrows. These are Morphisms in the Kleisli Category and are analogous to `Functions`.

To help us, we'll model Arrows in `Hask` first:

```
newtype Function a b = Function (a -> b)
```

This definition models the fact that an Arrow from Object `a` to Object `b` in the Hask Category is represented by the Type `Function a b`. The Data Constructor takes a Function with the Type `a -> b`.

And now, we'll do the analogous thing for Kleisli Arrows:

```
newtype Kleisli m a b = Kleisli (a -> m b)
```

This definition models the fact that a Kleisli Arrow from Object `a` to Object `b` in the Kleisli Category is represented by the Type `Kleisli m a b`. The Data Constructor takes a Function with the Type Signature `a -> m b`.

Please take the time to recognize the similarities before moving on.

Next, we'll make the `Semigroupoid` Instance for `Kleisli`, one line at a time:

```
instance semigroupoidKleisli :: Monad m => Semigroupoid (Kleisli m) where
```

Here `m` is constrained to be a `Monad` since it represents the Monad for the Kleisli Arrows. It's the `m` in the Monadic Function Type Signature, i.e. `a -> m b`.

We have to hold `m` constant since the Type Parameter to `Semigroupoid` has a Kind Signature of `Type -> Type`, which can be seen by looking at the Typeclass definition for `Semigroupoid`:

```
class Semigroupoid a where
  compose :: ∀ b c d. a c d -> a b c -> a b d ❶
```

❶ `a's` Kind is `Type -> Type`. We can see this because `a` takes 2 Type Parameters, e.g. `a c d`.

Now we can write the left-hand side of the `Semigroupoid` Instance for `Kleisli`:

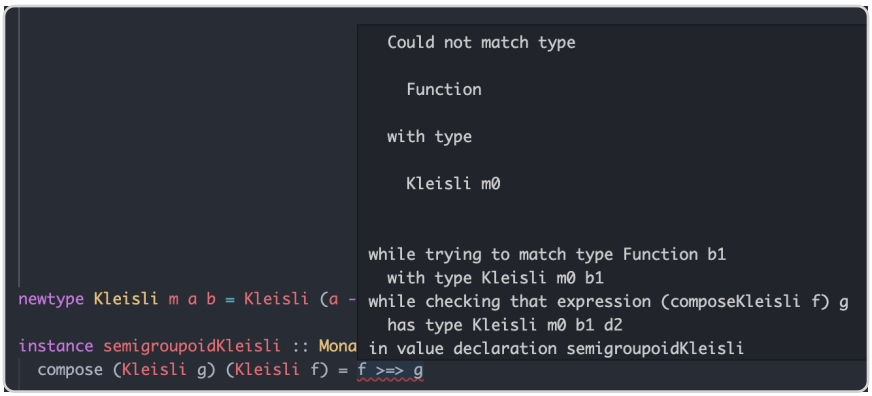
```
instance semigroupoidKleisli :: Monad m => Semigroupoid (Kleisli m) where
  compose (Kleisli g) (Kleisli f) = ??? ❶
```

❶ Remember, compose is `g` AFTER `f`. This is why the Function names are, from left to right, `g` and then `f` and not the other way around.

And now we just need to compose the Monadic Functions use `composeKleisli`, i.e. `>=>`:

```
instance semigroupoidKleisli :: Monad m => Semigroupoid (Kleisli m) where
  compose (Kleisli g) (Kleisli f) = f >=> g -- COMPILER ERROR!!
```

Here's the compiler error:



This says that the compiler encountered a `Function` when it expected `Kleisli m0`. Can you see why?

Well, `f >==> g` has the Type, `a -> m b`. So that's the `Function` it found. And what it wants is a `Kleisli m`.

If it's still not clear what to do, we can write the Type Signature for `compose` specialized for `Kleisli`:

```
instance semigroupoidKleisli :: Monad m => Semigroupoid (Kleisli m) where
  compose
    :: ∀ b c d
    .  (Kleisli m) c d
    -> (Kleisli m) b c
    -> (Kleisli m) b d ❶
  compose (Kleisli g) (Kleisli f) = f >==> g -- COMPILER ERROR!!
```

❶ `a` has been replaced with `(Kleisli m)`.

We can see that `compose`, should return a `Kleisli`. With this new information, it should be obvious that we can fix our code by simply using the `Kleisli` Data Constructor to wrap the `Function`:

```
instance semigroupoidKleisli :: Monad m => Semigroupoid (Kleisli m) where
  compose (Kleisli g) (Kleisli f) = Kleisli $ f >==> g ❶
```

❶ The Type Signature specialized for `Kleisli` was removed.

So, now that this compiles, it means that we have a valid `Semigroupoid`.

We're going to handwave the Associativity proof for `compose`, since we've already proven that `composeKleisli` is Associative thanks the Associativity Law for `bind`. Remember, `>=>` is implemented in terms of `>>=`. So as long as our `Monad`, `m`, is a law abiding Monad, Kleisli Composition is Associative.

The last step to show is that `Kleisli` is a Category. We'll do this by writing its `Category` Instance. We'll do so one line at a time:

```
instance categoryKleisli :: Monad m => Category (Kleisli m) where ❶
```

❶ We need the `Monad` Constraint since `Semigroupoid` has a `Monad` Constraint and it's a Superclass of `Category`.

We have to hold `m` constant here too, since the Type Parameter to `Category` has a Kind Signature of `Type -> Type`, which can be seen by looking at the Typeclass definition:

```
class Semigroupoid a <= Category a where ❶
  identity :: ∀ t. t -> t
```

❶ `a` is the SAME Type as it is in `Semigroupoid` whose Kind is `Type -> Type`.

Since we had trouble last time, we're going to first write the `identity` Type Signature specialized for `Kleisli`. Hopefully, it'll help us write the code:

```
instance categoryKleisli :: Monad m => Category (Kleisli m) where
  identity :: ???
```

Wow. I didn't expect this part to be so hard. I'm not sure what to put here. Do you know?

To figure this out, we'll start with an easier problem that's just like this one. We're going to write the Type Signature for Pure Functions. We'll start with the code:

```
instance categoryFn :: Category (->) where
  identity x = x
```

And now, we'll add the Type Signature:

```
instance categoryFn :: Category (->) where
  identity :: ∀ a. a -> a
  identity x = x
```

That was much easier. What we'd like to do is use this as a template for the Type Signature for our `Kleisli` version. Unfortunately, `a -> a` doesn't look much like our `Kleisli` Type.

So, let's rewrite the Type Signature using Prefixed Notation:

```
instance categoryFn :: Category (->) where
  identity :: ∀ a. (->) a a ❶
  identity x = x
```

❶ `->` is now in prefixed position.

And then let's rename `->` to `Function`:

```
instance categoryFn :: Category (->) where
  identity :: ∀ a. Function a a ❶
  identity x = x
```

❶ We replaced `(->)` with `Function`.

That's more like it. We have `Function a a`. This looks much closer to our `Kleisli` Type.

It should be easy now to see that we just need to replace `Function` with `Klesli m` and we'll get the Type Signature we need for our `Kleisli` version for `identity`:

```
instance categoryKleisli :: Monad m => Category (Kleisli m) where
  identity :: ∀ a. (Kleisli m) a a ❶
  identity = ??? ❷
```

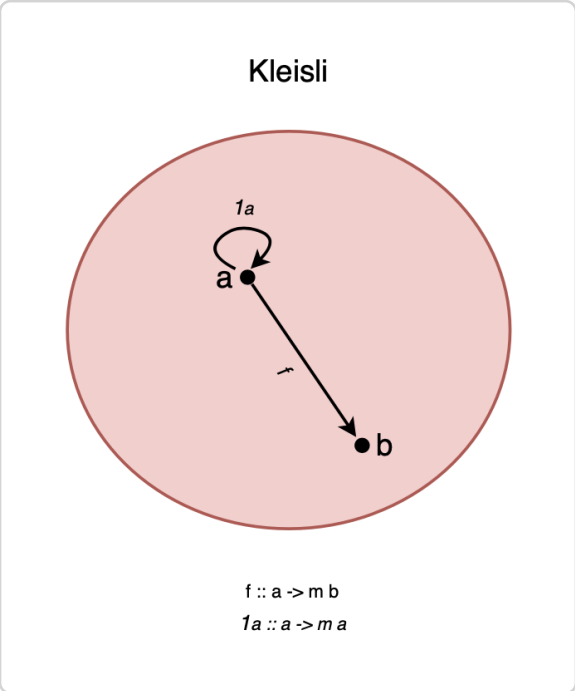
❶ We replaced `Function` with `(Kleisli m)`. The Parentheses are redundant and are only there to help us see how it parallels the Pure Function version we wrote earlier.

❷ Notice that `identity` is NOT a Function.

What's really important, at this point, is to realize that `identity` for `Hask` is an Arrow in `Hask`, and Arrows in `Hask` are Functions. Therefore, `identity` is a Function. And since Functions are built into the Language, we simply wrote `identity` as a Function.

However, `identity` for `Kleisli` is a Kleisli Arrow ,which is a Monadic Function. There is NO built-in mechanism for Kleisli Arrows. This is why `identity` is `Kleisli m a a` instead of some sort of Function.

Before we try to finish the righthand side of `identity`, we should take a look at the Kleisli Category diagram with an Identity Arrow added:



In this diagram, we can see that the Arrow from `a` to `b`, is a Function, `f :: a -> m b`.

We can also see that the Identity Arrow on `a` is 1_a `:: a -> m a`. This Type Signature should look familiar since it's the Type Signature for `pure`:

```
class Apply f <= Applicative f where
  pure :: ∀ a. a -> f a
```

So, our `identity` implementation for `Kleisli` will probably use `pure` in some way.

Let's return to our code:

```
instance categoryKleisli :: Monad m => Category (Kleisli m) where
  identity :: ∀ a. Kleisli m a a
  identity = ???
```

The next step is to construct a `Kleisli`, so we'll use the `Kleisli` Data Constructor to accomplish that:

```
instance categoryKleisli :: Monad m => Category (Kleisli m) where
  identity :: ∀ a. Kleisli m a a
  identity = Kleisli ???
```

The Data Constructor needs a Function as its Parameter. And since `identity` is a Kleisli Arrow from `a` to `a`, the Function will have a Type Signature of `a -> m a`, which we know is just the Function `pure`.

So we'll pass `pure` to the Data Constructor:

```
instance categoryKleisli :: Monad m => Category (Kleisli m) where
  identity :: ∀ a. Kleisli m a a
  identity = Kleisli pure ❶
```

❶ The fact that `pure` in `identity` shouldn't be too surprising when you look back at the Left and Right Identity Law in the Monad Laws.

We have now written the identity for the Kleisli Category. But does it behave like the other `identity`, i.e. the one from the Hask Category.

Well, here's the Identity Laws:

```
identity <<< h = h    [Left Identity]
h <<< identity = h    [Right Identity]
```

We just have to show that this works for `Kleisli` like it works for `Function`.

But one important detail here is that while `h` is a Function in PureScript, it's also an Arrow in some Category. In the Hask Category, it's also a Function. But in the Klesli Category, it's a Kleisli Arrow.

This means that `h` has the Type Signature of `Kleisli m a b` when we're looking at the Identity Laws for the Kleisli Category.

We can construct a `Kleisli m a b` by using the `Kleisli` Data Constructor and passing it the Function, `f :: a -> m b`. This would be represented by `Kleisli f`.

So let's specialize the Identity Laws for `Kleisli` by replacing `h` with `Kleisli f`:

```
identity <<< Kleisli f = Kleisli f    [Left Identity]
Kleisli f <<< identity = Kleisli f    [Right Identity]
```

We're going to need to refer to the `Kleisli` Instances:

```
instance semigroupoidKleisli :: Monad m => Semigroupoid (Kleisli m) where
  compose (Kleisli g) (Kleisli f) = Kleisli $ f >=> g ❶

instance categoryKleisli :: Monad m => Category (Kleisli m) where
  identity = Kleisli pure ❷
```

And we're also going to need to refer to `composeKleisli` written infixed:

```
f >=> g = \x -> f x >>= g ❸
```

And finally, we're going to need the Identity Laws for Bind:

```
g x >>= pure = g x                    [Left Identity (Bind)]
pure x >>= f = f x                    [Right Identity (Bind)]
```

We're now ready to prove Left Identity:

```
identity <<< Kleisli f       = Kleisli f -- Left Identity
Kleisli pure <<< Kleisli f   = Kleisli f -- Substituted using ❷
Kleisli $ f >=> pure         = Kleisli f -- Substituted using ❶
Kleisli \x -> f x >>= pure   = Kleisli f -- Substitute using ❸
Kleisli \x -> f x            = Kleisli f -- Left Identity (Bind)
Kleisli f                    = Kleisli f -- ETA reduction
```

And now, the Right Identity:

```
Kleisli f <<< identity       = Kleisli f -- Left Identity
Kleisli f <<< Kleisli pure   = Kleisli f -- Substituted using ❷
Kleisli $ pure >=> f         = Kleisli f -- Substituted using ❶
Kleisli $ \x -> pure x >>= f = Kleisli f -- Substitute using ❸
Kleisli \x -> f x            = Kleisli f -- Right Identity (Bind)
Kleisli f                    = Kleisli f -- ETA reduction
```

So we've proven that our `identity` for `Kleisli` works just like it does for `Function`, i.e. it's law abiding.

This barely scratches the surface of the Kleisli Category and we were fast and loose without our "proofs", but hopefully this has given you a tiny glimpse as to where the names and concepts come from when you encounter them in the PureScript or Haskell.

Chapter 19. Coding Monads

Return to the project that you've been coding in and create a file called `Ch19.purs` and add the following:

```
module Ch19 where

import Prelude ❶

import Effect (Effect)
import Effect.Console (log)

test :: Effect Unit
test = do
  log "placeholder" ❷
```

❶ Once again we are not writing anything from Prelude so we'll just import all of it.

❷ We'll delete this soon enough. It's just here to keep the compiler happy.

Next, change your `Main` module to import `Ch19` and call its `test` Function:

```
module Main where

import Prelude
import Effect (Effect)

import Ch19 as Ch19

main :: Effect Unit
main = Ch19.test
```

If you run:

```
npx spago run
```

you should see it build and print:

```
[info] Build succeeded.
placeholder
```

19.1. Writing Monad Instance for Maybe

We're going to start simple by making Maybe a Monad. Here's a starting point for that process:

```
import Data.Generic.Rep (class Generic)
import Data.Show.Generic (genericShow)

data Maybe a = Nothing | Just a
derive instance genericMaybe :: Generic (Maybe a) _

instance showMaybe :: Show a => Show (Maybe a) where
  show = genericShow

instance functorMaybe :: Functor Maybe where
  map f (Just x) = Just $ f x
  map _ Nothing  = Nothing
```

Now write the following:

- Apply Instance (for Review)
- Applicative Instance (for Review)
- Bind Instance
- Monad Instance

The Apply and Applicative are for review so please don't cheat yourself by copying the code we wrote earlier.

Here's the test code that you can now replace the placeholder test code with:

```
log $ show $ Just (_ * 10) <*> Just 20 ❶
log $ show $ Just (_ * 10) <*> pure 20 ❷
log $ show $ Just 20 >>= pure <<< (_ * 10) ❸
log $ show do ❹
  x <- Just 20
  let y = x * 10
  pure y
log $ show $ Just 20 >>= const Nothing >>= \y -> Just $ y + 42 ❺
log $ show do ❻
  _ <- Just 20
  y <- Nothing
  pure $ y + 42
```

❶ Apply version that prints (Just 200).

❷ The Applicative version of the previous line. This uses pure instead of Just.

❸ The Bind version of the previous line. Note that pure <<< (_ * 10) is Point-free.

❹ The do version of previous line. Note there is NO in keyword on the let.

❺ Bind version that prints Nothing. Notice that this Short-circuits.

❻ The do version of the previous line.

Please take a moment to look at the different ways the test code was written. There are many ways to skin a cat and so it's good to see the different ways so you'll recognize them when look look at real code.

Then code up the Instances for Maybe before reading on.

19.2. Code for Monad Instance for Maybe

Starting with Apply's Instance definition:

```
instance applyMaybe :: Apply Maybe where
```

Then the failure case:

```
instance applyMaybe :: Apply Maybe where
  apply Nothing  _ = Nothing
```

We Short-circuit the Computation once we encounter a Nothing since there's no Function to work with. We have no choice but Short-circuit.

And finally the success case:

```
instance applyMaybe :: Apply Maybe where
  apply Nothing  _  = Nothing
  apply (Just f) x  = f <$> x
```

We unwrap f and then map it over x, which is still in Context, i.e. it's a Maybe.

Next up is Applicative, starting with the definition:

```
instance applicativeMaybe :: Applicative Maybe where
```

And now we're going to use Just to wrap Pure Values:

```
instance applicativeMaybe :: Applicative Maybe where
  pure = Just
```

Remember, if we used `Nothing` instead of `Just`, we'd break the Applicative Identity Law:

```
 pure identity <*> v = v      [Identity]
```

If `pure` was `Nothing`, it would Short-circuit and result in `Nothing` on the left-hand side, which is not equal to `v`.

And that's it for our review for `Maybe`. Now we'll move on to `Bind`:

```
instance bindMaybe :: Bind Maybe where
```

And now for the failure case:

```
instance bindMaybe :: Bind Maybe where
  bind Nothing _  = Nothing
```

Once again, the behavior here is consistent, since it Short-circuits when `Nothing` is encountered.

And now for the success case:

```
instance bindMaybe :: Bind Maybe where
  bind Nothing _  = Nothing
  bind (Just x) f = f x
```

Notice that we don't have to wrap `f x` in `Just`. This is because `f` is a Monadic Function that will return a `Maybe`.

And finally, the easiest instance to code, `Monad`:

```
instance monadMaybe :: Monad Maybe
```

Hopefully, that was pretty simple. If not, please go back and review the code we've written before you tackle `Either`.

Run your code to see how it performs.

19.3. Writing Monad Instance for Either

Now we're going to make Either a Monad. Here's a starting point for that process:

```
data Either a b = Left a | Right b
derive instance functorEither :: Functor (Either a) ❶
derive instance genericEither :: Generic (Either a b) _

instance showEither :: (Show a, Show b) => Show (Either a b) where
  show = genericShow
```

❶ Unlike Maybe, we're letting the compiler do the dirty work of writing the Functor Instance.

Now write the following:

- Apply Instance (for Review)
- Applicative Instance (for Review)
- Bind Instance
- Monad Instance

Remember that Apply and Applicative are for review so try to write these without cheating.

Here's the test code:

```
log $ show $ Right (_ * 10) <*> (Right 20 :: Either Unit _) ❶ ❷
log $ show $ Right (_ * 10) <*> (pure 20 :: Either Unit _) ❸
log $ show $ (Right 20 :: Either Unit _) >>= pure <<< (_ * 10) ❹ ❺
log $ show do ❻
  x <- Right 20 :: Either Unit _
  let y = x * 10
  pure y
log $ show
  $ Right 20 >>= const (Left "error") >>= \y -> Right $ y + 42 ❼
log $ show do ❽
  _ <- Right 20
  y <- Left "error"
  pure $ y + 42
```

❶ Apply version that prints (Right 200).

❷ Had to explicitly specify a Type for the Left. That Type has to be showable, which is why it's Unit.

❸ The Applicative version of the previous line. This uses pure instead of Right.

❹ The Bind version of the previous line. Note that pure <<< (_ * 10) is Point-free.

❺ Had to explicitly specify a Type for the `Left` that is showable, i.e. `Unit`.

❻ The `do` version of previous line. Note there is NO `in` keyword on the `let`.

❼ `Bind` version that prints `Left "error"`. Notice that this Short-circuits.

❽ The `do` version of the previous line.

Next code up the Instances for `Either` before reading on.

19.4. Code for `Monad` Instance for `Either`

Starting with `Apply's` Instance definition:

```
instance applyEither :: Apply (Either a) where
```

Notice that we're keeping the `a` constant since the Kind for `Apply's` Type Parameter is `Type -> Type`.

Then the failure case:

```
instance applyEither :: Apply (Either a) where
  apply (Left x)  _  = Left x
```

We Short-circuit the Computation once we encounter a `Left` as `Either` was explicitly designed to do. This maintains the Semantics of `Either`.

And finally the success case:

```
instance applyEither :: Apply (Either a) where
  apply (Left x)  _  = Left x
  apply (Right f) x = f <$> x
```

We unwrap `f` and then map it over `x`, which is still in Context, i.e. it's an `Either a`.

Next up is `Applicative`, starting with the definition:

```
instance applicativeEither :: Applicative (Either a) where
```

Notice that we're keeping the `a` constant again.

And now we're going to use `Right` to wrap Pure Values:

```
instance applicativeEither :: Applicative (Either a) where
  pure = Right
```

Remember, if we used Left instead of Right, we'd break the Applicative Identity Law:

```
 pure identity <*> v = v      [Identity]
```

If pure was Left, it would Short-circuit and result in Left on the left-hand side, which is not equal to v.

And that's it for our review for Either. Now we'll move on to Bind:

```
instance bindEither :: Bind (Either a) where
```

Once again we hold a constant.

And now for the failure case:

```
instance bindEither :: Bind (Either a) where
  bind (Left x) _ = Left x
```

Once again, the behavior here is consistent, since it Short-circuits when Left is encountered.

And now for the success case:

```
instance bindEither :: Bind (Either a) where
  bind (Left x) _ = Left x
  bind (Right y) f = f y
```

Notice that we don't have wrap f y in Right since f is a Monadic Function that will return an Either.

And finally, the easiest instance to code, Monad:

```
instance monadEither :: Monad (Either a)
```

The hardest part here was remembering to hold a constant. It can be difficult to know to do this since Monad has no Typeclass Methods to consult. With Bind, we can look at the Type Signature for bind:

```
class Apply m <= Bind m where
  bind :: ∀ a b. m a -> (a -> m b) -> m b
```

We can see that `m` takes 1 Type Parameter, e.g. `m a`.

Unfortunately, there are no methods to consult for `Monad`:

```
class (Applicative m, Bind m) <= Monad m
```

But, we can see that `Monad` takes a Type Parameter `m` and that Type Parameter is the same `m` as it is for `Applicative` and `Bind`, which both have Kind Signatures of `Type -> Type`.

So, this why we must hold `a` constant the way we did for `Applicative` and `Bind`.

19.5. Monadic Parsers

We're going to return to our Applicative Parser and the code we wrote in the `Parser.purs` file. And we're going to upgrade our `Parser` and make it a `Monad`. Then we'll be able to write much more powerful `Parsers`.

But before we do, change your `Main` module to:

```
module Main where

import Prelude
import Effect (Effect)

import Parser as P

main :: Effect Unit
main = P.test
```

Before we leap into writing new `Parser` Instances , let's review the ones we've already written. Here's the `Functor` Instance:

```
instance functorParser :: Functor (Parser e) where ❶
  map :: ∀ a b. (a -> b) -> Parser e a -> Parser e b ❷
  map f p = Parser \s -> map f <$> parse p s ❸
```

❶ We held `e`, our error Type constant.

❷ `map's` Type Signature specialized for `Parser`.

❸ The mapping Function, `f`, isn't applied until the `Parser's` Function is called.

We run the `Parser`, `p`, and then map `map f` over the result. The reason for ***double mapping*** is because we have a `Tuple` inside of an `Either`. We have to make 2 hops, if you will.

Also, remember that `Functors` of Functions don't apply their mapping Functions until the Function is called. This is in stark contrast to `Functors` of Values, which immediately apply the mapping Function, e.g. `Maybe` and `Either` do this.

Here's our `Apply` Instance:

```
instance applyParser :: Apply (Parser e) where ❶
  apply :: ∀ a b. Parser e (a -> b) -> Parser e a -> Parser e b ❷
  apply p1 p2 = Parser \s -> case parse p1 s of
    Left err -> Left err
    Right (Tuple s1 h) -> case parse p2 s1 of
      Left err -> Left err
      Right (Tuple s2 x) -> Right $ Tuple s2 (h x)
```

❶ We held `e`, our error Type constant.

❷ `apply's` Type Signature specialized for `Parser`.

This code looks even more complex than when we originally wrote it. That's because we now know that this can be simplified, but we'll get to that soon enough. For now, let's try to refresh our memories on what this does.

We first run `Parser`, `p1`, i.e. the LEFTMOST `Parser`. It's very important that we run these in the right order since `apply` is Left-Associative.

Then we check `p1's` results for errors and if it was successful, then, and only then, do we run the second `Parser`, `p2` with the remaining `String`, `s1`.

We then check `p2's` run for errors and if it was successful, we return its remaining `String`, `s2`, and the result of applying `p1's` returned Function, `h`, to `p2's` returned Value, `x`. This step is the key step, since it's the whole point of `apply`, to take a Function from a Context and apply it to a Value in a Context.

Now, in `Parser.purs`, add the `Bind` and `Monad` instances for `Parser`. One up front hint, remember that the `f` in `bind` is a Monadic Function, i.e. `a -> m b`. Give it try before reading on.

First, as always, we'll start with the instance definition:

```
instance bindParser :: Bind (Parser e) where
```

Once again, we hold `e` constant. This is because `Parser` has Kind `Type -> Type -> Type`, which can be seen by it's definition:

```
newtype Parser e a = Parser (ParseFunction e a) ❶
```

❶ Takes a Type, `e`, and another Type, `a`, and returns a Type `Parser e a`.

Notice how the Type Constructor, i.e. the righthand `Parser`, takes 2 Type Parameters. That's why it has the aforementioned Kind Signature.

Next, we'll add the left-hand side of the implementation:

```
instance bindParser :: Bind (Parser e) where
  bind p f = ???
```

We don't destructure the `Parser`, `p`, here since we've been using `parse` to run the `Parsers` (see `map` and `apply` implementations).

Now we'll add our favorite pattern regarding `Functors` that contain Functions:

```
instance bindParser :: Bind (Parser e) where
  bind p f = Parser \s -> ???
```

And now, we're going to do something we haven't done before. We're going to work in the `Either` Monad. We've made this choice since `parse p s` returns an `Either`.

To remind us of this fact, we'll need the following for reference:

```
type ParseFunction e a =
  ParserError e => String -> Either e (ParserState a)

newtype Parser e a = Parser (ParseFunction e a)

parse :: ∀ e a. ParserError e => Parser e a -> ParseFunction e a
parse (Parser f) = f
```

And, now we'll write the Types involved in running a `Parser` with `parse p s`:

```
parse :: Parser e a -> ParseFunction e a
parse p :: ParseFunction e a
parse p :: String -> Either e (ParserState a) ❶
parse p s :: Either e (ParserState a)
```

❶ We expanded `ParseFunction e a`.

Since the act of calling our `Parser` returns `Either`, we can work in the `Either` Monad. But what exactly does that mean? That means that we're going to create a `do` block where every computation results in an `Either`.

We start by first using the `do` keyword:

```
instance bindParser :: Bind (Parser e) where
  bind p f = Parser \s -> do
    ???
```

The first step is to call the `Parser`, `p`:

```
instance bindParser :: Bind (Parser e) where
  bind p f = Parser \s -> do
    ??? <- parse p s
```

Now we just need to capture what it returns. But, remember, when we code in the Either Monad, we code JUST the Happy Path. So, if `parse p s` SUCCEEDS, then it returns a Value of Type `ParserState a`, which is `Tuple String a`:

```
type ParserState a = Tuple String a
```

So, let's add that to our `bind` implementation:

```
instance bindParser :: Bind (Parser e) where
  bind p f = Parser \s -> do
    Tuple s1 x <- parse p s
    ???
```

Now we can finally apply `f` to `x`, but before we do, we should ask ourselves what Type is `f x`. Let's write out the Types:

```
x :: a
f :: a -> Parser b
f x :: Parser b
```

So `f x` is another Parser. We need to run that Parser and we can do that with `parse`:

```
instance bindParser :: Bind (Parser e) where
  bind p f = Parser \s -> do
    Tuple s1 x <- parse p s ❶
    parse (f x) s1 ❶
```

❶ Each computation in the `do` block is an `Either`. That's what it means to work in the `Either` Monad.

And since `parse` returns an `Either`, it's the final Value of our `do` block, which in the `Either` Monad must be an `Either`.

Stepping back, we can say that `bind` returns a `Parser` that contains a Function that takes a `String`, `s`, and returns an `Either`. Keep in mind, that the whole `do` block evaluates to an `Either`.

Notice how much nicer this code looks than our `apply` implementation where we had to explicitly add `case` expressions. We don't pollute our code with the `if` logic since it's already in the `bind` implementation for `Either`:

```
instance bindEither :: Bind (Either a) where
  bind (Left x) _ = Left x ❶
  bind (Right y) f = f y
```

❶ The compiler generates the `if` logic when it does Pattern Matching to determine which version of `bind` to call.

We're leveraging the `if` logic in the `bind` implementation for `Either` by coding in the `Either` Monad allowing us to code just the Happy Path. `bind` effectively factors out the error logic.

Just for a point of comparison, here's `bind` written using `do` and using the `bind` Operator, `>>=` from `Either`:

```
instance bindParser :: Bind (Parser e) where
  bind p f = Parser \s -> do
    Tuple s1 x <- parse p s
    parse (f x) s1

instance bindParser :: Bind (Parser e) where
  bind p f = Parser \s -> parse p s >>= \(Tuple s1 x) -> parse (f x) s1 ❶
```

❶ While this is equivalent, the `do` syntax is far easier to read.

Now, that we've seen how to code in the `Either` Monad using the `do` syntax, change your `apply` implementation for `Parser` to operate in the `Either` Monad. Use what we did in `do` version of `bind` as a simple example to work from.

I'd suggest commenting out the code you have so you can use it for reference. Also, remember we do NOT

have to check for errors. The `Either` Monad does that for us in the `bind` implementation.

Don't read any further until you're finished coding.

Let's use the current implementation as our reference and note what each step does:

```
instance applyParser :: Apply (Parser e) where
  apply p1 p2 = Parser \s -> case parse p1 s of ❶
    Left err -> Left err ❷
    Right (Tuple s1 h) -> case parse p2 s1 of ❸
      Left err -> Left err ❹
      Right (Tuple s2 x) -> Right $ Tuple s2 (h x) ❺
```

❶ Execute first `Parser`, `p1`.

❷ Check for errors.

❸ Execute second `Parser`, `p2`.

❹ Check for errors.

❺ Apply the returned Function, from `p1`, to the returned Value, from `p2`, and wrap everything up in a `Tuple` in an `Either`.

Let's start with this:

```
instance applyParser :: Apply (Parser e) where
  apply p1 p2 = Parser \s -> do
    ???
```

`apply` returns a `Parser`, which contains a Function that takes a `String`, `s` and returns an `Either`, represented by the `do` block.

The first thing we did in our old code was to execute `Parser`, `p1`. Let's add that:

```
instance applyParser :: Apply (Parser e) where
  apply p1 p2 = Parser \s -> do
    ??? <- parse p1 s
```

We just need to capture what it returns. But, remember, when we code in the Either Monad, we code JUST the Happy Path. So, if `parse p s` SUCCEEDS, then it returns a Value of Type `ParserState a`, which is `Tuple String a`. This is the `Right` side of the `Either` since we're only concerned with the Happy Path.

Let's add that:

```
instance applyParser :: Apply (Parser e) where
  apply p1 p2 = Parser \s -> do
    Tuple s1 f <- parse p1 s ❶
    ???
```

❶ `f` represents the Function that's returned from the Parser. Remember, the first `Parser` in an `Apply` returns a Function and the second `Parser` returns a Value.

The next thing we did in our old code was to check for errors. We do NOT need to do that since it's handled by the `Either` Monad.

After that, we executed `p2`. Let's add that and capture its successful results:

```
instance applyParser :: Apply (Parser e) where
  apply p1 p2 = Parser \s -> do
    Tuple s1 f <- parse p1 s
    Tuple s2 x <- parse p2 s1 ❶
    ???
```

❶ `x` represents the Value that's returned from the Parser. Remember, the first `Parser` in an `Apply` returns a Function and the second `Parser` returns a Value.

The next thing our old code did was to check for errors from the second `Parser`, but, once again, we don't have to check for errors in the `Either` Monad. That's `bind's` job. And, even though the `do` syntax obscures the use of `bind`, it is being used between each line of code in the `do` block.

And the final thing our old code did was to apply the returned Function, from `p1`, to the returned Value, from `p2`, and wrap everything up in a `Tuple` in an `Either`:

```
instance applyParser :: Apply (Parser e) where
  apply p1 p2 = Parser \s -> do
    Tuple s1 f <- parse p1 s
    Tuple s2 x <- parse p2 s1
    pure $ Tuple s2 $ f x ❶
```

❶ You may still not be used to this syntax, i.e. all the `$'s`. It's used quite often in idiomatic PureScript and Haskell. It's equivalent to `pure $ Tuple s2 (f x)`.

We use `pure` here instead of `Right`. Technically, you can use `Right` but by using `pure`, we can swap out Monads, if, for example, `parse` changes to return something other than `Either`, and our code will still work. This is because the compiler will use the new Monad's implementation for `pure`.

We saw this when we changed `fullNameEither`, which we eventually renamed to `fullNameValid`. It went from returning an `Either` to returning a `State` and later a `Writer`. The final line in the `do` block used `pure`

and we never needed to change it even though we kept changing the Monad.

Using `pure` is a good habit to acquire.

Now that we have a cleaner `apply` implementation, we should compare it with `bind`:

```
instance bindParser :: Bind (Parser e) where
  bind p f = Parser \s -> do
    Tuple s1 x <- parse p s
    parse (f x) s1
```

Once again, there isn't much difference. This is important to realize since it justifies why we can replace all the code we wrote for `apply` with `ap`:

```
import Control.Monad (ap)

instance applyParser :: Apply (Parser e) where
  apply = ap
```

But this still seems like magic. So let's look once again at the implementation of `ap`:

```
ap :: ∀ a b m. Monad m => m (a -> b) -> m a -> m b
ap ff fx = do
    f <- ff ❶
    x <- fx ❷
    pure $ f x
```

❶ This line matches what we wrote by hand, i.e. `Tuple s1 f <- parse p1 s`.

❷ This line matches what we wrote by hand, i.e. `Tuple s2 f <- parse p2 s1`.

But how does `f <- ff` work when it doesn't call `parse` like our handwritten implementation?

And what about the next line of `ap`. How does `x <- fx` get passed `s1`, i.e. the `String` returned by executing the first `Parser`?

To fully understand what's going on, we'll need to write `bind` without the `do` syntax:

```
instance bindParser :: Bind (Parser e) where
  bind p f = Parser \s -> parse p s
    >>= (\(Tuple s1 x) -> parse (f x) s1) ❶
```

❶ Redundant Parentheses have been added to segregate the Functions passed to `Either's bind`, i.e. `>>=`.

And we'll also write `ap` without the `do`:

```
ap :: ∀ a b m. Monad m => m (a -> b) -> m a -> m b
ap ff fx = ff >>= (\f -> fx >>= (\x -> pure $ f x)) ❶
```

❶ Redundant Parentheses have been added to segregate the Functions passed to `m's bind`, i.e. `>>=`.

And we're going to need `pure` from our `Parser`:

```
instance applicativeParser :: Applicative (Parser e) where
  pure x = Parser \s -> pure $ Tuple s x
```

We'll start with `ap` and replace the `bind` implementation from `Parser` one step at a time to see the final computation.

This is going to be a daunting task. Not because it's difficult, but because there's so many symbols. So, go through this slowly:

```
p >>= f = Parser \s -> parse p s >>=
    (\(Tuple s1 x) -> parse (f x) s1)  -- infixed `bind` for reference

pure v = Parser \s -> pure $ Tuple s v -- `pure` for reference
-----------------------------------------------------------------------
ff >>= (\f -> fx >>= (\y -> pure $ f y)) ❶

p1 >>= (\f -> p2 >>= (\y -> pure $ f y)) ❷

Parser \s -> parse p1 s >>=
  (\(Tuple s1 x) -> parse ((\f -> p2 >>= (\y -> pure $ f y)) x) s1) ❸

Parser \s -> parse p1 s >>=
  (\(Tuple s1 x) -> parse (p2 >>= (\y -> pure $ x y)) s1) ❹

Parser \s -> parse p1 s >>=
  (\(Tuple s1 x) -> parse (Parser \s -> parse p2 s >>=
    (\(Tuple s2 z) -> parse ((\y -> pure $ x y) z) s2)) s1) ❺

Parser \s -> parse p1 s >>=
  (\(Tuple s1 x) -> parse (Parser \s -> parse p2 s >>=
    (\(Tuple s2 z) -> parse (pure $ x z) s2)) s1) ❻

Parser \s -> parse p1 s >>=
  (\(Tuple s1 x) -> parse p2 s1 >>=
    (\(Tuple s2 z) -> parse (pure $ x z) s2)) ❼

Parser \s -> parse p1 s >>=
  (\(Tuple s1 x) -> parse p2 s1 >>=
    (\(Tuple s2 z) -> parse (Parser \s -> pure $ Tuple s $ x z) s2)) ❽

Parser \s -> parse p1 s >>=
  (\(Tuple s1 x) -> parse p2 s1 >>=
    (\(Tuple s2 z) -> pure $ Tuple s2 $ x z)) ❾
```

❶ `ap's` implementation with `x` renamed to `y` to avoid name confusion.

❷ Renamed `ff` to `p1` and `fx` to `p2` since we're working with `Parsers`.

❸ Substituted the whole formula into `bind` where `p1` is bound to `p` and `(\f -> p2 >>= (\y -> pure $ f y))` is bound to `f`.

❹ Applied `x` to `(\f -> p2 >>= (\y -> pure $ f y))` binding it to `f`. Note that, in this case, `x` is actually a Function. Remember that we're working with `apply` where the first Parameter is a Function in a Context.

❺ Substituted `p2 >>= (\y -> pure $ x y)` into `bind` where `p2` is bound to `p` and `(\y -> pure $ x y)` is bound to `f`. Also renamed `s1` to `s2` and `x` to `z` to avoid name confusion.

❻ Applied `z` to `(\y -> pure $ x y)` binding it to `y`. Note that `z` is a Value since we're working with the second Parameter of `apply`, which is a Value in a Context.

❼ Evaluated `parse (Parser \s -> parse p2 s >>= (\(Tuple s2 z) -> parse (pure $ x z) s2)) s1`. This is a 2-step process. First, substitute the `Parser` into `parse`, which returns the Function that's in the `Parser`, i.e `\s -> parse p2 s >>= (\(Tuple s2 z) -> parse (pure $ x z) s2)`. Then simply apply `s1` to that Function where `s1` is bound to `s`.

❽ Substituted `pure $ x z` into `pure` where `x z` binds to `v`.

❾ Evaluated `parse (Parser \s -> pure $ Tuple s $ x z) s2` where `s2` is bound it to `s`.

If your head didn't explode after working through that, here's the final result with `x` renamed to `f` and `z` renamed to `x`:

```
apply p1 p2 = Parser \s -> parse p1 s
  >>= (\(Tuple s1 f) -> parse p2 s1
    >>= (\(Tuple s2 x) -> pure $ Tuple s2 $ f x))
```

Our final result through all those substitutions is just the de-sugared version of our handwritten `apply`, i.e. if you take the `do` version of `apply` that we wrote and rewrite it using `>>=`:

```
instance applyParser :: Apply (Parser e) where
  apply p1 p2 = Parser \s -> do
    Tuple s1 f <- parse p1 s
    Tuple s2 x <- parse p2 s1
    pure $ Tuple s2 $ f x
```

Hopefully, this arduous process has shed some light on what appears from time to time as magic. The real take-away here is that `bind` and `pure` can be combined to produce `apply`. This can be seen in the implementation of `ap`, which uses both `pure` and, even though it's hidden in the `do` notation, `bind`.

Once again, `bind` is doing all kinds of things in the backgroud for us. In the case of `Parser`, it's passing the `Strings` and it's calling `parse`. Even though it cannot be see from the code in `ap`:

```
ap :: ∀ a b m. Monad m => m (a -> b) -> m a -> m b
ap ff fx = do
    f <- ff
    x <- fx
    pure $ f x
```

19.6. Coding with Monadic Parsers

Let's take some of the `Parsers` we wrote using the Applicative-style and rewrite them using the `do` syntax.

First, we'll take a look at `twoChars`:

```
twoChars :: ∀ e. Parser e (Tuple Char Char)
twoChars = Tuple <$> char <*> char
```

`twoChars` parses 2 characters and returns them in a `Tuple`. Rename the old version to `twoCharsA` to denote it's the Applicative version. Then recode it using `bind`, i.e. `>>=`, and call it `twoCharsB`.

Only advance once those steps have been completed.

We'll start with the Type Signature, which is unchanged:

```
twoCharsB :: ∀ e. Parser e (Tuple Char Char) ❶
```

❶ Don't forget to call it `twoCharsB`.

This Parser is Polymorphic in `e` meaning that it'll work with any error Type.

Next, we'll call the `Parser`, `char`, once and capture it's return value:

```
twoCharsB :: ∀ e. Parser e (Tuple Char Char)
twoCharsB = char >>= \c1 -> ???
```

And we'll need to do it again to get the second character:

```
twoCharsB :: ∀ e. Parser e (Tuple Char Char)
twoCharsB = char >>= \c1 -> char >>= \c2 -> ???
```

And now we can return the characters in the `Tuple`. We must be careful to place them in the same order as `twoCharsA`, i.e. the first character in the first position of the `Tuple`:

```
twoCharsB :: ∀ e. Parser e (Tuple Char Char)
twoCharsB = char >>= \c1 -> char >>= \c2 -> pure $ Tuple c1 c2 ❶
```

❶ Don't forget to wrap the `Tuple` in a `Parser` since that's the Type that we're returning. We do this by using `pure`.

Now, before reading any further, rewrite this Function using `do` notation. Call it `twoChars`.

We'll start with the exact same Type Signature as before:

```
twoChars :: ∀ e. Parser e (Tuple Char Char)
```

Next, we'll add the `do`:

```
twoChars :: ∀ e. Parser e (Tuple Char Char)
twoChars = do
  ???
```

Next we'll call `char` and capture the character it'll return:

```
twoChars :: ∀ e. Parser e (Tuple Char Char)
twoChars = do
  c1 <- char
  ???
```

And now we'll do it again:

```
twoChars :: ∀ e. Parser e (Tuple Char Char)
twoChars = do
  c1 <- char
  c2 <- char
  ???
```

And finally, we'll return the result in a `Tuple`:

```
twoChars :: ∀ e. Parser e (Tuple Char Char)
twoChars = do
  c1 <- char
  c2 <- char
  pure $ Tuple c1 c2 ❶
```

❶ Again, don't forget to wrap the `Tuple` in a `Parser` since that's the Type that we're returning.

You should be able to test using our old test code, if you like.

Here's another `Parser` we wrote using `Apply`:

```
threeChars :: ∀ e. Parser e String
threeChars =
  (\c1 c2 c3 -> fromCharArray [c1, c2, c3]) <$> char <*> char <*> char
```

Rename this to `threeCharsA` and write it again but this time use the `do` syntax. Call this new Function `threeChars`. Don't read on until you've finished.

So as usual, we start with the Type Signature, which doesn't change:

```
threeChars :: ∀ e. Parser e String
```

Next, we'll start the Function with `do`:

```
threeChars :: ∀ e. Parser e String
threeChars = do
  ???
```

Next, we'll call `char` 3 times capturing the characters into Variables:

```
threeChars :: ∀ e. Parser e String
threeChars = do
  c1 <- char
  c2 <- char
  c3 <- char
  ???
```

Then we can just return the final String:

```
threeChars :: ∀ e. Parser e String
threeChars = do
  c1 <- char
  c2 <- char
  c3 <- char
  pure $ fromCharArray [c1, c2, c3] ❶
```

❶ Don't forget to use `pure` to wrap the final `String` in a `Parser`.

Here we're coding in the `Parser` Monad. This is why we need to return a `Parser`. Remember, `Parsers` must be run using `parse`. All we've done here is build a complex composition of `Parsers` each of which contains their own parsing Function.

Next, we'd like to have a way to parse specific characters. So we're going create a `Parser` that lets us do that by parsing a single character and then checking it against a Predicate.

Let's call it `satisfy`. It will take a Predicate and return a Parser. It will call `char` internally to parse the character and then use the Predicate to make sure it matches.

Before we code it together, see how far you can get with it. Please don't just read ahead. Try it.

We'll start `satisfy` with it's Type Signature:

```
satisfy :: ∀ e. ParserError e => (Char -> Boolean) -> Parser e Char
```

We need to constrain `e` to be limited to Types with `ParserError` instances. This Constraint is necessary because `satisfy` can fail. Also, notice our `Parser` is going to return a character that satisfies the Predicate, hence the Type `Parser e Char`. And our Predicate will accept a `Char` and if it meets its requirements, then it will return `true`.

Now, we can begin to code the `Parser`:

```
satisfy :: ∀ e. ParserError e => (Char -> Boolean) -> Parser e Char
satisfy pred = char >>= \c -> ???
```

The Parameter, `pred`, is our Predicate. We'll use it to validate the parsed character.

So far, we call the `Parser`, `char` and `bind` it to `c`. Notice, I wrote it with `bind` and not `do`. Sometimes, `>>=` is easier to use than `do` when you only have 1 line of code. Otherwise, it would've been multiple lines of code.

Next thing we'll need to do is make sure it meets the requirements of the Predicate:

```
satisfy :: ∀ e. ParserError e => (Char -> Boolean) -> Parser e Char
satisfy pred = char >>= \c -> if pred c then ??? else ???
```

Now, we need to figure out what to do in each case. First, the success case:

```
satisfy :: ∀ e. ParserError e => (Char -> Boolean) -> Parser e Char
satisfy pred = char >>= \c -> if pred c then pure c else ???
```

In the success case, we simply return the `Parser` that always returns the character that `char` parsed.

Now, we should handle the error case. We know it has to return a `Parser`, so this is our usual starting point:

```
satisfy :: ∀ e. ParserError e => (Char -> Boolean) -> Parser e Char
satisfy pred = char >>= \c -> if pred c then pure c else Parser \s -> ???
```

The next problem to solve is what does the parsing Function return?

We know it must be an `Either` and we also know that it's an error so we'll need a `Left`:

```
satisfy :: ∀ e. ParserError e => (Char -> Boolean) -> Parser e Char
satisfy pred = char
  >>= \c -> if pred c then pure c else Parser \_ -> Left ??? ❶
```

❶ It's clear that we're not going to use the `String` Parameter to the `Parser` function. It's been renamed from `s` to `_`.

Now all we need is a good error Value. Let's look at our existing errors to see if we already have one:

```
data PError
  = EOF
```

Well, that's underwhelming. Okay, let's add an error that makes sense for `satisfy`:

```
data PError
  = EOF
  | InvalidChar String ❶
```

❶ This has a `String` to allow there to be a message for what was expected.

But this error is needed by our library Function, `satisfy`, and the library won't know about our `PError` Type because that's defined in our Application. Well, not really, since we're working in a single file here for learning purposes, but we've made `satisfy` general by constraining the error Type to be `ParserError`. So it could be moved out into another module or into a library.

If we want to use `InvalidChar` from `satisfy`, we'll have to add it to the `ParserError` Typeclass:

```
class ParserError e where
  eof :: e
  invalidChar :: String -> e
```

And now we'll have to update the `ParserError` Instance for `PError`:

```
instance parserErrorPError :: ParserError PError where
  eof = EOF
  invalidChar = InvalidChar ❶
```

❶ This was added.

Now we can use it:

```
satisfy
  :: ∀ e
  .  ParserError e
  => String ❶
  -> (Char -> Boolean)
  -> Parser e Char
satisfy expected pred = char >>= \c ->
  if pred c then pure c else Parser \_ -> Left $ invalidChar expected
```

❶ Added `String` Parameter in case there's an error. It'll be what was expected, hence, the Variable name `expected`.

Okay, but this isn't great. I had to construct a Parser on the fly just to return an error. Blech!

Write a Function called `fail`, that will return a Parser that always fails. Give it a try and then continue reading.

Here's fail's Type Signature:

```
fail :: ∀ e a. ParserError e => e -> Parser e a
```

Once again, we need the Contraint because this `Parser` will fail. In fact, it ALWAYS fails.

If you got stumped writing the Type Signature, stop now and see if you can finish the Function. Then read on.

We write the right-hand side first:

```
fail :: ∀ e a. ParserError e => e -> Parser e a
fail e = ???
```

Here `e` is the error.

Next, we'll write the left-hand side:

```
fail :: ∀ e a. ParserError e => e -> Parser e a
fail e = Parser \s -> ???
```

We've seen this pattern before.

Next, we have to return an error that has a Type of `Either`:

```
fail :: ∀ e a. ParserError e => e -> Parser e a
fail e = Parser \s -> Left e
```

Just because this compiles doesn't mean we're done. We should look to see if we can make it better. That is reduce it or generalize it. Can you see what we could improve on?

The first thing I note is that we're not using the `String`, `s`, that's passed to us. So let's eliminate it:

```
fail :: ∀ e a. ParserError e => e -> Parser e a
fail e = Parser \_ -> Left e
```

But now, it should be obvious that the pattern `\_ ->` tells us that we have a case for using `const`. Let's add that:

```
fail :: ∀ e a. ParserError e => e -> Parser e a
fail e = Parser $ const $ Left e
```

And now, we have a reduced version of our original.

Go back and change `satisfy` to use `fail` before reading on.

Here's the change:

```
satisfy
  :: ∀ e
  .  ParserError e
  => String
  -> (Char -> Boolean)
  -> Parser e Char
satisfy expected pred =
  char >>= \c -> if pred c then pure c else fail $ invalidChar expected
```

Now that we've factored out the clutter this reads a lot better. Much better.

We're going to use a new library to help us write the next `Parser`, so let's add it now. Update your `spago.dhall` file to add the `unicode` package:

```
{-
Welcome to a Spago project!
You can edit this file as you like.
-}
{ name = "my-project"
, dependencies =
  [ "bifunctors"
  , "console"
  , "contravariant"
  , "effect"
  , "either"
  , "foldable-traversable"
  , "integers"
  , "lists"
  , "maybe"
  , "newtype"
  , "nonempty"
  , "prelude"
  , "profunctor"
  , "psci-support"
  , "strings"
  , "tuples"
  , "unfoldable"
  , "unicode" ❶
  ]
, packages = ./packages.dhall
, sources = [ "src/**/*.purs", "test/**/*.purs" ]
}
```

❶ Add this line to your file.

Then run the following command to install and build that new package:

```
npx spago build
```

Next, write a Parser that will parse a single digit and call it `digit`. Use `satisfy` and `isDecDigit`. Consult `Pursuit` for details on `isDecDigit` when you code it.

Please do not read on until you're done coding.

Here's the Type Signature for `digit`:

```
digit :: ∀ e. ParserError e => Parser e Char ❶
```

❶ We need the Constraint because we're going call `satisfy`, which has this Constraint because it can fail.

Next, we use `satisfy` and `isDecDigit` from the `purescript-unicode` package:

```
import Data.CodePoint.Unicode (isDecDigit) ❶
import Data.String.CodePoints (codePointFromChar) ❷

digit :: ∀ e. ParserError e => Parser e Char
digit = satisfy "digit" (isDecDigit <<< codePointFromChar) ❸
```

❶ Don't forget to import `isDecDigit`.

❷ We also have to import `codePointFromChar` to convert a `Char` to `CodePoint`.

❸ `"digit"` is the Value we'll get in `InvalidChar` if we didn't parse the expected character.

That wasn't too hard. `satisfy` did all the heavy lifting for us.

So next, write `letter` that parses just a single alpha character. Use `isAlpha`. Try it first before reading on.

This is nearly identical to `isDecDigit`:

```
import Data.CodePoint.Unicode (isDecDigit, isAlpha) ❶

letter :: ∀ e. ParserError e => Parser e Char
letter = satisfy "letter" (isAlpha <<< codePointFromChar) ❷
```

❶ Don't forget to add `isAlpha` to the imports.

❷ `"letter"` is error Value that tells us what we expected to parse.

Now, we want to write a `Parser` that we'll call `alphaNum`. While it's possible to write this using `isAlphaNum` from `Data.CodePoint.Unicode`, we don't need to do all that extra work. Instead, we should be able to use `letter` and `digit` to accomplish this.

We could accomplish this the brute force way:

```
alphaNum :: ∀ e. ParserError e => Parser e Char
alphaNum = Parser \s -> case (parse letter s :: Either PError _) of ❶
  Right x -> Right x
  Left _ -> case parse digit s of ❷
    Left err -> Left err
    Right y -> Right y
```

❶ The compiler couldn't determine the Type for `e` so it could validate that it has a `ParserError` Instance. So just to get it to compile I used my Application's error Type. This is just a temporary hack since we're NOT going to keep this code for reasons that will soon become apparent.

❷ Here's why it couldn't figure out the error Type. We ignore the `Left` Value. And even if we did give it a name, we'd have to use it before the compiler could try to infer its Type.

Looking at this code make my eyes hurt. What we want is to try `letter` and if it fails then to try `digit`, but we have to write this mess.

Another thing you might be thinking is why is this code in our library. Typically, this kind of code with `case` expressions has been in `Parser's` Instances. Yet, here it is out it the wild. That should be a big red flag.

Can you think of a Typeclass that lets us try one thing and then if it fails we try another?

Do you remember `Alt`?

Here's it's definition to help remind us:

```
class Functor f <= Alt f where
  alt :: f a -> f a -> f a

infixl 3 alt as <|> ❶
```

❶ Take note that this is Left-Associative. This means we will favor the left over the right.

Alt, is a choice we make between 2 Functors. In the case of `Parsers`, we want to choose the one that Parsers successfully favoring the left-hand side of the `Alt` Operator, `<|>`.

Now, if `Parser` supported `Alt`, we could write the following:

```
alphaNum :: ∀ e. ParserError e => Parser e Char
alphaNum = letter <|> digit
```

That would be so much better than the brute force approach. So, go ahead and code up the `Alt` Instance for `Parser` before reading on. Feel free to cheat and look at our brute force implementation for clues (minus the explicit Type specification, of course). Also, don't forget to import `Control.Alt`.

We start with the usually starting point, the Type Signature:

```
import Control.Alt (class Alt, (<|>)) ❶

instance altParser :: Alt (Parser e) where
```

❶ Import `Alt`. Note the `class` keyword.

Not surprisingly, we're holding `e` constant.

Now we'll code the left-hand side first:

```
instance altParser :: Alt (Parser e) where
  alt p1 p2 = ??? ❶
```

❶ `p1` and `p2` are both `Parsers`.

And, of course, we're returning a `Parser`, so we'll use our starting pattern:

```
instance altParser :: Alt (Parser e) where
  alt p1 p2 = Parser \s -> ???
```

Now we'll have to do some of the same work we did in the brute force approach, but we'll only have to do this once. So let's try the leftmost `Parser` first, since `<|>` is Left-Associative:

```
instance altParser :: Alt (Parser e) where
  alt p1 p2 = Parser \s -> case parse p1 s of
    ???
```

An now we'll handle the error case where `p1` fails to parse:

```
instance altParser :: Alt (Parser e) where
  alt p1 p2 = Parser \s -> case parse p1 s of
    Left _ -> parse p2 s
    ???
```

In the error case, we simply just try the second `Parser`, `p2` which will return success or an error. This will be the final result of the `<|>` operation.

And finally, we have to handle the success case for `p1`:

```
instance altParser :: Alt (Parser e) where
  alt p1 p2 = Parser \s -> case parse p1 s of
    Left _ -> parse p2 s
    Right x -> Right x
```

In the success case, we just return the success value, which has a Type of `Either`.

Now, this code is forever concealed in this instance code and we can leverage it by using `<|>`.

Now, add `alphaNum` to your code:

```
alphaNum :: ∀ e. ParserError e => Parser e Char
alphaNum = letter <|> digit
```

So, what will be the error if `alphaNum` fails to parse?

It'll be `invalidChar "digit"`.

And what if we were to rearrange the order of `Parsers`?:

```
alphaNum :: ∀ e. ParserError e => Parser e Char
alphaNum = digit <|> letter
```

Now it'll be `invalidChar "letter"`.

Neither of these are accurate. They just happen to be the last error. What we'd like the error to be is `invalidChar "alphaNum"`.

So, modify `alphaNum` to return `invalidChar "alphaNum"` as its error before reading on.

Here's how we do that:

```
alphaNum :: ∀ e. ParserError e => Parser e Char
alphaNum = letter <|> digit <|> fail (invalidChar "alphaNum")
```

We use a `Parser` which always fails, i.e. `fail`. By putting it last, it'll be the error that this `Parser` returns if the previous 2 fail.

At this point, we should test our `Parsers`, but before we do, write a version of `count` that will return a `String`. Call it `count'`. It should delegate to `count`, which we wrote in an earlier Chapter.

Hint: Look at how you converted `Array Char` to `String` in your existing test code.

Give it a try before reading on.

`count'` is a simple helper Function that takes the same exact Parameters as `count` but returns a `String`, therefore the following is its Type Signature:

```
count' :: ∀ e. Int -> Parser e Char -> Parser e String ❶
```

❶ Notice how we're now expecting a `Parser e Char` instead of `Parser e a` like `count` takes. That's because, we're going to call `count` and we want it to give us back a `Parser e (Array Char)` so we can convert the `Array Char` to a `String`.

`count'` returns a `Parser e String`, which is a `Parser` that returns a `String` when it successfully parses.

Next we'll just call `count` and then convert the `Array Char` to a `String` using `fromCharArray`:

```
count' :: ∀ e. Int -> Parser e Char -> Parser e String
count' n p = fromCharArray <$> count n p
```

Remember that we're mapping here with `<$>` since the `Array Char` is in a Context, i.e. in a `Parser`.

And now you can add the following to your test code:

```
log $ show $ parse' (count' 3 digit) "123456" ❶
log $ show $ parse' (count' 3 digit) "abc456" ❷
log $ show $ parse' (count' 4 letter) "Freddy" ❸
log $ show $ parse' (count' 10 alphaNum) "a1b2c3d4e5" ❹
log $ show $ parse' (count' 10 alphaNum) "######" ❺
```

❶ Prints `(Right (Tuple "456" "123"))`.

❷ Prints `(Left (InvalidChar "digit"))`.

❸ Prints `(Right (Tuple "dy" "Fred"))`.

❹ Prints `(Right (Tuple "" "a1b2c3d4e5"))`.

❺ Prints `(Left (InvalidChar "alphaNum"))`.

Run the tests and make sure our code is working.

And it is NOT:

```
[error] Some of your project files import modules from packages that are not
in the direct dependencies of your project.
To fix this error add the following packages to the list of dependencies in
your config:
- control
```

Add `control` to `spago.dhall` and rerun the program.

19.7. Writing a Date Parser

When we first started talking about parsers, I gave the example of a Regular Expression that we could use to parse a date:

```
\d{1,2}\/\d{1,2}\/\d{4}
```

I think you can see why this is a terrible bit of code to maintain compared to our `Parser` Type and the Monadic Parsers we've built so far.

So, what we're going to do is write a `Parser` for dates. We'll accept them in 2 formats:

```
10/2/1962
1962-10-02
```

These are our specifications. And now we'll do, what is called, a ***Gap Analysis*** to see what we're missing, i.e. based on these specs and what we've written so far, what pieces are missing.

First, we already have a `Parser` for the following:

```
\d{4}
```

This says we'll parse 4 digits and we have `count'` for this. But we don't have a `Parser` for:

```
\d{1,2}
```

This says we're going to parse 1 or 2 digits. We need to write a `Parser` that will accept a range of counts. We'll call it `range`.

Next, we'd like to model our parsed date since it's a complex structure unlike the things that we've parsed so far, e.g. `Array Char`, `String`, `Char`. We'll call this Type `DateParts`.

And finally, we'll need 2 `Parsers`. One to handle each format we're supporting.

That seems to be all that we'll need to accomplish our goal of date parsing. So let's get started.

First thing to do is model the Type. Please code up a Type that will model the parsed date before reading on.

Modeling is always difficult because there are always so many ways to do it. Our goal should be to find the best model for our situation. That's difficult to do in the real world but even more so in an artificial environment that we work in when we're learning.

So, I'm going to through a few possibilities and then I'm going to pick one based on totally artificial criteria.

Here's the most naïve approach:

```
data DateParts = DateParts Int Int Int
```

Looking at the 3 `Ints` it's impossible to tell which is year and which is day or month. Let's fix that:

```
newtype Year = Year Int
newtype Month = Month Int
newtype Day = Day Int

data DateParts = DateParts Year Month Day
```

Keep in mind that these are the parts that we parsed out of a `String`. This is not guaranteed to be a valid date. That's beyond the scope of this problem. In the real world, we'd first parse the date and then use a date library to do the hard work of validation for us.

Can you see any improvements that we could make on model? I can only see one.

It might be nice to know what format we originally encountered when we parsed the date just in case we want to print it back out in it's original format.

We'll create a Sum Type for that:

```
data DateParts
  = YearFirstDate Year Month Day
  | MonthFirstDate Year Month Day
```

Now, that we've done that, let's start over but take a different tact. We'll start with:

```
type DateParts =
  { year :: Int
  , month :: Int
  , day :: Int
  }
```

Now, we'll want to replace `Int` for a different reason here. We did it before to help us understand which `Int` was which. But with a Record we don't have that confusion, since we have fields with name.

When we created Types for year, month and day, we used `newtype`. This gave us the added benefit of keeping developers, including us, from supplying things to the `DateParts` Data Constructor in the wrong order.

Here order doesn't matter as much, unless we write a Function to build this Record, then Parameters can get out of order. But even if we don't, it's still worth modeling year, month and day so that we don't get our wires crossed, e.g. passing 12 into the year or 1973 into the month:

```
newtype Year = Year Int
newtype Month = Month Int
newtype Day = Day Int

type DateParts =
  { year :: Year
  , month :: Month
  , day :: Day
  }
```

This is Isomorphic to our previous model except for keeping track of the original format. Let's add that:

```
data DateFormat = YearFirst | MonthFirst

type DateParts =
  { year :: Year
  , month :: Month
  , day :: Day
  , format :: DateFormat
  }
```

The models are now Isomorphic, i.e. I can write a set of Functions to translate between the 2 Models with no loss of information.

Which one is better? Which one do you think and why? The why is the most important part to answer.

Decide which one you prefer and why. Be careful and avoid Post Rationalizing your intuitive selection, which is an easy trap to fall into.

For example, maybe we don't like Records because we haven't used them much and so the other model is more attractive to us. And now, to justify that SUBJECTIVE decision, we conjure up a rational, seemingly OBJECTIVE, reason for our choice. We are human and are all susceptible to this. And so it's really important to be aware of this so that we fight against it.

So, with all of that in mind, what's you choice and WHY? Please think about it before reading further.

Okay, now that you have your choice, I'll tell you mine.

The way I'm going to decide is by pushing the models to their limits. Let's imagine we have dozens and dozens of formats that we want to support. How does each model perform under those conditions?

Let's imagine that scenario with the first model:

```
data DateParts
  = YearFirstDate Year Month Day
  | MonthFirstDate Year Month Day
  | DayFirstDate Year Month Day
  | SomeOtherFormat Year Month Day
  | AndYetAnotherFormat Year Month Day
  | SomeEsotericFormat Year Month Day
  | UghYetAnotherFormat Year Month Day
  | KillMeFormat Year Month Day
```

Notice how we had to repeat `Year Month Day` for all of these. Not pretty.

Let's see how our second model performs under these same conditions:

```
data DateFormat
  = YearFirst
  | MonthFirst
  | DayFirst
  | SomeOther
  | AndYetAnother
  | SomeEsoteric
  | UghYetAnother
  | KillMe

type DateParts =
  { year :: Year
  , month :: Month
  , day :: Day
  , format :: DateFormat
  }
```

This model seems to handle addition formats better than our first model. It's not a huge win but it's a win.

But, there is one huge win the second model has over the first and that's the requirement for `case` expressions. When we work with the first model, we ALWAYS have to write a `case` expression to access year, month and day:

```
getYear :: DateParts -> Year
getYear = case _ of
  YearFirstDate year _ _ -> year
  MonthFirstDate year _ _ -> year
  DayFirstDate year _ _ -> year
  SomeOtherFormat year _ _ -> year
  AndYetAnotherFormat year _ _ -> year
  SomeEsotericFormat year _ _ -> year
  UghYetAnotherFormat year _ _ -> year
  KillMeFormat year _ _ -> year
```

And `getMonth` will have a similar implementation as will `getDay`. Lots of boilerplate.

Now, we could write a SINGLE Function that converts the date to a Record:

```
getDate :: DateParts -> { year :: Year, month :: Month, day :: Day }
getDate = case _ of
  YearFirstDate year month day -> { year, month, day }
  MonthFirstDate year month day -> { year, month, day }
  DayFirstDate year month day -> { year, month, day }
  SomeOtherFormat year month day -> { year, month, day }
  AndYetAnotherFormat year month day -> { year, month, day }
  SomeEsotericFormat year month day -> { year, month, day }
  UghYetAnotherFormat year month day -> { year, month, day }
  KillMe year month day -> { year, month, day }
```

But we've lost the format information. Well, we can just add that. No problem:

```
getDate
  :: DateParts
  -> { year :: Year, month :: Month, day :: Day, format :: DateFormat }
getDate = case _ of
  YearFirstDate year month day ->
    { year, month, day, format: YearFirst }
  MonthFirstDate year month day ->
    { year, month, day, format:  MonthFirst}
  DayFirstDate year month day ->
    { year, month, day, format:  DayFirst}
  SomeOtherFormat year month day ->
    { year, month, day, format:  SomeOther}
  AndYetAnotherFormat year month day ->
    { year, month, day, format:  AndYetAnother}
  SomeEsotericFormat year month day ->
    { year, month, day, format:  SomeEsoteric}
  UghYetAnotherFormat year month day ->
    { year, month, day, format:  UghYetAnother}
  KillMeFormat year month day ->
    { year, month, day, format:  KillMe}
```

Well, that's a lot less boilerplate code. But wait?

We just wrote a Homomorphic Function from our first model to our second. Why not just adopt the second model and we won't have write any of these Functions.

So that's my choice. The second model.

Notice the process was to push the model to its limits. That's because it's hard to see the benefits of one model over another when there are only 2 formats. Had we not thought to do this, we would've wound up

writing these Functions and we might not have noticed that it's boilerplate code since there's only 2 cases. But it's still boilerplate.

Now that we have our model, go ahead and add it to your code. Here it is for reference:

```
newtype Year = Year Int
newtype Month = Month Int
newtype Day = Day Int

data DateFormat
  = YearFirst
  | MonthFirst

type DateParts =
  { year :: Year
  , month :: Month
  , day :: Day
  , format :: DateFormat
  }
```

The next thing that our Gap Analysis uncovered is that we need a Function like `count'` but something that takes a range.

Let's examine a Regular Expression for such a scenario:

```
\d{3,8}
```

This say that we always want 3 digits and if possible 5 more digits. We can decompose this into 2 steps:

```
\d{3}\d{0,5}
```

We have `count` that handles `\d{3}`. But we need to write something that handles `\d{0,5}`.

Let's write a Function that we'll call `atMost` that parses ***at most*** a specified count.

Write the Type Signature for this Function before reading further.

We're going to start simply and, if we can, generalize later:

```
atMost :: ∀ e a. Int -> Parser e a -> Parser e (Array a)
```

Notice that we did NOT make `atMost` take a `Parser e Char`. We're trying to keep this a bit general in that

sense. But we didn't got totally general like we did with `count`, which uses an `Unfoldable`. Instead we're going to start off with a concrete type `Array`.

This will help us think about the problem concretely. We can always try to generalize it once we have code to look at. It turns out that this is the same approach we took when we wrote `count`.

Next, write just the left-hand side of `atMost`. Feel free to cheat and look at what we did for `count`. Write that much before reading on.

Here's the left-hand side:

```
atMost :: ∀ e a. Int -> Parser e a -> Parser e (Array a)
atMost n p
  ???
```

Now we should use Guards here like we did in `count`. Add a check for an invalid value of `n` and then continue reading.

```
atMost :: ∀ e a. Int -> Parser e a -> Parser e (Array a)
atMost n p
  | n <= 0 = pure [] ❶ ❷
  ???
```

❶ We don't use `none` here like we did in `count` because we're using `Array`. So, instead, we use the hardcoded value `[]`, which just so happens to be `Array's` Value for `none`.

❷ Remember, `pure` returns a `Parser` that returns a constant value. In this case, it always returns `[]`.

Next, write just the left-hand side of the general case before advancing.

Here's that much:

```
atMost :: ∀ e a. Int -> Parser e a -> Parser e (Array a)
atMost n p
  | n <= 0    = pure []
  | otherwise = ???
```

Let's think linearly here. We have to parse a single character or, more generally, we have to parse an `a`. How do we do that?

We need to use `p`. It's all we've got. So add that to your code and then continue reading.

Here's that minor change:

```
atMost :: ∀ e a. Int -> Parser e a -> Parser e (Array a)
atMost n p
  | n <= 0    = pure []
  | otherwise = p ???
```

And if we stopped there, all we'd have accomplished is parsing once. But we want to parse `n` times. So we need to repeat this process until `n` is exhausted. That means we need to recurse.

But how do we combine `p` with a recursive call to `atMost`?

Notice one thing. `p` and `atMost` are both `Parsers`. So our question now changes. How do you combine `Parsers`?

We'll the answer should be on the tip of your tongue since that's what we've been doing all along. Here's an example of how we combined 2 `Parsers` in the past:

```
twoChars :: ∀ e. Parser e (Tuple Char Char)
twoChars = do
  c1 <- char
  c2 <- char
  pure $ Tuple c1 c2
```

We're combining `char` with `char` here. But this example is probably not helping you see how we're combining 2 `Parsers` and that's because `do` notation hides that fact.

So, instead, let's look at our `bind` version:

```
twoCharsB :: ∀ e. Parser e (Tuple Char Char)
twoCharsB = char >>= \c1 -> char >>= \c2 -> pure $ Tuple c1 c2
```

See how we "call" `char` and then bind its resulting Value to `c1` and then we "call" `char` again and bind its resulting Value to `c2`.

I put the word ***call*** in quotes because that's how we think of it as we're coding this, but `char` is just a `Parser` and this Function just COMBINES `Parsers`. It isn't until it's EXECUTED that `c1` and `c2` get bound to Values.

And so we combine `Parsers` using `bind`.

Before reading further, just add `>>=` to `atMost` and bind the result of `p` to a variable called `c` then return here.

Here's that small bit added:

```
atMost :: ∀ e a. Int -> Parser e a -> Parser e (Array a)
atMost n p
  | n <= 0    = pure []
  | otherwise = p >>= \c -> ???
```

What do we want to have happen next?

What we want is to recurse with `atMost` which will return a `Parser` that "contains" an `Array`. We want to `Cons` to that `Array`, the character, `c`, that we parsed using `p`. You can import the `Array` operator for `Cons`, i.e. `(:)`, from `Data.Array`.

Go ahead and write that before moving on.

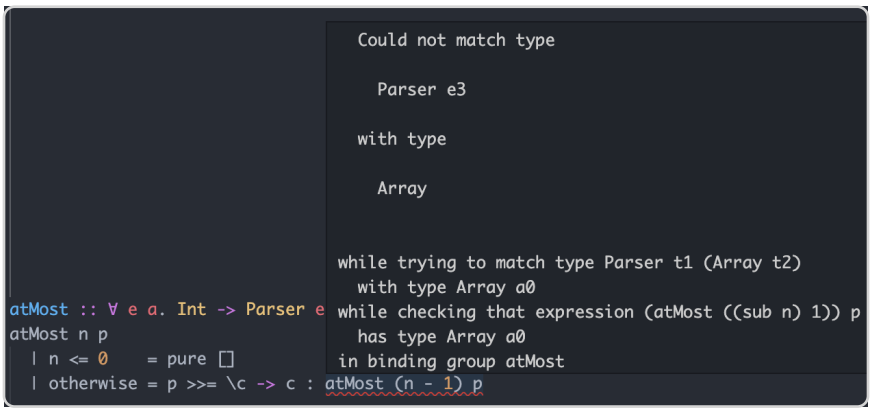
You may have written the following and got stuck on the compiler error:

```
import Data.Array ((:)) ❶

atMost :: ∀ e a. Int -> Parser e a -> Parser e (Array a)
atMost n p
  | n <= 0    = pure []
  | otherwise = p >>= \c -> c : atMost (n - 1) p  -- COMPILER ERROR!!
```

❶ You'll need to import the `Cons` Operator from `Array`.

If you did, here's the error you got:



And that's because you're trying to `Cons` to the head of a `Parser e3` and not an `Array`.

The `Array` is in the Context of the `Parser`. The way we can affect the `Array` is to `map` a Function over the `Parser`.

Go ahead and `map` the `Cons` Operator over the `Parser` before reading ahead. If you don't know where to start then please read on, but spend a minute thinking about how to do that even if you have to use a Lambda.

Let's do this one step at a time. First, let's refactor what we wrote. Even though it's wrong, refactoring will help us see how to `map` the `Cons` Operator.

```
atMost :: ∀ e a. Int -> Parser e a -> Parser e (Array a)
atMost n p
  | n <= 0    = pure []
  | otherwise = p >>= \c -> (c : _) $ atMost (n - 1) p
    -- COMPILER ERROR!!
```

We're still applying `Cons` to a `Parser`, so we haven't fixed anything. This refactoring step is just to help you see what to do next.

But before you do, take a minute to see how `c : []` is the same as `(c : _) $ []`.

Now, instead of Function Application, `$`, we're going to change the code to use `map`, `<$>`:

```
atMost :: ∀ e a. Int -> Parser e a -> Parser e (Array a)
atMost n p
  | n <= 0    = pure []
  | otherwise = p >>= \c -> (c : _) <$> atMost (n - 1) p
```

Now, this compiles. But does it work?

Let's look at what it does. First, it's a `Parser`. And that `Parser` will first parse using `p` and if that succeeds then... Wait. What happens if `p` fails?

If `p` fails, then `atMost` fails. That means that the whole `Parser` fails. And since we recurse with `atMost`, if at any point that fails then the whole thing fails.

We never want `atMost` to fail. We want it to OPTIONALLY parse no more than the specified number of elements. So `atMost` should ALWAYS succeed.

Okay, so this code won't work as it stands. We need to handle the cases where `p` or `atMost (n - 1) p` fails, or the combination of them fails.

We could add of bunch of checks inside `atMost` but that'll be a mess. So, we're going to factor it out.

We're going to write a `Parser`, called `optional`, that takes another `Parser` and if it `fails`, it defaults with a default Value.

Go ahead and write just the Type Signature. Then read on.

Here's the Type Signature for `optional`:

```
optional :: ∀ e a. a -> Parser e a -> Parser e a
```

We have a `Parser` and a Value. We can wrap that Value in a `Parser` using `pure` for the default `Parser`.

This will give us 2 `Parsers`, the original and one that always returns our default Value. If the first one fails, we want to parse with the second one.

Do you remember how we select the outcome of 2 Parsers based on failure? Take a look at what we wrote for `alphaNum` and finish writing `optional` before continuing.

Here's the implementation:

```
optional :: ∀ e a. a -> Parser e a -> Parser e a
optional x p = p <|> pure x
```

We're using `Alt`, which, if you remember, will try the first `Parser` and then if that fails, it'll try the second. Well, the second `Parser` here NEVER fails, which is exactly what we want to happen with `atMost`.

Return to `atMost` and use `optional` to make our `Parser` never fail. Then read on to see how you did.

Let's look at our deficient version of `atMost`:

```
atMost :: ∀ e a. Int -> Parser e a -> Parser e (Array a)
atMost n p
  | n <= 0    = pure []
  | otherwise = p >>= \c -> (c : _) <$> atMost (n - 1) p
```

We can see that in the error case, we're returning a `Parser` that always returns `[]`. And in the `otherwise` case, we're returning a `Parser`, `p >>= \c -> (c : _) <$> atMost (n - 1) p`. And if that massive `Parser` fails, we want to return a `Parser` that always returns `[]`, which is exactly what `optional` will do for us.

So all we have to do is pass `p >>= \c -> (c : _) <$> atMost (n - 1) p` to `optional`:

```
atMost :: ∀ e a. Int -> Parser e a -> Parser e (Array a)
atMost n p
  | n <= 0    = pure []
  | otherwise = optional [] $ p >>= \c -> (c : _) <$> atMost (n - 1) p
```

This code looks pretty scary. So, let's write some good tests to make sure that we haven't convinced ourselves that we know what we're doing.

We'll want to test boundary conditions. There are 3. One, when we pass a count of 0 or less. Two, when we can't parse anything. And three, when we could parse more but stop because we've hit the max.

Before we write those tests, let's write a version of `atMost` called `atMost'` that returns `Parser e String`. Look to `count'` to see how to do this. Write that Function before reading on.

Here's that Function:

```
atMost' :: ∀ e. Int -> Parser e Char -> Parser e String
atMost' n p = fromCharArray <$> atMost n p
```

The reason we're writing our `Parsers` to be `Parser e a` is for the greatest flexibility. We may want to use our `Parser` in a non-text scenario. But, generally speaking, we're mostly going to use it with `Strings`, which is why we're writing the prime versions, e.g. `atMost'`.

Now add the following to your test code and run it to make sure `atMost'` works:

```
log $ show $ parse' (atMost' (-2) alphaNum) "a1b2c3"❶
log $ show $ parse' (atMost' 2 alphaNum) "$_$"❷
log $ show $ parse' (atMost' 2 alphaNum) "a1b2c3"❸
```

❶ Prints `(Right (Tuple "a1b2c3" ""))`.

❷ Prints `(Right (Tuple "$_$" ""))`.

❸ Prints `(Right (Tuple "b2c3" "a1"))`.

If that prints out what we expect then we're good to go on to `range`.

But... before that, we promised ourselves that we'd generalize `atMost` later. It's really easy to just look at it and decide that it's fine the way it is, since it was difficult to write in the first place. If you're feeling that way now, maybe it's a good time to take a break and return here when you're ready.

I think it's worth making any library Function as flexible as possible. And, `atMost`, is clearly a general purpose `Parser`. So it's worth looking at it in that light.

Here's our current version:

```
atMost :: ∀ e a. Int -> Parser e a -> Parser e (Array a)
atMost n p
  | n <= 0    = pure []
  | otherwise = optional [] $ p >>= \c -> (c : _) <$> atMost (n - 1) p
```

We could change `Array` to an `Unfoldable` like we did in `count`. That worked out nicely. So, copy this Function and comment out one of the copies. We'll keep it around in case our experiment goes awry.

Next, change the Type Signature to replace `Array` with `Unfoldable`. Look at `count's` Type Signature for clues on how to do that.

Do this before returning here. By the way, we should expect compiler errors.

Here's the Type Signature change:

```
atMost :: ∀ e a f. Unfoldable f => Int -> Parser e a -> Parser e (f a)
atMost n p
  | n <= 0    = pure []   -- COMPILER ERROR!!
  | otherwise = optional [] $ p >>= \c -> (c : _) <$> atMost (n - 1) p
```

Here's the error:

```
  Could not match type

    Array

  with type

    f1


while trying to match type Array t3
  with type f1 a2
while checking that expression pure []
  has type Parser e0 (f1 a2)
in binding group atMost
atMost :: ∀ e a
atMost n p
  | n <= 0    = pure []
  | otherwise = optional [] $ p >>= \c -> (c : _) <$> atMost (n - 1) p
```

The compiler found an `Array` but was expecting Type `f1`, which is our `f`, which is an `Unfoldable`. If you remember, we saw this when we were generalizing `count`.

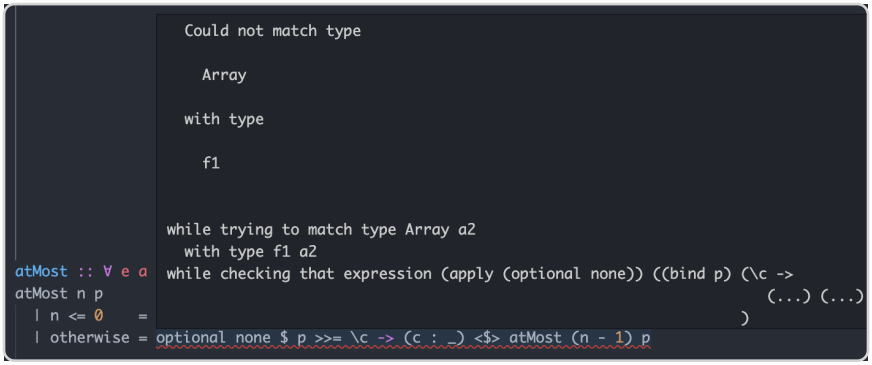
There's an Unfoldable helper called `none`. It uses the Generating Function from `Unfoldable` to generate 0 elements. For `Array`, it will return `[]`. For `List`, it'll return `Nil`. And so on.

So replace `[]` with `none` and see if it compiles. Make sure to change `[]` to `none` in both places.

Of course it doesn't. Here's the new code:

```
atMost :: ∀ e a f. Unfoldable f => Int -> Parser e a -> Parser e (f a)
atMost n p
  | n <= 0    = pure none
  | otherwise = optional none $ p >>= \c -> (c : _) <$> atMost (n - 1) p
                -- COMPILER ERROR!!
```

And here's the new compiler complaint:



Once again, it's complaining that it found an `Array` but was expecting an `f1`, i.e. an `Unfoldable`. But we're not using `Array` explicitly anymore. So that means we must be using it IMPLICITLY.

We must be calling a Function that expects an `Array`. Can you see where that is happening?

If you said, the `Cons` Operator, `:`, then you're right. We imported that Operator earlier when we first used it in `atMost`. We cannot use that anymore since we don't know if we're working with an `Array`. We also want to be able to work with `Lists` or any other `Unfoldable`.

How do we solve this dilemma? We need a Function like `Cons` but for any `Unfoldable`. Is there such a Function in PureScript?

If you peruse `Pursuit` for such a thing, you'll probably come up empty. I did. But, not is all lost. We can just pass a Function to `atMost` that will be a cons-like Function. Then we'll pass `:` from `Data.Array` when we're working with `Arrays` and we'll pass `:` from `Data.List` when we're working with `Lists`, etc.

So modify your Type Signature for `atMost` to take a cons-like Function as its first Parameter. If you're stuck on what the Type Signature for the cons-like Function should be, look at the Type Signature for `Cons` for `Array` or `List` and generalize it.

Then add a new first Parameter called `cons` and replace `:` with that Variable.

Make these changes and then continue on.

Here are those changes:

```
atMost
  :: ∀ e a f
  .  Unfoldable f
  => (a -> f a -> f a) ❶
  -> Int
  -> Parser e a
  -> Parser e (f a)
atMost cons n p ❷
  | n <= 0    = pure none
  | otherwise = optional none $ p
                  >>= \c -> cons c <$> atMost cons (n - 1) p ❸ ❹
```

❶ The cons-like Function takes an `a` and an `Unfoldable` of `a's` and returns an `Unfoldable` of `a's`. Reread the previous sentence and replace `cons-like` with `cons` and `Unfoldable` with `Array` or `List` and it's still true.

❷ We added the `cons` Parameter as the first Parameter. It's first so we can Partially Apply it easily.

❸ `(c : _)` has been replaced with `cons c`, a prefixed equivalent of `:`.

❹ We had to pass `cons` to the recursive call. This is easy to miss, but, luckily, the compiler is there to remind us.

So, now `atMost` will parse `0` or `n` times, but no more. Later, we'll see how to parse `0` or more and `1` or more.

But we're getting an error on `atMost'`. That's not surprising since we added an extra Parameter to `atMost`. So go fix that before reading on.

All we have to do is use the `(:)` as our cons-like function when calling `atMost`:

```
atMost' :: ∀ e. Int -> Parser e Char -> Parser e String
atMost' n p = fromCharArray <$> atMost (:) n p ❶
```

❶ We're passing `:` from `Data.Array` here. This is NOT the same Operator from `Data.List`.

Now we're ready to see if our code is working. Running the program displays the error:

```
[error] Some of your project files import modules from packages that are not
in the direct dependencies of your project.
To fix this error add the following packages to the list of dependencies in
your config:
- arrays
```

Add `arrays` to `spago.dhall` and rerun the program and verify that the last 3 lines of test code work:

```
log $ show $ parse' (atMost' (-2) alphaNum) "a1b2c3"❶
log $ show $ parse' (atMost' 2 alphaNum) "$_$"❷
log $ show $ parse' (atMost' 2 alphaNum) "a1b2c3"❸
```

❶ Prints `(Right (Tuple "a1b2c3" ""))`.

❷ Prints `(Right (Tuple "$_$" ""))`.

❸ Prints `(Right (Tuple "b2c3" "a1"))`.

And now we're ready to write `range`. It will take a `min` Value and a `max` Value. It will also take a `Parser` to parse at least `min` but no more than `max`.

It will accomplish this by leveraging both `count` and `atMost`. As usual, we'll write a concrete version and then generalize it later. Just write the Type Signature for a `range` that returns a `Parser` that returns an `Array a`, like we've done before.

Do this before reading any further.

Here's the Type Signature for `range`:

```
range :: ∀ e a. Int -> Int -> Parser e a -> Parser e (Array a)
```

Next, write the left-hand side of the equal sign of the Function implementation before reading on.

Here's that portion:

```
range :: ∀ e a. Int -> Int -> Parser e a -> Parser e (Array a)
range min max p = ???
```

Now, we're going to write the implementation using Guards. Code up the error case, i.e. where `min` and `max` are bad Values. Keep in mind that this is a bit more complicated that our other error cases with only 1 number. Do this before reading on.

We have to worry about 3 cases. One, `min` is 0 or less. Two, `max` is 0 or less. And, when `max` is less than `min`.

Here's that part:

```
range :: ∀ e a. Int -> Int -> Parser e a -> Parser e (Array a)
range min max p
  |    min <= 0
    || max <= 0
    || max < min  = pure [] ❶
  | otherwise     = ???
```

❶ Give an invalid range, we simply return an empty `Array` having consumed no characters from the `String`.

Now we need to think about what we want `range` to do in detail. We know that `range` will need to parse AT LEAST `min` times with the given `Parser`, `p`. We can use `count` for that.

Then if that succeeds, we'll want to optionally continue with the same `Parser`, `p`, until we reach the `max`. We'll use `atMost` to do that, but be careful. We don't want `atMost` to parse `max` times. That's because, at this point,`count` has already parsed `min` times.

Keep all of this in mind as you code up the success case. One quick hint is to remember how we combined `Parsers` in `atMost`. Bigger hint, we used `>>=`.

So, with all that, you have everything you'll need to code the rest of `range`. Do that before reading on.

The first step is to call `count` with `min` as the count:

```
range :: ∀ e a. Int -> Int -> Parser e a -> Parser e (Array a)
range min max p
  |    min <= 0
    || max <= 0
    || max < min  = pure []
  | otherwise     = count min p >>= ???
```

Next, we want to capture the result into a Variable, we'll call it `cs` to as if it were an `Array` of `Characters`, just to help us think:

```
range :: ∀ e a. Int -> Int -> Parser e a -> Parser e (Array a)
range min max p
  |    min <= 0
    || max <= 0
    || max < min  = pure []
  | otherwise     = count min p >>= \cs -> ???
```

And now we'll want to call `atMost` which, if we pass it `Cons`, i.e. `(:)`, from `Data.Array`, will return an `Array`. And we'll take that `Array` and append it to `cs` using `<>`:

```
range :: ∀ e a. Int -> Int -> Parser e a -> Parser e (Array a)
range min max p
  |    min <= 0
    || max <= 0
    || max < min  = pure []
  | otherwise     = count min p
                      >>= \cs -> (cs <> _) <$> atMost (:) (max - min) p
```

Okay, so that last step was quite a leap. So, let's look at it in detail.

Remember mapping `(c : _)` over the recursive call to `atMost` that we did earlier? Well, we're doing a similar thing here with `(c <> _)` over `atMost`.

Notice the count we send to `atMost` is `max - min`. That's because we've already parsed `min` times and we want to parse as many times that's left. Here are some concrete examples:

```
min  max   max - min  count  atMost
---  ---  ----------  -----  ------
 2    5       3         2      3
 3    7       4         3      4
 0    9       9         0      9 ❶
```

❶ This case points out a bug in our code.

Notice that the number sent to `count` plus the number sent to `atMost` is equal to `max`. That's because:

```
min + (max - min) = max
```

And now looking at these concrete examples, it's clear that we have a bug. Can you see it? Check out the case where `min = 0`. In that case, I think we still want to parse `atMost 9`. This makes sense since a `0` for `min` means that we want `0` required and `9` for `max` means we want optionally no more than `9`.

So we need to change our code:

```
range :: ∀ e a. Int -> Int -> Parser e a -> Parser e (Array a)
range min max p
  |    min < 0 ❶
    || max <= 0
    || max < min  = pure []
  | otherwise     = count min p
                      >>= \cs -> (cs <> _) <$> atMost (:) (max - min) p
```

❶ Changed `<=` to `<`.

Notice how we still treat `max = 0` as an error case. I think this is a reasonable requirement.

Make sure your code matches the above code before moving on.

Now, we should generalize `range`. First, we'll look to `atMost's` Type Signature, before and after, to give us a clue as to how we've generalized in the past:

```
atMost :: ∀ e a. Int -> Parser e a -> Parser e (Array a)

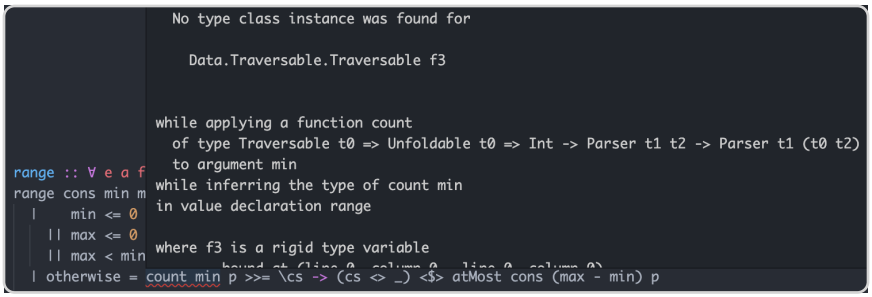
atMost
  :: ∀ e a f
  .  Unfoldable f
  => (a -> f a -> f a)
  -> Int
  -> Parser e a
  -> Parser e (f a)
```

Notice, that we changed `Array` to an `Unfoldable`, i.e. `f`, and we added a cons-like Function as its first Parameter. Go ahead and make similar changes to `range` before reading on. Expect compiler errors.

Here are those same changes, which results in a compiler error:

```
range :: ∀ e a f
  .  Unfoldable f
  => (a -> f a -> f a)
  -> Int
  -> Int
  -> Parser e a
  -> Parser e (f a)
range cons min max p
  |    min < 0
    || max <= 0
    || max < min  = pure none
  | otherwise = count min p -- COMPILER ERROR!!
                >>= \cs -> (cs <> _) <$> atMost cons (max - min) p
```

Here's the error:



We missed the `Traversable` requirement. That's because we used `atMost` as our Type Signature example and it doesn't have this Constraint, but `count` does. Maybe we should have used that instead:

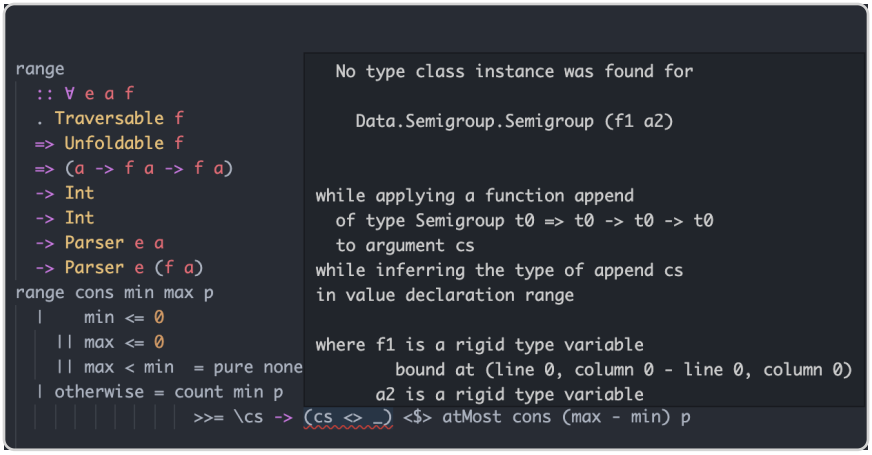
```
count :: ∀ e a f
  .  Traversable f
  => Unfoldable f
  => Int
  -> Parser e a
  -> Parser e (f a)
```

So we'll add this missing Constraint:

```
range :: ∀ e a f
  .  Traversable f ❶
  => Unfoldable f
  => (a -> f a -> f a)
  -> Int
  -> Int
  -> Parser e a
  -> Parser e (f a)
range cons min max p
  |    min < 0
    || max <= 0
    || max < min  = pure none
  | otherwise = count min p
                  >>= \cs -> (cs <> _) <$> atMost cons (max - min) p
                  -- COMPILER ERROR!!
```

❶ Added this Constraint.

Now we're getting a new compiler error:



The compiler is complaining that `f1 a2` isn't constrained to `Semigroup`. You can see by the red underline that we've used `<>`. This is why it's complaining. When we were using a Monomorphic Type, viz. `Array`, the compiler knew that `Array` has a `Semigroup` instance. But now we're using a Polymorphic Type, viz. `f`, and so it has no such knowledge.

So, go ahead and add that too and you should now have:

```
range
  :: ∀ e a f
  .  Semigroup (f a) ❶
  => Traversable f
  => Unfoldable f
  => (a -> f a -> f a)
  -> Int
  -> Int
  -> Parser e a
  -> Parser e (f a)
range cons min max p
  |    min < 0
    || max <= 0
    || max < min  = pure none
  | otherwise = count min p
                >>= \cs -> (cs <> _) <$> atMost cons (max - min) p
```

❶ Added this Constraint.

And finally, the compiler is silent. So make sure your code matches not that we've fixed all the problems.

Now write `range'` that returns a `Parser e String`. Look at back to see how we've done this before.

Code it up before continuing further.

Here's the Type Signature:

```
range' :: ∀ e. Int -> Int -> Parser e Char -> Parser e String
```

Notice that we don't need the `(a -> f a -> f a)` Parameter, i.e. the cons-like Function, we can simply pass `:` to `range` for the `cons` Function.

Now, for the implementation:

```
range' :: ∀ e. Int -> Int -> Parser e Char -> Parser e String
range' min max p = fromCharArray <$> range (:) min max p
```

Now we've implemented enough `Parsers` that we can handle the following:

```
\d{1,2}     -- range
\d{4}       -- count
-           -- satisfy
```

Now we finally have enough to be able parse dates.

The first `Parser` we're going to write is for dates that are formated year first, e.g. `1962-10-02` or `1962-10-2`. We'll call it `yearFirst`.

Before we write the Type Signature, let's review our date Types:

```
newtype Year = Year Int
newtype Month = Month Int
newtype Day = Day Int

data DateFormat
  = YearFirst
  | MonthFirst

type DateParts =
  { year :: Year
  , month :: Month
  , day :: Day
  , format :: DateFormat
  }
```

You should have enough now to write the Type Signature for `yearFirst`. Go ahead and write the Type Signature before reading further.

Here's the Type Signature:

```
yearFirst :: ∀ e. ParserError e => Parser e DateParts
```

We're using `e` and constraining it to `ParserError` since this `Parser` can fail.

This Function would probably be in a higher-level library that depends on our base parsing library. But we're just going to write all this code in the same file since we're just trying to learn how to write these.

It might be worth it to break up the Functions we've written once we're all done. The `Parser` Type along with `char`, `count`, `atMost` and similar Parsers would be in a low-level parser library. Then `yearFirst` and `monthFirst` (one we write it) would be in a higher-level `Date` parsing library. And finally, our Application code, which includes our Application ParserError Type, i.e. `PError`.

Next, write the implementation for `yearFirst` remembering to use `count'`, `range'` and `satisfy'` for the specific components as mentioned earlier. We're using the `String` versions of our `Parsers` since we're parsing `Strings`. Also, don't forget that we wrote a `Parser` to parse a single digit called `digit` that you'll want to use.

By the way, you could write this Function using the `Applicative` only approach, i.e. by NOT using `bind`, but trust me, writing it using `do`, which uses `bind` under the covers, will make your code so much easier to reason about.

Go ahead and give it shot then read on to see how you did.

If you got stuck, please only read through the code until you get past your roadblock then stop and try to complete this Function. You'll get so much more out of it.

First, we'll add the first line of the implementation:

```
yearFirst :: ∀ e. ParserError e => Parser e DateParts
yearFirst = do
  ???
```

Now we want to parse the year, which is 4 digits:

```
yearFirst :: ∀ e. ParserError e => Parser e DateParts
yearFirst = do
  year <- count' 4 digit ❶
  ???
```

❶ We're using `count'` so we'll get `Strings`.

But what Type is `year`? Well, to answer that, we'll should ask what Type is `count 4 digit`?

Looks like it's time to write out a few Types:

```
count' 4 digit :: Parser e String
year :: String ❶
```

❶ Remember, `<-` extracts the `a` in the `Monad`, which, for `Parser`, is the second Type Parameter, which, in this example, is `String`.

Since `year` is a `String`, we're going to have to convert it to an `Int`. We can do that with `fromString` from `Data.Int`.

If you don't already know about this Function and need you to look for it, but you don't know what it's called, you can use `Pursuit`, and search for the Type Signature,`String -> Maybe Int`. Don't forget the `Maybe` since NOT all `Strings` are valid `Ints`. Also, make sure that you use the proper case for the Type Signature.

Let's use `fromString` and figure out the new Type of `year`:

```
import Data.Int (fromString) ❶

yearFirst :: ∀ e. ParserError e => Parser e DateParts
yearFirst = do
  year <- fromString <$> count' 4 digit ❷
  ???
```

❶ Don't forget to import `fromString`.

❷ We map `fromString` over the `Parser` to affect the `Parser's` return Type.

Now what Type is `year`?

To answer that, you have to take into account the Type of `fromString`:

```
fromString :: String -> Maybe Int
```

Before we used `fromString` it had a Type of `String`, but now it has a Type of `Maybe Int`. Still not what we want.

It's very unfortunate that we have to deal with a `Maybe` since we know that we'll always have a `String` that will always be a valid `Int`. We know this for 2 reasons. First, the `Parser` we used guarantees this, i.e. `digit`. And second, the `String` is only 4 digits long, so there's no way that we'd overflow `Int`. Not even close.

So, we're going to use `fromMaybe` with a `0` as the default since we know the default will NEVER be used. By the way, if it is then we'll need to fix `digit`.

Let's add `fromMaybe 0`:

```
import Data.Int (fromString)
import Data.Maybe (Maybe(..), fromMaybe) ❶

yearFirst :: ∀ e. ParserError e => Parser e DateParts
yearFirst = do
  year <- fromMaybe 0 <<< fromString <$> count' 4 digit ❷
  ???
```

❶ Added `fromMaybe` to the import list.

❷ We compose our Functions before mapping and thanks to `Functor's` Composition Law, i.e. `map (g <<< f) = map g <<< map f`, it's equivalent.

Now what Type is `year`?

Here's the Type Signature of `fromMaybe` to help you answer that question:

```
fromMaybe :: ∀ a. a -> Maybe a -> a
```

Since the first Parameter passed to `fromMaybe` in our code is an `Int`, `year` must be an `Int`.

We're close to what we want but we're not quite there. Let's look at `DateParts`, the Type our `Parser` returns:

```
newtype Year = Year Int
newtype Month = Month Int
newtype Day = Day Int

type DateParts =
  { year :: Year ❶
  , month :: Month
  , day :: Day
  , format :: DateFormat
  }
```

❶ The Type for this `year` is `Year`.

We can use the `Year` Data Constructor make our `Int` a `Year` so that it matches the Type we'll need to construct `DateParts`:

```
yearFirst :: ∀ e. ParserError e => Parser e DateParts
yearFirst = do
  year <- Year <<< fromMaybe 0 <<< fromString <$> count' 4 digit
  ???
```

If you feel like you can continue coding the rest of the `yearFirst` at this point, then stop reading here and see how far you get. Once you're done or you hit your next brick wall, come back and see how to keep going.

The next step in the process is to parse the character, `-`. We can use `satisfy` to accomplish that. Here's `satify` for reference:

```
satisfy
  :: ∀ e
  .  ParserError e
  => String
  -> (Char -> Boolean)
  -> Parser e Char
satisfy expected pred =
  char >>= \c -> if pred c then pure c else fail $ invalidChar expected
```

`satisfy` takes 2 Parameters. The first Parameter is a `String` that tells us what character this `Parser` expected in the case of failure. The second is a Predicate to make sure that the character is correct.

If you got stuck here, it might be worth it to stop and see if you can write the next line.

Here's the next line:

```
yearFirst :: ∀ e. ParserError e => Parser e DateParts
yearFirst = do
  year <- Year <<< fromMaybe 0 <<< fromString <$> count' 4 digit
  void $ satisfy "-" (_ == '-') ❶ ❷ ❸
  ???
```

❶ Using `void` to discard the `Char` that the `Parser` returns. Since it's always `-`, we don't need it in a Variable.

❷ The Predicate is the wildcard Function, `(_ == '-')`, that returns `true` if the character equals `-`.

❸ Notice the Single Quote to define a character literal.

That line of code sucks. It's ugly and we're going to need to use this ugly line of code over and over again. Let's fix that by encapsulating this mess in a helper Function.

Write a `Parser` called `constChar` that lets us factor this ugly line of code out. Here's what I want to be able to write:

```
yearFirst :: ∀ e. ParserError e => Parser e DateParts
yearFirst = do
  year <- Year <<< fromMaybe 0 <<< fromString <$> count' 4 digit
  constChar '-' ❶
  ???
```

❶ This looks so much better.

Try to write `constChar` before continuing.

We'll start coding `constChar` by writing its Type Signature:

```
constChar :: ∀ e. ParserError e => Char -> Parser e Char ❶ ❷
```

❶ We constrain `e` to be a `ParserError` since we'll eventually call `satisfy`, which also has this Constraint, since it can fail.

❷ The first Parameter is the character that we're expecting to parse.

Next, we'll add left-hand side of the implementation:

```
constChar :: ∀ e. ParserError e => Char -> Parser e Char
constChar c = ???
```

Next, we'll delegate to `satisfy`:

```
constChar :: ∀ e. ParserError e => Char -> Parser e Char
constChar c = satisfy ??? (_ == c) ❶
```

❶ Need to replace `???` with `c` that's been converted to a `String`.

The problem with the first Parameter to `satisfy` is that it wants a `String` that will tell us what was expected if the Predicate fails.

How do you take a `Char` and convert it to a `String`. We could put it in an `Array` and then call `fromCharArray`. But that just feels like overkill.

Stop reading and go to `Pursuit` and search for the Type Signature `Char -> String`.

When I did this, I found `singleton` from `Data.String.CodeUnits`, which fits our needs perfectly. We'll add it:

```
import Data.String.CodeUnits (uncons, fromCharArray, singleton) ❶

constChar :: ∀ e. ParserError e => Char -> Parser e Char
constChar c = satisfy (singleton c) (_ == c)
```

❶ Add `singleton` to the import list.

Make sure you have this implementation of `constChar` in your code.

Let's return to `yearFirst` and use `constChar`:

```
yearFirst :: ∀ e. ParserError e => Parser e DateParts
yearFirst = do
  year <- Year <<< fromMaybe 0 <<< fromString <$> count' 4 digit
  constChar '-'
  ???
```

Do you think you can finish this? If so, stop and see how far you can get.

The next thing we'll need to parse is the month. We're going to use `range'` to do that:

```
yearFirst :: ∀ e. ParserError e => Parser e DateParts
yearFirst = do
  year <- Year <<< fromMaybe 0 <<< fromString <$> count' 4 digit
  constChar '-'
  month <- Month <<< fromMaybe 0 <<< fromString <$> range' 1 2 digit
  ???
```

Now, I'm hating the duplicate code. Let's stop and factor it out. By the way, there would be nothing wrong with finishing this Function and then factoring this out. I like to factor things out as soon as I see them, otherwise I risk forgetting. Granted, stopping in the middle of coding can feel a bit jarring, but I think you'll see that it'll be the worth the effort.

So, go ahead and write a Function that factors out the `fromMaybe` and `fromString` part that we're forced to repeat. Call the Function `digitsToNum`. Then read on.

The Type Signature for `digitsToNum`:

```
digitsToNum :: String -> Int
```

Remember, we "know" that we have a `String` of `digits` since our `Parsers` are using `digit`. If we were putting this in a library, I'd suggest documenting the limited usefulness of this Function with the

appropriate caveats. In other words, user beware.

Here's the left-hand side of the implementation:

```
digitsToNum :: String -> Int
digitsToNum s = ???
```

When we look back to what we did in `yearFirst`, we can see that we used Function Composition. So, let's write this Point-free:

```
digitsToNum :: String -> Int
digitsToNum = ??? ❶
```

❶ Removed the `s` Parameter since we're going Point-free.

And now, we'll just cut and paste the composition of `fromMaybe` and `fromString` and put it on the right-hand side:

```
digitsToNum :: String -> Int
digitsToNum = fromMaybe 0 <<< fromString
```

And now we can use this in `yearFirst`:

```
yearFirst :: ∀ e. ParserError e => Parser e DateParts
yearFirst = do
  year <- Year <<< digitsToNum <$> count' 4 digit
  constChar '-'
  month <- Month <<< digitsToNum <$> range' 1 2 digit
  ???
```

Do you think you can finish this now? Give a try before moving forward.

The next step is to parse the second – character:

```
yearFirst :: ∀ e. ParserError e => Parser e DateParts
yearFirst = do
  year <- Year <<< digitsToNum <$> count' 4 digit
  constChar '-'
  month <- Month <<< digitsToNum <$> range' 1 2 digit
  constChar '-'
  ???
```

And the last thing to parse is the day:

```
yearFirst :: ∀ e. ParserError e => Parser e DateParts
yearFirst = do
  year <- Year <<< digitsToNum <$> count' 4 digit
  constChar '-'
  month <- Month <<< digitsToNum <$> range' 1 2 digit
  constChar '-'
  day <- Day <<< digitsToNum <$> range' 1 2 digit
  ???
```

And the final step is to return a `DateParts`. Give this last part a try, but here's a hint, don't forget to use `pure`.

Here's the final line:

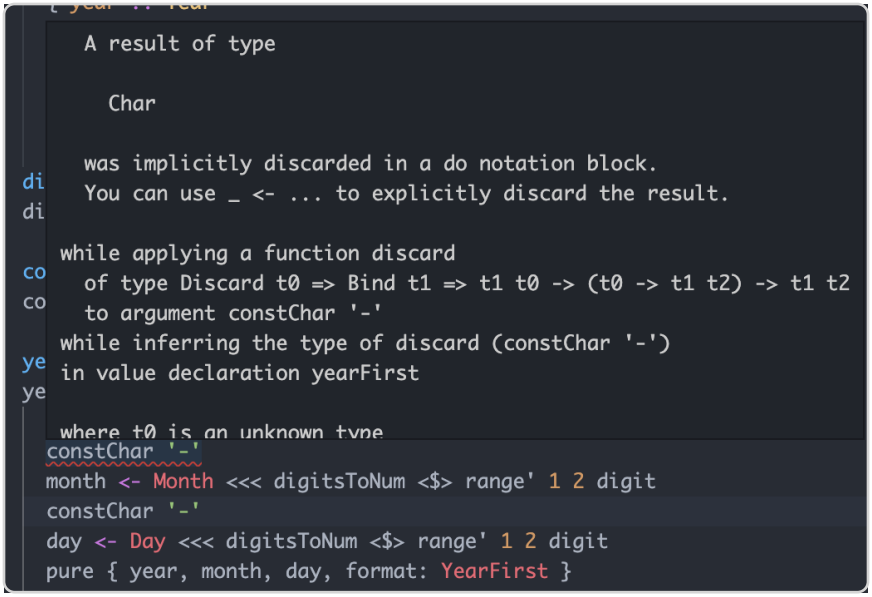
```
yearFirst :: ∀ e. ParserError e => Parser e DateParts
yearFirst = do
  year <- Year <<< digitsToNum <$> count' 4 digit
  constChar '-'                                  -- COMPILER ERROR!!
  month <- Month <<< digitsToNum <$> range' 1 2 digit
  constChar '-'
  day <- Day <<< digitsToNum <$> range' 1 2 digit
  pure { year, month, day, format: YearFirst }
```

There's a lot going on in the last line. First, we use `pure` since we must return a `Parser`.

Second, we don't have to specify `year: year`. We're using a short-cut. If the field name matches a Variable, then we can just specify the field name. Notice how this is NOT the case with the `format` field.

And third, we are setting `format` as `YearFirst`.

But, we have a compiler error:



Can you see what we forgot?

Here's a hint. We replaced `void $ satisfy "-" (_ == '-')` with `constChar '-'` and here's the definition of `constChar`:

```
constChar :: ∀ e. ParserError e => Char -> Parser e Char ❶
constChar c = satisfy (singleton c) (_ == c)
```

❶ `constChar` parses a `Char`.

Can you see the problem? If not, maybe looking at the following code will help:

```
c <- constChar '-'
```

The Type for `c` is `Char`. This is because `constChar` returns `Parser e Char`. `Char` is not discardable, i.e. it doesn't have a `Discard` Instance. We can, however, turn `constChar` into a `Parser e Unit` by using `void`:

```
constChar :: ∀ e. ParserError e => Char -> Parser e Unit ❶
constChar c = void $ satisfy (singleton c) (_ == c) ❷
```

❶ `Char` was changed to `Unit`.

❷ `void` was added. It takes an `m a -> m Unit`. Remember, `m`, is `Parser e`. So, its Type Signature specialized for `Parser` is `Parser e Char -> Parser e Unit`.

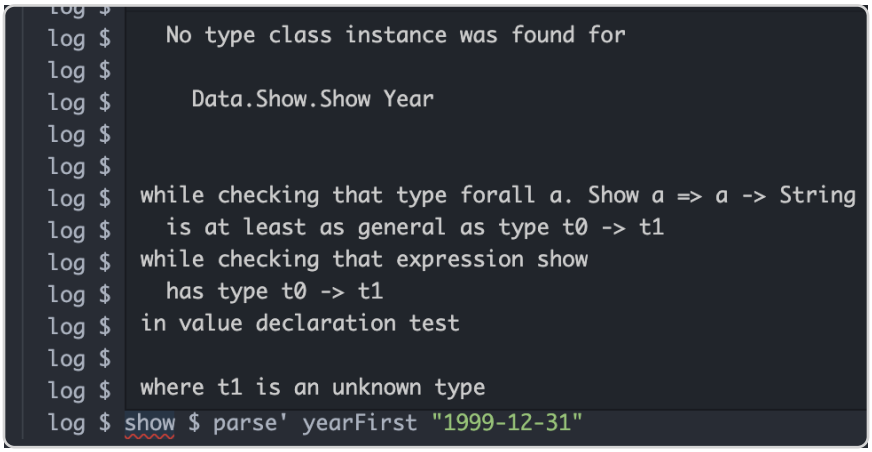
Make sure to make these changes to your code. And if you did everything right, it should compile.

Now, let's add some test code to test our newly minted `Parser`:

```
log $ show $ parse' yearFirst "1999-12-31" -- COMPILER ERROR!! ❶
```

❶ Once it compiles, it should print `(Right (Tuple "" { day: (Day 31), format: YearFirst, month: (Month 12), year: (Year 1999) }))`. Just a point of interest, the fields are printed in alpha order.

And, of course, we have a compiler error:



Well, from this, it's clear that we're missing a `Show` Instance for `Year`. Please remedy this and add one for `Month`, `Day` and `DateFormat` while your at it.

Stop reading here and return when the compiler error is gone or you run into issues and need help.

Here are the changes to make the date Types showable:

```
newtype Year = Year Int
derive instance genericYear :: Generic Year _

instance showYear :: Show Year where
  show = genericShow

newtype Month = Month Int
derive instance genericMonth :: Generic Month _

instance showMonth :: Show Month where
  show = genericShow

newtype Day = Day Int
derive instance genericDay :: Generic Day _

instance showDay :: Show Day where
  show = genericShow

data DateFormat
  = YearFirst
  | MonthFirst
derive instance genericDateFormat :: Generic DateFormat _

instance showDateFormat :: Show DateFormat where
  show = genericShow
```

Notice that we didn't have to make `DateParts` showable. That's because `Record` is a Primitive Type and because of that, it's showable as long as each of its fields are showable.

Now's a good time to test. Run the program and see if it outputs what you'd expect.

At this point, you should be able to write all of the month first `Parser`. Give it a try and call it `monthFirst`. Use `yearFirst` as a model for how to code it. Once it's complete, continue reading.

We can simply rearrange the lines of code in `yearFirst`, change the constant character from `-` to `/` and change the `format` to `MonthFirst`:

```
monthFirst :: ∀ e. ParserError e => Parser e DateParts
monthFirst = do
  month <- Month <<< digitsToNum <$> range' 1 2 digit ❶
  constChar '/' ❷
  day <- Day <<< digitsToNum <$> range' 1 2 digit ❸
  constChar '/' ❷
  year <- Year <<< digitsToNum <$> count' 4 digit ❹
  pure { year, month, day, format: MonthFirst } ❺
```

❶ Moved month parsing first.

❷ Parse `/` instead of `-`.

❸ Moved day parsing second.

❹ Moved year parsing last.

❺ Changed format to `MonthFirst`. This change is easy to forget if you copy/pasted the code from `yearFirst`.

Now, let's add some test code to test `monthFirst`

```
log $ show $ parse' monthFirst "12/31/1999" ❶
```

❶ Prints `(Right (Tuple "" { day: (Day 31), format: YearFirst, month: (Month 12), year: (Year 1999) }))`.

Once you've run the test for `monthFirst` and it works, write a `Parser` called `date` that parses dates in any support format, i.e. `YearFirst` and `MonthFirst`. Then read ahead to see how you did.

The Type Signature for `date`:

```
date :: ∀ e. ParserError e => Parser e DateParts
```

And now we'll just try both `Parsers`:

```
date :: ∀ e. ParserError e => Parser e DateParts
date = yearFirst <|> monthFirst ❶
```

❶ If we were to support more formats, we'd simply add them here.

So far, we've modeled a parser and called it `Parser`. We also have a simple Model for parse errors with a Typeclass called `ParserError`.

We have also created what are called ***Parser Combinators***. Parser Combinators are High-order Functions

that accept one or more parsers and produce a new parser. Typically, as their name suggests, the parsers that are passed in are combined in some way to produce the new parser.

The combinators that we've created include `satisfy`, `optional`, `count`, `range`, `atMost`, etc. We can image that we'd put all of these in a separate module in our parser library.

And we now have a `Parser` that can parse dates in different formats.

Overall, we have a platform for quickly creating new parsers that are far easier to read than `\d{1,2}\/\d{1,2}\/\d{4}`.

But even after all of that, we still don't have a way to parse what the following Regular Expression does:

```
\d*[A-Za-z]+
```

This will parse Zero or More digits and then One or More alpha characters. But our Combinators do not support this.

We'll implement Parser Combinators that will allow us to do this soon enough, but before we do, let's step back and take inventory.

19.8. How Does `Parser` Actually Work

We've been using our `Parser` for a bit now and it might still seem like magic. So, I'd like to look at what's really going on under the covers one more time to, hopefully, dispel that myth.

Let's start with `map`:

```
instance functorParser :: Functor (Parser e) where
  map f p = Parser \s -> map f <$> parse p s
```

And let's compare it with `map` from `Maybe`:

```
instance functorMaybe :: Functor Maybe where
  map _ Nothing   = Nothing
  map f (Just x)  = Just $ f x
```

In the case of `Maybe`, the mapping Function, `f`, is called IMMEDIATELY on `x`.

In the case of `Parser`, the mapping Function, `f`, is DEFERRED, i.e. it will not be called until `parse` is called. Calling `parse` causes the `Parser's` parsing Function to be called, which is when the mapping Function will be used to map the parsed result from one Type to another.

We can generalize this by stating that Functors of Values will apply mapping Functions IMMEDIATELY, whereas Functors of Functions will DEFER their execution until the Functor is "run", e.g. runReader, runState or parse. We've recently seen mapping Functions deferred in Reader, State and now Parser.

Understanding this can help us think about the following code in 2 equally valid ways, abstractly or concretely:

```
twoChars :: ∀ e. Parser e (Tuple Char Char)
twoChars = Tuple <$> char <*> char
```

Thinking abstractly, we can say: Tuple is mapped over the first Parser, char, to produce a Parser of Type Parser e (a -> Tuple Char a). Then that Parser will be applied to the second Parser, char, to produce a Parser of Parser e (Tuple Char Char).

Thinking concretely, we can say: The result of running the first Parser, char, will be mapped over by the Tuple Data Constructor, which will be short 1 Parameter. This Value will supplied by running the second Parser, char, to produce a final result of Type Tuple Char Char.

Either way is good. But, the second way really explains what's going on under the hood and hopefully, lessens that feeling of magic or of confusion, which is caused by the fact that so much of the Parser code is hidden from our everyday view thanks to <$>, <*>, >>= and the do notation.

We have to keep in mind that we've written a lot of code that's hidden in those instance implementations.

Now, let's look at apply:

```
instance applyParser :: Apply (Parser e) where
  apply p1 p2 = Parser \s -> do
    Tuple s1 f <- parse p1 s
    Tuple s2 x <- parse p2 s1
    pure $ Tuple s2 $ f x
```

And let's compare it against Maybe:

```
instance applyMaybe :: Apply Maybe where
  apply Nothing  _  = Nothing
  apply (Just f) x  = f <$> x
```

Once again, we can see that Maybe's implementation applies the mapping Function immediately, whereas, in Parser, the Function is deferred until the Parser is executed.

This is why we think of these mapping Functions as operating on what the Parser ***returns***, i.e. if you have a Parser e (Array Char) and you map fromCharArray over it, you get a Parser that returns a String.

`fromCharArray` will operator NOT on the `Parser` Function itself, but instead, will be deferred and used when the `Parser` is run to map over the `Parser's` results.

And finally, we'll look at `bind`:

```
instance bindParser :: Bind (Parser e) where
  bind p f = Parser \s -> do
    Tuple s1 x <- parse p s
    parse (f x) s1
```

And for contrast, here's `Maybe's` implementation:

```
instance bindMaybe :: Bind Maybe where
  bind Nothing _  = Nothing
  bind (Just x) f = f x
```

As we've seen before, `Maybe` calls the Monadic Function, `f`, immediately on its Value, but `Parser` defers that call until the `Parser` is run.

When we see `bind` used with `Parsers`, we think more explicitly about the Values returned by running the `Parser`. Take the following example:

```
twoCharsB :: ∀ e. Parser e (Tuple Char Char)
twoCharsB = char >>= \c1 -> char >>= \c2 -> pure $ Tuple c1 c2
```

In this version of `twoChars` that uses `bind`, we capture the character that `char` parses in the first Parameter to the Lambda Function, i.e. `c1`. Then we do it again with `c2`. Concrete thinking is a bit more out in the open with `bind`, since we are responsible for capturing its results.

Hiding the gory details from our everyday view is a real benefit because it greatly reduces our daily cognitive overhead. But, the downside is that we can easily be confused and have a feeling that we're missing something.

One thing that all magic tricks have in common is that something has to happen out of our view. But when the trick is performed with nothing concealed, we can see how the trick works and how we were once fooled. Now, when we see that same trick performed with the hidden elements hidden, we're no longer amazed by it because it's lost all of its mystery.

While this can be sad, if you really enjoy magic, it's a huge win for us when it comes to understanding how our `Functors`, `Applicatives` and `Monads` actually work.

19.9. some and many Combinators

There are 2 Parser Combinators that are commonly used in Parsing Libraries, `some` and `many`. Here are their definitions:

```
some = 1 or more (+ in Regular Expressions)
many = 0 or more (* in Regular Expressions)
```

We can define them slightly differently:

```
some = one and many
many = some or none
```

`some` is one successful parse and possibly more using `many`.

`many` is successfully parsing `some` or `none`.

The problem here is that they're defined in terms of each other, i.e. they are ***Mutually Recursive***. This means that when we call `many` it will call `some` which will call `many` which will call `some`, etc. until the stack overflows.

In a lazy language like Haskell, this is not a problem. But PureScript is a strict language, which means it'll eagerly try to evaluate all Parameter Values to a Function. This causes infinite loops.

Let's look at a simple example of how that could happen by looking at a program that can crash:

```
import Data.Maybe (fromMaybe)
import Partial.Unsafe (unsafeCrashWith) ❶

test :: Int
test = Just 10 # fromMaybe (unsafeCrashWith "Got a Nothing") ❷

main :: Effect Unit
main = do
  log $ show test
```

❶ We need to import a crash Function.

❷ Here we only want to crash if we have a `Nothing`.

When we run this code, and `test` is called, the first Parameter to `fromMaybe` will be evaluated. This Parameter is the result from calling `unsafeCrashWith`, which will crash our program.

This evaluation will happen even though `fromMaybe` doesn't need this Value, since we're passing it `Just`

10. But the compiler doesn't know any of that. All it knows is that there's a Parameter that is the result of a Function call and that Function needs to be called first, so that the result can be passed to fromMaybe.

The way to fix this is the following:

```
import Data.Maybe (fromMaybe') ❶
import Partial.Unsafe (unsafeCrashWith)

test :: Int
test = Just 10 # fromMaybe' \_ -> unsafeCrashWith "Got a Nothing" ❷

main :: Effect Unit
main = do
  log $ show test
```

❶ Using fromMaybe' instead (note the prime).

❷ Passing a Function to fromMaybe' as opposed to a Value like with fromMaybe.

Notice that we solve this problem by passing a Function into fromMaybe'. The compiler can safely send this Function as the first Parameter to fromMaybe' as it is. There's no evaluation necessary, since it's just a Function.

We can say that fromMaybe' is the lazy version of fromMaybe. Let's compare their Type Signatures:

```
fromMaybe :: ∀ a. a -> Maybe a -> a
fromMaybe' :: ∀ a. (Unit -> a) -> Maybe a -> a
```

Notice that fromMaybe' gets a Function that takes a Unit as it's first Parameter. This Function returns the Value of a ONLY when it's called. In our case, that Value is unsafeCrashWith "Got a Nothing", which will only crash our program if we pass fromMaybe' a Nothing.

We can say that Lazy Evalutation ***defers*** the evaluation of the Value.

PureScript has some facilities like fromMaybe' to defer evaluations. There are 2 modules, Control.Lazy and Data.Lazy.

Data.Lazy is useful for evaluating lazily-computed Values of a. This is used heavily in Data.List.Lazy, which implements lazy linked lists.

Control.Lazy is for Types that contain a Function, like our Parser does. There's a Typeclass called Lazy to aid in the kinds of problems we noted above with some and many, which we'll soon encounter.

Here's the Lazy Typeclass from Control.Lazy:

```
class Lazy l where
  defer :: (Unit -> l) -> l
```

Notice the telltale signs of Lazy Evaluation in the Type Signature of `defer`. The first Parameter is a Function that takes `Unit` as its first Parameter. This is just like what we saw in the Type Signature of `fromMaybe'`, i.e. the lazy version of `fromMaybe`.

If we make our `Parser` an instance of `Lazy`, then we'll be able to use `defer` in places where calling another `Parser` would cause an infinite loop. This will make more sense once we see the code for `some` and `many`. The important point here is that we're going to make `Parser` an instance of `Lazy` so we can defer calling the actual `Parser` we want to use, since that `Parser` would cause an infinite loop.

In this section, we'll code together. This will help things move quickly along as we develop these combinators.

Let's start with `some`. Here are the definitions again for reference:

```
some = one and many
many = some or none
```

So we should keep in mind that `some` will parse once and then call `many`.

As usually, we'll start coding with concrete Types and generalize later. Here's `some's` Type Signature:

```
some :: ∀ e a. Parser e a -> Parser e (Array a)
```

Next, we'll code the left-hand side of the equal sign:

```
some :: ∀ e a. Parser e a -> Parser e (Array a)
some p = ???
```

Now we want to call the `Parser`, `p`, once:

```
some :: ∀ e a. Parser e a -> Parser e (Array a)
some p = p ???
```

And then we want to `Cons` the result of `p` onto the front of calling `many`:

```
some :: ∀ e a. Parser e a -> Parser e (Array a)
some p = (:) <$> p <*> many p
```

We first map the Cons Operator over the Parser, p, and then apply the many p Parser, which should contain an Array a. Well, it will once we write it.

This is a classic usage of Applicative Parsing. But as a review, let's break it down a bit by writing the Types:

```
(:) :: a -> Array a -> Array a
p :: Parser e a
(:) <$> p :: Parser e (Array a -> Array a) ❶
many p :: Parser e (Array a)
(:) <$> p <*> many p :: Parser e (Array a) ❷
```

❶ (:) get's its first Parameter from p, i.e. a Value of Type a, leaving a Function of Type Array a -> Array a in the resulting Parser.

❷ (:) <$> p get's its first Parameter from many p, i.e. a Value of Type Array a, leaving a Value of Type Array a in the resulting Parser.

Now, we're ready to write many. From the above definitions, many is a choice between some and none, which says that we'll be using the <|> Operator in the implementation. But first the Type Signature:

```
many :: ∀ e a. Parser e a -> Parser e (Array a)
```

And now the implementation:

```
many :: ∀ e a. Parser e a -> Parser e (Array a)
many p = some p <|> pure []
```

With many, either there are 1 or more characters (some) that can be parse or there are none, i.e. many is 0 or more.

We now have exactly what we expected, Mutual Recursion:

```
some :: ∀ e a. Parser e a -> Parser e (Array a)
some p = (:) <$> p <*> many p ❶

many :: ∀ e a. Parser e a -> Parser e (Array a)
many p = some p <|> pure [] ❷
```

❶ some calls many.

❷ many calls some.

We need to figure out where to break this loop. There are only 2 places we could break this loop, where we call many in some or where we call some in many.

Since it actually doesn't matter which we choose, we'll defer the many call in some because that's what is typically done in libraries that implement these Functions.

If we can replace the call to many p with an equivalent Parser whose only job is to defer the calling to many, then we will break the infinite loop.

We can accomplish that by making a Lazy Instance for our Parser. We'll do that one step at a time:

```
import Control.Lazy (class Lazy) ❶

instance lazyParser :: Lazy (Parser e a) where
```

❶ We'll need to import the Lazy Typeclass.

Notice we use Parser e a. That's because the l in Lazy's Typeclass definition has a Kind Signature of Type.

Next, we should write the Type Signature for defer, specialized for Parser e a, to help us code it:

```
instance lazyParser :: Lazy (Parser e a) where
  defer :: (Unit -> Parser e a) -> Parser e a ❶
  ???
```

❶ There is no need for ∀ e a here since both e and a are specified in the instance definition line which implies ∀.

Now, we'll write the left-hand side of defer:

```
instance lazyParser :: Lazy (Parser e a) where
  defer :: (Unit -> Parser e a) -> Parser e a
  defer f = ???
```

Now we have to return a Parser. There are 2 ways to get a Parser. I'm going to code the WRONG way first:

```
instance lazyParser :: Lazy (Parser e a) where
  defer :: (Unit -> Parser e a) -> Parser e a
  defer f = f unit -- WRONG!
```

This will compile but it won't defer the evaluation. This is because `f unit` gets evaluated immediately.

The way to get this to be deferred is to create a brand new `Parser` like we've done in the past:

```
instance lazyParser :: Lazy (Parser e a) where
  defer :: (Unit -> Parser e a) -> Parser e a
  defer f = Parser \s -> ???
```

Now, we'll get the Parser out of the Function, `f`, by passing it `unit`:

```
instance lazyParser :: Lazy (Parser e a) where
  defer :: (Unit -> Parser e a) -> Parser e a
  defer f = Parser \s -> f unit -- COMPILER ERROR!!
```

We're going to get a compiler error since we're not done yet. So, we'll ignore it.

Since `f unit` is a `Parser`, we'll need to run it by using `parse` along with the `String` to parse, i.e. `s`:

```
instance lazyParser :: Lazy (Parser e a) where
  defer :: (Unit -> Parser e a) -> Parser e a
  defer f = Parser \s -> parse (f unit) s
```

Let's step back for a second and appreciate what we've just built. We have a `Parser` that when run will execute another `Parser` and do nothing else. On the surface, the outer `Parser` seems redundant and it is except for the fact that it will DELAY the execution of the inner `Parser` until the outer `Parser` is EXECUTED.

What's great about the outer `Parser`, is that it's Functionally equivalent to the inner one, except that it's NOT identical to the inner one. That means that we can use it as a safe replacement for ANY `Parser` at any time or place and it won't affect the result of parsing.

But, the only real benefit to using this `Parser` proxy is when we can break a Mutual Recursive definition as we have with `some` and `many`.

By the way, if the `Lazy` Typeclass didn't exist, we could've just written a Parser Combinator called `defer` that would have accomplished the same objective:

```
defer :: ∀ e a. (Unit -> Parser e a) -> Parser e a
defer f = Parser \s -> parse (f unit) s
```

This isn't as general, and, therefore not as nice, as the `Lazy` Typeclass, but will work just the same.

Go ahead and add the following to your code:

```
import Control.Lazy (class Lazy)

instance lazyParser :: Lazy (Parser e a) where
  defer :: (Unit -> Parser e a) -> Parser e a
  defer f = Parser \s -> parse (f unit) s

some :: ∀ e a. Parser e a -> Parser e (Array a)
some p = (:) <$> p <*> many p

many :: ∀ e a. Parser e a -> Parser e (Array a)
many p = some p <|> pure []
```

Now, we want to defer the `many` call in the `some` implementation. We're going to do this by replacing it with a deferred `Parser`:

```
import Control.Lazy (class Lazy, defer) ❶

some :: ∀ e a. Parser e a -> Parser e (Array a)
some p = (:) <$> p <*> defer \_ -> many p ❷
```

❶ We need to add `defer` to the import list.

❷ `defer \_ -> many p` is our stand-in `Parser` that's Functionally equivalent to `many p`.

Make this change to your code.

Now, write `some'` and `many'` that return `Parser e String`. You can look back at how we did this with `count'` and `range'`. Do this before moving on.

Here's the implementations:

```
some' :: ∀ e. Parser e Char -> Parser e String ❶
some' p = fromCharArray <$> some p ❷

many' :: ∀ e. Parser e Char -> Parser e String ❶
many' p = fromCharArray <$> many p ❷
```

❶ a changed to Char and Array a changed to String.

❷ We map fromCharArray over the Parser to convert Array Char to String.

Now, we're ready to test out code. Let's hope we did the Lazy part right otherwise we'll get a stack overflow.

Add the following to your test code:

```
log $ show $ parse' (some' digit) "2343423423abc" ❶
log $ show $ parse' (many' digit) "_2343423423abc" ❷
log $ show $ parse' (some' digit) "_2343423423abc" ❸
```

❶ Prints (Right (Tuple "abc" "2343423423")).

❷ Prints (Right (Tuple "_2343423423abc" "")).

❸ Prints (Left (InvalidChar "digit")).

And run the test to see if you get the expected output.

Notice on the first line of our test code that some' consumes all digits that it can. That's because some and more are ***Greedy***, i.e. they will continue to consume until failure.

Also, notice that many' doesn't fail on "_2343423423ab" like some' does. That's because some' must parse at least 1, whereas many' can parse 0.

By the way, if you're like me, you may not see the logic to the names some and many as they relate to 1 or more and 0 or more. The way I remember the differences is that some rhymes with 1. As is too often in Math and Programming, these names are terrible.

At this point, we should return to some and many and make them more general like we did with so many of our other Parser Combinators. Let's do that together starting with some. Here's what we've got so far:

```
some :: ∀ e a. Parser e a -> Parser e (Array a)
some p = (:) <$> p <*> defer \_ -> many p
```

Let's remove Array and replace it with Unfoldable:

```
some :: ∀ e a f. Unfoldable f => Parser e a -> Parser e (f a) ❶ ❷ ❸
some p = (:) <$> p <*> defer \_ -> many p   -- COMPILER ERROR!!
```

❶ Added `f` to `∀`.

❷ Added `Unfoldable` Constraint on `f`.

❸ Replaced `Array` with `f`.

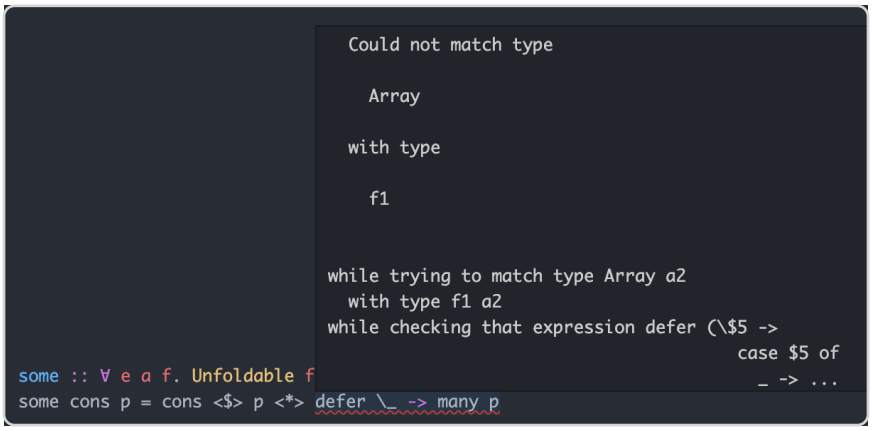
We've got an error because we haven't replaced `Array's Cons` Operator, `(:)`, yet.

Let's do that now:

```
some :: ∀ e a f
  .  Unfoldable f
  => (a -> f a -> f a) ❶
  -> Parser e a
  -> Parser e (f a)
some cons p = cons <$> p <*> defer \_ -> many p   -- COMPILER ERROR!!
```

❶ An addition Parameter was added for the cons-like Function.

The compiler is complaining about the following:



Notice that it's underlined the call to `many`. That' because we haven't changed `many`, yet.

Let's do that now:

```
many :: ∀ e a f
  .  Unfoldable f
  => (a -> f a -> f a) ❶
  -> Parser e a
  -> Parser e (f a)
many cons p = some cons p <|> pure [] -- COMPILER ERROR!! ❷
```

❶ Same changes we made to the Type Signature for `some`.

❷ The only reason we need the `cons` Parameter is because `some` needs it.

Now we have to replace `[]` with `none`:

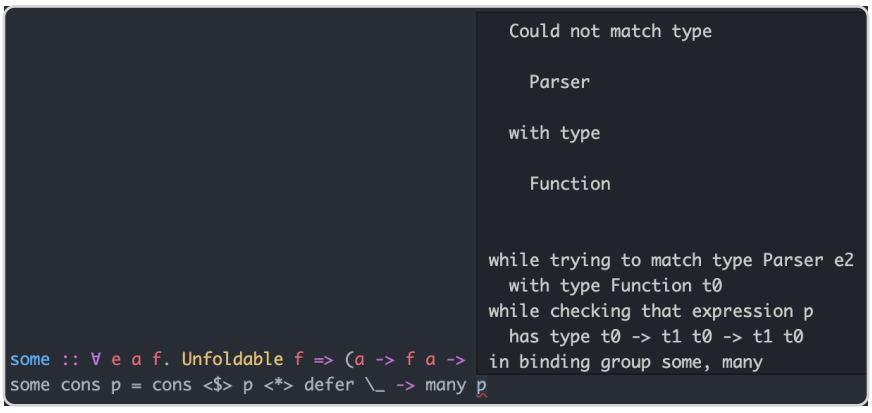
```
many :: ∀ e a f
  .  Unfoldable f
  => (a -> f a -> f a)
  -> Parser e a
  -> Parser e (f a)
many cons p = some cons p <|> pure none ❶
```

❶ Replaced `[]` with `none`.

Now `many` compiles but `some` has a new error:

```
some :: ∀ e a f
  .  Unfoldable f
  => (a -> f a -> f a)
  -> Parser e a
  -> Parser e (f a)
some cons p = cons <$> p <*> defer \_ -> many p -- COMPILER ERROR!!
```

Here's that error:



When the compiler looked at the first Parameter of our call to `many`, it found a `Parser`, `p`, but expected a `Function`. That's because we added the `cons` Parameter to `many` but forget to add it to our call.

Let's fix that now:

```
some :: ∀ e a f
  .  Unfoldable f
  => (a -> f a -> f a)
  -> Parser e a
  -> Parser e (f a)
some cons p = cons <$> p <*> defer \_ -> many cons p ❶
```

❶ Added `cons` as `many's` first Parameter.

Now, the compiler is making a similar complaint about `many'`, which we'll fix in a similar way:

```
many' :: ∀ e. Parser e Char -> Parser e String
many' p = fromCharArray <$> many (:) p ❶
```

❶ Added the `(:)` Operator as our cons-like Function.

We might as well fix `some'` before the compiler complains about it too:

```
some' :: ∀ e. Parser e Char -> Parser e String
some' p = fromCharArray <$> some (:) p ❶
```

❶ Added the `(:)` Operator as our cons-like Function.

At this point we could stop, but stepping back and looking over the code we've written, we can see that `some` will never return an empty `Unfoldable` since it's supposed to parse `1 or more`.

How can we change it to reflect this restriction?

Do you remember `NonEmpty`? Here's it's definition:

```
data NonEmpty f a = NonEmpty a (f a)
```

We can use this on `some` but NOT `many` since it can be empty, since `many` is `0 or more`. This means that when `many` calls `some`, we'll have to convert the `NonEmpty` that we get back to something that can be empty.

Okay, so one problem at a time. Let's first modify `some` to return `NonEmpty`:

```
import Data.NonEmpty (NonEmpty, (:|)) ❶

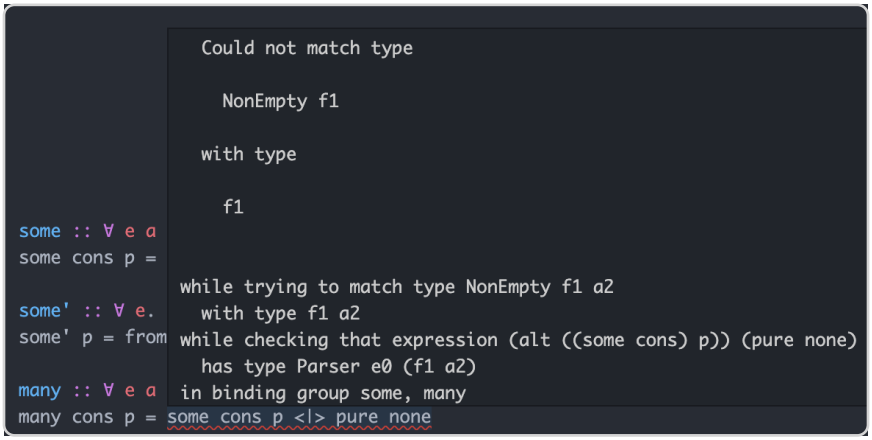
some :: ∀ e a f
  .  Unfoldable f
  => (a -> f a -> f a)
  -> Parser e a
  -> Parser e (NonEmpty f a) ❷
some cons p = (:|) <$> p <*> defer \_ -> many cons p ❸
```

❶ Added this import Statement.

❷ `f a` changed to `NonEmpty f a`.

❸ Replaced the first `cons` with the `NonEmpty` Cons Operator but left the second `cons` alone since `many` need to use it.

Now, the compiler is complaining about `many`:



From the error and red underline, it should be clear that it's complaining because we haven't converted `some's` return value, i.e. a `NonEmpty` to something that's NOT `NonEmpty`, which is what `many` returns.

We need a way to convert a `NonEmpty f a` to `f a`. Searching `Pursuit` for `NonEmpty f a -> f a` produces nothing useful. Searching `Pursuit` for `NonEmpty` displays all the `NonEmpty` Functions, which, if you browser through carefully, will reveal a possible candidate called `fromNonEmpty`.

Here's it's Type Signature:

```
fromNonEmpty :: ∀ a r. (a -> f a -> r) -> NonEmpty f a -> r
```

The Type Signature isn't what we searched for, but if we replace `r` with `f a` then it's exactly what we want. We just have to give it a `cons` Function, which, luckily, we have.

Let's use `fromNonEmpty` to convert the return Value from `some` in `many`:

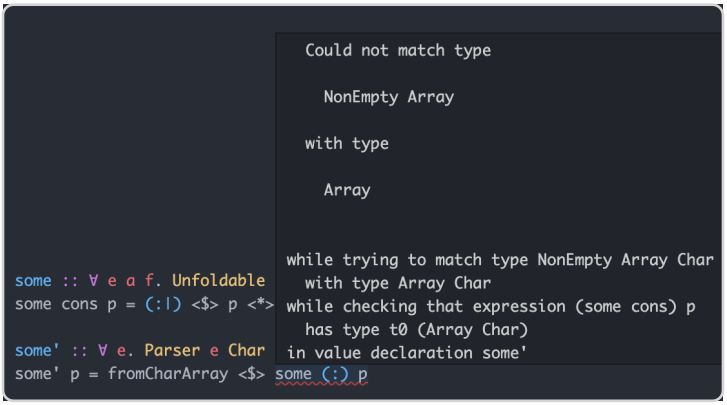
```
import Data.NonEmpty (NonEmpty, (:|), fromNonEmpty) ❶

many :: ∀ e a f
  .  Unfoldable f
  => (a -> f a -> f a)
  -> Parser e a
  -> Parser e (f a)
many cons p = fromNonEmpty cons <$> some cons p <|> pure none ❷
```

❶ Added `fromNonEmpty` to import list.

❷ We map `fromNonEmpty cons` over `some`.

And now, we get an error on `some'`:



The error says that it got a `NonEmpty Array` when it expected `Array`. Looks like we need to convert the return value here too:

```
some' :: ∀ e. Parser e Char -> Parser e String
some' p = fromCharArray <<< fromNonEmpty (:) <$> some (:) p ❶
```

❶ Composed `fromNonEmpty (:)` with `fromCharArray`, which gets mapped over `some's` return Type, i.e. `NonEmpty Array Char`.

And the compiler is finally quiet.

Here's our final generalized code:

```
import Data.NonEmpty (NonEmpty, fromNonEmpty, (:|))

some :: ∀ e a f
  . Unfoldable f
  => (a -> f a -> f a)
  -> Parser e a
  -> Parser e (NonEmpty f a)
some cons p = (:|) <$> p <*> defer \_ -> many cons p

some' :: ∀ e. Parser e Char -> Parser e String
some' p = fromCharArray <<< fromNonEmpty (:) <$> some (:) p

many :: ∀ e a f
  . Unfoldable f
  => (a -> f a -> f a)
  -> Parser e a
  -> Parser e (f a)
many cons p = fromNonEmpty cons <$> some cons p <|> pure none

many' :: ∀ e. Parser e Char -> Parser e String
many' p = fromCharArray <$> many (:) p
```

Please update your code to match this before moving on.

By the way, `some` and `many` have been implemented for `Arrays` and `Lists` in PureScript. Here's the Type Signature for them for `Arrays`:

```
some :: ∀ f a. Alternative f => Lazy (f (Array a)) => f a -> f (Array a)
many :: ∀ f a. Alternative f => Lazy (f (Array a)) => f a -> f (Array a)
```

These are more general than ours in one way but less in another. This is specifically for `Arrays`. It also has been written for `List` both normal and lazy. So ours is more general because we work with any `Unfoldable`.

But, unlike ours, these will work with ANY `Alternative` that has a `Lazy` Instance.

It turns out that we can add this flexibility. Let's change `Parser e` to be any `Monad`, `m`. Doing this will make these functions more flexible and allow us to use them with more Types than just our `Parser`:

```
some :: ∀ a f m  ❶ ❷
  .  Unfoldable f
  => (a -> f a -> f a)
  -> m a ❶
  -> m (NonEmpty f a) ❶
some cons p = (:|) <$> p <*> defer \_ -> many cons p -- COMPILER ERROR!!

many :: ∀ a f m ❶ ❷
  .  Unfoldable f
  => (a -> f a -> f a)
  -> m a ❶
  -> m (f a) ❶
many cons p = fromNonEmpty cons <$> some cons p <|> pure none
```

❶ `Parser e` was replaced by `m`.

❷ We also removed `e` from `∀`.

The choice to use `m` was because our `Parser e` is a `Monad`. We may never wind up needing to constrain `m` to be a `Monad`. We'll have to wait and see once we get it to compile.

Speaking of the compiler, we have a new compiler error:

```
    No type class instance was found for

      Control.Apply.Apply m3


  while applying a function apply
    of type Apply t0 => t0 (t1 -> t2) -> t0 t1 -> t0 t2
    to argument (map NonEmpty) p
  while inferring the type of apply ((map NonEmpty) p)
  in binding group some, many

  where m3 is a rigid type variable
          bound at (line 0, column 0 - line 0, column 0)
        t0 is an unknown type
some :: ∀ e a
some cons p = (:|) <$> p <*> defer \_ -> many cons p
```

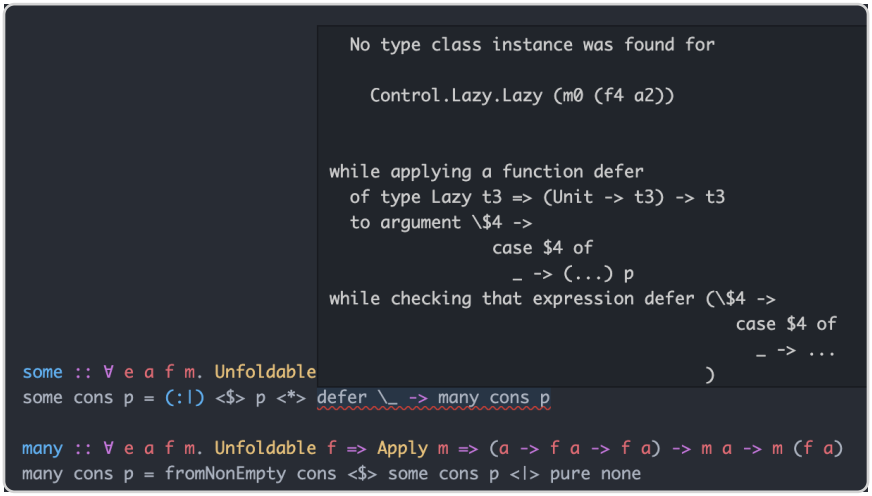
We need to add an `Apply` Constraint. That's because we're using `<*>`. Let's add an `Apply m` Constraint to both while we're at it:

```
some :: ∀ a f m
  .  Unfoldable f
  => Apply m ❶
  => (a -> f a -> f a)
  -> m a
  -> m (NonEmpty f a)
some cons p = (:|) <$> p <*> defer \_ -> many cons p -- COMPILER ERROR!!

many :: ∀ a f m
  .  Unfoldable f
  => Apply m ❶
  => (a -> f a -> f a)
  -> m a
  -> m (f a)
many cons p = fromNonEmpty cons <$> some cons p <|> pure none
```

❶ Added the `Apply` Constraint.

We fixed that but now we have a new error:



This error is because we are missing a `Lazy` Constraint. That's because we're using `defer`. The compiler gives us a clue how to write the Constraint with the Type it inferred, `Contol.Lazy.Lazy (m0 (f4 a2))`.

`m (f a)` is an `Apply` with an `Unfoldable` of `a`'s. An example of this is `Parser e (Array a)` where `m` is `Parser e` (remember we made this substitution earlier) and `f` is `Array`.

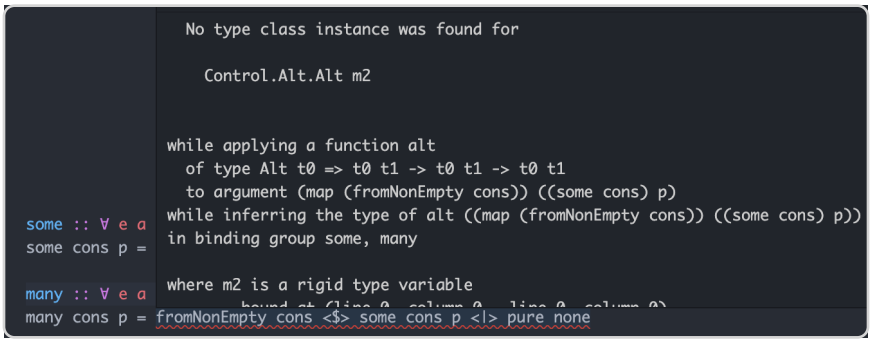
Let's add a `Lazy (m (f a))` Constraint to both, `some` and `many`:

```
some :: ∀ a f m
  .  Unfoldable f
  => Apply m
  => Lazy (m (f a)) ❶
  => (a -> f a -> f a)
  -> m a
  -> m (NonEmpty f a)
some cons p = (:|) <$> p <*> defer \_ -> many cons p

many :: ∀ a f m
  .  Unfoldable f
  => Apply m
  => Lazy (m (f a)) ❶
  => (a -> f a -> f a)
  -> m a
  -> m (f a)
many cons p = fromNonEmpty cons <$> some cons p <|> pure none
              -- COMPILER ERROR!!
```

❶ Added `Lazy` Constraint.

And now a new error:

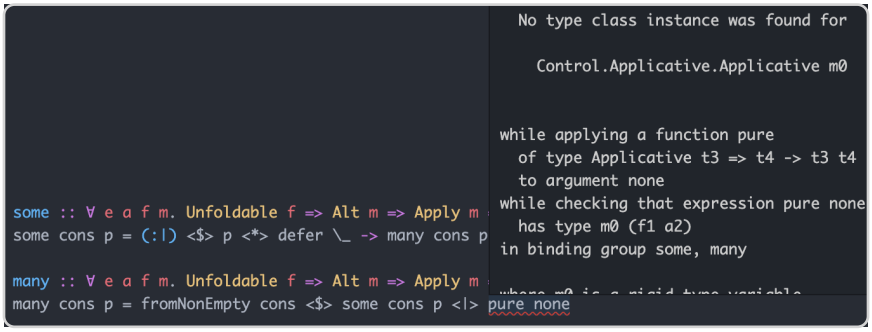


Since we're using `<|>`, we need to add an `Alt m` Constraint to both:

```
some :: ∀ a f m
  .  Unfoldable f
  => Alt m
  => Apply m
  => Lazy (m (f a))
  => (a -> f a -> f a)
  -> m a
  -> m (NonEmpty f a)
some cons p = (:|) <$> p <*> defer \_ -> many cons p

many :: ∀ a f m
  .  Unfoldable f
  => Alt m
  => Apply m
  => Lazy (m (f a))
  => (a -> f a -> f a)
  -> m a
  -> m (f a)
many cons p = fromNonEmpty cons <$> some cons p <|> pure none
              -- COMPILER ERROR!!
```

And yet another error:



Since we're using `pure`, we need an `Applicative m` Constraint. But wait. We already have an `Apply`, which is the Superclass for `Applicative`. So, we'll swap out `Apply` for `Applicative`:

```
some :: ∀ a f m
  .  Unfoldable f
  => Alt m
  => Applicative m ❶
  => Lazy (m (f a))
  => (a -> f a -> f a)
  -> m a
  -> m (NonEmpty f a)
some cons p = (:|) <$> p <*> defer \_ -> many cons p

many :: ∀ a f m
  .  Unfoldable f
  => Alt m
  => Applicative m ❶
  => Lazy (m (f a))
  => (a -> f a -> f a)
  -> m a
  -> m (f a)
many cons p = fromNonEmpty cons <$> some cons p <|> pure none
```

❶ Swapped out `Apply` for `Applicative`.

Finally, we have a happy compiler. Notice how we didn't need to constrain `m` to be a `Monad`. It only needs to be an `Applicative`.

Also, notice that we didn't need to change any of our code to make our Functions more general. In this final step at generalization, we only needed to change the Types from specific ones to more general ones. This won't always be the case, but many times we can generalize a Function by simply picking more general Types.

Now's a good time to make these changes to your code.

And now we can use these versions of `some` and `many` in myriad of situations, not just for `Parsers`. And we don't have to write multiple versions of these Functions for different Types, i.e. as long as they implement `Unfoldable`. It was definitely worth the effort to make these function general.

Notice that we got caught up in a tight loop of changing our code and getting an error then making another change and getting another error, and then changing our code again, etc. You'll find yourself playing this game in the hopes that your code will eventually compile.

This strategy doesn't always work. It was a successful here because we were just adding Constraints and changing Types. But typically that won't be the case and I'd advise against blindly making changes in the hopes that the error messages will just magically go away. It's always better to slow down and understand what's wrong first, and then change your code.

At this point, It's probably a good idea to run the tests again to make sure that we didn't break anything. Chances are pretty slim since we just changed Types and the compiler is our test for those kinds of changes.

Another point worth making is that our final Type Signatures look scary. But, we didn't just write these scary Type Signatures the first time. We initially wrote simple ones and as we generalized, we rewrote them over and over until they were more complex. But you saw the process. It's not magic. And it's not hard. It's just many incremental changes from a concrete version of the code to a very abstract version.

So, when you see these kinds of Type Signatures in the wild, don't panic. If there are Constraints that you don't know, then go learn them. And if there are Types you're not familiar with, then go become familiar. Because that's all it takes as you have now witnessed.

And because we understand both our Types and Classes we used in our Constraints, we can easily justify every part of our Type Signatures, for example, in `some`:

```
some :: ∀ a f m
  .  Unfoldable f          ❶
  => Alt m                 ❷
  => Applicative m         ❸
  => Lazy (m (f a))        ❹
  => (a -> f a -> f a)     ❺
  -> m a                   ❻
  -> m (NonEmpty f a)      ❼
```

❶ An `Unfoldable`. This is because we used `none` in `many`.

❷ Because we call `many` and it uses `<|>`.

❸ Because we call `many` and it uses `pure`.

❹ Because we need to break the infinite loop with `defer`.

❺ cons-like Function, which means we expect our `Unfoldable` to be a list-like structure.

❻ The original `Applicative` of `a's` that we're going to leverage to produce a new `Applicative`. With `Parsers`, this is a single `Parser` that we'll use over and over again as needed.

❼ Our new `Applicative` that produces a `NonEmpty` list-like structure. With `Parsers`, this is a `Parser` that will produce a list-like result when the `Parser` is run.

If we were to look upon this Type Signature for the first time, we might find it intimidating. And it will be if we don't understand `Unfoldable` or `Lazy`, for example. But since we do, we can look at each line one at a time and understand, albeit slowly, what this Type Signature means. And sometimes, we'll need to look at the implementation of a Function to get the kind of insight as we were able to provide above for `some`.

For complex Type Signatures like this one, this process may take quite a bit of time, so make sure to set your expectations accordingly. Don't beat yourself up for taking 20 minutes or more to understand one. Understanding complex things takes time and practice. The more you do it, the faster you'll get.

19.10. Using some and many

Let's write a couple of simple Parsers to leverage some and many.

First, write a Parser that will get as many digits as it can, i.e. 1 or more. Call it digits. Give that a try before continuing.

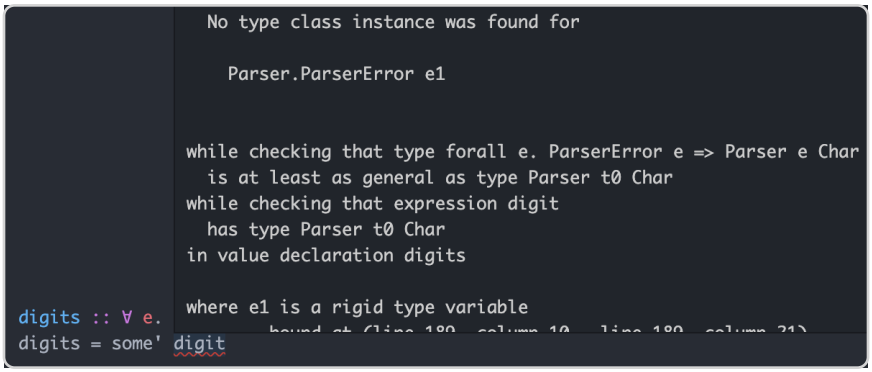
We start coding digits with the Type Signature:

```
digits :: ∀ e. Parser e String
```

Next, we leverage some':

```
digits :: ∀ e. Parser e String
digits = some' digit
```

And now we have an error:



It should be clear from the red underline that we're calling a Function that has a ParserError Constraint, but we don't.

So we'll add that Constraint to our Function:

```
digits :: ∀ e. ParserError e => Parser e String
digits = some' digit
```

And we're done. It was super simple and that's because all of the hard work was done in building some.

Now, write a Parser that performs the same thing as the following Regular Expression:

```
(\d{1,4}), ([a-zA-Z ]+)([0-9]*)
```

If you're not too familiar with Regular Expressions, I'll break this down for you:

```
(\d{1,4})     -- at least 1 digit and no more than 4 [Captured]
,             -- a comma AND a space (even though you can't see it here)
([a-zA-Z ]+)  -- 1 or more alpha OR space characters [Captured]
([0-9]*)      -- zero or more digits [Captured]
```

We're going to return an `Array String`. This will simulate ***Capture Groups*** in Regular Expressions, which are defined using Parentheses. Here we have 3 Capture Groups, `(\d{1,4})`, `([a-zA-Z ]+)`, and `([0-9]*)`.

Notice that the comma and space are outside of any Capture Groups, i.e. they are not contained by Parentheses and, therefore, are parsed but NOT returned.

Give this a try and don't forget about the 2 spaces when writing this `Parser`. We'll call this `ugly`.

Don't read further until you've given it a try.

We'll start with the Type Signature:

```
ugly :: ∀ e. Parser e (Array String)
```

Now we'll add the `do` block:

```
ugly :: ∀ e. Parser e (Array String)
ugly = do
  ???
```

We started coding with a `do` block because this `Parser` is somewhat complex and a Monadic approach seems appropriate. If we're wrong, we can always rewrite it in the `Applicative` style, i.e. with `<$>` and `<*>`.

Now, we parse this part `\d{1,4}`, i.e. at least 1 digit and no more than 4:

```
ugly :: ∀ e. Parser e (Array String)
ugly = do
  p1 <- range' 1 4 digit
  ???
```

Now for the next part, `,` (don't forget the space):

```
ugly :: ∀ e. Parser e (Array String)
ugly = do
  p1 <- range' 1 4 digit
  constChar ','
  constChar ' '
  ???
```

Next, we'll do `[a-zA-Z ]+`, i.e. 1 or more alpha OR space characters:

```
ugly :: ∀ e. Parser e (Array String)
ugly = do
  p1 <- range' 1 4 digit
  constChar ','
  constChar ' '
  p2 <- some' (letter <|> constChar ' ') ❶ ❷
  ???
```

❶ Side note: If this fails, the expected Value will be space. This isn't very helpful, especially since we have 2 spaces in our parse. This shortcoming is because we didn't spend more time on providing good error processing. Good parsing libraries provide mechanisms for this. Unfortunately, we don't have the time to develop one here.

❷ Using `some'` since it returns a `String`.

And finally, `[0-9]*`, i.e. zero or more digits:

```
ugly :: ∀ e. Parser e (Array String)
ugly = do
  p1 <- range' 1 4 digit
  constChar ','
  constChar ' '
  p2 <- some' (letter <|> constChar ' ')
  p3 <- many' digit ❶
  ???
```

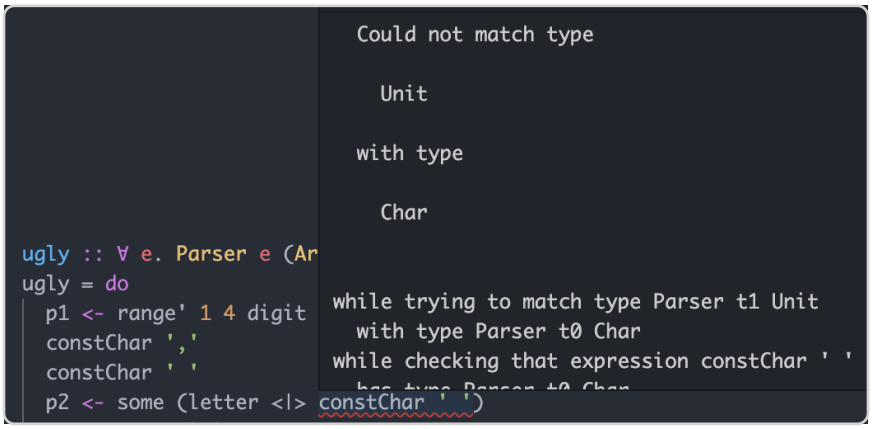
❶ Using `many'` since it returns a `String`.

And now we need to return the parts:

```
ugly :: ∀ e. Parser e (Array String)
ugly = do
  p1 <- range' 1 4 digit
  constChar ','
  constChar ' '
  p2 <- some' (letter <|> constChar ' ') -- COMPILER ERROR!!
  p3 <- many' digit
  pure [p1, p2, p3] ❶
```

❶ Here's where we're returning our "Capture Groups".

Here's our compiler error:



Can you see what the problem is?

Let's look at the definition of `letter` and compare it with `constChar`:

```
letter :: ∀ e. ParserError e => Parser e Char
letter = satisfy "letter" (isAlpha <<< codePointFromChar)

constChar :: ∀ e. ParserError e => Char -> Parser e Unit
constChar c = void $ satisfy (singleton c) (_ == c)
```

They parse different things. We decided to make `constChar` return `Unit` so we didn't have to add `void`. Maybe that wasn't such a great decision, but then it's worked well for us up to this point.

So write a `Parser` called `constChar'` that is a `Parser` of `Char`. Do this before moving on.

I'm going to write `constChar'` by stealing almost all of the code from `constChar`:

```
constChar' :: ∀ e. ParserError e => Char -> Parser e Char ❶
constChar' c = satisfy (singleton c) (_ == c) ❷
```

❶ Changed Unit to Char.

❷ Stole this code from constChar MINUS the void.

Now, it makes sense to just have constChar call this Function:

```
constChar :: ∀ e. ParserError e => Char -> Parser e Unit
constChar = void <<< constChar'
```

This change is important, but often overlooked. There's no reason to keep duplicate code around, so leveraging constChar here is a good change.

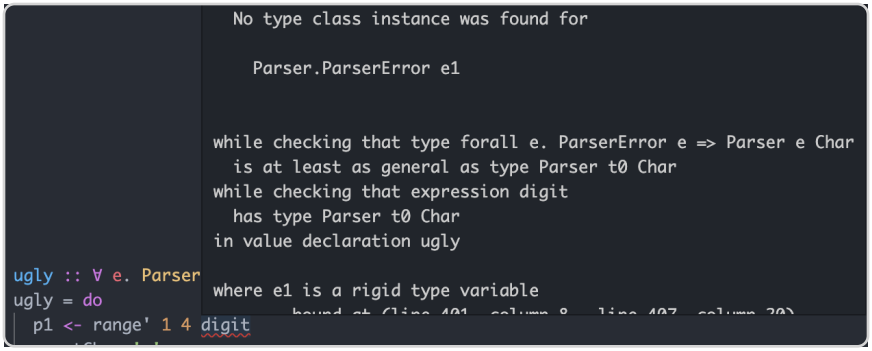
Update your code to match these Functions.

Now, let's fix ugly:

```
ugly :: ∀ e. Parser e (Array String)
ugly = do
  p1 <- range' 1 4 digit      -- COMPILER ERROR!!
  constChar ','
  constChar ' '
  p2 <- some' (letter <|> constChar' ' ') ❶
  p3 <- many' digit
  pure [p1, p2, p3]
```

❶ Using constChar' now.

And the compiler still isn't happy, but for a different reason:



From this, it's clear that we need to constrain e to be a ParserError. The reason for this is that range has that Constraint and since we're using range, we'll also need this Constraint:

```
ugly :: ∀ e. ParserError e => Parser e (Array String) ❶
ugly = do
  p1 <- range' 1 4 digit
  constChar ','
  constChar ' '
  p2 <- some' (letter <|> constChar' ' ')
  p3 <- many' digit
  pure [p1, p2, p3]
```

❶ Added `ParserError` Constraint.

And now, we have a compiled Function.

Before we write the test code, let's take a look at this Function. Notice how we only programmed the Happy Path? It so easy to take this for granted, but this is a huge win if you've ever tried to write code that judiciously checks for errors.

Good error checking in Imperative Languages dominates the code and can be so overpowering that it completely obscures what the code is supposed to do. This is a huge win for PureScript and Haskell because it lets us concentrate on the problem at hand.

Now, you can add the following to the your test code:

```
log $ show $ parse' ugly "17, some words" ❶
log $ show $ parse' ugly "5432, some more words1234567890" ❷
```

❶ Prints `(Right (Tuple "" ["17","some words",""]))`.

❷ Prints `(Right (Tuple "" ["5432","some more words","1234567890"]))`.

Run it and see how it goes.

One last thing. We should be able to write `ugly` in the Applicative style instead of the `do` notation.

I'm just going to give it to you fully written as a point of comparison:

```
uglyA :: ∀ e. ParserError e => Parser e (Array String)
uglyA =
  (\p1 p2 p3 -> [p1, p2, p3])
    <$> range' 1 4 digit
    <* constChar ',' ❶
    <* constChar ' ' ❶
    <*> some' (letter <|> constChar' ' ')
    <*> many' digit
```

❶ The operator <* ignores the result of right argument, i.e. it only relies on it's Side-effect, i.e. parse the character or characters.

While this form is just as valid, it's a bit clunky and has the burden of the Lambda Function, which makes this form less desirable, if you ask me. In the case where the Function is a Data Constructor, however, this form has its merits. You'll have to decide which you like better for each situation.

And just for fun, here's the version that just uses >>=:

```
uglyB :: ∀ e. ParserError e => Parser e (Array String)
uglyB =
  range' 1 4 digit
    >>= \p1 -> constChar ','
      >>= \_ -> constChar ' '
        >>= \_ -> some' (letter <|> constChar' ' ')
          >>= \p2 -> many' digit
            >>= \p3 -> pure [p1, p2, p3]
```

Notice that it's just a mechanical process to convert the do notation to >>=.

Also, notice, that everywhere that >>= is used, the error checks that we coded in bind for Parser get performed. This frees us up to code the Happy Path here.

Well, we could work on this parse for months, but we're going to have to put it away for now. Hopefully, this process has taught you that parsers are not magic. The existing libraries are just far more extensive versions of what we've built. But ours has most of the basic ideas.

Next time you need to parse some arbitrary String, you can comfortably reach for an Applicative or a Monadic Parsing Library in either PureScript or Haskell instead of relying on writing the hieroglyphics of Regular Expressions.

19.11. The RWS Monad

When we first learned about Monads, we went through the coding of Writer, Reader and State. We can now leverage these in programs that we write. But what if we want to have a Computation that writes a log and updates State? How do we combine Monads?

Combining different Monads is a subject for the next Chapter, but we can use the features of these 3 Monads today by just writing a combination Monad called RWS.

The first step in writing any Monad is to model its Type and to do that, we'll need to ask the all important question of what does the Type contain, a Value or a Function?

If you remember, Writer contains a Value and both Reader and State contain Functions. So, if we're going to write a combination Monad, we'll probably want the RWS Type to contain a Function.

In Reader, we pass r, i.e. the read-only Value. In State, we pass s, the read-write Value. So, we're going to need something that contains all of the data for all 3 Monads.

Before we start coding RWS, please return to Ch19.purs. All of our RWS code will be in Ch19.purs. We wouldn't want to commingle our Parser code with the RWS code.

For reference, here are the definitions of the 3 Monads we're combining:

```
newtype Reader r a = Reader (r -> a)
newtype Writer w a = Writer (Tuple a w)
newtype State s a = State (s -> Tuple a s)
```

Now, see if you can model a Type that contains the data for all 3 Monads. Call it RWSResult, then continue reading.

A Record is the first thing that pops into my mind for managing this data:

```
type RWSResult r w s =
  { r :: r ❶
  , w :: w ❷
  , s :: s ❸
  }
```

❶ r is the Reader Type.

❷ w is the Writer Type.

❸ s is the State Type.

This is a just Type Alias, which seems sufficient for our needs here. We'll change it if our initial assessment is wrong. Go ahead and change your code to match this.

Before we write the newtype for RWS, let's review State's Type definition:

```
newtype State s a = State (s -> Tuple a s)
```

State contains a Function that takes a State Value of Type s and returns a Tuple that contains the Pure Computation, a, and the Monadic one, s. We're going to want to do something similar.

Don't read any further until you've written the Type for RWS.

Let's write the Type definition for RWS one step at a time:

```
newtype RWS r w s a = ???
```

We're going to need the Type Parameters `w`, `r` and `s` since `RWSResult` requires them. As usual, we put the `a` last so we can make `Functor`, `Apply`, `Applicative` and `Monad` Instances for this Type.

Next we'll add the Data Constructor:

```
newtype RWS r w s a = RWS (??? -> ???)
```

Now, to help us figure out the Function Type Signature, we'll look at what we did with `State`. `State's` Function takes in the `State`, i.e. `s` and returns the `Tuple a s`.

Let's do the analogous thing:

```
import Data.Tuple (Tuple) ❶

newtype RWS r w s a =
  RWS (RWSResult r w s -> Tuple a (RWSResult r w s)) ❷
```

❶ Import the `Tuple` Type.

❷ Everywhere we had `s` in `State`, we've replaced with `RWSResult r w s`.

And that is our final Type.

Before we go on, let's keep in mind the following:

```
r = read-only, i.e. don't care about the Value on output
w = write-only, i.e. don't care about the Value on input
s = read-write
```

We don't care about `r's` Value when it's returned from any function because it cannot be written to and so should never change. In stark contrast, with `w`, we don't care about the Value on the input to any function since it cannot be read. And for `s`, its Value will matter on both input and output since it can be read from and written to.

These are going to be important points that we'll need to keep this in mind when coding up the Instances for `RWS`.

Now, start by writing a `Functor` Instance for RWS without reading ahead.

We'll start with the `Functor` definition:

```
instance functorRWS :: Functor (RWS r w s) where
```

We keep all Type Parameters constant except for a.

Next, we add the left-hand side of the equal sign:

```
instance functorRWS :: Functor (RWS r w s) where
  map f (RWS g) = ??? ❶
```

❶ We destructure the RWS to get at its Function, g.

map has to return an RWS that contains a Function. That means it's time for our favorite pattern for Functors that contain Functions:

```
instance functorRWS :: Functor (RWS r w s) where
  map f (RWS g) = RWS \rws -> ???
```

Now we can call g passing it the configuration it needs, i.e. rws:

```
instance functorRWS :: Functor (RWS r w s) where
  map f (RWS g) = RWS \rws -> g rws ???
```

What Type does g rws have? If you cannot answer this, then write out the Types for g and rws and then g rws. Do this before moving on.

Writing out the Types is always helpful when they aren't obvious:

```
g :: RWSResult r w s -> Tuple a (RWSResult r w s) ❶
rws :: RWSResult r w s
g rws :: Tuple a (RWSResult r w s) ❷
```

❶ This Type came from the RWS Type definition.

❷ Here's the answer to our earlier question.

So, let's capture and destructure the Tuple returned by g rws:

```
import Data.Tuple (Tuple(..)) ❶

instance functorRWS :: Functor (RWS r w s) where
  map f (RWS g) = RWS \rws -> g rws # \(Tuple x rws') -> ???
```

❶ Added the (..) to include the Tuple Data Constructor.

We've used `g` and we've used `rws`. The only thing we haven't done is to apply `f` to a Value of Type `a`, which in our case is `x`:

```
instance functorRWS :: Functor (RWS r w s) where
  map f (RWS g) =
    RWS \rws -> g rws # \(Tuple x rws') -> Tuple (f x) rws' ❶
```

❶ After applying `f` to `x`, we wrap the return value in a `Tuple`.

Now, write the `Apply` Instance before moving on. Hint: we're going to be making `RWS` a `Monad` so feel free to cheat.

Here's the implementation:

```
instance applyRWS :: Apply (RWS r w s) where
  apply = ap      -- COMPILER ERROR!!
```

We have a compiler error because we haven't written the `Monad` Instance yet. We'll ignore this for now.

Next, write the `Applicative` Instance and then read on to see how you did.

First, we write the instance definition:

```
instance applicativeRWS :: Applicative (RWS r w s) where
```

Now we write the left-hand side of the implementation:

```
instance applicativeRWS :: Applicative (RWS r w s) where
  pure x = ???
```

And on the right-hand side we start with the usual:

```
instance applicativeRWS :: Applicative (RWS r w s) where
  pure x = RWS \rws -> ???
```

And we're just going to wrap up `x` and `rws` in a `Tuple`:

```
instance applicativeRWS :: Applicative (RWS r w s) where
  pure x = RWS \rws -> Tuple x rws
```

But, there's a bug here. We've been passed some log in `w`. To see this better, here's the current code rewritten to show the variables in `rws`:

```
instance applicativeRWS :: Applicative (RWS r w s) where
  pure x = RWS \{ r, w, s } -> Tuple x { r, w, s }
```

The problem is that `w` contains the log so far. And we're taking that whole log and making it the log output for just the `pure` Function. This will duplicate parts of our log. Not good.

Remember that we said earlier, we do NOT care about the `w` on the input since we should NOT be reading the log since it's write-only. Well, by just returning `rws`, we "read" `w` and then made it `pure's` log output.

`pure` should produce NO log, i.e. it should have a `mempty` for the log just like it did in `Writer`:

```
instance applicativeWriter :: Monoid w => Applicative (Writer w) where
  pure x = Writer (Tuple x mempty)
```

So, we're going to fix `RWS's` implementation of `pure` to make sure its log Value is empty:

```
instance applicativeRWS :: Monoid w => Applicative (RWS r w s) where ❶
  pure x = RWS \{r, s} -> Tuple x { r, w: mempty, s} ❷
```

❶ Don't forget the `Monoid` Constraint. The compiler won't.

❷ We dropped the destructuring of the `w` field from the Record since we don't use it.

Unlike, `Reader` and `State`, `Writer` contains a Value not a Function. This is why working with `w` in `RWS` is a bit clunky since we're passing `w` into the `RWS` Functions, which we never did in `Writer`. In `Writer`, we just output a log Value.

So, we have to remember, moving forward, to IGNORE the Value of `w` on the input to an `RWS` Function. We only care about its output Value. You can see that in `pure's` implementation.

When writing the `Bind` Instance of `RWS`, the outputs of `w` will be appended, but the input values of `w` will be ignored.

Speaking of `Bind`, write the `Bind` Instance. Don't read on until you're done or hit a brick wall.

The `Bind` Instance definition:

```
instance bindRWS :: Bind (RWS r w s) where
```

Next, the left-hand side of the equal sign:

```
instance bindRWS :: Bind (RWS r w s) where
  bind (RWS g) f = ???
```

We destructure the RWS to get to its Function.

And now we'll start on the right-hand side with our pattern:

```
instance bindRWS :: Bind (RWS r w s) where
  bind (RWS g) f = RWS \rws -> ???
```

We cannot use f at this point since it wants an a and we have nothing of that Type, at least not yet. We can get an a if we pass rws to g, but it'll be buried in a Tuple:

```
instance bindRWS :: Bind (RWS r w s) where
  bind (RWS g) f = RWS \rws -> g rws # \(Tuple x rws') -> ??? ❶
```

❶ We are destructuring the result of g rws to get the Value, x, which is of Type a.

Now that we have a Value of Type a, i.e. x, we can call f with it:

```
instance bindRWS :: Bind (RWS r w s) where
  bind (RWS g) f = RWS \rws -> g rws # \(Tuple x rws') -> f x ???
```

Are we done? Please try to answer that question. How did you determine your answer?

The way to know if we're done is by knowing the Type of f x. We know, from the Data Constructor, that this Function must return Tuple a (RWSResult r w s):

```
newtype RWS r w s a = RWS (RWSResult r w s -> Tuple a (RWSResult r w s))
```

And now we have to ask ourselves what Type is f x? We know x's Type is a, but what's f's Type?

Let's look at bind's Type Signature to see:

```
bind :: ∀ a b. m a -> (a -> m b) -> m b
```

And now, we can specialize that for RWS by replacing m with RWS r w s. Notice that we're using RWS r w s and NOT just RWS. That's because RWS r w s was what we specified when we defined the Bind Instance:

```
instance bindRWS :: Bind (RWS r w s) where ❶
```

❶ `(RWS r w s)` is our `m`.

So, now here's `bind's` Type Signature specialized for `RWS`:

```
bind :: ∀ a b. RWS r w s a -> (a -> RWS r w s b) -> RWS r w s b
```

Now we can answer what is `f x`:

```
x :: a
f :: a -> RWS r w s b
f x :: RWS r w s b
```

Here's where we left off:

```
instance bindRWS :: Bind (RWS r w s) where
  bind (RWS g) f = RWS \rws -> g rws # \(Tuple x rws') -> f x ???
```

Knowing what we know now, are we done?

If you said no, then you're right. Our Lambda is supposed to return a `Tuple`, but instead we have an `RWS`.

This means that, we're going to need a Run Function like we've written so many times before, e.g. `parse`, `runState`, `runReader`, `runFoldL`, etc.

Here's `runState` as a reference:

```
runState :: ∀ a s. State s a -> (s -> Tuple a s)
runState (State f) = f
```

Go ahead and write `runRWS` and see if you can finish the `Bind` Instance before advancing. Hint: you can look at how we've written these Run Functions before since they're all pretty much the same.

Here's our Run Function's Type Signature:

```
runRWS
  :: ∀ r w s a
  .  RWS r w s a
  -> (RWSResult r w s -> Tuple a (RWSResult r w s)) ❶
```

❶ Redundant Parentheses to show the fact that it returns a Function.

Note that it takes an RWS and returns the Function within that Context. This means all we have to do is remove the Function from its Context and just return it:

```
runRWS
  :: ∀ r w s a
  .  RWS r w s a
  -> (RWSResult r w s -> Tuple a (RWSResult r w s))
runRWS (RWS f) = f
```

By the way, it's always good to list the Type Parameters in the same order in the ∀ part, e.g. always list them as `r w s a` or `r w s` when the `a` is not present. Consistency helps us chunk `r w s a` as a single element. If we see `r w s a` in one place and `w a r s` in another, it's not obvious that they're the same set of Type Parameters.

If you still haven't finished `bind` yet, take the time to do so now. Hint: the next step will involve using `runRWS`.

Now we can use `runRWS` on `f x`:

```
instance bindRWS :: Bind (RWS r w s) where
  bind (RWS g) f = RWS \rws -> g rws
    # \(Tuple x rws') -> runRWS (f x) rws' ❶
```

❶ We pass `rws'` to the Run Function so that we can get the final result, i.e. a `Tuple`.

Since `runRWS (f x)` returns a Function that wants an `RWSResult`, we pass it `rws'`. This will give us back a `Tuple`, which is what this Lambda is supposed to return. This means that this will compile now.

But are we done?

No, unfortunately, we haven't combined the `Writer` logs like we did in the `Bind` Instance for `Writer`:

```
instance bindWriter :: Monoid w => Bind (Writer w) where
  bind (Writer (Tuple x l1)) f = f x
    # \(Writer (Tuple y l2)) -> Writer (Tuple y (l1 <> l2)) ❶
```

❶ Here's where <> gets used to combine the logs.

We need to add code to combine the logs one step at a time. First change is to destructure `rws'` AFTER `g` is called to get access to the log that `g` produced:

```
instance bindRWS :: Bind (RWS r w s) where
  bind (RWS g) f = RWS \rws -> g rws
    # \(Tuple x rws'@{ w }) -> runRWS (f x) rws' ❶
```

❶ We're Pattern Matching `w`, here to capture any logging that happened as a result of calling `g`. We're also using the `@` syntax to name the full Record to `rws'` so we can pass it to `runRWS` later.

The next step is to do the same AFTER calling `runRWS` to capture its logs:

```
instance bindRWS :: Bind (RWS r w s) where
  bind (RWS g) f = RWS \rws -> g rws
    # \(Tuple x rws'@{ w }) -> runRWS (f x) rws'
      # \(Tuple y rws''@{ w:w' }) -> ??? ❶
```

❶ The syntax of `w:w'` Pattern Matches the `w` field RENAMING it to `w'`.

Now, we can COMBINE the logs and wrap everything up in a `Tuple`:

```
instance bindRWS :: Semigroup w => Bind (RWS r w s) where ❶
  bind (RWS g) f = RWS \rws -> g rws
    # \(Tuple x rws'@{ w }) -> runRWS (f x) rws'
      # \(Tuple y rws''@{ w:w' }) -> Tuple y rws'' { w = w <> w' } ❷
```

❶ Do NOT forget the `Semigroup` Constraint since we're using `<>`.

❷ This syntax, `rws'' { w = w <> w' }`, updates the `w` field in `rws''` leaving all other fields in tact.

If your implementation is missing the code to combine the logs from `g` and `runRWS`, please update your code now.

And finally, we're ready to code the easiest of all of the instances. Please write the `Monad` Instance before reading further.

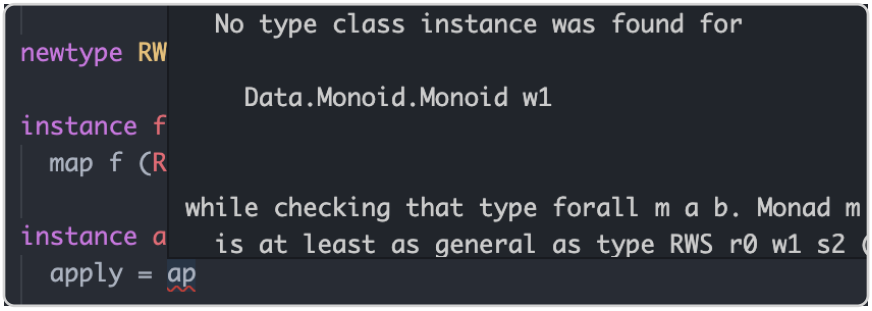
Here's the one liner:

```
instance monadRWS :: Monoid w => Monad (RWS r w s) ❶
```

❶ Since `Monad` is both a `Bind` and `Applicative`, we're going to use the `Monoid` Constraint since `Applicative` has a `Monoid` Constraint and it's the Superclass of `Bind's` Constraint, i.e. `Semigroup`.

At this point, we'd expect the compiler error on ap to go away since we now have a Monad Instance. But that is not the case.

So, we need to take a closer look at it:

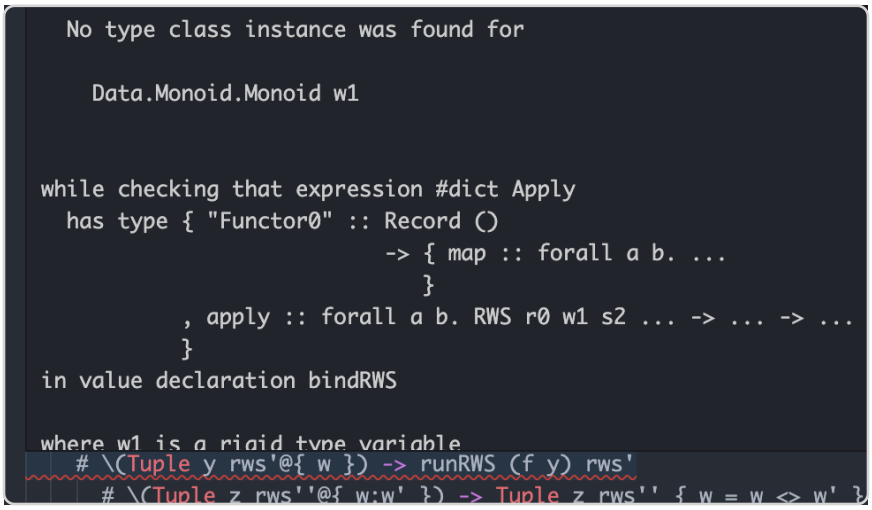


The compiler is complaining that we didn't constrain w to a Monoid. But why?

Since ap is using the Monad Instance of RWS and the Monad is contrained to be a Monoid, Apply will need the same Constraint:

```
instance applyRWS :: Monoid w => Apply (RWS r w s) where
  apply = ap
```

And now the compiler is happy with this instance and unhappy with Bind:



Can you see why this is now a problem? We just changed Apply to have the Monoid Constraint on w.

The reason it's a problem now, is because Apply is Bind's Superclass. So, now Bind needs the same Constraint as Apply:

```
instance bindRWS :: Monoid w => Bind (RWS r w s) where ❶
  bind (RWS g) f = RWS \rws -> g rws
    # \(Tuple x rws'@{ w }) -> runRWS (f x) rws'
      # \(Tuple y rws''@{ w:w' }) -> Tuple y rws'' { w = w <> w' }
```

❶ Changed `Semigroup` to `Monoid`.

And now the compiler is quiet.

Let's not forget that when we wrote the `Monad` Instance, we did NOT prove that `RWS` meets the Monad Laws. But this Chapter is long enough. So we'll just rely on the fact that this `Monad` has existed in `Haskell` for some time and that someone else did their due diligence before putting it there since we're not going to.

Next, we'll need to write the helper functions that I like to call the Monad's API. Here's `RWS's` API that we're going to write:

```
tell :: ∀ r w s. w -> RWS r w s Unit ❶
ask :: ∀ r w s. RWS r w s r ❷
get :: ∀ r w s. RWS r w s s ❸
put :: ∀ r w s. s -> RWS r w s Unit ❹
```

❶ `tell` "pushes" data from the Pure Computation to the Monadic one. This is why the `a` Parameter for `RWS` is `Unit`. We're only calling this function for its Side-effect, which is to send the log.

❷ `ask` "pulls" data from the Monadic Computation to the Pure one. This is why the `a` position of `RWS` is `r`. It's the Value we want to retrieve.

❸ `get` "pulls" data from the Monadic Computation to the Pure one. This is why the `a` position of `RWS` is `s`. It's the Value we want to retrieve.

❹ `put` "pushes" data from the Pure Computation to the Monadic one. This is why the `a` Parameter for `RWS` is `Unit`. We're only calling this function for its Side-effect, which is to send the State.

There are a lot more API Functions that we could write, but this should be good enough so we can write a pretty decent test of our `Monad`.

Go ahead and write `tell` before reading the solution.

We start with the given Type Signature and the right-hand side of the equation:

```
tell :: ∀ r w s. w -> RWS r w s Unit
tell w = ???
```

And then on the right-hand side, we'll start with our pattern while destructuring the Parameter to the Lambda to gain access to the elements of `RWSConfig`:

```
tell :: ∀ r w s. w -> RWS r w s Unit
tell w = RWS \{ r, s } -> ??? ❶
```

❶ Remember, we don't care about `w's` Value on input since it's a write-only Value. This is why we don't specify `w` when we're destructuring the Record.

Now, we'll return a `Tuple` where the Pure Computation is `unit` and the `RWSResult` is unchanged EXCEPT for the log, i.e. the `w`:

```
tell :: ∀ r w s. w -> RWS r w s Unit
tell w = RWS \{ r, s } -> Tuple unit { r, w, s }
```

Next write `ask` referring to its Type Signature above. Then read on.

We'll start writing `ask` like we did with `tell`:

```
ask :: ∀ r w s. RWS r w s r
ask = RWS \{ r, s } -> ??? ❶
```

❶ Once again, we don't care about `w's` Value on input since it's a write-only Value. This is why we don't specify `w` when we're destructuring the Record.

Next, we'll return `r` as the first part of the Tuple, this is how data from the Monadic Computation crosses over to the Pure Computation:

```
ask :: ∀ r w s. RWS r w s r
ask = RWS \{ r, s } -> Tuple r ??? ❶
```

❶ `r`, in the first position of the `Tuple`, is the Value that `bind` will pass to the Pure Computation. It's the thing we are ***asking*** for.

And finally, we'll pass everything from `RWSResult` in the second position of the `Tuple` EXCEPT for the log which will be empty since this function should NOT produce a log:

```
ask :: ∀ r w s. Monoid w => RWS r w s r ❶
ask = RWS \{ r, s } -> Tuple r { r, w: mempty, s } ❷
```

❶ We must constrain `w` to be able to use `mempty`.

❷ We make sure to set the log to empty here.

Now, please code up `get` before reading on.

We'll start in the usual place:

```
get :: ∀ r w s. RWS r w s s
get = RWS \{ r, s } -> ??? ❶
```

❶ Once again, we don't care about `w`'s Value on input since it's a write-only Value. This is why we don't specify `w` when we're destructuring the Record.

Next, we'll return `s` as the first part of the Tuple, this is how data from the Monadic Computation crosses over to the Pure Computation:

```
get :: ∀ r w s. RWS r w s s
get = RWS \{ r, s } -> Tuple s ??? ❶
```

❶ `s`, in the first position of the `Tuple`, is the Value that `bind` will pass to the Pure Computation. It's the thing we are ***getting***.

And finally, we'll pass everything from `RWSResult` in the second position of the `Tuple` EXCEPT for the log which will be empty since this function should NOT produce a log:

```
get :: ∀ r w s. Monoid w => RWS r w s s ❶
get = RWS \{ r, s } -> Tuple s { r, w: mempty, s } ❷
```

❶ We must constrain `w` to be able to use `mempty`.

❷ We make sure to set the log to empty here.

And the final API function to code up is `put`. Please do so before reading the solution.

Once again, we'll start with:

```
put :: ∀ r w s. s -> RWS r w s Unit
put s = RWS \{ r } -> ??? ❶
```

❶ In addition to not caring about `w` on the input, we also don't care about the old State here and, therefore, we don't specify either when destructuring the Record.

Next, we'll return `unit` as the first part of the Tuple, i.e. the Pure Computation result:

```
put :: ∀ r w s. s -> RWS r w s Unit
put s = RWS \{ r } -> Tuple unit ??? ❶
```

❶ `put` doesn't return any Value. This is why `unit` is returned in the first position of the `Tuple`.

And finally, we'll pass ONLY `r` from `RWSResult` in the second position of the `Tuple`. As usual, the log, `w`, will be empty. But the difference here with `put` is that the Value passed in as `s` becomes our new State:

```
put :: ∀ r w s. Monoid w => s -> RWS r w s Unit ❶
put s = RWS \{ r } -> Tuple unit { r, w: mempty, s } ❷
```

❶ We must constrain `w` to be able to use `mempty`.

❷ We make sure to set the log to empty here and to set the new State to the Value, `s`, that was passed into `put`.

Now, it's time for testing. If we've done everything right, we should have ourselves a pretty powerful `Monad`.

But before we forget, let's make sure that `Main` is calling our `Ch19.test` again:

```
module Main where

import Prelude
import Effect (Effect)

import Ch19 as Ch19 ❶

main :: Effect Unit
main = Ch19.test ❷
```

❶ Add the import for `Ch19` and remove the `Parser` import (or comment it out).

❷ Change this to call `Ch19.test`.

Next, add the following to your `Ch19` module:

```
type Config = { debugModeOn :: Boolean } ❶

type Counter = Int ❷
```

❶ This Record will be our read-only information.

❷ This counter is our State.

And then add the following Function:

```
rwsTest :: RWS Config (Array String) Counter Unit ❶
rwsTest = do
  tell ["test the log"]                              ❷
  tell ["test the log2", "test the log3"]            ❸
  config <- ask                                      ❹
  tell ["the config is " <> show config]             ❺
  counter <- get                                     ❻
  tell ["old counter is " <> show counter]           ❼
  put $ counter + 1                                  ❽
  newCounter <- get                                  ❾
  tell ["new counter is " <> show newCounter]        ❿
  pure unit                                          ⓫
```

❶ Notice the Type Parameters. The `r` is `Config`, the read-only Value, i.e. our configuration. The `w` is `Array String`, the write-only Value, i.e. our log. The `s` is `Counter`, our read-write Value, i.e. our State. And finally, the last is `Unit` since we're not doing any computation in `rwsTest`. We're only interested in testing the Side-effects.

❷ Append this message to the log.

❸ Then append these 2 messages to the log.

❹ Retrieve the read-only configuration Record into `config`.

❺ Append a message with the configuration Record Value to the log.

❻ Retrieve the current counter Value into `counter`.

❼ Append a message with the current counter Value to the log.

❽ Increment the counter Value by 1 and save it to the State.

❾ Retrieve the updated counter Value into `newCounter`.

❿ Append a message with the new counter Value to the log.

⓫ Set the final result of the Pure Computation.

And finally, add the following to the test code:

```
log $ show
  $ runRWS rwsTest { r: { debugModeOn: true }, w: mempty, s: 0 } ❶
```

❶ Prints `(Tuple unit { r: { debugModeOn: true }, s: 1, w: ["test the log","test the log2","test the log3","the config is { debugModeOn: true }","old counter is 0","new counter is 1"] })`.

Let's break down the output we expect:

```
unit                      -- result of the Pure Computation
r: { debugModeOn: true }  -- read-only value we originally passed in
s: 1                      -- the final counter value
w:                        -- the log
[ "test the log"
, "test the log2"
, "test the log3"
, "the config is { debugModeOn: true }"
, "old counter is 0"
, "new counter is 1"
]
```

If we've done everything correctly, the code in `rwsTest` should produce a final `RWSResult` that reflects the operations it performed (see above).

And now we have a combination `Monad` that combines `Reader`, `Writer` and `State` into one.

Before `RWS`, we could only write code in `State` or `Writer` or `Reader`. We had no way of performing `Reader` Side-effects along side of `Writer` Side-effects.

But what if we wanted to combining `RWS` with other `Monads`? How would we combine just `Reader` and `State`? Do we have to write another Monad, e.g. `RS`? This approach worked for this particular case, but it's not a good general solution.

What we need is a general way to combine 2 or more `Monads`.

Chapter 20. Monad Stacks

In the previous Chapters, we've seen how we can write in the Happy Path when coding in a Short-circuiting Monad like `Either` or `Maybe`. And we can simulate Global State with the `State` Monad.

But what if we'd like to do both, that is, code within a Monadic Context without having to worry about error checking and to maintain some global-like State? Can we combine `Monads` somehow? Maybe compose them or something.

Turns out that `Monads` do not compose. The reason for this is clear when you look at the specialize `bind` Type Signature for 2 different `Monads`:

```
bind :: ∀ a b. Reader r a -> (a -> Reader r b) -> Reader r b
bind :: ∀ a b. State s a -> (a -> State s b) -> State s b
```

There is no way we can give a Monadic Function, e.g. `a -> State s b` to `bind` for `Reader`. The Types won't match. `bind` for `Reader` expects a Monadic Function of Type `a -> Reader r b`.

What we need is a Function that converts a `State s a` into a `Reader`, somehow.

Let's look at a simple example of how this might work:

```
transformStateIntoReader :: ∀ r s a. State s a -> Reader r (State s a) ❶

rdr :: ∀ r. Reader r Unit

rdr2 :: ∀ r. Reader r Unit

st :: ∀ s. State s Unit

x = rdr >>= \_ -> rdr2 >>= \_ -> transformStateIntoReader(st) >>= \_ -> pure
unit
```

❶ `State` becomes the Pure Value of `Reader`, i.e. it's in the `a` position.

Here `transformStateIntoReader` is some magical function that will wrap the `State` in a `Reader` somehow. But the problem is that the only way we could do this is to hijack the `a` Type Parameter. This is too bad since it's our Pure Computation. We've lost our computation once we do that.

What we really need is a version of `Reader` where we can transform `State` into a `Reader` without losing the Pure Computation Type Parameter. That means that we're going to have to make a different Type with an extra Type Parameter for this transformation:

```
newtype ReaderT r m a = ReaderT (r -> m a)
```

We've added the `m` Type Parameter. This will represent our underlying `Monad` that we've transformed into a `ReaderT`. Notice that the Monadic Function now returns our Pure Computation in the underlying `Monad's` Context.

Once we run `ReaderT`, we're going to need to run the underlying `Monad` to get to our final computation, for example:

```
 runState (runReaderT rdrT r) s
```

Here we first run the `ReaderT` which returns a `State`, which now needs to be run with `runState` and an initial State, `s`.

This should work nicely, but what if we want to do the opposite, i.e. transform a `Reader` into a `State`?

To do this means, we'll need a transformer version of `State`:

```
newtype StateT s m a = StateT (s -> m (Tuple a s))
```

`m` has been added in the same 2 places as we did for `ReaderT`. It's been added as a Type Parameter for the Type Constructor and it's been added as the Context for the contained Function's returned Value.

But what about `Writer`?

We can do the same with `WriterT`:

```
newtype WriterT w m a = WriterT (m (Tuple a w))
```

We can imagine nesting them all:

```
type Nested r s w a = ReaderT r (StateT s (WriterT w ???)) a
```

Looks like we don't have a way to terminate this. The `???` is where we need a `Monad` that will terminate it. We need a `Monad` that isn't a transformer, i.e. it doesn't take another `Monad` in its Type Constructor.

We could just use `Writer` here instead of `WriterT`, but it would be nice if we had a full solution using just our new transformer versions of our `Monads`.

So, we'll need a `Monad` that will terminate the infinite nesting, but we don't want this `Monad` for its Side-effect. We just need it to terminate this definition without doing anything.

Can you remember a Monad like that? If you can't remember, then can you remember a Function that does nothing?

identity has no effect on its input. We can think of it as a Function that does nothing and, in the same vein, the Identity Monad has no Side-effect and can also be thought of as a Monad that does nothing.

So, let's use Identity in place of our ???:

```
type Nested r s w a = ReaderT r (StateT s (WriterT w Identity)) a
```

Now our definition terminates.

And once we write these transformer versions of these Monads, we'll be able to easily recreate RWS. And, as it turns out, we can also create Reader from ReaderT by using our do-nothing Monad, Identity. Think of it as ReaderT combined with a do-nothing Monad:

```
type Reader r a = ReaderT r Identity a
```

And we can ETA-reduce this by canceling the a on both sides of the equal sign:

```
type Reader r = ReaderT r Identity
```

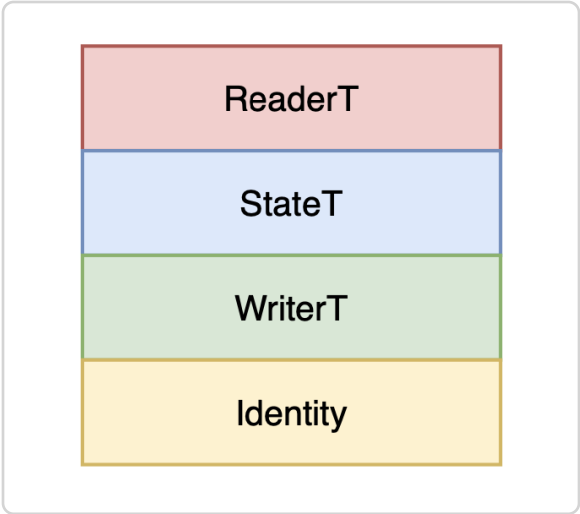
And we can do the same for Writer and State:

```
type Writer w = WriterT w Identity
type State s = StateT s Identity
```

Looks like we don't need the version of Reader, Writer and State that we wrote earlier. We just need these transformer versions.

Now what about that magical wrapping function? How do we take a StateT and wrap it into a ReaderT? Or, worse yet, how do we wrap a WriterT into a StateT and then wrap that into a ReaderT?

Let's look at our nested Monads as a Stack:



What we need a way to ***lift*** `WriterT` into `StateT`. We can imagine a Function that lifts any `Monad`, not just `WriterT`:

```
liftIntoStateT :: ∀ s m a. Monad m => m a -> StateT s m a ❶
```

❶ This imaginary Function takes any `Monad`, `m a`, and lifts it into `StateT`.

And here's a Function that lifts any `Monad` into `ReaderT`:

```
liftIntoReaderT :: ∀ r m a. Monad m => m a -> ReaderT r m a ❶
```

❶ This imaginary Function takes any `Monad`, `m a`, and lifts it into `ReaderT`.

We should stop right here and make a Typeclass for this since we have the classic condition for a Typeclass, i.e. same Function for multiple Types.

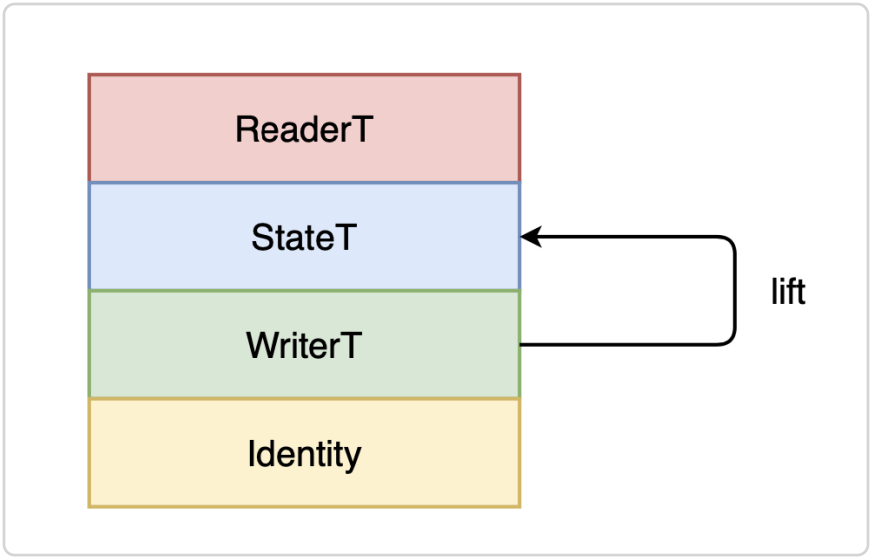
We'll call it `MonadTrans` to represent `Monads` that can transform other `Monads`:

```
class MonadTrans t where
  lift :: ∀ m a. Monad m => m a -> t m a
```

Here `t` is the Monad Transformer and `m` its underlying `Monad`. `lift` will take a Value, `a`, in a Context, `m`, and ***lift*** it into itself, `t m a`.

Now we just have to make instances for our Monad Transformers, which will actually just wrap the underlying `Monads`. This will effectively move them up one slot in our Monad Stack.

Here's an example of lifting `WriterT` into `StateT`, i.e. one layer up in the following Stack:



Next, let's imagine how we could use this Stack, assuming `ReaderT` has the same API as `Reader` as does `StateT` and `WriterT`:

```
type Nested r s w a = ReaderT r (StateT s (WriterT w Identity)) a

nested :: Nested { debug :: Boolean } { count :: Int } String Unit
nested = do
  { debug } <- ask ❶
  { count } <- lift get ❷
  lift <<< lift $ tell $ "The count is: " <> show count ❸
  pure unit
```

❶ No need to `lift` here since the `ReaderT` is on the top of the Monad Stack and we're coding in the `ReaderT` Monad.

❷ We need to `lift` once since `get` returns a `StateT` and it's one level deep in our Monad Stack.

❸ We need to `lift` twice since `tell` returns a `WriterT` and it's two levels deep in our Monad Stack.

Let's write some Functions that run each Layer in our Stack. We're going to assume that `runReaderT`, `runStateT` and `runWriterT` already exist:

```
runReaderLayer
  :: ∀ r s w a
  . ReaderT r (StateT s (WriterT w Identity)) a -- Nested r s w a
  -> StateT s (WriterT w Identity) a ❶
runReaderLayer nested = runReaderT nested { debug: false } ❷
```

❶ Returns the underlying Monad Stack with `StateT` now on top.

❷ We pass some read-only Value, i.e. `{ debug: false }`. Many times this is a configuration Record.

Now we'll write a Function to run the returned Stack:

```
runStateLayer
  :: ∀ s w a
  . StateT s (WriterT w Identity) a
  -> WriterT w Identity a ❶
runStateLayer nested = runStateT nested { count: 0 } ❷
```

❶ Returns a Stack with `WriterT` on the top.

❷ We pass some Initial State, i.e. `{ count: 0 }`.

And finally, we'll write a Function to run that returned Stack:

```
runWriterLayer
  :: ∀ w a
  . WriterT w Identity a
  -> Identity a
runWriterLayer nested = runWriterT nested
```

And we can use `unwrap` from `Data.Newtype` to get our Pure Computation, since `Identity` has a `Newtype` Instance:

```
runIdentityLayer
  :: ∀ a
  . Identity a
  -> a
runIdentityLayer nested = unwrap nested
```

Notice how each Function, ran the topmost Layer of the Stack and returned all of the Layers below. We used that smaller Stack as the input to the next Function, which ran the topmost Layer and returned a smaller Stack and so on from Function to Function until we exhausted the Stack. Then all we had to do was unwrap to get our final result.

We did this with many Functions to show what's happening at each step of the process, but it's far better to write a single Function that runs the whole Stack:

```
runNested
  :: ∀ a
  .  Nested { debug :: Boolean } { count :: Int } String Unit
  -> a
runNested = flip runReaderT { debug: false } ❶
        >>> flip runStateT { count: 0 } ❶ ❷
        >>> runWriterT
        >>> unwrap
```

❶ Here we `flip` some of the run Functions that take additional Parameters.

❷ Notice the use of `>>>`, i.e. the flipped `compose` Operator. This was done so we could have the run calls in the same order of the Monad Stack, i.e. `ReaderT` on top. This is a purely aesthetic choice.

`runNested` executes our Monad Stack starting at the top of the Stack and works down. That's because each Monad is lifted into the one above it and then into the one above that until it reaches the top.

We can think of this process as simply wrapping the lower `Monad` into the upper `Monad` as they get lifted up the Monad Stack.

Let's look back at our `nested` Function:

```
type Nested r s w a = ReaderT r (StateT s (WriterT w Identity)) a

nested :: Nested { debug :: Boolean } { count :: Int } String Unit
nested = do
  { debug } <- ask
  { count } <- lift get
  lift <<< lift $ tell $ "The count is: " <> show count
  pure unit
```

Let's rewrite it using `>>=`:

```
type Nested r s w a = ReaderT r (StateT s (WriterT w Identity)) a

nested :: Nested { debug :: Boolean } { count :: Int } String Unit
nested =
  ask
    >>= \{ debug } -> lift get
      >>= \{ count } ->
              lift <<< lift $ tell $ "The count is: " <> show count
        >>= \_ -> pure unit
```

Notice how each Monad is a `ReaderT`, i.e. the expression left of `->` in each Lambda:

```
ask :: ReaderT r (StateT s (WriterT w Identity)) { debug :: Boolean }

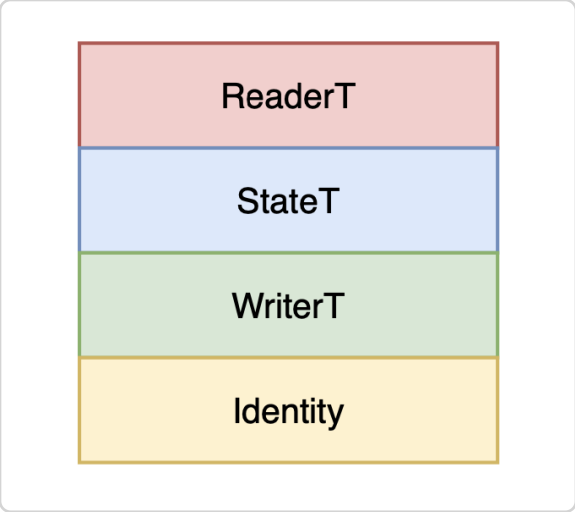
lift get
  :: ReaderT r (StateT s (WriterT w Identity)) { count :: Int }

lift <<< lift $ tell $ "The count is: " <> show count
  :: ReaderT r (StateT s (WriterT w Identity)) Unit

pure unit :: ReaderT r (StateT s (WriterT w Identity)) Unit
```

Thanks to all those `lifts`, which transform our `Monads` into `ReaderT`, we can `bind` all of these `Monads` together. And then when we run the uppermost `Monad`, we wind up with a set of `Monads` that are the same Type as the next Layer down in the Monad Stack.

Here's our stack again for reference:



We continue running them one at a time until we've run them all. We cannot run `Identity` since it's just a dummy wrapper, so we use `unwrap` to access its Value:

```
nested :: ReaderT { debug :: Boolean }
            (StateT { count :: Int } (WriterT String Identity)) Unit
nested = ... ❶

runTopLayer :: StateT { count :: Int } (WriterT String Identity) Unit ❷
runTopLayer = nested
  #   flip runReaderT { debug: false }

runTop2Layers
  :: WriterT String Identity (Tuple Unit { count :: Int } ) ❸
runTop2Layers = nested
  #   flip runReaderT { debug: false }
  >>> flip runStateT { count: 0 }

runTop3Layers
  :: Identity (Tuple (Tuple Unit { count :: Int } ) String ) ❹
runTop3Layers = nested
  #   flip runReaderT { debug: false }
  >>> flip runStateT { count: 0 }
  >>> runWriterT

runTop4Layers :: Tuple (Tuple Unit { count :: Int } ) String ❺
runTop4Layers = nested
  #   flip runReaderT { debug: false }
  >>> flip runStateT { count: 0 }
  >>> runWriterT
  >>> unwrap
```

❶ Implementation details here don't matter

❷ Runs the top Layer of the Monad Stack. Notice that `ReaderT` is gone from the returned Stack.

❸ Runs the top 2 Layers of the Monad Stack. Notice that `StateT` is gone from the returned Stack.

❹ Runs the top 3 Layers of the Monad Stack. Notice that `WriterT` is gone from the returned Stack.

❺ Runs all 4 Layers of the Monad Stack. Notice that `Identity` is gone from the return Value.

20.1. Reducing the need for `lift`

The biggest problem with the following code is that there are too many calls to `lift` and we have to `lift` multiple times depending on how far down the Stack a `Monad` resides:

```
nested :: Nested { debug :: Boolean } { count :: Int } String Unit Unit
nested = do
  { debug } <- ask ❶
  { count } <- lift get ❷
  lift <<< lift $ tell $ "The count is: " <> show count ❸
  pure unit
```

❶ No `lift` calls for `ReaderT`.

❷ 1 `lift` call for `StateT`.

❸ 2 `lift` calls for `WriterT`.

In this example, all of the `lift` calls are there to wrap different Monad API calls. Here we `lift` the `StateT's` API Function, `get`, once, since it's the second from the top in our Monad Stack. We `lift` the `WriterT's` API Function, `tell`, twice, since it's the third from the top of the Stack.

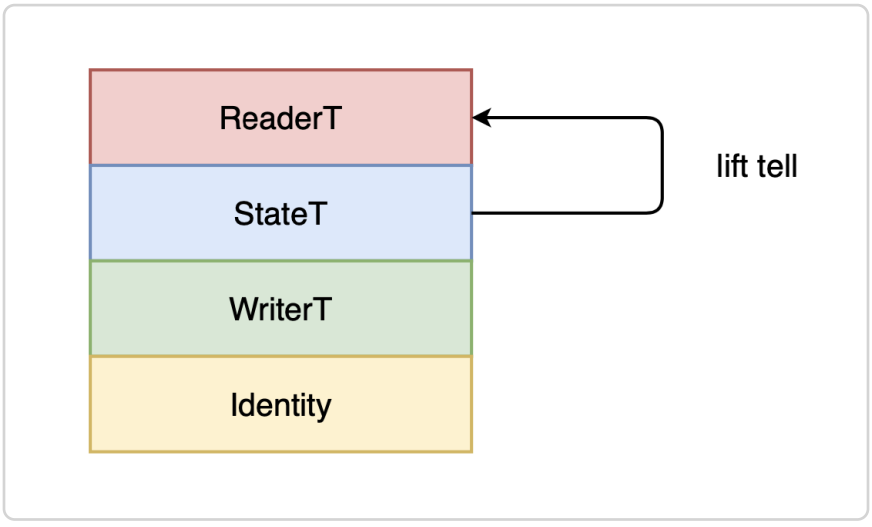
Counting the number of `lift` calls is painful, but imagine the scenario where we change the order of our Stack. We'd have to pour through our code changing the number of calls to `lift` to get the counts right again. The compiler would help us here since the Types would be wrong, but this is a tedious and painful process.

Since we often use API Functions to interact with these `Monads`, it would be nice if we didn't have to worry about lifting them. And even better, we wouldn't have to worry about how many times to call `lift`.

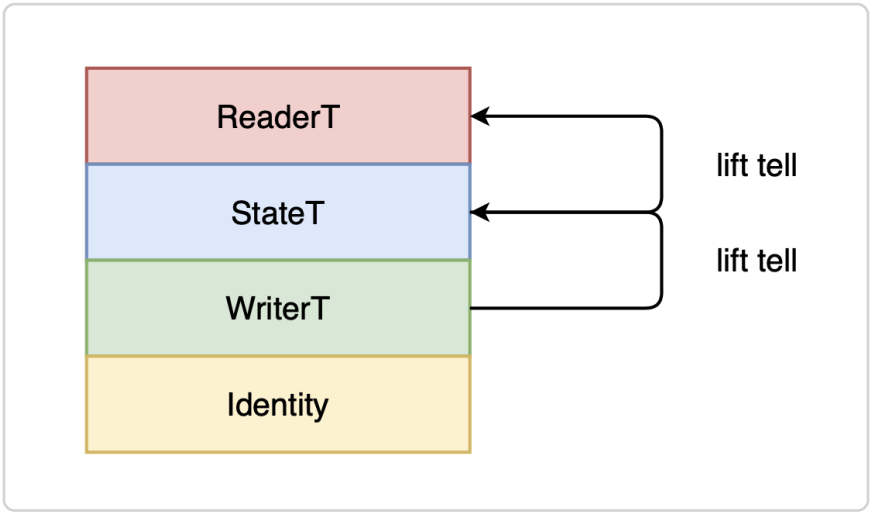
So how can we do that?

One way that we could do this is to require that `ReaderT`, `StateT`, and `WriterT` each support the same API. `ReaderT`, for example, would support `tell`, which is also supported by `StateT`. But both of them would delegate the `tell` call to their underlying `Monad`.

`ReaderT's tell` Function will lift the `tell` Function of its underlying `Monad`, which, in the case of our example Stack, is `StateT`:

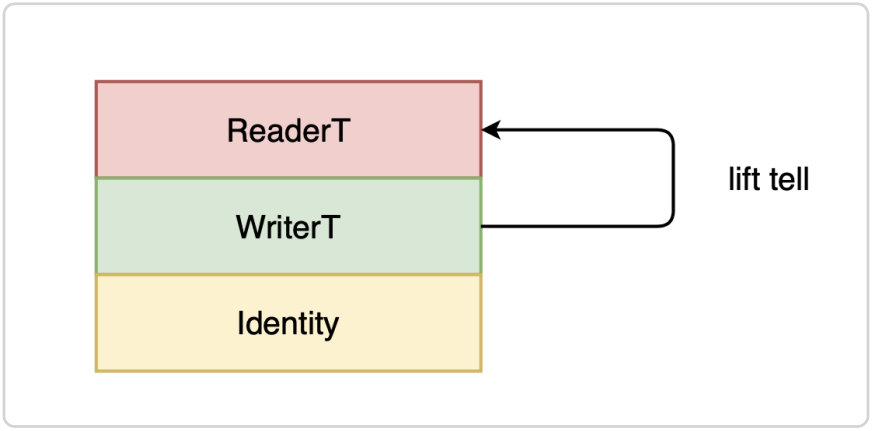


StateT's tell Function will do the same and lift the tell Function of its underlying Monad, which, in the case of our example Stack, is WriterT:



WriterT is where tell is implemented, so it simply appends to the log.

Notice how tell get's lifted 2 times with this approach. It's lifted just the right amount of times needed for this Stack. If we had a different Stack, it would be lifted just the right number of times for that particular Stack, for example:



In this Stack, it's only lifted once since the lifting terminates with the Monad that actually implements the API call, in this case, WriterT.

The correct way to implement API calls is to use Typeclasses since we have multiple Types, i.e. different Monad Transformers, and the same set of Functions. Here's the Typeclass in PureScript that supports tell:

```
class Monad m <= MonadTell w m | m -> w where ❶
  tell :: w -> m Unit
```

❶ Notice the Functional Dependency, | m -> w that says that for each Monad, m, there can be ONE AND ONLY ONE w. This may seem limiting but in practice your program will typically only have 1 Monad Stack. This dependency was probably added to help the compiler determine Types better, which means

that we don't have to specify explicit Types in our code.

Let's look at `ReaderT's` implementation of `MonadTell`. First, we'll need to look at `lift` for `ReaderT`:

```
instance monadTransReaderT :: MonadTrans (ReaderT r) where
  lift :: ∀ m a. Monad m => m a -> ReaderT r m a ❶
  lift = ReaderT <<< const
```

❶ The Type Signature specialized for `ReaderT`.

`lift` simply wraps the Monadic Value, `m a`, in a `ReaderT` whose function ignores the read-only value, via `const`, and returns that Monadic Value.

Here's the code for `tell` from `ReaderT`:

```
instance monadTellReaderT
    :: MonadTell w m => MonadTell w (ReaderT r m) where
  tell :: w -> ReaderT r m Unit
  tell = lift <<< tell
```

It's easy to think that this is an infinite recursion with `tell` on both sides of the equal sign. And this would be the case if `tell` wasn't a Typeclass Method since it would be referring to itself.

But when we're dealing with a Typeclass Method on both sides of the equal sign, we're typically looking at different instances of that Method.

In this case, the `tell` on the right-hand side is the `tell` that's implemented by the underlying `Monad`, `m`.

The compiler will see that we're composing it with our `lift` Function and determine that it can only be the `tell` from the underlying `Monad` since that's what `lift` expects, i.e. an `m a` (see Type Signature for `lift` specialized for `ReaderT`).

20.2. Monad Transformers and their APIs

Monad Transformers are `Monads` that transform other `Monads`. They do this by taking Monadic Values from other `Monads` and then transforming them to themselves by lifting them into their own Context. When they are run, they simply return the original Monadic Value.

Monad Transformers are in the PureScript library, `purescript-transformers`. This means that you'll have to add `transformers` to your `spago.dhall` file to use them.

Here are some of the more common Monad Transformers:

```
ReaderT r m a
WriterT w m a
StateT s m a
ExceptT e m a ❶
```

❶ We haven't seen this Type yet. It's for handling errors.

And the general form is:

```
 t m a
```

`t` is the `Monad` Transformer, `m` is the underlying `Monad` and `a` is the Pure Value.

This form makes sense when you consider `lift's` Type Signature:

```
class MonadTrans t where
  lift :: ∀ m a. Monad m => m a -> t m a ❶
```

❶ Here `t` could be `ReaderT r` or `WriterT w` or `StateT s` or `ExceptT e`, etc.

The following are the Typeclass definitions for many of the more common APIs:

```
class Monad m <= MonadAsk r m | m -> r where
  ask :: m r

class Monad m <= MonadTell w m | m -> w where
  tell :: w -> m Unit

class Monad m <= MonadState s m | m -> s where
  state :: ∀ a. (s -> (Tuple a s)) -> m a

class Monad m <= MonadThrow e m | m -> e where
  throwError :: ∀ a. e -> m a

class MonadThrow e m <= MonadError e m | m -> e where
  catchError :: ∀ a. m a -> (e -> m a) -> m a
```

The following are many of the more common helper API Functions:

```
asks :: ∀ r m a. MonadAsk r m => (r -> a) -> m a

get :: ∀ m s. MonadState s m => m s
put :: ∀ m s. MonadState s m => s -> m Unit

modify :: ∀ s m. MonadState s m => (s -> s) -> m s
modify_ :: ∀ s m. MonadState s m => (s -> s) -> m Unit
```

These helpers are written in terms of the Typeclass Methods, which is why they all have Constraints. It's also why these only have to be written once.

20.3. The Error Monad Transformer, ExceptT

We saw a glimpse of ExceptT earlier. But what exactly does ExceptT do and why is it useful?

To answer that question, we should start with its definition:

```
newtype ExceptT e m a = ExceptT (m (Either e a))
```

You can see that its not very different from Either. So, why not just use Either and make an EitherT? The reason is a complicated. But it basically has to do with history and how competing library development in the Haskell ecosystem attempted to avoid name collisions. The PureScript developers decided to keep the name to maintain the familiarity with the Haskell Type.

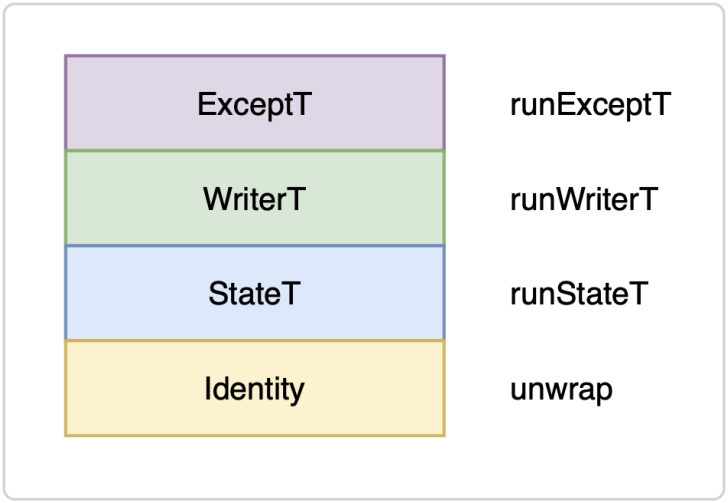
Suffice it to say, ExceptT is basically just EitherT.

Please do NOT confuse this with Exceptions from other languages. It is NOT an Exception. Think about how Either works and just add the transformer logic and you have ExceptT.

The MonadThrow and MonadError Typeclasses are implemented in ExceptT. All other Monad Transformers simply delegate these Typeclass Methods to their underlying Monads with the expectation that ExceptT or something similar will be below it in the Monad Stack.

20.4. Monad Stack Order

The order of the Monad Stack matters. Take the following example:



We'll use the REPL to help us determine the final result based on how we'd run this Stack:

```
$ npx spago repl
PSCi, version 0.14.1
Type :? for help

> import Prelude
> import Control.Monad.State.Trans
> import Control.Monad.Writer.Trans
> import Control.Monad.Except.Trans
> :t flip runStateT unit <<< runWriterT <<< runExceptT ❶
forall t17 t18 t20 t9. ExceptT t20 (WriterT t17 (StateT Unit t9)) t18 -> t9
(Tuple (Tuple (Either t20 t18) t17) Unit)
```

❶ We're using `unit` as the Initial State.

Let's clean up that Type Signature a bit:

```
-- original
flip runStateT unit <<< runWriterT <<< runExceptT ❶
  :: forall t17 t18 t20 t9
  .  ExceptT t20 (WriterT t17 (StateT Unit t9)) t18 ❷
  -> t9 (Tuple (Tuple (Either t20 t18) t17) Unit)
-- cleaned up
flip runStateT s <<< runWriterT <<< runExceptT ❸
  :: ∀ e w s m a
  .  ExceptT e (WriterT w (StateT s m)) a ❹
  -> m (Tuple (Tuple (Either e a) w) s) ❹
-- return Value
m (Tuple (Tuple (Either e a) w) s) ❺
```

❶ We run our Stack from the top down, which can be seen from right to left since we're composing the run Functions.

❷ Notice how this Type matches our Stack.

❸ Replaced `unit` with `s`.

❹ Replaced `Unit` with `s`.

❺ Final Value.

Notice that we wind up with the final Value in some `Monad`, `m`. This would be `Identity` in our above Stack diagram.

Another way to figure out the final return Type is by looking at the Type Signatures for each of the run Functions and apply some simple analysis:

```
runExceptT :: ∀ e m a. ExceptT e m a -> m (Either e a)
runWriterT :: ∀ w m a. WriterT w m a -> m (Tuple a w)
runStateT :: ∀ s m a. StateT s m a -> s -> m (Tuple a s)
```

When we first call `runExceptT`, we'll get `m (Either e a)`. `(Either e a)` will be the `a` for the next Monad in the Stack, which in this case is `WriterT`.

Then we call `runWriterT`, which returns `m (Tuple a w)` and after substituting what the previous `Monad` run returned into `a`, we get:

```
m (Tuple (Either e a) w)
```

`Tuple (Either e a) w` will be the `a` for the next `Monad` in the `Stack`, i.e. `StateT`.

And finally, we call `runStateT`, which returns `m (Tuple a s)` and after substituting what the previous `Monad` run returned, we get:

```
m (Tuple (Tuple (Either e a) w) s)
```

When we unwrap this, since `m` is `Identity` in our Stack diagram, we wind up with:

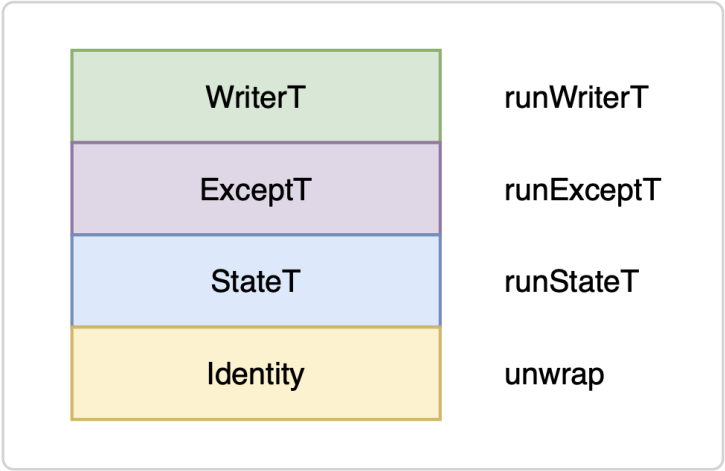
```
Tuple (Tuple (Either e a) w) s
```

Notice that when we get an error, we also get the log, `w`, the final State, `s` and the error, `e`. Obviously, we don't get the Pure Computational Value since the computation failed.

This means that our log and state are the values they were right at the point of failure. This can be very

helpful for determining what went wrong.

Now, let's move `ExceptT` down 1 Layer the Stack:



To run this Stack, we'll call `runWriterT` first:

```
m (Tuple a w)
```

Then we'll call `runExceptT` returning:

```
m (Either e (Tuple a w)) ❶
```

❶ `a` substituted by `Tuple a w`, i.e. the `a` from `runWriterT`.

And finally, we'll call `runStateT`:

```
m (Tuple (Either e (Tuple a w)) s) ❶
```

❶ `a` substituted by `Either e (Tuple a w)`, i.e. the `a` from `runStateT`.

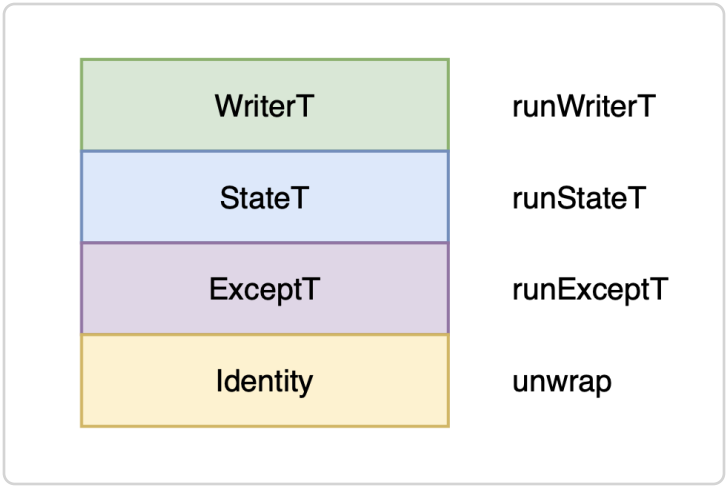
Unwrapping since `m` is `Identity`:

```
Tuple (Either e (Tuple a w)) s
```

This time, when we get an error, we LOSE the log, `w`. That's because `Either e (Tuple a w)` is going to only give us the log if there is NO error.

Seems like we lose everything above the `ExceptT` Layer if an error occurs.

Let's test that hypothesis by moving `ExceptT` down 1 more Layer and see if we lose our State, `s`:



When we do this we get:

```
m (Tuple a w)                          -- runWriterT
m (Tuple (Tuple a w) s)                -- runStateT
m (Either e (Tuple (Tuple a w) s))     -- runExceptT
```

After unwrapping:

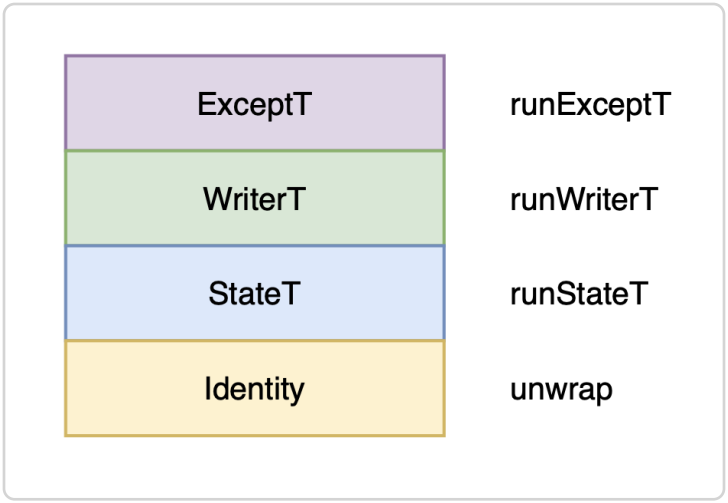
```
Either e (Tuple (Tuple a w) s)
```

With this Stack, we also lose State, `s`, if an error occurs. In fact, with `ExceptT` at the bottom of our Transformer Stack, which is not including the base `Monad`, `Identity`, we lose everything when an error occurs.

20.5. Coding with Monad Transformers

Let's write some code that operates within a Monad Stack.

To do so, we're going to pick a Stack that has `ExceptT` at the top of the Stack so we don't lose any information:



Let's create a `Monad` definition for our Application:

```
import Control.Monad.Except.Trans (ExceptT)
import Control.Monad.State.Trans (StateT)
import Control.Monad.Writer.Trans (WriterT)
import Data.Identity (Identity)

type AppM e w s a = ExceptT e (WriterT w (StateT s Identity)) a ❶
```

❶ The `M` in the Type name reminds us that it's a Monad, and, in particular, a Monad Stack.

Notice how `AppM` can be ETA-reduced, i.e. we can cancel the `a` on the right-hand side of both sides of the equal sign.

This can be done because `AppM` is simply a Type Alias. You cannot do this with a Type definition. Although the ETA-reduced version is valid, I'm going to leave it NOT reduced for clarity sake.

One other important point here is that `WriterT`, `StateT` and `Identity` are all short 1 Type Parameter:

```
type AppM e w s a = ExceptT e (WriterT w (StateT s Identity)) a
```

Looking at `WriterT w (StateT s Identity)`, we can see that it's missing it's final Type Parameter, `a`:

```
newtype WriterT w m a = WriterT (m (Tuple a w)) ❶
```

❶ `WriterT` normally takes 3 Type Parameters, `w`, `m` and `a`.

So, what happened to `WriterT's` final Type Parameter?

It gets it from `ExceptT`:

```
newtype ExceptT e m a = ExceptT (m (Either e a)) ❶
```

❶ In our Stack, as defined by `AppM`, `m` is `WriterT w (StateT s Identity)`, which is passed its final Type Parameter, i.e. `Either e a`.

And then `WriterT` give `StateT` its `a`:

```
newtype WriterT w m a = WriterT (m (Tuple a w)) ❶
```

❶ `m` here is `State s Identity`, which is passed `Tuple a w` as its final Type Parameter.

And finally, `WriterT` gives `Identity` its `a`:

```
newtype StateT s m a = StateT (s -> m (Tuple a s)) ❶
```

❶ `m` here is `Identity`, which is passed `Tuple a s` as its final Type Parameter.

Now that we understand `AppM's` definition, we can begin writing our Application:

```
type AppM e w s a = ExceptT e (WriterT w (StateT s Identity)) a

app :: AppM String String Int Unit
app = do
  pure unit
```

This is a good starting point. Now let's add logging:

```
type AppM e w s a = ExceptT e (WriterT w (StateT s Identity)) a

app :: AppM String String Int Unit
app = do
  tell "Starting App..." ❶
  pure unit
```

❶ Remember, there is no need to call `lift` here since we're running in the `ExceptT` Monad and it has a `MonadTell` instance.

It would be nice to have line endings automatically added to the end of our log. Let's write a Function called `log` that will do that for us:

```
log :: ∀ m. MonadTell String m => String -> m Unit ❶ ❷
log s = tell $ s <> "\n"
```

❶ We constrain the `Monad`, `m`, to have a `MonadTell` Instance.

❷ The return value here matches `tell's`, i.e. `m Unit`.

Let's update `app` to use `log`:

```
app :: AppM String String Int Unit
app = do
  log "Starting App..."
  pure unit
```

Now let's add some sort of error checking:

```
import Prelude ❶

app :: AppM String String Int Unit
app = do
  log "Starting App..."
  n <- get ❷
  when (n == 0) $ void $ throwError "WE CANNOT HAVE A 0 STATE!" ❸
  pure unit
```

❶ Import `Prelude` for `when`.

❷ Get the `Int` from the State.

❸ Check to see if we got a `0` from the State and if we did, then we have an error.

This code uses a `when`, which is a helper Function used often in `do` blocks when there's something you want to do only if the specified condition holds true. There is a corresponding helper called `unless`, that does the same when the condition is false.

You can think of them as an `if` without an `else`. But how can that be? In PureScript and Haskell, we can't have an `if` without an `else`.

The way it works is by having the `else` part of `when` return a default Value.

To understand this in action, let's look at the `when` Function in detail:

```
when :: ∀ m. Applicative m => Boolean -> m Unit -> m Unit ❶
when true m = m ❷
when false _ = pure unit ❸
```

❶ Notice that it returns an `m Unit`, i.e. `Unit` in some `Applicative` or a `Monad` since all `Monads` are `Applicatives`. This is important since this allows us to have an `if` with a default `else`.

❷ In the `if` case, the `m` is a Monadic Value, but it must not have a Pure Computational Value, otherwise, we'd need to provide the `else` part explicitly. It must be a `Unit` in the Monadic Context. We can always use `void` on a Monadic Value to replace its Value with `unit` so that we can use it with `when`.

❸ In the `else` case, it does nothing but return a default Value of `unit` in the Monadic Context.

Since `throwError` has the Type Signature, `e -> m a`, `void` must be used to dispose of the Pure Computation, i.e. change `m a` to `m Unit`. However, the Side-effect of Short-circuiting is still preserved:

```
when (n == 0) $ void $ throwError "WE CANNOT HAVE A 0 STATE!"
```

Let's continue adding to our Application code by adding code after the error checking to modify the State and write to the log once again:

```
app :: AppM String String Int Unit
app = do
  log "Starting App..."
  n <- get
  when (n == 0) $ void $ throwError "WE CANNOT HAVE A 0 STATE!"
  put $ n + 1 ❶
  log "Incremented State" ❷
  pure unit
```

❶ Update the State by adding `1` to the old State.

❷ Log that we incremented State.

Now, it's time to write a Function that will let us run the `AppM` Monad Stack:

```
runApp :: Int -> AppM String String Int Unit -> ?x ❶ ❷
```

❶ `runApp` takes a State, i.e. an `Int` and an `AppM`, i.e. our Application `Monad` and returns something.

❷ The return Type is just a Type Hole for now.

If you're not sure what to put as the return Type, you can put what's called a ***Type Hole***, in this case, `?x`. The compiler will generate an error for you at this spot and tell you what Type it inferred. One caveat to using Type Holes in PureScript is that there must not be any other compiler errors.

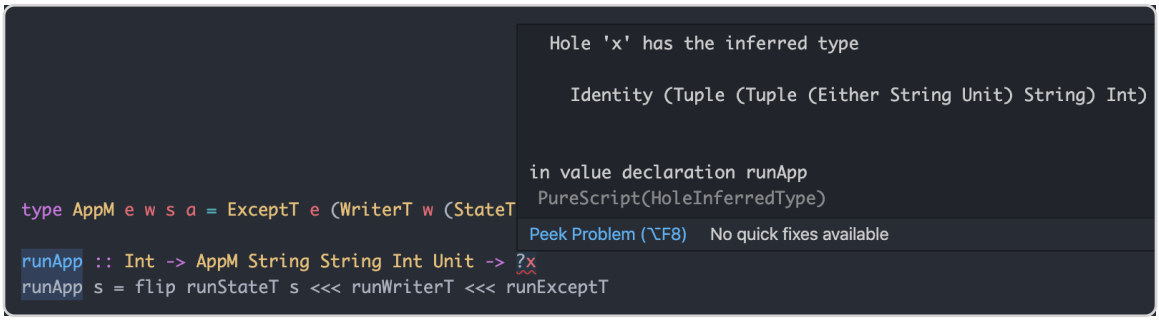
The syntax is `?` followed by a lowercase name. I usually don't care about the name since I'm using an IDE, i.e. VSCode. The name is more important if you're compiling at the command line and you have several Type Holes and need to distinguish them from one another.

In Haskell, Type Holes are unnamed and are the underscore character, `_` and do not have the same restriction for requiring a clean compile to provide useful Type inference information.

Now, we'll just add the Stack execution code that matches the order of our Monad Stack, i.e. `ExceptT` first, then `WriterT` and so on:

```
runApp :: Int -> AppM String String Int Unit -> ?x
runApp s = flip runStateT s <<< runWriterT <<< runExceptT
```

Once this compiles, the compiler will tell us the following:



This saves us from having to figure out this complex Type and we can now replace our Type Hole with the inferred Type. Let's also add `unwrap` to eliminate the `Identity Monad`:

```
import Data.Newtype (unwrap) ❶

runApp
  :: Int
  -> AppM String String Int Unit
  -> Tuple (Tuple (Either String Unit) String) Int ❷
runApp s = unwrap <<< flip runStateT s <<< runWriterT <<< runExceptT
```

❶ Import `unwrap`.

❷ The inferred Type from the Type Hole minus `Identity` since we're unwrapping now.

And just because our code compiles doesn't mean we're done. Let's take a look at it in full for any areas of improvement:

```
import Prelude

import Control.Monad.Except.Trans (ExceptT, runExceptT, throwError)
import Control.Monad.State.Trans (StateT, get, put, runStateT)
import Control.Monad.Writer.Trans ( class MonadTell, WriterT
                                  , runWriterT, tell)
import Data.Either (Either)
import Data.Identity (Identity)
import Data.Newtype (unwrap)
import Data.Tuple (Tuple)

type AppM e w s a = ExceptT e (WriterT w (StateT s Identity)) a

runApp
  :: Int
  -> AppM String String Int Unit
  -> Tuple (Tuple (Either String Unit) String) Int
runApp s = unwrap <<< flip runStateT s <<< runWriterT <<< runExceptT

log :: ∀ m. MonadTell String m => String -> m Unit
log s = tell $ s <> "\n"

app :: AppM String String Int Unit
app = do
  log "Starting App..."
  n <- get
  when (n == 0) $ void $ throwError "WE CANNOT HAVE A 0 STATE!"
  put $ n + 1
  log "Incremented State"
  pure unit
```

The first thing I notice is how the AppM is too complex, i.e. it has too many Type Parameters and it's used in several Type Signatures.

We're going to simplify it by introducing another Type Alias:

```
type AppStack e w s a = ExceptT e (WriterT w (StateT s Identity)) a

type AppM = AppStack String String Int Unit
```

Now we can simply refer to AppM in Function Type Signatures, which means that if we ever change our Monad Stack, we won't have to change all of our Type Signatures.

And if we change the order, e.g. swap `WriterT` and `StateT's` positions, we'll only need to change these 2 Type Aliases. The rest of our Type Signatures will remain unchanged.

Now we can change the Type Signatures to only use `AppM`:

```
runApp
  :: Int
  -> AppM ❶
  -> Tuple (Tuple (Either String Unit) String) Int
runApp s = unwrap <<< flip runStateT s <<< runWriterT <<< runExceptT

app :: AppM ❶
app = do
  log "Starting App..."
  n <- get
  when (n == 0) $ void $ throwError "WE CANNOT HAVE A 0 STATE!"
  put $ n + 1
  log "Incremented State"
  pure unit
```

❶ Use just `AppM` in these 2 locations.

Another improvement that comes to mind is to create a Type Alias for our Application result Type:

```
type AppResult = Tuple (Tuple (Either String Unit) String) Int
```

Now that cleans up the Type Signature for `runApp`:

```
runApp :: Int -> AppM -> AppResult
runApp s = unwrap <<< flip runStateT s <<< runWriterT <<< runExceptT
```

And if we change the Stack order, we'll only have to change `AppResult's` definition.

Okay, that's not completely true. Our code will rely on the shape of our result. Somewhere in our code, we're going to have to dig into those nested `Tuples` to get at the data. Depending on how we code our Application, much of our code could be coupled to this nested `Tuple` structure.

We can fix that by writing a Function to convert the nested `Tuples` into a data structure that's independent of our Stack order:

```
import Data.Either (Either(..)) ❶
import Data.Maybe (Maybe(..)) ❶
import Data.Tuple (Tuple(..)) ❶

type AppEffects = ❷
  { log :: String
  , state :: Int
  , result :: Maybe Unit
  }

results :: AppResult -> Tuple (Maybe String) AppEffects ❸
results (Tuple (Tuple (Left err) l) s)
  = Tuple (Just err) { log: l, state: s, result: Nothing }
results (Tuple (Tuple (Right _) l) s)
  = Tuple Nothing { log: l, state: s, result: Just unit }

runApp :: Int -> AppM -> Tuple (Maybe String) AppEffects ❹
runApp s = results
  <<< unwrap
  <<< flip runStateT s
  <<< runWriterT
  <<< runExceptT
```

❶ Added `(..)` in these 3 places to import the Data Constructors.

❷ `AppEffects` contains all of the data, except for the error, from `AppResults` in a form that's independent of our Stack order. Notice that `result` is a `Maybe`. That's because we won't have a final result if an error occurs.

❸ `results` converts `AppResults` to `Tuple (Maybe String) AppEffects`. The `Maybe String` is our possible error.

❹ We're now leveraging this improvement in our `runApp` Function.

If we add to or subtract from our Stack, then `AppEffects`, `results` and `runApp` would be the only places that would need to change.

The rest of our code will now access `Tuple (Maybe String) AppEffects`, since it will never change with the Stack order.

Now if anything in our Stack changes, we'll only need to change the following:

```
type AppStack e w s a = ExceptT e (WriterT w (StateT s Identity)) a

type AppM = AppStack String String Int Unit

type AppResult = Tuple (Tuple (Either String Unit) String) Int

type AppEffects =
  { log :: String
  , state :: Int
  , result :: Maybe Unit
  }

results :: AppResult -> Tuple (Maybe String) AppEffects
results (Tuple (Tuple (Left err) l) s)
  = Tuple (Just err) { log: l, state: s, result: Nothing }
results (Tuple (Tuple (Right _) l) s)
  = Tuple Nothing { log: l, state: s, result: Just unit }

runApp :: Int -> AppM -> Tuple (Maybe String) AppEffects
runApp s = results
  <<< unwrap
  <<< flip runStateT s
  <<< runWriterT
  <<< runExceptT
```

This is a real win since it localizes the knowledge of the Stack order, the returned structure and, to some degree, the returned contents to these few definitions.

The rest of our codebase should remain relatively unchanged by changes to our Stack.

Now, at this point, I'm not loving the `Tuple (Maybe String) AppEffects` in 2 places. Let's factor that out:

```
type AppStack e w s a = ExceptT e (WriterT w (StateT s Identity)) a

type AppM = AppStack String String Int Unit

type StackResult = Tuple (Tuple (Either String Unit) String) Int ❶

type AppEffects =
  { log :: String
  , state :: Int
  , result :: Maybe Unit
  }

type AppResult = Tuple (Maybe String) AppEffects ❷

results :: StackResult -> AppResult ❸
results (Tuple (Tuple (Left err) l) s)
  = Tuple (Just err) { log: l, state: s, result: Nothing }
results (Tuple (Tuple (Right _) l) s)
  = Tuple Nothing { log: l, state: s, result: Just unit }

runApp :: Int -> AppM -> AppResult ❸
runApp s = results
  <<< unwrap
  <<< flip runStateT s
  <<< runWriterT
  <<< runExceptT
```

❶ Renamed `AppResult` to `StackResult` since it's more accurate and I really wanted to free up `AppResult` for what the true result is.

❷ Repurposing `AppResult` here, which reflects the true results for running the Application.

❸ Changed the Types here to reflect the above changes.

Now, not only are the names better, but the Type Signatures are simpler.

Here's a test Function to run our Application:

```
import Effect (Effect)
import Effect.Console as Console ❶

test :: Effect Unit
test = do
  Console.log $ show $ runApp 0 app ❷ ❸
  Console.log $ show $ runApp 99 app ❷ ❹
```

❶ Import this modules with an alias since we have our own `log` Function and we don't want `log` from this module to name collide.

❷ Fully qualify the name of the `log` Function.

❸ We expect this to fail since the initial State is `0`, which, if you remember, is our error case.

❹ We expect this to succeed.

When we run our test code, we get the following:

```
(Tuple (Just "WE CANNOT HAVE A 0 STATE!")
  { log: "Starting App...\n"
  , result: Nothing
  , state: 0 })
(Tuple Nothing
  { log: "Starting App...\nIncremented State\n"
  , result: Just unit
  , state: 100 })
```

We have 2 test cases here, an error case and a success case. In both cases, we have our final State and our log. That's because we placed `ExceptT` at the top of our Monad Stack. This is great since we can see our program's State when it failed.

Let's look at our application in the context of the error test case:

```
app :: AppM
app = do
  log "Starting App..."
  n <- get
  when (n == 0) $ void $ throwError "WE CANNOT HAVE A 0 STATE!"
  put $ n + 1                -- NEVER RAN!!
  log "Incremented State"    -- NEVER RAN!!
  pure unit                  -- NEVER RAN!!
```

When there's an error, all of the code in the `do` block after the `throwError` doesn't get executed. We can see

this by the fact that our State was never incremented like it was in the success case and our log is one line shorter.

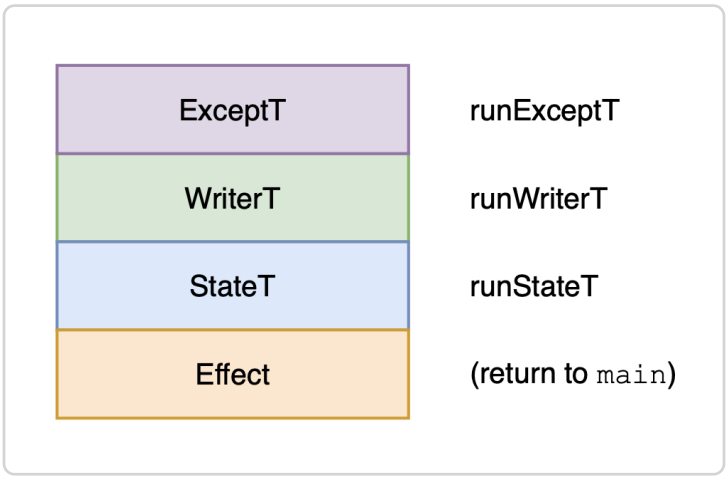
20.6. Coding with Effect at the Base

The base Monad, i.e. the Monad at the bottom of the Stack, is usually Effect. This allows us to return our app to the main Function, which runs in the Effect Monad.

We will do a deeper dive into Effect in the Beyond Section of this book, but for now, just realize that if you want to effect the outside world, e.g. talk over the network, write to a file, update the browser's DOM, etc., you're going to need to do that inside the Effect Monad.

By the way, in Haskell, this Monad is called IO. Same ideas though.

We're going to change our Stack to have Effect at the base:



When we run our Application with Effect at the base, the result will be an Effect. In our case, we'll return it to test, which also runs in Effect and is called by main.

The only way to run this Effect is to eventually return it to main where it will ultimately be "run" by PureScript. I've placed the word ***run*** in quotes because it doesn't actually run it, but instead compiles it to Javascript, which will run the effect as if it was calling runEffect for us. This is just a convenient way of thinking about it.

Next, we'll change our code to replace Identity with Effect to match our new Stack:

```
type AppStack e w s a = ExceptT e (WriterT w (StateT s Effect)) a ❶

type AppM = AppStack String String Int Unit

type StackResult = Tuple (Tuple (Either String Unit) String) Int

type AppEffects =
  { log :: String
  , state :: Int
  , result :: Maybe Unit
  }

type AppResult = Tuple (Maybe String) AppEffects

results :: StackResult -> AppResult
results (Tuple (Tuple (Left err) l) s)
  = Tuple (Just err) { log: l, state: s, result: Nothing }
results (Tuple (Tuple (Right _) l) s)
  = Tuple Nothing { log: l, state: s, result: Just unit }

runApp :: Int -> AppM -> Effect AppResult ❷
runApp s = map results ❸
  <<< flip runStateT s
  <<< runWriterT
  <<< runExceptT

log :: ∀ m. MonadTell String m => String -> m Unit
log s = tell $ s <> "\n"

app :: AppM
app = do
  log "Starting App..."
  n <- get
  when (n == 0) $ void $ throwError "WE CANNOT HAVE A 0 STATE!"
  put $ n + 1
  log "Incremented State"
  pure unit
```

❶ Replaced `Identity` with `Effect`.

❷ Added `Effect` to the Type Signature.

❸ `map` the `results` Function over the return Value of `flip runStateT s`, which is `Effect StackResult`. This call to `map` replaced the `results <<< unwrap` call when `Identity` was our base `Monad`.

Notice how we only had to make changes to 3 parts of our code. This is because of the work we did in refactoring to localize the impact of Stack changes.

Now, we'll need to rewrite `test`:

```
test :: Effect Unit
test = do
  result1 <- runApp 0 app ❶
  Console.log $ show result1
  result2 <- runApp 99 app ❶
  Console.log $ show result2
```

❶ Since `runApp` now runs in `Effect`, we need to extract its results.

The output of our new `test` Function will be identical to what it was when `Identity` was our base `Monad`.

20.7. Coding `WriterT`

We've been talking for quite some time on Monad Transformers, but, as always, what they do can seem like magic, that is, right up until the time you look at the code. And even less so, when you write the code yourself.

We're going to code `WriterT` together to help us see how it does the things that are unique to Monad Transformers.

Let's start with the Type:

```
newtype WriterT w m a = WriterT (m (Tuple a w))
```

Notice the extra Type Parameter `m` when compared with our `Writer` implementation:

```
newtype Writer w a = Writer (Tuple a w)
```

That's because `WriterT's` results are now within the Context of its underlying `Monad`, i.e. the `Monad` just below it in the Stack.

Next, we'll write `runWriterT` one line at a time:

```
runWriterT :: ∀ w m a. WriterT w m a -> ???
```

Writing this much is easy. But what does this return. With `Writer`, we simply returned a `Tuple a w`, i.e. the Pure Computation and the log, but now we have to consider the underlying `Monad`.

Remember, the underlying Monad will be run AFTER WriterT. So, we'll need to return the results in the underlying Monad's Context.

Let's add the typical Writer return result, i.e. Tuple a w, but in the underlying Monadic Context, m:

```
runWriterT :: ∀ w m a. WriterT w m a -> m (Tuple a w)
```

And now, we'll code the left-hand side of the equal sign:

```
runWriterT :: ∀ w m a. WriterT w m a -> m (Tuple a w)
runWriterT (WriterT mx) = ??? ❶
```

❶ mx reminds us that we have a Value in a Monadic Context. We know this by looking at WriterT's Data Constructor.

When we think about what to return, there aren't too many possibilities since we only have mx. The first question that pops to mind is whether mx has the right Type Signature for our return value and it does. So we're just going to return it:

```
runWriterT :: ∀ w m a. WriterT w m a -> m (Tuple a w) ❶
runWriterT (WriterT mx) = mx
```

❶ Returns a Value in the underlying Monad of WriterT, i.e. m.

Next, we'll write the Functor Instance one step at a time:

```
instance functorWriterT :: Functor (WriterT w m) where ❶
```

❶ As usual, we hold all Type Parameters constant except the last, i.e. the a.

And now for the left-hand side of the equal sign:

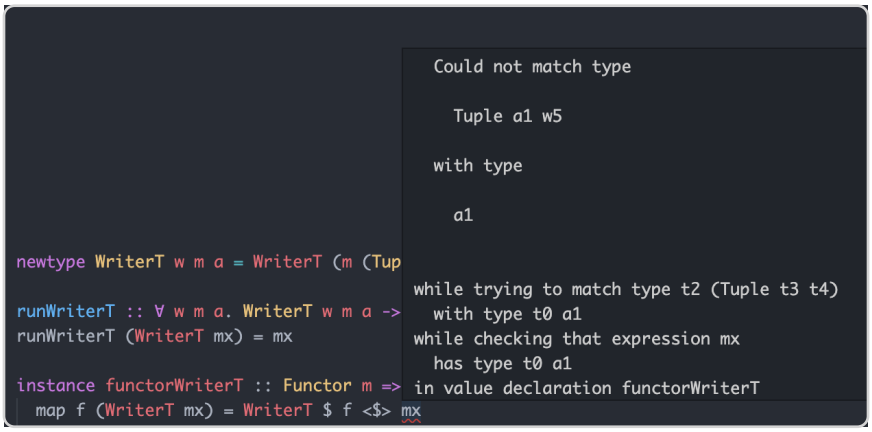
```
instance functorWriterT :: Functor (WriterT w m) where
  map f (WriterT mx) = ??? ❶
```

❶ We destructure WriterT to gain access to the Monadic Value inside.

Now, we have to apply our mapping Function, f, over the Monadic Value and then wrap things back up in a WriterT:

```
instance functorWriterT :: Functor (WriterT w m) where
  map f (WriterT mx) = WriterT $ f <$> mx -- COMPILER ERROR!!
```

We have the following error:



From the error, I can see that I forgot all about the `Tuple`. What I wrote would work if `mx :: m a` was true, but, instead, we have `mx :: m (Tuple a w)`.

Let's fix that by mapping over a Lambda to extract out the `Tuple`:

```
instance functorWriterT :: Functor (WriterT w m) where
  map f (WriterT mx) = WriterT $ mx <#> \(Tuple x w) -> ??? ❶
```

❶ Using the flipped version of `<$>`, i.e. `<#>`. This reads better when the Lambda is on the right-hand side.

Now we can apply the mapping Function, `f`, to just the first entry of the `Tuple` remembering to return a `Tuple`:

```
instance functorWriterT :: Functor (WriterT w m) where
  map f (WriterT mx) = WriterT $ mx <#> \(Tuple x w) -> Tuple (f x) w
```

Note that the extra mapping into the underlying `Monad` is where this `Functor` implementation differs from `Writer`. We can probably expect that this will hold true for all of the instances we write.

Next up is the `Apply` Instance:

```
instance applyWriterT :: Apply (WriterT w m) where ❶
```

❶ As usual, we hold all Type Parameters constant except the last, i.e. the `a`.

Now for the left-hand side of the equal sign:

```
instance applyWriterT :: Apply (WriterT w m) where
  apply (WriterT mf) (WriterT mx) = ??? ❶
```

❶ `mf` reminds us that a Function is inside the underlying `Monad`. And `mx` reminds us that a Value is inside the underlying `Monad`.

At this point, we could take the same approach we did with `Functor`, but we'll have 2 Lambdas. One for `mf` and another for `mx`. That code would be ugly and verbose.

But, we know we have a underlying `Monad`, so we can leverage the `do` notation to help keep our code cleaner:

```
instance applyWriterT :: Apply (WriterT w m) where
  apply (WriterT mf) (WriterT mx) = do
    ???
```

The first thing we'll do is extract the `Tuple` from `mf` (this time I didn't forget the `Tuple`):

```
instance applyWriterT :: Apply (WriterT w m) where
  apply (WriterT mf) (WriterT mx) = do
    Tuple f w1 <- mf
    ???
```

And now we'll extract `mx's Tuple`:

```
instance applyWriterT :: Apply (WriterT w m) where
  apply (WriterT mf) (WriterT mx) = do
    Tuple f w1 <- mf
    Tuple x w2 <- mx
    ???
```

And now we can apply `f` to `x` and wrap everything back up in the `Tuple` and then in the underlying `Monad`:

```
instance applyWriterT :: Apply (WriterT w m) where
  apply (WriterT mf) (WriterT mx) = do
    Tuple f w1 <- mf
    Tuple x w2 <- mx
    pure $ Tuple (f x) ??? ❶
```

❶ The `Tuple` Data Constructor wraps up things into a `Tuple` and `pure` wraps the `Tuple` in the underlying

Monad. The compiler will figure out which pure implementation to use based on what the underlying Monad is.

The only problem left to solve is how to combine logs and place them in the second position of the Tuple.

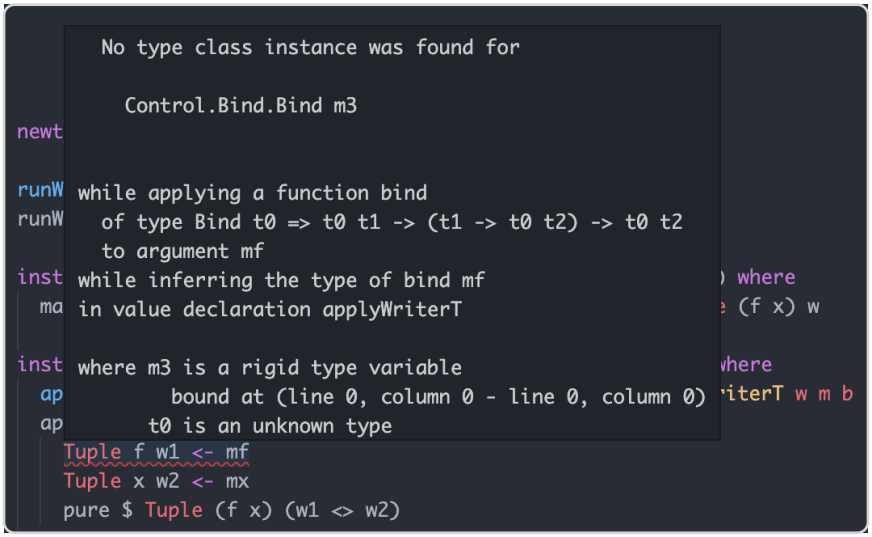
Well, that should be easy if the log Type is a Semigroup. So, let's make it one:

```
instance applyWriterT :: Semigroup w => Apply (WriterT w m) where ❶
  apply (WriterT mf) (WriterT mx) = do
    Tuple f w1 <- mf -- COMPILER ERROR!!
    Tuple x w2 <- mx
    pure $ Tuple (f x) (w1 <> w2) ❷
```

❶ Added the Semigroup Constraint.

❷ Combined logs with <>.

The compiler is complaining:

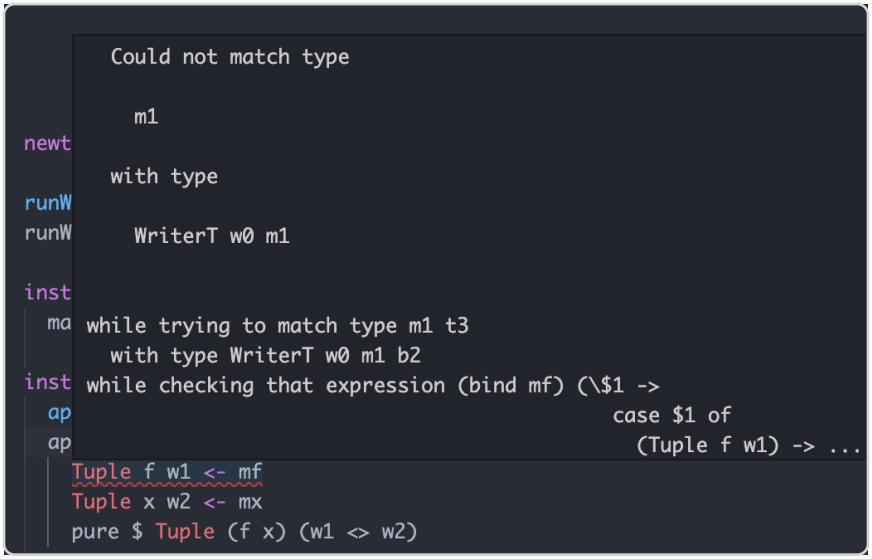


It's looking for a Bind Instance for m3, or what we called m, since we're using do notation to leverage our underlying Monad. So let's constrain m to be a Bind:

```
instance applyWriterT
    :: (Semigroup w, Bind m) => Apply (WriterT w m) where ❶
  apply (WriterT mf) (WriterT mx) = do
    Tuple f w1 <- mf -- COMPILER ERROR!!
    Tuple x w2 <- mx
    pure $ Tuple (f x) (w1 <> w2)
```

❶ A syntactical note here: We have to add multiple Constraints for a Typeclass Instance in a set of Parentheses separated by commas.

And we still haven't appeased the compiler:



The compiler found an `m1` or what we called `m` and expected a `WriterT w0 m1`. Can you see what I did wrong?

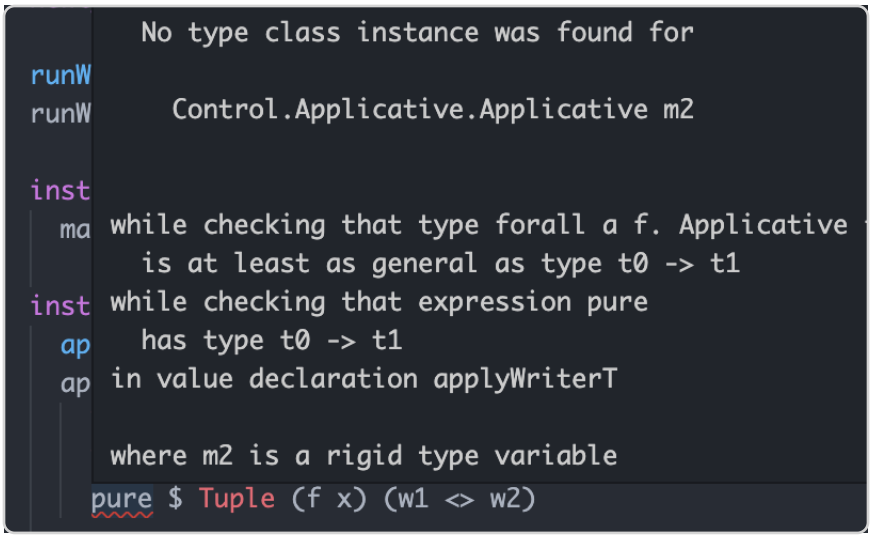
We're returning our underlying `Monad`, `m`, when `apply` should return a `WriterT w m`. The mistake I made was forgetting that the whole `do` block is a Monadic Value of our underlying `Monad`, not a `WriterT`.

Let's fix that by using `WriterT's` Data Constructor, which takes the underlying `Monad` as its one and only Parameter:

```
instance applyWriterT
    :: (Semigroup w, Bind m) => Apply (WriterT w m) where
  apply (WriterT mf) (WriterT mx) = WriterT do ❶
    Tuple f w1 <- mf
    Tuple x w2 <- mx
    pure $ Tuple (f x) (w1 <> w2) -- COMPILER ERROR!!
```

❶ Construct a new `WriterT` using the Data Constructor.

And now we have a new problem:



Since we're using `pure`, it wants `m` to have an `Applicative` Constraint. But, we already have a `Bind` Constraint. If we combine these 2 Constraints, then we have a `Monad` Constraint since they are both the Superclasses for `Monad`.

Let's change our `Bind` Constraint to a `Monad` one:

```
instance applyWriterT
    :: (Semigroup w, Monad m) => Apply (WriterT w m) where ❶
  apply (WriterT mf) (WriterT mx) = WriterT do
    Tuple f w1 <- mf
    Tuple x w2 <- mx
    pure $ Tuple (f x) (w1 <> w2)
```

❶ Changed `Bind` Constraint to `Monad`.

Looking back at what we've done, we can see that we're making sure to operate in the underlying `Monad`. This seems to be a reoccurring theme.

You may be asking if we could have skipped writing this and simply cheated by using `ap` since `WriterT` is ultimately going to be a `Monad`. And the answer is yes, but we wouldn't have learned as much.

Next, we're going to write the `Applicative` Instance:

```
instance applicativeWriterT
    :: (Semigroup w, Monad m) => Applicative (WriterT w m) where ❶
```

❶ We've proactively put the same Constraints here as we did for `Apply` since its our Superclass.

Next, the left-hand side of the equal sign:

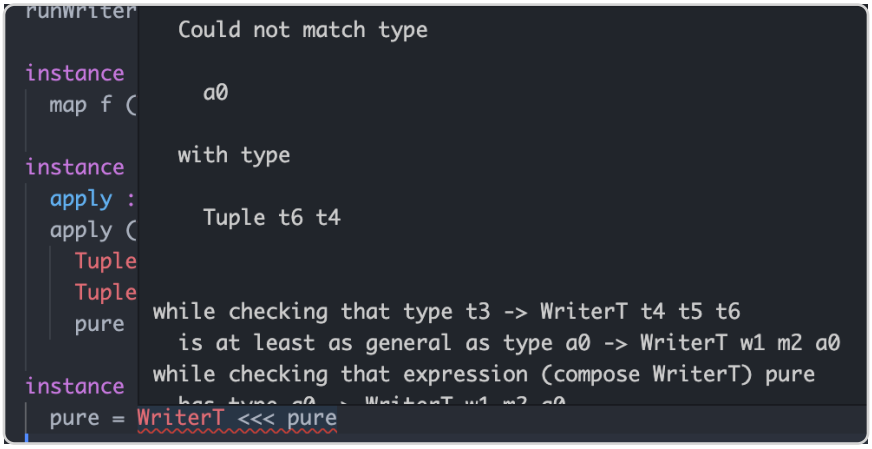
```
instance applicativeWriterT
    :: (Semigroup w, Monad m) => Applicative (WriterT w m) where
  pure x = ???
```

Now we'll need to wrap up the Pure Value ***twice***, first in the underlying `Monad` by using its `pure` and finally in a `WriterT`:

```
instance applicativeWriterT
    :: (Semigroup w, Monad m) => Applicative (WriterT w m) where
  pure = WriterT <<< pure -- COMPILER ERROR!! ❶
```

❶ Removed the `x` Parameter when changing to Point-free notation.

Looks like the compiler found something:



Again, I forgot about the `Tuple`. To fix this, we're going to need to wrap up the Pure Value in a `Tuple` first, then in the underlying `Monad` and then `WriterT`:

```
instance applicativeWriterT
    :: (Semigroup w, Monad m) => Applicative (WriterT w m) where
  pure x = WriterT $ pure $ Tuple x ??? ❶
```

❶ Had to abandon Point-free notation.

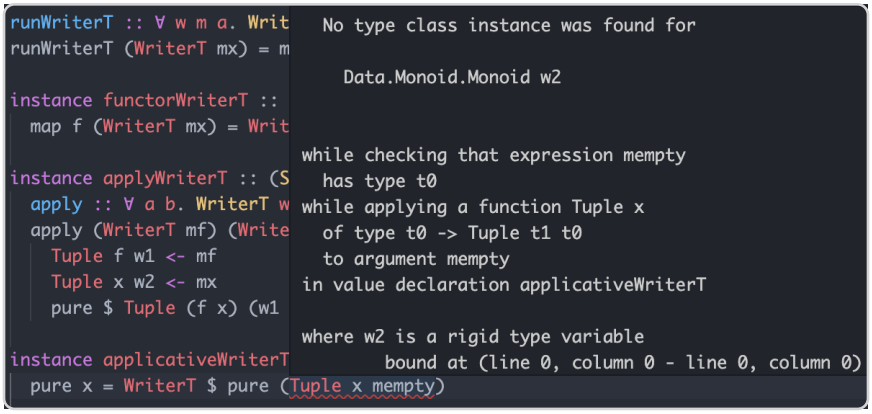
But what do we put in place of `???` for our log?

Well, we want our log to be empty for Pure Values just like we did for `Writer`. So we'll use `mempty`:

```
instance applicativeWriterT
    :: (Semigroup w, Monad m) => Applicative (WriterT w m) where
  pure x = WriterT $ pure $ Tuple x mempty -- COMPILER ERROR!! ❶
```

❶ Added `mempty`.

And still, the compiler isn't quite satisfied:



Can you see what I did wrong?

I need to upgrade our `Semigroup` to a `Monoid` now that we're using `mempty`:

```
instance applicativeWriterT
    :: (Monoid w, Monad m) => Applicative (WriterT w m) where ❶
  pure x = WriterT $ pure $ Tuple x mempty
```

❶ Changed `Semigroup` to `Monoid`.

And finally, we've quelled the compiler's objections.

Again, we had to take into consideration the underlying `Monad`. We can expect this same concern will rear it's ugly head as we code the `Bind` Instance:

```
instance bindWriterT
  :: (Semigroup w, Monad m) => Bind (WriterT w m) where ❶
```

❶ Same Constraints as `Apply` since it's our Superclass.

Next, we'll write the left-hand side:

```
instance bindWriterT :: (Semigroup w, Monad m) => Bind (WriterT w m) where
  bind (WriterT mx) f = ??? ❶ ❷
```

❶ We destructure to get at the underlying Monadic Value, mx.

❷ Remember that f is a Monadic Function, i.e. it will return a WriterT.

At this point, what would you do next?

To be honest, it's not readily apparent to me what to do next. But, what is readily apparent is that I don't know what f looks like. I don't know what its Type Signature is. So the best thing we can do to alleviate that is to write the specialized Type Signature for bind:

```
instance bindWriterT :: (Semigroup w, Monad m) => Bind (WriterT w m) where
  bind :: ∀ a b. WriterT w m a -> (a -> WriterT w m b) -> WriterT w m b ❶
  bind (WriterT mx) f = ???
```

❶ We don't need to add the w and m to the ∀ since they're defined in the instance definition. In fact, if we do add them, they will be INDEPENDENT from our earlier w and m, which is definitely not what we want.

Okay, now that we've done that, can you see what to do next?

If not, then maybe we should write down the Types:

```
mx :: m (Tuple a w)
f :: a -> WriterT w m b
```

How about now?

Looking at the Types, it's obvious that we cannot simply pass mx to f. We're going to need to extract an a from it first, not forgetting that it's inside of a Tuple:

```
instance bindWriterT :: (Semigroup w, Monad m) => Bind (WriterT w m) where
  bind :: ∀ a b. WriterT w m a -> (a -> WriterT w m b) -> WriterT w m b
  bind (WriterT mx) f = WriterT do ❶
    Tuple x w <- mx
    ???
```

❶ In PureScript, there is no need to add a $ before the do keyword. However, this is not the case in Haskell without the use of a Language Extension called ***BlockArguments***.

We now have an a, i.e. x, that we can pass to f, so let's do that:

```
instance bindWriterT :: (Semigroup w, Monad m) => Bind (WriterT w m) where
  bind :: ∀ a b. WriterT w m a -> (a -> WriterT w m b) -> WriterT w m b
  bind (WriterT mx) f = WriterT do
    Tuple x w <- mx
    ??? <- f x ❶
```

❶ `f x` is a `WriterT w m b`.

Well, we cannot `bind` a `WriterT` to anything yet since we're currently operating in the underlying `Monad`. We need to convert the `WriterT` to the underlying `Monad`. Can you think of something that might do that?

Do you remember `runWriterT` and what it returns? Let's look back at it:

```
runWriterT :: ∀ w m a. WriterT w m a -> m (Tuple a w) ❶
runWriterT (WriterT mx) = mx
```

❶ Returns a Value in the underlying `Monad` of `WriterT`, i.e. `m`.

We can safely use this on our `WriterT w m b`:

```
instance bindWriterT :: (Semigroup w, Monad m) => Bind (WriterT w m) where
  bind :: ∀ a b. WriterT w m a -> (a -> WriterT w m b) -> WriterT w m b
  bind (WriterT mx) f = WriterT do
    Tuple x w <- mx
    ??? <- runWriterT $ f x ❶
```

❶ Use `runWriterT` to get a Monadic Value in the underlying `Monad's` Context.

`runWriterT` returns a Value of Type `Tuple a w`, so now we'll capture it:

```
instance bindWriterT :: (Semigroup w, Monad m) => Bind (WriterT w m) where
  bind :: ∀ a b. WriterT w m a -> (a -> WriterT w m b) -> WriterT w m b
  bind (WriterT mx) f = WriterT do
    Tuple x w1 <- mx ❶
    Tuple y w2 <- runWriterT $ f x ❷
    ???
```

❶ Renamed `w` to `w1` since we now have `w2`.

❷ `y` is our final result. Notice how `x` was passed to `f`, i.e. it's been used to produce our final result `y`.

And finally, we're ready to return our final result, `y`, and the combined logs in a `Tuple` all wrapped up in the underlying `Monad`:

```
instance bindWriterT :: (Semigroup w, Monad m) => Bind (WriterT w m) where
  bind (WriterT mx) f = WriterT do
    Tuple x w1 <- mx
    Tuple y w2 <- runWriterT $ f x
    pure $ Tuple y $ w1 <> w2 ❶
```

❶ `pure` wraps up our `Tuple` in the underlying `Monad`.

And as always, the `Monad` Instance is pretty trivial:

```
instance monadWriterT :: (Monoid w, Monad m) => Monad (WriterT w m)
```

Notice, we had to make `w` a `Monoid` here. Do you know why?

It's because a `Monad` is a `Bind` and an `Applicative`. `Bind` is only a `Semigroup` for `w`, but `Applicative` is `Monoid`. Since a `Monoid` is also a `Semigroup`, we only need to specify `Monoid` to meet both requirements.

20.8. Coding ReaderT

Coding `ReaderT` should be similar to coding `WriterT` with the notable exception that `ReaderT` contains a Function instead of a Value.

Let's start with the Type:

```
newtype ReaderT r m a = ReaderT (r -> m a)
```

Notice `ReaderT` contains a Monadic Function that returns a Value in the Context of the underlying `Monad`, `m`.

Compare that with `Reader`:

```
newtype Reader r a = Reader (r -> a)
```

Next, let's write `runReaderT` since we're probably going to need it in `bind` like we did when we wrote `WriterT`:

```
runReaderT :: ∀ r m a. ReaderT r m a -> (r -> m a) ❶
runReaderT (ReaderT mf) = mf ❷
```

❶ Returns a Monadic Function that returns a Value in the underlying `Monad`, i.e. `m`.

❷ The name `mf` reminds us that it's a Monadic Function, which means it's a Function of the form `a -> m b`

or specifically `r -> m a`.

And now we can write the `Functor` starting with its Type Signature:

```
instance functorReaderT :: Functor (ReaderT r m) where
```

And now the left-hand side of the equal sign:

```
instance functorReaderT :: Functor (ReaderT r m) where
  map f (ReaderT mg) = ??? ❶
```

❶ We destructure the `ReaderT` to get access to the underlying Function. The name `mg` reminds us that we have a Monadic Function.

Next, we use our pattern for constructing Monads of Functions, which is something we didn't do with `WriterT` since it's a Monad of a Value:

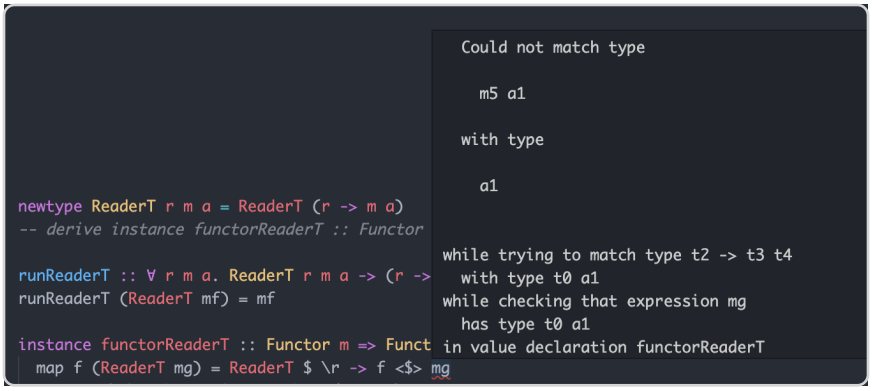
```
instance functorReaderT :: Functor (ReaderT r m) where
  map f (ReaderT mg) = ReaderT \r -> ???
```

Now it's time to think. We have a Function, `mg`, in the underlying `Monad` and we have a Function `f` we'd like to `map` with. So why not just rely on the underlying `Monad's map` to do the dirty work for us?

I can't think of any reason why that won't work:

```
instance functorReaderT :: Functor (ReaderT r m) where
  map f (ReaderT mg) = ReaderT \r -> f <$> mg -- COMPILER ERROR!!
```

And, of course, the compiler begs to differ:



Well, it looks like I've got a Type mismatch. But it's not readily clear what I did wrong. Can you see where I went wrong?

When we get a Type mismatch and we cannot figure out why, it's always good to write down the Types:

```
f :: a -> b
mg :: r -> m a
```

Okay, now I can see that I'm trying to `map` over the `a` in `mg` by simply mapping. But `mg` is a Function not an `m a`.

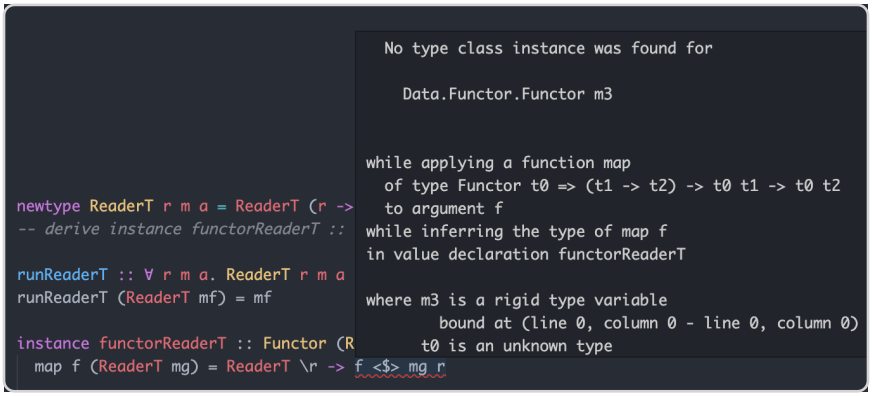
Instead, I FIRST need to call the Function, `mg`, and then I'll have the `m a` that I want:

```
f :: a -> b
mg :: r -> m a
mg r :: m a
```

So let's do that:

```
instance functorReaderT :: Functor (ReaderT r m) where
  map f (ReaderT mg) = ReaderT \r -> f <$> mg r -- COMPILER ERROR!!
```

And now, we have a brand new compiler error:



There's no Functor Instance for `m3` or what we called `m`. That's because I'm thinking of `m` as a `Monad` or in this case a `Functor` since all I use on it is `<$>`. But the compiler can't read my mind.

So, we'll fix my mistake by adding a `Functor` Constraint on `m`:

```
instance functorReaderT :: Functor m => Functor (ReaderT r m) where ❶
  map f (ReaderT mg) = ReaderT \r -> f <$> mg r
```

❶ Added the `Functor` Constraint since we're using `<$>`.

And now we're ready to move on to the `Apply` Instance:

```
instance applyReaderT :: Functor m => Apply (ReaderT r m) where ❶
```

❶ We proactively added the `Functor` Constraint since our `Functor` Instance has it and it's our Superclass.

Next, we'll add the left-hand side:

```
instance applyReaderT :: Functor m => Apply (ReaderT r m) where
  apply (ReaderT fmf) (ReaderT fmx) = ??? ❶ ❷
```

❶ The name `fmf` reminds us that we have a Function that returns a Function in the underlying `Monad`.

❷ The name `fmx` reminds us that we have a Function that returns a Value in the underlying `Monad`.

We'll add our common pattern:

```
instance applyReaderT :: Functor m => Apply (ReaderT r m) where ❶
  apply (ReaderT fmf) (ReaderT fmx) = ReaderT \r -> ???
```

Now, we have to stop and think. We want to apply `fmf` to `fmx` but only AFTER we call the functions to get their return values. We don't want to forget to call the functions like I did last time.

And since `fmf` returns a Function and `fmx` returns a Value, both of which are in the underlying `Monad`, we can just leverage the underlying `Monad's` implementation for `<*>`:

```
instance applyReaderT :: Functor m => Apply (ReaderT r m) where
  -- COMPILER ERROR!!
  apply (ReaderT fmf) (ReaderT fmx) = ReaderT \r -> fmf r <*> fmx r ❶
```

❶ Leveraging the underlying `Monad's apply` implementation.

And now the compiler has an objection:

```
newtype ReaderT r m a = ReaderT (r -> m a)
-- derive instance functorReaderT :: Functor m => Fu

runReaderT :: ∀ r m a. ReaderT r m a -> (r -> m a)
runReaderT (ReaderT mf) = mf

instance functorReaderT :: Functor m => Functor (Rea
  map f (ReaderT mg) = ReaderT \r -> f <$> mg r

instance applyReaderT :: Functor m => Apply (ReaderT
  apply (ReaderT fmf) (ReaderT fmx) = ReaderT \r -> fmf r <*> fmx r
```

```
  No type class instance was found for

    Control.Apply.Apply m3


while applying a function apply
  of type Apply t0 => t0 (t1 -> t2) -> t0 t1 -> t0 t2
  to argument fmf r
while inferring the type of apply (fmf r)
in value declaration applyReaderT

where m3 is a rigid type variable
        bound at (line 0, column 0 - line 0, column 0)
      t0 is an unknown type
```

We're missing an `Apply` instance for our `m`. I constrained it to `Functor`, but then used `<*>`. Let's fix my

hasty mistake:

```
instance applyReaderT :: Apply m => Apply (ReaderT r m) where
  apply (ReaderT fmf) (ReaderT fmx) = ReaderT \r -> fmf r <*> fmx r
```

Of course, we can always cheat with ap, but for learning, coding it by hand is far better.

And now, we can embark on the Applicative Instance:

```
instance applicativeReaderT
  :: Apply m => Applicative (ReaderT r m) where ❶
```

❶ Proactively adding Apply Constraint since our Superclass has that Constraint. But, this time, I'll keep in mind that this Constraint may change based on our code.

Now we can code the left-hand side:

```
instance applicativeReaderT
    :: Apply m => Applicative (ReaderT r m) where
  pure x = ???
```

And now our pattern:

```
instance applicativeReaderT
    :: Apply m => Applicative (ReaderT r m) where
  pure x = ReaderT \r -> ???
```

And now we need to think a little harder. We need to return an m a where m is the underlying Monad. And all Monads are also Applicatives. So why not do what we've done before and pass the buck to the underlying Monad?

Since it's worked for us before, let's try it, but this time remembering to adjust the Constraint on m:

```
instance applicativeReaderT
    :: Applicative m => Applicative (ReaderT r m) where ❶
  pure x = ReaderT \r -> pure x ❷
```

❶ Changed the Constraint on m from Apply to Applicative, since we're using the underlying Monad's pure implementation.

❷ Delegating to the underlying Monad's pure.

Looking at what we've written, we realize that we're not using `r`. Let's remedy that:

```
instance applicativeReaderT
    :: Applicative m => Applicative (ReaderT r m) where
  pure x = ReaderT \_ -> pure x ❶
```

❶ Removed `r`.

And now, our Lambda is just `const`:

```
instance applicativeReaderT
    :: Applicative m => Applicative (ReaderT r m) where
  pure x = ReaderT $ const $ pure x
```

And now we'll ETA-reduce this canceling `x` on both sides remembering to convert `$` to `<<<`:

```
instance applicativeReaderT
    :: Applicative m => Applicative (ReaderT r m) where
  pure = ReaderT <<< const <<< pure
```

Since we can no longer reduce this, let's tackle `Bind`:

```
instance bindReaderT :: Apply m => Bind (ReaderT r m) where ❶
```

❶ The `Apply` Constraint is because `Apply` is our Superclass and it has the same Constraint. I have a hunch that may wind up changing this to `Bind`. Time will tell.

Next, the left-hand side:

```
instance bindReaderT :: Apply m => Bind (ReaderT r m) where
  bind (ReaderT fmx) f = ??? ❶
```

❶ The name `fmx` is to remind us that we have a Function that will return a Value in the underlying `Monad`.

We use our typical pattern to start the right-hand side:

```
instance bindReaderT :: Apply m => Bind (ReaderT r m) where
  bind (ReaderT fmx) f = ReaderT \r -> ???
```

One thing we can steal from `WriterT's` implementation is the fact that we coded in the underlying `Monad`

via a do block. So let's start there even though it may prove to be a dead end or overkill. We'll find out soon enough either way:

```
instance bindReaderT :: Apply m => Bind (ReaderT r m) where
  bind (ReaderT fmx) f = ReaderT \r -> do
    ???
```

And if history is any help here, we can expect to use runReaderT at some point. If you remember, we needed runWriterT in WriterT's implementation of bind.

We also have learned that fmx is a Function that needs to be called with r. This means that fmx r will give us a Monadic Value, which we can extract with >>= or <- since we're using do notation.

Let's do that first:

```
instance bindReaderT :: Apply m => Bind (ReaderT r m) where
  bind (ReaderT fmx) f = ReaderT \r -> do
    x <- fmx r ❶
    ???
```

What Type is x here?

If you're having trouble figuring that out, here's the definition of ReaderT to help:

```
newtype ReaderT r m a = ReaderT (r -> m a)
```

fmx is the contained Function and we've passed an r to it. That means that x is of Type a, which is the exactly the Type that bind's Function, f, wants.

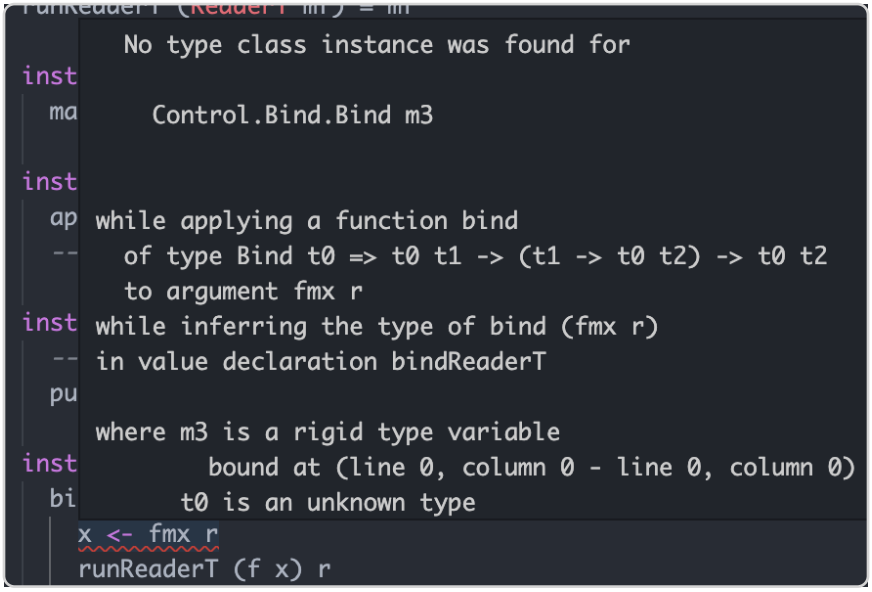
This means that we can just apply f to x, which will give us a ReaderT that we'll use runReaderT on, so that we can get a Value in the underlying Monad.

And at that point, we'll be done. So there' no need to extract and rewrap the Value. We can just end with the runReaderT call:

```
instance bindReaderT :: Apply m => Bind (ReaderT r m) where
  bind (ReaderT fmx) f = ReaderT \r -> do
    x <- fmx r -- COMPILER ERROR!!
    runReaderT (f x) r
```

Oh, no. Can you guess what I forgot to do? Don't look at the following compiler error just yet and see if you can see it.

It's a good thing we have the compiler to keep us honest. Here's the error:



I used `bind` for the underlying `Monad`, `m`, but forgot to constrain it to `Bind`. It's only constrained to be an `Apply`. Even though I tried to remember, I still forgot.

So, let's change the Constraint to `Bind`:

```
instance bindReaderT :: Bind m => Bind (ReaderT r m) where ❶
  bind (ReaderT fmx) f = ReaderT \r -> do
    x <- fmx r
    runReaderT (f x) r
```

And the compiler is content. But looking closely at the `do` block reveals that it's only 2 lines long. This means we could've written it more tersely using `>>=`.

Let's do that:

```
instance bindReaderT :: Apply m => Bind (ReaderT r m) where
  bind (ReaderT fmx) f = ReaderT \r -> fmx r >>= \x -> runReaderT (f x) r
```

Take a second to see how these 2 forms are equivalent.

I like the `>>=` version better, but you may like the `do` syntax better. This is an aesthetic choice.

And now it's time for the easiest instance:

```
instance monadReaderT :: Monad m => Monad (ReaderT r m)
```

One important thing to notice is that we've been thinking of `m` as the underlying `Monad`. But we've never needed to constrain `m` to be a `Monad` until now.

That's because a `Monad` is both a `Bind` and an `Applicative`, and in our `Applicative`, we constrained `m` to be an `Applicative` and in our `Bind`, we constrained `m` to be a `Bind`. The combinational effect of both of those Constraints ***is*** a `Monad` Constraint.

20.9. Making `ReaderT` Easier to Use

Our `ReaderT` is a Monad Transformer but it's not easy to use. We don't have a way to `lift` another `Monad` into the `ReaderT` and we haven't made any instances for the common Monad APIs.

In this section, we'll do that. First, we'll write `lift` one line at a time. But, don't forget, `lift` is a Typeclass Method from `MonadTrans`. So, we'll need to make an instance of `MonadTrans`, which has only one Method, `lift`:

```
class MonadTrans t where
  lift :: ∀ m a. Monad m => m a -> t m a
```

Let's start one line at a time:

```
instance monadTransReaderT :: MonadTrans (ReaderT r) where ❶ ❷
```

❶ We hold all Type Parameters, except the rightmost, constant.

❷ Notice that `t` in the Typeclass definition is `ReaderT r`. That's because `t` takes 2 Type Parameters, `m` and `a`. `ReaderT r` is missing 2 of its Type Parameters, `m` and `a`.

Next, we'll write the left-hand side of the equal sign:

```
instance monadTransReaderT :: MonadTrans (ReaderT r) where
  lift mx = ??? ❶
```

❶ The name `mx` reminds us that we're dealing with a Monadic Value, i.e. a Value in a Context, but not just any Context. This is a Value in `ReaderT's` underlying `Monad's` Context.

Now, we'll start with our pattern on the right-hand side:

```
instance monadTransReaderT :: MonadTrans (ReaderT r) where
  lift mx = ReaderT \r -> ???
```

Now, we have to stop and think about what `ReaderT's` Function returns. Let's look back at the `ReaderT`

definition:

```
newtype ReaderT r m a = ReaderT (r -> m a)
```

From this, we can see that it has to return a Value of Type `a` in a Context, `m`. We have exactly that in `mx`, so we can just return that:

```
instance monadTransReaderT :: MonadTrans (ReaderT r) where
  lift mx = ReaderT \r -> mx
```

Now, since we're not using `r`, we can eliminate it:

```
instance monadTransReaderT :: MonadTrans (ReaderT r) where
  lift mx = ReaderT \_ -> mx
```

And now we can replace `\_ -> mx` with `const mx`:

```
instance monadTransReaderT :: MonadTrans (ReaderT r) where
  lift mx = ReaderT $ const mx
```

And now we can compose the Data Constructor, `ReaderT`, with `const`:

```
instance monadTransReaderT :: MonadTrans (ReaderT r) where
  lift mx = (ReaderT <<< const) mx
```

And finally, ETA-reduce, i.e. cancel `mx` on the right-hand side on both sides of the equal sign:

```
instance monadTransReaderT :: MonadTrans (ReaderT r) where
  lift = ReaderT <<< const
```

And we're done.

But wait. The right-hand side of the equal sign seems awfully familiar. Didn't we have something like that when we wrote `pure`?

Let's look at `pure`:

```
instance applicativeReaderT
    :: Applicative m => Applicative (ReaderT r m) where
  pure = ReaderT <<< const <<< pure
```

Looks like we can replace `ReaderT <<< const` with `lift`:

```
instance applicativeReaderT
    :: Applicative m => Applicative (ReaderT r m) where
  pure = lift <<< pure
```

This makes sense. To create a `ReaderT` from a Pure Value, we first create the underlying `Monad`, `m`, from that Pure Value and then just lift that.

Next, we're going to write the `MonadAsk` Instance. Here's its definition:

```
class Monad m <= MonadAsk r m | m -> r where ❶
  ask :: m r ❷
```

❶ The `| m -> r` is the Functional Dependency that states if we know the `Monad`, `m`, we can determine the read-only Value, `r` since there can be ONE AND ONLY ONE `r` for this `m`. This helps remove potential ambiguities.

❷ `ask` returns the read-only Value in the Context of this `Monad`. In our case, `m` is going to be `ReaderT r m`.

Let's start with `ReaderT's` Instance one step at a time:

```
instance monadAskReaderT :: MonadAsk r (ReaderT r m) where
```

Next, we'll write the specialized Type Signature for `ask`, to help us see how to write the code:

```
instance monadAskReaderT :: MonadAsk r (ReaderT r m) where
  ask :: (ReaderT r m) r ❶
```

❶ The redundant Parentheses is there to help us see how `m`, in the Typeclass definition, is replaced by `ReaderT r m`.

Next, we'll write the left-hand side:

```
instance monadAskReaderT :: MonadAsk r (ReaderT r m) where
  ask :: ReaderT r m r
  ask = ??? ❶
```

❶ Takes no Parameters.

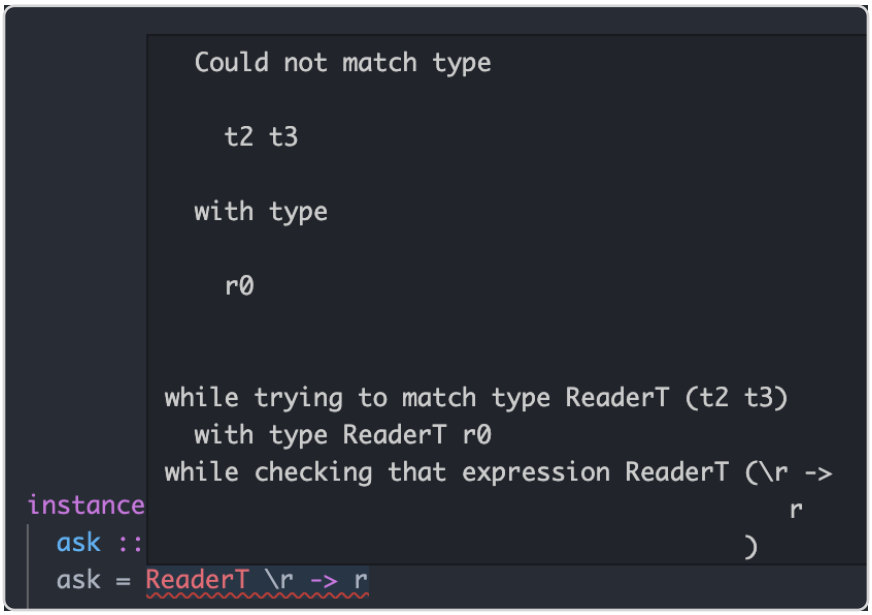
Next, we use our pattern on the right-hand side:

```
instance monadAskReaderT :: MonadAsk r (ReaderT r m) where
  ask :: ReaderT r m r
  ask = ReaderT \r -> ???
```

And now we have to think a bit. What are we supposed to return? Remember that `ask` gives us the read-only Value. So that's what we need to return:

```
instance monadAskReaderT :: MonadAsk r (ReaderT r m) where
  ask :: ReaderT r m r
  ask = ReaderT \r -> r -- COMPILER ERROR!!
```

And we have a compiler error:



The compiler error says that something of Type `t2 t3` doesn't match something of Type `r0`. Can't argue with that. But what exactly is it telling us?

Hard to say. Let's look at our `ReaderT's` definition once again to make sure we're returning the right Type:

```
newtype ReaderT r m a = ReaderT (r -> m a)
```

Can you see where I went wrong?

I returned an `a` not an `m a`. Let's fix that. But how? We need to put the `a` into Context `m`. How do we do that?

We do that by using `pure`, which we can do since `m` is a `Monad` and all `Monads` are `Applicatives`:

```
instance monadAskReaderT :: MonadAsk r (ReaderT r m) where
  ask :: ReaderT r m r
  ask = ReaderT \r -> pure r
```

And now we can ETA-reduce `\r -> pure r` to simply `pure`:

```
instance monadAskReaderT :: MonadAsk r (ReaderT r m) where
  ask :: ReaderT r m r
  ask = ReaderT pure
```

That compiles. So, now on to `MonadTell`. Here's its definition:

```
class Monad m <= MonadTell w m | m -> w where ❶
  tell :: w -> m Unit ❷
```

❶ Same Functional Dependency definition as we saw in `MonadAsk`. This is usually put in to help the compiler when there's ambiguity doing Type Inference. Without this, we'd find ourselves having to be explicit about our Types when we use `tell`.

❷ `tell` takes a `Monoid`, `w`, and appends it to its internal log. Notice how it returns a Value of Type `Unit`. That's because we're calling `tell` for its Side-effect not to have it compute something.

Now `ReaderT's` instance:

```
instance monadTellReaderT :: MonadTell w (ReaderT r m) where
```

And now the left-hand side of the implementation:

```
instance monadTellReaderT :: MonadTell w (ReaderT r m) where
  tell w = ???
```

On the right-hand side, we'll start with our usual pattern:

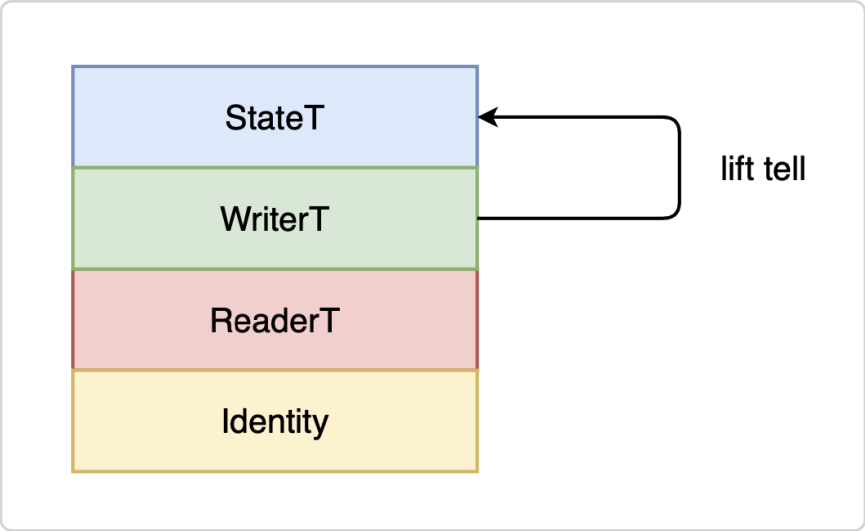
```
instance monadTellReaderT :: MonadTell w (ReaderT r m) where
  tell w = ReaderT \r -> ???
```

At this point, we have to stop and think about a few things. We are coding in ReaderT, which doesn't maintain a log since that's what WriterT's job is.

We know that ReaderT will be in a Monad Stack, which must contain a WriterT somewhere BELOW ReaderT, otherwise the call to tell would have been already handled.

Remember, we run from the top of the Stack down. If a WriterT is ABOVE our ReaderT in the Stack, then the tell call will already be handled when WriterT is run.

If that's not clear, then let's look at it in detail. Here's an example Stack where ReaderT's instance for tell will NEVER be used:



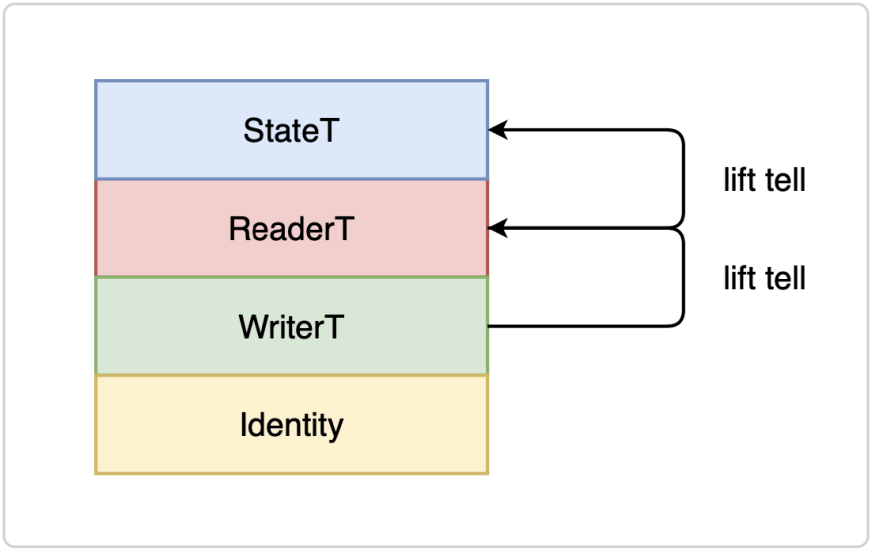
When we code in this Stack, we are coding at the top most level, i.e. at the StateT level:

```
app :: ∀ s r. StateT s (WriterT String (ReaderT r Identity)) Unit
app = do
  tell ""
  pure unit
```

The call to tell is going to be handled by StateT, which will lift the tell Function of it's underlying Monad, WriterT. And since WriterT is where tell is implemented, it will handle the call.

Notice how ReaderT, never gets involved in this process. That's because it's BELOW WriterT in the Stack.

Now let's look at a Stack where ReaderT's instance for tell WILL be called:



When we code in this Stack, we are still coding at the top most level, i.e. at the `StateT` level:

```
app :: ∀ s r. StateT s (ReaderT r (WriterT String Identity)) Unit ❶
app = do
  tell ""
  pure unit
```

❶ Our Stack definition has changed from the previous example.

Just like before, when `tell` is called, it's handled by `StateT's` instance, which `lifts` the `tell` Function of it's underlying `Monad`, which, in this case, is `ReaderT`.

Now our `ReaderT` instance, i.e. the one that we're trying to code, will be used. And it'll need to do the same thing as `StateT's` instance, i.e. `lift` the `tell` implementation from its underlying `Monad`, which, in this Stack, is `WriterT`.

So, we can safely code `ReaderT's tell` to assume that `WriterT` or some other `Monad` that acts like `WriterT` is BELOW it somewhere in the Stack.

Let's return to our code:

```
instance monadTellReaderT :: MonadTell w (ReaderT r m) where
  tell w = ReaderT \r -> ???
```

We just need to lift the underlying `Monad's` instance of `tell`. Let's look at `lift` to see what it does before we call it:

```
instance monadTransReaderT :: MonadTrans (ReaderT r) where
  lift = ReaderT <<< const
```

We can see that `lift` already creates a `ReaderT` for us. So we can remove that from our `tell`:

```
instance monadTellReaderT :: MonadTell w (ReaderT r m) where
  tell w = ???
```

And now we'll can just simply `lift` the `tell` Function:

```
instance monadTellReaderT :: MonadTell w (ReaderT r m) where
  tell w = lift $ tell w -- COMPILER ERROR!!
```

Here's the compiler's objection to our simple solution:

```
  No type class instance was found for

    Control.Monad.Monad m2


while checking that type forall m a t. MonadTrans t =>
  is at least as general as type t0 -> t1
while checking that expression lift
  has type t0 -> t1
in value declaration monadTellReaderT

where m2 is a rigid type variable
instance mo                                    bound at (line 0, column 0 - line 0, column 0)
  tell w = lift $ tell w
```

We're lifting `m` but there's no Constraint on `m` to say that it's a `Monad`. But why do we need this?

If we look at `lift's` definition again, we'll see that it constrains `m` to be a `Monad`:

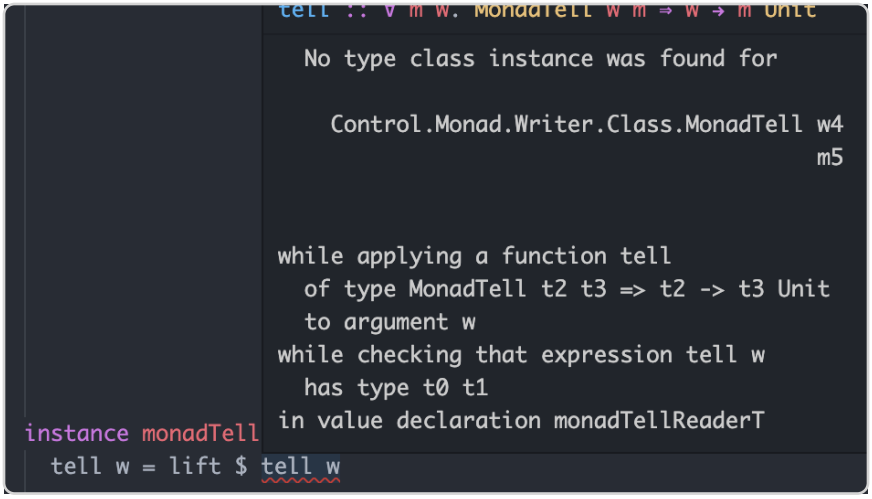
```
class MonadTrans t where
  lift :: ∀ m a. Monad m => m a -> t m a
```

Since we call `lift`, we'll need to add that same Constraint:

```
instance monadTellReaderT :: Monad m => MonadTell w (ReaderT r m) where ❶
  tell w = lift $ tell w -- COMPILER ERROR!!
```

❶ Constrain `m` to be a `Monad`.

And now we have a different error:



Notice that the compiler error is underlining the `tell` call. And the error message tells us that we're missing the `MonadTell` Constraint. This is because we're delegating to our underlying `Monad`'s implementation of `tell`.

For us to do that, the underlying `Monad`, `m`, had better support `MonadTell`.

Let's constrain `m` to be `MonadTell`:

```
instance monadTellReaderT
    :: (Monad m, MonadTell w m) => MonadTell w (ReaderT r m) where ❶
  tell w = lift $ tell w
```

❶ Don't forget the `w` in the Constraint for the first Parameter to `MonadTell`.

But wait. Isn't `MonadTell` also a `Monad`? Let's look at `MonadTell's` definition to see if we right:

```
class Monad m <= MonadTell w m | m -> w where
  tell :: w -> m Unit
```

Yes, it is. `Monad` is `MonadTell's` Superclass. So we can remove the `Monad` Constraint since `MonadTell` already handles that for us:

```
instance monadTellReaderT
    :: MonadTell w m => MonadTell w (ReaderT r m) where
  tell w = lift $ tell w
```

And for one more improvement, let's ETA-reduce and write it Point-free:

```
instance monadTellReaderT :: MonadTell w (ReaderT r m) where
  tell = lift <<< tell
```

Remember that the righthand `tell` is from `m`'s implementation so there's no infinite recursion here.

And now, we can write `MonadState` for `ReaderT`. Here's its definition:

```
class Monad m <= MonadState s m | m -> s where ❶
  state :: ∀ a. (s -> Tuple a s) -> m a ❷
```

❶ Functional Dependency that we've seen before.

❷ `state` takes a Function that takes a State, `s`, and returns both a computation, `a`, and a new State, and then it'll return that computational result in the Context of the underlying `Monad`.

The Type Signature for `state` can be confusing at first glance. But, let's imagine, just for a moment, that `get`, `put` and `modify` were part of `MonadState`:

```
-- FICTITIOUS
class Monad m <= MonadState s m | m -> s where
  get :: m s
  put :: s -> m Unit
  modify :: (s -> s) -> m s
```

Here's `StateT's` definition for reference:

```
newtype StateT s m a = StateT (s -> m (Tuple a s)) ❶
```

❶ `StateT's` contained Function returns a `Tuple` in the underlying `Monad`.

And now, we're going to implement our FICTITIOUS `MonadState` for `StateT`:

```
-- FICTITIOUS
instance monadStateStateT :: Monad m => MonadState s (StateT s m) where
  get :: StateT s m s
  get = StateT \s -> pure $ Tuple s s

  put :: s -> StateT s m Unit
  put s = StateT \_ -> pure $ Tuple unit s

  modify :: (s -> s) -> StateT s m s
  modify f = StateT \s -> let s' = f s in pure $ Tuple s' s'
```

In all 3 Methods, we use `pure` to wrap the `Tuple` in the underlying `Monad`.

Let's factor out the `pure` from each Method:

```
-- FICTITIOUS
instance monadStateStateT :: Monad m => MonadState s (StateT s m) where
  get :: StateT s m s
  get = StateT $ pure <<< \s -> Tuple s s

  put :: s -> StateT s m Unit
  put s = StateT $ pure <<< \_ -> Tuple unit s

  modify :: (s -> s) -> StateT s m s
  modify f = StateT $ pure <<< \s -> let s' = f s in Tuple s' s'
```

And notice that each Lambda is basically the same pattern. In fact, they all share the same Type Signature, i.e. `s -> Tuple a s`.

That's the exact Type Signature of the first Parameter to `state` in the REAL `MonadState`.

I can imagine that whoever coded this noticed this pattern and factored it out into a SINGLE Method called `state`. Then they factored out the other API Functions, e.g. `get`, `put`, and `modify`, to name only a few, and implemented them once as helper functions in terms of `state`:

```
get :: ∀ m s. MonadState s m => m s
get = state \s -> Tuple s s

put :: ∀ m s. MonadState s m => m Unit
put = state \_ -> Tuple unit s

modify :: ∀ m s. MonadState s m => (s -> s) -> m s
modify f = state \s -> let s' = f s in Tuple s' s'
```

Notice how each API Function is written in terms of `state`. This factoring greatly reduced the code required to implement the `MonadState` Typeclass.

Let's return from that minor detour to code up the `MonadState` instance for `ReaderT`. Here's the Typeclass definition:

```
class Monad m <= MonadState s m | m -> s where
  state :: ∀ a. (s -> Tuple a s) -> m a
```

Now, just like with `tell`, we know that `ReaderT` is not responsible for dealing with State management.

Instead, it will delegate that responsibility to its underlying `Monad` knowing that `StateT` or some other `Monad` that's ultimately responsible for State, will be below it in the Stack, otherwise it won't be called.

To accomplish this, it's going to `lift` the `state` Function from its underlying `Monad`. Let's look at how we did that with `tell` to see if we can do something similar with `state`:

```
instance monadTellReaderT
    :: MonadTell w m => MonadTell w (ReaderT r m) where
  tell = lift <<< tell
```

I don't see any reason why we cannot just do the same thing we've done here:

```
instance monadStateReaderT
    :: MonadState s m => MonadState s (ReaderT r m) where ❶
  state = lift <<< state
```

❶ We make sure to constrain `m` to support `MonadState` since we're delegating to its implementation.

And since this compiles, we can make a mental note that this may just be a pattern we can count on. For example, we could expect `WriterT's` implementation for `state` to be:

```
instance monadStateWriterT
    :: (Monoid w, MonadState s m) => MonadState s (WriterT w m) where ❶
  state = lift <<< state
```

❶ We have 2 Constraints here. First, `w` must be a `Monoid` since `WriterT's lift` implementation has that Constraint. And second, since we're lifting the underlying `Monad's state` Function, it had better be a `MonadState`.

And it turns out that this is the same implementation as the one in the PureScript library with the notable difference of ours being Point-free.

So, when we implement Methods that are outside of our responsibility, e.g. writing to the log in `ReaderT`, we can simply `lift` our underlying `Monad's` implementation.

Keep this in mind when we code `StateT` in the next Chapter.

And that completes the more common API functions. We didn't have to write `get`, `put`, `modify_`, `modify` or `asks` since they're all written in terms of the instance implementations.

By the way, there are other API Functions, but they're used far less frequently, so we're not concerned with them here. But feel free to check out `Pursuit` to see what other Typeclasses are in the Monad Transformer package.

20.10. The Power of the Combinator

A Monad Transformer is a Combinator of sorts. Just like with Parser Combinators that take other `Parsers`, Monad Transformers take other `Monads`.

And just like we saw with `Parsers`, the real power comes from combining multiple Combinators, as seen in the `Parser` we wrote called `digits`:

```
digits :: ∀ e. ParserError e => Parser e String
digits = some' digit
```

`digits` is built from the Combinator `some'`, which is a Parser Combinator that takes a `Parser`, in this case, `digit`. But let's not forget that `digit` is also built from another Parser Combinator, viz. `satisfy`:

```
digit :: ∀ e. ParserError e => Parser e Char
digit = satisfy "digit" (isDecDigit <<< codePointFromChar)
```

And when we wrote `some`, which `some'` is based on, we didn't think about all of the complexity involved with a `Parser` that was built using a `Combinator`, and whether that one was build on top of yet another Combinator, and so on.

We just worried about working with the `Parser` we'd been passed.

The same is true for Monad Transformers. They only concern themselves with the `Monad` they've been passed in their Type definitions, which is what we've been calling the underlying `Monad`.

If each `Monad` worries about its underlying `Monad`, then we can get some pretty complex behavior out of a Stack of them. And that's exactly what we've seen.

Sometimes trying to understand this complexity can make your head spin, but if you look back at the code we've written, you'll notice that we didn't get bogged down in the complexity of the whole. We only concentrated on one part at a time, i.e. the Monad Transformer we were working with.

If that Transformer worked with its underlying `Monad` properly, then it doesn't matter to us whether that `Monad` is another Transformer like `StateT`, or if it's just a plain `Monad` like `Identity` or `Effect`.

Once again, we've see how combining simple things give us a lot of power, e.g. Functional Composition, Monadic Binds, Parser Combinators and now Monad Transformers.

Chapter 21. Coding Monad Transformers

Return to the project that you've been coding in and create a file called `Ch21.purs` and add the following:

```
module Ch21 where

import Prelude ❶

import Effect (Effect)
import Effect.Console (log)

test :: Effect Unit
test = do
  log "placeholder" ❷
```

❶ Once again we are not writing anything from Prelude so we'll just import all of it.

❷ We'll delete this soon enough. It's just here to keep the compiler happy.

Next, change your `Main` module to import `Ch21` and call its `test` Function:

```
module Main where

import Prelude
import Effect (Effect)

import Ch21 as Ch21

main :: Effect Unit
main = Ch21.test
```

If you run:

```
npx spago run
```

you should see it build and print:

```
[info] Build succeeded.
placeholder
```

21.1. Writing the StateT Monad Transformer

In the previous Chapter, we wrote both WriterT and ReaderT together and now it's your turn to write a Monad Transformer from scratch, viz. StateT.

Let's get started by coding the Type. Remember that State holds a Function that takes a State Value, s, and produces a Pure Computation, a, and the Monadic Computation, i.e. the new State.

Keeping this in mind, write the StateT Type definition before reading ahead. Try to do it without looking back at how you coded State. It'll be good to see how much you've remembered.

In fact, it might be helpful if you write State's definition first. Try this before reading on and if you get stuck, then read ahead slightly for a simple hint.

If you got stuck remembering, here's State's Type definition:

```
newtype State s a = State (s -> Tuple a s)
```

If you needed this hint, stop now and try to complete the StateT Type definition before moving on.

StateT is very close to the above State definition with the biggest differences being that StateT has an underlying Monad and that the Function returns its Computation in that Monad:

```
import Data.Tuple (Tuple) ❶

newtype StateT s m a = StateT (s -> m (Tuple a s))
```

❶ Don't forget to import Tuple.

Take a second to compare this definition to State's.

Next, write runStateT. And once again, try to do it without looking at the code you've written for runState. Give it a try before reading on.

All of the run Functions that we've written so far simply unwrap the contained Function:

```
runStateT :: ∀ s m a. StateT s m a -> (s -> m (Tuple a s)) ❶
runStateT (StateT f) = f
```

❶ I left in the redundant Parentheses to help communicate that we're returning a Function.

Now, write StateT's Functor Instance and when you do, keep in mind that we have to work with the underlying Monad.

Also, remember that StateT's contained Function returns an `m (Tuple a s)`. Don't the forget the `m`.

Give it a try before reading on.

Let's start coding together one line at a time:

```
instance functorStateT :: Functor (StateT s m) where ❶
```

❶ We hold all Type Parameters constant except the last one, i.e. `a`.

And now the left-hand side of the implementation:

```
instance functorStateT :: Functor (StateT s m) where
  map f (StateT mg) = ??? ❶
```

❶ The name `mg` is to remind us that we have a Monadic Function, i.e. it will return a Pure Value in a Monadic Context.

And now we'll use the pattern when working with `Functors` of Functions:

```
instance functorStateT :: Functor (StateT s m) where
  map f (StateT mg) = StateT \s -> ???
```

At this point, we've been coding from habit. Not too much thought has gone into the code so far, but, now, we have to think and strategize our approach.

We have only 2 things to work with. We have `f` and `mg`. Go ahead and write out their Types to help you see what you're working with. Do it for yourself before you read on.

I cannot stress enough the benefit for writing out the Types. This will save you so much time, effort and greatly reduce the headaches caused by constant compiler errors.

Here are the Types of all Variables that we have in scope:

```
s :: s
f :: a -> b
mg :: s -> m (Tuple a s)
```

For us to be able to use `f` on `a`, we're going to have to get to the `a` that's buried in the `Tuple`. But we can't begin to do that without first calling the Function, `mg`, which will return the `Tuple` in question.

If you didn't finish coding, stop now and see if having these Types helps you finish coding the instance. Once you're done, continue on.

Looking at the Types, we see we must call `mg` passing it an `s`. The only Variable of Type `s` is a `s`, so we'll use it:

```
instance functorStateT :: Functor (StateT s m) where
  map f (StateT mg) = StateT \s -> mg s ???
```

Now, let's add the `mg s` Type to our list of Types:

```
s :: s
f :: a -> b
mg :: s -> m (Tuple a s)
mg s :: m (Tuple a s)
```

We need to get INSIDE the `m` to access the `Tuple`, which means we need to `map`:

```
import Data.Tuple (Tuple(..)) ❶

instance functorStateT :: Functor (StateT s m) where
  map f (StateT mg) = StateT \s -> mg s <#> \(Tuple x s') -> ??? ❷
```

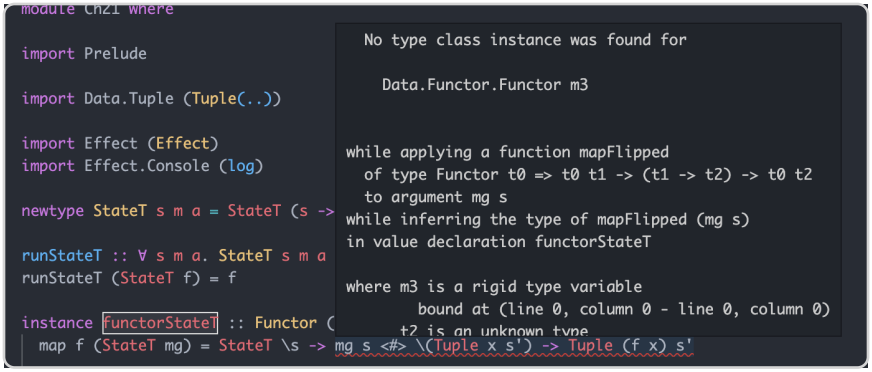
❶ Added `(..)` to get the Data Constructor since we're using it to do Pattern Matching in the Lambda.

❷ Using the flipped version of `<$>` for better readability since we're using a Lambda.

Now that we're working inside of `m`, we can apply `f` to `x`, which has Type `a`, and put it back into a `Tuple` with the new State `s'`:

```
instance functorStateT :: Functor (StateT s m) where
  -- COMPILER ERROR!!
  map f (StateT mg) =
    StateT \s -> mg s <#> \(Tuple x s') -> Tuple (f x) s'
```

But now we have a complaint from the compiler. See if you can figure out what's wrong before reading on:



If you remember, we saw this same issue when we were coding `ReaderT`. We've used `<#>`, which is `map`, which requires at least a `Functor`. So we need to add a `Functor` Constraint for `m`:

```
instance functorStateT :: Functor m => Functor (StateT s m) where
  map f (StateT mg) =
    StateT \s -> mg s <#> \(Tuple x s') -> Tuple (f x) s'
```

Notice that we only constrain `m` as much as we need for the instance code to compile. We could just make it a `Monad` and it would still work. But, the limited approach is better since it tells us what is required by the instance implementation and in our case, we've only used `map` and so a `Functor` Constraint is sufficient.

Next thing to code is the `Apply` Instance. Go as far as you can before reading more.

We'll start with the instance definition:

```
instance applyStateT :: Functor m => Apply (StateT s m) where
```

We've proactively added the same `Functor` Constraint that we have in our `Functor` Instance since it's our Superclass.

And now we'll code the left-hand side of the equal sign:

```
instance applyStateT :: Functor m => Apply (StateT s m) where
  apply (StateT fmf) (StateT fmx) = ???
```

We have named `fmf` and `fmx` as such to remind us that we dealing with Functions that returns a Function in a Monad and a Value in a Monad, respectively.

Now we can add our common pattern to create a `StateT`:

```
instance applyStateT :: Functor m => Apply (StateT s m) where
  apply (StateT fmf) (StateT fmx) = StateT \s -> ???
```

And this is usually where we have to stop and think.

Since the Types here are more complex than for map, let's write the specialized Type Signature for apply. Go ahead and write out the Type Signature specialized for StateT before reading on.

Here's the Type Signature specialized for StateT:

```
instance applyStateT :: Functor m => Apply (StateT s m) where
  apply :: ∀ a b. StateT s m (a -> b) -> StateT s m a -> StateT s m b
  apply (StateT fmf) (StateT fmx) = StateT \s -> ???
```

Now, write out the Types for fmf and fmx before reading on.

Here are the Types:

```
fmf :: s -> m (Tuple (a -> b) s) ❶
fmx :: s -> m (Tuple a s)
```

❶ Don't forget that fmf, once called, will return a Function, albeit inside of a Tuple that's inside of a Monad.

In apply, we always work with the lefthand Parameter first, i.e. fmf and then work with the right-hand side next.

Keeping that in mind along with all the Types, go ahead and see if you can finish this before reading the solution.

The next step is to call fmf and destructure the Tuple to get at the mapping Function:

```
instance applyStateT :: Functor m => Apply (StateT s m) where
  apply (StateT fmf) (StateT fmx) = StateT \s -> fmf s ???
```

And now we'll need to map to a Lambda like we did in our map implementation:

```
instance applyStateT :: Functor m => Apply (StateT s m) where
  apply (StateT fmf) (StateT fmx) = StateT \s ->
    fmf s <#> \(Tuple f s') -> ???
```

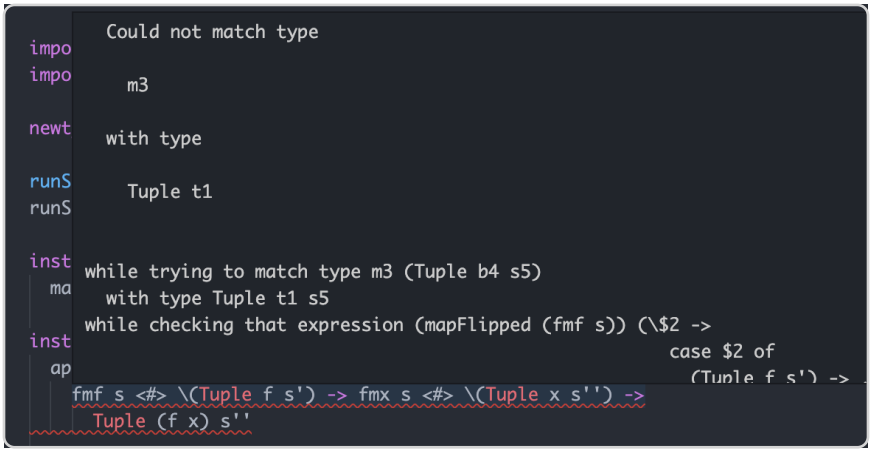
And now we do the same thing for fmx:

```
instance applyStateT :: Functor m => Apply (StateT s m) where
  apply (StateT fmf) (StateT fmx) = StateT \s ->
    fmf s <#> \(Tuple f s') -> fmx s <#> \(Tuple x s'') -> ???
```

And finally, we can put everything back together:

```
instance applyStateT :: Functor m => Apply (StateT s m) where
  -- COMPILER ERROR!!
  apply (StateT fmf) (StateT fmx) = StateT \s ->
    fmf s <#> \(Tuple f s') -> fmx s <#> \(Tuple x s'') -> Tuple (f x) s''
```

Here's the error the compiler's giving us:



Can you see what's wrong?

If not, then look closely at the part that's underlined. The compiler is seeing an `m3`, or what we called `m`, and it wants a `Tuple t1`.

Does that help?

I'm not sure about you, but it's not totally clear to me . Sometimes when this happens, I read a bit further into the error message:

```
while trying to match type m3 (Tuple b4 s5)
  with type Tuple t1 s5
```

This is a little more helpful. The underlined part's Type is supposed to be `m (Tuple b s)`. It's a `b` and not an `a` here because we've applied `f` to `x`. So, the compiler Type `m3 (Tuple b4 s5)` seems to correspond to the Type the underlined part is supposed to be.

So, we can conclude that the Type the compiler found is `Tuple t1 s5` instead. But where?

It's certainly isn't the code that it's underlined. We know that's got to be a `Monad` since we're mapping.

This error message isn't helpful. At this point, there's a few options we have at our disposal.

We can write out the Types of our code by hand. And this will work as we've seen so many times in the past.

Or we can let the compiler help us with those Types.

Since, we've written out Types by hand before, I'm going to show you a technique using Type Holes that I use from time to time when I'm too lazy to figure out complex Types in my head.

21.2. Type Holes and Undefined

Type Holes are a great tool for determining Types when you cannot figure out a Type or they are overly complex.

The problem with Type Holes is that your code must compile before the compiler will evaluate the Type Holes. This is unfortunate, since many times we're trying to code something and the compiler is complaining that our Types don't match, which is the very time we will need Type Holes to work.

So, we have a chicken and egg problem. We cannot get our code to compile because the Types don't match and we cannot figure out the Types using Type Holes since our code doesn't compile.

What we need is a way to easily get our code to compile so we can get Type help using Type Holes. But how?

We'd like to be able to type something like:

```
myFunctionThatWontCompile = someMagicalThingThatWillCompile
```

We need something that will compile no matter where we put it. And that thing is called `undefined`. It's in a library called `purescript-undefined` and here's its Type Signature:

```
undefined :: ∀ a. a
```

This function will produce an `a` out of thin air. Actually, it doesn't. It can't. No function can. But this is its Type Definition.

At this point, you're probably wondering how you can write such a Function in PureScript. The answer is you cannot.

And now, your next question is probably how did the library authors write this then. And the answer is that they wrote it in Javascript.

We haven't delved into how to write Javascript code that can be called from PureScript just yet, we'll do that in the Beyond Section of the book, but it is possible.

Here's the PureScript code:

```
module Undefined where

foreign import undefined :: forall anything. anything
```

Now, I know I said you cannot write this in PureScript and yet here it is written in PureScript. But it's using a `foreign import` which means that it expects to get this from definition from Javascript. That's what `foreign` means in `PureScript` (much more in the Beyond Section of this book).

And for the impatient, here's the Javascript code it uses:

```
"use strict";

exports.undefined = undefined
```

`undefined` will almost always crash your program, but that's okay. We're not going to run our program with `undefined` in it, unless we forget to remove it.

We only want it to make the compiler happy. Since it returns an `a` (or `anything`), it will unify with ANY Type the compiler is looking for. This breaks the chicken and egg cycle and allows the compiler to evaluate our Type Holes.

To use `undefined`, we'll need to edit our `spago.dhall` file:

```
{-
Welcome to a Spago project!
You can edit this file as you like.
-}
{ name = "my-project"
, dependencies =
  [ "arrays"
  , "bifunctors"
  , "console"
  , "contravariant"
  , "control"
  , "effect"
  , "either"
  , "foldable-traversable"
  , "integers"
  , "lists"
  , "maybe"
  , "newtype"
  , "nonempty"
  , "prelude"
  , "profunctor"
  , "psci-support"
  , "strings"
  , "tuples"
  , "undefined" ❶
  , "unfoldable"
  , "unicode"
  ]
, packages = ./packages.dhall
, sources = [ "src/**/*.purs", "test/**/*.purs" ]
}
```

❶ Added `undefined`. I added here because I like to keep my libraries in alpha order.

Then comment out the `apply` code TEMPORARILY so that our code will build with NO errors, save your file and then run:

```
npx spago build
```

This will install and build the new library and your program. We commented out the bad code so we wouldn't get any errors.

We'll ignore this warning for obvious reasons:

```
[warn] None of your project files import modules from some projects that are
in the direct dependencies of your project.
These dependencies are unused. To fix this warning, remove the following
packages from the list of dependencies in your config:
- undefined
```

Now would be a good time to UNCOMMENT the `apply` Function. So do that now.

One last thing to mention is that Haskell has `undefined` built into its `Prelude`. So you can use this technique there as well. PureScript decided to force us to explicitly use a library, most likely for safety reasons.

At this point, we can return to `apply` to use our new technique.

21.3. Back to Writing the `StateT` Monad Transformer

Here's where we left off:

```
instance applyStateT :: Functor m => Apply (StateT s m) where
  -- COMPILER ERROR!!
  apply (StateT fmf) (StateT fmx) = StateT \s ->
    fmf s <#> \(Tuple f s') -> fmx s <#> \(Tuple x s'') -> Tuple (f x) s''
```

Let's change the code to use `undefined`:

```
import Undefined (undefined)

instance applyStateT :: Functor m => Apply (StateT s m) where
  apply (StateT fmf) (StateT fmx) = StateT \s ->
    let x = (fmf s <#> \(Tuple f s') -> fmx s <#> \(Tuple x s'') ->
              Tuple (f x) s'') :: ?x ❶ ❷
    in undefined ❸
```

❶ Since we're using Lambda's, it's important to add Parentheses so we're getting the Type for all of our code. Without them, `?x` would return the Type for the right-hand side of the rightmost Lambda, which is `Tuple (f x) s''` and we already know its Type.

❷ We use `?x` to create a named Type Hole.

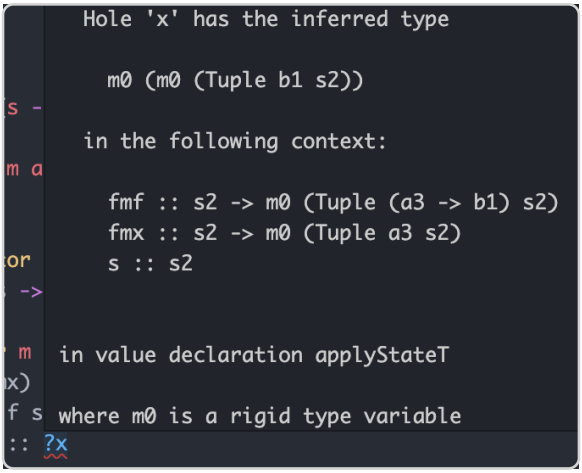
❸ `undefined` is the body of our Function.

By using the `let` expression, we can assign some random variable, here `x`, to the code we were trying to figure out it's Type Signature for, not forgetting to add the Type Hole, of course.

We're not going to use x and the compiler has no preconceived ideas about its Type. So it will always compile.

Then we simple put undefined in the in expression, which will always compile.

After making these changes, our Function compiles and the Type Hole causes the following compiler error:



One nice thing about Type Holes is that below the Inferred Type is a list of all of the Types for all the other Variables that are involved in the code that's part of the Type Hole.

But the real news here is in the Type that it inferred:

```
m0 (m0 (Tuple b1 s2))
```

We have our underlying Monad in our underlying Monad. Not good. But how did this happen?

See if you can see where I went wrong by looking back at our code before we starting using undefined:

```
instance applyStateT :: Functor m => Apply (StateT s m) where
  -- COMPILER ERROR!!
  apply (StateT fmf) (StateT fmx) = StateT \s ->
    fmf s <#> \(Tuple f s') -> fmx s <#> \(Tuple x s'') -> Tuple (f x) s''
```

Can you see why I have nested underlying Monads?

It's because I'm mapping TWICE! This gives us a Monad in the SAME Monad. So, how do we fix this?

If you're inclination to use join, you're on the right track. But remember, join is written in terms of bind and with bind, we never have this problem.

So, why not just use bind?

Let's start over:

```
instance applyStateT :: Functor m => Apply (StateT s m) where
  apply (StateT fmf) (StateT fmx) = StateT \s -> ???
```

At this point, we're going want to start with a do block:

```
instance applyStateT :: Functor m => Apply (StateT s m) where
  apply (StateT fmf) (StateT fmx) = StateT \s -> do
    ???
```

Now, we're going to call fmf and extract its Tuple:

```
instance applyStateT :: Functor m => Apply (StateT s m) where
  apply (StateT fmf) (StateT fmx) = StateT \s -> do
    Tuple f s' <- fmf s
    ???
```

And now, we'll call fmx and extract its Tuple:

```
instance applyStateT :: Functor m => Apply (StateT s m) where
  apply (StateT fmf) (StateT fmx) = StateT \s -> do
    Tuple f s' <- fmf s
    Tuple x s'' <- fmx s' ❶
    ???
```

❶ Do NOT forget to pass the updated State to fmx, i.e. the State that's returned and possibly updated by fmf.

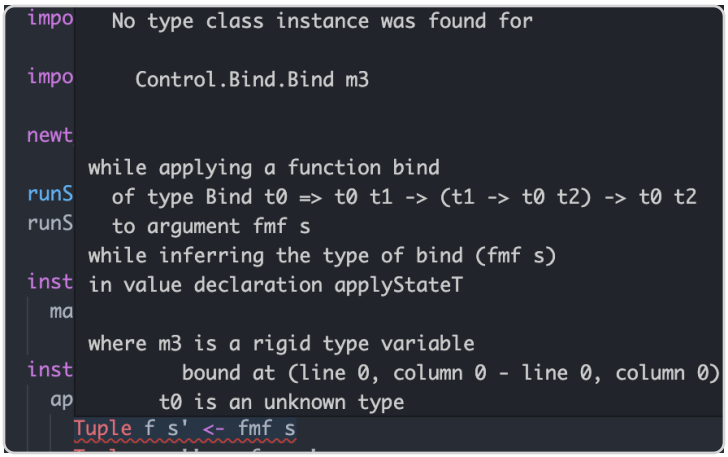
And now we can apply f to x and wrap everything up in a Tuple and then wrap that up in the underlying Monad:

```
instance applyStateT :: Functor m => Apply (StateT s m) where
  apply (StateT fmf) (StateT fmx) = StateT \s -> do
    Tuple f s' <- fmf s -- COMPILER ERROR!!
    Tuple x s'' <- fmx s'
    pure $ Tuple (f x) s'' ❶
```

❶ Make sure to return the final State s'' since it may have been updated by fmx.

And after all of that…

We have a compiler error:

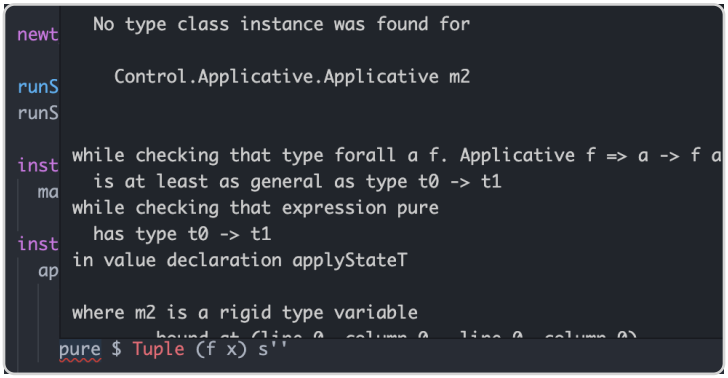


And this one is an easy one. We've used `do` syntax to code in the underlying `Monad`, so we'll have to ensure that `m` is more than just a `Functor`. So, we'll change the `Functor` Constraint to `Bind`:

```
instance applyStateT :: Bind m => Apply (StateT s m) where ❶
  apply (StateT fmf) (StateT fmx) = StateT \s -> do
    Tuple f s' <- fmf s
    Tuple x s'' <- fmx s'
    pure $ Tuple (f x) s'' -- COMPILER ERROR!!
```

❶ Replace `Functor` Constraint with `Bind`.

And now the compiler isn't happy for a different reason:



Another easy error to fix. Since we're using `pure` to construct our underlying `Monad`, it needs to be an `Applicative` too.

But wait, it's already a `Bind` and a `Bind` and an `Applicative` is a `Monad`. That means that all we have to do is change our Constraint from `Bind` to `Monad` and we should be good:

```
instance applyStateT :: Monad m => Apply (StateT s m) where ❶
  apply (StateT fmf) (StateT fmx) = StateT \s -> do
    Tuple f s' <- fmf s
    Tuple x s'' <- fmx s'
    pure $ Tuple (f x) s''
```

❶ Replace `Bind` Constraint with `Monad`.

Don't forget to remove the import for `Undefined` since we don't need it anymore. Well, I can't say that we won't need it later, but we should remove it just the same so we don't forget and leave it behind.

Luckily, the compiler will tell us when an import is "redundant", i.e. not necessary.

And we probably want to remove it from our `spago.dhall` so `spago` won't bug us with warnings on every single build that we aren't using the `undefined` package.

Now, we're ready to write the `Applicative` Instance. The `Applicative` is usually pretty easy, so please don't read further until you've given it a try.

Here's the instance definition:

```
instance applicativeStateT :: Monad m => Applicative (StateT s m) where
```

The Constraint is the same as the Constraint we used on our `Apply` Instance since it's `Applicative's` Superclass. If we forget this, the compiler will be sure to remind us.

Next, we'll start the implementation with our typical pattern:

```
instance applicativeStateT :: Monad m => Applicative (StateT s m) where
  pure x = StateT \s -> ???
```

The `StateT` Function must return a `Tuple` in the underlying `Monad`, so we should add that:

```
instance applicativeStateT :: Monad m => Applicative (StateT s m) where
  pure x = StateT \s -> pure $ Tuple ??? ???
```

The State will not change as a result of running this, so we'll just return `s` in the second position.

```
instance applicativeStateT :: Monad m => Applicative (StateT s m) where
  pure x = StateT \s -> pure $ Tuple ??? s
```

The first position of the Tuple is the Pure Computation. In the case of pure, it's just x our Pure Value. So, we'll add that to the first position:

```
instance applicativeStateT :: Monad m => Applicative (StateT s m) where
  pure x = StateT \s -> pure $ Tuple x s
```

The next thing you should code up is the Bind Instance. Please give it a try before advancing any further.

Here's the instance definition:

```
instance bindStateT :: Monad m => Bind (StateT s m) where
```

The Constraint is a Monad since Applicative, our Superclass, has that same Constraint.

Next, the left-hand side of the expression:

```
instance bindStateT :: Monad m => Bind (StateT s m) where
  bind (StateT fmx) f = ??? ❶
```

❶ The name fmx is to remind us that we have a Function that will return a Monadic Value.

Now, we're going to use our common pattern:

```
instance bindStateT :: Monad m => Bind (StateT s m) where
  bind (StateT fmx) f = StateT \s -> ???
```

Now what?

What is the next step? And how do you figure out what to do next here?

To answer this question, you first need to state what you HAVE and what you WANT. We'll do that with Types. But before we do, we should add the Type Signature for bind specialized for StateT:

```
instance bindStateT :: Monad m => Bind (StateT s m) where
  bind :: ∀ a b. StateT s m a -> (a -> StateT s m b) -> StateT s m b
  bind (StateT fmx) f = StateT \s -> ???
```

Now we can list what we HAVE followed by what we WANT with particular attention paid to the Types:

```
-- HAVE
fmx :: s -> m (Tuple a s)
f :: a -> StateT s m b
s :: s
\s -> ??? :: s -> m (Tuple b s)

-- WANT
StateT \s -> ??? :: StateT s m b
```

If the Type for `fmx` isn't clear, then remember that the Function `StateT` contains, `fmx`, is of Type `StateT s m a` (see specialized Type Signature for `bind`). Once you see that, `fmx's` Type Signature then comes from `StateT's` Type definition:

```
newtype StateT s m a = StateT (s -> m (Tuple a s))
```

Next question to ask is how can I combine the things I HAVE to give me the things I WANT. Take a look at what we have and see if you can combine them to get us closer to what we want.

When I do this, I think about the Monadic Function, `f`, and how it needs an `a` before I can use it. Then I look around and the only `a` that's available to me is in `fmx`.

But before I can get the `a` I have to first call `fmx` with an `s`. So we'll write down what we can compute with `fmx` and `f`:

```
-- HAVE
fmx :: s -> m (Tuple a s)
f :: a -> StateT s m b
s :: s
\s -> ??? :: s -> m b

-- COMPUTED
fmx s :: m (Tuple a s)

-- WANT
StateT \s -> ??? :: StateT s m b
```

We'll that got us one step closer to the `a`, but it's locked up in our underlying `Monad`. But I don't know anything about that `Monad`, other than the fact that it's a `Monad`.

But that fact is quite useful. That means I can use any `Monad` Typeclass Methods to get at that `a`.

So what's the easiest way to get to that `a`?

If you're thinking like I'm thinking then you're thinking about >>=:

```
instance bindStateT :: Monad m => Bind (StateT s m) where
  bind :: ∀ a b. StateT s m a -> (a -> StateT s m b) -> StateT s m b
  bind (StateT fmx) f = StateT \s -> fmx s >>= \(Tuple x s') -> ??? ❶
```

❶ Remember that the contained Function in StateT returns Tuple a s.

Now that we have access to a Value of Type of a, i.e. x, we can finally pass it to f:

```
instance bindStateT :: Monad m => Bind (StateT s m) where
  bind :: ∀ a b. StateT s m a -> (a -> StateT s m b) -> StateT s m b
  bind (StateT fmx) f = StateT \s -> fmx s >>= \(Tuple x s') -> f x ???
```

So what Type is f x?

Looking back at the Types we wrote down, we can see that f is of Type a → StateT s m b. That means f x is Type StateT s m b.

But remember that StateT's contained Function returns an m (Tuple a s). So, we have StateT s m b with f x, but we want to return m (Tuple a s).

How can we get from one to the other?

Remember runStateT? If not, here it is again:

```
runStateT :: ∀ s m a. StateT s m a -> s -> m (Tuple a s) ❶
runStateT (StateT f) = f
```

❶ Removed redundant Parentheses we had earlier.

This is the perfect function to consume our StateT s m b and produce what we need, i.e. m (Tuple b s).

We just need to provide the current State as the second Parameter, which we can get from the Tuple we destructured to extract s':

```
instance bindStateT :: Monad m => Bind (StateT s m) where
  bind :: ∀ a b. StateT s m a -> (a -> StateT s m b) -> StateT s m b
  bind (StateT fmx) f =
    StateT \s -> fmx s >>= \(Tuple x s') -> runStateT (f x) s'
```

And it compiles.

For the last piece, code up the Monad instance. This is by far the easiest to do, so you might not put much thought into the code. But, keep in mind, we have to add the proper Constraints.

And since Monad is both a Bind and an Applicative, you'll need to take both of those instances's Constraints into consideration.

Go ahead and give it a try before reading on.

Both Applicative and Bind have the Monad Constraint on m, so we'll need to add that for our Monad instance:

```
instance monadStateT :: Monad m => Monad (StateT s m)
```

And that's it. A lot of thinking and a few compiler errors but we now have the following Monad Transformer:

```
newtype StateT s m a = StateT (s -> m (Tuple a s))

runStateT :: ∀ s m a. StateT s m a -> (s -> m (Tuple a s))
runStateT (StateT f) = f

instance functorStateT :: Functor m => Functor (StateT s m) where
  map f (StateT mg) =
    StateT \s -> mg s <#> \(Tuple x s') -> Tuple (f x) s'

instance applyStateT :: Monad m => Apply (StateT s m) where
  apply (StateT fmf) (StateT fmx) = StateT \s -> do
    Tuple f s' <- fmf s
    Tuple x s'' <- fmx s'
    pure $ Tuple (f x) s''

instance applicativeStateT :: Monad m => Applicative (StateT s m) where
  pure x = StateT \s -> pure $ Tuple x s

instance bindStateT :: Monad m => Bind (StateT s m) where
  bind :: ∀ a b. StateT s m a -> (a -> StateT s m b) -> StateT s m b
  bind (StateT fmx) f =
    StateT \s -> fmx s >>= \(Tuple x s') -> runStateT (f x) s'

instance monadStateT :: Monad m => Monad (StateT s m)
```

There's a lot of work behind those 16 lines of code. But, once written, this code should pay us in dividends every time we use it.

But, not before we implement the Transformer APIs. We're not going to implement all of them, only the most common:

```
MonadState
MonadAsk
MonadTell
MonadThrow
MonadError
```

Let's take a look at them one at a time. We'll start with `MonadState`, but before we do, add `transformers` to your `spago.dhall` file.

Then run:

```
npx spago build
```

This will install and build the new library and your program.

We'll ignore the warnings. There are 2 types of warnings here. Libraries that haven't upgraded their code to account for the deprecated features in PureScript version 14 and the package we just added but have yet to use.

Now we're ready to write the `MonadState` Instance for `StateT`. Here's its definition for reference:

```
class Monad m <= MonadState s m | m -> s where
  state :: ∀ a. (s -> Tuple a s) -> m a
```

Use this to help you code the `MonadState` Instance for `StateT` before reading the solution.

We start with the instance definition:

```
import Control.Monad.State.Class (class MonadState) ❶

instance monadStateStateT :: MonadState s (StateT s m) where ❷
```

❶ You'll need to import the class definition.

❷ We need the `Monad` Constraint because `MonadState` has `Monad m` as its Superclass.

Then we write the left-hand side of the implementation using the definition for the `state` Method as our guide:

```
instance monadStateStateT :: MonadState s (StateT s m) where
  state f = ???
```

Now, we'll use our pattern:

```
instance monadStateStateT :: MonadState s (StateT s m) where
  state f = StateT \s -> ???
```

Now, we'll have to stop and think what to do next. But before we do, let's just write out the Types to make our life easy:

```
f :: s -> Tuple a s
s :: s
```

These are the only Values that we have to work with. So, it's clear that the only thing we can do is apply `s` to `f`:

```
f :: s -> Tuple a s
s :: s
f s :: Tuple a s
```

And finally, we need to wrap that `Tuple` in the underlying Monad. But how do we know that?

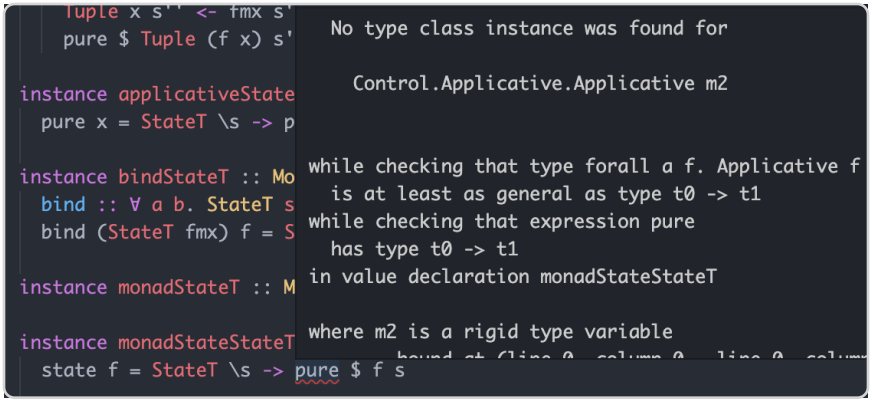
We know that because of the `StateT` definition, which shows that the contained Function is `s -> m (Tuple a s)`:

```
newtype StateT s m a = StateT (s -> m (Tuple a s))
```

So, let's take `f s` and use `pure` to wrap it in the underlying `Monad`, `m`:

```
instance monadStateStateT :: MonadState s (StateT s m) where
  state f = StateT \s -> pure $ f s -- COMPILER ERROR!!
```

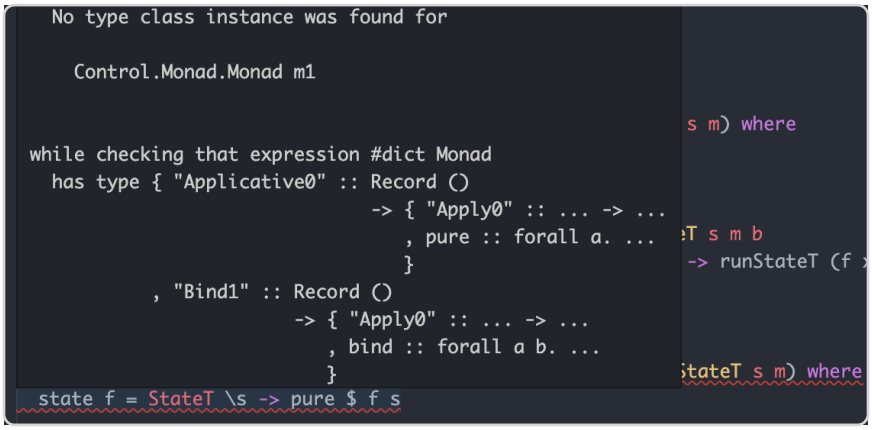
And now we unhappy compiler:



Noticing that `pure` is underlined in red, and reading the error, it's clear that we need to add the `Applicative` Constraint:

```
-- COMPILER ERROR!!
instance monadStateStateT :: Applicative m => MonadState s (StateT s m) where
  state f = StateT \s -> pure $ f s
```

And now the compiler is underlying our whole Function:



The compiler wants us to add a `Monad` instance now. But why?

Let's look at the `MonadState` Constraints:

```
class Monad m <= MonadState s m | m -> s where
  state :: ∀ a. (s -> Tuple a s) -> m a
```

We can see that it expects `m`, which, in our case, is `StateT` to be a `Monad`. And it is:

```
instance monadStateT :: Monad m => Monad (StateT s m)
```

But `StateT's` definition for `Monad` requires its `m`, that is the underlying `Monad` to be a `Monad`. This is most

likely why we're being asked to add this Constraint to our MonadState instance.

Sometimes, it's not perfectly clear why the compiler will ask us for Constraints. This is one such case. Our analysis may be wrong, i.e. the compiler could have a completely different reason for asking for this. But, our analysis has proven that it's not an unreasonable request.

We'll make that change, but we'll remove the Applicative first. Since all Monads are Applicatives, the Applicative Constraint is redundant:

```
instance monadStateStateT :: Monad m => MonadState s (StateT s m) where
  state f = StateT \s -> pure $ f s
```

Now, just because this compiles, doesn't mean we're done. This can be ETA-reduced. Before reading on to see how to do so, ETA-reduce this code, then continue on reading to see how you did.

I'm going to give you the answer first to see how you did:

```
instance monadStateStateT :: Monad m => MonadState s (StateT s m) where
  state f = StateT $ pure <<< f
```

Is this what you got?

If not, then I'll show you all the steps. We start with:

```
instance monadStateStateT :: Monad m => MonadState s (StateT s m) where
  state f = StateT \s -> pure $ f s
```

Then we change $ to <<<:

```
instance monadStateStateT :: Monad m => MonadState s (StateT s m) where
  state f = StateT \s -> (pure <<< f) s
```

Take a moment to make sure this step makes sense. If it doesn't, then it's worth looking at the compose Function with a Type Signature that's specialized for Functions:

```
compose :: ∀ b c d. (c -> d) -> (b -> c) -> (b -> d)
compose f g x = f (g x)
```

We can move the x Parameter to the right-hand side of the equal sign as a Lambda:

```
compose :: ∀ b c d. (c -> d) -> (b -> c) -> (b -> d)
compose f g = \x -> f $ g x ❶
```

❶ Changed from using Parentheses to using $.

Now, I'm going to rewrite the left-hand side of the equal sign that's not valid PureScript but is Algebraically valid:

```
f <<< g = \x -> f $ g x
```

And now, I'm going to move the x back to the left-hand side:

```
(f <<< g) x = f $ g x
```

And now I'm going to replace f with pure, g with f and x with s so that it matches our example above:

```
(pure <<< f) s = pure $ f s
```

Which is how we got from:

```
instance monadStateStateT :: Monad m => MonadState s (StateT s m) where
  state f = StateT \s -> pure $ f s
```

to:

```
instance monadStateStateT :: Monad m => MonadState s (StateT s m) where
  state f = StateT \s -> (pure <<< f) s
```

And now we can ETA-reduce it by canceling s on both sides:

```
instance monadStateStateT :: Monad m => MonadState s (StateT s m) where
  state f = StateT $ pure <<< f ❶
```

❶ Using $ instead of Parentheses.

So many times we rush off to code other functions the moment that the compiler shuts up. But we should always step back and look at what we've just coded and ask, "Can I this make this better?".

And if we can make it better, then we should. And, if done right, it's always worth it.

Next up is MonadAsk. Here's its definition for reference:

```
class Monad m <= MonadAsk r m | m -> r where
  ask :: m r
```

Go ahead and write the MonadAsk Instance for StateT before reading on to the solution.

Let's start with the instance definition:

```
import Control.Monad.Reader.Class (class MonadAsk) ❶

instance monadAskStateT :: MonadAsk r (StateT s m) where
```

❶ You'll need to import the class definition.

Next, we'll write the left-hand side of the equal sign:

```
instance monadAskStateT :: MonadAsk r (StateT s m) where
  ask = ??? ❶
```

❶ ask takes no Parameters.

And now, we'll use our contained Function pattern:

```
instance monadAskStateT :: MonadAsk r (StateT s m) where
  ask = StateT \s -> ???
```

We know that our return result will be wrapped up in the Context of the underlying Monad, so let's add that:

```
instance monadAskStateT :: MonadAsk r (StateT s m) where
  ask = StateT \s -> pure $ ???
```

At this point, we need to stop and think a bit.

What Type do we need to return?

If you cannot answer that, then it's time to write the specialized Type Signature. So do that and then read on.

The Type Signature specialized for StateT is as follows:

```
instance monadAskStateT :: MonadAsk r (StateT s m) where
  ask :: StateT s m r ❶ ❷
  ask = StateT \s -> pure $ ???
```

❶ Redundant Type Signature specialized for StateT.

❷ Notice that all Type Parameters, s, m and r are defined in the instance definition and therefore do not need to be put in a ∀ section of the Type Signature.

ask is a State s m r. We're going to replace r in place of a in the StateT definition to help us see what the contained Function Type is:

```
newtype StateT s m r = StateT (s -> m (Tuple r s))
```

This shows us that the contained Function need to return a Tuple r s:

```
instance monadAskStateT :: MonadAsk r (StateT s m) where
  ask :: StateT s m r ❶
  ask = StateT \s -> pure $ Tuple ??? s
```

Okay, so we're not changing the State, but where are we supposed to get a Value of Type r?

There is no r here. All we have is s. Well, actually, we have more than just s. Can you see what else we have?

We have the underlying Monad, m, here. That's our only hope for getting an r. But how can we get an r from the underlying Monad?

The only way possible is if the underlying Monad has a MonadAsk Instance. Can you see why this is important?

It's important because we can then call the underlying Monad's ask Method and return that r. Well, if we do that, then ask will return r already in the underlying Monad's Context, which means we no longer need to use pure.

Go ahead and change the above code to remove the right-hand side of the Lambda. Next, you're going to finish the Lambda by leveraging the underlying Monad's ask implementation. Do this before you read on.

Let's make those changes one step at a time:

```
import Control.Monad.Reader.Class (class MonadAsk, ask) ❶

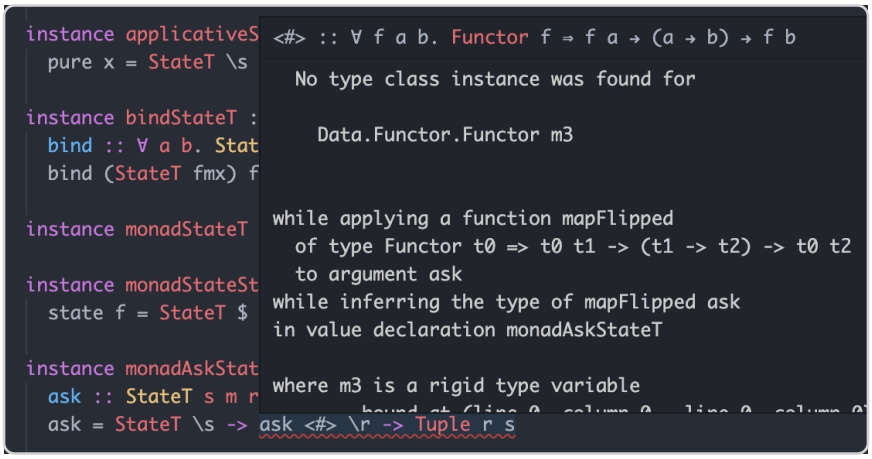
instance monadAskStateT :: MonadAsk r (StateT s m) where
  ask :: StateT s m r
  ask = StateT \s -> ask ???
```

❶ Added `ask` to the import.

Here `ask` has the Type `m r`. We need to change that `r` to a `Tuple r s` since that's what `StateT's` contained Function returns. We'll use `map` to accomplish that:

```
instance monadAskStateT :: MonadAsk r (StateT s m) where
  ask :: StateT s m r
  ask = StateT \s -> ask <#> \r -> Tuple r s -- COMPILER ERROR!!
```

And now we have a compiler problem:

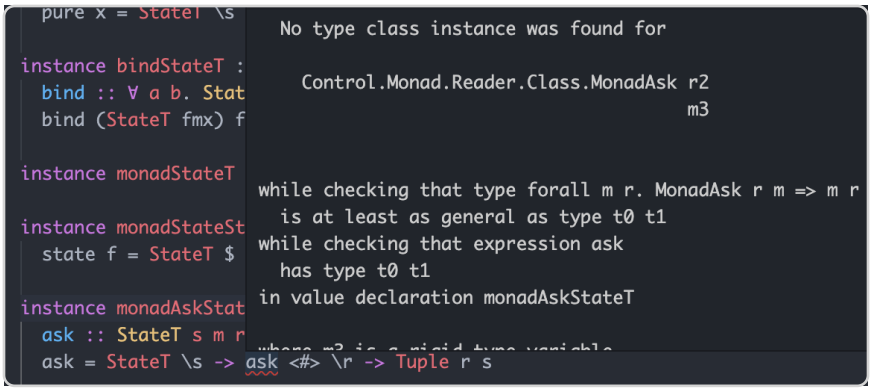


The compiler wants a `Functor` Constraint on `m3` or what we called `m`. Can you see why?

Well, it's because we mapped over the returned Value of `ask`, which is the Type `m r`. So, `m` has to be a `Functor`:

```
instance monadAskStateT :: Functor m => MonadAsk r (StateT s m) where
  ask :: StateT s m r
  ask = StateT \s -> ask <#> \r -> Tuple r s -- COMPILER ERROR!!
```

And now we have a brand new problem:



We need to add another Constraint, `MonadAsk`. But why?

That's because we're calling `ask` on the underlying `Monad`, which means it must have a `MonadAsk` Instance.

The error message helps us to add this Constraint:

```
Control.Monad.Reader.Class.MonadAsk r2 m3
```

This reminds us that `MonadAsk` takes 2 Parameters, an `r2` or what we called `r` and an `m3` or what we called `m`.

We'll add the Constraint:

```
instance monadAskStateT :: (Functor m, MonadAsk r m) =>
                           MonadAsk r (StateT s m) where
  ask :: StateT s m r
  ask = StateT \s -> ask <#> \r -> Tuple r s
```

Notice how we added Parentheses around the 2 Type Constraints and separated them with a comma.

But wait! Isn't `Monad` the Superclass for `MonadAsk`:

```
class Monad m <= MonadAsk r m | m -> r where
  ask :: m r
```

That means that we don't need the `Functor` Constraint anymore since all `Monads` are `Functors`. So, let's remove it:

```
instance monadAskStateT :: MonadAsk r m => MonadAsk r (StateT s m) where
  ask :: StateT s m r
  ask = StateT \s -> ask <#> \r -> Tuple r s
```

Please update your code to reflect these changes before moving on.

Now, we're ready to write MonadTell. Here's its definition for reference:

```
class Monad m <= MonadTell w m | m -> w where
  tell :: w -> m Unit
```

Go ahead and write the instance using what we learned with MonadAsk to help you. Then continue reading.

If you are like me, you're probably thinking that you can just copy the MonadAsk code and change it over to MonadTell:

```
import Control.Monad.Writer.Class (class MonadTell, tell) ❶

instance monadTellStateT
      :: MonadTell r m => MonadTell r (StateT s m) where
  tell w = StateT \s -> tell w <#> \_ -> Tuple unit s ❷
```

❶ Added this import.

❷ Since `tell w` returns `m Unit`, we discard the Lambda Parameter and simply hardcode `unit`. For the record, the Parameter is also `unit`.

At this point, you're probably thinking of one or two things, e.g., wow, why didn't I think of that, or if you did think of it, then you're probably thinking, that was pretty easy to code.

And it was easy to code. TOO EASY.

This should be a huge red flag. We didn't think. We just did simple substitution. That means that we've just created boilerplate code, i.e. redundant code, code we should factor out somehow.

So, take those 2 instance implementations and factor out the common code. Here they are again for reference:

```
instance monadAskStateT
    :: MonadAsk r m => MonadAsk r (StateT s m) where
  ask = StateT \s -> ask <#> \r -> Tuple r s

instance monadTellStateT
    :: MonadTell r m => MonadTell r (StateT s m) where
  tell w = StateT \s -> tell w <#> \_ -> Tuple unit s ❶
```

❶ Remember that the discarded Parameter is `unit`.

Before you can try to factor out the common code, you first have to identify the common code. So, do that

first before reading on.

Here's what I see as common code:

```
StateT \s -> ??? <#> \x -> Tuple x s
```

Where I've placed `???` is where the delegated call to the underlying `Monad` is made.

Now, take this information and use it to factor out the common code into a Function that we'll call `liftStateT`. Do that before reading further.

We'll start with part of the Type Signature:

```
liftStateT :: m a -> ???
```

The first Parameter is the ***result*** of calling the underlying `Monad's` Method, i.e. a Value of Type `a` in the Monadic Context, `m`.

Looking back at the common code we identified, we can see that we'll want this to return a `StateT`, so we'll add that:

```
liftStateT :: ∀ s m a. m a -> StateT s m a ❶
```

❶ Had to add the `∀` part.

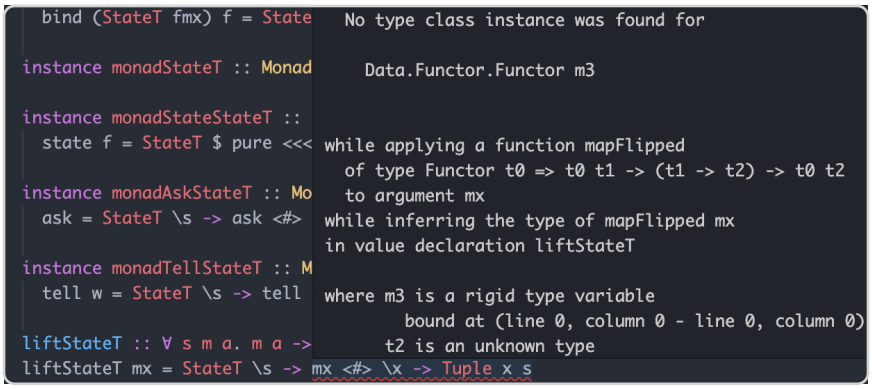
And now for the left-hand side:

```
liftStateT :: ∀ s m a. Functor m => m a -> StateT s m a
liftStateT mx = ???
```

And finally, we will return a `StateT` using the common code replacing the `???` in the common code with `mx`:

```
liftStateT :: ∀ s m a. m a -> StateT s m a  -- COMPLILER ERROR!
liftStateT mx = StateT \s -> mx <#> \x -> Tuple x s
```

And to no one's surprise, the compiler complains:



We have to make `m` a `Functor` instance for the same reason we did before, which is because we're using `<#>` on it:

```
liftStateT :: ∀ s m a. Functor m => m a -> StateT s m a
liftStateT mx = StateT \s -> mx <#> \x -> Tuple x s
```

Now you should be able to rewrite your 2 instances using `liftStateT`. So make sure that your version of `liftStateT` is up to date with the one above and rewrite the `MonadAsk` and `MonadTell` instances to use `liftStateT`, then read on.

Here's the rewrite:

```
instance monadAskStateT
    :: MonadAsk r m => MonadAsk r (StateT s m) where
  ask = liftStateT ask

instance monadTellStateT
    :: MonadTell r m => MonadTell r (StateT s m) where
  tell = liftStateT <<< tell ❶
```

❶ Point-free notation by canceling the `w` on both sides of the equal sign.

This is so much cleaner and we've factored out the boilerplate code.

This was quite helpful. So helpful that we could probably use this kind of help if we were writing instances for `WriterT` or `ReaderT` or `ExceptT` or any Monad Transformer.

When we have the same Function for different Types that means that we should make a Typeclass. Let's call it `MonadTrans`:

```
class MonadTrans t where
  lift :: ∀ m a. Monad m => m a -> t m a
```

I'm sure at this point you're calling foul, if not, earlier. We've already seen `lift` in the previous Chapter. Back then, I simply introduced it to you whole cloth.

This time, we've taken a path that shows us a possible scenario for how `lift` came about.

So, now comment out your `liftStateT`, change the 2 instances to use `lift` and add the following import:

```
import Control.Monad.Trans.Class (class MonadTrans, lift)
```

You're going to get a compiler error telling you that there is no instance of `MonadTrans` for `StateT`. So go ahead and write the `StateT` Instance for `MonadTrans` stealing liberally from your `liftStateT` implementation.

Do that before reading the solution.

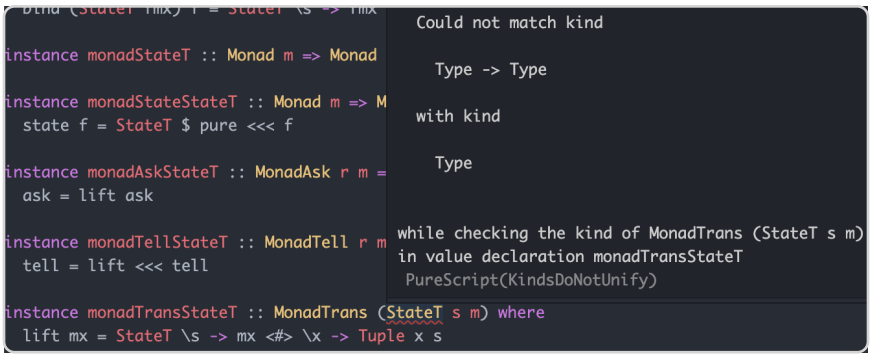
Like we've done a bunch of times before, we start with the instance definition:

```
instance monadTransStateT :: MonadTrans (StateT s m) where
```

Next, we'll just steal the implementation from `liftState`:

```
-- COMPILER ERROR!!
instance monadTransStateT :: MonadTrans (StateT s m) where
  lift mx = StateT \s -> mx <#> \x -> Tuple x s
```

And now the compiler is complaining for some odd reason:



Based on where the red underline is, I suspect I've given `StateT` too many or too few Type Parameters for `MonadTras's` liking. Let's look back at `MonadTrans's` definition again:

```
class MonadTrans t where
  lift :: ∀ m a. Monad m => m a -> t m a
```

So, I've provided `t` with `StateT s m`. Can you see where I went wrong?

If `t` is `StateT s m`, then the return value for `lift` is `m a -> (StateT s m) m a`. And there's problem. There are 2 `m's`. I need to remove the one I specified for `t`:

```
instance monadTransStateT :: MonadTrans (StateT s) where ❶
  lift mx = StateT \s -> mx <#> \x -> Tuple x s
```

❶ `m` was removed.

And now you're ready to code the `MonadThrow` Instance for `StateT`. Here's it's definition for reference:

```
class Monad m <= MonadThrow e m | m -> e where
  throwError :: ∀ a. e -> m a
```

Give it try then continue reading.

Here's the instance definition:

```
import Control.Monad.Error.Class (class MonadThrow) ❶

instance monadThrowStateT :: MonadThrow e (StateT s m) where ❷
```

❶ We need to import the MonadThrow class.

❷ Don't forget the error Type, `e`.

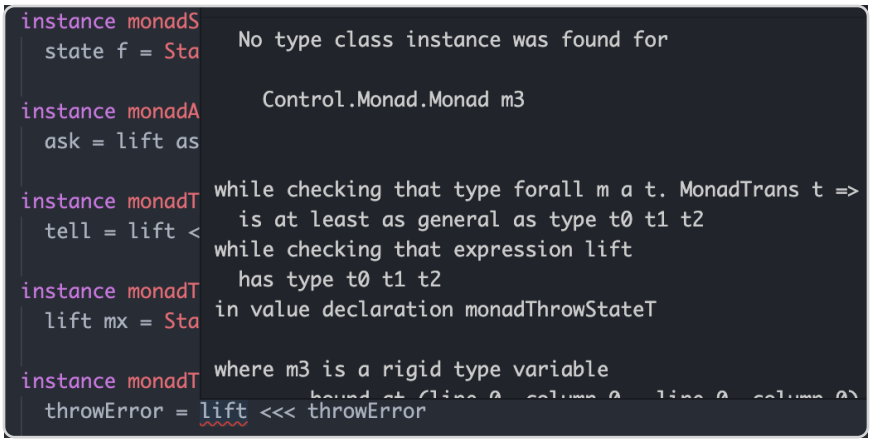
Now, I'm going to copy what we did in `MonadTell`:

```
import Control.Monad.Error.Class (class MonadThrow, throwError) ❶

instance monadThrowStateT
    :: MonadThrow e m => MonadThrow e (StateT s m) where
  throwError = lift <<< throwError -- COMPILER ERROR!!
```

❶ Have to add `throwError` to the import list.

The compiler is complaining:

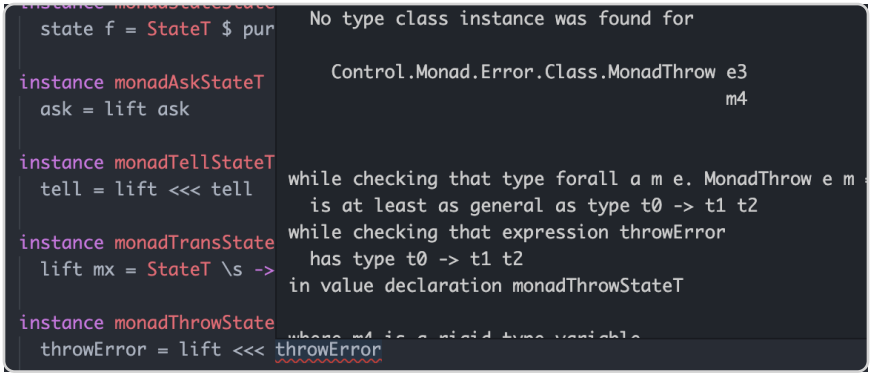


It's telling us that `m` must be a `Monad`. But why?

Turns out that `lift` has that Constraint and since we've call `lift`, we'll need to constrain our `m` similarly:

```
instance monadThrowStateT :: Monad m => MonadThrow e (StateT s m) where
  throwError = lift <<< throwError -- COMPILER ERROR!!
```

And... a compiler error:



We need to add the Constraint for `MonadThrow` just like we did for `MonadAsk` and `MonadTell`. I completely forgot. If you remembered then kudos to you.

But the one thing I do remember is that we can remove the `Monad` Constraint once we add `MonadThrow` and that's because its Superclass is `Monad`:

```
instance monadThrowStateT
    :: MonadThrow e m => MonadThrow e (StateT s m) where
  throwError = lift <<< throwError
```

Now it compiles.

All that's left to do is code the `MonadError` Instance for `StateT`. Here's its definition for reference:

```
class MonadThrow e m <= MonadError e m | m -> e where
  catchError :: ∀ a. m a -> (e -> m a) -> m a ❶
```

❶ Notice that the Type Signature is very different from ask or tell. This should be a clue that we're going to need to take a different approach than we did with those instances.

Before we move on to the code MonadError, we should understand catchError. Here's our AppM code from the previous Chapter that has been modified to catch the error:

```
validate :: Int -> AppM
validate n = when (n == 0) $ void $ throwError "WE CANNOT HAVE A 0 STATE!"

app :: AppM
app = do
  log "Starting App..."
  n <- get
  -- when (n == 0) $ void $ throwError "WE CANNOT HAVE A 0 STATE!" ❶
  catchError (validate n) (\err -> do ❷
    log $ "We encountered an error: " <> err
    put 1
  )
  put $ n + 1
  log "Incremented State"
  pure unit
```

❶ Our old code threw the error and Short-circuited the computation. It's been commented out.

❷ In this new version, we catch the error thrown by validate and handle it by writing to the log and making the State non-zero. This does NOT Short-circuit the computation, but, instead, provides an alternative computation that never fails.

Notice how the Lambda is running in the AppM Monad. This matches the second Parameter to catchError, i.e. e -> m a, where m in our case is ExceptT String (WriterT String (StateT Int Effect)) Unit, or AppM as we like to refer to it:

```
type AppStack e w s a = ExceptT e (WriterT w (StateT s Effect)) a

type AppM = AppStack String String Int Unit
```

Since catchError's Type Signature is very different from ask and tell we cannot simply lift the underlying Monad's implementations.

Since this instance is complex and fraught with caveats, which we will soon see, we're going to code this

together.

Here's MonadError for reference:

```
class MonadThrow e m <= MonadError e m | m -> e where
  catchError :: ∀ a. m a -> (e -> m a) -> m a
```

Let's start with the instance definition:

```
import Control.Monad.Error.Class (class MonadThrow, class MonadError,
throwError) ❶

instance monadErrorStateT :: MonadError e (StateT s m) where
```

❶ Added MonadError to the import list.

At this point, I'm thinking about Constraints. Since StateT doesn't handle errors, we have to delegate this call to our underlying Monad.

For this to work, our underlying Monad must support MonadError. We've seen this before with MonadAsk and MonadTell. We need to add this as a Constraint:

```
instance monadErrorStateT
    :: MonadError e m => MonadError e (StateT s m) where ❶
```

❶ The underlying Monad is constrained to be a MonadError.

Since this Method is so different from the others, we're going add the Type Signature for catchError specialized for StateT to help us keep track of our Types:

```
instance monadErrorStateT
    :: MonadError e m => MonadError e (StateT s m) where
  catchError :: ∀ a. StateT s m a -> (e -> StateT s m a) -> StateT s m a
```

Next we'll add the left-hand side of the implementation:

```
instance monadErrorStateT
    :: MonadError e m => MonadError e (StateT s m) where
  catchError :: ∀ a. StateT s m a -> (e -> StateT s m a) -> StateT s m a
  catchError (StateT fmx) f = ??? ❶ ❷
```

❶ The name fmx is to remind us that we have a Function that returns a Value in the underlying Monad.

❷ `f` is the Function that will be called to handle the error. It gets the error, `e` and returns a `StateT` which "replaces" the `StateT` that failed.

Next, we use our usual pattern:

```
instance monadErrorStateT
    :: MonadError e m => MonadError e (StateT s m) where
  catchError :: ∀ a. StateT s m a -> (e -> StateT s m a) -> StateT s m a
  catchError (StateT fmx) f = StateT \s -> ???
```

And now comes the hard part. This is where we do our thinking.

We have 3 things to work with:

- the underlying `Monad`, i.e. its `catchError` implementation
- `fmx`
- `f`

We know that we're going to delegate to the underlying `Monad`, at some point, since `StateT` doesn't handle errors. To do this, we'll call the underlying `Monad's catchError`.

Let's add that:

```
import Control.Monad.Error.Class (class MonadThrow, class MonadError,
throwError, catchError) ❶

instance monadErrorStateT
    :: MonadError e m => MonadError e (StateT s m) where
  catchError :: ∀ a. StateT s m a -> (e -> StateT s m a) -> StateT s m a
  catchError (StateT fmx) f = StateT \s -> catchError ??? ???
```

❶ Added `catchError` to the import list.

`catchError's` first Parameter is a Value in the Context of the underlying `Monad`. So how can we get such a thing?

To answer this, look back at the 2 things we have left to work with. Can you see anything that can give us this?

If not, then it's time to write out the Types. To do this, we're going to consult the specialized Type Signature for `catchError`:

```
fmx :: s -> m (Tuple a s)
f :: e -> StateT s m a
```

Now can you see something that will give us a Monadic Value in the underlying Monad?

We can see from the Types above that `fmx` will give us that. But, we'll have to give it a State Value, which we can since we have a State Value, `s`, from our Lambda expression:

```
instance monadErrorStateT
    :: MonadError e m => MonadError e (StateT s m) where
  catchError :: ∀ a. StateT s m a -> (e -> StateT s m a) -> StateT s m a
  catchError (StateT fmx) f = StateT \s -> catchError (fmx s) ???
```

Looking at the second Parameter in `catchError's` Type Signature, we can see that we have a Function that accepts the error and returns a `StateT`. Let's add the skeleton of that:

```
instance monadErrorStateT
    :: MonadError e m => MonadError e (StateT s m) where
  catchError :: ∀ a. StateT s m a -> (e -> StateT s m a) -> StateT s m a
  catchError (StateT fmx) f = StateT \s ->
                catchError (fmx s) \e -> ??? ❶
```

❶ Added the Lambda, `\e -> ???`.

In this error handler, we'll need to call `f`, which is the error handler that `StateT's catchError` was originally passed. Let's add that:

```
instance monadErrorStateT
    :: MonadError e m => MonadError e (StateT s m) where
  catchError :: ∀ a. StateT s m a -> (e -> StateT s m a) -> StateT s m a
  catchError (StateT fmx) f = StateT \s ->
                catchError (fmx s) \e -> f e ???
```

Now, before we can move on, we need to know what Type `f e` is:

```
f :: e -> StateT s m a
f e :: StateT s m a
```

Do you remember what we did when we had a similar situation, i.e. we have a `StateT` on the right-hand side of the Lambda?

Look back at the `bind` implementation to see what we did before reading on.

Did you notice that we just called `runStateT`?

Since it worked there to give us a Monadic Value in the underlying `Monad`, it'll work here:

```
instance monadErrorStateT
    :: MonadError e m => MonadError e (StateT s m) where
  catchError :: ∀ a. StateT s m a -> (e -> StateT s m a) -> StateT s m a
  catchError (StateT fmx) f = StateT \s ->
                 catchError (fmx s) \e -> runStateT (f e) ???
```

Now here's the kicker. `runStateT` wants a State for its second Parameter, but where can we get a State from?

The only State we have is the SAME one we passed to `fmx`. So that's the only one we can use. And so we will:

```
  instance monadErrorStateT
      :: MonadError e m => MonadError e (StateT s m) where
    catchError :: ∀ a. StateT s m a -> (e -> StateT s m a) -> StateT s m a
    catchError (StateT fmx) f = StateT \s ->
                   catchError (fmx s) \e -> runStateT (f e) s
```

Go ahead and make sure to add this to your code before reading on.

Notice that we passed the SAME State to `fmx` and to `f`. This is very different than what we did everywhere else when dealing with State in multiple places in a Function.

Typically, we captured the MODIFIED State from the first Function that uses State and passed that modified version to the next Function that needs a State.

We cannot do that here and so the best we can do is use the SAME State. That means that any changes that `fmx` makes to the State will be lost if `fmx` calls `throwError`.

We'll look at this issue in more detail later.

21.4. Testing our StateT Implementation

We'll want to test out our newly minted Monad Transformer to see if it behaves correctly. To do this, we're going to steal the ideas and some of the code we saw in the previous Chapter.

Let's start with the Application Stack definition:

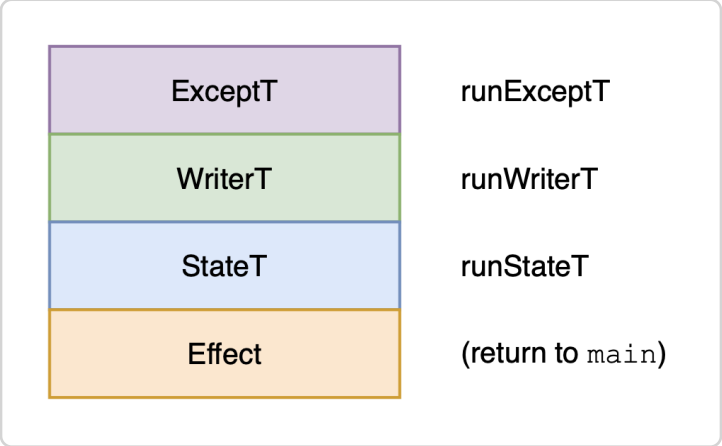
```
import Control.Monad.Except.Trans (ExceptT) ❶
import Control.Monad.Writer.Trans (WriterT)

type AppStack e w s a = ExceptT e (WriterT w (StateT s Effect)) a
```

❶ We'll need these imports to make this Type Alias.

Please add this to your code.

Here's a diagram of our Stack:



Now, define a Type Alias for our Application Monad called `AppM` that leverages our Stack definition. We'll want our error to be a `String`, our State to be an `Int`, our log to be a `String` and the final result of our application to be `Unit`.

Code this before moving on.

The Application Monad simply provides Monomorphic Types for each of the Polymorphic Types in the `AppStack` definition:

```
type AppM = AppStack String String Int Unit
```

Next, we'll need a way to run our Monad Stack, so we'll need a Function called `runApp` that will take an Initial State and an Application Monad and produce a result.

Try and write the Type Signature for `runApp` before reading further.

This Type Signature requires the analysis we spoke of in the previous Chapter, i.e. walking through the Types in our Stack to figure out the return Type. Or, we could just cheat with Type Holes. Cheating is so much easier, so let's use a Type Hole:

```
runApp :: Int -> AppM -> Effect _
```

Remember that we have the `Effect` Monad at the bottom of our Stack. There is no `runEffect` function since `Effect` is what we return to PureScript via our `main` Function.

This fact is why we're returning the `Effect` Monad. We'll talk more about `Effect` and `main` in the Beyond Section of the book.

The `_` is our Type Hole. Notice that it NOT named. You can have unnamed Type Holes in Type Signatures. It's there to just cause the compiler to provide us with the inferred Type once our code compiles.

The next step is to code the rest of the run Function, remembering to use the individual run Functions for each Monad starting at the top and working our way down until we reach `Effect`.

For reference, here are the Type Signatures for the run Functions:

```
runExceptT :: ∀ e m a. ExceptT e m a -> m (Either e a)
runWriterT :: ∀ w m a. WriterT w m a -> m (Tuple a w)
runStateT :: ∀ s m a. StateT s m a -> (s -> m (Tuple a s))
```

Please write the rest of `runApp` before reading further.

We'll start by executing the top Monad on the Stack, i.e. `ExceptT`:

```
runApp :: Int -> AppM -> Effect _
runApp st = ??? <<< runExceptT
```

And then the next Monad down the Stack, i.e. `WriterT`:

```
runApp :: Int -> AppM -> Effect _
runApp st = ??? <<< runWriterT <<< runExceptT
```

And finally, the last RUNNABLE Monad in the Stack, i.e. `StateT`:

```
runApp :: Int -> AppM -> Effect _
runApp st = flip runStateT st <<< runWriterT <<< runExceptT ❶
```

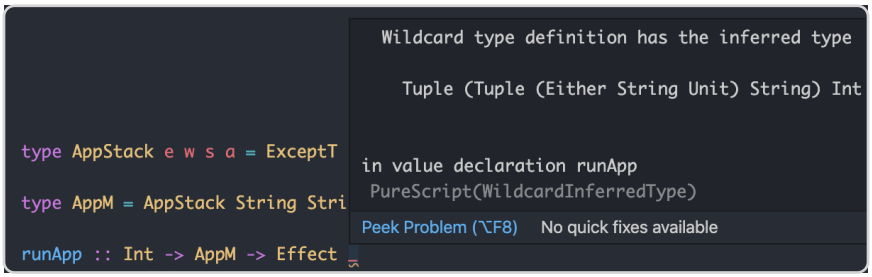
❶ By flipping `runStateT`, we can provide the second Parameter first, which is the Initial State. This allows us to write `runApp` using Function Composition making it Point-free with the `AppM` Parameter going unnamed.

We get some compiler errors because we didn't import the `runWriterT` and `runExceptT` definitions. We do NOT need to include `StateT` from the libraries since we wrote our own.

Let's add the imports:

```
import Control.Monad.Except.Trans (ExceptT, runExceptT)
import Control.Monad.Writer.Trans (WriterT, runWriterT)
```

And now that our code compiles, the compiler has a warning regarding our Type Hole:



Go ahead and update your Type Signature from this warning message. Hint: You can hover over the yellow underlined code and then select the text from the pop-up message and then copy and paste it into your code if your using VSCode.

Here's that change:

```
import Data.Either (Either) ❶

runApp
  :: Int
  -> AppM
  -> Effect (Tuple (Tuple (Either String Unit) String) Int)
runApp st = flip runStateT st <<< runWriterT <<< runExceptT
```

❶ This is the first time we're using `Either` in this module, so we need to import it.

Let's factor out that return Type, minus the `Effect`, part like we did in the previous Chapter. Make a Type Alias for it and call it `StackResult`. Then continue on.

Here's that refactor:

```
type StackResult = Tuple (Tuple (Either String Unit) String) Int

runApp :: Int -> AppM -> Effect StackResult
runApp st = flip runStateT st <<< runWriterT <<< runExceptT
```

Next, build a Type Alias called `AppEffects` that is a Record to store the Side-effects of running our Stack. This will hold the Side-effects of our Stack in an order-independent way. Don't forget to include the Pure Computational result.

Please give it try first, then continue reading.

And here it is:

```
type AppEffects =
  { log :: String ❶
  , state :: Int ❷
  , result :: Maybe Unit ❸ ❹
  }
```

❶ The Side-effect of the `WriterT` Monad will be stored here. It's the final log.

❷ The Side-effect of the `StateT` Monad will be stored here. It's the final State.

❸ The Pure Computation Value will be stored here. In this case, the only Value it could be is `Unit`. It's here to show what `AppEffects` should contain in general, even though, in our case, it's trivial.

❹ This is a `Maybe` since it's possible for errors, which means we won't have a final result.

Now that we have `AppEffects`, the code that calls our Application Code, i.e. our Monad Stack, can be insulated from the Stack Order.

Now, let's create a Type Alias called `AppResult` that contains our Side-effect Values from running our Monad Stack, i.e. `AppEffects` and, optionally, the error if one occurred.

Do this first, then continue on.

There are 2 approaches we could take, create a Record or use a `Tuple`. Both are fine, but since we only have 2 things, I'm going to use a `Tuple`:

```
type AppResult = Tuple (Maybe String) AppEffects ❶
```

❶ The error is in the first position of the `Tuple`. I used a `Maybe` here since its possible that no errors occurred.

Go ahead and change your code to match this so that we're coding to the same Types moving forward.

By the way, we could've made our `result` in `AppResults` an `Either` and put either the Pure Computational result or the error there. Then we wouldn't need the `Tuple`. What we have here is Isomorphic to that so we're just going to leave what we have for now. We'll change it later if it proves problematic.

Now, we want to write a Function, which we will call `results`, to take our `StackResult`, which is tightly coupled to our Stack Order, and convert it to `AppResult`, which is loosely coupled to our Stack Order.

When two or more things are ***tightly coupled*** and one of them changes, then the others must change.

When two or more things are ***loosely coupled*** and one of them changes, the others most likely will not have to change. Many times, it's just a change that needs to be made in an interface or maybe a mapping Function.

In our case, the `results` Function you're about to write will take our `StackResult`, which is tightly coupled to the Stack Order and map it to `AppResult`, which is loosely coupled.

For the record, it's tightly coupled to the Monads in the Stack, but not the Order. And the `result` Function will be tightly coupled to our Stack Order.

Now, go ahead and write `results` that takes a `StackResult` and produces an `AppResult` before advancing further.

We'll start with the Type Signature:

```
results :: StackResult -> AppResult
```

The Type Signature reflects that `results` maps `StackResult` to `AppResult`. The term ***map*** here might be confusing. I'm not using it in the context of `Functors`. It's mapping the way Functions do from their Domain to their Codomain.

Next, we'll write the error case:

```
import Data.Either (Either(..)) ❶
import Data.Maybe (Maybe(..)) ❷

results :: StackResult -> AppResult
results (Tuple (Tuple (Left err) l) s)
  = Tuple (Just err) { log: l, state: s, result: Nothing } ❸
???
```

❶ Added `(..)` to import `Left`.

❷ Added `(..)` to import `Nothing`.

❸ Notice that there's no result. That's because an error occurred.

And now for the success case:

```
results :: StackResult -> AppResult
results (Tuple (Tuple (Left err) l) s)
  = Tuple (Just err) { log: l, state: s, result: Nothing }
results (Tuple (Tuple (Right result) l) s) ❶
  = Tuple Nothing { log: l, state: s, result: Just result }
```

❶ In our case, `result` here is just `unit`.

Notice how the only difference is in the first Position of the `Tuple` and the computational result. We still capture the log and State.

We now have enough plumbing to allow us to write our Application code. This code will run in our Application Monad, `AppM`. We'll call this Function, `app`.

Start this process by just writing the Type Signature before reading any further.

Here's the Type Signature:

```
app :: AppM
```

Take a moment to realize that when we call this Function, it won't execute any of the instructions we've written. It'll wrap `Monads` in other `Monads` and compose the Monadic Functions together to produce a single, monolithic `Monad` of Type `AppM`.

This monolithic `Monad` will be executed later in our code when we pass it to `runApp` along with the Initial State. We can also decide NOT to execute this `Monad`. It's totally up to us.

The important point is to NOT think of calling `app` as executing the code. It's deferred until we call `runApp`, which executes our Stack from the top to bottom, one layer at a time.

Now, here are the requirements for the body of `app` in the specified order:

- write to the log "Starting App..."
- get the State
- check the State to make sure it's non-zero, otherwise error with "WE CANNOT HAVE 0 STATE!"
- add 1 to the State
- write to the log "Incremented State"
- return the Pure Computational Value

Write the rest of the body of `app` before reading the following hints.

If you're struggling with this, keep in mind:

- start coding in a `do` block
- `get` and `put` are used for reading and writing State (look them up in `Pursuit`)
- `tell` writes to the log
- the Pure Computation Value is the `a` in `AppStack`
- go look up `when` in `Pursuit` for how to check for non-zero

- the function `throwError` Short-circuits our `do` block

Hopefully, these hints will help you finish. Please continue working on the code if you haven't finished.

The first thing to add to `app` is a `do` block since we're operating in the `AppM` Monad. Actually, we would do this for any Monad:

```
app :: AppM
app = do
  ???
```

The first step in the requirements is to log a `String`. We'll use `tell`:

```
app :: AppM
app = do
  tell "Starting App..."
  ???
```

If you got stuck and this has helped break through that brick wall, stop reading and see if you can finish coding this Function.

Next, we're going to get the State. We'll use `get` and capture the State into `n`:

```
import Control.Monad.State.Class (class MonadState, get) ❶

app :: AppM
app = do
  tell "Starting App..."
  n <- get
  ???
```

❶ Added `get` to the import list.

And then we'll check to make sure that the State is non-zero. We'll use `when`:

```
app :: AppM
app = do
  tell "Starting App..."
  n <- get
  when (n == 0) $ void $ throwError "WE CANNOT HAVE A 0 STATE!" ❶
  ???
```

❶ `throwError` is an `m a`, but `when` wants an `m Unit`. So, we use `void` to convert `m a` to `m Unit`.

The next requirement is to add 1 to the State. We'll use `put`:

```
import Control.Monad.State.Class (class MonadState, get, put) ❶

app :: AppM
app = do
  tell "Starting App..."
  n <- get
  when (n == 0) $ void $ throwError "WE CANNOT HAVE A 0 STATE!"
  put $ n + 1
  ???
```

❶ Added `put` to the import list.

And then we'll write to the log using `tell` again:

```
app :: AppM
app = do
  tell "Starting App..."
  n <- get
  when (n == 0) $ void $ throwError "WE CANNOT HAVE A 0 STATE!"
  put $ n + 1
  tell "Incremented State"
  ???
```

And finally, we'll return the Pure Computational Value. But before we do that, we'll look back at our definitions:

```
type AppStack e w s a = ExceptT e (WriterT w (StateT s Effect)) a ❶

type AppM = AppStack String String Int Unit ❷
```

❶ `a` is the Pure Computation result.

❷ Here we set the Polymorphic Type Parameter, `a`, to the Monomorphic Type, `Unit`.

So, we have to return a Value of Type, `Unit`. Luckily, that will be easy, since there's only 1 inhabitant of the Type, `Unit`:

```
app :: AppM
app = do
  tell "Starting App..."
  n <- get
  when (n == 0) $ void $ throwError "WE CANNOT HAVE A 0 STATE!"
  put $ n + 1
  tell "Incremented State"
  pure unit ❶
```

❶ Don't forget the `pure`. This is because all `do` blocks must return a Monadic Value.

And now, change your test code to be the following:

```
test :: Effect Unit
test = do
  result1 <- runApp 0 app
  log $ show result1 ❶
  result2 <- runApp 99 app
  log $ show result2 ❷
```

❶ Prints `(Tuple (Tuple (Left "WE CANNOT HAVE A 0 STATE!") "Starting App…") 0)`.

❷ Prints `(Tuple (Tuple (Right unit) "Starting App…Incremented State") 100)`.

Go ahead and run you code to see if it works.

Now let's improve our code so we get the decoupled output, i.e. `AppResult`, instead of the coupled output, `StackResult`.

We'll do this by calling `results`.

There will be 2 places where this could be done. One is better than the other. Take a look and see where you'd put the call to `results` before reading on.

We could put the call to `results` in the test code, but then we'd have to put it in once for every time we call `runApp`. So, why not just put it in `runApp`?

And that's what we're going to do. If you haven't done that already, go ahead and modify `runApp` to use `results` now. Then read on.

Here's the modified `runApp` Function:

```
runApp :: Int -> AppM -> Effect AppResult ❶
runApp st =
  (results <$> _) <<< flip runStateT st <<< runWriterT <<< runExceptT ❷
```

❶ Changed `StackResult` to `AppResult`.

❷ Created an anonymous Function to keep the code Point-free.

If you did the non-Point-free version, that's fine too. It looks like this:

```
runApp :: Int -> AppM -> Effect AppResult
runApp st ap =
  results <$> (flip runStateT st $ runWriterT $ runExceptT ap) ❶
```

❶ Notice that `<<<` was changed to `$`, i.e. the opposite change that's made during an ETA-reduction.

Personally, I prefer the Point-free version.

Let's look back at the test code and what we can expect it to print out with this new change:

```
test :: Effect Unit
test = do
  result1 <- runApp 0 app
  log $ show result1 ❶
  result2 <- runApp 99 app
  log $ show result2 ❷
```

❶ Prints `(Tuple (Just "WE CANNOT HAVE A 0 STATE!") { log: "Starting App…", result: Nothing, state: 0 })`.

❷ Prints `(Tuple Nothing { log: "Starting App…Incremented State", result: (Just unit), state: 100 })`.

Run your code to make sure you get the expected output.

This is much easier to look at, but notice the `log` output for the success case, i.e. the second call to `runApp`. Can you see anything wrong or deficient with it?

Since we chose to make our log Type a `String`, we have a run-on log, i.e. our `Strings` are concatenated without some sort of delimiter. Let's fix that by adding line endings to our log entries.

There are two ways to accomplish this. One is better than the other.

The first way is to just add `"\n"` to the end of every `tell` line of code. Or we could write a Function called `log` to write to the log and have it add `"\n"` to every `String` it gets.

The latter solution is far better than requiring `"\n"` to be placed at the end of every line. It's better for 2 reasons. First, it relieves an unnecessary burden from the developer.

Second, we can do this once and forget about it. But if we had required it to be done at each `tell` call, we'd have to remember every time and we know that somewhere, someone's going to forget to do this. Most likely us.

Go ahead and write just the Type Signature for a Function called `log` that will solve this problem before reading any more.

Here's the Type Signature:

```
log :: String -> ??? ❶
```

❶ Not obvious what the return Type is.

If you don't know what it returns, then just put a Type Hole there for now. Then go ahead and write the rest of the Function.

If you get a compiler error in the test code, don't worry, we'll fix that soon enough.

Do this and then read on.

Here's the Signature with a Type Hole:

```
log :: String -> _
```

Now we'll call `tell` on the supplied `String`:

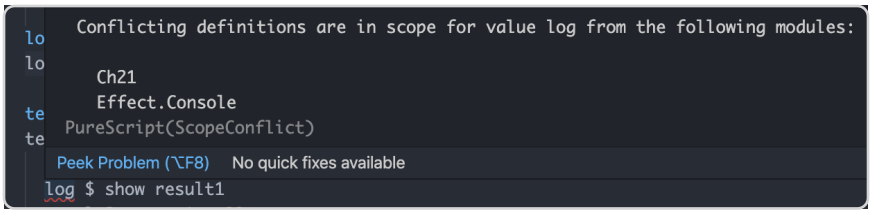
```
log :: String -> _
log s = tell s
```

Now we'll add the line ending:

```
log :: String -> _
log s = tell $ s <> "\n" ❶
```

❶ Don't forget `$` or Parentheses.

And now we're getting that compiler error mentioned earlier:



We now have a local Function called `log`, which conflicts with the `log` we're importing from the `Event.Console` module.

To fix this ambiguity, we'll fully qualify the import of that Module:

```
import Effect.Console as Console ❶
```

❶ Notice that the import list that included `log` is missing and the `as` clause has been added to create an alias for the module that we're calling `Console`.

And now, we'll have to change the calls to the Console `log` in our test code:

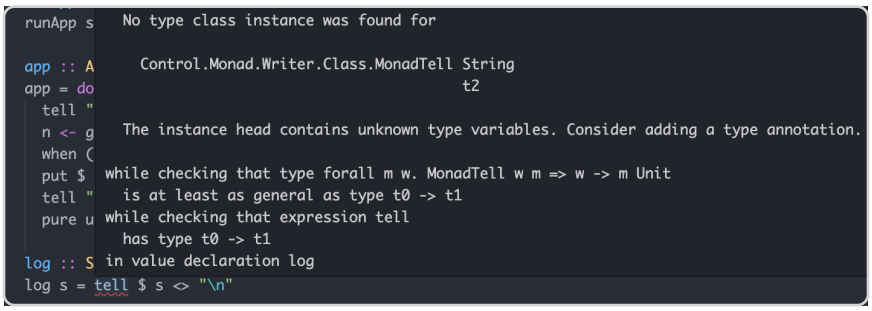
```
test :: Effect Unit
test = do
  result1 <- runApp 0 app
  Console.log $ show result1 ❶
  result2 <- runApp 99 app
  Console.log $ show result2 ❶
```

❶ Use the fully qualified name for `log`.

And now we're getting a compiler error on our local `log` Function:

```
log :: String -> _
log s = tell $ s <> "\n" -- COMPILER ERROR!
```

Here's the compiler error:



This says that we're missing a Constraint of `MonadTell` from our `log` code. But we're actually missing more than that. We don't even have a `Monad` in our Type Signature. That's why the error message refers to `t2`.

That Type is an unknown Type to the compiler.

So, we need to add a `Monad` to our Type Signature before we can constrain it. But how?

Well, we know that `log` is operating in SOME `Monad` that supports `MonadTell`. This means that `log` will return a Value in the Context of that `Monad`. So, let's add that fact to our Type Hole:

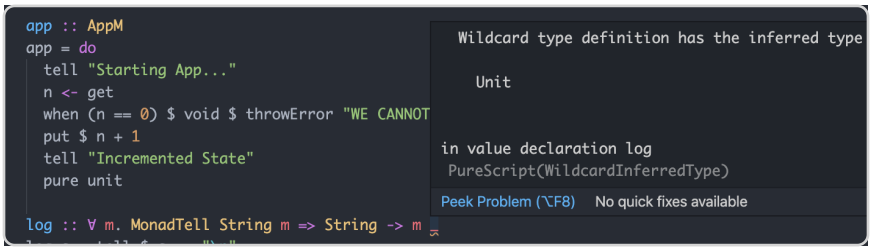
```
log :: ∀ m. String -> m _ ❶
log s = tell $ s <> "\n" -- COMPILER ERROR!
```

❶ We know that we're returning some Monadic Value. The Type of the Value is unknown to us at the moment.

And we still have the same error, but at least now we have a `Monad` to constrain. So, we'll constrain it now:

```
log :: ∀ m. MonadTell String m => String -> m _
log s = tell $ s <> "\n"
```

And now we're getting the following Type Hole warning:



The compiler has inferred that our contained Value is of Type `Unit`. So, we can replace our Type with `Unit`:

```
log :: ∀ m. MonadTell String m => String -> m Unit
log s = tell $ s <> "\n"
```

It turns out that we could've just looked up the Type on `tell` to figure out what our return Type would be since `log` just calls `tell`:

```
class Monad m <= MonadTell w m | m -> w where
  tell :: w -> m Unit
```

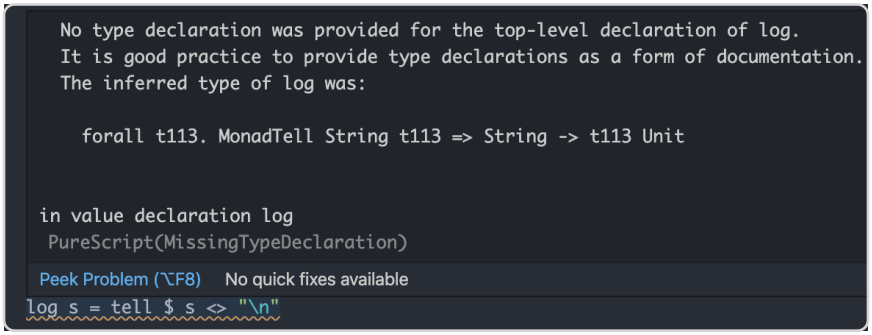
I took you down this path as practice for when Type Holes are a bit more helpful. Just a note about this approach, it doesn't always work out smoothly, in cases where we need a Constraint.

So, it's always best to put forth the effort to figure out the Types and not rely on the compiler for help.

Another approach you can use for particularly complex Type Signatures is to skip coding the Type Signature altogether. Once your Function compiles, you'll get a warning that tells you what the Type Signature should be.

Try this approach out now by commenting out your Type Signature for `log` and see what warning you get from the compiler. Do that before reading on.

The warning I get is:



The suggested Type Signature here is:

```
forall t237. MonadTell String t237 => String -> t237 Unit
```

Although, this Type Signature is relatively simple, it's still difficult to figure out what we should call `t237`. We'd have to look to `MonadTell` to see what its second Parameter is called:

```
class Monad m <= MonadTell w m | m -> w where
  tell :: w -> m Unit
```

And from this we can see that `t237` is `m`. So, we'll make that substitution:

```
∀ m. MonadTell String m => String -> m Unit ❶
```

❶ We're using `∀` instead of `forall`.

Now we have an Application running in a Monad Stack built from Monad Transformers. We have access to all the APIs and our implementation of `StateT` seems to work well as a replacement for the PureScript library version of `StateT`.

But everything, unfortunately, isn't that simple. There's always nuances to consider.

21.5. Problems with using StateT with ExceptT

Let's look back at our implementation for `catchError`:

```
instance monadErrorStateT
    :: MonadError e m => MonadError e (StateT s m) where
  catchError :: ∀ a. StateT s m a -> (e -> StateT s m a) -> StateT s m a
  catchError (StateT fmx) f =
    StateT \s -> catchError (fmx s) \e -> runStateT (f e) s
```

Take a look at the above code and pay particular attention to the State, `s`. Notice that it's used in 2 places.

It's first passed to `fmx`. That Function is the function that `catchError` is going to TRY to evaluate and if and only if there's an error, it'll call the error handler, i.e. the Lambda.

Now, look carefully at the Lambda, especially where `s` is passed to `runStateT`. Notice how it's the SAME `s` we passed to the code that had an error, i.e. `fmx`.

But what if `fmx` changed the State? Don't we want these changes?

We either always want the changes or we never want them. The problem is that we're going to get the changes if and only if there is no error. If there is an error, any State changes will be lost.

This may or may not be what we want or expect. And, to make matters worse, this behavior depends on the Order of the Stack. Other Stack orders will always keep the State changes.

This means that we cannot easily predict the behavior of our programs, which is definitely a bad situation.

At this point, you may be wondering if our implementation is broken and all we have to do is fix it. Well, I thought that too when I first saw this problem.

Here was my first attempt to fix it:

```
instance monadErrorStateT
    :: MonadError e m => MonadError e (StateT s m) where
  -- DOES NOT FIX THE PROBLEM!!!!!
  catchError :: ∀ a. StateT s m a -> (e -> StateT s m a) -> StateT s m a
  catchError (StateT fmx) f = StateT \s -> fmx s >>= \(Tuple x s') -> ❶
          catchError (pure $ Tuple x s') (\e -> runStateT (f e) s')
```

❶ Using `bind`, I'm capturing the changed State in `s'`.

My thought here was to capture the State changes in `s'` from `fmx` and then pass the UPDATED State to `runStateT`.

I thought this would fix things. It does NOT.

The reason it fails is because of the way that the Stack ordering works and the way `bind` for `ExceptT` Short-circuits. Let's me explain.

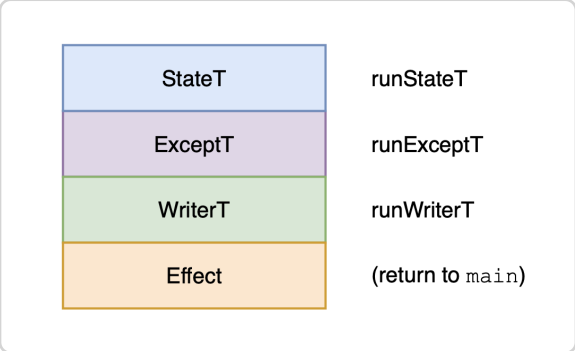
First thing to understand is WHEN will `StateT's` version of `catchError` be called. To answer that, look back at our valid implementation:

```
instance monadErrorStateT
    :: MonadError e m => MonadError e (StateT s m) where
  catchError :: ∀ a. StateT s m a -> (e -> StateT s m a) -> StateT s m a
  catchError (StateT fmx) f =
    StateT \s -> catchError (fmx s) \e -> runStateT (f e) s
```

Notice how we're calling `catchError` inside our implementation. We know this isn't a recursive call. It's actually calling the `catchError` of the underlying `Monad`. That means that `ExceptT` is NOT above us in the Stack, because if it was, our `catchError` will never be called since `ExceptT` would've already handled it.

For ours to be called we'd have to be ABOVE `ExceptT` in the Stack.

So, let's look at a Stack where that's the case:



Now, when we call `catchError` in our Monadic Code, we'll call `StateT's` implementation of `catchError` first. It will delegate to the underlying `Monad's` implementation, which in the above Stack is `ExceptT`. This is where the real implementation is.

If `ExceptT` was on top of `StateT`, then `ExceptT's` implementation would be called first. Since it's the real implementation, i.e. it doesn't delegate the responsibility like `StateT` does, `StateT's` implementation will never be called.

Let's return to my fix that didn't work:

```
instance monadErrorStateT
    :: MonadError e m => MonadError e (StateT s m) where
  -- DOES NOT FIX THE PROBLEM!!!!!
  catchError :: ∀ a. StateT s m a -> (e -> StateT s m a) -> StateT s m a
  catchError (StateT fmx) f = StateT \s -> fmx s >>= \(Tuple x s') -> ❶
          catchError (pure $ Tuple x s') (\e -> runStateT (f e) s')
```

Look at `fmx s >>= ...`.

That's were things go wrong. Since `ExceptT` is below us in the Stack, it's going to call it's `bind`, which Short-circuits on errors. So, during the error case, the Lambda NEVER GETS CALLED. That's the whole point of Short-circuiting.

This means that `s'` will NOT capture the changes made in the failed computation.

It turns out that there's no way to fix this. We just never know enough about the underlying `Monads` to extract the modified State.

So if we cannot fix it, we'll have to learn to live with it, which means we need to understand how to properly use `catchError` and what pitfalls we can expect.

Before we can do that, we'll need to understand about the `spy` Function. `spy` will take any Value and display its Internal Representation on the Console no matter if it has a `Show` Instance or not. It's used for debugging purposes.

Here's its Type Signature:

```
spy :: ∀ a. DebugWarning => String -> a -> a
```

First thing to notice about `spy's` Type Signature is that `DebugWarning` is a marker Constraint that the compiler will look for and give you a warning that you're using `spy` in a Function.

The reason for this is because `spy` breaks the contract that all Functions in PureScript are Pure. It does this because it prints to the Console. This is ONLY to be used for debugging purposes. It should NEVER be used in production code.

Realize that when you use `spy` and the compiler memoizes the result of a Function call, your `spy` output may or may not occur once for every call to the Function where it's used.

Memoize is the term for remembering the result of a computation to avoid having to call a Function again. The compiler remembers the result by storing it in memory somewhere, or, in the case of Haskell, which generates binary code, it save it in a CPU register.

The compiler can make these sorts of optimizations in Pure Functional Programming since it knows that all Functions are Pure and calling it with the same input will produce the same output.

The second thing to notice about `spy's` Type Signature is that its first Parameter is a `String`. This is what `spy` will prefix the output with followed by a colon and a space. This is useful to help to you know what you've printed out.

The Value that `spy` will output is the second Parameter, `a`.

Third thing to notice is that `spy` returns an `a`. So, it's like `identity` but with a REAL WORLD Side-effect. Not Pure Side-effects like we've been doing with `Monads`.

Here's how you can use it:

```
import Debug (spy) ❶

add :: Int -> Int -> Int
add x y = (spy "X" x) + (spy "Y" y)
```

❶ You'll need to add the `debug` package to your `spago.dhall` file to be able to import `Debug`.

Now, when `add` is called like:

```
add 10 99
```

Then the values of `x` and `y` will be displayed to the Console:

```
X: 10
Y: 99
```

When your PureScript program is running under Node.js, like we've been doing all along, then the value is output just like `Effect.Console.log` does.

But, when your code is running in the Browser, then your output will be in the Browser's Console log.

When using `spy` in Monadic Code, you'll want to use it in the following ways:

```
app :: AppM
app = do
  s <- get
  pure $ spy "Our state" s ❶
  let x = spy "Our state" s ❷
  pure unit
```

❶ Since `spy` returns `s`, which is our State, we need to wrap that Pure Value in our `Monad's` Context using `pure`. Remember, that every line of our `do` block must be a Monadic Value.

❷ We can assign some random value to the result of `spy`. We're never going to use this Value but since PureScript isn't lazy like Haskell, it will still be evaluated.

Either of these ways works fine, but I tend to use the `let` approach.

Now we're ready to look at an example of how you might use `catchError`. Let's start with the Stack like the above diagram where `StateT` is on top of `ExceptT`:

```
type AppM = StateT Int (ExceptT String (WriterT String Effect)) Unit

runApp
  :: Int
  -> AppM
  -> Effect (Tuple (Either String (Tuple Unit Int)) String)
runApp s = runWriterT <<< runExceptT <<< flip runStateT s
```

If you'd like to following along with this code, you can create a file called `Ch21a.purs` and copy all the code from `Ch21.purs` making sure you change the module name to `Ch21a`. Then replace `AppM` and `runApp` with the above code.

And now we'll replace our validation Function:

```
import Debug (spy) ❶

-- COMPILER WARNING!! ❷
validate :: Int -> AppM
validate n = do
  s <- get
  let x = spy "s in validate" s
  log "HEY!!!!!!!!!!!!!!!!!!!" -- This will NOT be lost on the error case
  put 10 -- This will be lost on the error case but kept on success
  when (n == 0) $ void $ throwError "WE CANNOT HAVE A 0 STATE!"
```

❶ Added to import `spy`. Don't forget to add `debug` to your `spago.dhall`.

❷ The compiler has a warning since you're using `spy`.

First thing to notice about `validation` is that it will throw an error if `n` is zero.

Second, is that it has 2 Side-Effects. It writes to the log and it puts `10` to the State.

And finally, it outputs the Value of the State to the console upon calling this Function, via `spy`.

Next, let's look at a slightly different version of `app` that calls this validation routine:

```
app :: AppM
app = do
  log "Starting App..."
  n <- get
  catchError (validate n) (\err -> do
    s <- get
    let x = spy "s in error handler" s
    log $ "We encountered an error: (" <> err <> ")"
    put 100
  )
  s <- get
  let x = spy "s in app" s
  put $ s + 1
  log "Incremented State"
  pure unit
```

Notice the call to `catchError`. We're attempting to call `validate` and if it fails, then we will call the supplied Lambda.

Also, notice that our error handler sets a default value of `100` to the State and that it logs the fact that we encountered an error. The code in `app` NEVER Short-circuits since we're using `catchError`.

We're also spying on the State AFTER `validate` was called and FAILED, i.e. inside of the error handler. And we're spying on the State AFTER the `catchError` call.

These calls to `spy` are there to help demonstrate that, in fact, we lose the State changes that `validate` made but NOT the writes to the log that it also made.

State is lost because `StateT` is above `ExceptT` in this Stack, but the log is kept because `WriterT` is below `ExceptT`.

Remember, we talked about Stack Order and how we want `ExceptT` on the top of the Stack so that we get back the most information after running the Stack. As we will soon see, that's not the only reason we want `ExceptT` on the top of the Stack.

Now, here's the code we're using for testing:

```
log :: ∀ m. MonadTell String m => String -> m Unit
log s = tell $ s <> "\n"

test :: Effect Unit
test = do
  result1 <- runApp 0 app
  Console.log $ show result1
  result2 <- runApp 99 app
  Console.log $ show result2
```

We've seen this test code before. The only thing to notice here is that the first time we call `runApp` we pass a zero Initial State. This will cause an error in `validate`, where as the second `runApp` will not thanks to its non-zero Initial State.

Now let's run this and we'll do a postmortem on the output. Here's the salient code :

```
validate :: Int -> AppM
validate n = do
  s <- get
  let x = spy "s in validate" s
  log "HEY!!!!!!!!!!!!!!!!!!" -- This will NOT be lost on the error case
  put 10 -- This will be lost on the error case but kept on success
  when (n == 0) $ void $ throwError "WE CANNOT HAVE A 0 STATE!"

app :: AppM
app = do
  log "Starting App..."
  n <- get
  catchError (validate n) (\err -> do
    s <- get
    let x = spy "s in error handler" s
    log $ "We encountered an error: (" <> err <> ")"
    put 100
  )
  s <- get
  let x = spy "s in app" s
  put $ s + 1
  log "Incremented State"
  pure unit
```

Here's the test code's output, first the error run and then the non-error:

```
s in validate: 0 ❶
s in error handler: 0 ❷
s in app: 100 ❸
(Tuple (Right (Tuple unit 101)) "Starting App...\nHEY!!!!!!!!!!!!!!!!!!\nWe
encountered an error: (WE CANNOT HAVE A 0 STATE!)\nIncremented State\n") ❹

s in validate: 99 ❺
s in app: 10 ❻ ❼
(Tuple (Right (Tuple unit 11)) "Starting
App...\nHEY!!!!!!!!!!!!!!!!!!\nIncremented State\n") ❽
```

❶ This shows the Initial State Value that's passed to `validate`. This Value is INVALID since it's zero.

❷ We're seeing the State when we get an error here, which is `0`. This is THE PROBLEM. In `validate`, we set the State to `10` before we throw the error. The change to `10` is LOST in `StateT's catchError` implementation.

❸ This shows the default State set by the error handler, i.e. `100`.

❹ Notice that `"HEY!!!!!!!!!!!!!!!!!!"` is in the log. It was NOT LOST when the error was thrown. This is because `WriterT` is BELOW `ExceptT` in the Stack. If it wasn't, then this too would be LOST.

❺ This shows the Initial State Value that's passed to `validate`. This Value is VALID since it's non-zero.

❻ Notice that there's no output from the error handler here. That's because the Initial State is VALID.

❼ This is the value of the State AFTER returning from `validate`, where the State is set to `10`. This is NOT LOST since we never throw an error.

❽ This is the result of a successful run. It's log is missing the error handler log entry as we'd expect.

If you look at the code for `validate` and if you read the comments, you'll see which lines of code that will be rolled back when an error occurs.

This is quite a big problem since it means that we have to keep this in mind when we write our code.

So what can be done about this?

21.6. Guidelines for Using `ExceptT`

The most obvious thing you can do is to NOT call `catchError` ever. This is not a great solution and besides, you cannot always control your codebase and although you decided to never call `catchError`, someone who comes along after you just might not know not to do this.

Even if you write documentation warning them, they will probably not read it and they can easily fall into this trap.

So, to best avoid the aforementioned problem, you can simply put `ExceptT` on the top of your Stack.

As we learned earlier, this is a good approach because we gain the most information at the end of a run that fails. But now, we have a second reason, i.e. we don't want `StateT's` implementation of `catchError` to be called. And, as it turns out, we don't want any Monad Transformer's `catchError` to be called.

This approach will work fine for 99% of all cases. But what about that 1%? How do we solve those cases?

In our example, let's say we'd wanted our `validate` Function to keep all Side-effects if it succeeds and none if it fails. But let's suppose that runs counter to the Stack we're working in, which it did in the above example.

To solve that, we'd have to build a different Monad Stack for the `validate` Function. We'd build a Stack with ExceptT on the bottom of the Stack.

Then we'd run that Validation Stack and, in the successful case, we would use the results to update our Stack's state, log and whatever else we needed to.

In the error case, we'd optionally call some error handling code very much like `catchError` does, but, instead, we'd probably use `when` or possibly an if-then-else.

This is a bit clunky but since it's the 1% case, it's a fine solution.

So, the biggest walkway here is put `ExceptT` on top. Always. Except when you want to have a complete rollback of all Side-effects when an error occurs, then put it on the bottom.

Part IV: Beyond

Chapter 22. Synchronous and Asynchronous Effects

We've talked a little about the `Effect` Monad mostly to help us write simple test code and build Monad Stacks that could be passed back to our `main` Function.

But what exactly can we do with `Effect`? One way of thinking about `Effect` is that we can use it to interface with the real world.

Another way to think of `Effect` is that it's a `Monad` for Synchronous Effects. And our whole program runs in this `Monad`:

```
main :: Effect Unit
main = do
  log "Hello, world!"
```

No matter how complex our program gets, it eventually becomes one massive Function that runs in the `Effect` Monad.

22.1. Working with the `Effect` Monad

If we want to get a Random Number by using a traditional Random Number Generator, we'll have to call a non-pure Function. That's because each time we call it, we pass it nothing and we get a different number back.

There's a package called `purescript-random` and in that package, the module `Effect.Random` has Functions for generating Random Numbers. Once such function is `random`:

```
random :: Effect Number
```

This Function calls Javascript's `Math.random` Function under the covers and because it's non-pure, it runs inside the `Effect` Monad.

The reason for this is to keep our PureScript code Pure. We have a `main` program that is Pure through and through. `main` always returns `Effect Unit`. So, it doesn't run anything.

Remember when we were looking at Monad Stacks and we returned a `Monad` back from `app`. We didn't execute anything. We just composed or combined a bunch of `Monads` via `bind`, which composes a bunch of Functions.

Nothing was run until we called `runApp`.

We can conceptualize main as if it returns the Effect Monad to the PureScript runtime code, which will then execute it.

This is how we can code in Pure Functions and still affect the outside world. We're never calling random in our code. We only build a Monad, i.e. Effect, that will process the result of such a call:

```
main :: Effect Unit
main = do
  n <- random
  log $ show n
```

Here random isn't a Function. It's a Monad, or more precisely, it's an Effect Number, i.e. a Monadic Value that will bind to n in main:

```
main :: Effect Unit
main = random >>= \n -> log $ show n
```

The bind for Effect is not coded in PureScript. It's code that the compiler will generate, i.e. imperative non-Pure code. But at that point it's out of our hands.

We did our jobs as PureScript programmers, i.e. to combine Monads, and, therefore, Functions, to build a program that WHEN run will process the data given to it. We did this with Pure Functional code.

main is like any other Monad, a complex combination of Functions to be run at a later time. That later time is when we execute the Javascript code that the compiler generated.

Like random, all Browser APIs run in the Effect Monad, e.g. to get the Browser Window:

```
import Web.HTML (window)

window :: Effect Window
```

And to get the current Browser location:

```
import Web.HTML.Window (location)

location :: Window -> Effect Location
```

And to get the href from that location as a String:

```
import Web.HTML.Location (href)

href :: Location -> Effect String
```

And so on.

File I/O also runs in the `Effect` Monad. There are libraries for running in Node. One of those libraries is `Node.FS.Sync`. It has the following Function for reading a File:

```
readTextFile :: Encoding -> FilePath -> Effect String
```

This is the Synchronous version of read, which means that it will block the current thread until the data is fully read from the File. But, when writing in Node, Synchronous reads are usually only done at the beginning of the program to load a configuration file or other initialization information.

Normally, we don't want to block Javascript, since it runs in a Single Thread of execution. We would rather read Asynchronously and handle the result in a callback like we do in Javascript:

```
fs.readFile(fileName, 'utf8', function (err, data) {
  if (err) throw err;
  console.log(fileName)
  console.log(data)
});
```

We can do this in PureScript by using a different module called `Node.FS.Async`:

```
type Callback a = Either Error a -> Effect Unit ❶

readTextFile :: Encoding -> FilePath -> Callback String -> Effect Unit
```

❶ `Callback` is an alias for a Function that will handle the callback and process the File's data.

Here's how we could use this:

```
import Prelude

import Data.Either (Either(..))
import Effect (Effect)
import Effect.Console (log)
import Effect.Exception (Error)
import Node.Encoding (Encoding(..))
import Node.FS.Async (readTextFile)

displayFile :: Either Error String -> Effect Unit
displayFile (Right fileData)  = log fileData ❶
displayFile (Left err)        = log $ show err ❷

main :: Effect Unit
main = do
  readTextFile ASCII "somefile.txt" displayFile ❸
```

❶ Writes the file contents to the console log.

❷ Writes the error, e.g. `Error: ENOENT: no such file or directory, open 'somefile.txt'`.

❸ Initiates the file read with a `Callback`, i.e. `displayFile`.

The `Callback` approach isn't very good. Notice the Type for `Callback`:

```
type Callback a = Either Error a -> Effect Unit
```

`Callbacks` can only do Side-effects since it returns `Effect Unit`.

This is the limitation of running in Node or more specifically in Javascript. It only has a single thread of execution at a time, which is why we cannot afford to block the current thread. If we do, then nothing else can run.

If we're reading a File when we start a program, then we can certainly read it synchronously since we don't mind waiting for the File I/O to complete. But if we're reading a File in a running server under Node, then we don't want to halt the whole server just to read that File.

If you're writing a non-browser program in Haskell, you won't have to worry about these issues since it has proper threading, i.e. multiple concurrent threads.

What we need in PureScript is a non-blocking way to do operations that allow us to write as if our thread has been blocked like we can in Haskell.

22.2. Working with the Aff Monad

While Effect is the Monad that our program ultimately runs in, there's another more powerful effects Monad called Aff, which is a Monad for Asynchronous Effects.

To get any real work done, we're going to rely heavily on Aff. If we want to run HTTP requests while handling user input from the Browser, we're going to use Aff.

On the backend, if we're running in Node, we're going to want to handle multiple requests at a time, which means we're going to run our request handlers in Aff.

Learning how to work within Aff and its ancillary services is imperative to building an application of any level of complexity.

So let's start that process by looking at how we can read a File in an Asynchronous way:

```
import Prelude

import Effect (Effect)
import Effect.Aff (Aff, launchAff_)
import Effect.Class.Console (log) ❶
import Node.Encoding (Encoding(..))
import Node.FS.Aff (readTextFile) ❷

readAFile :: Aff Unit ❸
readAFile = do
  text <- readTextFile ASCII "test.txt"
  log text

test :: Effect Unit
test = launchAff_ readAFile ❹
```

❶ We're importing a different version of log here. This version will work in Effect and any Monad that has a MonadEffect Instance. It turns out that Aff has such an instance meaning we can also call log in Aff.

❷ We need to add node-fs-aff to our spago.dhall file to import this module.

❸ We are running in the Aff Monad.

❹ launchAff_ is one of many ways of running Asynchronous processes, in this case, readAFile.

This code uses a different Node library that allows us to easily work with files in the Aff Monad without having to deal with Callback, which makes our coding much easier.

But, this particular library will throw exceptions and crash your program if it receives an error. In the above code, if test.txt doesn't exist, this will crash.

But this is easy to fix:

```
import Prelude

import Control.Monad.Error.Class (try) ❶
import Data.Either (Either(..)) ❷
import Effect (Effect)
import Effect.Aff (Aff, launchAff)
import Effect.Class.Console (log)
import Node.Encoding (Encoding(..))
import Node.FS.Aff (readTextFile)

readAFile :: Aff Unit
readAFile = do
  result <- try $ readTextFile ASCII "test.txt" ❸
  case result of ❹
    Right fileData -> log fileData
    Left err -> log $ show err

test :: Effect Unit
test = launchAff_ readAFile
```

❶ Import `try` from `MonadError`.

❷ We import `Either` along with it Data Constructors.

❸ Now we `try` to read the File returning an `Either Error String`.

❹ We now check the results of the File read.

The magic elixir here is `try`:

```
try :: ∀ e m a. MonadError e m => m a -> m (Either e a) ❶
```

❶ This takes a Monadic Computation and if it throws an exception, it will catch that exception and return it instead as an `Either` within that same Monadic Context.

As it turns out both `Effect` and `Aff` have `MonadError` Instances, which is why we can use it here.

While the previous approach was simpler, it didn't handle errors and would crash our app.

With this approach, our program is more robust but we have to check for errors explicitly using a `case` expression.

If we were running in a Monad Stack that contained `ExceptT`, then we could convert the `Either` to `ExceptT` using `except`:

```
except :: ∀ e m a. Applicative m => Either e a -> ExceptT e m a
```

This approach would allow us to code the Happy Path since ExceptT will short-circuit for us.

22.2.1. Fibers

Let's imagine now that we want to read a file in one Asynchronous Process while another does something else, e.g. logs to the console every second.

We can run these 2 processes in what PureScript calls ***Fibers***. These aren't Threads as we might have in Haskell or in other multi-threaded languages. We're running in Javascript, so we're limited to a single thread of execution.

Javascript has ***Cooperative Multitasking***, i.e. the current running thread MUST relinquish control to allow other parts of the program to run. That's why you see callbacks in Node and Browser APIs.

The PureScript libraries do a good job of abstracting this away for us as we've seen with reading a File. Although our first example had an explicit callback, we eventually found a library that reduced that cognitive overhead. And if we don't find one, we could probably write one.

So, how do we create a Fiber and run code in it?

One way is to use forkAff:

```
forkAff :: ∀ a. Aff a -> Aff (Fiber a)
```

forkAff takes something that runs in Aff and forks a separate process, i.e. a Fiber, where it will run.

Let's start by writing a Function that runs in Aff that will write to the console every second:

```
import Prelude

import Data.Time.Duration (Milliseconds(..)) ❶
import Effect.Aff (Aff, delay) ❷
import Effect.Class.Console (log) ❸

logEverySecond :: Aff Unit
logEverySecond = go 0 where ❹
  go x = do ❺
    log $ show x ❻
    delay (Milliseconds 1000.0) ❼
    go $ x + 1 ❽
```

❶ Import Milliseconds from Data.Time.Duration.

❷ Import delay, which will block our Fiber for the specified number of milliseconds.

❸ Notice that we're importing the version of log that works in MonadEffect.

❹ Call go with the initial value to log as 0.

❺ Notice the do block starts here, which is running in the Aff Monad.

❻ Log x to the console.

❼ Sleep for 1 second.

❽ Recurse with an incremented Value for x.

Next, let's write another Function that'll wait for 2 seconds and then read from a file. We'll call it readAFileAfterTwoSeconds:

```
readAFileAfterTwoSeconds :: Aff Unit
readAFileAfterTwoSeconds = do
  delay (Milliseconds 2000.0) ❶
  result <- try $ readTextFile ASCII "test.txt"
  case result of
    Right text -> log text
    Left err -> log $ show err
```

❶ We've added a delay for 2 seconds to our most recent version of readAFile.

Here's the contents of test.txt:

```
This is a test. ❶
```

❶ The file has a newline at the end of the file.

Now, we'd like to have both of these Functions running simultaneously. We'll let them run for 5 seconds and then terminate them.

First, we'll write a Function to terminate them called kill:

```
import Effect.Aff (Aff, Fiber, delay, killFiber) ❶
import Effect.Exception (error) ❷

kill :: ∀ a. Fiber a -> Aff Unit
kill = killFiber (error "Killing you softly...") ❸ ❹
```

❶ Added Fiber and killFiber to import list.

❷ Added `error :: String -> Error`. This is passed to the `Fiber` upon canceling. We'll do `Canceler` handling later.

❸ This is Point-free where the `Fiber` is the unnamed Parameter.

❹ The `Error` is constructed using `error`. This error will be passed to the `Canceler`, i.e. the `Fiber's` abortion handler. More on this later.

Second, we'll write some test code to run everything:

```
import Effect (Effect) ❶
import Effect.Aff (Aff, Fiber, delay, killFiber, launchAff_, forkAff) ❷

test :: Effect Unit
test = launchAff_ do ❸
  logger <- forkAff logEverySecond ❹
  fileReader <- forkAff readAFileAfterTwoSeconds ❹
  delay (Milliseconds 5000.0) ❺
  kill logger ❻
  kill fileReader ❻
```

❶ Import `Effect` for `test`.

❷ Added `launchAff_` and `forkAff`.

❸ Ultimately our program runs in `Effect`, which is why `test` runs in `Effect`. But, we want to run our `Fibers` in `Aff`, so we launch an Asynchronous Process using `launchAff_`. We're using `launchAff_ :: Effect Unit` instead of `launchAff :: Effect (Fiber a)` since we don't care about the `Fiber` that this process runs in.

❹ We fork these 2 processes capturing their `Fibers` so we can kill them later.

❺ Sleep for 5 seconds.

❻ Kill our 2 processes. Chances are that the `fileReader` has already exited, but it doesn't hurt to do this just in case.

The above code uses 2 different ways to start an Asynchronous Process, `launchAff_` and `forkAff`. Their differences can be seen from their Type Signatures:

```
launchAff_ :: ∀ a. Aff a -> Effect Unit
forkAff :: ∀ a. Aff a -> Aff (Fiber a)
```

Since `test` is running in the `Effect` Monad, we have to use `launchAff_`. Once we're in the `Aff` Monad, we can use `forkAff`.

When this code is run, we get the following output:

```
0
1
2
This is a test.
 ❶
3
4
```

❶ The blank line is because the file has a newline at the end of the file.

Fibers can return values after they're finished running, but we have to use `runAff` to capture that value:

```
runAff
  :: ∀ a. (Either Error a -> Effect Unit)
  -> Aff a
  -> Effect (Fiber Unit)
```

The first Parameter is a callback Function that will be called when the `Fiber` exits. The second Parameter is the code to run in `Aff`, e.g. a `do` block. `runAff` returns the `Fiber` that was run in the `Effect` Context.

We're going to change `readAFileAfterTwoSeconds` to return the results of reading the file:

```
readAFileAfterTwoSeconds :: Aff String ❶
readAFileAfterTwoSeconds = do
  delay (Milliseconds 2000.0)
  result <- try $ readTextFile ASCII "test.txt"
  pure $ case result of ❷
    Right text -> text ❸
    Left err -> show err ❸
```

❶ Return `Aff String` instead of `Aff Unit`. This `String` represents either an error or the contents of the file. In any real program, this would be `Aff (Either Error String)`, but we're keeping it simple here.

❷ We need to wrap our `Strings` in `Aff` using `pure`.

❸ Return `Strings` in these 2 places.

Now, our test code will have to change to use `runAff`:

```
import Effect.Aff ( Aff, Fiber,
                    delay, killFiber, launchAff_, forkAff, runAff) ❶

test :: Effect Unit
test = do ❷
  fileReader <- runAff case _ of ❸ ❹
    Left err -> log $ show err
    Right result -> log $ "File contents: " <> show result
    $ readAFileAfterTwoSeconds ❺
  launchAff_ do
    logger <- forkAff logEverySecond
    delay (Milliseconds 5000.0)
    kill logger
    kill fileReader
```

❶ We need to add `runAff`.

❷ Added a `do` for the `Effect` Monad.

❸ We must call `runAff` from `Effect`, which is why it's done outside `launchAff_'s do` block.

❹ We're using a wildcard in the `case` expression, which is the `Error` passed to `runAff's` first Parameter, a Function, which is the exit handler for the `Fiber`.

❺ The second Parameter to `runAff` is the code that will run in the `Aff` Monad. In this case, `readAFileAfterTwoSeconds`.

Now, when we run our code we get:

```
0
1
2
File contents: "This is a test.\n"
3
4
```

And if the file is missing:

```
0
1
"Error: ENOENT: no such file or directory, open 'test.txt'"
2
3
4
```

If you're running in Windows, the error message may be different.

22.2.2. Cancelers

Sometimes you want to do some sort of clean-up when the Fiber is killed. This is called a Canceler. You can attach a Canceler to an Aff with cancelWith:

```
cancelWith :: ∀ a. Aff a -> Canceler -> Aff a
```

This will take an Aff and attach a Canceler to it.

But what is a Canceler? Here's its definition:

```
newtype Canceler = Canceler (Error -> Aff Unit)
```

The Data Constructor takes a Function which takes the Error passed by killFiber.

Here's how we'd change our program to use cancelWith on our infinite loop process, logEverySecond:

```
import Effect.Aff ( Aff, Fiber, Canceler(..),
                    delay, killFiber, launchAff_, runAff, cancelWith) ❶
import Effect.Exception (error, message) ❷

kill :: ∀ a. Fiber a -> Aff Unit
kill = killFiber (error "Killing you softly...") ❸

test :: Effect Unit
test = do
  logger <- runAff (const $ pure unit) ❹ ❺
    $ logEverySecond
      # flip cancelWith (Canceler (log <<< message)) ❻ ❼
  fileReader <- runAff case _ of
    Left err -> log $ show err
    Right result -> log $ "File contents: " <> show result
    $ readAFileAfterTwoSeconds
  launchAff_ do
    delay (Milliseconds 5000.0)
    kill logger
    kill fileReader
```

❶ Removed forkAff and added Canceler and cancelWith.

❷ Added message to get just the message from Error. If we show Error, it'll display the error message

and the Stack Trace. With message, we just get the error text.

❸ There were no changes to kill. It's just here for reference so we can see the error message that it will pass, which will show up in the program's output.

❹ Using runAff so we can add a Canceler.

❺ logEverySecond never exits since it only gets canceled, which means that this case will never be called. const $ pure unit is here just to make the Type Checker happy.

❻ cancelWith is flipped since it's first Parameter is an Aff a, which, in this case, is logEverySecond.

❼ The Canceler Function will display the error message by using message on Error as opposed to displaying the whole Stack Trace via show.

This program's output is:

```
0
1
2
File contents: "This is a test.\n"
3
4
Killing you softly...
```

22.2.3. AVars

Imagine you have a Fiber that reads a File and you want that data to be processed by another Fiber.

We can accomplish this by using a mechanism called an AVar, which stands for ***Asynchrounous Variable***.

An AVar is an Asynchronous Variable that can contain data, or can be empty. A Fiber can read and write to the AVar and even block until data becomes available.

To see how one might use AVars let's write a program that runs 2 Fibers, one, which reads a File and sends it to the other via an AVar:

```
import Prelude

import Control.Monad.Error.Class (try)
import Data.Either (Either(..))
import Effect (Effect)
import Effect.Aff (Aff, launchAff_, forkAff)
import Effect.Aff.AVar (AVar) ❶ ❷
import Effect.Aff.AVar as AVar ❶ ❸
import Effect.Class.Console (log)
import Node.Encoding (Encoding(..))
import Node.FS.Aff (readTextFile)

readAFileAfterTwoSeconds :: AVar String -> Aff Unit ❹
readAFileAfterTwoSeconds fileAVar = do
  delay (Milliseconds 2000.0)
  result <- try $ readTextFile ASCII "test.txt"
  case result of
    Right text -> AVar.put text fileAVar ❺
    Left err -> log $ show err

processFile :: AVar String -> Aff Unit ❻
processFile fileAVar = do
  text <- AVar.take fileAVar ❼
  log text

test :: Effect Unit
test = launchAff_ do
  fileAVar <- AVar.empty ❽
  void $ forkAff $ processFile fileAVar ❾
  void $ forkAff $ readAFileAfterTwoSeconds fileAVar ❾
```

❶ Make sure to import `Effect.Aff.AVar` to get the `Aff` version of the `AVar` API. `Effect.AVar` is for the API in the `Effect` Monad, which is complicated with callbacks and doubly-nested `Effects`. I avoid this API whenever possible. Both of these modules are in the package `purescript-avar`, which means that `avar` must be added to `spago.dhall`.

❷ Import JUST the Type here. That's because on the next line we're importing everything else qualified by the name `AVar`. This avoids the awkward Type name of `AVar.AVar`.

❸ Import the API qualified. This is done to avoid name collision and ambiguity since the API Functions have generic names, e.g. `put`, `take`, `kill`, `new`, etc. Instead, we'll refer to them with fully-qualified names, e.g. `AVar.put`.

❹ `readAFileAfterTwoSeconds` takes an `AVar` that it'll `put` the File data to. Notice how the `AVar` Type takes a single Type Parameter. This is the Type of data that you can `put` to it. In this case, it's a `String`.

❺ When the file is successfully read, the AVar is written to using put.

❻ processFile takes an AVar that it'll take the File data from. Notice how the AVar Type takes a single Type Parameter. This is the Type of data that you can take from it. In this case, it's a String.

❼ Here's where the AVar is read from. This BLOCKS the Fiber until the data becomes available, which, in the case of this program, will take at least 2 seconds thanks to the delay in readAFileAfterTwoSeconds.

❽ This is how to create a new AVar that's initially empty. To create an AVar with initial data, use AVar.new and pass it an initial Value.

❾ Both Fibers get passed the fileAVar to facilitate inter-Fiber communication.

This program's output is:

```
This is a test.
```

AVars can be used as a poor-man's queue where multiple Fibers write to the same AVar. In that scenario, trying to put to a non-empty AVar will block the Producer until the AVar is emptied by the Consumer using take.

And trying to take an empty AVar will block the Consumer until the AVar is written to by the Producer.

AVars can be part of a read-only Record that's passed around by ReaderT. This is possible since the AVar, itself, is never mutated.

If this is not readily intuitive, think of AVars as Constant References. The data structure that represents the AVar is immutable while the data structure that the AVar refers to is mutable via the AVar API.

You may be wondering if AVars are just a case of Globally Shared State. They are NOT. The AVar still needs to be passed around like any other immutable Value. But since the underlying storage is mutable, access to that underlying data is controlled by an API that guarantees coherence.

AVars can be used in programs that have a single Producer and multiple Consumers, but only if all Consumers are of equal stature.

For example, imagine you have a single Producer that produces requests for one of multiple Consumers to do an Ajax call to a Web Service.

All Consumers would call take, which will block them until something is put to the AVar. Once the Producer calls put, one and only one of the Consumers will get to work on the request. All others will still be blocked. There is no way to discriminate which Consumers will get some values as opposed to others when using a single AVar.

And there is no way for all Consumers to consume a single put to an AVar.

But what if we want to have one or more Producers and one or more Consumers such that a single Value

produced from any Producer will be consumed by all Consumers?

22.2.4. Busses

A Bus is the PureScript Type that implements what's known as a ***Data Bus***, which is a data structure and associated API that facilitates data to be written by one or more Publishers. This data will be broadcast to one or more Subscribers, who will each get a copy of the data.

Busses can be read-only, write-only or read-write.

Let's write a simple program that uses a Bus:

```
import Prelude

import Control.Monad.Error.Class (try)
import Data.Either (Either(..))
import Data.String.Common (toUpper, toLower) ❶
import Data.Tuple (Tuple(..))
import Effect (Effect)
import Effect.Aff.Bus (BusR, BusW) ❷
import Effect.Aff.Bus as Bus ❸
import Effect.Aff (Aff, launchAff_, forkAff)
import Effect.Class.Console (log)
import Node.Encoding (Encoding(..))
import Node.FS.Aff (readTextFile)

readAFile :: BusW String -> Aff Unit ❹
readAFile fileBus = do
  result <- try $ readTextFile ASCII "test.txt"
  case result of
    Right text -> Bus.write text fileBus ❺
    Left err -> log $ show err

processFile :: (String -> String) -> BusR String -> Aff Unit ❻
processFile convert fileBus = do
  text <- Bus.read fileBus ❼
  log $ convert text ❽

test :: Effect Unit
test = launchAff_ do
  fileBus <- Bus.make ❾
  let Tuple readBus writeBus = Bus.split fileBus ❿
  void $ forkAff $ processFile toUpper readBus ⓫
  void $ forkAff $ processFile toLower readBus ⓬
  void $ forkAff $ readAFile writeBus ⓭
```

❶ Import `String` conversion Functions to be used by the Subscribers.

❷ Import JUST the Types here. That's because on the next line we're importing everything else qualified by the name `Bus`. This avoids the awkward Type name of `Bus.BusR` or `Bus.BusW`.

❸ Import the API qualified. This is done to avoid name collision and ambiguity since the API Functions have generic names, e.g. `read`, `write`, `split`, etc. Instead, we'll refer to them with fully-qualified names, e.g. `Bus.read`.

❹ `readAFile` takes a `Bus` that it'll `write` the File data to. Notice how the `BusW` Type is a write-only `Bus` that takes a single Type Parameter. This is the Type of data that you can `write` to the `Bus`. In this case, it's a

`String`. This is similar to our `AVar` version.

❺ When the file is successfully read, the `Bus` is written to.

❻ `processFile` takes 2 Parameters. First, a `String` conversion Function. Second, it takes a read-only `Bus`, i.e. `BusR`, that it'll `read` the File data from. Notice how the `BusR` Type takes a single Type Parameter. This is the Type of data that you can `read` from the `Bus`. In this case, it's a `String`.

❼ Here's where the `Bus` is read from. This will block the Subscriber if there is no data available at the time of the call.

❽ Before logging, `processFile` applies the `String` conversion Function.

❾ `make` creates a bidirectional `Bus`, i.e. `BusRW`, a read-write `Bus`.

❿ `split` splits a bidirectional `Bus` into a `BusR` and `BusW`, i.e. a read-only and a write-only `Bus`, respectively.

⓫ The first `processFile` converts the File's contents using `toUpper`. It's also passed the read-only `Bus`.

⓬ The second `processFile` converts the File's contents using `toLower`. It too is passed the read-only `Bus`.

⓭ The write-only `Bus` is passed to `readFile`.

This program's output is:

```
THIS IS A TEST.

this is a test.
```

Chapter 23. Coding With Effects

Return to the project that you've been coding in and create a file called `Ch23a.purs` and add the following:

```
module Ch23a where

import Prelude ❶

import Effect (Effect)
import Effect.Class.Console (log) ❷

test :: Effect Unit
test = do
  log "placeholder" ❸
```

❶ Once again we are not writing anything from Prelude so we'll just import all of it.

❷ We're now using this version of `log` so we can use it in `Effect` and `Aff`.

❸ We'll delete this soon enough. It's just here to keep the compiler happy.

Next, change your `Main` module to import `Ch23a` and call its `test` Function:

```
module Main where

import Prelude
import Effect (Effect)

import Ch23a as Ch23a

main :: Effect Unit
main = Ch23a.test
```

If you run:

```
npx spago run
```

you should see it build and print:

```
[info] Build succeeded.
placeholder
```

23.1. Simple AVar Program Specification

First, we're going to write a simple program to use `AVars`. To prepare our project for this, we'll need to add 2 packages to our `spago.dhall` file, viz. `aff` and `avar`.

Once you've done this, run:

```
npx spago build
```

We'll have 3 Fibers. One that's half a clock, i.e. the Tick part of the clock. The second that the second half of the clock, i.e the Tock part. They will communicate via an AVar.

And the third is a bomb that waits for a specified number of Tick-Tock combinations before detinating with a `BOOM!!` message to the console.

They will all communicate using a single AVar, which we'll call `ttAVar` which stands for tick-tock AVar.

The Tick Fiber will look at `ttAVar` once a second and if it's Value is tock, then it'll change it to tick. The Tock Fiber will do the opposite.

THe Bomb Fiber will check `ttAVar` every 500 milliseconds to see if the time has changed from tick to tock. When it does, it'll consider that a count down and once it reaches the specified number, it'll detinate and our program will exit.

If that's a bit confusing, don't worry, we'll build this one step at a time.

Start by modeling the Type for `ttAVar`, i.e. the Tick-Tock Value. Do that before reading on.

Here's the Type:

```
data TickTock = Tick | Tock
```

Now, since there's no other obvious Types to model, we'll start coding our Fibers. We'll model Types as we go if we need them along the way.

First, just write the Type Signature for the Tick Fiber. Call it `tick`. Re-read the above specifications if you get stuck on this step. Give it try before reading on.

The Fiber only needs the `ttAVar` and it'll return nothing but still runs in `Aff`:

```
import Effect.Aff.AVar (AVar) ❶ ❷

tick :: AVar TickTock -> Aff Unit
```

❶ Added this import for our Type Signature.

❷ Make sure you use the `Aff` version of `AVars`. There's another set of Functions for using `AVars` in the `Effect` Monad by importing `Data.AVar`. This API is far more complex, so I always aviod it when possible.

Make a plan for what you think the Tick Fiber should do before coding it. Then write the code and return here once you're done or hit a brick wall.

Here's my plan:

- Read the `ttAVar`
- Delay 1 second
- If it's `Tock` then change it to `Tick` otherwise leave it alone
- Recurse

Since we have the Type Signature, we'll start with left-hand side of the implementation:

```
tick :: AVar TickTock -> Aff Unit
tick ttAVar = do ❶
  ???
```

❶ Start with a `do` block since we're operating in `Aff`.

Next, read the `AVar`:

```
import Effect.Aff.AVar as AVar ❶

tick :: AVar TickTock -> Aff Unit
tick ttAVar = do
  tt <- AVar.take ttAVar ❷
  ???
```

❶ Add this import with the alias so that instead writing `take` which is ambiguous to anyone read my code, I prefix it with `AVar.` to qualify which `take` I'm using.

❷ We `take` the `AVar` since we'll probably change it.

Next, we'll delay for 1 second:

```
tick :: AVar TickTock -> Aff Unit
tick ttAVar = do
  tt <- AVar.take ttAVar
  delay (Milliseconds 1000.0)
  ???
```

Next, we'll check to see if we should change it to `Tock`:

```
import Data.Time.Duration (Milliseconds(..)) ❶
import Effect.Aff (Aff, delay) ❷

tick :: AVar TickTock -> Aff Unit
tick ttAVar = do
  tt <- AVar.take ttAVar
  delay (Milliseconds 1000.0)
  if tt == Tock then AVar.put Tick ttAVar else AVar.put tt ttAVar
  ???
```

❶ Added to import `Milliseconds` and its Data Constructor.

❷ Added `delay` to the import list.

Looking at this code tells me that I can factor out the call to `AVar.put`:

```
tick :: AVar TickTock -> Aff Unit
tick ttAVar = do
  tt <- AVar.take ttAVar
  delay (Milliseconds 1000.0)
  flip AVar.put ttAVar if tt == Tock then Tick else tt ❶
  ???
```

❶ Flipped the Parameters to `AVar.put` to get the `ttAVar` up front.

Now, that I've written this, I can see a problem with the `if` logic. Can you see what wrong?

The `if` is unnecessary. If the value is `Tock` then we change it to `Tick`, otherwise we put back the current Value. But the current Value can ONLY be `Tick` since there's only 2 possible Values of `TickTock`.

So, we're always writing back `Tick`. Let's fix my mistake:

```
tick :: AVar TickTock -> Aff Unit
tick ttAVar = do
  void $ AVar.take ttAVar ❶
  delay (Milliseconds 1000.0)
  AVar.put Tick ttAVar
  ???
```

❶ We no longer care what the current value is.

And finally, we recurse:

```
tick :: AVar TickTock -> Aff Unit
tick ttAVar = do
  void $ AVar.take ttAVar
  delay (Milliseconds 1000.0)
  AVar.put Tick ttAVar
  tick ttAVar
```

A couple of important things to note here. First, the `delay` MUST be between the `AVar.take` and the `AVar.put`. Why do you think this is so important?

The reason for the critical placement of the `delay` is that the Value must stay `Tick` or `Tock` long enough for the Bomb Fiber to see that it's transitioned. So, before we change it, we delay 1 second giving the Bomb Fiber plenty of time to see its current Value, before we change it.

You may have noticed that in the Specifications, I specified that the Bomb Fiber only delays for 500 milliseconds. This is half the time of the Tick and Tock Fibers. This is because of what's known as the ***Nyquist Frequency***.

Basically, when sampling a signal, in our case a clock, the sample rate must be at least twice the frequency of the signal to make sure we get a good sample.

Imagine we look at the clock every minute, but it changes once a second. It's changing much faster than we're checking and because of that we're missing many of it's changes from Tick to Tock. In fact, it's most likely that sampling once a minute will always return a Tick or a Tock making us think that the clock isn't running at all.

The second thing to notice about the Tick Fiber is that it never exits. This means we'll have to use `killFiber` if we hope to have our program exit.

Now, write the Tock Fiber before advancing further.

Here's `tock`, which is nearly identical to `tick`:

```
tock :: AVar TickTock -> Aff Unit
tock ttAVar = do
  void $ AVar.take ttAVar
  delay (Milliseconds 1000.0)
  AVar.put Tock ttAVar ❶
  tock ttAVar ❷
```

❶ Put `Tock` instead of `Tick` like we did in `tick`.

You may be thinking that we should combine these 2 Fibers into a single Clock Fiber and you'd be right. Originally, my thinking was to have you build a simple program that has many Fibers working together, but now looking at this, it seems that a single Clock Fiber makes more sense.

Combine these 2 Fibers into one called `clock` before moving on.

Here's the results of combining `tick` and `tock`:

```
clock :: AVar TickTock -> Aff Unit
clock ttAVar = do
  void $ AVar.take ttAVar
  delay (Milliseconds 1000.0)
  AVar.put Tock ttAVar
  void $ AVar.take ttAVar
  delay (Milliseconds 1000.0)
  AVar.put Tick ttAVar
  clock ttAVar
```

Next, make a plan for how the Bomb Fiber will work before reading on.

Here's my plan:

- If count = detination count then log "BOOM!!" and exit Fiber
- Delay 500 milliseconds
- If waiting for `Tick` and `ttAVar` is `Tick` then wait for `Tock`.
- If waiting for `Tock` and `ttAVar` is `Tock` then increment count
- Recurse

Looking at this plan, it's clear that we'll need another Type. Sure we could use a `Boolean` called `waitingForTick` or `waitingForTock`, but a Type communicates so much more.

Model a Type called `BombState` before reading on:

Here's my take on that:

```
data BombState = WaitingTick | WaitingTock
```

Now code the Bomb Fiber expecting that it will be passed `detinationCount`. Go as far as you can before reading on.

We'll start with the Type Signature:

```
bomb :: AVar TickTock -> Int -> Aff Unit
```

This will take `ttAVar` and the `detinationCount`.

Next, we'll start the implementation in the usual fashion:

```
bomb :: AVar TickTock -> Int -> Aff Unit
bomb ttAVar detinationCount = ???
```

We know at this point that we're going to have to keep track of some state, i.e. the current `count` and the `BombState`. But we're not passing it into the Fiber. So, looks like we're going to need a `go` Function, i.e. a local Function that we can recurse with:

```
bomb :: AVar TickTock -> Int -> Aff Unit
bomb ttAVar detinationCount = go ??? where
  go :: ???
```

Before we can fill in the first `???`, we'll have to figure out what the Type Signature is for `go`. Before reading any further, write the Type Signature for `go` keeping in mind the state it must keep track of.

As stated earlier, `go` needs to keep track of `count` and `BombState`:

```
bomb :: AVar TickTock -> Int -> Aff Unit
bomb ttAVar detinationCount = go ??? where
  go :: Int -> BombState -> Aff Unit
  go count state = do ❶
    ???
```

❶ Starting with a `do` since `go` is in `Aff`.

Now, we can fill in the `???`. Our initial `count` will be `0` and we'll first wait for `Tick`:

```
bomb :: AVar TickTock -> Int -> Aff Unit
bomb ttAVar detinationCount = go 0 WaitingTick where ❶
  go :: Int -> BombState -> Aff Unit
  go count state = do
    ???
```

❶ Set initial `count` and `state` on `go`.

Now, stop here and see if you can finish this. Then return back here.

If we've reached our destination count, we are done:

```
bomb :: AVar TickTock -> Int -> Aff Unit
bomb ttAVar detinationCount = go 0 WaitingTick where
  go :: Int -> BombState -> Aff Unit
  go count state = do
    if count == detinationCount then log "BOOM!!"
    else do
      ???
```

Now, we delay 500 milliseconds:

```
bomb :: AVar TickTock -> Int -> Aff Unit
bomb ttAVar detinationCount = go 0 WaitingTick where
  go :: Int -> BombState -> Aff Unit
  go count state = do
    if count == detinationCount then log "BOOM!!"
    else do
      delay (Milliseconds 500.0)
      ???
```

Now, we'll read the `AVar`. Notice that we're reading the `AVar` and not taking it. That's because we're NOT changing it:

```
bomb :: AVar TickTock -> Int -> Aff Unit
bomb ttAVar detinationCount = go 0 WaitingTick where
  go :: Int -> BombState -> Aff Unit
  go count state = do
    if count == detinationCount then log "BOOM!!"
    else do
      delay (Milliseconds 500.0)
      tt <- AVar.read ttAVar
      ???
```

Now, we check to see if we on `Tick` and if so, we start waiting for `Tock`:

```
bomb :: AVar TickTock -> Int -> Aff Unit
bomb ttAVar detinationCount = go 0 WaitingTick where
  go :: Int -> BombState -> Aff Unit
  go count state = do
    if count == detinationCount then log "BOOM!!"
    else do
      delay (Milliseconds 500.0)
      tt <- AVar.read ttAVar
      case state of
        WaitingTick ->
          if tt == Tick then log "Tick" *> go count WaitingTock ❶ ❷
          else go count state ❸
        WaitingTock -> ???
```

❶ I've added a `log` of `Tick` when we see a `Tick` for the clock. This is so we can watch as our program runs.

❷ Recurse with a state change, i.e. `WaitingTock`.

❸ Recurse with NO state change.

If you think you can finish, stop reading and give it a try. Don't forget to refer to the aforementioned plan for this Function. When you're done return back here and continue reading.

The final step is to count if we find a `Tock`:

```
bomb :: AVar TickTock -> Int -> Aff Unit
bomb ttAVar detinationCount = go 0 WaitingTick where
  go :: Int -> BombState -> Aff Unit
  go count state = do
    if count == detinationCount then log "BOOM!!"
    else do
      delay (Milliseconds 500.0)
      tt <- AVar.read ttAVar
      case state of
        WaitingTick ->
          -- COMPILER ERROR!!
          if tt == Tick then log "Tick" *> go count WaitingTock
          else go count state
        WaitingTock ->
          if tt == Tock then log "Tock" *> go (count + 1) WaitingTick ❶
          else go count state
```

❶ Increment the `count` when we see a `Tock`.

The compiler error is complaining:

```
  No type class instance was found for

    Data.Eq.Eq TickTock
```

Fix this error before reading on.

We need add an `Eq` instance for `Tick`:

```
data TickTock = Tick | Tock
derive instance eqTickTock :: Eq TickTock
```

And now we can delete our placeholder `test` Function and rewrite the `test` Function to create the `AVar` and fork the Fibers before reading further.

Here's the first part of the `test` Function:

```
import Effect.Aff (Aff, delay, launchAff_) ❶

test :: Effect Unit
test = launchAff_ do ❷
  ???
```

❶ Added `launchAff_` to the import list.

❷ Using `launchAff_` to run in `Aff`. This will make working with `Fibers` and `Affs` much easier than working in `Effect`.

Next, we'll create the `AVar`:

```
test :: Effect Unit
test = launchAff_ do
  ttAVar <- AVar.empty ❶
  ???
```

❶ Create this empty. This gives us time to start the Fibers before the clock starts to run.

Next, we'll start the Clock Fiber and the Bomb Fiber:

```
import Effect.Aff (Aff, delay, launchAff_, forkAff) ❶

test :: Effect Unit
test = launchAff_ do
  ttAVar <- AVar.empty ❶
  clockFiber <- forkAff $ clock ttAVar
  bombFiber <- forkAff $ bomb ttAVar 3 ❷
  ???
```

❶ Added 'forkAff' to the import list.

❷ The `detinationCount` is 3 seconds.

Now that the Fibers are running, we can start the clock. Realize that the Clock Fiber will block when it tries to `take` the `AVar` before this step. The same is true for the Bomb Fiber when it tries to `read` the `AVar`:

```
test :: Effect Unit
test = launchAff_ do
  ttAVar <- AVar.empty ❶
  -- COMPILER WARNING!!
  clockFiber <- forkAff $ clock ttAVar
  -- COMPILER WARNING!!
  bombFiber <- forkAff $ bomb ttAVar 3
  AVar.put Tick ttAVar
```

We'll ignore the warning that reminds us that we created `clockFiber` and `bombFiber` but we're not using them. Yet.

Now, run your program with:

```
npx spago run
```

And we get `spago` errors:

```
[error] Some of your project files import modules from packages that are not
in the direct dependencies of your project.
To fix this error add the following packages to the list of dependencies in
your config:
- datetime
```

Fix this by adding the missing package to `spago.dhall` and rerun.

It should produce the following output:

```
Tick
Tock
Tick
Tock
Tick
Tock
BOOM!!
```

The only problem is that the program NEVER exits. Hit `Ctrl-C` to kill it. How should we solve this problem?

There are 2 ways we can handle this. First approach we could take is to have the Bomb Fiber communicate to the Clock Fiber to let it know that it exited. Then the Clock Fiber will exit.

The other appraoch is to watch the Bomb Fiber and when it exits then we kill the Clock Fiber.

The first approach is nice if Clock Fiber needs to do a controlled shutdown, e.g. it needs to do some clean up work. But our Clock Fiber is very simple and has no such requirement. There's no reason to create all the complexity of this approach when it's unnecessary.

So, we're going to choose the second approach. But how do we know when a Fiber exits?

Go search `Pursuit` for `killFiber` and look on the same page for another Function that we can use to know when our Bomb Fiber has exited. When you find it return here.

I hope you were able to find `joinFiber`:

```
joinFiber :: Fiber ~> Aff
```

This Type Signature doesn't tell use too much. But the docs say:

```
Blocks until the fiber completes, yielding the result. If the fiber throws an
exception, it is rethrown in the current fiber.
```

This tells us that we can call `joinFiber` and then when it returns we can safely kill our Clock Fiber using `killFiber`.

Go ahead and change the `test` to properly kill the Clock Fiber once the Bomb Fiber has exited. Then keep reading.

Here are those changes:

```
import Effect.Aff (Aff, delay, launchAff_, forkAff, killFiber, joinFiber) ❶
import Effect.Exception (error) ❷

test :: Effect Unit
test = launchAff_ do
  ttAVar <- AVar.empty
  clockFiber <- forkAff $ clock ttAVar
  bombFiber <- forkAff $ bomb ttAVar 3
  AVar.put Tick ttAVar
  joinFiber bombFiber ❸
  killFiber (error "Exploded") clockFiber ❹
```

❶ Added `killFiber` and `joinFiber` to the import list.

❷ Added this import.

❸ Join the Bomb Fiber blocking until Bomb Fiber exits.

❹ Kill the Clock Fiber with an `Error`. This `Error` is ignored by `clock` since it was started with `forkAff` and now `runAff` or some other Function that supports `Cancelers`.

Now run you program and it should exit properly.

But instead you get the following:

```
[error] Some of your project files import modules from packages that are not
in the direct dependencies of your project.
To fix this error add the following packages to the list of dependencies in
your config:
- exceptions
```

Fix this and try again. The program should now exit on its own.

23.2. Effects Program Specifications

Next, we're going to write a very small application that produces Random Numbers with Publishers and Subscribers to those numbers.

But before we do, create a file called `Ch23b.purs`:

```
module Ch23b where

import Prelude

import Effect (Effect)
import Effect.Class.Console (log)

test :: Effect Unit
test = do
  log "placeholder"
```

Also, change your `main` to call `Ch23b.test`:

```
module Main where

import Prelude
import Effect (Effect)

import Ch23b as Ch23a

main :: Effect Unit
main = Ch23b.test
```

The appliation we're going to write in `Ch23b` will run separate `Fibers` for each Publisher and Subscriber in a Monad Stack that contains `ReaderT` and `StateT`. Our Stack will have `Aff` at the base since we're running in a `Fiber`.

Publishers will produce Random Numbers delaying 1 second before doing so. Then they will publish a `String` message onto a `Bus` when their randomly generated number passes some Predicate.

The Publisher will then countdown a count in the State and continue only if the count is positive.

There will be 3 Publishers.

One will check for values over `0.5`. Another will check for values less than `0.5`. The final one will check to see if the value is over `0.1`.

Subscribers will listen to the `Bus` and log the data they get from the `Bus` to the console. There will be 1 Subscriber.

Stop here and take the time to reread these specifications.

Once you've done that, ask yourself, if you know enough to write this program. I suspect you won't think so and you'd be right. There's still a few things you'll need to either figure out or refresh your memory on.

So, I'm going to walk you through the process one step at a time.

First, write down all of the requirements that you don't know how to do. This is what I call ***Technological Hurdles***. Take a few minutes to create a list before reading on.

I'm going to produce a list of Technological Hurdles that will contain some things that we've already talked about in previous Chapters. This is because it's very likely that you've forgotten some things or maybe you're out of practice or just haven't quite mastered them yet:

- How to create a Random Number
- How to make a Monad Stack
- How to create a `Fiber`
- How to run a Monad Stack in a `Fiber`

- How to create a `Bus`
- How to Publish to a `Bus`
- How to Subscribe to a `Bus`

Hopefully, I've listed all of the things on your list. Don't worry if not since we're going to go over every piece of how to write this.

The whole point of making a list of Technological Hurdles is to give us a list of things that we must solve before we can ever hope to write the program at hand. Once we've conquered this list, writing the program is a foregone conclusion and should be pretty easy.

23.2.1. Creating a Random Number

How do we create a Random Number?

How would you go about figuring this out if you weren't reading this book?

There are 2 ways that pops to mind. First, use Google. The problem with this approach is PureScript is pretty obscure and I wouldn't expect there to be a StackOverflow question that answers this for us.

At the time of this writing, a Google search produces no such easy answers. Give it a try and see if the world has changed.

The next approach is to search `Pursuit` for the word `random`. When I do this, I'm hoping that I'll find some Function with that name. Before reading further do this search for yourself and see if you have any luck.

This approach doesn't always work, especially when you guess the wrong word for the Function. Luckily though, there aren't too many synonyms for `random`. When we search for `random`, we find that the Function we're looking for is first in the list:

```
import Effect.Random (random)

random :: Effect Number
```

Can you see a problem with this Function? If not, go back and reread the specifications.

The problem is that `random` runs in the `Effect` Monad but the specifications says that we're going to have 3 Publishers each running in their own `Fiber`.

But from last Chapter, we learned that `Fibers` run in `Aff` not `Effect`. So how do we solve this problem?

We could find a version of `random` that runs in `Aff`. So, go look for one and see if you can find it.

Turns out, there isn't one.

So, now what?

The only thing we can do here is try to figure out a way to convert an `Effect` to an `Aff`. But how do we figure out how to do that? What do we even search for?

If I didn't know the answer, I would have no idea what to look for. So, we can't really search, but we can browse. We know that there are Functions for running `Fibers` in `Aff`. We've seen them from the last Chapter.

Look back at the previous Chapter and see what those Functions are called. Then search `Pursuit` for them and browse through the Functions in the same module to see if anything will help us there.

When I look back at the previous Chapter, I find `forkAff`, `runAff` and `launchAff_`. Searching `Pursuit` for these Functions finds them in the same module, `Effect.Aff`.

Browsing the Functions in `Effect.Aff`, there are a couple of candidates for solving our problem, `makeAff`, `launchAff` and `runAff`.

First off, we'll reject `forkAff` since it must be running in `Aff` to start with.

Next, we'll look at `makeAff`:

```
makeAff
  :: ∀ a. ((Either Error a -> Effect Unit) -> Effect Canceler)
  -> Aff a
```

The Type Signature looks scary. So, we're going to skip that for now and come back only if we have to.

Next, we look at `launchAff`:

```
launchAff :: ∀ a. Aff a -> Effect (Fiber a)
```

This looks backwards from what we want. We have an `Effect` and we want an `Aff`. So that won't work.

And next `runAff`:

```
runAff
  :: ∀ a. (Either Error a -> Effect Unit)
  -> Aff a
  -> Effect (Fiber Unit)
```

And this Type Signature looks pretty scary too. Now, we have no choice but to tackle the scary Type Signatures.

So we read the docs for `runAff` and find:

```
Forks an Aff from an Effect context and also takes a callback to run when it
completes, discarding the Fiber.
```

This returns a new `Fiber`. Not exactly what we want, since it's overkill. But we could make this work if we have to.

Let's return to `makeAff` in the hopes that it's more to our liking:

```
makeAff
  :: ∀ a. ((Either Error a -> Effect Unit) -> Effect Canceler)
  -> Aff a
```

And let's look at its docs:

```
Constructs an Aff from low-level Effect effects using a callback. A Canceler
effect should be returned to cancel the pending action. The supplied callback
may be invoked only once. Subsequent invocation [sic] are ignored.
```

That's a lot to digest so let's eat it one bite at a time:

```
Constructs an Aff from low-level Effect effects using a callback...
```

This sounds exactly what we want. We have an `Effect Number` and we'd like an `Aff Number`. Let's keep reading:

```
...A Canceler effect should be returned to cancel the pending action...
```

Okay, so this requirement seems doable. We can return a do-nothing `Canceler`. Search the `Effect.Aff` page on `Pursuit` for `Canceler` and see if you can find one we could use before reading on.

If you found `nonCanceler`, then you hit the nail on the head. So we can return that as our `Canceler`.

Let's keep reading:

```
...The supplied callback may be invoked only once. Subsequent invocation [sic]
are ignored.
```

This is telling us that our `Callback` may be invoked once. That makes me think that it could never be called. It's not too clear when it would never be called. Maybe they're referring to the fact that the `Fiber` may not

be killed. If so, then that makes sense.

We now have a potential solution for this Technological Hurdle. Let's write a little test program to make sure it's going to work. We're going to work in `Ch23a.purs`.

The first thing we're going to want to do is write a version of `random` that runs in `Aff`. We're going to use `makeAff` on `random` to do that. But how?

Let's revisit its Type Signature:

```
makeAff
  :: ∀ a. ((Either Error a -> Effect Unit) -> Effect Canceler)
  -> Aff a
```

Parsing this one part at a time shows us that the first Parameter is a Function. And that Function's first Parameter is another Function.

The first Function may not look too familiar but it should. Or to be more accurate, the authors of this Type Signature should have used the following Type Alias to make things easier to understand:

```
type Callback a = Either Error a -> Effect Unit
```

We saw this in the previous Chapter in `Node.FS.Async`. Too bad it's not in the main libraries. So, we're going to have to remember that any time we see `Either Error a -> Effect Unit` that we're looking at a Callback Function.

One thing to realize about this Callback Function is that, unlike Callbacks in Javascript, we're being given a Callback that we must call. If you've programmed in Javascript at all, then you're used to providing the callback. But in the case of `makeAff`, it's providing us the Callback that it expects us to call.

The next part of `makeAff's` Type Signature is the first Parameter's return Type, which is the `Canceler`. This will be easy since we found `nonCanceler`.

But, one huge issue that's easy to miss here is that we don't return a `Canceler` but instead return `Effect Canceler`. So, we'll try to use a `do` block when we're coding that Lambda.

And finally, `makeAff` returns the `Aff a` that we so desperately need, i.e. a version of `random` or any Function for that matter, that runs in `Aff` instead of `Effect`.

Go ahead and write out the new Type Signature for our version of `random`. We'll call it `randomAff`. Do this before reading more.

The Type Signature is actually quite simple:

```
import Effect.Aff (Aff) ❶

randomAff :: Aff Number
```

❶ Import `Aff`.

Now, write the rest using `makeAff`. Here it is for reference:

```
makeAff
  :: ∀ a. ((Either Error a -> Effect Unit) -> Effect Canceler)
  -> Aff a
```

Read on only when you've finished or have gotten stuck.

This part of the code is NOT trivial. It'll take some practice before it becomes second nature. Let's take it one step at a time. We know we're going to use `makeAff`. So, let's add that:

```
import Effect.Aff (Aff, makeAff) ❶

randomAff :: Aff Number
randomAff = makeAff ???
```

❶ Added `makeAff` to import list.

Next, we know that `makeAff` takes a single Parameter that's a Function. And that Function takes a single Parameter, which we determined is a Callback Function. So let's write that much:

```
randomAff :: Aff Number
randomAff = makeAff \cb -> ??? ❶
```

❶ `cb` stands for our callback Function.

We know that our Lambda is going to return an `Effect Canceler` so that means that our code is going to run in the `Effect` Monad. So, let's add the `do` keyword:

```
randomAff :: Aff Number
randomAff = makeAff \cb -> do
  ???
```

And now we can return the Canceler:

```
import Effect.Aff (Aff, makeAff, nonCanceler) ❶

randomAff :: Aff Number
randomAff = makeAff \cb -> do
  pure nonCanceler
```

❶ Added `nonCaceler` to import list.

Well, that compiles. But wait. How does `random` get called? And what are we supposed to do with the Callback Function, i.e. `cb`, that we've been passed?

Let's attack those one step at a time. First, let's call `random`:

```
import Effect.Random (random) ❶ ❷

randomAff :: Aff Number
randomAff = makeAff \cb -> do
  n <- random ❸
  pure nonCanceler
```

❶ Don't forget to import `random`.

❷ You have to add `random` to your `spago.dhall`.

❸ We can call `random` here since we coding in the `Effect` Monad in this `do` block.

Since we needed to add `random` to `spago.dhall`, you'll have to run the following to get this code to compile in the editor, i.e. if you're using VSCode:

```
npx spago build
```

Saving the file will show warnings that we'll ignore for now.

Now, what do we do with this Random Number, `n`? Also, we still haven't used our Callback, `cb`. What do think?

If you're stuck, here's the Callback Function's Type Signature:

```
Either Error a -> Effect Unit
```

It takes an `Either Error a`. But what's `Error`? Look it up on `Pursuit` by searching for `makeAff`, clicking on it and once you have the docs for `makeAff`, click on `Error`. (This procedure is to make sure we're looking at the correct `Error` since the word 'error" is probably used by many libraries making searching difficult.)

When you do this, you will find:

```
data Error :: Type
```

Okay, so that's not helpful at all. But the doc says:

```
The type of JavaScript errors.
```

And if we browser ever so slightly down we find:

```
error :: String -> Error
```

This will construct an `Error` from a `String`. So, we probably want to use this to pass an error to the Callback Function:

```
cb $ Left $ error "Something went horribly awry"
```

But `random` can't error. So, what do we pass to `cb` in the success case?

Don't forget it's Type is `a`:

```
Either Error a -> Effect Unit
```

If you answered `Right`, you're right (pun intended):

```
import Data.Either (Either(..)) ❶

randomAff :: Aff Number
randomAff = makeAff \cb -> do
  n <- random
  cb $ Right n
  pure nonCanceler
```

❶ Import `Either's` Data Constructors to be able to use `Right`.

Looking back over our code, we can see that we provide `makeAff` with a computation in the `Effect` Monad that it'll run in `Aff` for us. That computation calls `random` and returns the successful result, via the Callback Function that `makeAff` passed to us.

This is a lot of work for something that doesn't have a `Canceler`. And it is. But, there is an alternative that's

much simpler and we'll change `randomAff` as soon as we learn of it.

Okay, so it looks like we've overcome our first Technological Hurdle.

23.2.2. Making a Monad Stack

The specifications stated that we need to have a `StateT` and a `ReaderT` but it didn't specify exactly what for. So, let's add that now.

- A read-only structure will be passed to all `Fibers` that contains the read-write `Bus`.
- Each `Fiber` will have a State that contains a count of how many Random Numbers to generate before exiting.

These specifications are a pretty terrible design, but are here only to help us flex our muscles.

First, create a Type Alias called `Config` that contains the `Bus` before reading further. This will be what `ReaderT` passes to every Function.

Here's the Type Alias for `Config`:

```
import Effect.Aff.Bus (BusRW) ❶ ❷

type Config = { bus :: BusRW String } ❸ ❹
```

❶ Import `BusRW`.

❷ Need to add `aff-bus` to `spago.dhall`.

❸ `BusRW` is a bidirectional bus, i.e. we can read and write to it. This is just easier since both Publishers and Subscribers are going to get this.

❹ `String` is the Type of the Values that will be written to the `Bus`.

Don't forget to run the following every time you add a package to `spago.dhall` so the compiler will work in VSCode:

```
npx spago build
```

Also, remember that you if you've typed this code and saved the file BEFORE you added the required package, in this case, `aff-bus`, you'll have to resave the file AFTER you've rebuilt.

Now, create a Type Alias for the State called `State` that contains our `count` before reading on.

Here's the Type Alias for `State`:

```
type State = { count :: Int }
```

Now, make a Type Alias called `FiberM`, since this represents the Monad Stack that our Fibers will run in. Put `ReaderT` on the top of the Stack and `Aff` on the bottom of the Stack.

Do this before reading ahead.

Let's start slowly:

```
import Control.Monad.Reader.Trans (ReaderT) ❶
import Control.Monad.State.Trans (StateT)

type FiberM a = ReaderT r? m? a ❷
```

❶ Import `ReaderT` and `StateT`.

❷ The `r?` and `m?` are placeholders until we figure out what Monomorphic Types to put here.

We need to replace `r?` with `Config`:

```
type FiberM a = ReaderT Config m? a
```

Next, we should replace `ReaderT's` underlying `Monad`, `m?`, with `StateT`:

```
type FiberM a = ReaderT Config (StateT s? m?) a ❶
```

❶ The `s?` and `m?` are placeholders until we figure out what Monomorphic Types to put here.

Now we need to replace `s?` with `State`:

```
type FiberM a = ReaderT Config (StateT State m?) a
```

And last, we replace `StateT's` underlying `Monad`, `m?`, with `Aff` since it's at the base of the Stack:

```
type FiberM a = ReaderT Config (StateT State Aff) a
```

When you see a nested Type that represents a Monad Stack, realize that the outer most or leftmost Monad, in this case `ReaderT`, is on top of the Stack. The next Monad, going from left to right is the next highest in the Stack and so on. In our case, the second from the top of the Stack is `StateT`. And finally, we have the base of the Stack containing `Aff`.

And now, we've overcome another Technological Hurdle.

23.2.3. Creating a `Fiber`

Although we've done this in the previous Chapter, it's possible you've already forgotten how to do this. One way to refresh your memory is to reread the previous Chapter.

Another way is to look at the last bit of code we wrote in that Chapter. Since the latter is far simpler, take the time now to turn back and look at the last program we wrote at the very end of the previous Chapter to see how we create a `Fiber`.

Return here once you've found it.

Okay, so I'm going to assume you looked and you either found out how to create a `Fiber` or came up short. If you figured it out, great, but pretend, for just a moment, that you did not.

Let's say, you know you want to create an Asynchronous Process but you don't know how. But you do know about the module `Effect.Aff`. Look that module up on `Pursuit` and browse the module's documentation for a Function that will create a new `Fiber`.

Try it and see what you find.

When I do this, the first thing I encounter is `launchAff`:

```
launchAff :: ∀ a. Aff a -> Effect (Fiber a)
```

This takes code written in the `Aff` Monad and returns the `Fiber`. Note that this runs in the `Effect` Monad. So, it's good for running a `Fiber` from inside our `main` or, in our case throughout this book, inside our test code.

But what if we're not in `Effect`. What if we're in `Aff` and we'd like to create another `Fiber`?

Search the page for `Aff a -> Aff (Fiber a)` and then continue on.

When I do this, I find:

```
forkAff :: ∀ a. Aff a -> Aff (Fiber a)
```

This is the `Aff` equivalent of `launchAff`.

So, now we know how to create a `Fiber` in either `Effect` or `Aff`. We have now knocked down another Technological Hurdle.

23.2.4. Running a Monad Stack in a `Fiber`

We learned in the Chapter on Monad Stacks, how to run our Application in a Monad Stack, but how do we do this in a `Fiber`?

Well, when faced with such a question, we should ask ourselves what's the difference between running in the `main` of our program and running in a `Fiber`?

How is the code different? Do we actually code differently?

The answer to this is not really. We code pretty much the same either way.

The next question to ask is how did we run our Stack back then?

From our earlier work, we know that we're going to compose together a bunch of run Functions and then pass the Monad to them and we'll get our final result, albeit buried in a Tuple or two Tuples or more.

We can use that experience when we write a Function called `runFiberM` that'll run the `FiberM` Monad Stack.

Go ahead and write `runFiberM` using `FiberM's` definition as your guide and then read on to see how you did.

If you got stuck on the Type Signature, here's what I wrote first:

```
runFiberM :: FiberM Unit -> _
```

I'm letting the compiler figure out the result type for me. Then I realized what I should've written is the following, since we're running in a `Fiber`:

```
runFiberM :: FiberM Unit -> Aff ?x
```

So stop here and see if you can finish coding, then continue reading.

The next step is the left-hand side of the implementation:

```
runFiberM :: FiberM Unit -> Aff ?x
runFiberM = ??? ❶
```

❶ No Parameter here since our run Functions for Monad Stacks are usually written Point-free, i.e. using Function Composition.

And now, we've got to think about things, in particular, we have to consider our Stack:

```
type FiberM a = ReaderT Config (StateT State Aff) a
```

From this, we can see that we need to run `ReaderT` first, since it's on the top of our Stack:

```
import Control.Monad.Reader.Trans (ReaderT, runReaderT) ❶

runFiberM :: FiberM Unit -> Aff ?x
runFiberM = ??? <<< flip runReaderT { bus } ❷
```

❶ Added `runReaderT` to the import list.

❷ We pass a `Config` that contains our `bus`.

We're passing a `Bus` into the `runReaderT` Function, but where are we getting this from?

The only place we can get this from is from whoever calls us. So let's add it:

```
runFiberM :: BusRW String -> FiberM Unit -> Aff ?x ❶
runFiberM bus = ??? <<< flip runReaderT { bus } ❷
```

❶ Added `BusRW String`.

❷ Added `bus` Parameter.

We could've also passed `Config` instead of `BusRW String`. In this case, it's six of one, half a dozen of the other since `Config` only has 1 field. But if it had multiple fields, then passing `Config` would've been a much better choice.

Realize that `runReaderT` returns it's underlying Monad. We can see this from its Type Signature:

```
runReaderT :: ∀ r m a. ReaderT r m a -> r -> m a ❶
```

❶ Removed the redundant Parentheses that are present in the `Pursuit` documentation.

Notice the return Type is `m a`. But what Type is `m`?

To figure that out we have to look at `FiberM`:

```
--              ReaderT   r            m          a
type FiberM a = ReaderT Config (StateT State Aff) a ❶
```

❶ `m` is `StateT State Aff`.

So `runReaderT's` return Type, i.e. `m a`, in our case will be `StateT State Aff`.

This tells us that the next thing we need to do is to run `StateT`:

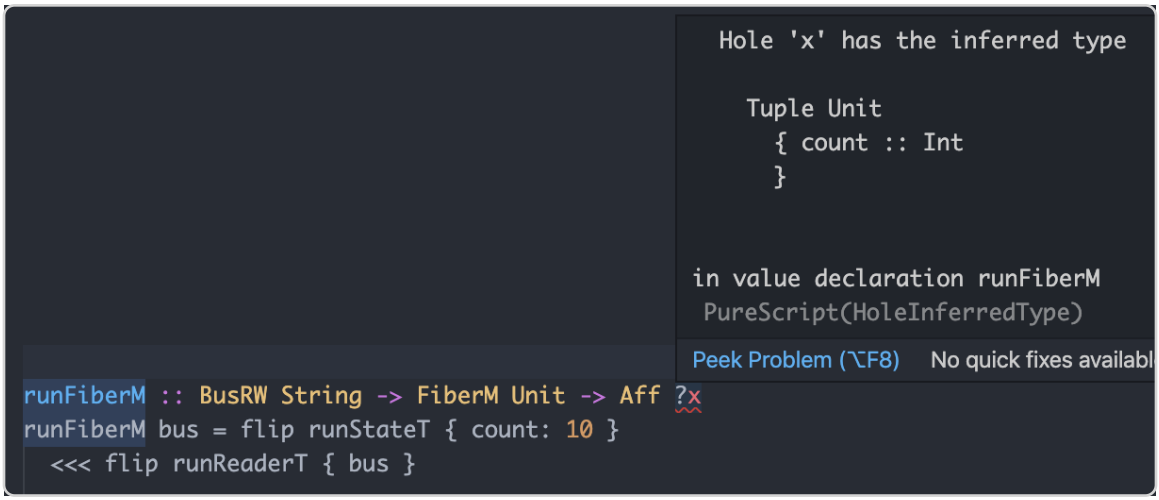
```
import Control.Monad.State.Trans (StateT, runStateT) ❶

runFiberM :: BusRW String -> FiberM Unit -> Aff ?x -- COMPILER WARNING!!
runFiberM bus = flip runStateT { count: 10 } ❷
  <<< flip runReaderT { bus }
```

❶ Added `runStateT` to the import list.

❷ Pass in our read-write `State`. We could also pass this in to the Function, but for now, we're just going to hardcode it here.

Now we have the following warning:



So, we can now just copy and paste our Type Hole, changing `{ count

Int }` to `State`:

```
import Data.Tuple (Tuple) ❶

runFiberM :: BusRW String -> FiberM Unit -> Aff (Tuple Unit State) ❷
runFiberM bus = flip runStateT { count: 10 } <<< flip runReaderT { bus }
```

❶ Import `Tuple`.

❷ The Type Alias, `State`, is used here instead of the Type `{ count :: Int }` from the compiler warning.

And another Technological Hurdle falls.

23.2.5. Making a `Bus`

We've made a `Bus` in the previous Chapter, but let's assume for a moment that you're coding and you don't have this book handy and you want to figure out how to create a `Bus`.

What approach would you take?

We've already established that Google and, in particular, StackOverflow isn't going to be much help. So, we are going to rely on `Pursuit` again.

Search `Pursuit` for `Bus` and then continue reading.

When I do, I find the first entry is a Type called `Bus` in a module called `Effect.Aff.Bus` and when I click on it, I find a most confusing definition:

```
data Bus (r :: # Type) a ❶
```

❶ As of version 0.14 of PureScript, `# Type` is replaced with `Row Type`.

The first thing I notice is that this is a Type Constructor without a Data Constructor. That tells me that there will be a Function that constructs a `Bus`. This Function is called a ***Smart Constructor***.

By the way, there is a Data Constructor, but it's not exported from the module, `Effect.Aff.Bus`, which is why it's not in the documentation.

The next thing I notice is that the first Type Parameter to the Type Constructor is a Row Type. I can tell that this is the case because of the special syntax for Row Types, i.e. `# Type`.

And the last thing I notice is that the Type is `Type`. That means that `r` is a Row Type of ANY Type.

Looking slightly down the page I find the following related Type Aliases:

```
type BusRW = Bus (read :: Cap, write :: Cap)
type BusR' r = Bus (read :: Cap | r)
type BusR = BusR' ()
type BusW' r = Bus (write :: Cap | r)
type BusW = BusW' ()
```

But what is `Cap`? Clicking on it brings up the following definition:

```
data Cap
```

Clicking on the `Source` link to see the implementation brings up:

```
data Cap
```

This Type has no Data Constructor. It's what I call a ***Marker Type***.

It's just some Type that has meaning to us. We can easily surmise that `Cap` is a Type for the labels in the Row Type that differentiates the Capabilities that a `Bus` has.

Let's look in detail at the different `Bus` definitions.

Here's `BusRW`:

```
type BusRW = Bus (read :: Cap, write :: Cap)
```

This is a read-write `Bus`. It's Row has both Capabilities.

Here's `BusR'`:

```
type BusR' r = Bus (read :: Cap | r)
```

This is a `Bus` that AT LEAST has read Capabilities. It could have more, but it must have read. The `| r` tells us that this Row could be extended.

Here's `BusR`:

```
type BusR = BusR' ()
```

This is a `Bus` that ONLY has read Capabilities. It leverages `BusR'` and gives it a Row with NO additional Capabilities.

Here's `BusW'`:

```
type BusW' r = Bus (write :: Cap | r)
```

This `Bus` has AT LEAST write Capabilities. It could have more, but it must have write. The `| r` tells us that this Row could be extended.

Here's `BusW`:

```
type BusW = BusW' ()
```

This is a `Bus` that ONLY has write Capabilities. It leverages `BusW'` and gives it a Row with NO additional Capabilities.

This is all fine and good, but how do we create a `Bus`? Check out the rest of the `Effect.Aff.Bus` module to see if you can find a way, then continue on.

Browsing the page, you probably came across `make`:

```
make :: ∀ m a. MonadEffect m => m (BusRW a)
```

There's a few things of note here.

First, it doesn't just run in `Effect` or `Aff`. Instead, it runs in ANY `Monad`, `m`, that supports `MonadEffect`. But what is `MonadEffect`?

Click on it and find out before reading further.

Here's `MonadEffect's` Typeclass definition, which can be accessed from `Pursuit` by clicking on the `Source` link:

```
class Monad m <= MonadEffect m where
  liftEffect :: forall a. Effect a -> m a
```

The documentation says:

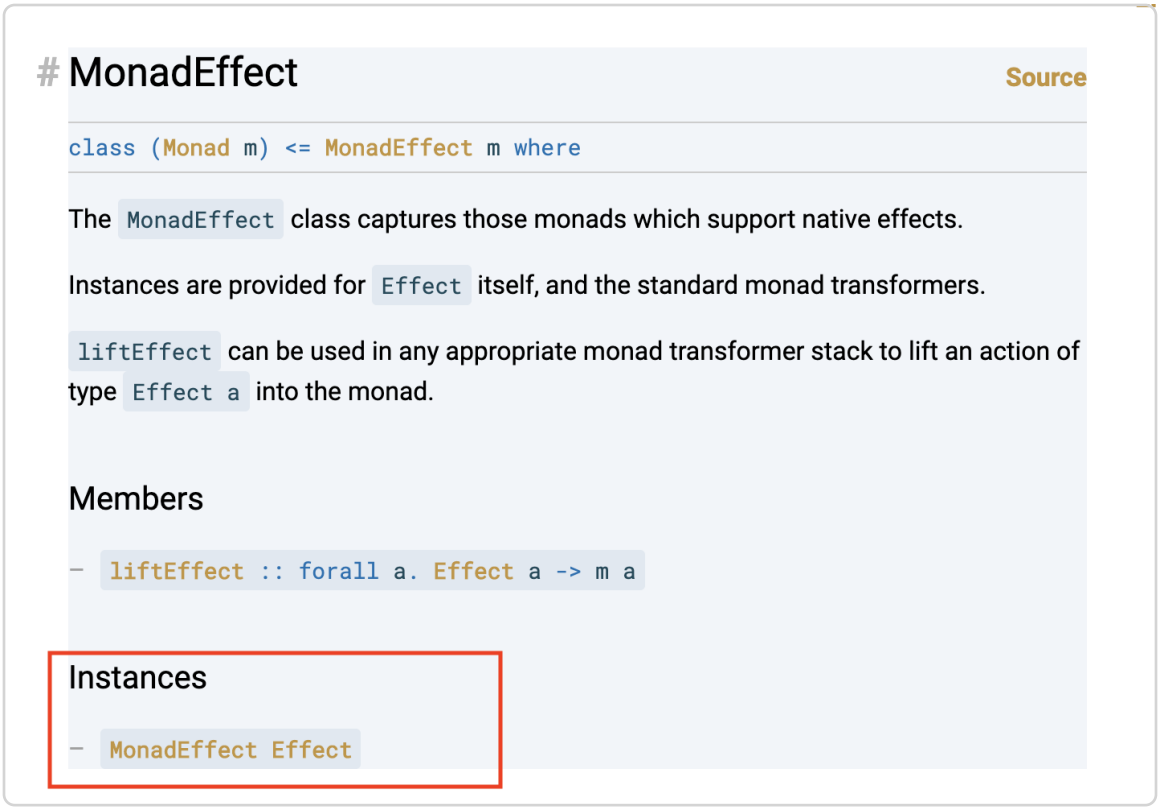
```
The MonadEffect class captures those monads which support native effects.
```

Looking at `liftEffect`, it's clear that it takes something that runs in `Effect` and ***lifts*** it into the `Monad`, `m`.

The docs also say that Monad Transformers support this, i.e. they all have instance of `MonadEffect`. But does this work when `Effect` is NOT at the base of the Stack, e.g. when `Aff` is at the base, like it is in our Stack?

The real question here is can we lift `Effect` into `Aff`?

How can we determine if this is true since the only Instance listed in the `MonadEffect` docs is `Effect`:



MonadEffect Source

```
class (Monad m) <= MonadEffect m where
```

The `MonadEffect` class captures those monads which support native effects.

Instances are provided for `Effect` itself, and the standard monad transformers.

`liftEffect` can be used in any appropriate monad transformer stack to lift an action of type `Effect a` into the monad.

Members

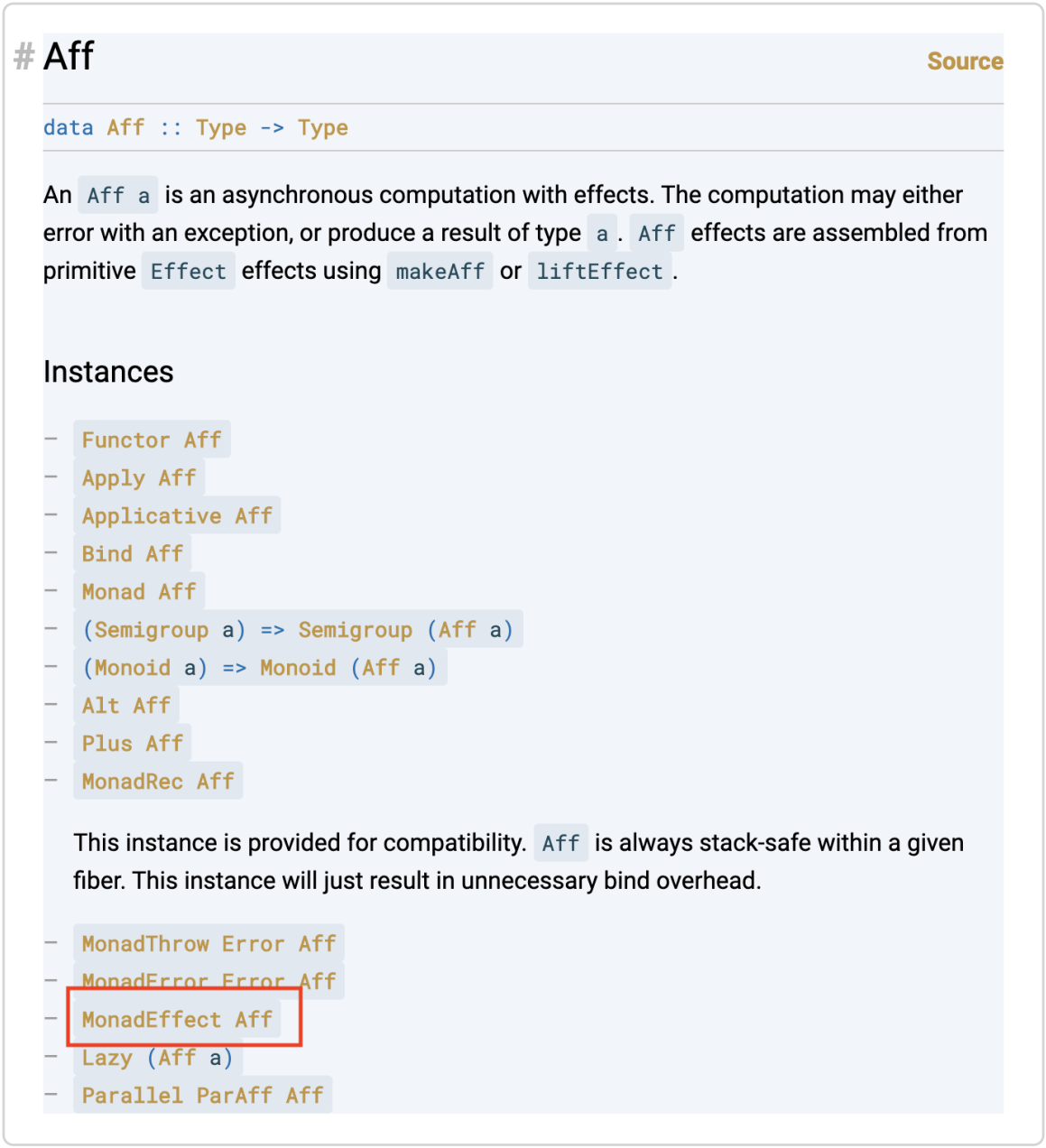
- `liftEffect :: forall a. Effect a -> m a`

Instances

- `MonadEffect Effect`

Search `Pursuit` for `Aff` and see what Instances `Aff` supports then continue reading.

When you do this, you will see that `Aff` indeed does support `liftEffect`:



Aff Source

```
data Aff :: Type -> Type
```

An `Aff a` is an asynchronous computation with effects. The computation may either error with an exception, or produce a result of type `a`. `Aff` effects are assembled from primitive `Effect` effects using `makeAff` or `liftEffect`.

Instances

- `Functor Aff`
- `Apply Aff`
- `Applicative Aff`
- `Bind Aff`
- `Monad Aff`
- `(Semigroup a) => Semigroup (Aff a)`
- `(Monoid a) => Monoid (Aff a)`
- `Alt Aff`
- `Plus Aff`
- `MonadRec Aff`

  This instance is provided for compatibility. `Aff` is always stack-safe within a given fiber. This instance will just result in unnecessary bind overhead.

- `MonadThrow Error Aff`
- `MonadError Error Aff`
- `MonadEffect Aff`
- `Lazy (Aff a)`
- `Parallel ParAff Aff`

This is great since that means we could `make` a `Bus` inside a `Fiber`.

At this point, we have no idea if we will need to, but, now, we know that `make` will work in `Effect` and `Aff`.

You may be wondering if there's a `liftEffect`, then is there a `liftAff`? Go look it up and come back when you have found the answer.

When you do this, you find out that, just like `liftEffect`, `liftAff` is a Typeclass Method:

```
class MonadEffect m <= MonadAff m where
  liftAff :: Aff ~> m ❶
```

❶ Written using Natural Transformation syntax.

Notice that `MonadEffect` is a Superclass of `MonadAff`, which means that any `Monad`, `m`, that can be lifted into `Aff` can also be lifted into `Effect`.

One major word of caution. All this lifting may make you think that you can lift anything into anything. Well, you can't.

The most salient example is that you cannot lift `Aff` into `Effect` using `liftAff`. That's because `Effect` does NOT have a `MonadAff` Instance. For that, we have to run an `Aff` as an `Effect` using `runAff` or `launchAff`. But realize these both fork the `Aff` and so we cannot wait for the results in our `Effect` code but must use the callback mechanism of `runAff` instead.

Now we have a way to make our `Bus` using `make` and we can do it in either `Effect` or `Aff` since they both have `MonadEffect` Instances.

This breaks down yet another Technological Hurdle.

But, before we move on to the next hurdle, let's simplify `randomAff` with `liftEffect`:

```
import Effect.Class (liftEffect) ❶

randomAff :: Aff Number
randomAff = liftEffect random
```

❶ Import `liftEffect`.

Make this change in your code before reading further.

You'll also need to modify your imports as such:

```
import Data.Either (Either(..)) ❶
import Effect.Aff (Aff) ❷
```

❶ Removed this import completely.

❷ Removed unused imports.

23.2.6. Publish to a `Bus`

We can probably just look in the same module as `make` to figure out how to publish to a `Bus`. So do that now and return here when you have the answer.

Here's the Function I found:

```
write :: ∀ a r. a -> BusW' r a -> Aff Unit
```

Notice the `BusW'`. This says that we need a `Bus` that AT LEAST has the write Capability. It could have others, but it must have at least that. This means we can pass it a read-write or a write-only `Bus`.

The docs say:

```
Pushes a new value to the Bus, yielding immediately.
```

This tells us that the Subscribers must get their data immediately afterwards. To be more accurate, one after another. We are in Javascript Land after all.

Only 1 more Technological Hurdler to go.

23.2.7. Subscribing to a `Bus`

After the previous section, one would expect that there's a Function called `read` that let's use read data that's published to a `Bus`. And we'd be right. On the same page of `Pursuit` we find:

```
read :: ∀ a r. BusR' r a -> Aff a
```

The docs say:

```
Blocks until a new value is pushed to the Bus, returning the value.
```

This lets us know that if there's no data on the `Bus` then our `Fiber` will relinquish control back to the main Javascript Event Loop, i.e. other parts of our program will be allowed to run.

When we return from this blocking call, we will have the data and we can continue with it.

And with that, we've overcome all of our Technological Hurdles and are poised to begin writing our program.

23.3. Coding our Effects Program

Now that we've done the work to figure out how to do all of the things we didn't already know how to do, aka. Technological Hurdles, we're ready to start coding our application.

Here are the specifications again:

[Our program] will run separate `Fibers` for each Publisher and Subscriber in a Monad Stack that contains `ReaderT` and `StateT`. Our Stack will have `Aff` at the base since we're running in a `Fiber`.

Publishers will produce Random Numbers delaying 1 second before doing so. Then they will publish a `String` message onto a `Bus` when their randomly generated number passes some Predicate.

The Publisher will then countdown a count in the State and continue only if the count is positive.

There will be 3 Publishers.

One will check for values over `0.5`. Another will check for values less than `0.5`. The final one will check to see if the value is over `0.1`.

Subscribers will listen to the `Bus` and log the data they get from the `Bus` to the console. There will be 1 Subscriber.

Take time to understand these specifications before moving on.

The next step in coding this program involves creating a plan of action. We've already written the following:

```
import Prelude

import Control.Monad.Reader.Trans (ReaderT, runReaderT)
import Control.Monad.State.Trans (StateT, runStateT)
import Data.Tuple (Tuple)
import Effect (Effect)
import Effect.Aff (Aff)
import Effect.Aff.Bus (BusRW)
import Effect.Class (liftEffect)
import Effect.Class.Console (log)
import Effect.Random (random)

type Config = { bus :: BusRW String }

type State = { count :: Int }

type FiberM a = ReaderT Config (StateT State Aff) a

randomAff :: Aff Number
randomAff = liftEffect random

runFiberM :: BusRW String -> FiberM Unit -> Aff (Tuple Unit State)
runFiberM bus = flip runStateT { count: 10 } <<< flip runReaderT { bus }
```

So, the question is what to do next? What would you write next?

If it was me, I'd want to write the main functionality, which we'll call `test` since our `main` already calls that. And even if I cannot get completely through it, writing it first will inform me as to all of the pieces I'm missing.

Another, perfectly valid approach, is to write ancillary support Functions that we need, but we've already

done some of that already, even though I'm pretty sure there will be more.

So, throw away the placeholder `test` and replace with the main functionality of this program in a new Function called `test`. When it comes to forking `Fibers`, call your Publisher `randomGenerator` and your Subscriber `logger`.

To get your code to compile, write those functions but just stub them out, i.e. make a do-nothing version of them. In this case, you can just return `pure unit` for now.

Give it a try and read on once you've finished or need a little help.

First, we always start with the Type Signature:

```
test :: Effect Unit
```

Next, the implementation as a `do` block:

```
test :: Effect Unit
test = do
  ???
```

Now, we should first create our `Bus`, which we know we can do in `Effect`:

```
import Effect.Aff.Bus as Bus ❶

test :: Effect Unit
test = do
  bus <- Bus.make
  ???
```

❶ Import qualified so we don't have name collision on simple names like `make`, `read` and `write`.

Next, we're going to fork our Publishers and Subscribers. We could do this with `runAff`, but we don't need their return Values since there will be none.

So, we're going to use `forkAff` instead which means we'll need to be in `Aff` to do that. So, we'll need to add `launchAff_`:

```
import Effect.Aff (Aff, launchAff_) ❶

test :: Effect Unit
test = launchAff_ do ❷
  bus <- Bus.make ❸
  ???
```

❶ Added `launchAff_` to import list.

❷ The body of `do` now runs in `Aff`.

❸ Luckily, `make` runs in both `Effect` and `Aff`.

At this point, we're ready to run the Publishers and Subscribers. It's always best to have the Subscribers run before the Publishers:

```
import Effect.Aff (Aff, launchAff_, forkAff) ❶

test :: Effect Unit
test = launchAff_ do
  bus <- Bus.make
  void $ forkAff $ runFiberM bus logger ❷ ❸ ❹
  ???
```

❶ Added `forkAff` to the import list.

❷ We're going to have to add a do-nothing version of `logger` later.

❸ Our Stack has `Aff` at the bottom, which is what `runFiberM` returns. That process is forked as a separate process by `forkAff`.

❹ We throw away the `Fiber` that `forkAff` returns using `void`.

Now, we'll run the Publishers, which will be `randomGenerator` with a different Predicate for each. In our case, the Predicate is a Function that takes a `Number` and returns a `Boolean`:

```
test :: Effect Unit
test = launchAff_ do
  bus <- Bus.make
  void $ forkAff $ runFiberM bus logger -- COMPILER ERROR!!
  void $ forkAff $ runFiberM bus $ randomGenerator (_ > 0.5) ❶ ❷
  void $ forkAff $ runFiberM bus $ randomGenerator (_ < 0.5) ❶ ❷
  void $ forkAff $ runFiberM bus $ randomGenerator (_ > 0.1) ❶ ❷
```

❶ We're going to have to add a do-nothing version of `randomGenerator` later.

❷ The Predicates come straight from the specifications.

Let's comment out all the lines with `logger` and `randomGenerator` for now since we haven't written them yet. This way other compiler errors will show as we code until we are ready to write them:

```
test :: Effect Unit
test = launchAff_ do
  bus <- Bus.make
  -- void $ forkAff $ runFiberM bus logger
  -- void $ forkAff $ runFiberM bus $ randomGenerator (_ > 0.5)
  -- void $ forkAff $ runFiberM bus $ randomGenerator (_ < 0.5)
  -- void $ forkAff $ runFiberM bus $ randomGenerator (_ > 0.1)
  pure unit ❶
```

❶ Had to add this since a `do` block must end with an Expression.

Now, it's clear that we're always doing `void $ forkAff $` for each `Fiber`. We can factor that out into `runFiberM`.

Go ahead and move that code to `runFiberM` before moving forward.

Here is the change:

```
-- COMPILER ERROR!
runFiberM :: BusRW String -> FiberM Unit -> Aff (Tuple Unit State)
runFiberM bus = void <<< forkAff ❶
  <<< flip runStateT { count: 10 }
  <<< flip runReaderT { bus }
```

❶ Notice how we've changed `void $ forkAff` to `void <<< forkAff` to keep `runFiberM` Point-free.

We'll return to this compiler error in a moment.

For now, though, we'll remove the duplicate code from `test` since it's been factored out into `runFiberM`:

```
test :: Effect Unit
test = launchAff_ do
  bus <- Bus.make
  -- runFiberM bus $ logger
  -- runFiberM bus $ randomGenerator (_ > 0.5)
  -- runFiberM bus $ randomGenerator (_ < 0.5)
  -- runFiberM bus $ randomGenerator (_ > 0.1)
  pure unit
```

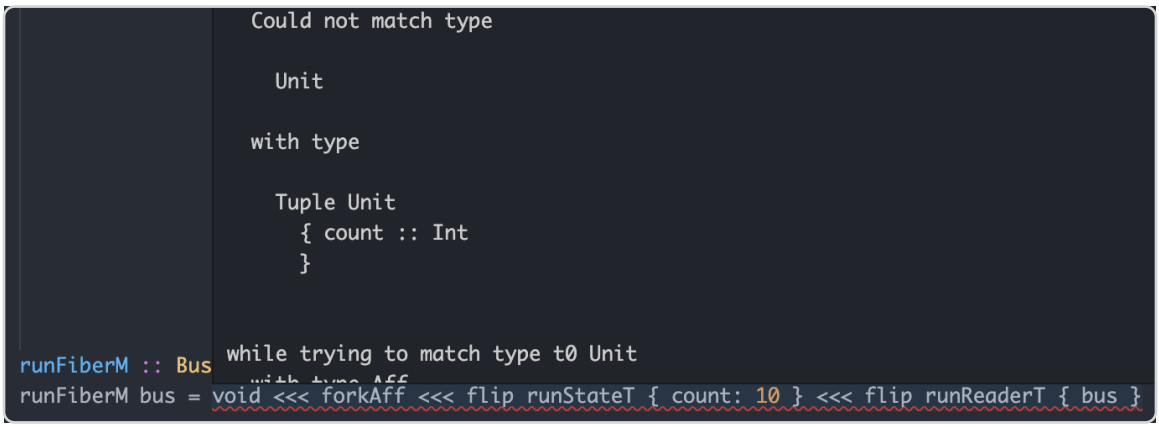
We're keeping these lines commented out for now.

Notice that we had to pass `bus` to `runFiberM`. We could factor that out too:

```
test :: Effect Unit
test = launchAff_ do
  bus <- Bus.make
  let forkFiberM = runFiberM bus ❶
  -- forkFiberM logger
  -- forkFiberM $ randomGenerator (_ > 0.5)
  -- forkFiberM $ randomGenerator (_ < 0.5)
  -- forkFiberM $ randomGenerator (_ > 0.1)
  pure unit
```

❶ This somewhat reduces the noise.

And now we're ready to deal with the compiler error on `runFiberM`, which shouldn't be too surprising since we've changed the output:



The compiler is expecting a `Unit` but got a `Tuple Unit State`. This is because we called `void` which voids the `Tuple`. Let's fix that:

```
runFiberM :: BusRW String -> FiberM Unit -> Aff Unit ❶
runFiberM bus = void <<< forkAff
  <<< flip runStateT { count: 10 }
  <<< flip runReaderT { bus }
```

❶ We don't need the `Tuple` import any more since we replaced `Tuple Unit State` with `Unit`.

Don't forget to remove the `Data.Tuple` import.

Let' uncomment out the code in `test`:

```
test :: Effect Unit
test = launchAff_ do
  bus <- Bus.make
  let forkFiberM = runFiberM bus
  forkFiberM logger
  forkFiberM $ randomGenerator (_ > 0.5)
  forkFiberM $ randomGenerator (_ < 0.5)
  forkFiberM $ randomGenerator (_ > 0.1)
```

❶ Deleted the `pure unit` since the final call to `forkFiberM` is a valid Expression to terminate our `do`.

Now we have a compiler error in our test code since we've not written `logger` yet. So, go ahead and create a do-nothing version of `logger` remembering to use `pure unit`.

Once you've done that, continue reading.

First, we write the Type Signature for `logger`:

```
logger :: Aff Unit
```

And finally, the do-nothing implementation:

```
logger :: Aff Unit
logger = pure unit
```

Now, the compiler should be pointing to the fact that `randomGenerator` is missing. Write that as a do-nothing Function as well. Don't forget that it takes a Parameter. Then continue on.

Here's its Type Signature:

```
randomGenerator :: (Number -> Boolean) -> Aff Unit ❶
```

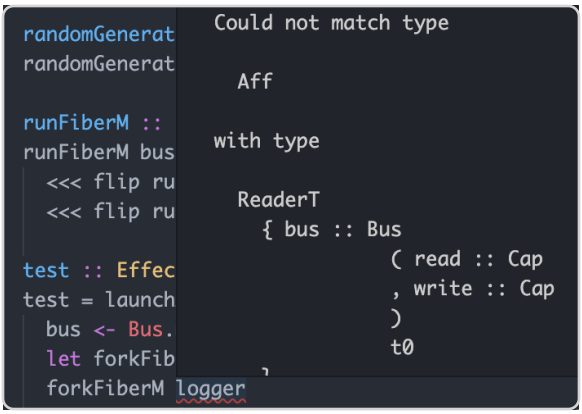
❶ First Parameter is a Predicate.

Now, we'll add the same do-nothing code:

```
randomGenerator :: (Number -> Boolean) -> Aff Unit
randomGenerator pred = pure unit ❶
```

❶ We've put in the Parameter `pred` even though it's not used yet.

But now we have an unexpected compiler error on `logger` in our test code:



Can you see where I went wrong?

I forgot that we're not running in `Aff`. We are running in our Stack, i.e. what we called `FiberM`. So, I've used the wrong `Monad` on `logger`. Now that I think about it, `randomGenerator` is wrong too.

Let's fix my mistakes:

```
logger :: FiberM Unit ❶
logger = pure unit

randomGenerator :: (Number -> Boolean) -> FiberM Unit ❶
randomGenerator pred = pure unit
```

❶ Changed `Aff` to `FiberM` since our `Fibers` run in the `FiberM` Monad, i.e they run in the Monad Stack defined by `FiberM`.

And now this compiles clean sans the warning that we aren't using `pred`. We will soon enough.

Now, what should we write?

Well, we have to write `logger` and `randomGenerator` and once we do that we're done. So, I'm going to suggest you start with `logger` since it's simpler.

Here's the `logger` specification for reference:

```
Subscribers will listen to the `Bus` and log the data they get from the `Bus`
to the console. There will be 1 Subscriber.
```

So, code `logger` before you continue reading on.

We currently have the following:

```
logger :: FiberM Unit
logger = pure unit
```

We need to replace the `pure unit` with real code. We're going to start with a `do` block:

```
logger :: FiberM Unit
logger = do
  ???
```

The next thing we want to do is read from the `Bus`. But how do we get the `Bus`? Remember that it's not passed to us.

We could pass it to this Function, but we're running in a Stack with `ReaderT` and our `Config` has a `Bus` in it:

```
type Config = { bus :: BusRW String }
```

This means we can read it from `ReaderT`. Do you remember how to do this?

To do this, we do the following:

```
import Control.Monad.Reader.Trans (ReaderT, runReaderT, ask) ❶

logger :: FiberM Unit
logger = do
  { bus } <- ask ❷
  ???
```

❶ Added `ask` to the import list.

❷ Read `Config` from `ReaderT`. Since it's a Record, we destructure it into a Variable called `bus`.

Next, we need to read from the `Bus`:

```
logger :: FiberM Unit
logger = do
  { bus } <- ask
  s <- Bus.read bus ❶
  ???
```

❶ This runs in `Aff`!!

But wait. `Bus.read` runs in `Aff`, but we're coding in `FiberM`. That means we need to lift this Function from

the `Aff` Layer in our Stack to the very top. Here's our Stack:

```
type FiberM a = ReaderT Config (StateT State Aff) a
```

This shows us that if we lift `Bus.read` once, it'll move from `Aff` to `StateT`, which means that we need to lift it one more time to get it from the `StateT` Layer to the `ReaderT` Layer:

```
import Control.Monad.Trans.Class (lift) ❶

logger :: FiberM Unit
logger = do
  { bus } <- ask
  s <- lift $ lift $ Bus.read bus
  ???
```

❶ Added import.

That's ugly but good enough for now. We'll come back and clean that up later.

Next, we need to log the `String` that we read from the `Bus`:

```
logger :: FiberM Unit
logger = do
  { bus } <- ask
  s <- lift $ lift $ Bus.read bus
  log $ "Logger: " <> s ❶
  ???
```

❶ Marking this output with `Logger:` to help distinguish it from other data in the log. Since this is the only place we'll write to the log, this is superfluous, but I like doing this as a matter of convention. This way if we change our program to log elsewhere, we'll won't have to come back here to tag its output.

Does `log` require lifting?

To answer that, we have to look at what `log` runs in. Remember that we're using the version of log that runs in `MonadEffect` from `Effect.Class.Console`:

```
log :: ∀ m. MonadEffect m => String -> m Unit
```

`log` runs in a Monad `m` where that Monad has an instance for `MonadEffect`. We know that `Aff` at the bottom of our Stack has a `MonadEffect` instance, but does `ReaderT`?

Go look in `Pursuit` for `ReaderT` and then search under the `Instance` section of `ReaderT` for

`MonadEffect`. Then return here.

It turns out that `ReaderT` has a `MonadEffect` Instance. In fact, all the Monad Transformers do. And because of this, `log` will run in `ReaderT` meaning that it does NOT require any lifting.

Next, we need to loop forever:

```
logger :: FiberM Unit
logger = do
  { bus } <- ask
  s <- lift $ lift $ Bus.read bus
  log $ "Logger: " <> s
  logger
```

And while this will work, the infinite recursion can get lost if our Function is large. Not ideal. But there is a better solution:

```
import Control.Monad.Rec.Class (forever) ❶

logger :: FiberM Unit
logger = forever do ❷
  { bus } <- ask
  s <- lift $ lift $ Bus.read bus
  log $ "Logger: " <> s
```

❶ Need to import `forever`.

❷ Using `forever` at the top of the `do` block makes it immediately clear that we have an infinite recursion.

Now, let's fix the ugly `lift $ lift $` part.

Write a function called `liftAffToFiberM` to lift an `Aff` to `FiberM`. Use the Natural Transformation syntax in the Type Signature, i.e. `~>`, since both are `Monads` and all `Monads` are `Functors`.

Code this and then continue on.

The code for this is very simple but will factor out the ugliness in our code:

```
liftAffToFiberM :: Aff ~> FiberM ❶
liftAffToFiberM = lift <<< lift ❷
```

❶ Use Natural Transformation syntax, which is equivalent to `∀ a. Aff a -> FiberM a`.

❷ Lift twice. With our Stack, we're first lifting the base `Monad`, `Aff`, to `StateT` and then `ReaderT`.

Now, update your `logger` code to make use of `liftAffToFiberM` before reading the solution.

You should now have the following:

```
logger :: FiberM Unit
logger = forever do
  { bus } <- ask
  s <- liftAffToFiberM $ Bus.read bus
  log $ "Logger: " <> s
```

Now we're ready to write the Publisher. But first, we should look again at the specification for Publishers:

```
Publishers will produce Random Numbers delaying 1 second before doing so. Then
they will publish a `String` message onto a `Bus` when their randomly
generated number passes some Predicate.

The Publisher will then countdown a count in the State and continue only if
the count is positive.
```

What would you write first? Would you just dive into `randomGenerator`?

After reading this, I feel like I should first write a delayed version of `random`, one that runs in `Aff` instead of `Effect` by using our `randomAff`.

So, write a Function that does this and call it `delayRandom`. You'll need `Milliseconds` from `Data.Time.Duration`. You should also look up `delay` in `Pursuit`.

Move on only after you've given this a try.

The Type Signature is always a good starting point:

```
delayRandom :: Aff Number
```

Next, we'll write the left-hand side:

```
delayRandom :: Aff Number
delayRandom = ??? ❶
```

❶ No Parameters.

And now we have to think a bit more. We need to first delay for 1 second and then get a random number remembering to use `randomAff`. Seems like we should start a `do` block here, but I'm guessing that we

probably won't need it since we only have to do 2 things, which almost always looks better using a `bind`.

I could be wrong, so we'll write it with `do` to start and improve it afterwards if need be:

```
delayRandom :: Aff Number
delayRandom = do
  ???
```

Next, we delay 1000 millliseconds:

```
import Data.Time.Duration (Milliseconds(..)) ❶
import Effect.Aff (Aff, launchAff_, forkAff, delay) ❷

delayRandom :: Aff Number
delayRandom = do
  delay $ Milliseconds 1000.0 ❸
  ???
```

❶ Import `Milliseconds` Type with its Data Constructors.

❷ Added `delay` to the import list.

❸ The Data Constructor, `Milliseconds`, wants a `Number`, i.e. `1000.0` NOT an `Int`, which would be `1000` without a decimal point.

And now, we can generate our Random Number:

```
delayRandom :: Aff Number
delayRandom = do
  delay $ Milliseconds 1000.0
  randomAff ❶
```

❶ `randomAff` returns `Aff Number` so there's no need for `pure` here.

Just as I suspected, the `do` block can be reduced to a single line of code. Rewrite `delayRandom` by using `>>=` and then we'll see how to remove the Lambda. Try it first, then read on.

Here's `delayRandom` using `bind`:

```
delayRandom :: Aff Number
delayRandom = delay (Milliseconds 1000.0) >>= \_ -> randomAff ❶
```

❶ We've dropped the `$` since we only want the `Milliseconds` to be Parenthesized, not the whole right side of the line.

Now, do you remember how we can remove the Lambda?

If you said `const`, you're right that will work, but I'd like to remove the `\_ ->` and not replace it with anything.

Do you remember how we've done this in the past?

It's okay if not, we haven't done this much. Remember, `applySecond`:

```
applySecond :: ∀ a b f. Apply f => f a -> f b -> f b
```

This takes an `Apply`, which all `Monads` are, and it throws out the computation from the first in favor of the second. That's what we did with our Lambda, we ignored the result of the first in favor of the second.

But how is this helping clean up our code?

Well, the Binary Operator for `applySecond` is `*>` and that helps us make our code cleaner:

```
delayRandom :: Aff Number
delayRandom = delay (Milliseconds 1000.0) *> randomAff
```

This is so much cleaner than all of our other versions. So, this was definitely worth doing.

We need to return to our specification for Publishers:

```
Publishers will produce Random Numbers delaying 1 second before doing so. Then
they will publish a `String` message onto a `Bus` when their randomly
generated number passes some Predicate.

The Publisher will then countdown a count in the State and continue only if
the count is positive.
```

At this point I don't see any other helper Functions that we may need. So, we're ready to write the `randomGenerator`. Start with our do-nothing version:

```
randomGenerator :: (Number -> Boolean) -> FiberM Unit
randomGenerator pred = pure unit
```

Remove the `pure unit`, our stand-in code, and code up this Function based on the above specs before reading any further.

We'll start with a `do` block:

```
randomGenerator :: (Number -> Boolean) -> FiberM Unit
randomGenerator pred = do
  ???
```

Next, we should check to make sure our count isn't `0` or negative. It's possible that the initial State is invalid and we want to make sure to check this as soon as possible.

Otherwise, if we start this `Fiber` with a count of `-1`, we could generate a Random Number and write it to the `Bus` before checking our `count`, which would not be good.

So, let's check up front:

```
import Control.Monad.State.Trans (StateT, runStateT, get) ❶

randomGenerator :: (Number -> Boolean) -> FiberM Unit
randomGenerator pred = do
  { count } <- get ❷
  unless (count <= 0) do ❸
    ???
```

❶ Added `get` to the import list.

❷ We read the `State` and destructure the Record to get access to `count`.

❸ We only want to do something if the count is valid. So, `unless` the `count` is `0` or less , we want to do the `do` block.

Now, we'll get our Random Number using `delayRandom`:

```
randomGenerator :: (Number -> Boolean) -> FiberM Unit
randomGenerator pred = do
  { count } <- get
  unless (count <= 0) do
    n <- delayRandom ❶
    ???
```

❶ This runs in `Aff`.

We'll need to lift `delayRandom`, which we can do using `liftAffToFiberM`:

```
randomGenerator :: (Number -> Boolean) -> FiberM Unit
randomGenerator pred = do
  { count } <- get
  unless (count <= 0) do
    n <- liftAffToFiberM delayRandom
    ???
```

Now we can write it to the `Bus`. But wait. We don't have a `Bus`. We're going to have to get the `Bus` first from the read-only data `ReaderT` provides us:

```
randomGenerator :: (Number -> Boolean) -> FiberM Unit
randomGenerator pred = do
  { count } <- get
  unless (count <= 0) do
    n <- liftAffToFiberM delayRandom
    { bus } <- ask ❶
    ???
```

❶ We're destructuring the `Config` Record to get at the `Bus`.

Now, we can write to it, but if and only if the `pred` returns `true`:

```
randomGenerator :: (Number -> Boolean) -> FiberM Unit
randomGenerator pred = do
  { count } <- get
  unless (count <= 0) do
    n <- liftAffToFiberM delayRandom
    { bus } <- ask
    when (pred n) $ Bus.write (show n) bus ❶
    ???
```

❶ This too runs in `Aff`.

We need to lift `Bus.write`:

```
randomGenerator :: (Number -> Boolean) -> FiberM Unit
randomGenerator pred = do
  { count } <- get
  unless (count <= 0) do
    n <- liftAffToFiberM delayRandom
    { bus } <- ask
    liftAffToFiberM $ when (pred n) $ Bus.write (show n) bus
    ???
```

Now we can recurse. Wait... We don't want to infinitely recurse. So, we'd better decrement our `count` first THEN recurse:

```
import Control.Monad.State.Trans (StateT, runStateT, get, put) ❶

randomGenerator :: (Number -> Boolean) -> FiberM Unit
randomGenerator pred = do
  { count } <- get
  unless (count <= 0) do
    n <- liftAffToFiberM delayRandom
    { bus } <- ask
    liftAffToFiberM $ when (pred n) $ Bus.write (show n) bus
    put { count: count - 1 } ❷
    randomGenerator pred ❸
```

❶ Added `put` to the import list.

❷ Modify the State, i.e. decrement the `count`.

❸ Recurse. It will terminate at the `unless` line once `count` is less than or equal to `0`.

Notice that we used `put` to modify the State. Seems like we could also use `modify_` here. Turns out that we could. Since our `State` only has 1 element, using `put` is equivalent to `modify_` here.

But, we may modify our program later and add another field to our `State` Record. If that happens, we'll regret not using `modify_` in the first place. So, let's change it now:

```
import Control.Monad.State.Trans (StateT, runStateT, get, modify_) ❶

randomGenerator :: (Number -> Boolean) -> FiberM Unit
randomGenerator pred = do
  { count } <- get
  unless (count <= 0) do
    n <- liftAffToFiberM delayRandom
    { bus } <- ask
    liftAffToFiberM $ when (pred n) $ Bus.write (show n) bus
    modify_ _ { count = count - 1 } ❷ ❸ ❹
    randomGenerator pred
```

❶ Remove `put` from our import list and replace it with `modify_`.

❷ Note this syntax. The first Parameter, i.e. `_`, is a wildcard for the `State`.

❸ The second Parameter is the update Record syntax in PureScript. Notice how it uses `=` instead of `:`.

❹ This line of code is equivalent to `modify_ \state -> state { count = count - 1 }`, but is easier to read.

Like I always say, just because it compiles, doesn't mean that we're done. Can you see any improvements we could make here? I see one right away.

Notice that we're using `liftAffToFiberM` in 2 places:

```
randomGenerator :: (Number -> Boolean) -> FiberM Unit
randomGenerator pred = do
  { count } <- get
  unless (count <= 0) do
    n <- liftAffToFiberM delayRandom ❶
    { bus } <- ask
    liftAffToFiberM $ when (pred n) $ Bus.write (show n) bus ❶
    modify_ _ { count = count - 1 }
    randomGenerator pred
```

❶ Using `liftAffToFiberM` in these 2 places.

Also, notice that in between these 2 locations, all we're doing is reading the `bus` from the Reader environment. We could move that line up one and then combine the 2 lines that use `liftAffToFiberM` into a `do` block:

```
randomGenerator :: (Number -> Boolean) -> FiberM Unit
randomGenerator pred = do
  { count } <- get
  unless (count <= 0) do
    { bus } <- ask ❶
    liftAffToFiberM do ❷
      n <- delayRandom ❸
      when (pred n) $ Bus.write (show n) bus ❸
    modify_ _ { count = count - 1 }
    randomGenerator pred
```

❶ Moved this up 1 line.

❷ This `do` block is running in the `Aff` Monad. Each line of the `do` block is an `Aff a`.

❸ These lines are easier to read with `liftAffToFiberM` factored out.

The next code improvement involves improving not the code, but the output that the code produces.

Right now, we're just outputting the numbers when they pass the Predicate. But we're going to have 3 Publishers. We'll have no idea which Publisher found a Random Number to its liking.

Let's add another Parameter that we will print along with a message:

```
randomGenerator :: String -> (Number -> Boolean) -> FiberM Unit ❶
randomGenerator valueType pred = do ❷
  { count } <- get
  unless (count <= 0) do
    { bus } <- ask
    liftAffToFiberM do
      n <- delayRandom
      when (pred n) $ flip Bus.write bus ❸
        $ "Found a value that is "
          <> valueType <> " (" <> show n <> ")" ❹
    modify_ _ { count = count - 1 }
    randomGenerator valueType pred ❺
```

❶ Added `String` Type as the first Type.

❷ Added our new Parameter called `valueType`.

❸ Flipping `Bus.write` lets us write this code more cleanly with all of the text concatenations happening at the end of our line.

❹ An example of what `valueType` might be is `"less than 0.1"`.

❺ Had to add `valueType` as a Parameter to the recursive call.

Now, we must add this Parameter to `randomGenerator` calls in our test code:

```
test :: Effect Unit
test = launchAff_ do
  bus <- Bus.make
  let forkFiberM = runFiberM bus
  forkFiberM logger
  forkFiberM $ randomGenerator "greater than 0.5" (_ > 0.5) ❶
  forkFiberM $ randomGenerator "less than 0.5" (_ < 0.5) ❶
  forkFiberM $ randomGenerator "greater than 0.1" (_ > 0.1) ❶
```

❶ Added `valueType` Parameter to improve logging output.

Now, after all of this, we should be able to run this. Since it's random numbers, it'll be hard to know what to expect, so let's test it by changing `randomAff` to `constant` in `delayRandom`:

```
constant :: Aff Number ❶
constant = pure 42.0

delayRandom :: Aff Number
delayRandom = delay (Milliseconds 1000.0) *> constant ❷
-- delayRandom = delay (Milliseconds 1000.0) *> randomAff ❸
```

❶ Will always be greater than `0.1` and `0.5`, but will never be less than `0.5`.

❷ Call `constant` instead of `randomAff`.

❸ Comment out real code for now.

Run this using:

```
npx spago run
```

When you do we get:

```
[error] Some of your project files import modules from packages that are not
in the direct dependencies of your project.
To fix this error add the following packages to the list of dependencies in
your config:
- tailrec
```

Fix this and now when we run this, we should get the following 2 lines repeated 10 times, i.e. 20 lines total:

```
Logger: Found a value that is greater than 0.5 (42.0)
Logger: Found a value that is greater than 0.1 (42.0)
```

If you're too lazy to count, like me, you can copy the output of your program and paste it into a new file and let the editor count the lines for you.

This only tested 2 our of the 3 Publishers's Predicate Functions since our number was `42.0`. Let's make it `0.0` so we can test the Publisher that tests for values less than `0.1`:

```
constant :: Aff Number
constant = pure 0.0
```

Now when run, we should get the following 10 times:

```
Logger: Found a value that is less than 0.5 (0.0)
```

Go ahead and delete or comment out `constant` and restore `delayRandom` back to its original code:

```
delayRandom :: Aff Number
delayRandom = delay (Milliseconds 1000.0) *> randomAff
```

Now, when run, you should get something like:

```
Logger: Found a value that is greater than 0.5 (0.938568738460007)
Logger: Found a value that is greater than 0.1 (0.3812324304953467)
Logger: Found a value that is greater than 0.1 (0.5058969659785102)
Logger: Found a value that is less than 0.5 (0.17046551999308068)
Logger: Found a value that is greater than 0.1 (0.42600641653043736)
Logger: Found a value that is greater than 0.5 (0.5824010610612658)
Logger: Found a value that is less than 0.5 (0.4375844856369162)
Logger: Found a value that is greater than 0.1 (0.5148930040569029)
Logger: Found a value that is less than 0.5 (0.3261672886687357)
Logger: Found a value that is greater than 0.1 (0.3168855112054627)
Logger: Found a value that is greater than 0.1 (0.7445062999508418)
Logger: Found a value that is greater than 0.1 (0.15464478518018088)
```

Your output will clearly vary since we're using Random Numbers. If we were using a Random Number Generator that we could Seed with a Value, then we could make it run predictably.

There's a package called `purescript-quickcheck` which has a Pseudo-Random Generator. It's API is not

intuitive at all. Later in the book, we'll use it to generate a Salt for a Password Hash. This library could, however, be used to make this program produce predictable output by providing it the same `Seed`.

By the way, there is a package on `Pursuit` called `purescript-pseudo-random`. It has a much better API for generating Random Values, but it's not included in the `spago` package sets. These package sets are curated libraries that are kept up to date with compiler and other ecosystem changes.

23.4. Javascript Runtime with AVars and Busses

Looking back on the program we just wrote, we can see that it does terminate. But when and why? Our `logger` Function runs forever using `forever`. Shouldn't it run forever?

To answer that question, let's look at the `Fibers` we've started. First, we'll look at our Publishers:

```
randomGenerator :: String -> (Number -> Boolean) -> FiberM Unit
randomGenerator valueType pred = do
  { count } <- get
  unless (count <= 0) do
    { bus } <- ask
    liftAffToFiberM do
      n <- delayRandom
      when (pred n) $ flip Bus.write bus
        $ "Found a value that is "
          <> valueType <> " (" <> show n <> ")"
    modify_ _ { count = count - 1 }
    randomGenerator valueType pred
```

Notice how we conditionally recurse. Once `count` reaches `0`, we exit the `Fiber` by simply returning from this Function.

Now, let's look at the Subscriber:

```
logger :: FiberM Unit
logger = forever do
  { bus } <- ask
  liftAffToFiberM do
    s <- Bus.read bus
    log $ "Logger: " <> s
```

This `Fiber` never exits. So, why does our program exit when this `Fiber` never does?

It turns out that the call to `Bus.read` is why. Let me explain.

Under the covers, `Bus` uses `AVar`. Here's the full definition of `Bus`:

```
data Bus (r :: Row Type) a = Bus (AVar a) (AVar (List (AVar a)))
```

We don't see the Data Constructor in the `Pursuit` documentation because it's not exported from this module. Instead, we use the Smart Constructor, `make`.

But from this internal implementation detail, we can see that `Busses` rely heavily on `AVars`. And it's the way `AVars` work that's the real reason our program exits.

Take the following program:

```
main :: Effect Unit
main = launchAff_ do
  avar <- AVar.empty
  log "before"
  x <- AVar.take avar
  log "after"
```

What do think this program will output?

Would you be surprise to learn it NEVER outputs `after` and that it exits? I sure was.

Talking with the one of the library developers, he explained to me that the callback for the `take` call is kept in an internal queue in the library.

It's not a Javascript callback and so there's nothing pending when the library yields control back to Javascript Main Event Loop.

When there are no outstanding Javascript callbacks in Node and the main thread has yielded control, Node will exit the program.

Is this a problem when running in a Browser? Probably not since your code never really exits unless it crashes.

But, our Publisher/Subscriber program is running in Node and, in that case, there nothing more for the Event Loop to process and there's no pending Javascript callbacks, so Node just exits.

Since `Busses` use `AVars`, which exhibit this behavior, then `Busses` too exhibit the same behavior. This is why `logger` isn't enough to keep our program alive and once the Publishers exit, so does our program.

You may be wondering why the Publishers don't suffer from the same issue as the Subscriber. To see why, let's look back at the Publisher code:

```
delayRandom :: Aff Number
delayRandom = delay (Milliseconds 1000.0) *> randomAff

randomGenerator :: String -> (Number -> Boolean) -> FiberM Unit
randomGenerator valueType pred = do
  { count } <- get
  unless (count <= 0) do
    { bus } <- ask
    liftAffToFiberM do
      n <- delayRandom ❶
      when (pred n) $ flip Bus.write bus
        $ "Found a value that is "
          <> valueType <> " (" <> show n <> ")"
    modify_ _ { count = count - 1 }
    randomGenerator valueType pred
```

❶ This schedules a callback through Javascript, which means our code will not exit before this call returns.

Notice the call to `delayRandom`. In `delayRandom`, we call `delay`, which is ultimately calling Javascript's `setTimeout`. This keeps our program alive since we have an outstanding callback that Javascript knows about.

Once we return from `delayRandom`, we recurse and either exit or call `delayRandom` again, which keeps our program alive.

In the case of `logger`, there was a pending callback that only the `AVar` library knows about. Javascript is none the wiser. So, when all of the Publishers exit, the Javascript Runtime sees that there are no outstanding requests and it promptly exits.

I'm not a huge fan of PureScript on the backend for these kinds of quirky issues. I'd almost always write my backend in Haskell given the chance. But if you're going to write PureScript and run it under Node, you'll need to keep these sort of issues in mind.

Chapter 24. JSON and Ajax

In many Browser applications that communicate with a backend server, JSON is used as the format for calling APIs. This is usually done via an HTTP POST and that's what we're going to do when we develop our application in the upcoming Chapters.

There is a canonical library for doing HTTP requests from PureScript called `purescript-affjax`. And there are 2 major ways for JSON encoding and decoding.

The first way is using PureScript's Generic Representation for PureScript Types that map directly to Javascript Types. This is the simplest library to use and is built into the `Foreign.Generic` modules. It's simplicity is both its strength and it shortcomings, e.g. you cannot encode an `Either` or a `Tuple`.

The other approach is a far more flexible library called ***Argonaut*** and can be found in the package `purescript-argonaut`. Decoding JSON will be very familiar to us since we've written out `Parser`. We can write our decoders using Applicative and Monadic code that's very similar to how we wrote Applicative and Monadic parsers.

24.1. Generic JSON Encoding and Decoding

Generic JSON encoding and decoding means that we don't have to write code to explicitly encode and decode our Types. While this sounds too good to be true, it actually can work in many instances. But it does have its limitations.

You can only encode the following Types:

```
Void ❶
Unit
Foreign ❷
String
Char
Boolean
Number
Identity a
Array a
Maybe a
Object v
Record r
```

❶ Actually can never be created.

❷ `Foreign` represents a JSON blob, i.e. some JSON data that we don't want to encode or decode explicitly.

Notice how `Either` isn't there. So if your Type has `Either`, you cannot use Generic JSON encoding or decoding. That's were `Argonaut` comes in.

You can, however, create a Type that's Isomorphic to `Either` and it'll work as long as its Types are in the above list. Another way of accomplishing this is using `newtype` since we can derive all of the underlying behavior normally found in `Either`.

Also, if you need to have complete control over the construction of the JSON object that represents your Type, e.g. you're interfacing to a Legacy system or a system that you don't control, you most likely will NOT be able to use Generic JSON encoding and decoding.

This is because Generic JSON encoding uses a very specific JSON structure that probably won't match your Legacy System's API.

Once again, this is where `Argonout` comes to the rescue. But for all the other cases, this library can greatly eliminate boilerplate code.

And in the cases where the library will cover 90% of the Types that you'll need to encode or decode, you can use `Argonaut` for the other 10%. So, your burden is still greatly reduced.

The Generic JSON encoding and decoding can be found in the following modules:

- `Foreign`
- `Foreign.Generic`
- `Foreign.Generic.Class`

Let's look at each of these modules one at a time staring with `Foreign`.

24.1.1. `Foreign` Module

The explanation for what the `Foreign` module contains is:

```
This module defines types and functions for working with foreign data.
```

But what is ***foreign data***?

Since PureScript works with Javascript, that is what is meant by foreign data.

There's only one thing from this module that we're going to be immediately concerned with and that's the Type Alias, `F`. It represents one or more errors when working with Javascript data.

Here's its definition:

```
type F = Except MultipleErrors
```

Remember that `Except` is built with `ExceptT` and the `Identity` Monad. There is a Function called `runExcept` that will run this mini-Stack. It will `unwrap` the value in the `Identity` Context for us.

Here is the Type Alias definition for `MultipleErrors`:

```
type MultipleErrors = NonEmptyList ForeignError
```

This is a list of at least 1 `ForeignError`. But what is a `ForeignError`?

Here's it's definition:

```
data ForeignError
  = ForeignError String
  | TypeMismatch String String
  | ErrorAtIndex Int ForeignError
  | ErrorAtProperty String ForeignError
```

These are errors that you can expect to get from JSON decoding. `ForeignError` has a `Show` Instance allowing us to write JSON decoding errors to a log.

24.1.2. `Foreign.Generic` Module

This module contains the Generic JSON encoders and decoders that allow us to delegate the encoding and decoding of our Types without having to write a single line of code.

It also contains the Function for encoding Types into `Strings`, viz. `encodeJSON`, and for decoding `Strings` into Types, viz. `decodeJSON`.

Please keep in mind, if you search `Pursuit` for these Functions, you will find 2 versions. The ones we're concerned with here have JSON in all UPPERCASE.

The other versions are `encodeJson` and `decodeJson`. These are part of the `Argonaut` library which we will delve into shortly.

24.1.3. `Foreign.Generic.Class` Module

There are 2 Typeclasses of major interest from this module, viz. `Encode` and `Decode`. If we need to make a Type that we want to have encoded or decoded into JSON for us, we'll need to create Instances of these Typeclasses.

The beauty of Generics is that it allows us to delegate to `genericEncode` and `genericDecode` from the `Foreign.Generic` module.

Let's imagine we have a Type:

```
data Portion = Bit | Chunk | Part
```

Next, we can derive `Generic` for our Type:

```
data Portion = Bit | Chunk | Part
derive instance genericPortion :: Generic Portion _
```

Now, we can leverage the Generic JSON encode and decoder for our Type to create an `Encode` and `Decode` Instance:

```
import Foreign.Generic (genericEncode, genericDecode, defaultOptions)

instance encodePortion :: Encode Portion where
  encode = genericEncode defaultOptions ❶

instance decodePortion :: Decode Portion where
  decode = genericDecode defaultOptions ❶
```

❶ We're using the default `Options`. More on this later.

Let's encode our Sum Type, `Portion`:

```
test :: Effect Unit
test = log $ encodeJSON Part ❶
```

❶ Prints `{"tag":"Part"}`.

Notice the key, `tag`. This tagging let's the decoder know which Data Constructor to call.

There is another important element in this module and that's the `Options` Type and `defaultOptions`. You can see that we've used `defaultOptions` in `encodePortion` and `decodePortion`.

Let's look at them in detail. First, let's look at the `Options` Type:

```
type Options =
  { sumEncoding :: SumEncoding ❶
  , unwrapSingleConstructors :: Boolean ❷
  , unwrapSingleArguments :: Boolean ❸
  , fieldTransform :: String -> String ❹
  }
```

❶ This controls the way Sum Types are encoded. We delve into this later.

❷ If `true` then a Type that has a single Data Constructor will not be tagged.

❸ If `false`, then a Type that has a single Type Parameter for it Data Constructor will have it's Parameters

returned as a single element instead of an `Array` in the `contents`, e.g. `{ contents: "abc" }` instead of `{ contents: [ "abc" ] }`. Here `"abc"` is a single Data Constructor Parameter of Type `String`, e.g. as we'd find in `newtype MyType = MyDataConstructor String`.

❹ This is a Function that let's you transform field names.

We'll look at examples of all of these options soon.

Here's the implementation for `defaultOptions` including the comments, which are formatted to automatically produce `Pursuit` documentation:

```
-- |
-- | - Represent sum types as records with `tag` and `contents` fields
-- | - Unwrap single arguments
-- | - Don't unwrap single constructors
-- | - Use the constructor names as-is
-- | - Use the field names as-is
defaultOptions :: Options
defaultOptions =
  { sumEncoding:
      TaggedObject
        { tagFieldName: "tag"
        , contentsFieldName: "contents"
        , constructorTagTransform: identity
        }
  , unwrapSingleConstructors: false
  , unwrapSingleArguments: true
  , fieldTransform: identity
  }
```

Let's now look at how changing these options affects the output of encoding. Let's start with an example Type:

```
newtype Single a = Single a
derive instance genericSingle :: Generic (Single a) _

instance decodeSingle :: Decode a => Decode (Single a) where
  decode = genericDecode defaultOptions ❶

instance encodeSingle :: Encode a => Encode (Single a) where
  encode = genericEncode defaultOptions ❶
```

❶ Using the default options for encoding and decoding.

Here's a test program to see what we get by default:

```
test :: Effect Unit
test = log $ encodeJSON $ Single true ❶
```

❶ Prints `{"contents":true,"tag":"Single"}`

Notice the use of `tag`. This is common and is also the way Haskell's Aeson library does it. Fun Fact: Aeson was Jason's father in Greek Mythology.

By why ***tag***?

That's because Sum Types are also called Tagged Unions. Here tags are the Data Constructor names. It's these tags that help the compiler do Pattern Matching.

Now, let's change `unwrapSingleConstructors` from the default of `false` to `true`:

```
instance decodeSingle :: Decode a => Decode (Single a) where
  decode = genericDecode defaultOptions
    { unwrapSingleConstructors = true } ❶

instance encodeSingle :: Encode a => Encode (Single a) where
  encode = genericEncode defaultOptions
    { unwrapSingleConstructors = true } ❶

test :: Effect Unit
test = log $ encodeJSON $ Single true ❷
```

❶ This is using the Record Update syntax. Notice the use of `=` instead of `:`. So, we want all the defaults but we want to override `unwrapSingleConstructors`.

❷ Prints `true`.

This program simply prints `true` since `Single` has a Single Data Constructor.

Next, we'll change `unwrapSingleArguments` from `true` to `false`:

```
instance decodeSingle :: Decode a => Decode (Single a) where
  decode = genericDecode defaultOptions { unwrapSingleArguments = false }

instance encodeSingle :: Encode a => Encode (Single a) where
  encode = genericEncode defaultOptions { unwrapSingleArguments = false }

test :: Effect Unit
test = log $ encodeJSON $ Single true ❶
```

❶ Prints `{"contents":[true],"tag":"Single"}`. Notice that the single Type Parameter to the Data Constructor is in an `Array`. This would ALWAYS be the case if any of `Single's` Data Constructors had 2 or more Type Parameters.

And finally, we're going to override `fieldTransform` with a different Type called `Mono`:

```
newtype MonoType a = MonoType { one :: a, two :: a, three :: a }
derive instance genericMonoType :: Generic (MonoType a) _

instance decodeMonoType :: Decode a => Decode (MonoType a) where
  decode = genericDecode defaultOptions { fieldTransform =
    case _ of
      "won" -> "one" ❶
      name -> name }

instance encodeMonoType :: Encode a => Encode (MonoType a) where
  encode = genericEncode defaultOptions { fieldTransform =
    case _ of
      "one" -> "won" ❷
      name -> name }

test :: Effect Unit
test = log $ encodeJSON $ MonoType { one: 1, two: 2, three: 3 } ❸
```

❶ Rename `won` to `one` when decoding JSON. Notice how this is the opposite mapping as encoding.

❷ Rename `one` to `won` when encoding JSON. Notice how this is the opposite mapping as decoding.

❸ Prints `{"contents":{"two":2,"three":3,"won":1},"tag":"MonoType"}`.

There is one final option that we haven't explored and that's `SumEncoding`. Here's its definition:

```
data SumEncoding
  = TaggedObject
    { tagFieldName :: String ❶
    , contentsFieldName :: String ❷
    , constructorTagTransform :: String -> String ❸
    }
```

❶ This is the name of the tag field name. In `defaultOptions` (see above), it's set to `"tag"`.

❷ This is the contents field name. In `defaultOptions` (see above), it's set to `"contents"`.

❸ This is a transformer Function to change Data Constructor tag names, which is defaulted to `identity`.

24.2. `Argonaut` JSON Encoding and Decoding

The `Argonaut` library give you complete control over the encoding and decoding process. If you remember what we did when we wrote our `Parser`, this is a far simpler, yet similar process.

Let's create a simple Type and see how we'd manually write encoders and decoders.

Here's a simple Type for Desserts:

```
data Dessert = IceCream | Cake | Pie
```

Now we'll want to print this Type out for testing purposes:

```
data Dessert = IceCream | Cake | Pie
derive instance genericDessert :: Generic Dessert _

instance showDessert :: Show Dessert where
  show = genericShow
```

Now we need to decide how this Type is to be represented in JSON. With the `Foreign.Generic` library, it would be represented as:

```
{"tag":"IceCream"}
{"tag":"Cake"}
{"tag":"Pie"}
```

The `tag` would be the Data Constructor name.

But we would like this to be a `String`. We know we can accomplish this by overriding the `defaultOptions`

in the Generic JSON library. Now, we're going to see how to accomplish this in Argonaut.

The reason we want Dessert to encode as a String is so that if we have the following Types:

```
data MainDish = Steak | Chicken

data SideDish = Rice | Asparagus

type Meal =
  { main :: MainDish
  , side :: SideDish
  , dessert :: Dessert
  }

type Meals = { meals :: Array Meal }
```

then Meals would be encoded like:

```
{
  "meals": [
    {
      "main": "Steak",
      "side": "Asparagus",
      "dessert": "Ice Cream"
    },
    {
      "main": "Chicken",
      "side": "Rice",
      "dessert": "Pie"
    }
  ]
}
```

Notice that we also want SideDish to encode to a String.

So we're going to code the decoder for Dessert one line at a time starting with the instance definition. But first here's the Typeclass:

```
class DecodeJson a where
  decodeJson :: Json -> Either JsonDecodeError a
```

And now we can write the instance definition:

```
import Data.Argonaut.Decode (class DecodeJson) ❶

instance decodeJsonDessert :: DecodeJson Dessert where
```

❶ Must include the Typeclass.

Now we can code the left-hand side:

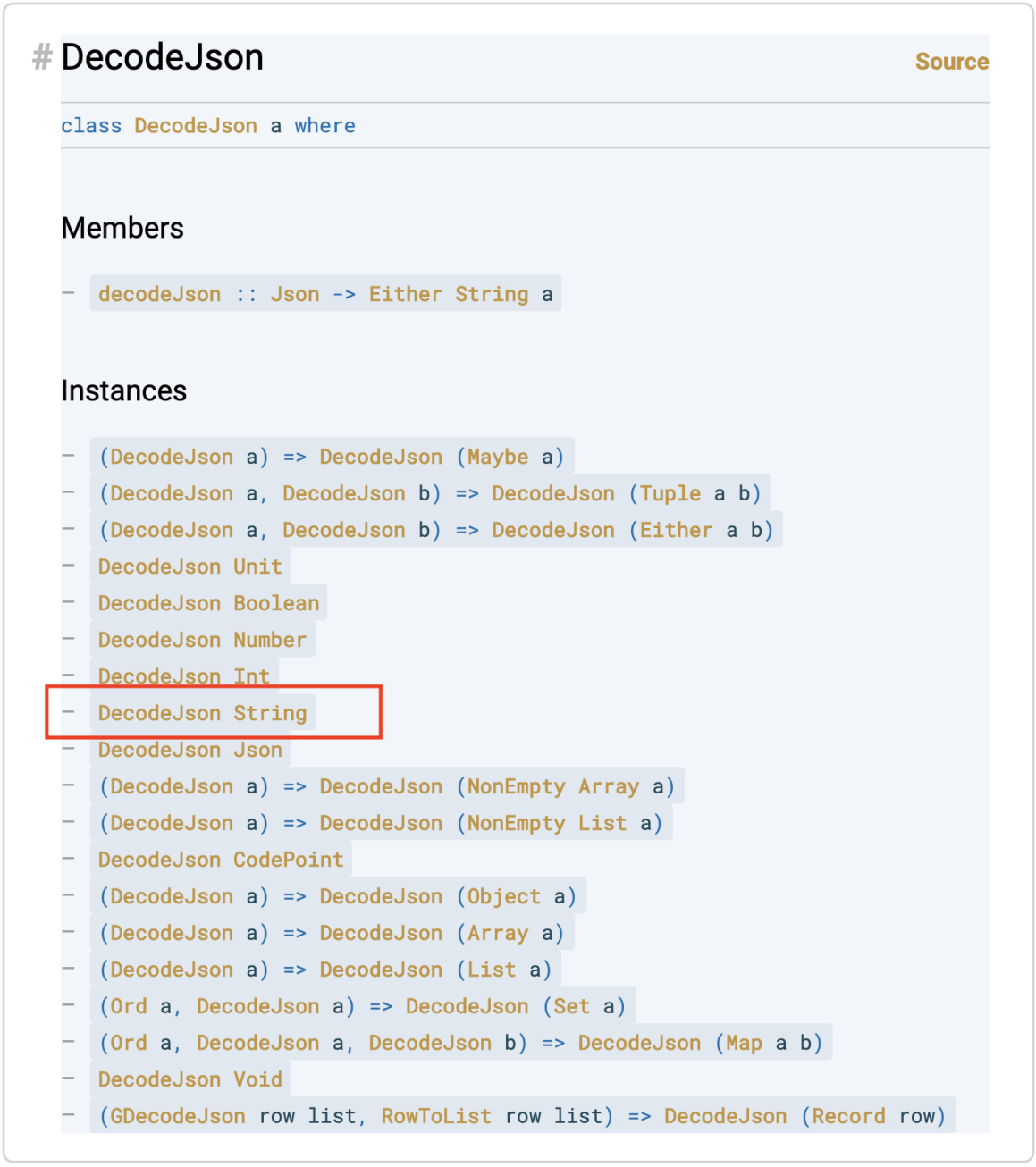
```
instance decodeJsonDessert :: DecodeJson Dessert where
  decodeJson json = ???
```

Since we're working in the `DecodeJson` Monad, we'll add a `do` block:

```
instance decodeJsonDessert :: DecodeJson Dessert where
  decodeJson json = do
    ???
```

Now we need a way to decode the JSON `String` that we're passed. Go look up `DecodeJson` in `Pursuit` and see if there's an instance for `String`. Then return here.

It turns out that there is such an instance:



DecodeJson Source

```
class DecodeJson a where
```

Members

- `decodeJson :: Json -> Either String a`

Instances

- `(DecodeJson a) => DecodeJson (Maybe a)`
- `(DecodeJson a, DecodeJson b) => DecodeJson (Tuple a b)`
- `(DecodeJson a, DecodeJson b) => DecodeJson (Either a b)`
- `DecodeJson Unit`
- `DecodeJson Boolean`
- `DecodeJson Number`
- `DecodeJson Int`
- `DecodeJson String`
- `DecodeJson Json`
- `(DecodeJson a) => DecodeJson (NonEmpty Array a)`
- `(DecodeJson a) => DecodeJson (NonEmpty List a)`
- `DecodeJson CodePoint`
- `(DecodeJson a) => DecodeJson (Object a)`
- `(DecodeJson a) => DecodeJson (Array a)`
- `(DecodeJson a) => DecodeJson (List a)`
- `(Ord a, DecodeJson a) => DecodeJson (Set a)`
- `(Ord a, DecodeJson a, DecodeJson b) => DecodeJson (Map a b)`
- `DecodeJson Void`
- `(GDecodeJson row list, RowToList row list) => DecodeJson (Record row)`

Let's leverage that:

```
instance decodeJsonDessert :: DecodeJson Dessert where
  decodeJson json = do
    s <- decodeJson json ❶
    ???
```

❶ This will ULTIMATELY call `String's` instance and place the `String` in `s`. I say ULTIMATELY because we haven't give the compiler enough information to infer that it's a `String` yet, but we will soon enough.

Next, we should check the value of the `String` and return the appropriate `Dessert`:

```
instance decodeJsonDessert :: DecodeJson Dessert where
  decodeJson json = do
    s <- decodeJson json
    case s of
      ???
```

Let's now make a decision that `"Ice Cream"` will represent `IceCream`:

```
instance decodeJsonDessert :: DecodeJson Dessert where
  decodeJson json = do
    s <- decodeJson json
    case s of ❶
      "Ice Cream" -> ???
```

❶ It's at this point that the compiler can infer that `s` is a `String` since we're comparing it to `"Ice Cream"`.

We want to return `IceCream` here but is that what `decodeJson` wants? We should check:

```
class DecodeJson a where
  decodeJson :: Json -> Either JsonDecodeError a
```

Looks like `a` is wrapped up in an `Either`, which is our `Dessert` in this case. So let's just use `pure` to do that:

```
instance decodeJsonDessert :: DecodeJson Dessert where
  decodeJson json = do
    s <- decodeJson json
    case s of
      "Ice Cream" -> pure IceCream
      ???
```

Next, we need to handle the other cases:

```
instance decodeJsonDessert :: DecodeJson Dessert where
  decodeJson json = do
    s <- decodeJson json
    case s of
      "Ice Cream" -> pure IceCream
      "Cake" -> pure Cake
      "Pie" -> pure Pie
      ???
```

That's all of the VALID cases. Now, we have to handle the case where we received a bad Value.

To do this, we need to return a failure, i.e. `Left`, of a `JsonDecodeError`:

```
data JsonDecodeError ❶
  = TypeMismatch String
  | UnexpectedValue Json
  | AtIndex Int JsonDecodeError
  | AtKey String JsonDecodeError
  | Named String JsonDecodeError
  | MissingValue
```

❶ From the source code of `Data.Argonaut.Decode.Error`.

We'll use `UnexpectedValue`:

```
instance decodeJsonDessert :: DecodeJson Dessert where
  decodeJson json = do
    s <- decodeJson json
    case s of
      "Ice Cream" -> pure IceCream
      "Cake" -> pure Cake
      "Pie" -> pure Pie
      _ -> Left $ UnexpectedValue json
```

Now, we're ready to write the encoder. This time we're getting a `Dessert` and turning it into a `String`.

Let's look at the Typeclass definition before we try to write an instance for it:

```
class EncodeJson a where
  encodeJson :: a -> Json
```

That looks simple enough. Let's get started with the instance definition:

```
import Data.Argonaut.Encode (class EncodeJson) ❶

instance encodeJsonDessert :: EncodeJson Dessert where
```

❶ Import the Typeclass.

Next, we handle each case using Pattern Matching:

```
instance encodeJsonDessert :: EncodeJson Dessert where
  encodeJson IceCream = ???
```

We need to return a Value of Type `Json`. At this point, you may be wondering what is a `Json`. Let's look at its definition from `Pursuit`:

```
data Json :: Type
```

Well, that's not helpful at all. All it tells us is that it's a `Type`. Let's look at its implementation by clicking on `Source` in `Pursuit` and then searching the code for it:

```
foreign import data Json :: Type
```

That's even less helpful. What is this even?

Well, this is a way to create a Type of a specific Kind. In this case, the Kind is `Type`.

This seems to be what I call a Marker Type. If I had to guess, I'd suspect that it's probably used to identify the return Type for the Javascript Functions that are used by this library.

You'll learn more if you follow along while we investigate this. Go to `Pursuit` now and continue reading.

Let's see by investigating `encodeJson` for `String` by searching for `EncodeJson` in `Pursuit` and then clicking on `Source` and then searching the code for `EncodeJson String`.

You should find:

```
instance encodeJsonJString :: EncodeJson String where
  encodeJson = fromString
```

Okay, now we have to look at `fromString`. Search on `Pursuit` for `fromString` making sure that it's the

one from Argonaut and then click on Source.

You should see:

```
foreign import fromString :: String -> Json
```

Aha! It's a Javascript function. We haven't talked about FFIs (Foreign Function Interfaces) yet, but we can find this by looking at the name of the module where fromString is defined.

To do that scroll to the top of the source in GitHub.

You should see that it's in purescript-argonaut-core/src/Data/Argonaut/Core.purs.

That means it's in the file Core.purs. If you click on the Directory where Core.purs resides, which, in this case, is Argonaut, it'll gives you all the files that are in the same directory as Core.purs. Do that now.

What you're looking for is a file with the SAME name as Core.purs but with a js extension. And when you find it, click on it and search the Javascript code for fromString.

You should find:

```
exports.fromString = id;
```

Okay, where is id? Looking around we find it right at the top of the file:

```
function id(x) {
  return x;
}
```

This is just identity but written in Javascript.

So a String has the exact same low-level representation as it does as a Json.

Basically, encodeJson on a String only changes its Type. Json is for Type safety when calling Javascript code to make sure that it can safely treat the Values passed to it as a JSON element.

For example, notice the other from Functions in that same file:

```
exports.fromBoolean = id;
exports.fromNumber = id;
exports.fromString = id;
exports.fromArray = id;
exports.fromObject = id;
```

All of these JSON primitives, e.g. `Boolean`, `Number`, `Object`, `Array`, etc., have the same representation as a `Json`.

Let's look at `Boolean's encodeJson` implementations:

```
instance encodeJsonJBoolean :: EncodeJson Boolean where
  encodeJson = fromBoolean
```

We can see that encoding a `Boolean` doesn't really change it's form, but it does change its Type. We saw this with `unsafeCoerse` when we were looking at `undefined`.

In this case, its not unsafe. In fact, it's here to make it safe.

So, we can look at `Json` as a Type that lets us know that the data is represented as a Javascript JSON element. We can look at `stringify` to see that it expects a `Json`:

```
stringify :: Json -> String
```

And if we look at the implementation we find:

```
foreign import stringify :: Json -> String
```

We can see that this definition is in `Core.purs`, which means we'll have to look back at `Core.js` for the real implementation:

```
exports.stringify = function (j) {
  return JSON.stringify(j);
};
```

And we can see why `stringify` wants to make sure that it has a valid JSON element. We can safely pass a `String` to this but ONLY after we've passed it through `encodeJson`, which does nothing other than change it's Type.

And since there are only a handful of these coercing encoding Functions, viz. all of the JSON elements, we

can ensure that when `stringify` and similar Functions that expect `Json` are called, that we aren't passing invalid things to it, e.g. a Function.

Now, that we've answered that question, let's return to the encoder for `Dessert`:

```
instance encodeJsonDessert :: EncodeJson Dessert where
  encodeJson IceCream = ???
```

We want to encode `IceCream` as `"Ice Cream"` so we might be tempted to code:

```
instance encodeJsonDessert :: EncodeJson Dessert where
  encodeJson IceCream = "Ice Cream"
  ???
```

But as we just learned, a `String` is not a `Json`, but it could be if we called its `encodeJson` instance:

```
instance encodeJsonDessert :: EncodeJson Dessert where
  encodeJson IceCream = encodeJson "Ice Cream"
  ???
```

And now we can just write the other 2 cases:

```
instance encodeJsonDessert :: EncodeJson Dessert where
  encodeJson IceCream = encodeJson "Ice Cream"
  encodeJson Cake = encodeJson "Cake"
  encodeJson Pie = encodeJson "Pie"
```

And so here's the decoder for contrast:

```
instance decodeJsonDessert :: DecodeJson Dessert where
  decodeJson json = do
    s <- decodeJson json
    case s of
      "Ice Cream" -> pure IceCream
      "Cake" -> pure Cake
      "Pie" -> pure Pie
      _ -> Left $ UnexpectedValue json
```

It's important to look at the decoder and the encoder to make sure that they will be inverse functions of each other minus the fact that the decoder can fail with an `Either`.

All encoders and decoders should follow the following law:

```
decodeJson <<< encodeJson = pure
```

Remember, `pure` is Identity in the Kleisli Category. Here the `Monad` is `Either` since `decodeJson` can fail. In this case, it cannot since `encodeJson` is called first, assuming we have written a proper decoder.

Look over our encoder and decoder to make sure we can roundtrip data through them and get back what we started with.

The following code is a test, albeit incomplete, of roundtripping:

```
log $ show $ (decodeJson $ encodeJson Pie :: Either _ Dessert) ❶ ❷
```

❶ Should print `(Right Pie)`, i.e. equivalent to `pure Pie`.

❷ We have to add an explicit Type definition to let the compiler know what Type `decodeJson` should produce. This is because it cannot figure that out since it's just give some `Json` Value, which could be anything.

Now, we should write an encoder and decoder for `Meal`. Here's its definition again:

```
data MainDish = Steak | Chicken

data SideDish = Rice | Asparagus

type Meal =
  { main :: MainDish
  , side :: SideDish
  , dessert :: Dessert
  }
```

But before we can do that, we must have encoders for both `MainDish` and `SideDish`. These are nearly identical to `Dessert`, so I'm just going to give you them complete:

```
instance encodeJsonMainDish :: EncodeJson MainDish where
  encodeJson Steak = encodeJson "Steak"
  encodeJson Chicken = encodeJson "Chicken"

instance decodeJsonMainDish :: DecodeJson MainDish where
  decodeJson json = do
    s <- decodeJson json
    case s of
      "Steak" -> pure Steak
      "Chicken" -> pure Chicken
      _ -> Left $ UnexpectedValue json

instance encodeJsonSideDish :: EncodeJson SideDish where
  encodeJson Asparagus = encodeJson "Asparagus"
  encodeJson Rice = encodeJson "Rice"

instance decodeJsonSideDish :: DecodeJson SideDish where
  decodeJson json = do
    s <- decodeJson json
    case s of
      "Asparagus" -> pure Asparagus
      "Rice" -> pure Rice
      _ -> Left $ UnexpectedValue json
```

Make sure you take a second to look them over to see how they're nearly identical to what we wrote for `Dessert`.

With that out of the way, we can turn our attention to the encoder for `Meal`. But wait. Meal isn't a Type. It's a Type Alias for a Record.

It turns out that `Record` already has an `encodeJson` and a `decodeJson` instance. So we don't have to write them.

But, what if `Meal` was a proper Type:

```
newtype Meal = Meal ❶
  { main :: MainDish
  , side :: SideDish
  , dessert :: Dessert
  }
```

❶ Using `newtype` since there's only 1 Data Constructor and it has only 1 Parameter, i.e. the Record.

Now, we have to manually write an encoder and decoder.

At this point, you might be thinking that you'd like all Records to be Type Aliases instead of a Type. Well, that can work in many instances and as we've just seen that it's saved us a bunch of work. But there are things that we cannot do with Type Aliases.

For example, we cannot make Typeclass Instances for them. Records already have `Eq`, `Ord`, `Show`, `EncodeJson`, and `DecodeJson` to name a few of the salient ones.

But what if we have the following Type:

```
type ShouldBeAType =
  { x :: Int
  , f :: Int -> Int
  }
```

And now you want to compare 2 of these to see if they are equal or if one is greater than the other. You'd get a compiler error because there's neither an `Eq` nor an `Ord` Instance for Functions.

We would have to make our Record a Type:

```
newtype ATypeNow = ATypeNow
  { x :: Int
  , f :: Int -> Int
  }
```

And then provide a custom `Eq` Instance that IGNORES the Function, `f`:

```
instance eqATypeNow :: Eq ATypeNow where
  eq (ATypeNow { x:x1 }) (ATypeNow { x:x2 }) = x1 == x2
```

Writing an encoder for `newtype Meal` isn't as terrible as you might think. Here's the `encodeJson` Instance:

```
newtype Meal = Meal
  { main :: MainDish
  , side :: SideDish
  , dessert :: Dessert
  }

instance encodeJsonMeal :: EncodeJson Meal where
  encodeJson (Meal r) = encodeJson r ❶ ❷
```

❶ We destructure `Meal` to get to the contained Record.

❷ We call `encodeJson` on the Record.

This is boilerplate, but only 2 lines of boilerplate. Not too bad.

24.3. Ajax

Now that we know how to encode and decode JSON, we're ready to send and receive it from a remote server.

To do this, we're going to be using a package called `purescript-affjax`, which requires `affjax` to be put in our `spago.dhall` file.

We're going to be coding against a test API at JSON Placeholder.

24.3.1. GET Request

Let's start with a simple retrieval. According the API Guide, a `GET` to https://jsonplaceholder.typicode.com/posts/1 will produce:

```
{
  "id": 1,
  "title": "...",
  "body": "...",
  "userId": 1
}
```

So, let's write code to `GET` from this URL.

We'll start by modeling the response:

```
type GetPostRes =
  { id :: Int
  , title :: String
  , body :: String
  , userId :: Int
  }
```

Notice how we're starting with a Type Alias. We may need to make this a proper Type later, if, for example, we wish to give this Type an `Eq` Instance. We'll convert it later if the need arises, but for now, we're going to leave it as is.

Since all of our Types for this request are JSON Types, we should be able to use the Foreign library via

Generics.

And since `Records` already have `Encode` and `Decode` Instances, there's nothing more for us to do.

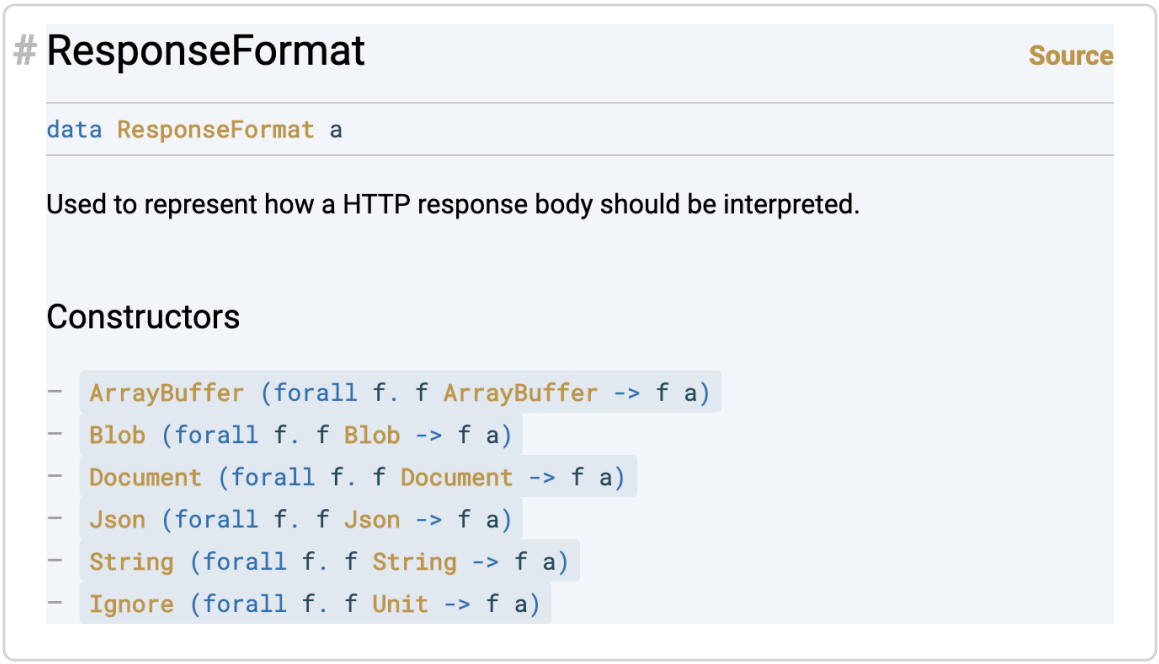
Now, we're ready to make the `Ajax` call to make the `GET` request. When we browse the `affjax` package we find the `Affjax` module has the following Function:

```
get :: ∀ a. ResponseFormat a -> URL -> Aff (Either Error (Response a))
```

A Function called `get` seems like a promising candidate for making an `HTTP GET` call. And so much more after reading the docs:

```
Makes a `GET` request to the specified URL.
```

Now, we have to figure out what Types its Parameters are. We'll start with `ResponseFormat`. Clicking on `ResponseFormat` in the `Pursuit` documentation produces:

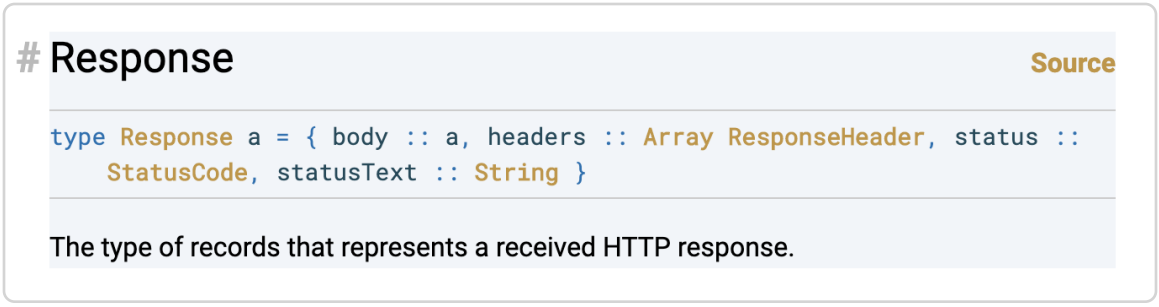


From this we can see, there are a lot of different response formats. But there is one that looks like what we're going to want and that's the `Json` one.

Another thing to notice is that the `Json` Data Constructor takes a Function that takes what looks like a `Functor` with `Json` in it, i.e. `f Json`, and produces that same `Functor` with an `a`, i.e. `f a`,. That probably has to do with some sort of conversion to the appropriate PureScript Type from JSON.

We'll worry about that later, if we have to.

Next thing to understand is `Response`. Hitting the back arrow on the Browser and clicking on `Response` gives us:

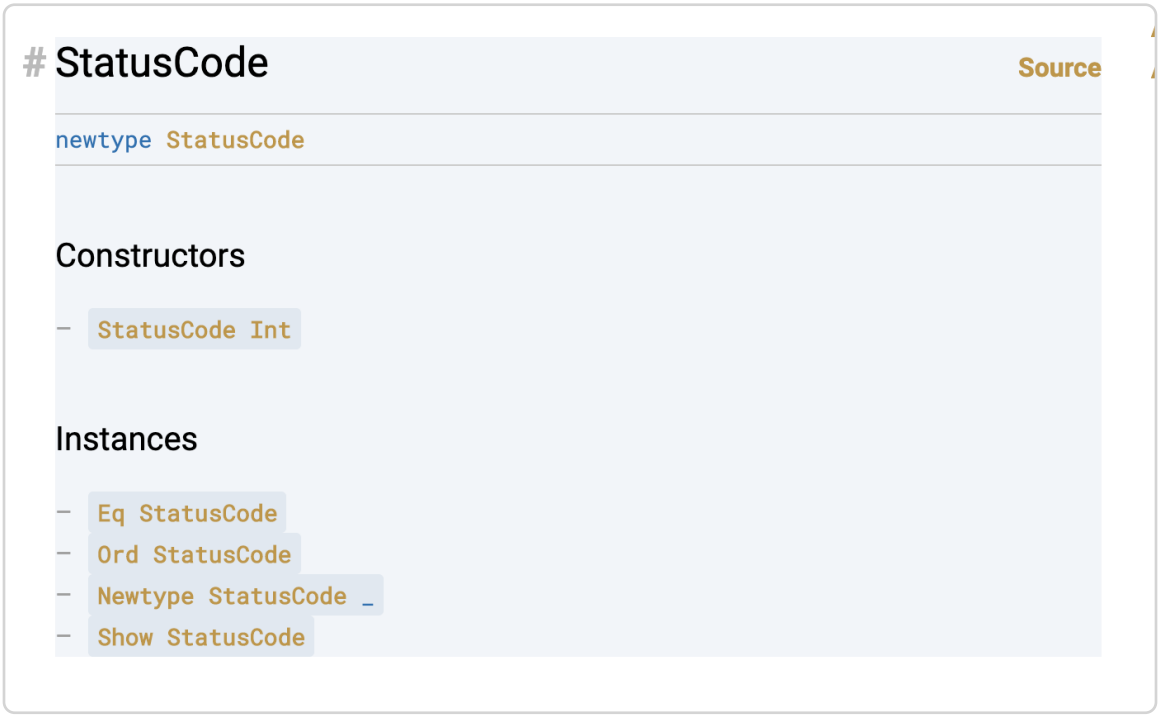


This is hard to read. So, when this happens, I usually just click on the `Source` link to the right, which produces:

```
type Response a =
  { status :: StatusCode
  , statusText :: String
  , headers :: Array ResponseHeader
  , body :: a
  }
```

Let's look at each field in detail.

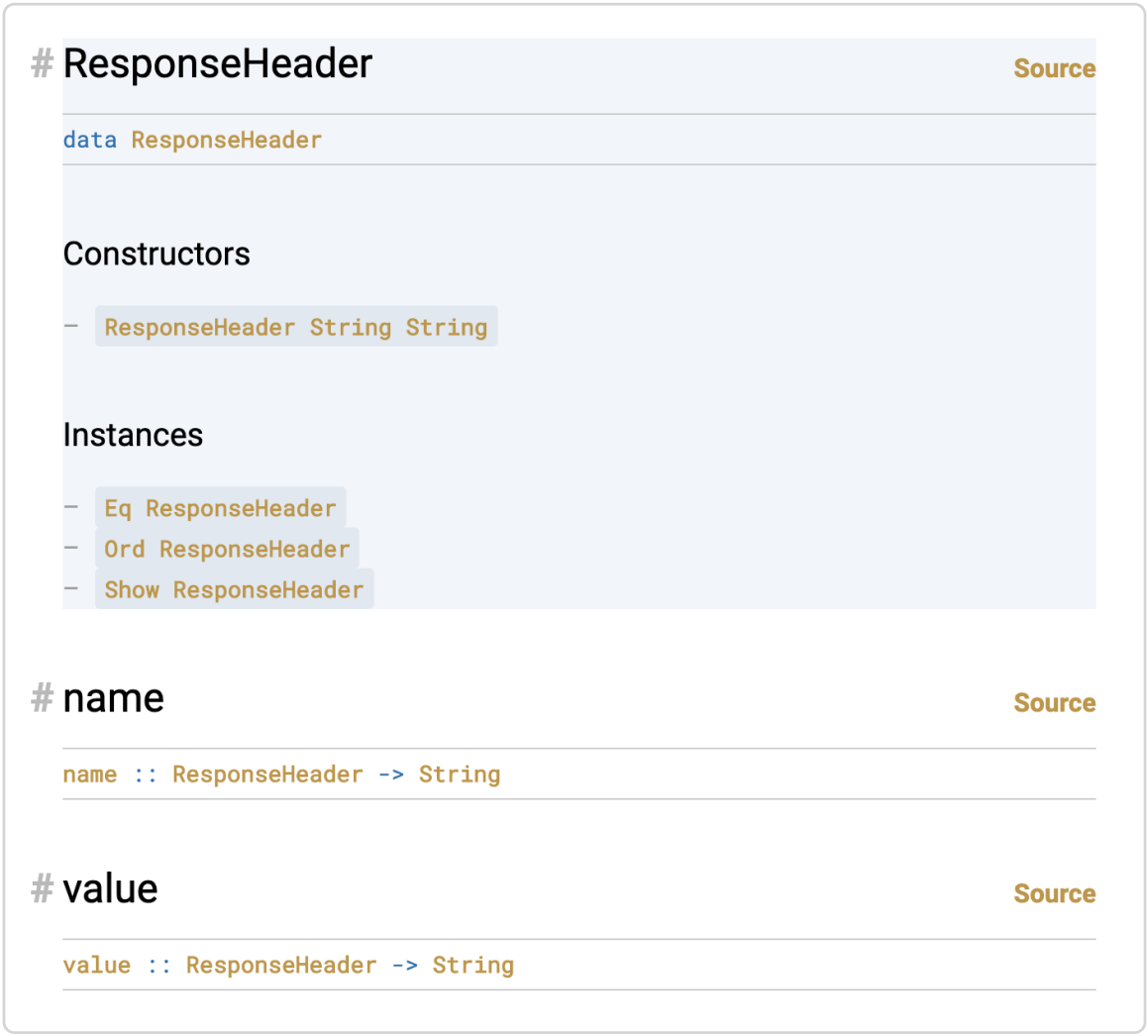
`status` appears to be the HTTP Status code. Clicking on the back arrow in the Browser and clicking on `Status` produces:



So this is just a wrapper around an `Int`. This has got to be the HTTP Status code, e.g. `200` for Success or `404` for Not Found, etc.

`statusText` must be the human-readable `status`.

`headers` seems to be the HTTP Response Headers. Clicking the back button on the Browser and clicking on `ResponseHeader` produces:



ResponseHeader Source

```
data ResponseHeader
```

Constructors

- `ResponseHeader String String`

Instances

- `Eq ResponseHeader`
- `Ord ResponseHeader`
- `Show ResponseHeader`

name Source

```
name :: ResponseHeader -> String
```

value Source

```
value :: ResponseHeader -> String
```

So, from this we can surmise that these are indeed the HTTP Response Headers where `name` will give us the name of the field in the header and `value` will give us its Value.

Here are the Response Headers for the favicon retrieval on the Pursuit website:

```
cache-control: public, max-age=31536000
content-type: image/vnd.microsoft.icon
date: Thu, 17 Sep 2020 16:16:39 GMT
etag: WTfCrMG3
expires: Thu, 31 Dec 2037 23:55:55 GMT
server: nginx/1.14.0 (Ubuntu)
status: 200
```

Each header has a `name`, which is a `String` followed by its Value, also a `String`. Note that the colon is NOT part of the `name`. It's there as a delimiter for parsing.

And the final field in `Response` is `body` which is of Type `a`. That's the payload we requested, i.e. the response of the HTTP Request.

Now we're ready to make our request. Let's revisit the `get` Type Signature:

```
get :: ∀ a. ResponseFormat a -> URL -> Aff (Either Error (Response a))
```

First thing we'll need is a `ResponseFormat`. We want `a` to be `Json` so, we're going to look on the same `Pursuit` page as `ResponseFormat` to see if there's a canned Function for this.

And it turns out that there is:

```
json :: ResponseFormat Json
```

Okay, so we can use that:

```
import Prelude

import Affjax as Ajax ❶
import Affjax.ResponseFormat as ResponseFormat ❷
import Effect (Effect)
import Effect.Aff (launchAff_) ❸
import Effect.Class.Console (log) ❹

type GetPostRes =
  { id :: Int
  , title :: String
  , body :: String
  , userId :: Int
  }

test :: Effect Unit
test = launchAff_ do
  Ajax.get ResponseFormat.json ???
```

❶ Import the main `Affjax` module qualified as `Ajax`.

❷ Import the `ResponseFormat` module qualified as `ResponseFormat`.

❸ Since `Ajax.get` runs in `Aff`, we need `launchAff_` to run it from `test`, which runs in `Effect`.

❹ Import the `MonadEffect` version of `log` so we can use it from both `Aff` and `Effect`.

Next, we need to add the URL:

```
test :: Effect Unit
test = launchAff_ do
  Ajax.get ResponseFormat.json
    "https://jsonplaceholder.typicode.com/posts/1"
```

The question at this point is what is `Ajax.get` returning. Let's add a Type Hole and find out:

```
test :: Effect Unit
test = launchAff_ do
  Ajax.get ResponseFormat.json
    "https://jsonplaceholder.typicode.com/posts/1" :: ?x
```

Hovering over the `?x`, we get the following pop-up:

```
Hole 'x' has the inferred type

  Aff
    (Either Error
       { body :: Json
       , headers :: Array ResponseHeader
       , status :: StatusCode
       , statusText :: String
       }
    )
```

Okay, so it seems as if we can get an `Either` if we extract it from the `Aff`:

```
import Data.Either (Either(..)) ❶

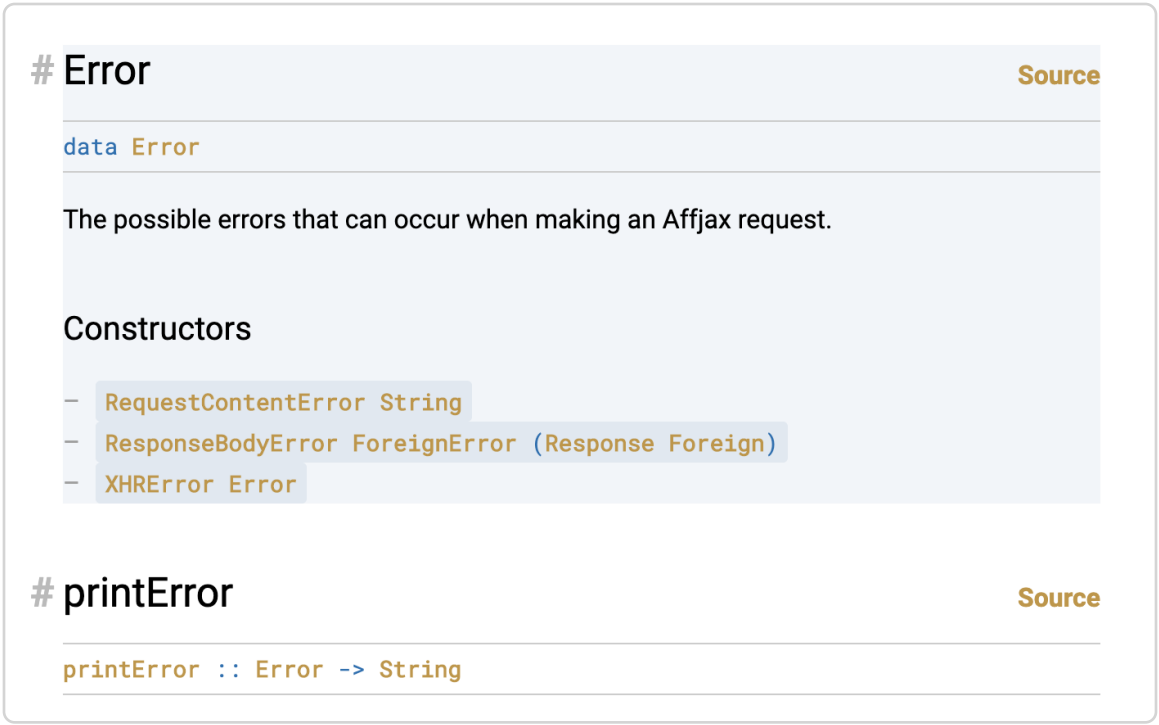
test :: Effect Unit
test = launchAff_ do
  result <- Ajax.get ResponseFormat.json
    "https://jsonplaceholder.typicode.com/posts/1"
  case result of ❷
    Left err -> ???
```

❶ Import `Either's` Data Constructors.

❷ Handle the results.

Now what can we do with the `Error` we get on the left-hand side of the `Either`?

Let's go look up `Error` in `Pursuit` being careful to get the right one:



Error Source

```
data Error
```

The possible errors that can occur when making an Affjax request.

Constructors

- `RequestContentError String`
- `ResponseBodyError ForeignError (Response Foreign)`
- `XHRError Error`

printError Source

```
printError :: Error -> String
```

Looking at the Type, we can see that it has 3 Data Constructors. I'm not sure we care about which type of error occurs at this point, just that one did.

Notice though, directly below the `Error` definition is a Function that'll take our `Error` and give us a `String`.

Why the library author just didn't give us a `Show` Instance is beyond me. But at least we can use `printError`:

```
test :: Effect Unit
test = launchAff_ do
  result <- Ajax.get ResponseFormat.json
    "https://jsonplaceholder.typicode.com/posts/1"
  case result of
    Left err -> log $ "ERROR: " <> Ajax.printError err ❶
    Right { body } -> ???
```

❶ Simply log on error condition.

The next question to ask is what Type is `body`?

Well, it's the `a` in `Response`:

```
type Response a =
  { status :: StatusCode
  , statusText :: String
  , headers :: Array ResponseHeader
  , body :: a
  }
```

And since we called `get`, we should look at that too:

```
get :: ∀ a. ResponseFormat a -> URL -> Aff (Either Error (Response a))
```

The first Parameter to our `get` call is `ResponseFormat.json`. This means that our `a` is `Json`, which means that our `body` is of Type `Json`.

We know how to deal with `Json`, we decode it remembering to provide an explicit Type for the compiler:

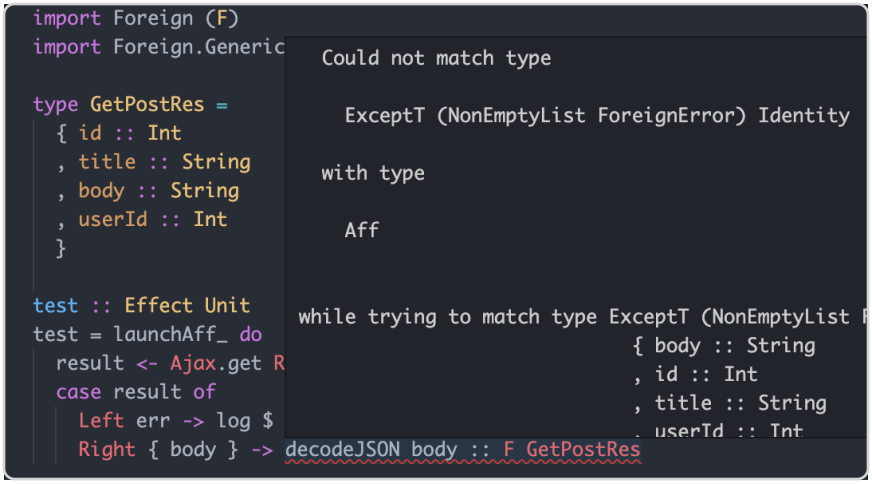
```
import Foreign (F) ❶
import Foreign.Generic (decodeJSON) ❷

test :: Effect Unit
test = launchAff_ do
  result <- Ajax.get ResponseFormat.json
    "https://jsonplaceholder.typicode.com/posts/1"
  case result of
    Left err -> log $ "ERROR: " <> Ajax.printError err
    Right { body } -> decodeJSON body :: F GetPostRes -- COMPILER ERROR!!
```

❶ Import the `Foreign` error Type Alias.

❷ Import `decodeJSON`.

We're not done coding yet, but the compiler error proves helpful:



This reminds us that `F` is an `ExceptT` with `Identity` on the bottom, aka. `Except`:

```
type F = Except MultipleErrors
```

So, we'll need to run this Monad Stack using `runExcept`:

```
runExcept :: ∀ e a. Except e a -> Either e a
```

This will give us an `Either`, so it looks like we'll need another `case` expression:

```
import Control.Monad.Except (runExcept) ❶

test :: Effect Unit
test = launchAff_ do
  result <- Ajax.get ResponseFormat.json
    "https://jsonplaceholder.typicode.com/posts/1"
  case result of
    Left err -> log $ "ERROR: " <> Ajax.printError err
    Right { body } ->
      case runExcept (decodeJSON body :: F GetPostRes) of ❷
        Left err -> ???
```

❶ Import `runExcept`.

❷ We must explicitly tell the compiler what Type we're decoding. This isn't always true. If we were to use the results of `decodeJSON` in such a way that the compiler could infer that it's a `GetPostRes` then we could remove this Type Signature.

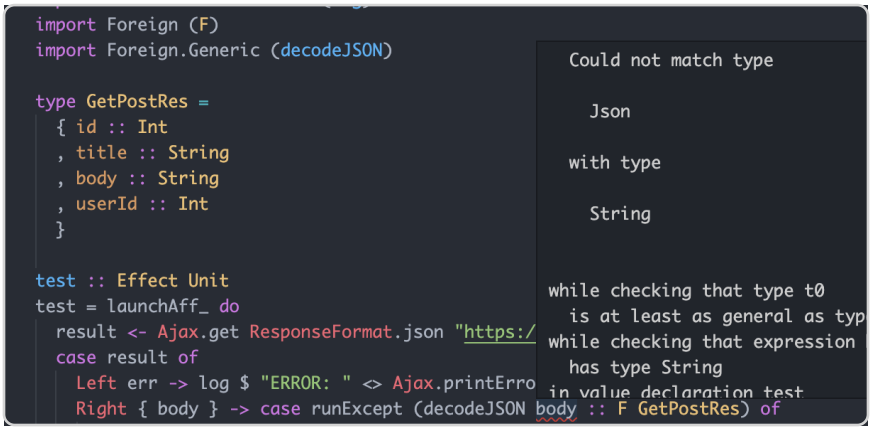
When the decoding gives us an error, we're going to log it out plus the body so we can examine it for errors:

```
test :: Effect Unit
test = launchAff_ do
  result <- Ajax.get ResponseFormat.json
    "https://jsonplaceholder.typicode.com/posts/1"
  case result of
    Left err -> log $ "ERROR: " <> Ajax.printError err
    Right { body } ->
      case runExcept (decodeJSON body :: F GetPostRes) of
        Left err -> log $ "ERROR: " <> show err <> " (" <> body <> ")"
        Right v -> ???
```

And for the successful case, we're just going to log the results the console when we get them. This is just test code after all:

```
test :: Effect Unit
test = launchAff_ do
  result <- Ajax.get ResponseFormat.json
    "https://jsonplaceholder.typicode.com/posts/1"
  case result of
    Left err -> log $ "ERROR: " <> Ajax.printError err
    Right { body } ->
      -- COMPILER ERROR!!
      case runExcept (decodeJSON body :: F GetPostRes) of
        Left err -> log $ "ERROR: " <> show err <> " (" <> body <> ")"
        Right v -> log $ show v
```

And we have a compiler error:



So, the problem is that `body` is Type `Json`, but `decodeJSON` wants a `String`. If we search the `Pursuit` for `Json -> String`, we come up short.

Granted we find `stringify` from the `Argonaut` library, but this should clue us to the fact that we're using

the wrong Type.

I made a bad decision to use `ResponseFormat.json`. So let's change it to `ResponseFormat.string`:

```
test :: Effect Unit
test = launchAff_ do
  result <- Ajax.get ResponseFormat.string ❶
    "https://jsonplaceholder.typicode.com/posts/1"
  case result of
    Left err -> log $ "ERROR: " <> Ajax.printError err
    Right { body } ->
      case runExcept (decodeJSON body :: F GetPostRes) of
        Left err -> log $ "ERROR: " <> show err <> " (" <> body <> ")"
        Right v -> log $ show v
```

❶ Changed `json` to `string`.

Now, it compiles.

In retrospect, I should've remembered that `Json` is a Javascript Type. This would be the Type we'd want if we were going to process this in Javascript. But since, we're not, we should just treat it like a `String`.

Running our program we get:

```
{ body: "quia et suscipit\nsuscipit recusandae consequuntur expedita et
cum\nreprehenderit molestiae ut ut quas totam\nnostrum rerum est autem sunt
rem eveniet architecto",
  id: 1,
  title: "sunt aut facere repellat provident occaecati excepturi optio
reprehenderit",
  userId: 1
}
```

You can test this by clicking on this link https://jsonplaceholder.typicode.com/posts/1 and checking what the Browser `GET` returns.

24.3.2. POST Request

Next, thing to add to our code will be an `HTTP POST`. Looking back at the API Guide, we find that we can `POST` to https://jsonplaceholder.typicode.com/posts to create a resource. By the way, a `GET` request to that same URL will produce a list of Blog Posts. So, if you click on that URL, don't be surprised that it's behaves differently than what we're expecting when we `POST` to that URL.

From the documentation, we see that the body of the `POST` should be:

```
{
  "title": "foo",
  "body": "bar",
  "userId": 1
}
```

So, let's model that request:

```
type CreateBlogPostReq =
  { title :: String
  , body :: String
  , userId :: Int
  }
```

And reading the documentation further, it says that we'll get an output of the following:

```
{
  "id": 101,
  "title": "foo",
  "body": "bar",
  "userId": 1
}
```

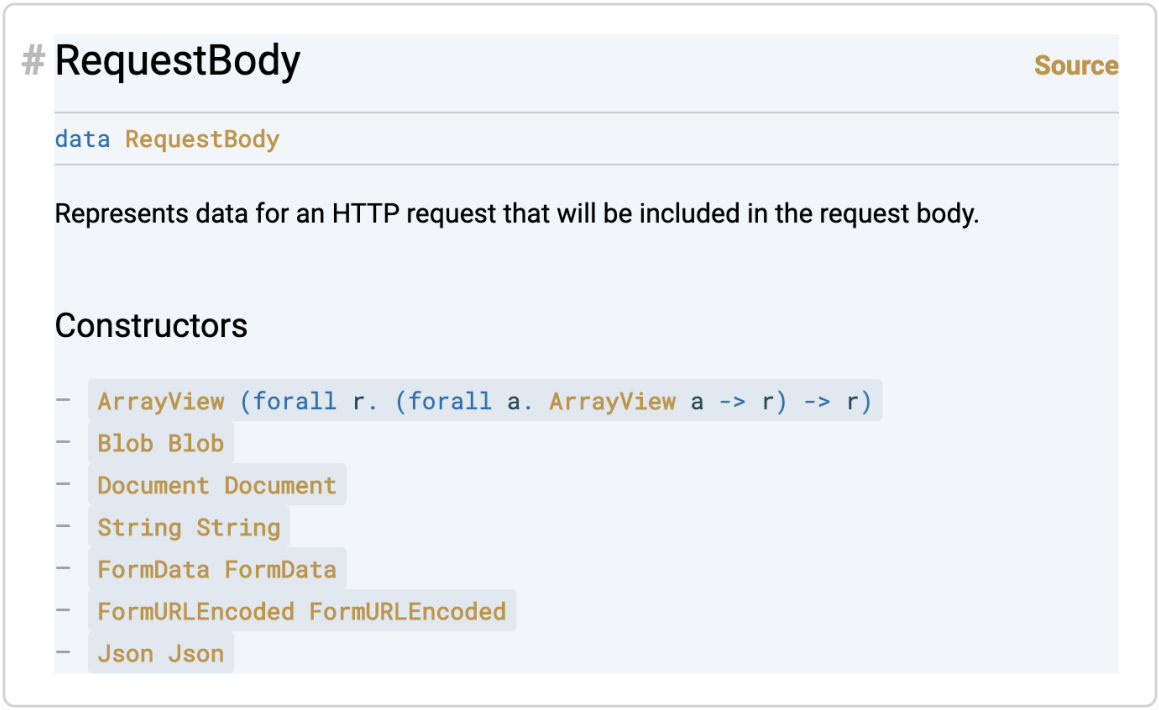
So, we'll model that:

```
type CreateBlogPostRes =
  { id :: Int
  , title :: String
  , body :: String
  , userId :: Int
  }
```

Now, we should look at the Type Signature for the `Ajax.post` Function:

```
post
  :: ∀ a. ResponseFormat a
  -> URL
  -> Maybe RequestBody
  -> Aff (Either Error (Response a))
```

This look identical to `get` with the addition of an OPTIONAL `RequestBody`. We should look at that in `Pursuit`:



RequestBody Source

```
data RequestBody
```

Represents data for an HTTP request that will be included in the request body.

Constructors

- `ArrayView (forall r. (forall a. ArrayView a -> r) -> r)`
- `Blob Blob`
- `Document Document`
- `String String`
- `FormData FormData`
- `FormURLEncoded FormURLEncoded`
- `Json Json`

There are many ways to encode our `RequestBody`, but we're just going to use `String` like we did last time. I don't want to make the same mistake I made before by choosing `Json`.

By the way, choosing `Json` is not a bad decision if you're using the `Argonaut` library.

Let's add the call to the code:

```
import Affjax.RequestBody as RequestBody ❶
import Data.Maybe (Maybe(..)) ❷
import Foreign.Generic (decodeJSON, encodeJSON) ❸

test :: Effect Unit
test = launchAff_ do
  result <- Ajax.get ResponseFormat.string
    "https://jsonplaceholder.typicode.com/posts/1"
  case result of
    Left err -> log $ "ERROR: " <> Ajax.printError err
    Right { body } ->
      case runExcept (decodeJSON body :: F GetPostRes) of
        Left err -> log $ "ERROR: " <> show err <> " (" <> body <> ")"
        Right v -> log $ show v
  postResult <- Ajax.post ResponseFormat.string ❹
    "https://jsonplaceholder.typicode.com/posts"
    (Just $ RequestBody.String $
      encodeJSON { userId:1, title:"title", completed:false })
  ???
```

❶ Import `RequestBody` qualified.

❷ Import `Maybe` with Data Constructors.

❸ Added `encodeJSON` to the import list.

❹ Added call to `post`.

The only real difference between the `get` call and the `post` one is the final Parameter. First, we started with `CreateBlogPostReq`:

```
{ userId:1, title:"title", completed:false }
```

Then we encoded it:

```
encodeJSON { userId:1, title:"title", completed:false }
```

And then we made a `RequestBody` out of it using the `String` Data Constructor:

```
RequestBody.String
  $ encodeJSON { userId:1, title:"title", completed:false }
```

And then we made it a `Maybe`:

```
Just $ RequestBody.String
  $ encodeJSON { userId:1, title:"title", completed:false }
```

Now, we have to deal with the results of the `POST` call, i.e. `postResult`. Processing this result will be similar to what we wrote for the `get` one.

So, let's factor that code out first, because that code is ugly.

We'll call our factored-out Function, `processAjaxResult`:

```
import Effect.Class (class MonadEffect) ❶

processAjaxResult
  :: ∀ m. MonadEffect m
  => Either Ajax.Error (Ajax.Response String)
  -> m Unit
processAjaxResult = case _ of
  Left err -> log $ "ERROR: " <> Ajax.printError err
  Right { body } ->
    case runExcept (decodeJSON body :: F GetPostRes) of
      Left err -> log $ "ERROR: " <> show err <> " (" <> body <> ")"
      Right v -> log $ show v
```

❶ Import `MonadEffect` class.

The body of this code is just a copy/paste of the `case` expression from our `test` code. The Type Signature is the only thing new here, so let's examine it:

```
processAjaxResult
  :: ∀ m. MonadEffect m
  => Either Ajax.Error (Ajax.Response String)
  -> m Unit
```

This is written to work in `MonadEffect`. This allows this Function to be called in any Monad that can log to the console, which also has the same requirement:

```
log :: ∀ m. MonadEffect m => String -> m Unit
```

Note that we could have hardcoded the `Monad` to be `Aff`, but it's always best to not hardcode `Monads` when it's not required. It keeps our code flexible.

The first Parameter to `processAjaxResult` is the result from our Ajax call, i.e. a Value of Type `Either`

`Ajax.Error (Ajax.Response String)`. Notice that this assumes we always use `ResponseFormat.string`.

`processAjaxResult` returns `m Unit`. It's only operation is the Side-effect of writing to the console.

Now, we're ready to modify our `test` code to use this:

```
test :: Effect Unit
test = launchAff_ do
  result <- Ajax.get ResponseFormat.string
    "https://jsonplaceholder.typicode.com/posts/1"
  processAjaxResult result ❶
  postResult <- Ajax.post ResponseFormat.string
    "https://jsonplaceholder.typicode.com/posts"
    (Just $ RequestBody.String $
      encodeJSON { userId:1, title:"title", completed:false })
  ??? ❷
```

❶ Call `processAjaxResult` in place of the code that we factored out, i.e. the `case` expression.

❷ We're still not done coding.

Now, here is a situation where we could use `bind` instead of having 2 lines of code. Here's the code that can be combined into a single line:

```
  result <- Ajax.get ResponseFormat.string
    "https://jsonplaceholder.typicode.com/posts/1"
  processAjaxResult result
```

We're extracting `result` to pass it to a Function, `processAjaxResult`. This is exactly what `bind` does, so let's use it:

```
test :: Effect Unit
test = launchAff_ do
  processAjaxResult =<< Ajax.get ResponseFormat.string ❶
    "https://jsonplaceholder.typicode.com/posts/1" ❷
  postResult <- Ajax.post ResponseFormat.string
    "https://jsonplaceholder.typicode.com/posts"
    (Just $ RequestBody.String $
      encodeJSON { userId:1, title:"title", completed:false })
```

❶ Notice how we eliminated the need to name a Variable, i.e. `result` is gone.

❷ Using the flipped version of `>>=` because it reads better. Notice how the flipped operator is NOT the same

characters in `>>=`, but, instead, is the mirror image of that operator, i.e. `=<<`.

Okay, now let's do the same for `postResult`:

```
test :: Effect Unit
test = launchAff_ do
  processAjaxResult =<< Ajax.get ResponseFormat.string
    "https://jsonplaceholder.typicode.com/posts/1"
  processAjaxResult =<< Ajax.post ResponseFormat.string
    "https://jsonplaceholder.typicode.com/posts"
    (Just $ RequestBody.String $
      encodeJSON { userId:1, title:"title", completed:false })
```

And this compiles. When we run this, we get:

```
{ body: "quia et suscipit\nsuscipit recusandae consequuntur expedita et
cum\nreprehenderit molestiae ut ut quas totam\nnostrum rerum est autem sunt
rem eveniet architecto", id: 1, title: "sunt aut facere repellat provident
occaecati excepturi optio reprehenderit", userId: 1 }
ERROR: (NonEmptyList (NonEmpty (ErrorAtProperty "userId" (TypeMismatch "Int"
"Undefined")) Nil)) ({
  "id": 101
})
```

We can see that the `POST` failed. Notice the actual response, which is at the end of the error:

```
({
  "id": 101
})
```

It seems as if the API has changed but the docs have not. So let's change our model:

```
type CreateBlogPostRes =
  { id :: Int
  }
```

Now when we run with our updated model we get:

```
[REDACTED GET OUTPUT]
ERROR: (NonEmptyList (NonEmpty (ErrorAtProperty "userId" (TypeMismatch "Int"
"Undefined")) Nil)) ({
  "id": 101
})
```

We still get an error. But we fixed our model. Okay, so now we'll need to look more carefully at the error message.

Take a look and see if you can figure out where I've messed up.

So, from `ErrorAtProperty`, I notice `userId` is the property that is causing the error. And from `TypeMismatch` I can see that it was expecting to parse an `Int` but instead got `Undefined`, i.e. `userId` was missing.

But why would it be still be looking for a `userId` when we removed it from `CreateBlogPostRes`?

We should search our code for `userId`.

When I do, I find:

```
type GetPostRes =
  { id :: Int
  , title :: String
  , body :: String
  , userId :: Int
  }

type CreateBlogPostReq =
  { title :: String
  , body :: String
  , userId :: Int
  }
```

Since, `CreateBlogPostReq` is a request we can rule it out. Now we should search the code for `GetPostRes`.

Here's what I found:

```
processAjaxResult
  :: ∀ m. MonadEffect m
  => Either Ajax.Error (Ajax.Response String)
  -> m Unit
processAjaxResult = case _ of
  Left err -> log $ "ERROR: " <> Ajax.printError err
  Right { body } ->
    case runExcept (decodeJSON body :: F GetPostRes) of ❶
      Left err -> log $ "ERROR: " <> show err <> " (" <> body <> ")"
      Right v -> log $ show v
```

❶ Here `GetPostRes` is the Type, which tells the compiler which `decodeJSON` to use. This is the problem.

So, I was a bit hasty when I factored out some code. The explicit Type works for the `GET` but not the `POST`.

I need a way to pass the correct Type to `processAjaxResult`. But how do you pass a Type?

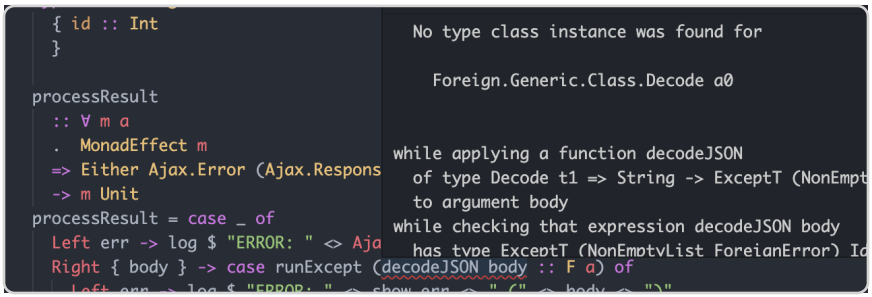
Well, let's try by just editing the Type Signature and add a Type `a` and replace `GetPostRes` with `a`:

```
processAjaxResult
  :: ∀ m a. MonadEffect m ❶
  => Either Ajax.Error (Ajax.Response String)
  -> m Unit
processAjaxResult = case _ of
  Left err -> log $ "ERROR: " <> Ajax.printError err
  -- COMPILER ERROR!!
  Right { body } -> case runExcept (decodeJSON body :: F a) of ❷
    Left err -> log $ "ERROR: " <> show err <> " (" <> body <> ")"
    Right v -> log $ show v
```

❶ Added `a` to `∀`.

❷ Replaced `GetPostRes` with `a`.

So, this is what we want but now the compiler complains:



Our `a` can't just be ANY OLD Type. It has to be decodable. Let's fix it:

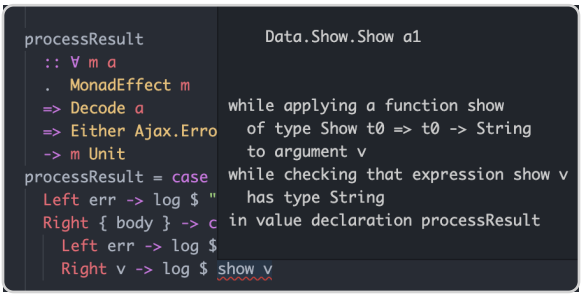
```
import Foreign.Generic.Class (class Decode) ❶

processAjaxResult
  :: ∀ m a
  .  MonadEffect m
  => Decode a ❷
  => Either Ajax.Error (Ajax.Response String)
  -> m Unit
processAjaxResult = case _ of
  Left err -> log $ "ERROR: " <> Ajax.printError err
  Right { body } -> case runExcept (decodeJSON body :: F a) of
    Left err -> log $ "ERROR: " <> show err <> " (" <> body <> ")"
    Right v -> log $ show v -- COMPILER ERROR!!
```

❶ Import the Decode class.

❷ Constrain a to be decodable.

And now another compiler issue:

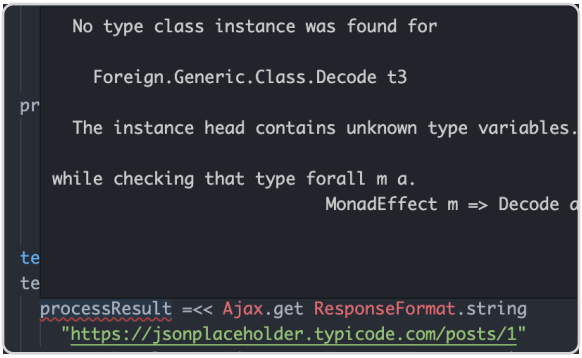


And we need it to be showable:

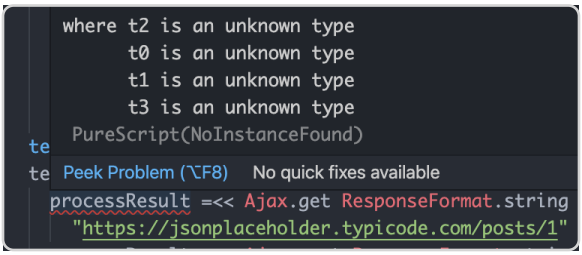
```
processAjaxResult
  :: ∀ m a
  .  MonadEffect m
  => Decode a
  => Show a ❶
  => Either Ajax.Error (Ajax.Response String)
  -> m Unit
processAjaxResult = case _ of
  Left err -> log $ "ERROR: " <> Ajax.printError err
  Right { body } -> case runExcept (decodeJSON body :: F a) of
    Left err -> log $ "ERROR: " <> show err <> " (" <> body <> ")"
    Right v -> log $ show v
```

❶ Constrain a for Show.

And as soon as that compiles, the compiler is unhappy with our test code:



The fact that it has a Type called `t3`, usually tells us that it's an unknown Type. We can see if that's true by scrolling to the bottom of the error message:



Yep. `t3` is an unknown Type. So it's looking for a Type that has a `Decode` Instance, but it doesn't know what Type.

This is a clear indicator that we need to tell the compiler what it's decoding. We removed any hope of the compiler knowing from `processAjaxResult`. All it has now is an `a`.

So, we'll have to add that here:

```
test :: Effect Unit
test = launchAff_ do
  processAjaxResult =<< Ajax.get ResponseFormat.string
    "https://jsonplaceholder.typicode.com/posts/1" :: ??? ❶
  processAjaxResult =<< Ajax.post ResponseFormat.string
    "https://jsonplaceholder.typicode.com/posts"
    (Just $ RequestBody.String $
      encodeJSON { userId:1, title:"title", completed:false })
```

❶ What Type do we put here that will define `a`? `processAjaxResult` returns an `m Unit`. There's no `a` there.

Okay, so it's not so clear how to tell the compiler what Type `a` is. If that's not clear, look at `processAjaxResult's` Type Signature:

```
processAjaxResult
  :: ∀ m a
  .  MonadEffect m
  => Decode a
  => Show a
  => Either Ajax.Error (Ajax.Response String)
  -> m Unit
```

Notice that a is only used in Constraints. There are no Parameters with a in the Type.

This is a perfect case of a ***Phantom Type***, but for a Type Signature. We saw Phantom Types early in the book, but it had to do with Types. But, this is a case where a Type is a Phantom Type in a Type Signature.

We can't have that. We'll need to have a Parameter that's passed to `processAjaxResult` that will have Type a in it.

Do you remember how we pass a Parameter just to define a Type?

We use `Proxy` from `Type.Proxy` to tell the compiler:

```
data Proxy a = Proxy
```

Let's change our Type Signature for `processAjaxResult` to include a `Proxy` Parameter:

```
import Type.Proxy (Proxy(..)) ❶

processAjaxResult
  :: ∀ m a
  .  MonadEffect m
  => Decode a
  => Show a
  => Proxy a ❷
  -> Either Ajax.Error (Ajax.Response String)
  -> m Unit
processAjaxResult _ = case _ of ❷
  Left err -> log $ "ERROR: " <> Ajax.printError err
  Right { body } -> case runExcept (decodeJSON body :: F a) of
    Left err -> log $ "ERROR: " <> show err <> " (" <> body <> ")"
    Right v -> log $ show v
```

❶ Import `Proxy` with its Data Constructor.

❷ Made `Proxy` the first parameter for 2 reasons. First, it works well at the call site when we use flipped

`bind`, i.e. `=<<`. Second, we can still use the wildcard in the `case` expression.

Now, we need to pass a `Proxy`. Remember, `Proxy` is a Data Constructor with NO Type Parameters. So, we define its Type explicitly, i.e. we specify its `a`, which will unify with `processAjaxResult's a`:

```
test :: Effect Unit
test = launchAff_ do
  processAjaxResult (Proxy :: _ GetPostRes) ❶
    =<< Ajax.get ResponseFormat.string
      "https://jsonplaceholder.typicode.com/posts/1"
  processAjaxResult (Proxy :: _ CreateBlogPostRes) ❷
    =<< Ajax.post ResponseFormat.string
      "https://jsonplaceholder.typicode.com/posts"
      (Just $ RequestBody.String $
        encodeJSON { userId:1, title:"title", completed:false })
```

❶ Added `Proxy` Parameter for `GetPostRes`. Notice that the compiler can figure out the `_` is `Proxy`.

❷ Added `Proxy` Parameter for `CreateBlogPostRes`. Notice that the compiler can figure out the `_` is `Proxy`.

Now when we run it we get:

```
{ body: "quia et suscipit\nsuscipit recusandae consequuntur expedita et
cum\nreprehenderit molestiae ut ut quas totam\nnostrum rerum est autem sunt
rem eveniet architecto", id: 1, title: "sunt aut facere repellat provident
occaecati excepturi optio reprehenderit", userId: 1 }
{ id: 101 } ❶
```

❶ The `POST` call works now.

Chapter 25. Coding With Ajax and JSON

Return to the project that you've been coding in and create a file called `Ch25a.purs` and add the following:

```
module Ch25a where

import Prelude ❶

import Effect (Effect)
import Effect.Class.Console (log) ❷

test :: Effect Unit
test = do
  log "placeholder" ❸
```

❶ Once again we are not writing anything from Prelude so we'll just import all of it.

❷ We're now using this version of `log` so we can use it in `Effect` and `Aff`.

❸ We'll delete this soon enough. It's just here to keep the compiler happy.

Next, change your `Main` module to import `Ch25a` and call its `test` Function:

```
module Main where

import Prelude
import Effect (Effect)

import Ch25a as Ch25a

main :: Effect Unit
main = Ch25a.test
```

If you run:

```
npx spago run
```

you should see it build and print:

```
[info] Build succeeded.
placeholder
```

25.1. Ajax Program Specifications

The program we're going to write will communicate to a backend that you'll run locally and that will take any JSON you send it and simply traverse it as a tree reversing the names of all of the keys.

For example, if given the following:

```
{
  "cost": 12.34,
  "size": 6,
  "attributes": [
    {
      "color": { "tag": "Red" },
      "shape": { "tag": "Square" },
      "type": { "tag": "One", "contents": [1, "one"] }
    }
  ]
}
```

it will response with:

```
{
  "tsoc": 12.34,
  "ezis": 6,
  "setubirtta": [
    {
      "roloc": { "gat": "Red" },
      "epahs": { "gat": "Square" },
      "epyt": { "gat": "One", "stnetnoc": [1, "one"] }
    }
  ]
}
```

This test server is written to run in Node. It can be found online in my GitHub Gist. It's called echo-server.js.

Here is the Javascript code for that server in case you're not online:

```
const http = require("http");

const hostname = "0.0.0.0";
const port = 3000;
```

```
const server = http.createServer((req, res) => {
  console.log(`\n${req.method} ${req.url}`);
  console.log(req.headers);

  const reverseArray = a => {
    var na = [];
    for (var i = 0; i < a.length; ++i) {
      const value = a[i];
      const nValue = value.constructor.name == 'Object'
        ? reverseKeys(value) : value;
      na.push(nValue);
    }
    return na;
  }
  const reverseKeys = o => {
    const no = {};
    for (const [key, value] of Object.entries(o)) {
      const nValue = value.constructor.name == 'Object'
        ? reverseKeys(value)
        : (value.constructor.name == 'Array'
          ? reverseArray(value) : value);
      no[key.split("").reverse().join("")] = nValue;
    }
    return no;
  };

  req.on("data", function (chunk) {
    console.log("BODY: " + chunk);
    res.statusCode = 200;
    res.setHeader("Content-Type", "text/plain");
    const o = reverseKeys(JSON.parse(chunk));
    const response = JSON.stringify(o, null, 2);
    console.log("RESPONSE: " + response);
    res.end(response);
  });

});

server.listen(port, hostname, () => {
  console.log(`Server running at http://localhost:${port}/`);
});
```

If you type in this program, call it `echo-server.js` and put it in the `src` directory. Otherwise, copy and paste it from my Gist.

The following is the data Model that we will use, so please add this to your code:

```
newtype Centimeters = Centimeters Number
newtype Kilograms = Kilograms Number
newtype Years = Years Int

type Personal =
  { height :: Centimeters
  , weight :: Kilograms
  , age :: Years
  }

newtype GPA = GPA Number

data Grade = Preschool | Kindergarten | Grade Int | High Int | College Int

type Student =
  { grade :: Grade
  , teacher :: Teacher
  , gpa :: GPA
  , personal :: Personal
  }

data TeachingStatus = Student | Probationary | NonTenured | Tenured

type Teacher =
  { grades :: Array Grade
  , numberOfStudents :: Int
  , personal :: Personal
  , status :: TeachingStatus
  }
```

We're going to send Teacher and Student to the Echo Server serially and then in parallel. We'll decode its response explicitly using the Foreign JSON decoding and then we'll decode it back into itself using a hand-written `Argonaut` decoder.

And then finally, in the FFI (Foreign Function Interface) Chapter, we'll reverse the keys back in Javascript BEFORE we decode it.

Let's get started.

25.2. Coding Ajax and JSON with Foreign

Before we get started, let's update our `spago.dhall` file to add the `purescript-foreign-generic` and

the `purescript-affjax` package:

```
{-
{-
Welcome to a Spago project!
You can edit this file as you like.
-}
{ name = "my-project"
, dependencies =
  [ "aff"
  , "aff-bus"
  , "arrays"
  , "avar"
  , "bifunctors"
  , "console"
  , "contravariant"
  , "control"
  , "datetime"
  , "effect"
  , "either"
  , "exceptions"
  , "foreign-generic" (1)
  , "foldable-traversable"
  , "integers"
  , "lists"
  , "maybe"
  , "newtype"
  , "nonempty"
  , "prelude"
  , "profunctor"
  , "psci-support"
  , "random"
  , "strings"
  , "tailrec"
  , "transformers"
  , "tuples"
  , "unfoldable"
  , "unicode"
  , "debug"
  ]
, packages = ./packages.dhall
, sources = [ "src/**/*.purs", "test/**/*.purs" ]
}
```

❶ I always like to keep my packages in alpha order.

Then run:

```
npx spago build
```

We'll ignore the warning this produces.

Now, let's start coding by making instances for `Centimeters` for decoding and encoding using the Foreign modules.

Look up the Foreign package, `purescript-foreign-generic`, on `Pursuit` and search for the encoding and decoding Typeclasses. Hint: Check out the `Foreign.Generic.Class` module.

Which Typeclasses did you find?

There are quite a few in that module. The ones that we're concerned with though are `Decode` and `Encode`. Go ahead and make instances for `Centimeters` for these 2 Typeclasses.

There are multiple ways to do this, so determine the best way and implement it before continuing further.

So, what are the 2 ways?

Here's the manual way first:

```
import Data.Generic.Rep (class Generic) ❶
import Foreign.Generic (genericEncode, genericDecode) ❷
import Foreign.Generic.Class (  class Encode ,class Decode ❸
                              , defaultOptions)

newtype Centimeters = Centimeters Number
derive instance genericCentimeters :: Generic Centimeters _ ❹

instance encodeCentimeters :: Encode Centimeters where ❺
  encode = genericEncode defaultOptions

instance decodeCentimeters :: Decode Centimeters where ❻
  decode = genericDecode defaultOptions
```

❶ Import `Generic` Typeclass, hence the `class` keyword.

❷ Import generic encode and decode functions. These functions are just like `genericShow`, i.e. they use the Generic Representation of the Type to encode and decode to and from JSON.

❸ Import the Typeclasses so we can write instances.

❹ Derive `Generic` for `Centimeters`.

❺ `Encode` Instance for `Centimeters`.

❻ `Decode` Instance for `Centimeters`.

If you didn't do this method, go ahead and COMMENT OUT your code and add this for now.

Next, add the following to your test code:

```
import Foreign.Generic (genericEncode, genericDecode, encodeJSON) ❶

test :: Effect Unit
test = do
  log $ encodeJSON $ Centimeters 12.34 ❷
```

❶ Added `encodeJSON` to import list.

❷ Prints `{"contents":12.34,"tag":"Centimeters"}`.

Run your program to make sure you get the expected output.

Notice how complex the output is for `Centimeters`. This is nice since it brings Type Safety into the JSON world. So, there are many times where this is the best solution like when you're writting Production Code.

But for our purposes here, i.e. learning, we're going to forego the niceties of Type Safety for simplicity and JSON readability by coding it by deriving via `newtype`:

```
import Foreign.Generic.Class (class Decode, class Encode) ❶
import Foreign.Generic (encodeJSON) ❶

newtype Centimeters = Centimeters Number
derive instance genericCentimeters :: Generic Centimeters _ ❷
derive newtype instance encodeCentimeters :: Encode Centimeters ❸
derive newtype instance decodeCentimeters :: Decode Centimeters ❸

-- instance encodeCentimeters :: Encode Centimeters where ❹
--   encode = genericEncode defaultOptions

-- instance decodeCentimeters :: Decode Centimeters where
--   decode = genericDecode defaultOptions
```

❶ Removed unused imports.

❷ We do NOT need this, but we'll leave it since we're going to need it when we write the `Show` Instance for this Type.

❸ Derive via `newtype`, i.e. delegate encoding and decoding to the wrapped Type, i.e. to `Number`.

❹ Don't worry about the warnings you get on our `import` declarations at the top of the module telling us

that they are redundant. This is because we commented out this bit of code. But, we'll need those imports soon enough. So, just leave them.

Now when you run the test code, you should get the following:

```
12.34 ❶
```

❶ With `derive newtype`, `Centimeters` delegates its `Encode` and `Decode` Instances to the underlying Type's instance Implementations, in this case `Number's`. This is why it displays a `Number`.

This is far nicer for our purposes here even though I usually prefer the Type Safe approach. That's because we just have a `Number` here. There's no guarantee that it's in `Centimeters`. Could be in feet, miles or dollars for all we know.

But, it'll do nicely for this bit of code and, therefore, we're going to want to the same thing for all of the `newtypes` in our code.

So, please do the same thing to all of the `newtypes` as we have done with `Centimeters` making sure to add the `Generic` derivation in preparation for writing `Show` Instances, then continue reading.

Your code should look like:

```
newtype Kilograms = Kilograms Number
derive instance genericKilograms :: Generic Kilograms _
derive newtype instance encodeKilograms :: Encode Kilograms
derive newtype instance decodeKilograms :: Decode Kilograms

newtype Years = Years Int
derive instance genericYears :: Generic Years _
derive newtype instance encodeYears :: Encode Years
derive newtype instance decodeYears :: Decode Years

newtype GPA = GPA Number
derive instance genericGPA :: Generic GPA _
derive newtype instance encodeGPA :: Encode GPA
derive newtype instance decodeGPA :: Decode GPA
```

Take a look at the code you have so far and take inventory of which Types, not Type Aliases, that can be encoded and decoded. Do that before reading my inventory list.

Here's my list:

```
-- Types that can be encoded and decoded
newtype Centimeters = Centimeters Number
newtype Kilograms = Kilograms Number
newtype Years = Years Int
newtype GPA = GPA Number

-- Types that cannot (yet)
data Grade = Preschool | Kindergarten | Grade Int | High Int | College Int
data TeachingStatus = Student | Probationary | NonTenured | Tenured
```

Our Type Aliases here are all Records. Records can be automatically encoded and decoded if and only if every field can be. That means, if we make `Grade` and `TeachingStatus` encodable and decodable, then our Type Aliases will also be.

So, go ahead and do that now by making the appropriate instances for `Grade` and `TeachingStatus`. Once, you've done that, continue on.

Here are the instances for `Grade`:

```
import Foreign.Generic (encodeJSON, genericEncode, genericDecode
                      , defaultOptions) ❶

data Grade = Preschool | Kindergarten | Grade Int | High Int | College Int
derive instance genericGrade :: Generic Grade _

instance encodeGrade :: Encode Grade where
  encode = genericEncode defaultOptions

instance decodeGrade :: Decode Grade where
  decode = genericDecode defaultOptions
```

❶ Added `genericEncode`, `genericDecode` and `defaultOptions` to the import list.

And now, if you struggled with `Grade` and didn't get to `TeachingStatus`, stop here and give it a try before reading on.

Here are the instances for `TeachingStatus`:

```
data TeachingStatus = Student | Probationary | NonTenured | Tenured
derive instance genericTeachingStatus :: Generic TeachingStatus _

instance encodeTeachingStatus :: Encode TeachingStatus where
  encode = genericEncode defaultOptions

instance decodeTeachingStatus :: Decode TeachingStatus where
  decode = genericDecode defaultOptions
```

Now that all of our Types can be encoded and decoded to and from JSON, we can change our test code to the following:

```
teacher :: Teacher
teacher =
  { grades: [ Preschool, Kindergarten, Grade 1 ]
  , numberOfStudents: 23
  , personal: {
      height: Centimeters 162.56
    , weight: Kilograms 63.5
    , age: Years 31
    }
  , status: NonTenured
  }

test :: Effect Unit
test = do
  log $ encodeJSON teacher ❶
```

❶ Prints `{"status":{"tag":"NonTenured"},"personal":{"weight":63.5,"height":162.56,"age":31},"numberOfStudents":23,"grades":[{"tag":"Preschool"},{"tag":"Kindergarten"},{"contents":1,"tag":"Grade"}]}`.

You can take the output and copy and paste it into a JSON file in VSCode and it will be pretty printed making it far more readable:

```
{
  "status": {
    "tag": "NonTenured"
  },
  "personal": {
    "weight": 63.5,
    "height": 162.56,
    "age": 31
  },
  "numberOfStudents": 23,
  "grades": [
    {
      "tag": "Preschool"
    },
    {
      "tag": "Kindergarten"
    },
    {
      "contents": 1, ❶
      "tag": "Grade"
    }
  ]
}
```

❶ Note that contents is the 1 and only Parameter to the `Grade` Data Constructor.

25.2.1. Encoding and Sending to the Echo Server

The next thing you're going to do is replace the `log $ encodeJSON teacher` in the test code with a call to the Echo Server to POST `teacher` as JSON. That call's results will then be logged to the console.

But before you do, there's a few things you'll need to know.

First, add `affjax` to `spago.dhall` and then build with:

```
npx spago build
```

Next, add the following import statements (you'll need more, but this is a good starting point):

```
import Affjax as Ajax
import Affjax.ResponseFormat as ResponseFormat
import Affjax.RequestBody as RequestBody
import Data.Bifunctor (lmap) ❶
import Effect.Aff (launchAff_) ❷
```

❶ This should give you a clue.

❷ This too is a clue.

The next thing to know is that the server listens on port `3000`. If that conflicts with something on your machine, you can always change that port by changing the following line of code in `echo-server.js`:

```
const port = 3000;
```

Another thing to know is that the URL to this server must have a trailing `/`. The only reason this is not typically needed in a Browser is because the server usually does a HTTP Redirect by returning Status Code 301 or 302 and adding the trailing `/` to the original URL.

For example, when you browse to `https://pursuit.purescript.org` the Browser will get redirected to `https://pursuit.purescript.org/`.

There's no such redirect in our Echo Server, so make sure not to forget the trailing `/`.

The Echo Server also doesn't support HTTPS, so the URL must have `http://` in it.

And last, but not least, look up `post` on `Pursuit` in the `Affjax` Module to help you.

Go ahead and replace the `log $ encodeJSON teacher` in the test code with a call to the Echo Server to POST `teacher` as JSON and see how far you get before returning here.

Let's code this one piece at a time. First, we know we must call `post`, but it runs in `Aff`, so we must also use `launcheAff_`:

```
test :: Effect Unit
test = launchAff_ do ❶
  Ajax.post ??? ❷
```

❶ Run in `Aff`, which makes the `do` block an `Aff` Monad.

❷ Call `post` fully qualified.

Now, the next Parameter to `post` is the `ResponseFormat`, which, in the last Chapter, we determined is `String`:

```
test :: Effect Unit
test = launchAff_ do
  Ajax.post ResponseFormat.string ???
```

Now, the next Parameter is the URL. But what type is URL?

Go to `Pursuit` and search for `post` in `Affjax` again. When you find the appropriate `post` click on it to bring up it's documentation. Then click on `URL` link in `post's` Type Signature. It should show you the following definition:

```
type URL = String
```

So, URL is just a Type Alias for `String`. Let's add the URL as a `String`:

```
test :: Effect Unit
test = launchAff_ do
  Ajax.post ResponseFormat.string "http://localhost:3000/" ??? ❶ ❷
```

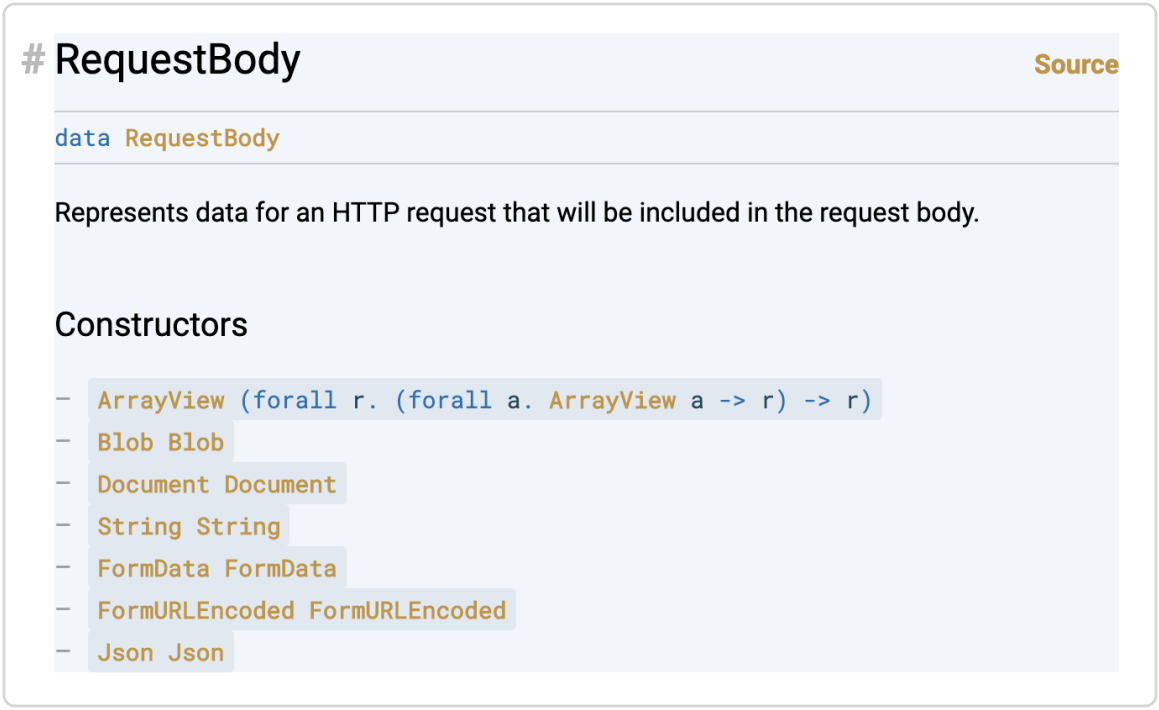
❶ Do NOT forget the trailing `/`.

❷ Notice it's `http://` NOT `https://`.

The third and final Parameter is the body. But look at the Type for the third Parameter:

```
Maybe RequestBody
```

And when we `RequestBody` we see many different Data Constructors supported:



But if you remember, in the last Chapter, we decided to use the `String` Data Constructor.

If you didn't get this far, stop and see if you can finish coding. And if not, at least code the third Parameter before returning to the book. It's okay if all you do is add the third Parameter, but try to go further.

Here's the third Parameter:

```
import Data.Maybe (Maybe(..)) ❶

test :: Effect Unit
test = launchAff_ do
  Ajax.post ResponseFormat.string "http://localhost:3000/"
    $ Just $ RequestBody.String $ encodeJSON teacher ❷
```

❶ Import Maybe Data Constructors.

❷ First, we encode `teacher`, then wrap it up in `Request.Body` and then a `Maybe`.

So, we've got our call to the Echo Server. What should we code next?

If your answer involves coding something, then I'd like to say that you've skipped a step.

First, we should ask ourselves, what Type does `Ajax.post` return. Go look at `Pursuit` to find out, then return back here.

This is what you should've gotten:

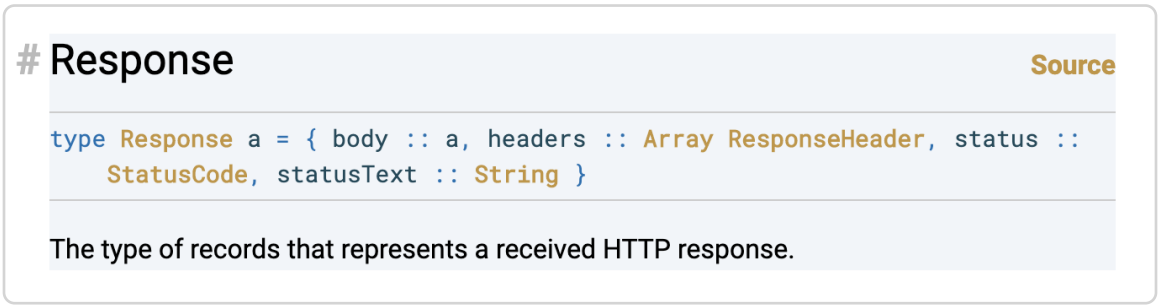
```
post
  :: ∀ a. ResponseFormat a
  -> URL
  -> Maybe RequestBody
  -> Aff (Either Error (Response a)) ❶
```

❶ The return Type.

We can see our return Type from this, but what Type is `a`? Or more important than that is what Type is `Response a`?

Look at `Response` on `Pursuit` to see and then return here.

You should have found:



`a` is the Type for the body. So, see if you can figure out what dictates the body Type in the `post` call, then keep reading.

Here's our code:

```
test :: Effect Unit
test = launchAff_ do
  Ajax.post ResponseFormat.string "http://localhost:3000/"
    $ Just $ RequestBody.String $ encodeJSON teacher
```

Take a look at the first Parameter, `ResponseFormat.string`. This is what dictates that the response will be a `String`.

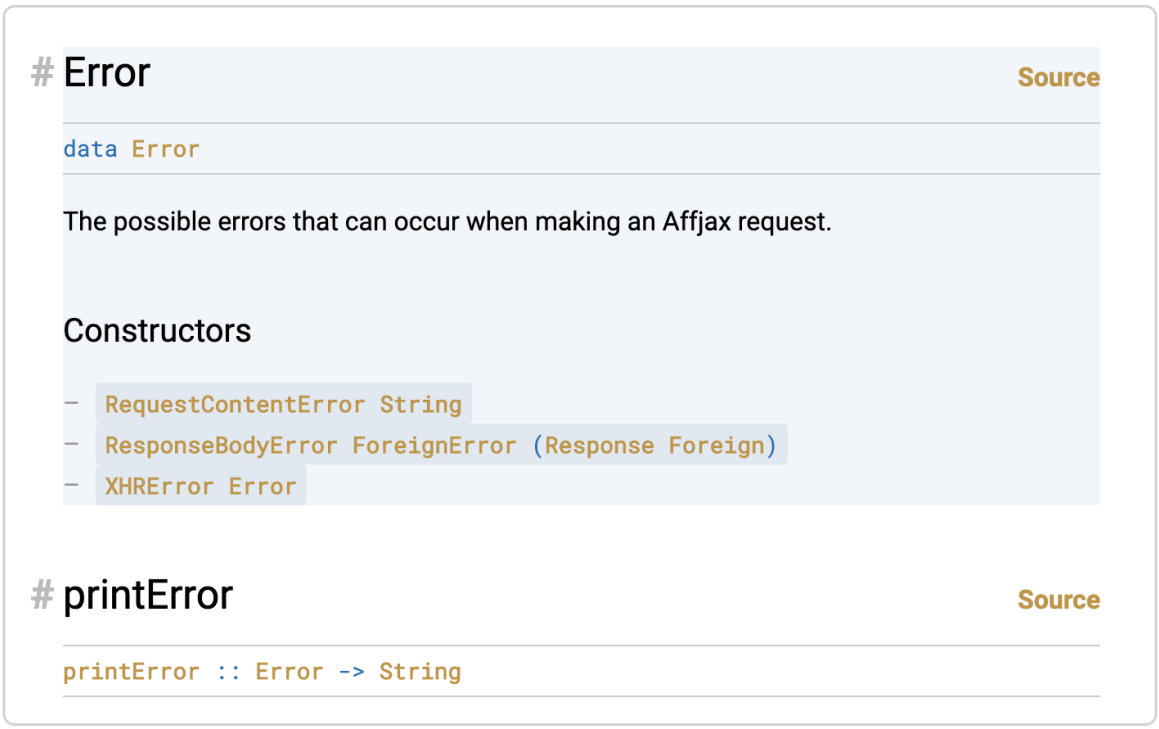
So, `a` is a `String` and that means our call to `post` returns a Value of Type:

```
Aff (Either Error (Response String))
```

Now, we just want to log the results from this. But the results are buried in `Aff` and then in `Either`. We're going to have to make 2 hops to get to them.

Let's take them one at a time. First, we'll look at `Error`. Can we `show` this? Take a look at `Pursuit` to see if it can be shown and then come back here.

You should have found:



Notice that there's no mention of a `Show` Instance for this Type. But, just below it is the definition for `printError`, which does effectively what `show` does. So, we can use that to make `Error` displayable.

Next, we turn our attention to `Response a`. We saw earlier that `Response` is a Record and Records have a `Show` Instance. That means we don't have to do anything to make is showable.

So, now we need to use `printError` on the `Left` of the `Either`. What mechanism have we learned about that lets us map on the left?

If you answered, `lmap` or `Bifunctor`, then you're correct. We can use `lmap Ajax.printError` on the result.

Keeping in mind that our `Either` is still buried in an `Aff`, go ahead and try to finish the code with all of this new information and see how you do before reading on.

Let's extract the result from the `Aff` Context:

```
test :: Effect Unit
test = launchAff_ do
  result <- Ajax.post ResponseFormat.string
    "http://localhost:3000/"
    $ Just $ RequestBody.String $ encodeJSON teacher
  ???
```

Next, we'll use `lmap` and log it:

```
test :: Effect Unit
test = launchAff_ do
  result <- Ajax.post ResponseFormat.string
    "http://localhost:3000/"
    $ Just $ RequestBody.String $ encodeJSON teacher
  log $ show $ lmap Ajax.printError result
```

While this code is just fine, we COULD combine this into one line of code. I'm not saying we should, but it's worth a try to see if we make the code better or worse.

First, let's map `lmap Ajax.printError` over the `Ajax.post` call:

```
test :: Effect Unit
test = launchAff_ do
  -- COMPILER ERROR!!
  lmap Ajax.printError <$> Ajax.post ResponseFormat.string
    "http://localhost:3000/"
    $ Just $ RequestBody.String $ encodeJSON teacher
```

The compiler error could be because we're not done yet or we have a legitimate problem. Hard to tell at this point.

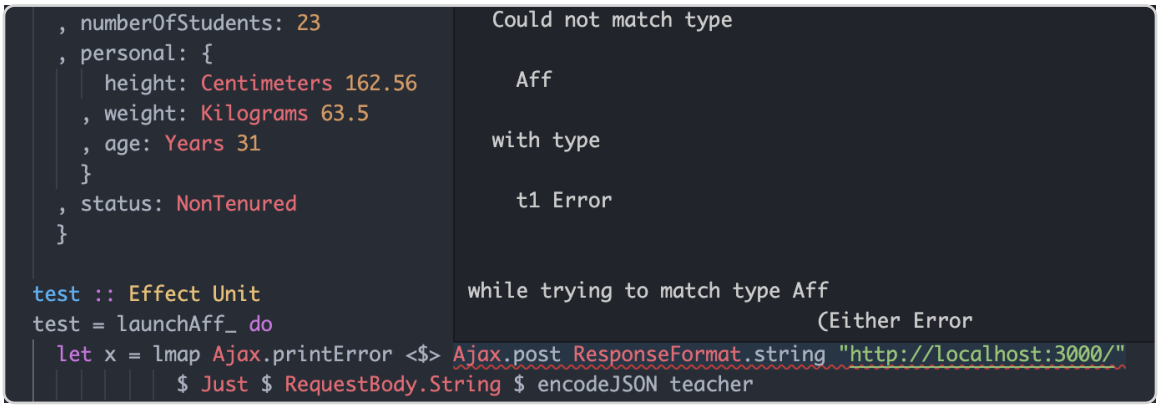
The way to fix that is to change our code to the following:

```
test :: Effect Unit
test = launchAff_ do
  -- COMPILER ERROR!!
  let x = lmap Ajax.printError <$> Ajax.post ResponseFormat.string
            "http://localhost:3000/"
            $ Just $ RequestBody.String $ encodeJSON teacher
  pure unit
```

This is sort of like using `undefined` without having to use `undefined`. We're simply setting some random Variable to the code we're writing to see if it compiles or not. The reason for this is because the compiler has NO preconceived ideas about what Type `x` should be. This is not true when we were using this code in the body of our `do` block.

And now, our `do` block only contains `pure unit`, which should make the compiler happy for the return Value of the `do` block.

And after all of this, we still have a compiler error:



Even with the red underlines, this error message makes no sense to me. But, experience has taught me when you have a lot of Binary Operators on a single line of code and it appears to be right, but the compiler is complaining, the solution is to add Parentheses and see if that fixes the problem.

I'm going to add Parentheses to the whole right-hand side of the `map` operator:

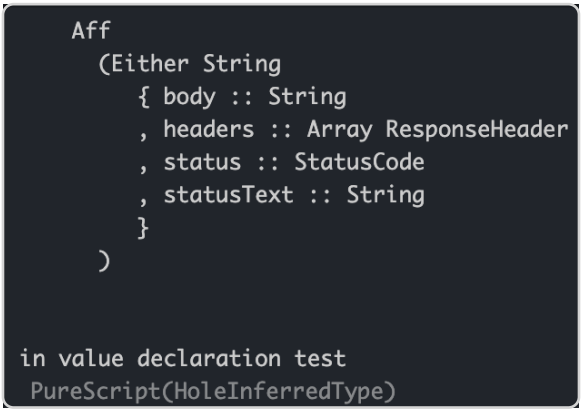
```
test :: Effect Unit
test = launchAff_ do
  let x = lmap Ajax.printError
          <$> (Ajax.post ResponseFormat.string "http://localhost:3000/"
          $ Just $ RequestBody.String $ encodeJSON teacher)
  pure unit
```

And the compiler errors are gone (only a warning that we're not using `x`). And now that we have our code in this configuration, we might as well take advantage of it by adding a Type Hole:

```
test :: Effect Unit
test = launchAff_ do
  let x = lmap Ajax.printError
          <$> (Ajax.post ResponseFormat.string "http://localhost:3000/"
          $ Just $ RequestBody.String $ encodeJSON teacher) :: ?x ❶
  pure unit
```

❶ Added a Type Hole.

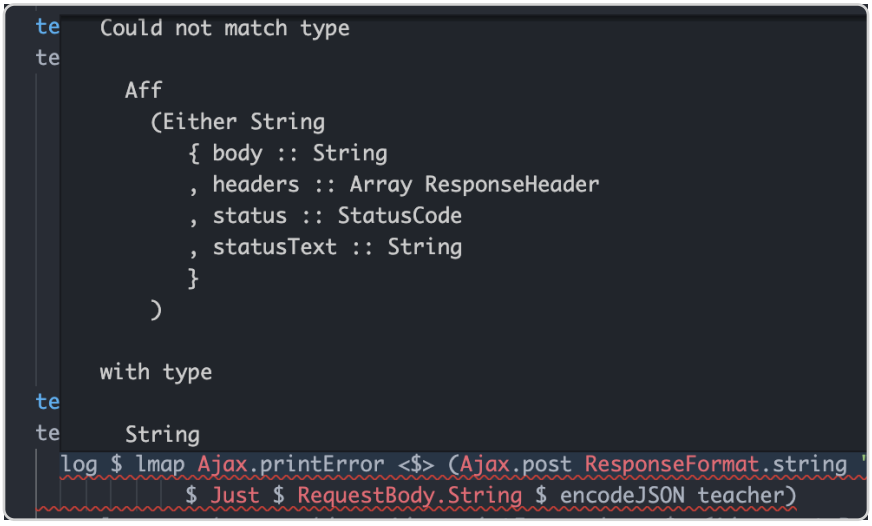
The warning for the Type Hole is:



So, based on this Type, we can see that our `Either` is totally showable. So, that means we can now remove `pure unit` and put our code back and then all we have to do is call `log` on it.

```
test :: Effect Unit
test = launchAff_ do
  -- COMPILER ERROR!!
  log $ lmap Ajax.printError
    <$> (Ajax.post ResponseFormat.string "http://localhost:3000/"
    $ Just $ RequestBody.String $ encodeJSON teacher)
```

And now we have a new compiler error:



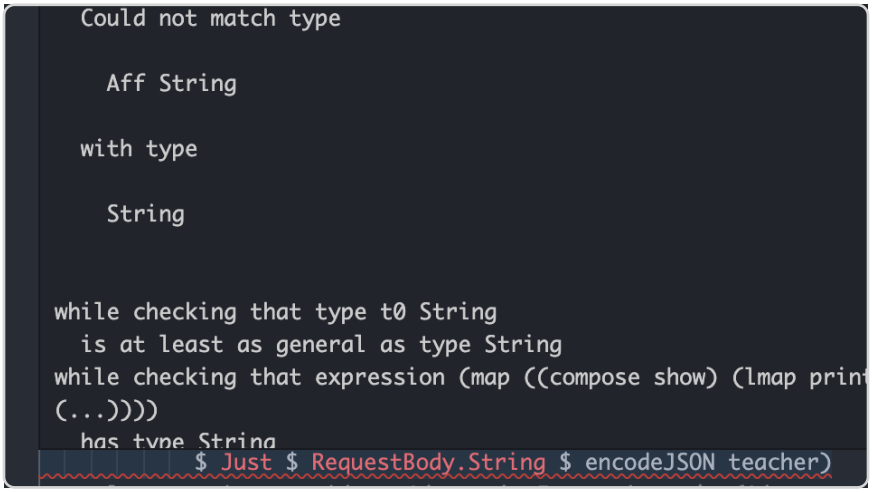
So, it looks like `log` has gotten an `Aff (Either …)` instead of the `String` it wants.

Looks like I jumped too fast to simply log our results. First the `Either` needs to be shown:

```
test :: Effect Unit
test = launchAff_ do
  -- COMPILER ERROR!!
  log $ show <<< lmap Ajax.printError
    <$> (Ajax.post ResponseFormat.string "http://localhost:3000/"
    $ Just $ RequestBody.String $ encodeJSON teacher)
```

Thanks to the Functor Laws, we can compose `show` with `lmap` to avoid 2 calls to `map`.

But we still have a problem:



And `log` still isn't quite getting the `String` it hoped for. The `String` is buried in the `Aff`.

What should we do to get at the `String`?

If you said `map`, you're close, but that will give us an `Aff (Aff String)`. What do we do when we see this?

You may think of `join`, but when you do think of that, you should instead see if `bind` will fix the problem that we created. And it will:

```
test :: Effect Unit
test = launchAff_ do
  log =<< show <<< lmap Ajax.printError ❶
    <$> (Ajax.post ResponseFormat.string "http://localhost:3000/"
    $ Just $ RequestBody.String $ encodeJSON teacher)
```

❶ Using the flipped version of `bind` because it's more readable.

Now, that was a fun exercise, but did we improve things. Let's compare it with the previous version:

```
test :: Effect Unit
test = launchAff_ do
  result <- Ajax.post ResponseFormat.string
    "http://localhost:3000/"
    $ Just $ RequestBody.String $ encodeJSON teacher
  log $ show $ lmap Ajax.printError result
```

Which do you like better?

Personally, I tend to lean toward the version with one line of code, but looking at this particular case, I think the version with 2 lines of code looks better. It's also easier to understand what's going on.

So, make sure you have the previous version in your module before moving on.

At this point, we are ready to run our code. Do so and see what you get.

What did you get?

This is what I got:

```
Left "There was a problem making the request: Cannot find module
'xhr2'\nRequire stack:\n-
/Users/charlesscalfani/dev/cscalfani/bookCode/chapter14RT/output/Affjax/foreig
n.js\n-
/Users/charlesscalfani/dev/cscalfani/bookCode/chapter14RT/output/Affjax/index.
js\n-
/Users/charlesscalfani/dev/cscalfani/bookCode/chapter14RT/output/Ch25a/index.j
s\n-
/Users/charlesscalfani/dev/cscalfani/bookCode/chapter14RT/output/Main/index.js
\n- /Users/charlesscalfani/dev/cscalfani/bookCode/chapter14RT/.spago/run.js")
```

The salient portion is before the Stack Trace:

```
Left "There was a problem making the request: Cannot find module 'xhr2'....
```

So what does that even mean?

What is `xhr2`?

How do we go about solving this?

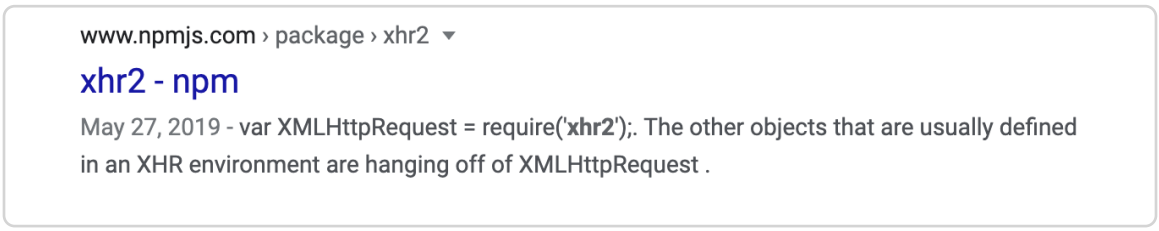
Let's search `Pursuit` for `xhr2`. Do that and see what you learn.

This is what I learned:



Okay, so let's Google it. Try it and see if you can figure out what's wrong.

Here's the first entry:



It's an NPM package. Another name for package in Node is module. So, click on that link and see what it does before returning here.

Here's what I noticed:



`XmlHttpRequest` is the original name Microsoft's Outlook Web Access team gave to what we currently refer to as `Ajax`.

This technology is now built in to all modern Web Browsers. But, it's clearly not native to Node otherwise this Node package wouldn't exist.

Let's do one more check before we install this package. Let's look at the docs for `Affjax`. Go to `Pursuit` and read the documentation on the main page of `purescript-affjax`.

You can get there by searching for `purescript-affjax` and then clicking on the package link that'll be first in the search results.

Here's what I saw:

A library taking advantage of `purescript-aff` to enable pain-free asynchronous AJAX requests and response handling.

Installation

```
bower install purescript-affjax
```

If you are intending to use the library in a Node.js setting rather than the browser, you will need an additional dependency from `npm`:

```
npm install xhr2
```

Okay, now this confirms it. We need to run:

```
npm install --save xhr2 ❶
```

❶ I added the `--save` to make sure `npm` updates our `package.json` file.

Please do install `xhr2` and rerun the program. What did you get?

Did you get:

```
(Left "There was a problem making the request: request failed")
```

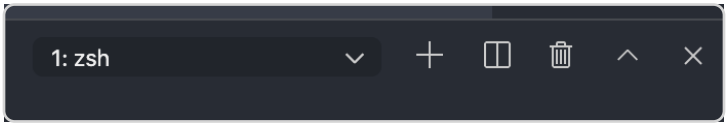
Well, now we're past that other error, but now what?

It turns out that we never ran the server. Run it from a terminal prompt via:

```
node src/echo-server.js ❶
```

❶ Notice that the `echo-server.js` file is under the `src` directory.

Then from another terminal prompt run the program. By the way, to do this is VSCode, you can click on the plus icon in the Terminal Window tool bar in the upper right-hand corner:



You can use the pull-down menu to switch between terminal windows.

You can also just run the server outside of VSCode or whatever editor that you're using by using the normal Terminal program.

What do you get when you run it?

This is what I get:

```
(Right { body: "{\n  \"sutats\": {\n    \"gat\": \"NonTenured\"\n  },\n
\"lanosrep\": {\n    \"thgiew\": 63.5,\n    \"thgieh\": 162.56,\n    \"ega\":
31\n  },\n  \"stnedutSfOrebmun\": 23,\n  \"sedarg\": [\n    {\n      \"gat\":
\"Preschool\"\n    },\n    {\n      \"gat\": \"Kindergarten\"\n    },\n    {\n
\"stnetnoc\": 1,\n      \"gat\": \"Grade\"\n    }\n  ]\n}", headers:
[(ResponseHeader "content-type" "text/plain"),(ResponseHeader "date" "Sat, 12
Jun 2021 20:23:19 GMT"),(ResponseHeader "connection" "keep-alive"
),(ResponseHeader "keep-alive" "timeout=5"),(ResponseHeader "content-length"
"300")], status: (StatusCode 200), statusText: "OK" })
```

Note that your output may differ slightly depending on the verion of `Node` that you're running. The difference being in the `RepsonseHeader`.

This is the full response. We can see that from the `ResponseHeader` data and `StatusCode`.

If we just want the body, we can change our program to only display that. Make that change before reading on.

Here's how we can get just the body:

```
import Data.Bifunctor (bimap) ❶

test :: Effect Unit
test = launchAff_ do
  result <- Ajax.post ResponseFormat.string
    "http://localhost:3000/"
    $ Just $ RequestBody.String $ encodeJSON teacher
  log $ show $ bimap Ajax.printError _.body result ❷
```

❶ Replaced `lmap` with `bimap`.

❷ Changed `lmap` to `bimap` so we could map the right-hand side with a Function to get just the `body` from the `Response` Record.

Now, run the program and you should get:

```
(Right "{\n  \"sutats\": {\n    \"gat\": \"NonTenured\"\n  },\n  \"lanosrep\":
{\n    \"thgiew\": 63.5,\n    \"thgieh\": 162.56,\n    \"ega\": 31\n  },\n
\"stnedutSfOrebmun\": 23,\n  \"sedarg\": [\n    {\n      \"gat\": \"
Preschool\"\n    },\n    {\n      \"gat\": \"Kindergarten\"\n    },\n    {\n
\"stnetnoc\": 1,\n      \"gat\": \"Grade\"\n    }\n  ]\n}")
```

Copy out just the JSON part (the part between the outermost quotes) and open up a new file in VSCode and paste it. Then, replace `\n` with an empty string. Then replace `\` with an empty string.

Next, set the file type using the status bar at the bottom right of the window by clicking on `Plain Text`:



In the dialog box, type `json` and then select JSON.

Next, bring up the Command Palette via Shift-Cmd-P or Shift-Ctrl-P on Windows and type `format`. Then pick `Format Document`.

You should get:

```
{
  "sutats": {
    "gat": "NonTenured"
  },
  "lanosrep": {
    "thgiew": 63.5,
    "thgieh": 162.56,
    "ega": 31
  },
  "stnedutSfOrebmun": 23,
  "sedarg": [
    {
      "gat": "Preschool"
    },
    {
      "gat": "Kindergarten"
    },
    {
      "stnetnoc": 1,
      "gat": "Grade"
    }
  ]
}
```

Notice how every key is reversed from what we sent it. That's what the Echo Server does as its response.

The next thing we'd like to do is decode this. But how might we do this? Think about it for a minute before reading on. See if you can devise a solution to this problem before you read mine.

I can think of 3 ways to do this. We're going to do them all. Here are the 3 ways:

- Model the Records with reversed field names.
- Use `Argonaut` and write our decoder explicitly.
- Steal the code from the Echo Server that takes the Object and reverses its key names and call it from PureScript.

We're going to do all of these, but we'll put off the Javascript solution until we get to the FFI Chapters.

Before we embark on this journey, let's first make all our Types showable to help us log them to the console.

Remember, you don't have to make Records showable because all Records are as long as all their fields are, e.g. the Type Alias `Teacher` won't need a `Show` Instance.

Do this change before reading on.

Here are the additions:

```
import Data.Show.Generic (genericShow) ❶

instance showCentimeters :: Show Centimeters where
  show = genericShow

instance showKilograms :: Show Kilograms where
  show = genericShow

instance showYears :: Show Years where
  show = genericShow

instance showGPA :: Show GPA where
  show = genericShow

instance showGrade :: Show Grade where
  show = genericShow

instance showTeachingStatus :: Show TeachingStatus where
  show = genericShow
```

❶ Import `genericShow`.

25.2.2. Decoding the Response Modeled with Reversed Names

Go ahead and model the Records with reversed field names, i.e. `Personal`, `Student` and `Teacher`. Name the new Type Aliases, `ReversedPersonal`, `ReversedStudent` and `ReversedTeacher`.

Do this before reading on.

Here are those new Type Aliases:

```
type ReversedPersonal =
  { thgieh :: Centimeters
  , thgiew :: Kilograms
  , ega :: Years
  }

type ReversedStudent =
  { edarg :: Grade
  , rehcaet :: ReversedTeacher
  , apg :: GPA
  , lanosrep :: ReversedPersonal
  }

type ReversedTeacher =
  { sedarg :: Array Grade
  , stnedutSfOrebmun :: Int
  , lanosrep :: ReversedPersonal
  , sutats :: TeachingStatus
  }
```

Now, we can write code to decode the response from the Echo Server.

First, comment out the following lines of code:

```
-- import Data.Bifunctor (bimap) ❶

test :: Effect Unit
test = launchAff_ do
  result <- Ajax.post ResponseFormat.string
    "http://localhost:3000/"
    $ Just $ RequestBody.String $ encodeJSON teacher
  -- log $ show $ bimap Ajax.printError _.body result ❷
  pure unit ❸
```

❶ We no longer need `bimap`. You can delete this import.

❷ We are going to process `result`, so we don't need this line.

❸ Added temporarily to make it compile otherwise you get the following error: `The last statement in a 'do' block must be an expression, but this block ends with a binder.`

We're going to ignore the compiler warnings for now.

Now, before we can write code that takes `result` and decodes it on the successful Ajax response case, we should make sure we fully understand what type `result` is.

Go ahead and add `result` into `test` in such a way that you can add a Type Hole and get the compiler will tell us `result's` Type. Try it before reading the solution.

The easiest way to do this is to just add a dummy Variable in a `let` expression:

```
test :: Effect Unit
test = launchAff_ do
  result <- Ajax.post ResponseFormat.string
    "http://localhost:3000/"
    $ Just $ RequestBody.String $ encodeJSON teacher
  let x = result :: ?x ❶
  pure unit
```

❶ Type Hole is the most important part. Note that `::?x` is a syntax error. You must have the space between the `:` and the `?`.

From the Type Hole error message, we get the following Type:

```
Either Error
  { body :: String
  , headers :: Array ResponseHeader
  , status :: StatusCode
  , statusText :: String
  }
```

The `Error` is from `Affjax` and the Record is the Ajax `Response` from the Echo Server. So, we can access `body` to get the `String` to decode.

But it's all wrapped up in an `Either`.

Go ahead and add a `case` expression to decode the `body` from the `Right` case before reading on. Also, don't forget to delete the `let` with the Type Hole.

We'll do this one step at a time. We'll start with the `case` expression to get to the `Right` of `result`:

```
import Data.Either (Either(..)) ❶

test :: Effect Unit
test = launchAff_ do
  result <- Ajax.post ResponseFormat.string
    "http://localhost:3000/"
    $ Just $ RequestBody.String $ encodeJSON teacher
  case result of
    Left err -> Ajax.printError err ❷
    Right { body } -> ??? ❸ ❹
```

❶ Import `Either's` Data Constructors.

❷ Again, we're using `Ajax.printError` to convert `Error` to a `String`.

❸ We get the Ajax `Response` in the `Right` case and destructure the Record to gain access to `body`.

❹ Removed `pure unit` since we don't need it anymore.

Next, we call `decodeJSON`:

```
import Foreign.Generic (encodeJSON, genericEncode, genericDecode
                      , defaultOptions, decodeJSON) ❶

test :: Effect Unit
test = launchAff_ do
  result <- Ajax.post ResponseFormat.string
    "http://localhost:3000/"
    $ Just $ RequestBody.String $ encodeJSON teacher
  case result of
    Left err -> Ajax.printError err
    Right { body } -> decodeJSON body :: ??? ❷
```

❶ Added `decodeJSON` to import list.

❷ We know from last Chapter that we have to explicitly tell the compiler what Type it's decoding, unless we're going to use the results of the decode such that the compiler can infer it. In our case, we're just going to log it, which means the compiler won't be able to infer its Type.

So, the big question here is what Type is `decodeJSON body`. Use `Pursuit` to figure that out. Keep in mind that we're passing `decodeJSON` a `String`. Then return here.

Here's what I found:

```
decodeJSON :: ∀ a. Decode a => String -> F a
```

So, the Type is `F ReversedTeacher` since we sent `teacher` to the Echo Server, who will reverse our `keys` and send the same JSON back to us, which should match `ReversedTeacher` if we did everything right.

We can just add `_ ReversedTeacher` as the Type since the compiler can figure out the `F` part and, besides, this way we don't have import the `Foreign` module just for this:

```
test :: Effect Unit
test = launchAff_ do
  result <- Ajax.post ResponseFormat.string
    "http://localhost:3000/"
    $ Just $ RequestBody.String $ encodeJSON teacher
  case result of
    -- COMPILER ERROR!!
    Left err -> Ajax.printError err
    Right { body } -> decodeJSON body :: _ ReversedTeacher
```

We're going to ignore the compiler error at this point since we're not finished. Looking at the `Left` case we have a `String`. So, that means that the `Right` case must also return a `String`.

And if that's the case, we can log the `case` expression since it evaluates to a `String`:

```
test :: Effect Unit
test = launchAff_ do
  result <- Ajax.post ResponseFormat.string
    "http://localhost:3000/"
    $ Just $ RequestBody.String $ encodeJSON teacher
  log $ case result of ❶
    Left err -> Ajax.printError err
    -- COMPILER ERROR!!
    Right { body } -> decodeJSON body :: _ ReversedTeacher
```

❶ Added call to `log`.

Now, we're not out of the woods yet. We still have to finish the `Right` case. We know that we have an `F ReversedTeacher`, but what was `F` again?

If you don't remember, then go look it up in `Pursuit` and return here.

Here's the Type Alias:

```
type F = Except MultipleErrors
```

It's just `Except` with `MultipleErrors` as it's error Type. So, we'll need to run that. Remember that `Except`

is just `ExceptT` with `Identity` on the base of the Monad Stack.

Let's add the `runExcept` call:

```
import Control.Monad.Except (runExcept) ❶

test :: Effect Unit
test = launchAff_ do
  result <- Ajax.post ResponseFormat.string
    "http://localhost:3000/"
    $ Just $ RequestBody.String $ encodeJSON teacher
  log $ case result of
    Left err -> Ajax.printError err
    Right { body } ->
    -- COMPILER ERROR!!
      runExcept (decodeJSON body :: _ ReversedTeacher) ❷ ❸
```

❶ Import `runExcept`.

❷ Added `runExcept`.

❸ Added Parentheses to keep the Type specification local to just `decodeJSON body`.

The compiler error is still complaining that we don't have a `String` yet. But what do we have? Go look up `runExcept` to see what it returns and then continue reading.

Here's `runExcept's` Type Signature according to `Pursuit`:

```
runExcept :: ∀ e a. Except e a -> Either e a
```

This means that we're going to need another `case` expression. Stop reading at this point and see if you can finish the code.

Here's the second `case` expression:

```
test :: Effect Unit
test = launchAff_ do
  result <- Ajax.post ResponseFormat.string
    "http://localhost:3000/"
    $ Just $ RequestBody.String $ encodeJSON teacher
  log $ case result of ❶
    Left err -> Ajax.printError err
    Right { body } ->
      case runExcept (decodeJSON body :: _ ReversedTeacher) of ❶
        Left err -> ???
```

❶ Added the `case`.

At this point, we have `MultipleErrors` in the `Left` case. Can we just call `show` on it? Go look up `MultipleErrors` in `Pursuit` to see if it has a `Show` Instance and then return here.

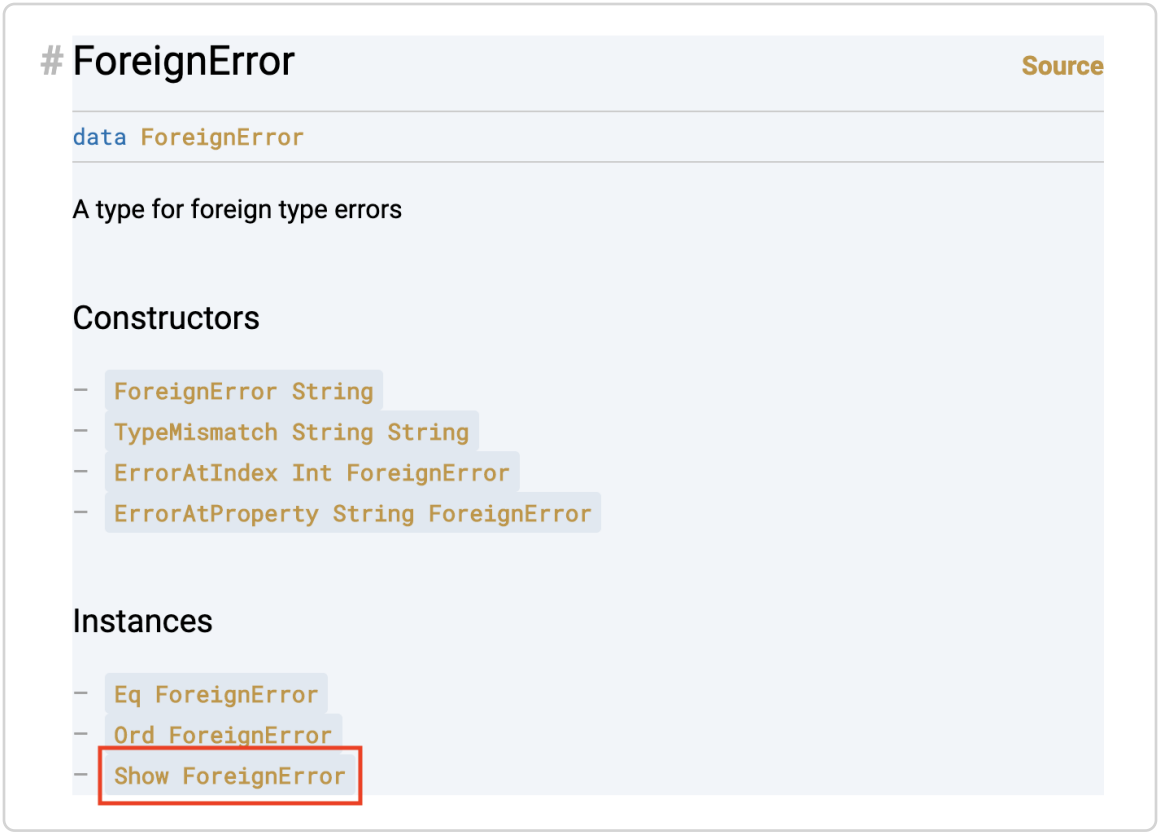
Here's what I found:

```
type MultipleErrors = NonEmptyList ForeignError
```

This says that we have a non-empty list of `ForeignError`. So, we know `List`, which is the bases for `NonEmptyList`, has a `Show` Instance but it has a Constraint on its `a`, that it too must be showable.

That means that we have to determine if `ForeignError` is showable. Check `Pursuit` and then return here.

We can see from the list of Instances in `Pursuit` that we can show this:



ForeignError Source

data ForeignError

A type for foreign type errors

Constructors

- ForeignError String
- TypeMismatch String String
- ErrorAtIndex Int ForeignError
- ErrorAtProperty String ForeignError

Instances

- Eq ForeignError
- Ord ForeignError
- Show ForeignError

So, let's call `show` in the `Left` case:

```
test :: Effect Unit
test = launchAff_ do
  result <- Ajax.post ResponseFormat.string
    "http://localhost:3000/"
    $ Just $ RequestBody.String $ encodeJSON teacher
  log $ case result of
    Left err -> Ajax.printError err
    Right { body } ->
      case runExcept (decodeJSON body :: _ ReversedTeacher) of
        Left err -> show err ❶
        Right ???
```

❶ Showing the `Error` since it has a `Show` Instance.

For the `Right` case, we have the result of successfully decoding `ReversedTeacher`, and since we made all of our Types showable earlier, we can just show it:

```
test :: Effect Unit
test = launchAff_ do
  result <- Ajax.post ResponseFormat.string
    "http://localhost:3000/"
    $ Just $ RequestBody.String $ encodeJSON teacher
  log $ case result of
    Left err -> Ajax.printError err
    Right { body } ->
      case runExcept (decodeJSON body :: _ ReversedTeacher) of
        Left err -> show err
        Right reversedTeacher -> show reversedTeacher ❶
```

❶ Showing `ReverseTeacher` since it has a `Show` Instance.

And we're done and the compiler error is gone.

Run your program and see what you get.

This is what I get:

```
(NonEmptyList (NonEmpty (ErrorAtProperty "sutats" (ErrorAtProperty "tag"
(TypeMismatch "String" "Undefined"))) ((ErrorAtProperty "sutats"
(ErrorAtProperty "tag" (TypeMismatch "String" "Undefined"))) :
(ErrorAtProperty "sutats" (ErrorAtProperty "tag" (TypeMismatch "String"
"Undefined"))) : (ErrorAtProperty "sutats" (ErrorAtProperty "tag"
(TypeMismatch "String" "Undefined"))) : Nil)))
```

Let's pull out only one of the errors and look at it:

```
(ErrorAtProperty "sutats" (ErrorAtProperty "tag" (TypeMismatch "String"
"Undefined")))
```

This tells us that it expected `tag` to be a `String` but was `Undefined`. The only thing I can figure from this is that `tag` is missing.

But, that shouldn't surprise anyone since the Echo Server reversed `tag` before responding back to us. Why is it still looking for `tag`?

Well, that's because of `defaultOptions`:

```
-- |
-- | - Represent sum types as records with `tag` and `contents` fields
-- | - Unwrap single arguments
-- | - Don't unwrap single constructors
-- | - Use the constructor names as-is
-- | - Use the field names as-is
defaultOptions :: Options
defaultOptions =
  { sumEncoding:
      TaggedObject
        { tagFieldName: "tag" ❶
        , contentsFieldName: "contents" ❷
        , constructorTagTransform: identity
        }
  , unwrapSingleConstructors: false
  , unwrapSingleArguments: true
  , fieldTransform: identity
  }
```

❶ Here's where the `tag` key is defined for the `tagFieldName`. We'll need to reverse this.

❷ While we're looking at `tag`, we see that `contents` is going to have a similar problem and will need to also be reversed.

Before moving on, create a new set of Options called `decodeOptions`. Use `defaultOptions` as your starting point and use the Record Update syntax to override the `sumEncoding`. Make sure to reverse both `tag` and `contents`, then read on.

Here's the code for `decodeOptions`:

```
import Foreign.Generic.Class (  class Decode ,class Encode
                              , Options, SumEncoding(..)) ❶

decodeOptions :: Options
decodeOptions = defaultOptions ❷
  { sumEncoding = TaggedObject
    { tagFieldName: "gat" ❸
    , contentsFieldName: "stnetnoc" ❹
    , constructorTagTransform: identity
    }
  }
```

❶ Import `Options` and `SumEncoding's` Data Constructors to be able to use `TaggedObject`.

❷ Start with `defaultOptions` replacing just `sumEncodings`. This uses the Record Update syntax.

❸ Reversed `tag`.

❹ Reversed `contents`.

Notice how `sumEncoding` has an `=` after it. That's because we're updating this Field in `defaultOptions`. Actually, we're updating it with a new copy and returning a new copy of this Record. Also, notice that `tagFieldName`, `contentsFieldName` and `constructorTagTransform` all have `:` after them. That's because this Record is being constructed from scratch.

Next, we'll need to change all of decoders to use `decodeOptions` instead of `defaultOptions`. Make that change now being careful NOT to change any encoders. They should still use `defaultOptions`.

Do this before reading on.

Here are those changes:

```
instance decodeGrade :: Decode Grade where
  decode = genericDecode decodeOptions

instance decodeTeachingStatus :: Decode TeachingStatus where
  decode = genericDecode decodeOptions
```

Notice that we only need to change our Sum Types. That should make sense since the only change to `defaultOptions` was `sumEncoding`.

Now, when you run the program, you should get:

```
{ lanosrep: { ega: (Years 31), thgieh: (Centimeters 162.56), thgiew:
(Kilograms 63.5) }, sedarg: [Preschool,Kindergarten,(Grade 1)],
stnedutSfOrebmun: 23, sutats: NonTenured }
```

Looking back on the code for `test`, you may be thinking that these nested `case` expressions are a blight on our code and I'd agree. So, we should move the processing out of the `test` code into a Function that has the ugly `case` expressions.

We're going to leave our working code alone for now and copy and paste from `test` into a Function called `processAjaxResult`.

Start by copying the outer `case` expression being sure to include the inner `case` into `processAjaxResult` as such:

```
processAjaxResult = case result of -- COMPILER ERROR!
  Left err -> Ajax.printError err
  Right { body } ->
    case runExcept (decodeJSON body :: _ ReversedTeacher) of
      Left err -> show err
      Right reversedTeacher -> show reversedTeacher
```

And once you save, you'll see a compiler error on `result`. It's clear that we need to pass in the `result`, so do that and then continue reading.

Adding `result` as the first Parameter:

```
processAjaxResult result = case result of
  Left err -> Ajax.printError err
  Right { body } ->
    case runExcept (decodeJSON body :: _ ReversedTeacher) of
      Left err -> show err
      Right reversedTeacher -> show reversedTeacher
```

Wait. We don't need to add a named Variable. We can simply go Point-free and use a wildcard:

```
processAjaxResult = case _ of
  Left err -> Ajax.printError err
  Right { body } ->
    case runExcept (decodeJSON body :: _ ReversedTeacher) of
      Left err -> show err
      Right reversedTeacher -> show reversedTeacher
```

Now, the compiler wants to suggest to us that we add a Type Signature. Go ahead and follow its advice but make sure you give the Types reasonable names. Do this before reading on.

Here's the first step in that process:

```
-- COMPILER ERROR!!
processAjaxResult
  :: ∀ t129
  .  Either Error { body :: String | t129 }
  -> String
```

The compiler is complaining about `Error`. That's because we fully qualified the `Affjax` import as `Ajax`. So,

I'm going to add that qualifier:

```
processAjaxResult
  :: ∀ t129
  .  Either Ajax.Error { body :: String | t129 } ❶
  -> String
```

❶ Added `Ajax.` prefix to qualify which `Error`.

Now the compiler is happy, but we're not. What does `t129` stand for. By the way, your number may or may not be `94`. It doesn't matter since your number is equally unhelpful.

We can see by how `t129` is used that its a Row Type. Tradition dictates that we call this `r`:

```
processAjaxResult
  :: ∀ r
  .  Either Ajax.Error { body :: String | r }
  -> String
```

But wait. We know that the Record is simply a `Response a` where `a` is a `String`:

```
processAjaxResult ❶
  :: Either Ajax.Error (Ajax.Response String) ❷
  -> String
```

❶ Removed the `∀` since we're not using `r` anymore.

❷ Fully qualified `Response` since `Affjax` is imported qualified.

Now, this is great but it's not as general as I would like. Take a look at both the Type Signature and the code for `processAjaxResult` and see if you can find all the places where we could generalize before reading my take.

Here's my take doing the analysis:

```
processAjaxResult
  :: Either Ajax.Error (Ajax.Response String)
  -> String
processAjaxResult = case _ of
  Left err -> Ajax.printError err
  Right { body } ->
    case runExcept (decodeJSON body :: _ ReversedTeacher) of ❶
      Left err -> show err
      Right reversedTeacher -> show reversedTeacher ❷ ❸
```

❶ We can only decode `ReversedTeacher`. This could be any `a`.

❷ If we generalize what we decode as some `a`, then that `a` had better have a `Show` Instance.

❸ We probably should rename `reversedTeacher` to something more generic once we can decode any `a`.

Remember from last Chapter, there's no `a` in the Type Signature that we can work with. So, add it as a `Proxy` and make the aforementioned generalizations before advancing forward.

First change to make is to add the `Proxy` Parameter:

```
import Type.Proxy (Proxy) ❶

processAjaxResult
  :: ∀ a
  .  Show a ❷
  => Proxy a ❸
  -> Either Ajax.Error (Ajax.Response String)
  -> String
processAjaxResult _ = case _ of ❹
  Left err -> Ajax.printError err
  -- COMPILER ERROR!!
  Right { body } -> case runExcept (decodeJSON body :: _ a) of ❺
    Left err -> show err
    Right reversedTeacher -> show reversedTeacher
```

❶ Added import for `Proxy`.

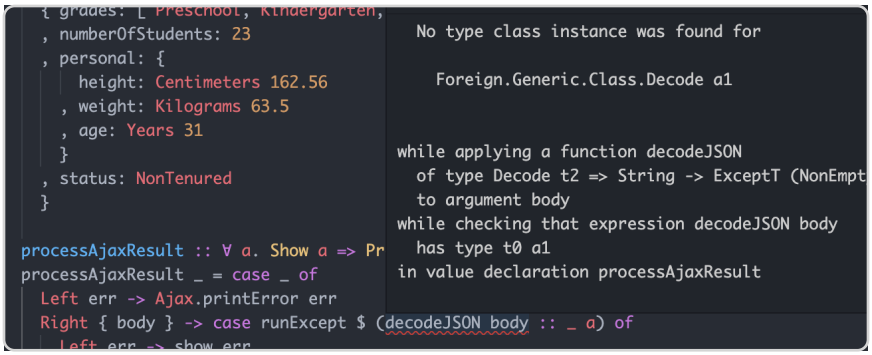
❷ Constrain `a` to be showable.

❸ Added `Proxy` Parameter Type.

❹ Added a don't-care Parameter for the `Proxy`.

❺ Decode `_ a` instead of `_ ReverseTeacher`.

Here's the compiler error:



Well, I remembered to make `a` showable, but forgot that it must have a `Decode` Instance too since we're calling `decodeJSON` and it's expecting it to decode a Value of Type `_ a`.

Let's fix that:

```
processAjaxResult
  :: ∀ a
  .  Show a
  => Decode a ❶
  => Proxy a
  -> Either Ajax.Error (Ajax.Response String)
  -> String
processAjaxResult _ = case _ of
  Left err -> Ajax.printError err
  Right { body } -> case runExcept (decodeJSON body :: _ a) of
    Left err -> show err
    Right reversedTeacher -> show reversedTeacher
```

❶ Added `Decode` Constraint.

And now the compiler is happy. But, I'm not. I see one more little thing that we could improve.

Notice the Variable name `reversedTeacher`. That's not accurate anymore. It's not just a reversed version of `Teacher` anymore. It could be any reversed JSON Object.

Let's rename it to `reversedRecord`:

```
processAjaxResult
  :: ∀ a
  .  Show a
  => Decode a
  => Proxy a
  -> Either Ajax.Error (Ajax.Response String)
  -> String
processAjaxResult _ = case _ of
  Left err -> Ajax.printError err
  Right { body } -> case runExcept (decodeJSON body :: _ a) of
    Left err -> show err
    Right reversedRecord -> show reversedRecord ❶
```

❶ Renamed `reverseTeacher` to `reversedRecord`.

At this point, stop to make sure your code matches this then continue.

Next, modify the test code to call `processAjaxResult` before reading any further.

Here's the change to `test` that leverages `processAjaxResult`:

```
import Type.Proxy (Proxy(..)) ❶

test :: Effect Unit
test = launchAff_ do
  result <- Ajax.post ResponseFormat.string
    "http://localhost:3000/"
    $ Just $ RequestBody.String $ encodeJSON teacher
  log $ processAjaxResult (Proxy :: _ ReversedTeacher) result ❷
```

❶ Added `(..)` to import the `Proxy` Data Constructor.

❷ Give it a `Proxy` with a Type of `Proxy ReversedTeacher`, but using `_` in place of `Proxy` since the compiler can figure that part out for itself and I want to keep this as readable as possible.

Just for grins, run the test and make sure you get:

```
 lanosrep: { ega: (Years 31), thgieh: (Centimeters 162.56), thgiew: (Kilograms
63.5) }, sedarg: [Preschool,Kindergarten,(Grade 1)], stnedutSfOrebmun: 23,
sutats: NonTenured }
```

25.2.3. Decoding the Response Using `Argonaut`

Copy all of your code and paste it into a new file called `Ch25b`. We're going to work in this file so make sure you close `Ch25a`.

Also change your `Main` to call `Ch25b's` test Function.

Modify `Ch25b's` module name to be `Ch25b`.

We'll need to add the `Argonaut` library to our `spago.dhall`. Go to `Pursuit` to figure out what needs to be added to your `spago.dhall` file. Then add it and build. Then reading on.

The following needs to be updated in the `spago.dhall` file:

```
{-
Welcome to a Spago project!
You can edit this file as you like.
-}
{ name = "my-project"
, dependencies =
  [ "aff"
  , "aff-bus"
  , "affjax"
  , "argonaut" (1)
  , "arrays"
  , "avar"
  , "bifunctors"
  , "console"
  , "contravariant"
  , "control"
  , "datetime"
  , "effect"
  , "either"
  , "exceptions"
  , "foreign-generic"
  , "foldable-traversable"
  , "integers"
  , "lists"
  , "maybe"
  , "newtype"
  , "nonempty"
  , "prelude"
  , "profunctor"
  , "psci-support"
  , "random"
  , "strings"
  , "tailrec"
  , "transformers"
  , "tuples"
  , "unfoldable"
  , "unicode"
  , "debug"
  ]
, packages = ./packages.dhall
, sources = [ "src/**/*.purs", "test/**/*.purs" ]
}
```

❶ Once again, placing it here to maintain alphabetic order.

Then just to make sure we did this right, we should run:

```
npx spago build
```

Now, we're ready to start explicitly write decoders to handle the Echo Server's propensity to reverse our keys. We'll start by deleting all of the `Reversed` versions of our Records.

Please make sure you're editing in `Ch25b` before you remove `ReversedPersonal`, `ReversedStudent` and `ReversedTeacher`.

When you do this, the last line of test will no longer compile, so comment it out and replace it with `pure unit` just for now so it compiles:

```
test :: Effect Unit
test = launchAff_ do
  -- COMPILER WARNING!!
  result <- Ajax.post ResponseFormat.string
    "http://localhost:3000/"
    $ Just $ RequestBody.String $ encodeJSON teacher
  -- log $ processAjaxResult (Proxy :: _ ReversedTeacher) result
  pure unit
```

In fact, comment out the `Ajax.post` line for now since we're going to break it too:

```
test :: Effect Unit
test = launchAff_ do
  -- result <- Ajax.post ResponseFormat.string
  --   "http://localhost:3000/"
  --   $ Just $ RequestBody.String $ encodeJSON teacher
  -- log $ processAjaxResult (Proxy :: _ ReversedTeacher) result
  pure unit
```

At this point, the compiler is warning us that our import statements have redundant imports. We'll fix those later since we're going to be constantly breaking them.

Since we no longer need `decodeOptions`, we can remove it. When you do, take special note to all the places where our code is now broken. These are all the places that we'll need to change from using `Foreign` to decode JSON to use `Argonaut`.

There are 2 places that you should get a compiler error:

```
instance decodeGrade :: Decode Grade where
  decode = genericDecode decodeOptions -- COMPILER ERROR!!

instance decodeTeachingStatus :: Decode TeachingStatus where
  decode = genericDecode decodeOptions -- COMPILER ERROR!!
```

We'll have to write our own decoders for these. But this isn't the only change we need to make. Our Records are Type Aliases and we won't be able to write custom `DecodeJson` Instances without them being proper Types.

So, we should change those now. Go ahead and change ONLY `Student` to be a `newtype` before reading on.

We have to add a Data Constructor to `Student`:

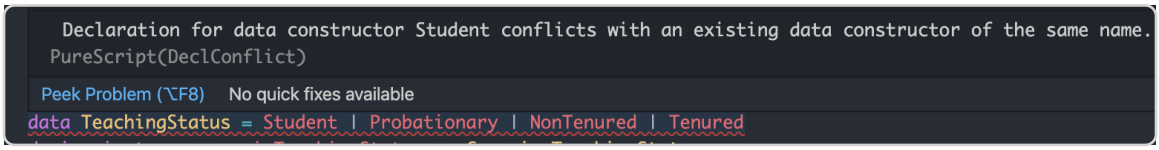
```
newtype Student = Student
  { grade :: Grade
  , teacher :: Teacher
  , gpa :: GPA
  , personal :: Personal
  }
```

This unfortunately gives us a compiler error on:

```
data TeachingStatus = Student | Probationary | NonTenured | Tenured
```

Can you see why?

Here's the compiler error:



Looks like we've defined `Student` twice. Once as a Data Constructor for `Student` and once as a Data Constructor for `TeachingStatus`.

We're going to have to rename one of them and `TeachingStatus` makes the most sense to rename. This may not always be possible. You may be making this change to an existing program where you've persisted JSON data to a Database with `TeachingStatus` already.

In that case, I'd suggest moving the definition of `Student` out into its own file and fully qualifying it when you import it. Then there wouldn't be a conflict.

But since we're just learning here, we're going change `Student` in `TeachingStatus`:

```
data TeachingStatus
  = StudentTeacher ❶
  | Probationary
  | NonTenured
  | Tenured
```

❶ Changed `Student` to `StudentTeacher`.

Please update your code to reflect this change.

Make the same change we did to `Student` for `Personal` and `Teacher` to make them proper Types. Then read on.

Here are those changes:

```
newtype Personal = Personal ❶ ❷
  { height :: Centimeters
  , weight :: Kilograms
  , age :: Years
  }

newtype Teacher = Teacher ❶ ❷
  { grades :: Array Grade
  , numberOfStudents :: Int
  , personal :: Personal
  , status :: TeachingStatus
  }
```

❶ Changed `type` to `newtype`.

❷ Added Data Constructor.

So, we now need to write a `DecodeJson` Instance for `Personal`, `Student` and `Teacher`. We'll also have to write instances for `Grade` and `TeachingStatus` since those Sum Types currently have `Foreign` decoders, i.e. they have a `Decode` Instance.

Let's start by deleting the `Decode` Instances for `Grade` and `TeachingStatus`. When you do that, you'll get a new compiler error.

The new error is on `teacher`. Look at it carefully:

```
teacher :: Teacher
teacher =
  { grades: [ Preschool, Kindergarten, Grade 1 ]
  , numberOfStudents: 23
  , personal: {
      height: Centimeters 162.56
    , weight: Kilograms 63.5
    , age: Years 31
    }
  , status: NonTenured
  }
```

Notice how `personal` doesn't have a Data Constructor. It didn't used to need one, but we added a Data Constructor to `Personal` when we made it a real Type. So, we need to add it here too.

Turns out that `teacher` doesn't have a Data Constructor either.

Go ahead and fix this before reading on.

Here's the fix:

```
teacher :: Teacher
teacher = Teacher ❶
  { grades: [ Preschool, Kindergarten, Grade 1 ]
  , numberOfStudents: 23
  , personal: Personal { ❷
      height: Centimeters 162.56
    , weight: Kilograms 63.5
    , age: Years 31
    }
  , status: NonTenured
  }
```

❶ Added `Teacher` Data Constructor.

❷ Added `Personal` Data Constructor.

With Type Aliases, we can just use Records, but now, with `newtype`, we need to pass those Records to the proper Data Constructors.

Next, we'll code an `Argonaut` decoder for `Grade`, i.e. a `DecodeJson` Instance.

Note the different names in the different libraries. `Foreign's` Typeclass is `Decode`, whereas `Argonaut's` is called `DecodeJson`. Same goes for `Encode` to `EncodeJson`.

But, before we write any decoder, we should know what format the encoder produces. Don't forget that we're still using the encoders from the `Foreign` library.

We can add code to our test Function to log out the JSON for different Types as we code.

We'll start by looking at `Grade's` definition:

```
data Grade = Preschool | Kindergarten | Grade Int | High Int | College Int
```

We have 2 different types of Data Constructors. Ones with no Parameters and ones with `Int` Parameters. We should add both types to our test code, encode them and log them to the console so we can see their JSON formats.

Add the following to your test code:

```
log $ encodeJSON Preschool
log $ encodeJSON $ Grade 3
```

Now, run the test code to see what it prints out.

Here's what I get:

```
{"tag":"Preschool"}
{"contents":3,"tag":"Grade"}
```

By the way, if you did not get this, you may have forgotten to change your `Main` to call `Ch25b's` test Function.

From this, we can see that our decode code will have to deal with `tag` backwards or `gat` and the same with `contents`.

Keeping this in mind, code the `DecodeJson` Instance for `Grade`. Don't forget to use `Pursuit` to help you. Go as far as you can get. I don't expect you to be able to finish. Once you hit a roadblock, come back and read on.

We'll start, as always, with the instance definition:

```
import Data.Argonaut (class DecodeJson) ❶

instance decodeJsonGrade :: DecodeJson Grade where
```

❶ Import the Typeclass.

Then the left-hand side of the implementation:

```
instance decodeJsonGrade :: DecodeJson Grade where
  decodeJson ???
```

It's not clear what to put next. I think it would be a good time to look up the definition of `DecodeJson`.

By the way, sometimes I find the `Pursuit` docs a bit verbose and, instead, just look at the source code, by clicking on the `Source` link on the right-hand side.

Here's the class definition from the source code:

```
class DecodeJson a where
  decodeJson :: Json -> Either JsonDecodeError a
```

We can see that `decodeJson` expect a `Json` unlike `decode` from `Foreign` which expected a `String`. We should keep this in mind when we uncomment out the Ajax code.

We'll want it to specify `ResponseFormat.json` instead of `ResponseFormat.string`. In fact, let's change that now even though the code is commented out. This way we won't forget later:

```
test :: Effect Unit
test = launchAff_ do
  log $ encodeJSON Preschool
  log $ encodeJSON $ Grade 3
  -- result <- Ajax.post ResponseFormat.string ❶
  --   "http://localhost:3000/"
  --   $ Just $ RequestBody.String $ encodeJSON teacher
  -- log $ processAjaxResult (Proxy :: _ ReversedTeacher) result ❷
```

❶ Changed `ResponseFormat.string` to `ResponseFormat.json`.

❷ Deleted `pure unit`.

Now that we know the Type Signature for `decodeJson`, we can add the rest of the left-hand side:

```
instance decodeJsonGrade :: DecodeJson Grade where
  decodeJson json = ???
```

At this point, we have a raw Json element. It could be a Number, String, Array or Object. We know from our test code output that it'll be an Object.

There are 2 Aggregate Json decoding Functions, `decodeJArray` and `decodeJObject` that are both found in

the `Data.Argonaut.Decode.Class` module.

Look them in `Pursuit` to see what they return, in particular `decodeJObject`, then continue reading.

Here's `decodeJObject's` Type Signature:

```
decodeJObject :: Json -> Either String (Object Json)
```

If this succeeds we'll get an `Object Json`, whatever that is. Click on the `Object` link in the docs for `decodeJObject` and see what you find. In particular, look for how we can find the value of a key since we're going to need to look up `gat` and `stnetnoc`, i.e `tag` and `contents` backwards.

Once you do, return here.

When I do this, I find `lookup`. Here's its Type Signature:

```
lookup :: forall a. String -> Object a -> Maybe a ❶
```

❶ This Function is from `Foreign.Object`.

Now, this isn't the best of Functions since it creates a `Maybe`. We could use `note` from `Prelude` to convert it to an `Either` and then we could code in the `Either` Monad since `decodeJObject` also returns an `Either`.

But a much bigger problem is that we have an `Object Json` which means that `lookup` will return a `Maybe Json` and not the Type we want from each field, e.g. a `String` or an `Int` or whatever.

So, let's take a different tact. We want a Function that has the following Type Signature:

```
String -> Object Json -> Either String a ❶
```

❶ Takes the `key`, e.g. `gat` or `stnetnoc` and the `Object Json` that `decodeJObject` gives us and returns an `a` on success.

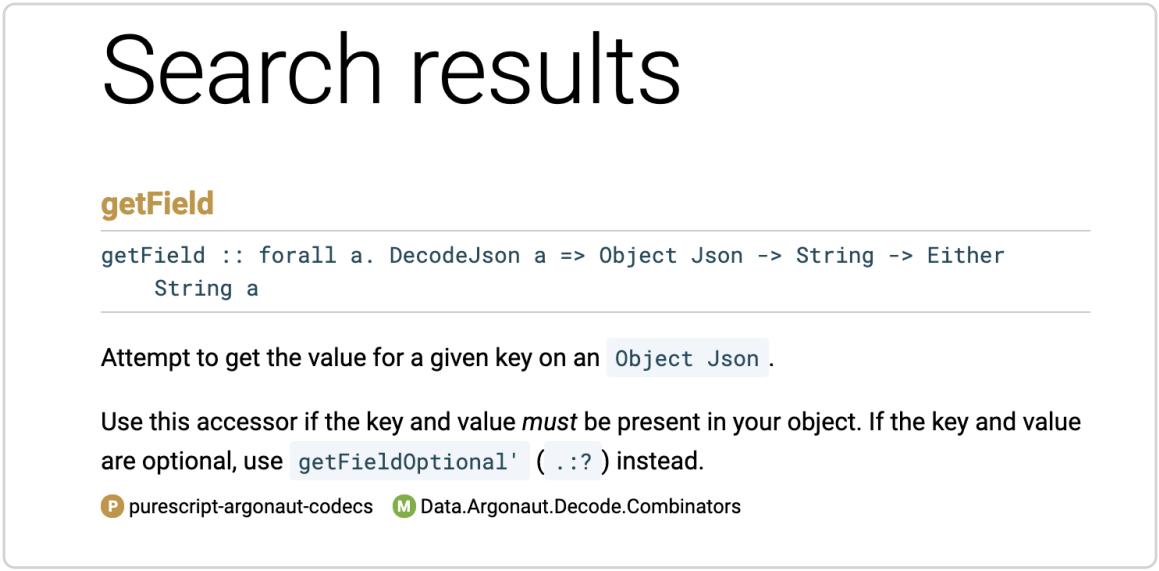
We should search `Pursuit` for that Type Signature. Do that and see what you find. Then return here.

So, when there are a lot of Functions that match this Type Signature, you may have to click `Load More Results` at the bottom of the page. I kept doing that until there were no more and then I searched the page for `Argonaut` and I got NOTHING.

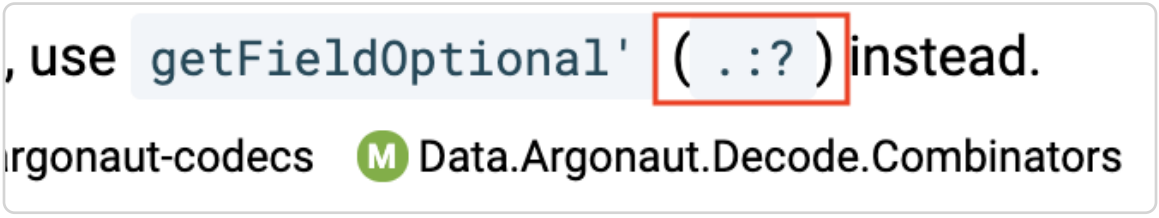
Let's swap the Parameters around:

```
Object Json -> String -> Either String a
```

This is a MUCH better search string. This is what I found as the first entry:



And when I read the docs here, I notice this part:



Notice that there's a Binary Operator for `getFieldOptional`, viz. `(.:?)`. I wonder if there's one for `getField`. I'll bet we can look for this operator and find the non-optional one on that page somewhere.

Search for `.:?` on `Pursuit` and then go to its documentation. Then once you're on that page, search for `getField` and see if you find an Operator Alias for it.

I found the Binary Operator `(.:)`. Unlike `lookup`, this give us exactly what we want, i.e. the correct Type wrapped in an `Either`.

Let's plan to use `.:` in our code.

But first, we need to start a `do` block so we're code in the `Either` Monad:

```
instance decodeJsonGrade :: DecodeJson Grade where
  decodeJson json = do
    ???
```

Then we'll get the object using `decodeJObject`:

```
import Data.Argonaut.Decode.Decoders (decodeJObject) ❶

instance decodeJsonGrade :: DecodeJson Grade where
  decodeJson json = do
    o <- decodeJObject json ❷
    ???
```

❶ Import `decodeJObject`.

❷ Remember that `decodeJObject` returns an `Either` and, therefore, can fail, e.g. if we gave it a JSON Array. If it fails, then this `do` block Short-circuits.

Now, we'll `getField` via its Binary Operator:

```
import Data.Argonaut (class DecodeJson, (.:)) ❶

instance decodeJsonGrade :: DecodeJson Grade where
  decodeJson json = do
    o <- decodeJObject json
    tag <- o .: "gat" ❷
    ???
```

❶ Add `(.:)` to import list.

❷ Remember that we're decoding what the Echo Server sends back. That's why we're looking up `"gat"` and NOT `"tag"`.

Now, see if you can continue coding the rest of this. All that's left is to check the Value of `tag` and call the appropriate `Grade` Data Constructors based on the value of `tag`.

You know enough at this point to finish this, so give it a try before reading on.

The next step is to use a `case` on `tag`:

```
instance decodeJsonGrade :: DecodeJson Grade where
  decodeJson json = do
    o <- decodeJObject json
    tag <- o .: "gat"
    case tag of
      "Preschool" -> ???
      "Kindergarten" -> ???
      "Grade" -> ???
      "High" -> ???
      "College" -> ???
      _ -> ??? ❶
```

❶ When decoding, this catch-all case is required since we may get an invalid `String`.

For the Data Constructors with no Parameters, we can just call them:

```
instance decodeJsonGrade :: DecodeJson Grade where
  decodeJson json = do
    o <- decodeJObject json
    tag <- o .: "gat"
    case tag of
      "Preschool" -> pure Preschool
      "Kindergarten" -> pure Kindergarten
      "Grade" -> ???
      "High" -> ???
      "College" -> ???
      _ -> ???
```

Now, let's work on the `Grade` Data Constructor first and use what we learn there to help us with the others.

Let's look back at the output of our test code to help us decode the Data Constructors that take Parameters:

```
{"tag":"Preschool"}
{"contents":3,"tag":"Grade"}
```

We can see that the `Int` Parameter for `Grade` is encoded in a key called `contents`, which means we'll be looking for `stnetnoc`:

```
instance decodeJsonGrade :: DecodeJson Grade where
  decodeJson json = do
    o <- decodeJObject json
    tag <- o .: "gat"
    case tag of
      "Preschool" -> pure Preschool
      "Kindergarten" -> pure Kindergarten
      "Grade" -> do ❶
        num <- o .: "stnetnoc" ❷
        ???
      "High" -> ???
      "College" -> ???
      _ -> ???
```

❶ We're coding in the `Either` Monad here again to benefit from it's Short-circuiting Side-effect.

❷ We get the field here they same way we did for `tag`.

See if you can finish this case and, if you can, then finish the rest or at least as far as you can get before returning here.

Since we now have all of the data necessary to construct a `Grade` we'll simply do so using the Data Constructor:

```
instance decodeJsonGrade :: DecodeJson Grade where
  decodeJson json = do
    o <- decodeJObject json
    tag <- o .: "gat"
    case tag of
      "Preschool" -> pure Preschool
      "Kindergarten" -> pure Kindergarten
      "Grade" -> do
        num <- o .: "stnetnoc"
        pure $ Grade num ❶
      "High" -> ???
      "College" -> ???
      _ -> ???
```

❶ Wrap this up in an `Either` since we're coding in the `Either` Monad.

If you didn't get this far, take what we've done with `Grade` and finish the code before reading on.

The rest is just a carbon copy of what we did with `Grade`:

```
instance decodeJsonGrade :: DecodeJson Grade where
  decodeJson json = do
    o <- decodeJObject json
    tag <- o .: "gat"
    case tag of
      "Preschool" -> pure Preschool
      "Kindergarten" -> pure Kindergarten
      "Grade" -> do
        num <- o .: "stnetnoc"
        pure $ Grade num
      "High" -> do
        num <- o .: "stnetnoc"
        pure $ High num
      "College" -> do
        num <- o .: "stnetnoc"
        pure $ College num
      _ -> ???
```

The fact that we've copy-pasta'd that code means that we have plenty of room to improve it. We'll wait until it compiles to do that though.

The last part is the catch-all case where we fail the decoder. We're going to want to consult `Pursuit` for a Function called `fail`. But if we just search blindly, we're going to get many.

Since, we're only concerned with the one from the `Argonaut`, it's faster to just search for `Data.Argonaut`, go to that page and search on that page for `fail`.

Go ahead and do that before reading on.

Here's what I found AFTER clicking on the source link:

```
fail :: ∀ a b. Show a => a -> Either String b
fail x = Left $ "Expected String or Number but found: " <> show x
```

This will produce an error message that will NOT be useful, so we can't use it. We'll simply add our own `Left`:

```
import Data.Argonaut (class DecodeJson, JsonDecodeError(..), (.:)) ❶

instance decodeJsonGrade :: DecodeJson Grade where
  decodeJson json = do
    o <- decodeJObject json
    tag <- o .: "gat"
    case tag of
      "Preschool" -> pure Preschool
      "Kindergarten" -> pure Kindergarten
      "Grade" -> do
        num <- o .: "stnetnoc"
        pure $ Grade num
      "High" -> do
        num <- o .: "stnetnoc"
        pure $ High num
      "College" -> do
        num <- o .: "stnetnoc"
        pure $ College num
      _ -> Left $ AtKey "tag" $ UnexpectedValue json ❷
```

❶ Added `JsonDecodeError` to import list.

❷ Failed the decoder with a custom message.

Looking at this, it's clear that we can simplify the Monadic code:

```
      "College" -> do
        num <- o .: "stnetnoc" ❶
        pure $ College num ❷
```

We don't need to write Monadically. We can just use `map`:

```
instance decodeJsonGrade :: DecodeJson Grade where
  decodeJson json = do
    o <- decodeJObject json
    tag <- o .: "gat"
    case tag of
      "Preschool" -> pure Preschool
      "Kindergarten" -> pure Kindergarten
      "Grade" -> Grade <$> o .: "stnetnoc"
      "High" -> High <$> o .: "stnetnoc"
      "College" -> College <$> o .: "stnetnoc" ❶
      _ -> Left $ AtKey "tag" $ UnexpectedValue json
```

❶ `map` the Data Constructor over the Value at key `stnetnoc`.

And now we can factor out the duplicate code, i.e. `o .: "stnetnoc"`:

```
instance decodeJsonGrade :: DecodeJson Grade where
  decodeJson json = do
    o <- decodeJObject json
    tag <- o .: "gat"
    let contents = o .: "stnetnoc" ❶
    case tag of
      "Preschool" -> pure Preschool
      "Kindergarten" -> pure Kindergarten
      "Grade" -> Grade <$> contents
      "High" -> High <$> contents
      "College" -> College <$> contents
      _ -> Left $ AtKey "tag" $ UnexpectedValue json
```

❶ We MUST be very careful here. If we coded this as `contents <- o .: "stnetnoc"`, then a failure to look up this field would Short-circuit the whole decoder. This would mean that Data Constructors that do NOT need this field would never decode.

This final version is cleaner.

One thing to note here is that if the Data Constructor `College` didn't take an `Int`, but instead took a `String`, we'd have to take into account that difference for `College` by providing a generalized Type Signature so the compiler won't overspecialize the Type:

```
instance decodeJsonGrade :: DecodeJson Grade where
  decodeJson json = do
    o <- decodeJObject json
    tag <- o .: "gat"
    let contents :: ∀ a. DecodeJson a => Either String a ❶
        contents = o .: "stnetnoc"
    case tag of
      "Preschool" -> pure Preschool
      "Kindergarten" -> pure Kindergarten
      "Grade" -> Grade <$> contents
      "High" -> High <$> contents
      "College" -> College <$> contents
      _ -> Left $ AtKey "tag" $ UnexpectedValue json
```

❶ This allows us to use `contents` more generally, for example if `College` took a `String` instead of an `Int`.

You do NOT need to add that last change, unless you want to, since `Grade`, `High` and `College` all take an `Int` and, if the compiler overspecializes, it will be fine.

Next, thing up is the `DecodeJson` Instance for `TeachingStatus`. That'll actually be easier to code than `Grade` was. Go ahead and code the whole thing up before reading on.

Since it's so simple, there's no need to code it one line at a time here. Instead, I'm just going to give it to you whole:

```
instance decodeJsonTeachingStatus :: DecodeJson TeachingStatus where
  decodeJson json = do
    o <- decodeJObject json
    tag <- o .: "gat"
    case tag of
      "StudentTeacher" -> pure StudentTeacher
      "Probationary" -> pure Probationary
      "NonTenured" -> pure NonTenured
      "Tenured" -> pure Tenured
      _ -> Left $ AtKey "tag" $ UnexpectedValue json
```

Now, we need to write encoders and decoders for the Records we made `newtypes`. We'll start with `Personal`.

Don't forget that we're encoding using the `Foreign` library. So all you need to do is make a `Generic` Instance for `Personal` and then use `derive newtype` to create the `Encode` Instance.

We can use `derive newtype` since all Records already have an `Encode` Instance. And Records encode to the same Field structure as their PureScript counterparts, e.g. `Personal` will encode to:

```
{ "height": 103.5,
  "weight": 43,
  "age": 12
}
```

Write that up for `Personal` before reading any more.

Here are the changes:

```
newtype Personal = Personal
  { height :: Centimeters
  , weight :: Kilograms
  , age :: Years
  }
derive instance genericPersonal :: Generic Personal _ ❶
derive newtype instance encodePersonal :: Encode Personal ❷
```

❶ Derive a `Generic` Instance.

❷ Delegate the `newtype's` Instance to the contained Type's Instance.

Do the same for `Student` and `Teacher` before reading further.

Here are those changes:

```
newtype Student = Student
  { grade :: Grade
  , teacher :: Teacher
  , gpa :: GPA
  , personal :: Personal
  }
derive instance genericStudent :: Generic Student _ ❶
derive newtype instance encodeStudent :: Encode Student ❷

newtype Teacher = Teacher
  { grades :: Array Grade
  , numberOfStudents :: Int
  , personal :: Personal
  , status :: TeachingStatus
  }
derive instance genericTeacher :: Generic Teacher _ ❶
derive newtype instance encodeTeacher :: Encode Teacher ❷
```

❶ Derive a `Generic` Instance.

❷ Delegate the `newtype's` Instance to the contained Type's Instance.

Now is a good time to remove the unused imports leaving you the following imports:

```
import Prelude

import Affjax as Ajax
import Control.Monad.Except (runExcept)
import Data.Argonaut (class DecodeJson, JsonDecodeError(..), (.:))
import Data.Argonaut.Decode.Decoders (decodeJObject)
import Data.Either (Either(..))
import Data.Generic.Rep (class Generic)
import Data.Show.Generic (genericShow)
import Effect (Effect)
import Effect.Aff (launchAff_)
import Effect.Class.Console (log)
import Foreign.Generic (decodeJSON, defaultOptions, encodeJSON, genericEncode)
import Foreign.Generic.Class (class Decode, class Encode)
import Type.Proxy (Proxy)
```

Now, write the decoder for `Personal` using the `Argonaut` library. Use what we've done with `Grade` and `TeachingStatus` as you guide. You're going to run into a couple of issues along the way. If you cannot overcome them, don't worry. Just return here and we'll go through it together.

Here's an example JSON to help:

```
{
  "contents": {
    "weight": 20,
    "height": 10,
    "age": 30
  },
  "tag": "Personal"
}
```

When you're done, return back here.

We'll start coding with the instance definition:

```
instance decodeJsonPersonal :: DecodeJson Personal where
```

Next we'll code up the left-hand side:

```
instance decodeJsonPersonal :: DecodeJson Personal where
  decodeJson json = ???
```

Since `Personal` is a Record, we're going to have a lot of fields to decode. This is a good case for coding it Monadically using a `do` block:

```
instance decodeJsonPersonal :: DecodeJson Personal where
  decodeJson json = do
    ???
```

Since it's a JSON Object, we will use `decodeJObject`:

```
instance decodeJsonPersonal :: DecodeJson Personal where
  decodeJson json = do
    o <- decodeJObject json
    ???
```

Next, we want to make sure that the `tag` is correct, i.e. we're decoding a `Personal` not just any old Type that happens to have the same fields as `Personal`, for example:

```
newtype Imposter = Imposter
  { hairColor :: HairColor
  , height :: Centimeters
  , weight :: Kilograms
  , occupation :: Occupation
  , age :: Years
  }
```

Here `Imposter` will have all of the fields that `Personal` has. The fact that it has additional fields won't deter `Personal's` decoder since it will simply ignore them.

The only real difference here is that the `tag` will be `Imposter`, which is why we want to make sure we check the `tag` in the decoder.

`tag` gives our JSON a bit of Type Safety:

```
instance decodeJsonPersonal :: DecodeJson Personal where
  decodeJson json = do
    o <- decodeJObject json
    tag <- o .: "gat" ❶
    if tag == "Personal" then do ❷
      ???
    else Left $ AtKey "tag" $ UnexpectedValue json ❸
```

❶ Get the `tag` Value from the JSON Object, `o`.

❷ Make sure `tag` is `Personal`.

❸ Return error. Notice that `fail` is not used here because its hardcoded error message isn't accurate for this situation.

If you didn't get this far, please stop reading and finish the code.

The next step is to get the fields. We'll name the variables the same as the field names to make the construction of the Record simpler, i.e. we won't have to say `{ height: height }` when `{ height }` will do. But we're not going to forget that the Echo Server will reverse our keys, so we'll get the fields with backward names:

```
instance decodeJsonPersonal :: DecodeJson Personal where
  decodeJson json = do
    o <- decodeJObject json
    tag <- o .: "gat"
    if tag == "Personal" then do
      height <- o .: "thgieh"
      weight <- o .: "thgiew"
      age <- o .: "ega"
      ???
    else Left $ AtKey "tag" $ UnexpectedValue json
```

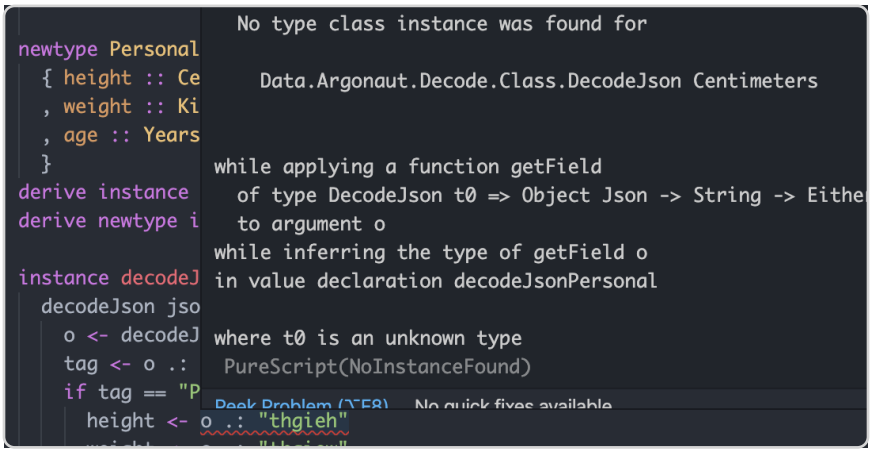
Notice the field name `Strings` are backwards. That's because we're decoding the response from the Echo Server.

And finally, we create a `Personal` and wrap it in an `Either`:

```
instance decodeJsonPersonal :: DecodeJson Personal where
  decodeJson json = do
    o <- decodeJObject json
    tag <- o .: "gat"
    if tag == "Personal" then do
      height <- o .: "thgieh" -- COMPILER ERROR!!
      weight <- o .: "thgiew"
      age <- o .: "ega"
      pure $ Personal { height, weight, age } ❶
    else Left $ AtKey "tag" $ UnexpectedValue json
```

❶ We're using a shorthand syntax here, which works when the name of the field and the name of the variable are the same.

And now we have a compiler error:



Looks like we're not going to be able to keep the current decoders for `Centimeters` since it's using the `Foreign` library.

Well, technically we could make it work, but then our code would look like:

```
import Data.Bifunctor (lmap) ❶

instance decodeJsonPersonal :: DecodeJson Personal where
  decodeJson json = do
    o <- decodeJObject json
    tag <- o .: "gat"
    if tag == "Personal" then do
      height <- do
        x <- o .: "thgieh" ❷
        lmap (TypeMismatch <<< show) <$> runExcept $ decodeJSON x ❸ ❹
      weight <- do
        x <- o .: "thgiew"
        lmap (TypeMismatch <<< show) <$> runExcept $ decodeJSON x
      age <- do
        x <- o .: "ega"
        lmap (TypeMismatch <<< show) <$> runExcept $ decodeJSON x
      pure $ Personal { height, weight, age }
    else Left $ AtKey "tag" $ UnexpectedValue json
```

❶ Import `lmap`.

❷ Extract the JSON data from the `Either`.

❸ Decode the JSON data and run the Except to get an `Either` and then `lmapping` to convert the `Error` to a `JsonDecodeError`. We are running in the `Either JsonDecodeError` Monad.

❹ Map the `Left` of the `Either` from `F` to `JsonDecodeError`.

This code looks terrible. But, we could fix that by factoring out the common code into a helper Function:

```
decodeForeign :: ∀ a. Decode a => Either JsonDecodeError String -> Either
JsonDecodeError a
decodeForeign fieldData = do ❶
  x <- fieldData
  lmap (TypeMismatch <<< show) <$> runExcept $ decodeJSON x
```

❶ Factored out the common code.

Then our code would clean up a bit:

```
instance decodeJsonPersonal :: DecodeJson Personal where
  decodeJson json = do
    o <- decodeJObject json
    tag <- o .: "gat"
    if tag == "Personal" then do
      height <- decodeForeign $ o .: "thgieh"
      weight <- decodeForeign $ o .: "thgiew"
      age <- decodeForeign $ o .: "ega"
      pure $ Personal { height, weight, age }
    else Left $ AtKey "tag" $ UnexpectedValue json
```

This solution is better but still we have a bit of clutter having to call our helper, `decodeForeign`. And it's a fine solution if we MUST keep decoding `Centimeters`, `Kilograms` and `Years` using the `Foreign` library.

But, we don't have that requirement and, instead, we going to just derive a decoder using `Argonaut` for these Types:

```
newtype Centimeters = Centimeters Number
derive instance genericCentimeters :: Generic Centimeters _
derive newtype instance encodeCentimeters :: Encode Centimeters
derive newtype instance decodeJsonCentimeters :: DecodeJson Centimeters ❶

newtype Kilograms = Kilograms Number
derive instance genericKilograms :: Generic Kilograms _
derive newtype instance encodeKilograms :: Encode Kilograms
derive newtype instance decodeJsonKilograms :: DecodeJson Kilograms ❶

newtype Years = Years Int
derive instance genericYears :: Generic Years _
derive newtype instance encodeYears :: Encode Years
derive newtype instance decodeJsonYears :: DecodeJson Years ❶
```

❶ Replaced the derive from `Decode` to `DecodeJson`.

So, go ahead make these changes to your code and watch the compiler error disappear:

```
instance decodeJsonPersonal :: DecodeJson Personal where
  decodeJson json = do
    o <- decodeJObject json
    tag <- o .: "gat"
    if tag == "Personal" then do
      height <- o .: "thgieh"
      weight <- o .: "thgiew"
      age <- o .: "ega"
      pure $ Personal { height, weight, age }
    else Left $ AtKey "tag" $ UnexpectedValue json
```

Next, code up the decoder for `Teacher` using the above decoder as reference. When you've got that done, return here for the solution.

Since, the solution is practically the same as for `Personal`, I'm going to give it to you already done:

```
instance decodeJsonTeacher :: DecodeJson Teacher where
  decodeJson json = do
    o <- decodeJObject json
    tag <- o .: "gat"
    if tag == "Teacher" then do
      grades <- o .: "sedarg"
      numberOfStudents <- o .: "stnedutSfOrebmun"
      personal <- o .: "lanosrep"
      status <- o .: "sutats"
      pure $ Teacher { grades, numberOfStudents, personal, status } ❶
    else Left $ AtKey "tag" $ UnexpectedValue json
```

❶ Construction of `Teacher` uses the shorthand syntax for Record field Values.

And now, do the same for `Student` before reading on.

Here's `Student's` decoder:

```
instance decodeJsonStudent :: DecodeJson Student where
  decodeJson json = do
    o <- decodeJObject json
    tag <- o .: "gat"
    if tag == "Student" then do
      grade <- o .: "edarg"
      teacher <- o .: "rehcaet"
      gpa <- o .: "apg" -- COMPILER ERROR!!
      personal <- o .: "lanosrep"
      pure $ Student { grade, teacher, gpa, personal }
    else Left $ AtKey "tag" $ UnexpectedValue json
```

Without looking at the editor pop-up, can you see why we get a compiler error?

We have the same problem we had with `Centimeters`, `Kilograms` and `Years`, i.e. `GPA` has a decoder that uses the wrong library, i.e. `Foreign` and not `Argonaut`.

Apply the same fix to `GPA` that we did to `Centimeters`, `Kilograms` and `Years before reading further.

Here's the fix:

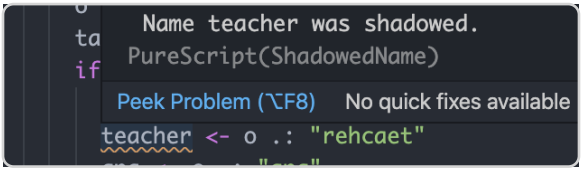
```
newtype GPA = GPA Number
derive instance genericGPA :: Generic GPA _
derive newtype instance encodeGPA :: Encode GPA
derive newtype instance decodeJsonGPA :: DecodeJson GPA ❶
```

❶ Replaced the derive from `Decode` to `DecodeJson`.

Now, you may have noticed that we got a Shadow variable warning:

```
instance decodeJsonStudent :: DecodeJson Student where
  decodeJson json = do
    o <- decodeJObject json
    tag <- o .: "gat"
    if tag == "Student" then do
      grade <- o .: "edarg"
      teacher <- o .: "rehcaet" -- COMPILER WARNING!!
      gpa <- o .: "apg"
      personal <- o .: "lanosrep"
      pure $ Student { grade, teacher, gpa, personal }
    else Left $ AtKey "tag" $ UnexpectedValue json
```

Here's the warning:



This warning is because we have global `teacher` in this module:

```
teacher :: Teacher
teacher = Teacher
  { grades: [ Preschool, Kindergarten, Grade 1 ]
  , numberOfStudents: 23
  , personal: Personal {
      height: Centimeters 162.56
    , weight: Kilograms 63.5
    , age: Years 31
    }
  , status: NonTenured
  }
```

Rename the global one to `testTeacher`:

```
testTeacher :: Teacher
testTeacher = Teacher
...
```

And there are no more compiler errors.

Let's delete `ReversedPersonal`, `ReverseStudent` and `ReversedTeacher` since we no longer need them.

At this point, we're ready to change our test code. Let's look at its current state:

```
test :: Effect Unit
test = launchAff_ do
  log $ encodeJSON Preschool
  log $ encodeJSON $ Grade 3
  -- result <- Ajax.post ResponseFormat.string
  --   "http://localhost:3000/"
  --   $ Just $ RequestBody.String $ encodeJSON teacher
  -- log $ processAjaxResult (Proxy :: _ ReversedTeacher) result
```

Let's delete the 2 log statements since we don't need them any more. If you remember, they were just there to help us see how these Data Constructors would be encoded into JSON.

```
test :: Effect Unit
test = launchAff_ do
  -- result <- Ajax.post ResponseFormat.string
  --   "http://localhost:3000/"
  --   $ Just $ RequestBody.String $ encodeJSON teacher
  -- log $ processAjaxResult (Proxy :: _ ReversedTeacher) result
```

Next, uncomment the remaining code:

```
import Affjax.RequestBody as RequestBody ❶
import Affjax.ResponseFormat as ResponseFormat
import Data.Maybe (Maybe(..))

test :: Effect Unit
test = launchAff_ do
  result <- Ajax.post ResponseFormat.string
    "http://localhost:3000/"
    $ Just $ RequestBody.String $ encodeJSON teacher -- COMPILER ERROR!!
  log $ processAjaxResult (Proxy :: _ ReversedTeacher) result
```

❶ Add back these imports. We cleaned them up eariler because the code that used them was commented out.

The compiler error is because we renamed `teacher` to `testTeacer`. We should fix that:

```
test :: Effect Unit
test = launchAff_ do
  result <- Ajax.post ResponseFormat.string
    "http://localhost:3000/"
    $ Just $ RequestBody.String $ encodeJSON testTeacher
  -- COMPILER ERROR!!
  log $ processAjaxResult (Proxy :: _ ReversedTeacher) result
```

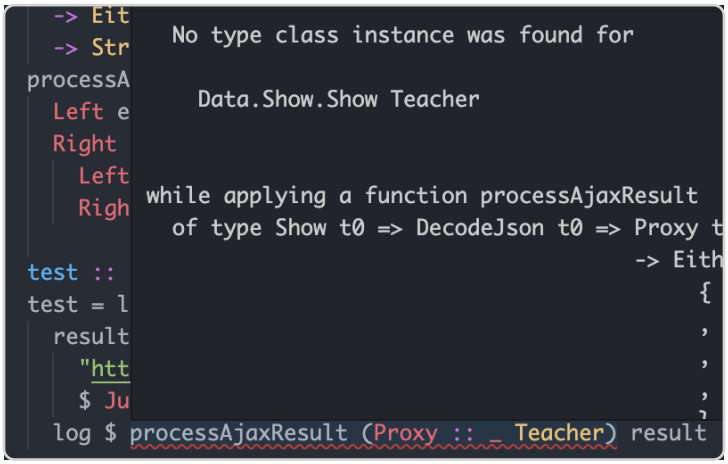
Since we deleted `ReverseTeacher`, we're getting this error. Let's change it to `Teacher` since its decoder is smart enough to look for reversed key names:

```
import Type.Proxy (Proxy(..)) ❶

test :: Effect Unit
test = launchAff_ do
  result <- Ajax.post ResponseFormat.string
    "http://localhost:3000/"
    $ Just $ RequestBody.String $ encodeJSON testTeacher
  -- COMPILER ERROR!!
  log $ processAjaxResult (Proxy :: _ Teacher) result
```

❶ Added `(..)` to get the `Proxy` Data Constructor.

And now a different error:



We're missing a `Show` Instance for `Teacher`. But wait. Why didn't we need this before?

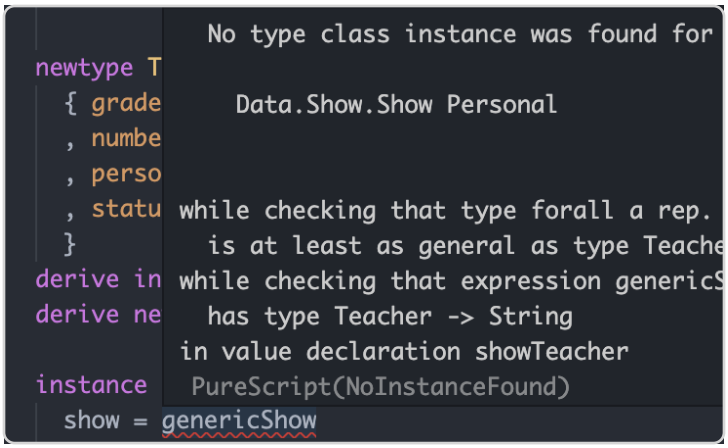
Well, if you remember correctly, it's because we were using `ReversedTeacher` and we didn't need a `Show` Instance because it was a Type Alias.

But now, `Teacher` is a `newtype`. Go ahead and add an instance for `Show` for `Teacher` before reading any more.

Here's that instance:

```
instance showTeacher :: Show Teacher where
  show = genericShow -- COMPILER ERROR!!
```

And now this error:



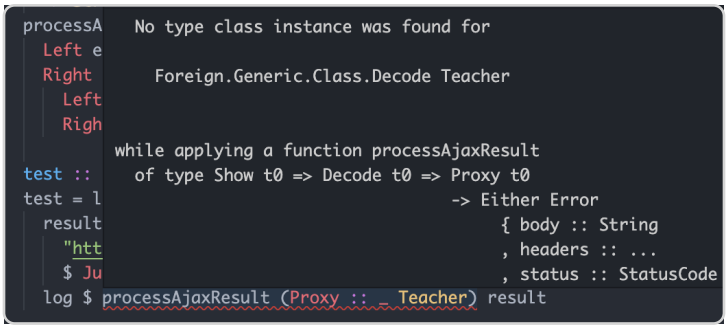
Looks like we need a `Show` Instance for `Personal` too. Make one and while you're at it, make one for `Student` too. Then read on.

Here are those instances:

```
instance showPersonal :: Show Personal where
  show = genericShow

instance showStudent :: Show Student where
  show = genericShow
```

We now have a different error on the same bit of code:



It's complaining that `Teacher` doesn't have a `Decode` Instance, which is correct. But why?

Turns out that `processAjaxResult` still expects whatever we pass to it will have a `Decode` Instance. We need to change that.

```
processAjaxResult
  :: ∀ a
  .  Show a
  => DecodeJson a ❶
  => Proxy a
  -> Either Ajax.Error (Ajax.Response String)
  -> String
processAjaxResult _ = case _ of
  Left err -> Ajax.printError err
  -- COMPILER ERROR!!
  Right { body } -> case runExcept (decodeJSON body :: _ a) of
    Left err -> show err
    Right reversedRecord -> show reversedRecord
```

❶ Changed from `Decode` to `DecodeJson`.

And now the compiler isn't happy that we're still calling `decodeJSON`, so let's fix that too:

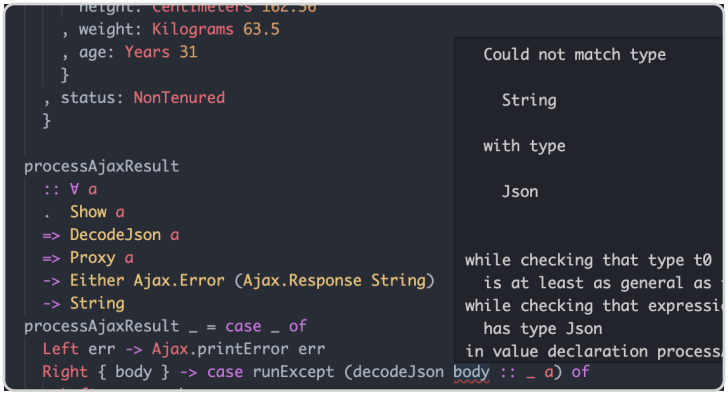
```
import Data.Argonaut (class DecodeJson, JsonDecodeError(..), (.:)
                    , decodeJson) ❶

processAjaxResult
  :: ∀ a
  .  Show a
  => DecodeJson a
  => Proxy a
  -> Either Ajax.Error (Ajax.Response String)
  -> String
processAjaxResult _ = case _ of
  Left err -> Ajax.printError err
  -- COMPILER ERROR!!
  Right { body } -> case runExcept (decodeJson body :: _ a) of ❷
    Left err -> show err
    Right reversedRecord -> show reversedRecord
```

❶ Added `decodeJson` to the import list.

❷ Changed `decodeJSON` to `decodeJson`.

And now this error:



The compiler can't match `Json` with `String`. What should we do next?

If you're not sure, then ask the question, how does the compiler know what type `body` is?

It gets it from `Response`:

```
type Response a =
  { status :: StatusCode
  , statusText :: String
  , headers :: Array ResponseHeader
  , body :: a
  }
```

And now, look at the Type Signature for `processAjaxResult` to see if you can find where we specify `Response`.

Can you see that we specified `String` as the `a` for `Response`? That's because we were using `decodeJSON`, but we changed over to `decodeJson`, which wants a `Json`.

So, we'll fix that:

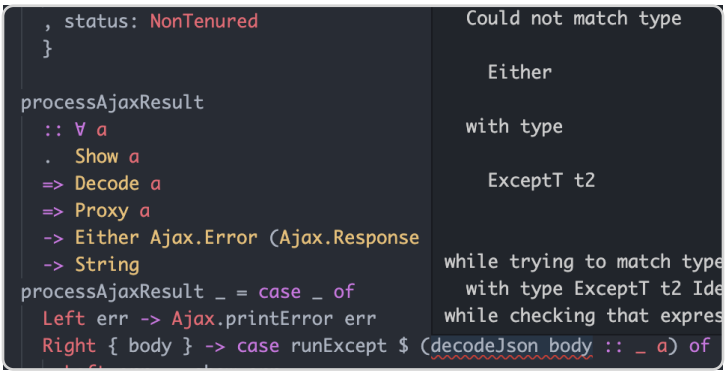
```
import Data.Argonaut (class DecodeJson, JsonDecodeError(..), Json, (.:) ❶
                    , decodeJson)

processAjaxResult
  :: ∀ a
  .  Show a
  => DecodeJson a
  => Proxy a
  -> Either Ajax.Error (Ajax.Response Json) ❷
  -> String
processAjaxResult _ = case _ of
  Left err -> Ajax.printError err
  -- COMPILER ERROR!!
  Right { body } -> case runExcept (decodeJson body :: _ a) of
    Left err -> show err
    Right reversedRecord -> show reversedRecord
```

❶ Added `Json` to import list.

❷ Changed from `String` to `Json`.

And now, a new compiler error:



Can you see my problem?

We don't need `runExcept` anymore like we did with `decodeJSON` since `decodeJson` already returns an `Either`. Let's remove it:

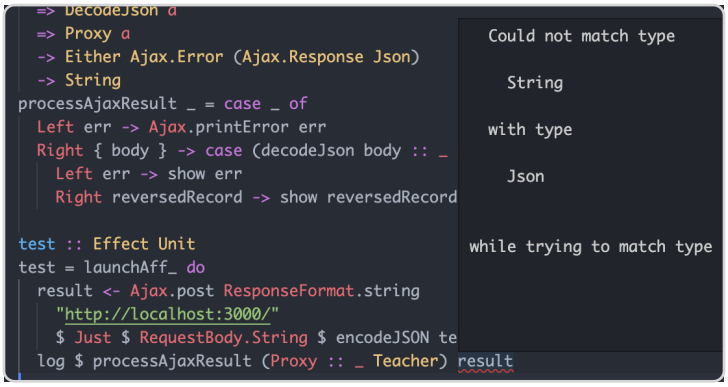
```
processAjaxResult
  :: ∀ a
  .  Show a
  => DecodeJson a
  => Proxy a
  -> Either Ajax.Error (Ajax.Response Json)
  -> String
processAjaxResult _ = case _ of
  Left err -> Ajax.printError err
  Right { body } -> case (decodeJson body :: _ a) of ❶
    Left err -> show err
    Right reversedTeacher -> show reversedTeacher
```

❶ Removed `runExcept`.

And now an error in our test code:

```
test :: Effect Unit
test = launchAff_ do
  result <- Ajax.post ResponseFormat.string
    "http://localhost:3000/"
    $ Just $ RequestBody.String $ encodeJSON testTeacher
  -- COMPILER ERROR!!
  log $ processAjaxResult (Proxy :: _ Teacher) result
```

Here's the error:



We have a Type Mismatch involving `result`. We know that we've moved from `String` to `Json` going from `Foreign` to `Argonaut`. And now, our `Ajax` call response is in the wrong format. This needs to be fixed:

```
test :: Effect Unit
test = launchAff_ do
  result <- Ajax.post ResponseFormat.json ❶
    "http://localhost:3000/"
    $ Just $ RequestBody.String $ encodeJSON testTeacher
  log $ processAjaxResult (Proxy :: _ Teacher) result
```

❶ Changed from `ResponseFormat.string` to `ResponseFormat.json`

And finally, there are NO more errors. Just warnings on our imports. Go ahead and clean those up.

And when we run our code we get:

```
[error] Some of your project files import modules from packages that are not
in the direct dependencies of your project.
To fix this error add the following packages to the list of dependencies in
your config:
- argonaut-codecs
```

After fixing this we now get:

```
(AtKey "gat" MissingValue)
```

Okay, so this error is pretty terrible. But it's clearly looking for the `tag` field in `Teacher` since it's the only thing we're working with.

But why can't it find it?

Let's print out the response from the `Ajax` call. To do that, we're going to need a Function called `stringify`:

```
import Data.Argonaut (class DecodeJson, JsonDecodeError(..), Json, (.:)
                    , decodeJson, stringify) ❶
import Data.Bifunctor (bimap) ❷

test :: Effect Unit
test = launchAff_ do
  result <- Ajax.post ResponseFormat.json
    "http://localhost:3000/"
    $ Just $ RequestBody.String $ encodeJSON testTeacher
  log $ show $ bimap Ajax.printError (stringify <<< _.body) $ result ❸ ❹
  log $ processAjaxResult (Proxy :: _ Teacher) result
```

❶ Added `stringify` to import list.

❷ Import `bimap`.

❸ Map the `Left` to convert the `Ajax.Error` to a `String`.

❹ Map the `Right` to get the `body` from the `Ajax.Response` and then `stringify` it.

You may remember `stringify` from Javascript. It does pretty much the same thing, i.e. takes a `Json`, which we have thanks to `ResponseFormat.json`, and it converts it to `String`.

When we run this we get:

```
(Right
"{\"sutats\":{\"gat\":\"NonTenured\"},\"lanosrep\":{\"thgiew\":63.5,\"thgieh\"
:162.56,\"ega\":31},\"stnedutSfOrebmun\":23,\"sedarg\":[{\"gat\":\"Preschool\"
},{\"gat\":\"Kindergarten\"},{\"stnetnoc\":1,\"gat\":\"Grade\"}]}")
(AtKey "gat" MissingValue)
```

Look through the output and see if you can find the key, `gat` with a Value of `Teacher`.

It turns out that it's not there. But why?

Let's look at our decoder for `Teacher` to see if we can see why that key is not there:

```
newtype Teacher = Teacher
  { grades :: Array Grade
  , numberOfStudents :: Int
  , personal :: Personal
  , status :: TeachingStatus
  }
derive instance genericTeacher :: Generic Teacher _
derive newtype instance encodeTeacher :: Encode Teacher ❶
```

❶ Here's where we derived the encoder.

It should be clear that we're delegating to the contained Type here since we using `derive newtype`. This bypasses the Data Constructor and only encodes the Record, i.e. the contained Type.

`gat` isn't found because `tag` was never generated.

Think about how to fix this before reading on.

We are deriving the encoder by delegating to the encoder for the Record. We have remove that and write our own. Look at how we MANUALLY wrote encoders for other Types and fix this so that it will include the `tag`.

Then read on.

First, we need to remove the `derive newtype` line and then write an instance using `genericEncode`:

```
newtype Teacher = Teacher
  { grades :: Array Grade
  , numberOfStudents :: Int
  , personal :: Personal
  , status :: TeachingStatus
  }
derive instance genericTeacher :: Generic Teacher _ ❶

instance encodeTeacher :: Encode Teacher where ❷
  encode = genericEncode defaultOptions
```

❶ Removed `derive newtype`.

❷ Explicit instance using `genericEncode`.

And now when we run it we get:

```
(Right
"{\"stnetnoc\":{\"sutats\":{\"gat\":\"NonTenured\"},\"lanosrep\":{\"thgiew\":6
3.5,\"thgieh\":162.56,\"ega\":31},\"stnedutSfOrebmun\":23,\"sedarg\":[{\"gat\"
:\"Preschool\"},{\"gat\":\"Kindergarten\"},{\"stnetnoc\":1,\"gat\":\"Grade\"}]
},\"gat\":\"Teacher\"}")
(AtKey "sedarg" MissingValue)
```

The decoding error is `"Expected field \"sedarg\""`.

Looking at this is difficult since all of the keys are backwards. So, lets change our test program to output `testTeacher` encoded and comment everything else out:

```
test :: Effect Unit
test = launchAff_ do
  log $ show $ encodeJSON testTeacher
  -- result <- Ajax.post ResponseFormat.json
  --   "http://localhost:3000/"
  --   $ Just $ RequestBody.String $ encodeJSON testTeacher
  -- log $ show $ bimap Ajax.printError (stringify <<< _.body) $ result
  -- log $ processAjaxResult (Proxy :: _ Teacher) result
```

Make this change and run it to see if you can figure out why our decoder cannot find `sedarg`, i.e. `grades`. Try to figure out the problem before reading on.

When we run our program, we get:

```
"{\"contents\":{\"status\":{\"tag\":\"NonTenured\"},\"personal\":{\"weight\":6
3.5,\"height\":162.56,\"age\":31},\"numberOfStudents\":23,\"grades\":[{\"tag\"
:\"Preschool\"},{\"tag\":\"Kindergarten\"},{\"contents\":1,\"tag\":\"Grade\"}]
},\"tag\":\"Teacher\"}"
```

Looking carefully through this, we can see that we have 2 top-level keys, `tag` and `contents`. That's the problem. We didn't take this into consideration when we removed the `derive newtype`. The encoding for `Teacher` is different now.

`Argonaut` puts the Data Constructor under `tag` and everything else under `contents`.

Let's add the fact that `grades` is under the `contents` key to the decoder for `Teacher`:

```
instance decodeJsonTeacher :: DecodeJson Teacher where
  decodeJson json = do
    o <- decodeJObject json
    tag <- o .: "gat"
    if tag == "Teacher" then do
      c <- o .: "stnetnoc" ❶
      grades <- c .: "sedarg" ❷
      numberOfStudents <- c .: "stnedutSfOrebmun" ❷
      personal <- c .: "lanosrep" ❷
      status <- c .: "sutats" ❷
      pure $ Teacher { grades, numberOfStudents, personal, status }
    else Left $ AtKey "tag" $ UnexpectedValue json
```

❶ Get `contents` into `c`.

❷ Get all the fields from the `contents` JSON Object, i.e. `c`.

Make these change to your code.

Then revert the changes we made to the test code:

```
test :: Effect Unit
test = launchAff_ do
  result <- Ajax.post ResponseFormat.json
    "http://localhost:3000/"
    $ Just $ RequestBody.String $ encodeJSON testTeacher
  log $ show $ bimap Ajax.printError (stringify <<< _.body) $ result
  log $ processAjaxResult (Proxy :: _ Teacher) result
```

Now run the program to see what you get.

We should get:

```
(Right
"{\"stnetnoc\":{\"sutats\":{\"gat\":\"NonTenured\"},\"lanosrep\":{\"thgiew\":6
3.5,\"thgieh\":162.56,\"ega\":31},\"stnedutSfOrebmun\":23,\"sedarg\":[{\"gat\"
:\"Preschool\"},{\"gat\":\"Kindergarten\"},{\"stnetnoc\":1,\"gat\":\"Grade\"}]
},\"gat\":\"Teacher\"}")
(AtKey "lanosrep" (AtKey "gat" MissingValue))
```

We failed to decode the `personal` key (show backwards as `lanosrep`). It too expects a `gat` but its missing as seen by `MissingValue`. This is because of it too is under the `contents` key.

Go ahead and fix `Personal's` encoding the way we fixed it for `Teacher`. And while you're at it, fix `Student` as well. Then continue reading.

Here are those changes:

```
instance encodePersonal :: Encode Personal where
  encode = genericEncode defaultOptions

instance encodeStudent :: Encode Student where
  encode = genericEncode defaultOptions
```

In both of these changes, we removed the `derive newtype`.

Now, we know we're going to have to make the same change to the decoders for `Personal` and `Student` that we made in `Teacher's`.

But before we do that, run your code as it is to see what kind of error you get. Then return here.

I get the following error:

```
(Right
"{\"stnetnoc\":{\"sutats\":{\"gat\":\"NonTenured\"},\"lanosrep\":{\"stnetnoc\"
:{\"thgiew\":63.5,\"thgieh\":162.56,\"ega\":31},\"gat\":\"Personal\"},\"stnedu
tSfOrebmun\":23,\"sedarg\":[{\"gat\":\"Preschool\"},{\"gat\":\"Kindergarten\"}
,{\"stnetnoc\":1,\"gat\":\"Grade\"}]},\"gat\":\"Teacher\"}")
(AtKey "lanosrep" (AtKey "thgieh" MissingValue))
```

Not surprising the `height` (shown backwards in the error as `thgieh`) field is missing under the `personal` key. This is because it's buried under contents now, which is just what we expected.

So, fix both `Personal` and `Student's` decoder in the same way we did for `Personal` and then continue reading.

Here are those changes:

```
instance decodeJsonPersonal :: DecodeJson Personal where
  decodeJson json = do
    o <- decodeJObject json
    tag <- o .: "gat"
    if tag == "Personal" then do
      c <- o .: "stnetnoc" ❶
      height <- c .: "thgieh" ❷
      weight <- c .: "thgiew" ❷
      age <- c .: "ega" ❷
      pure $ Personal { height, weight, age }
    else Left $ AtKey "tag" $ UnexpectedValue json

instance decodeJsonStudent :: DecodeJson Student where
  decodeJson json = do
    o <- decodeJObject json
    tag <- o .: "gat"
    if tag == "Student" then do
      c <- o .: "stnetnoc" ❶
      grade <- c .: "edarg" ❷
      teacher <- c .: "rehcaet" ❷
      gpa <- c .: "apg" ❷
      personal <- c .: "lanosrep" ❷
      pure $ Student { grade, teacher, gpa, personal }
    else Left $ AtKey "tag" $ UnexpectedValue json
```

❶ Get `contents` into `c`.

❷ Get all the fields from the `contents` JSON Object, i.e. `c`.

Now run your code to see what you get.

I get the following:

```
(Right
"{\"stnetnoc\":{\"sutats\":{\"gat\":\"NonTenured\"},\"lanosrep\":{\"stnetnoc\"
:{\"thgiew\":63.5,\"thgieh\":162.56,\"ega\":31},\"gat\":\"Personal\"},\"stnedu
tSfOrebmun\":23,\"sedarg\":[{\"gat\":\"Preschool\"},{\"gat\":\"Kindergarten\"}
,{\"stnetnoc\":1,\"gat\":\"Grade\"}]},\"gat\":\"Teacher\"}")
(Teacher { grades: [Preschool,Kindergarten,(Grade 1)], numberOfStudents: 23,
personal: (Personal { age: (Years 31), height: (Centimeters 162.56), weight:
(Kilograms 63.5) }), status: NonTenured })
```

And isolating the decoded `Teacher`:

```
(Teacher { grades: [Preschool,Kindergarten,(Grade 1)], numberOfStudents: 23,
personal: (Personal { age: (Years 31), height: (Centimeters 162.56), weight:
(Kilograms 63.5) }), status: NonTenured })
```

Which compares directly with `testTeacher`:

```
testTeacher :: Teacher
testTeacher = Teacher
  { grades: [ Preschool, Kindergarten, Grade 1 ]
  , numberOfStudents: 23
  , personal: Personal {
      height: Centimeters 162.56
    , weight: Kilograms 63.5
    , age: Years 31
    }
  , status: NonTenured
  }
```

We have now successfully taken `testTeacher`, encoded it, sent it to our Echo Server, who replies with our request with all of our JSON keys reverses, and then decoded that response back into a `Teacher` that matches exactly what we started with.

25.2.4. Serial and Parallel `Aff`

In the previous Chapter, we saw how we could make a `GET` Ajax called followed by a `POST`:

```
test :: Effect Unit
test = launchAff_ do
  processAjaxResult (Proxy :: _ GetPostRes)
    =<< Ajax.get ResponseFormat.string ❶
      "https://jsonplaceholder.typicode.com/posts/1"
  processAjaxResult (Proxy :: _ CreateBlogPostRes)
    =<< Ajax.post ResponseFormat.string ❷
      "https://jsonplaceholder.typicode.com/posts"
      (Just $ RequestBody.String $
        encodeJSON { userId:1, title:"title", completed:false })
```

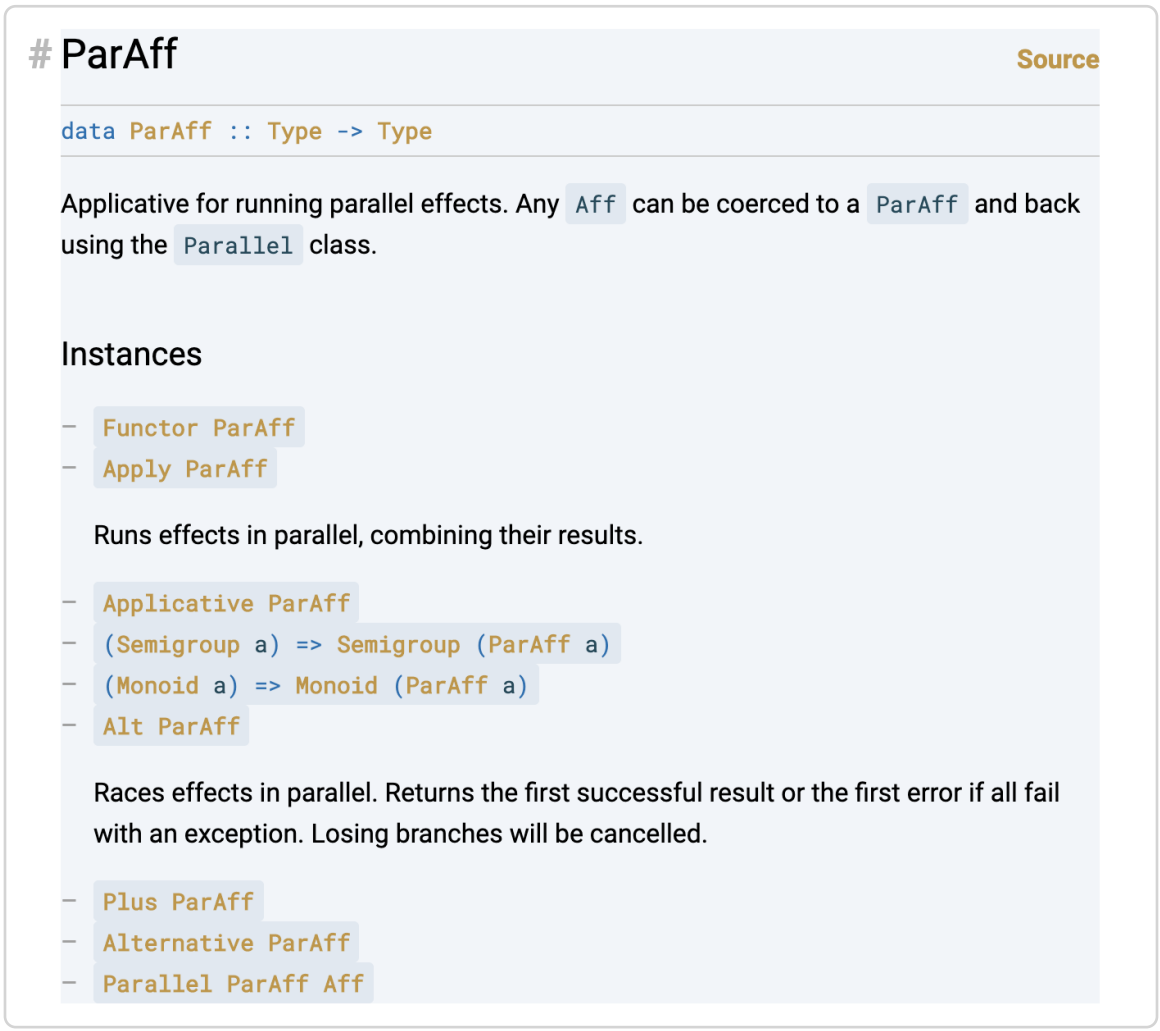
❶ Call `Ajax.get`.

❷ Call `Ajax.post`.

These are done serially, i.e. one right after the other. But what if we want to call them at the same time and

then continue executing our code after they BOTH complete?

To do that, we're going to need to learn about an Applicative that helps us run `Aff` in parallel called `ParAff`. Here's its documentation from `Pursuit`:



ParAff Source

```
data ParAff :: Type -> Type
```

Applicative for running parallel effects. Any `Aff` can be coerced to a `ParAff` and back using the `Parallel` class.

Instances

- `Functor ParAff`
- `Apply ParAff`

  Runs effects in parallel, combining their results.

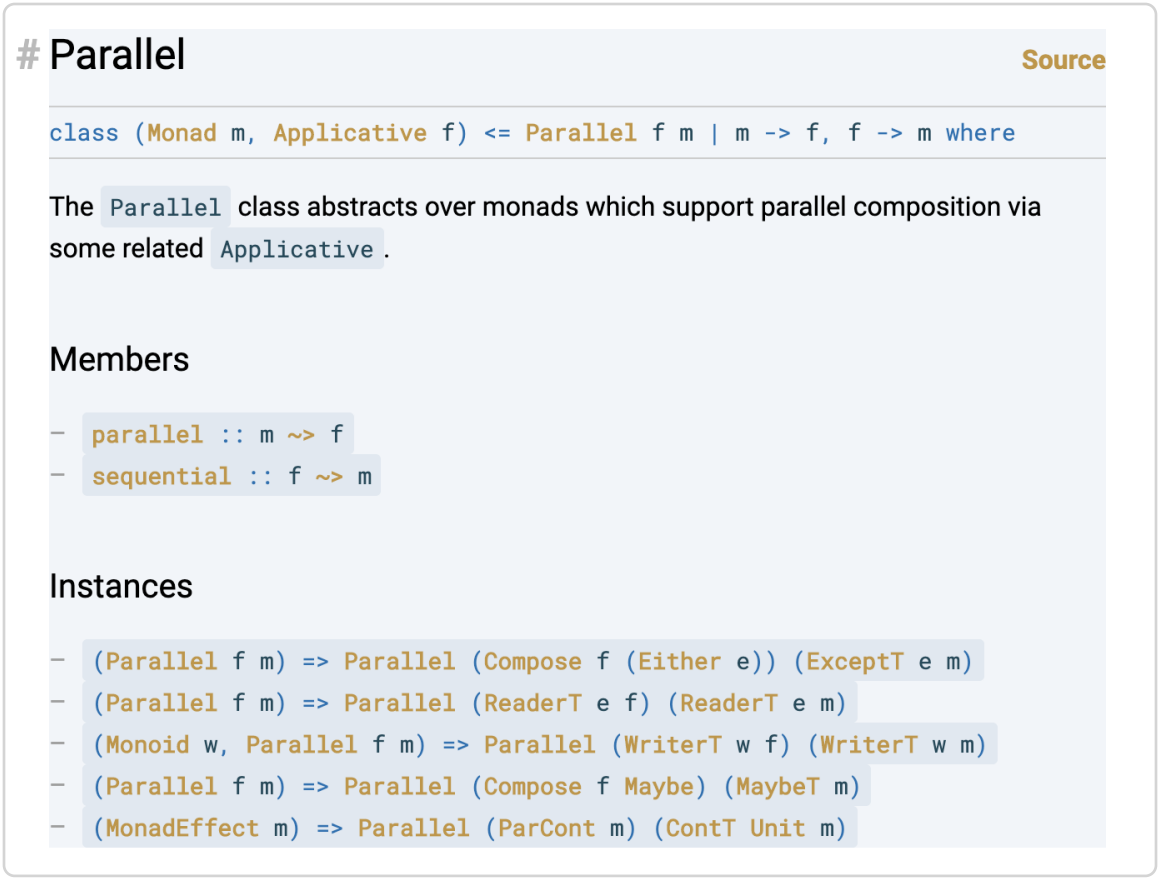
- `Applicative ParAff`
- `(Semigroup a) => Semigroup (ParAff a)`
- `(Monoid a) => Monoid (ParAff a)`
- `Alt ParAff`

  Races effects in parallel. Returns the first successful result or the first error if all fail with an exception. Losing branches will be cancelled.

- `Plus ParAff`
- `Alternative ParAff`
- `Parallel ParAff Aff`

Note that it says that `ParAff` is an "Applicative for running parallel effects". And it also mentions the `Parallel` class.

Looking `Parallel` up on `Pursuit` yields:



Parallel Source

```
class (Monad m, Applicative f) <= Parallel f m | m -> f, f -> m where
```

The `Parallel` class abstracts over monads which support parallel composition via some related `Applicative`.

Members

- `parallel :: m ~> f`
- `sequential :: f ~> m`

Instances

- `(Parallel f m) => Parallel (Compose f (Either e)) (ExceptT e m)`
- `(Parallel f m) => Parallel (ReaderT e f) (ReaderT e m)`
- `(Monoid w, Parallel f m) => Parallel (WriterT w f) (WriterT w m)`
- `(Parallel f m) => Parallel (Compose f Maybe) (MaybeT m)`
- `(MonadEffect m) => Parallel (ParCont m) (ContT Unit m)`

Here are it's Methods:

```
class (Monad m, Applicative f) <= Parallel f m | m -> f, f -> m where
  parallel :: m ~> f ❶
  sequential :: f ~> m ❶
```

❶ Note the use of the Natural Transformation, `~>`, e.g. `m ~> f` is equivalent to `∀ a. m a -> f a`.

So `parallel` lets us take a Monadic Computation, `m` and convert it into an `Applicative`, `f` and `sequential` lets us go back the other way.

Usually, `m` is the `Aff` Monad and the Applicative is `ParAff`.

This may seem confusing but it shouldn't once we look at a few examples:

```
doNothing :: Aff Unit
doNothing = sequential $ parallel $ pure unit
```

All this did was to take the `pure unit` and convert it to `ParAff` using `parallel` and then convert it back using `sequential`.

I called this `doNothing`, but in fact it's doing a lot under the covers. In the case of `parallel`, it simply is

coercing the `Aff Unit` to `ParAff Unit`.

Then `sequential` actually runs the computation in its own `Fiber`. Not much of a computation here, but realize that `sequential` is doing some magic `Fiber` stuff under the covers.

Now, let's imagine we're doing 2 Ajax calls to different APIs. We can do them in serial, i.e. make one API call and then when that finishes, we can make the second.

Or we can do them in parallel:

```
import Affjax as Ajax
import Control.Parallel (parallel, sequential) ❶

type Response = Either Ajax.Error (Ajax.Response String)

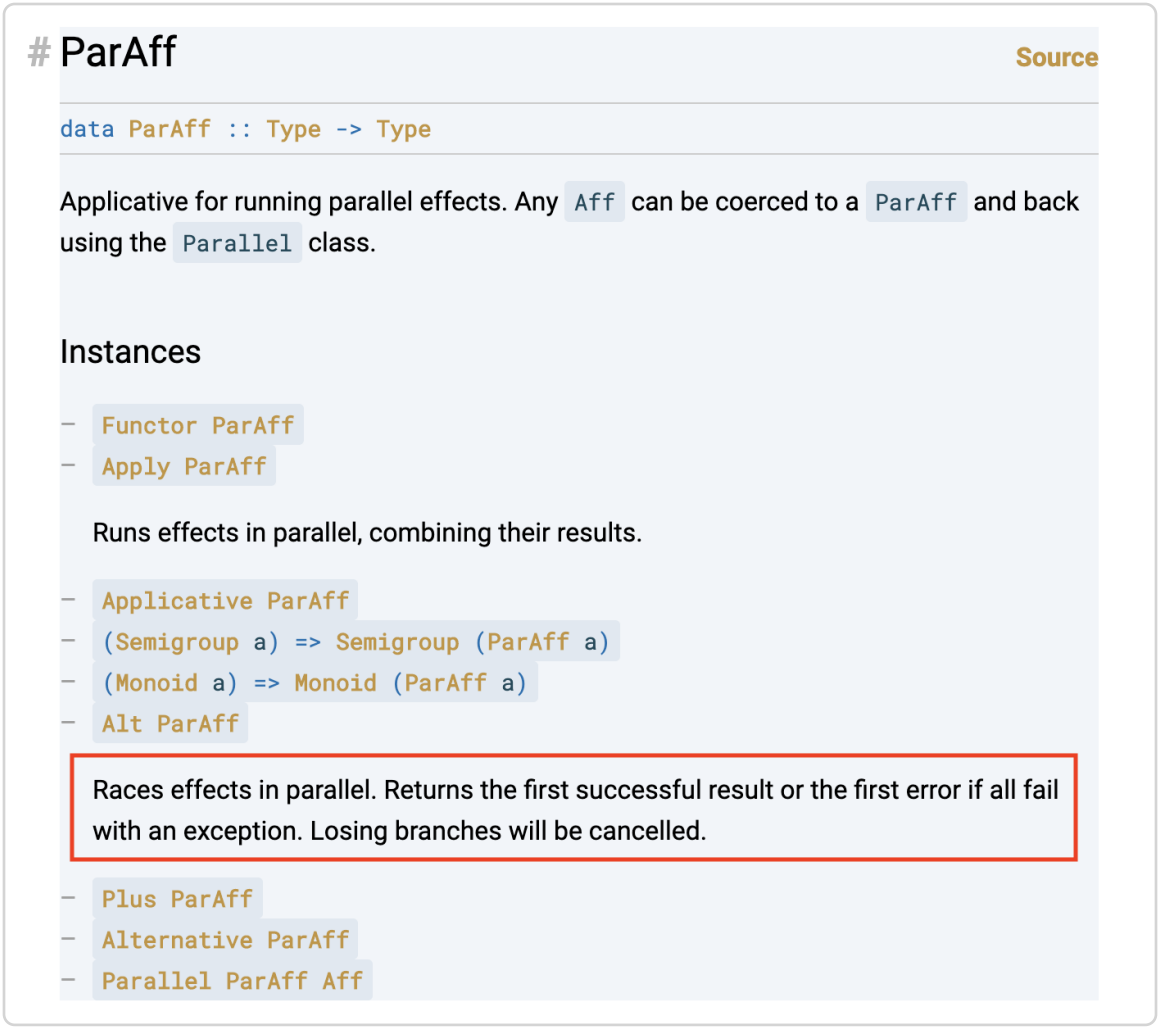
twoRequests :: Aff (Tuple Response Response)
twoRequests =
  sequential $
    Tuple <$> parallel (Ajax.get string "https://foo.com")
          <*> parallel (Ajax.get string "https://bar.com")
```

❶ Import `parallel` and `sequential`.

The 2 different `parallel` calls will run in their own `Fibers` and their results will be combined via `map` and `apply` to `Tuple`. But this won't happen until `sequential` is called.

What if we want to make an Ajax call with a timeout. What we need is a way to race 2 parallel computations and pick the winner, i.e. the one that finishes first.

Let's revisit the `ParAff` docs:



ParAff Source

```
data ParAff :: Type -> Type
```

Applicative for running parallel effects. Any `Aff` can be coerced to a `ParAff` and back using the `Parallel` class.

Instances

- `Functor ParAff`
- `Apply ParAff`

  Runs effects in parallel, combining their results.

- `Applicative ParAff`
- `(Semigroup a) => Semigroup (ParAff a)`
- `(Monoid a) => Monoid (ParAff a)`
- `Alt ParAff`

  Races effects in parallel. Returns the first successful result or the first error if all fail with an exception. Losing branches will be cancelled.

- `Plus ParAff`
- `Alternative ParAff`
- `Parallel ParAff Aff`

Notice the section where it says "Races effects in parallel" and how the "Losing branch will be cancelled". Also, notice how `Alternative` is under that section.

That tells me that we can write the following:

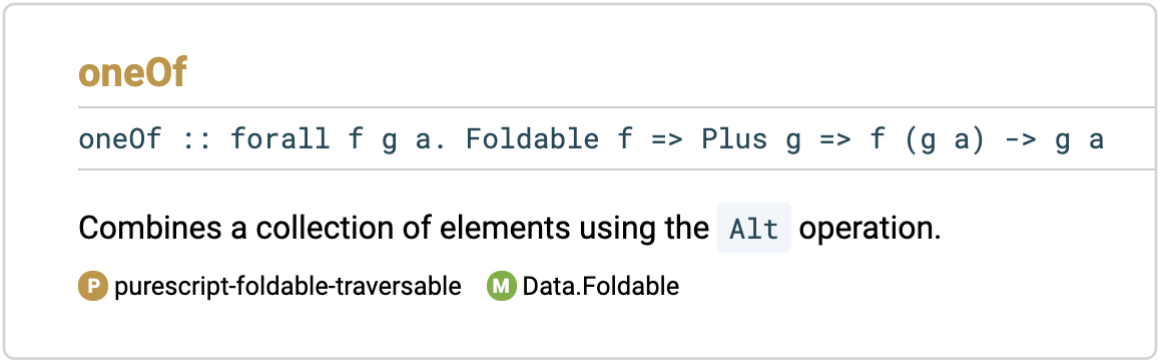
```
requestWithTimeout :: Number -> Aff (Maybe Response)
requestWithTimeout timeout =
  sequential $
        parallel (Just <$> Ajax.get string "https://foo.com") ❶
    <|> parallel (Nothing <$ delay (Milliseconds timeout)) ❷
```

❶ `map` over the Ajax call results to indicate a successful request using `<$>`.

❷ `applyFirst`, i.e. `<$`, to discard the right-hand results and return `Nothing` to indicate that we timed out.

The use of `<|>` races the 2 different `Fibers`. Whichever one finishes first wins and the other is cancelled, hence, their `Canceler` is called.

But what if you want to race a bunch. We can just use `<|>` between them all. Or we could use `oneOf` from `Data.Foldable`. Here's its `Pursuit` docs:



`oneOf` will add the `<|>` between each element of the `Foldable`:

```
requestWithTimeout :: Number -> Aff (Maybe Response)
requestWithTimeout timeout =
  sequential $ oneOf
    [ parallel (Just <$> Ajax.get string "https://foo.com")
    , parallel (Nothing <$ delay (Milliseconds timeout))
    ] ❶
```

❶ `Array` is our `Foldable`.

This code is equivalent to the earlier version that explicitly used `<|>`.

We can make this Function a bit cleaner by factoring out the `parallel`:

```
requestWithTimeout :: Number -> Aff (Maybe Response)
requestWithTimeout timeout =
  sequential $ oneOf $ parallel <$> ❶
    [ (Just <$> Ajax.get string "https://foo.com")
    , (Nothing <$ delay (Milliseconds timeout))
    ]
```

❶ Factored out `parallel` and map it over the `Array`.

To make many calls in `parallel` and keep the results of all of them, we can write code in an Applicative style like:

```
manyRequests :: Aff (Array Response)
manyRequests =
  sequential $
    (\p1 p2 p3 -> [p1, p2, p3]) ❶
      <$> parallel (Ajax.get string "https://foo.com")
      <*> parallel (Ajax.get string "https://bar.com")
      <*> parallel (Ajax.get string "https://baz.com")
```

❶ Our Lambda will collect up the results of each call and put them into an `Array`.

This can be made cleaner by using `parSequence`. This is just like `sequence` from `Data.Traversable` with the notable difference that the computations are performed over the `Traversable` in parallel instead of `serial`:

```
import Control.Parallel (parSequence, parallel, sequential) ❶

manyRequests :: Aff (Array Response)
manyRequests =
  parSequence
    [ Ajax.get string "https://foo.com"
    , Ajax.get string "https://bar.com"
    , Ajax.get string "https://baz.com"
    ]
```

❶ Added `parSequence` to the import list.

This version is so much cleaner.

25.2.5. Using Parallel with Ajax

Let's change our code to send both a test `Student` and test `Teacher` to the Echo Server.

First thing we'll need is a `Student`. Add `testStudent` to your code:

```
testStudent :: Student
testStudent = Student
  { grade: Grade 1
  , teacher: testTeacher
  , gpa: GPA 3.2
  , personal: Personal {
      height: Centimeters 107.9
    , weight: Kilograms 17.9
    , age: Years 5
    }
  }
```

Next, comment out the whole `test` for now. We'll use it as reference. Don't worry about the unused imports.

Then comment out `processAjaxResult`. We won't need it anymore, but may find it useful for reference.

Now, code a brand new `test` Function to send both `testStudent` and `testTeacher` in parallel to the Echo Server. Use what we've learned in the previous section to help you code this. Hint: look at the `parSequence` example.

Code this before reading on.

Coding this one step at a time, we start with `parSequence`:

```
test :: Effect Unit
test = launchAff_ do
  parSequence ???
```

Next, I need a `Traversable`, which will be an `Array` and each call to `Ajax.post` will be in there:

```
test :: Effect Unit
test = launchAff_ do
  parSequence
    [
      Ajax.post ResponseFormat.json
        "http://localhost:3000/"
        $ Just $ RequestBody.String $ encodeJSON testTeacher
    , Ajax.post ResponseFormat.json
        "http://localhost:3000/"
        $ Just $ RequestBody.String $ encodeJSON testStudent
    ]
```

Now can you see any improvements?

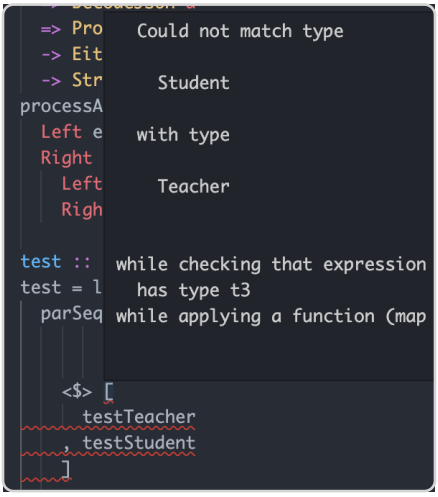
I see a lot of duplicate code. So go ahead and factor the duplicate code before reading any further.

We're going to factor our the call to `Ajax.post` and all of it's common Parameters. We'll use a Lambda:

```
test :: Effect Unit
test = launchAff_ do
  parSequence $ (\o -> Ajax.post ResponseFormat.json
                       "http://localhost:3000/"
                       $ Just $ RequestBody.String $ encodeJSON o) ❶
    <$> [ -- COMPILER ERROR!!
      testTeacher
    , testStudent
    ]
```

❶ This Lambda is mapped over our `Array`. Here `o` is the object to encode.

The compiler error is:



Can you see where I went wrong?

The compiler is saying that it cannot match `Student` with `Teacher`. I've made the mistake of placing `Teacher` and `Student` in the same `Array`. `Arrays` are Homogeneous, i.e. all elements must be of the same Type.

We need a way to convert them both to the same Type. Can you think of a way?

Looking at the Lambda may provide a clue. How about now?

Turns out that we can call `encodeJSON` on each of them and then the `Array` will contain elements of Type `Json`:

```
test :: Effect Unit
test = launchAff_ do
  parSequence $ (\json -> Ajax.post ResponseFormat.json ❶
                  "http://localhost:3000/"
                  $ Just $ RequestBody.String json) ❷
    <$> [
      encodeJSON testTeacher ❸
    , encodeJSON testStudent ❸
    ]
```

❶ Renamed Lambda Parameter to `json`.

❷ We can pass the `json` directly as the body of the `POST`.

❸ We encode each different element of the Array to homogenize them, i.e. make them the same Type.

The next step is to retrieve the results from the resulting `Array`. First, extract the results of the `parSequence` call and then add a `pure unit` at the end of the `do` block to keep the compiler happy for the moment.

Then read on.

Here's those changes:

```
test :: Effect Unit
test = launchAff_ do
  results <- parSequence $ (\json -> Ajax.post ResponseFormat.json ❶
       "http://localhost:3000/"
       $ Just $ RequestBody.String json)
    <$> [
      encodeJSON testTeacher
    , encodeJSON testStudent
    ]
  pure unit ❷
```

❶ Extract the results from the `Aff (Array a)`.

❷ This is here for now to keep the compiler happy. We'll probably delete this later.

Next, ignoring the compiler warning, we need to process `results`, but before we can do that we need to know `results's` Type.

How can we figure that out?

If we look up `parSequence` in `Pursuit` we see it returns an `m (t a)` where `m` is the Monad or in our case `Aff` and `t` is the `Traversable` which is an `Array` and `a` is something. That means it returns an `Aff (Array`

`a)`.

Once we extract from `Aff`, we have `Array a` as the Type for `results`.

Now, we need to figure out what `a` is.

We can figure that out by looking at `Pursuit` again and seeing what `Ajax.post` returns. When we do we see it returns `Aff (Either Error (Response a))`. But `parSequence` eliminates the `Aff` part and so we have an `Array` of `Either Error (Response a)`, giving us:

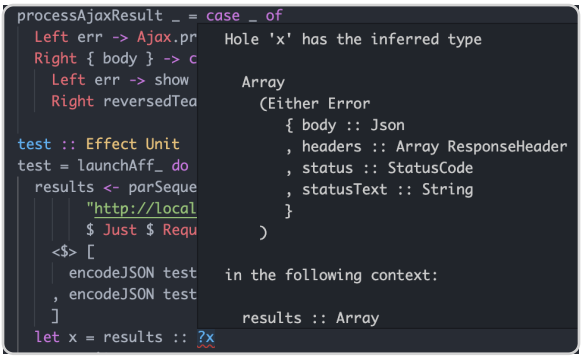
```
results :: Array (Either Error (Response a))
```

We can also create a Type Hole to figure this out:

```
test :: Effect Unit
test = launchAff_ do
  results <- parSequence $ (\json -> Ajax.post ResponseFormat.json
        "http://localhost:3000/"
        $ Just $ RequestBody.String json)
    <$> [
      encodeJSON testTeacher
    , encodeJSON testStudent
    ]
  let x = results :: ?x ❶
  pure unit
```

❶ Added dummy `let` so we could add a Type Hole to determine the Type of `results`.

Here's what the compiler tells us:



The Record in the second Parameter to `Either` is `Response`. So the Type-hole approach gave us what we needed but we had to remember that `Response` is a Record.

At this point, we're extremely confident that we have an `Array` of `Eithers`.

We want to process the `Eithers`, i.e. results of the Ajax calls, that's stored in this `Array`. But one or both could have failed. And if they didn't fail, we really only care about the `body` from the `Response` Record.

How do would you proceed?

Well, we want to convert the `Array (Either Error (Response a))` to `Either Error (Array (Response a))`. This way we either have an `Error` or an `Array` of `Responses`.

Do you remember something that does this?

If not, how about something that takes a `t (m a)` and makes it a `m (t a)`?

If still not, then search `Pursuit` for `t (m a) -> m (t a)` and see what you get.

I found `sequence`. Add that to your test code and then extract the bodies from the resulting `Either`. Do this before reading on.

So, here's `sequence` added:

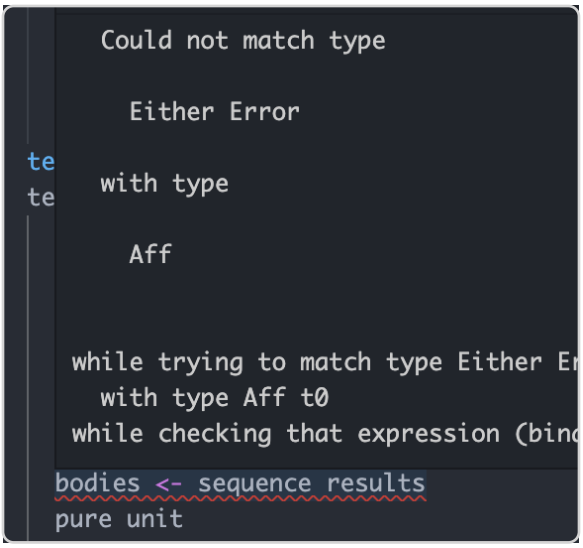
```
import Data.Traversable (sequence) ❶

test :: Effect Unit
test = launchAff_ do
  results <- parSequence $ (\json -> Ajax.post ResponseFormat.json
        "http://localhost:3000/"
        $ Just $ RequestBody.String json)
    <$> [
      encodeJSON testTeacher
    , encodeJSON testStudent
    ]
  bodies <- sequence results -- COMPILER ERROR!! ❷
  pure unit
```

❶ Import `sequence`.

❷ Added `sequence`.

Here's the compiler error:



Can you see where I went wrong?

We expected `sequence` to return an `Either` but I'm using its results as an `Aff`. I should have just used a `let` here:

```
test :: Effect Unit
test = launchAff_ do
  results <- parSequence $ (\json -> Ajax.post ResponseFormat.json
        "http://localhost:3000/"
        $ Just $ RequestBody.String json)
    <$> [
      encodeJSON testTeacher
    , encodeJSON testStudent
    ]
  let bodies = sequence results
  pure unit
```

I forgot to extract the `body` from the `Response`. If you haven't done that part yet, then do so now before reading further.

We can extract the `body` using `map`:

```
test :: Effect Unit
test = launchAff_ do
  results <- parSequence $ (\json -> Ajax.post ResponseFormat.json
        "http://localhost:3000/"
        $ Just $ RequestBody.String json)
    <$> [
      encodeJSON testTeacher
    , encodeJSON testStudent
    ]
  let bodies = map (_.body) <$> sequence results ❶
  pure unit
```

❶ Map twice. The first one, `<$>`, maps into the `Either` and the second one, `map`, maps into the `Array`.

So, you've probably lost track of the Types at this point, but can you figure out the Type for `bodies`?

If not, use a Type Hole to figure it out before reading on.

Thanks to the Type Hole, we find out that `bodies` is `Either Error (Array Json)`. We can just deal with this with a `case` expression:

```
test :: Effect Unit
test = launchAff_ do
  results <- parSequence $ (\json -> Ajax.post ResponseFormat.json
        "http://localhost:3000/"
        $ Just $ RequestBody.String json)
    <$> [
      encodeJSON testTeacher
    , encodeJSON testStudent
    ]
  log $ case map (_.body) <$> sequence results of ❶ ❷
    Left err -> Ajax.printError err ❸
    Right [teacherJson, studentJson] -> ??? ❹ ❺
```

❶ Using `case` to get inside the `Either`.

❷ We're going to log the result, so we're preemptively adding `log` here.

❸ Remember that this error is just the first error we get from `sequence`. That's because `Either` Short-circuits once we hit our first `Left`. If we want all of the errors, we'd have to code this differently.

❹ We must exactly match the `Array` elements here.

❺ Since going to `log` like we did before, we no longer need the `pure unit` at the bottom, which is why it's been deleted.

Next, we need to process the 2 different `Json` Values differently because one is `Teacher` encoded and the other is `Student` encoded.

Look at `processAjaxResult` to see what we can steal to decode them into a `Teacher` and `Student` and then turn those into `Strings`.

Once you see what to do, change the test code before moving on.

We're going to use `decodeJson` and `show`:

```
test :: Effect Unit
test = launchAff_ do
  results <- parSequence $ (\json -> Ajax.post ResponseFormat.json
        "http://localhost:3000/"
        $ Just $ RequestBody.String json)
    <$> [
      encodeJSON testTeacher
    , encodeJSON testStudent
    ]
  log $ case map (_.body) <$> sequence results of -- COMPILER ERROR!!
    Left err -> Ajax.printError err
    Right [teacherJson, studentJson] ->
          show (decodeJson teacherJson :: _ Teacher) <> "\n\n" ❶
      <>  show (decodeJson studentJson :: _ Student)
```

❶ We decode and show each with a couple of newlines in between.

The compiler error says:

```
  A case expression could not be determined to cover all inputs.
  The following additional cases are required to cover all inputs:

    (Right _)

  Alternatively, add a Partial constraint to the type of the enclosing value.

while applying a function $__unused
  of type Partial => t0 -> t0
  to argument case ((map (map (...))) (sequence results)) of
                (Left err) -> printError err
                (Right [ teacherJson, studentJson ]) -> (append (...)) ((...)
while checking that expression $__unused (case ((map (...)) (sequence results))
      <>  show (decodeJson studentJson :: _ Student)
```

I missed the last case here, i.e. when the `Array` isn't 2 long. We know this will always be the case, but we should probably put a nice message if the code changes up above but the `case` expression does not:

```
test :: Effect Unit
test = launchAff_ do
  results <- parSequence $ (\json -> Ajax.post ResponseFormat.json
        "http://localhost:3000/"
        $ Just $ RequestBody.String json)
    <$> [
      encodeJSON testTeacher
    , encodeJSON testStudent
    ]
  log $ case map (_.body) <$> sequence results of
    Left err -> Ajax.printError err
    Right [teacherJson, studentJson] ->
          show (decodeJson teacherJson :: _ Teacher) <> "\n\n"
      <>  show (decodeJson studentJson :: _ Student)
    Right _ ->
      "The number of Ajax calls is different than what's being processed."
```

Now go ahead and make this change and then clean up the unused imports.

And finally, run this to see what you get.

And of course, `spago` is complaining:

```
[error] Some of your project files import modules from packages that are not
in the direct dependencies of your project.
To fix this error add the following packages to the list of dependencies in
your config:
- parallel
```

Add this and run it again. When I do, I got:

```
(Right (Teacher { grades: [Preschool,Kindergarten,(Grade 1)],
numberOfStudents: 23, personal: (Personal { age: (Years 31), height:
(Centimeters 162.56), weight: (Kilograms 63.5) }), status: NonTenured }))

(Right (Student { gpa: (GPA 3.2), grade: (Grade 1), personal: (Personal { age:
(Years 5), height: (Centimeters 107.9), weight: (Kilograms 17.9) }), teacher:
(Teacher { grades: [Preschool,Kindergarten,(Grade 1)], numberOfStudents: 23,
personal: (Personal { age: (Years 31), height: (Centimeters 162.56), weight:
(Kilograms 63.5) }), status: NonTenured }) }))
```

We've decode the reversed-keys version of our Types every way we planned to except for using Javascript.

But, we'll have to wait until after we learn how to call Javascript from our PureScript code to do that.

Don't forget to switch terminal windows using the VSCode terminal pull-down and kill the Echo Server with a `Ctrl-C`.

Chapter 26. Foreign Function Interface (FFI)

There is a huge Javascript ecosystem with libraries to do all sorts of things. We've seen wrappers on the Node built-in modules, e.g. `Node.FS.Async`. You might wonder how these libraries interface to PureScript.

There's a mechanism called ***Foreign Function Interface***, or ***FFI*** for short, that allows PureScript to directly interface with Javascript.

Since PureScript doesn't do everything, sometimes we're forced to write Javascript or interface to a library that hasn't been wrapped yet.

In these cases, we need to understand how FFIs work.

Let's look at a simple example to call `join` from Node's `path` module:

```
module NodePath where

type FilePath = String

foreign import joinPath :: FilePath -> FilePath -> FilePath ❶
```

❶ `foreign import` specifies that the Function is written in a Javascript.

Let's assume that this code is in a file called `NodePath.purs`. That means that our Javascript code MUST be in a file called `NodePath.js` and it MUST be in the SAME directory as `NodePath.purs` or your code won't compile.

Here's how we'd write the Javascript code:

```
const path = require('path');

exports.joinPath = function(start) {
  return function(end) {
    return path.join(start, end);
  };
};
```

Notice that the Javascript Function is manually Curried. The top-level Function takes one Parameter and returns a Function that takes one Parameter and returns the result of Node's `join`.

That description matches the PureScript definition:

```
foreign import joinPath :: FilePath -> (FilePath -> FilePath) ❶
```

❶ Added redundant Parentheses to help see the Currying.

We can also write the Javascript using a more modern syntax that makes the Currying easier to see and to code:

```
const path = require('path');

exports.joinPath = start => end => {
  return path.join(start, end);
};
```

And now when we run the following test code:

```
test :: Effect Unit
test = do
  log $ joinPath "abc" "xyz.txt"
```

we get the following output on the console:

```
abc/xyz.txt
```

We can also write our Javascript in a more traditional sense, i.e. Uncurried:

```
const path = require('path');

exports.joinPathImpl = (start, end) => {
  return path.join(start, end);
};
```

But now, we'll need use special helper Functions for calling it:

```
import Data.Function.Uncurried (Fn2, runFn2) ❶ ❷

foreign import joinPathImpl :: Fn2 FilePath FilePath FilePath ❸ ❹

joinPath :: FilePath -> FilePath -> FilePath ❺
joinPath p1 p2 = runFn2 joinPathImpl p1 p2 ❻ ❼
```

❶ Import `Fn2` Type for Foreign Functions that take 2 Parameters.

❷ Import `runFn2` Function for calling Foreign Functions of 2 Parameters that are Uncurried.

❸ Changed from a normal Function Type Signature to the `Fn2` Type.

❹ Renamed to have the `Impl` suffix. This is one of a couple idiomatic naming convention for FFIs.

❺ This is the Function that is called from other PureScript Functions.

❻ Use `runFn2` to call the Uncurried Javascript Function.

❼ DO NOT CODE THESE AS POINT-FREE. When using `runFn0` through `runFn10`, the compiler will inline these calls when the Function Parameters are fully saturated, i.e. NOT Partially Applied.

Running this code still produces the same output:

```
abc/xyz.txt
```

Either method is fine, but when you're calling a Javascript Library where none of the Functions are Curried, using `Data.Function.Uncurried` is the easiest solution.

26.1. FFIs That Return PureScript Types

You never want to call PureScript Functions directly from Foreign code. For example, here is how you COULD create a `Maybe`:

```
Data_Maybe.Just.create(x)
Data_Maybe.Nothing.value
```

Both of these ways are brittle for 2 reasons. First, the compiler does dead code removal to reduce the size of your generated Javascript code. If your code didn't use `Just` for some reason, then `Data_Maybe.Just.create` could get eliminated during dead code removal.

And another reason is that the code generator may generate PureScript code differently with a new release of the compiler. While `Data_Maybe.Just.create` may work today, it isn't guaranteed to work in the future.

The way you solve this is to pass any PureScript Functions that your Javascript code may need to call as Parameters to that Function, for example:

```
exports.doSomethingImpl = (just, nothing, p1) {
  if (result = doSomething(p1))
    return just(result);
  else
    return nothing;
};
```

Since the compiler knows how to call Just and Nothing, when you pass them, they will always pass the current way to call them and since your using Just and Nothing, that code will never be dead code eliminated.

This rule holds for all PureScript Functions including your own.

26.2. FFIs and Effect

For Javascript Functions that perform real-world Effects, we need to make sure we create the Javascript code properly.

Another very important to thing to keep in mind is that it's not always obvious when a Javascript Function is performing real-world Effects. Sometimes you have to really investigate.

For example, here's the Node API definition for resolve:

path.resolve([...paths])

Added in: v0.3.4

- `...paths` <string> A sequence of paths or path segments
- Returns: <string>

The `path.resolve()` method resolves a sequence of paths or path segments into an absolute path.

The given sequence of paths is processed from right to left, with each subsequent `path` prepended until an absolute path is constructed. For instance, given the sequence of path segments: `/foo`, `/bar`, `baz`, calling `path.resolve('/foo', '/bar', 'baz')` would return `/bar/baz` because `'baz'` is not an absolute path but `'/bar' + '/' + 'baz'` is.

If, after processing all given `path` segments, an absolute path has not yet been generated, the current working directory is used.

The resulting path is normalized and trailing slashes are removed unless the path is resolved to the root directory.

Zero-length `path` segments are ignored.

If no `path` segments are passed, `path.resolve()` will return the absolute path of the current working directory.

Reading through this, it's not clear whether this Function is doing any real-world Effects. But the following worries me:

```
The path.resolve() method resolves a sequence of paths or path segments into
an absolute path.
```

How does it figure out an absolute path from a relative one without consulting the Operating System for the Current Directory. If it does, then that would warrant an Effect in PureScript. Or I may just be interpreting this wrong.

My gut is telling me that this probably is going to need to run in Effect, but the following bit of documentation slightly further down the page from the previous seals the deal:

```
If, after processing all given path segments, an absolute path has not yet
been generated, the current working directory is used.
```

That's all we need to know. It has to call the Operating System to find out what the "current working directory" is and therefore is NOT Pure.

Here's how we'd write the Javascript code so this can run in Effect:

```
const path = require('path');

exports.resolveImpl = (paths, path_) => {
  return () => path.resolve.apply(this, paths.concat(path_)); ❶ ❷
};
```

❶ Effects return Javascript FUNCTIONS not results.

❷ Concatenate the paths together to conform to resolve's terrible interface.

There's a lot to unpack here. The first thing to note is that resolve takes any number of arguments:

```
path.resolve([...paths])
```

We can't do this in PureScript. Good thing too. This is such a terrible interface to understand because of this. It turns out that the paths are processed from RIGHT to LEFT.

This means that the LAST path will be the path that you're trying to resolve using the leftmost paths. That's why this Function takes 2 Parameters, paths and path.

Here's the corresponding PureScript code:

```
foreign import resolveImpl
  :: Fn2 (Array FilePath) FilePath (Effect FilePath) ❶

resolve:: Array FilePath -> FilePath -> Effect FilePath ❶
resolve paths path = runFn2 resolveImpl paths path
```

❶ Remember that this runs in the Effect Monad.

And then some test code to use this:

```
test :: Effect Unit
test = do
  path <- resolve [ "/foo", "/bar" ] "baz" ❶ ❷
  log path
```

❶ Remember, resolve runs in Effect so we need to extract the result from the Monad.

❷ This is the example given in the Node API documentation for resolve. This way we'll know for sure that our code is working.

When we run this we get:

```
/bar/baz
```

This is exactly what the Node API documentation example says that we should get back from this call.

Let's return to something I glossed over, but is an important thing to understand.

Code that runs in Effect returns Javascript Functions:

```
const path = require('path');

exports.resolveImpl = (paths, path_) => {
  return () => path.resolve.apply(this, paths.concat(path_)); ❶
};
```

❶ Returning a Function since resolveImpl runs in Effect.

Do remember writing runReaderT and runWriterT, etc. for their respective Monads?

Well, there is no run Function for Effect. It's run by the compiler generated code. And now we can see that main returns a Javascript Function since it's Type is Effect Unit.

We can think of Effect as a set of instructions of what should be run at some point after our program runs. When our programs run, it calls our main, which does not execute our program, but, instead, ***purely*** generates an extremely complex set of instructions to be run later in the form of a single, massive, Javascript Function.

This is how we can stay in the Pure Functional world while still effecting the outside world. We create code in the Pure world that generates instructions to be run later in the Impure world.

26.3. FFIs and Aff

There are 2 ways to call a Javascript Function Asynchronously. The hard way and the easy way.

Let's start with the easy way. Here's how we can call our `resolve` Javascript Function Asynchronously:

```
import Effect.Aff (Aff, launchAff_, makeAff, nonCanceler)
import Effect.Class (liftEffect)
import Effect.Class.Console (log)

test :: Effect Unit
test = launchAff_ do ❶
  aPath <- liftEffect $ resolve [ "/foo", "/bar" ] "baz" ❷
  log aPath
```

❶ We're running in `Aff` thanks to `launchAff_`

❷ We convert `resolve` from an `Effect` to an `Aff` via `liftEffect` since we're running in `Aff`.

This case was really easy. But what if our Javascript Function takes a long time to complete and, therefore, has a `Callback` because we cannot afford to block the current thread.

This was not the case with `resolve` since it's a very fast Function even if it calls the Operating System for the current working directory.

But, `readTextFile` from Node's `fs` module is not fast. It could take a very long time, in computer time, to read a file. So, it needs a Callback.

But we don't want a Callback like we have in `Node.FS.Async` and we'd like to run it in `Aff` not `Effect`.

While this is the hard way, relatively speaking, we can do all of this nicely with a neat little Function from `Effect.Aff.Compat` called `fromEffectFnAff`.

Here's how we'd write our Javascript code:

```
exports._readTextFile = path => (onError, onSuccess) => {
  fs.readFile(path, { encoding: 'UTF-8' }, (err, data) =>
    err ? onError(err) : onSuccess(data));
  return (cancelError, onCancelerError, onCancelerSuccess) =>
    onCancelerSuccess();
};
```

There's a lot of code to look at here. So let's dissect it one piece at a time.

```
exports._readTextFile = path => (onError, onSuccess) =>
```

First, `_readTextFile` is an alternative naming convention for Javascript Functions similar to `readTextFileImpl`. I think I like this convention better.

Second, `_readTextFile` takes a `path` to read. It returns a Function that expects 2 Parameters, which are both Callback Functions.

`onError` is to be called when an error occurs in the Asynchronous operation. It will be called with the Javascript Error, which has an equivalent Type in `Effect.Exception`.

`onSuccess` is to be called passing the result when it succeeds.

We can see this in the next line of code:

```
  fs.readFile(path, { encoding: 'UTF-8' }, (err, data) =>
    err ? onError(err) : onSuccess(data));
```

On this line, we call Node's `readFile` with the `path` and an Encoding of `UTF-8`. The final Parameter is the Callback.

The Callback takes 2 Parameters. If the first Parameter isn't `undefined` or `null`, then an error occurred and `onError` is called with that error.

If the first Parameter doesn't have a value, then the second Parameter will have the file data in it and `onSuccess` is called passing the file data.

Here's the final line of code:

```
  return (cancelError, onCancelerError, onCancelerSuccess) =>
    onCancelerSuccess();
```

`_readTextFile` returns a `Canceler` Function. This Function will be called by the `Fiber` Manager if we've been canceled.

The first Parameter to this Function, `cancelError`, is the `Error` passed in from whatever canceled us. In an earlier Chapter, we called `killFiber` to cancel:

```
kill :: ∀ a. Fiber a -> Aff Unit
kill = killFiber (error "Killing you softly...") ❶
```

❶ `killFiber` takes an `Error`. This will be passed to the Canceler.

The second Parameter to the Canceler Function, onCancelerError, is the Function that our Canceler will call if we have any errors in whatever cleanup code we're running.

And the third and final Parameter, onCancelerSuccess, is the Function that our Canceler will call if it did its cleanup successfully. In this case, we have no cleanup and we just return success.

Now, here's the PureScript code to interface with _readTextFile:

```
import Effect.Aff.Compat (EffectFnAff, fromEffectFnAff) ❶ ❷

foreign import _readTextFile :: String -> EffectFnAff String ❸

readFile :: String -> Aff String
readFile path = fromEffectFnAff $ _readTextFile path ❹
```

❶ Import EffectFnAff, which is the Type that our Javascript code returns.

❷ Import the helper, fromEffectFnAff that will convert the EffectFnAff to an Aff.

❸ Define our Foreign Function. Note the return Type isn't String, but EffectFnAff String.

❹ Call our Foreign Function using our helper to make it an Aff.

Now, here's our test code:

```
test :: Effect Unit
test = launchAff_ do
  contents <- readFile "test.txt"
  log contents
```

When we create a file called test.txt and put This is a test in it, this code will display:

```
This is a test
```

Chapter 27. Coding With FFIs

Return to the project that you've been coding in and create a file called `Ch27a.purs` and add the following:

```
module Ch27a where

import Prelude ❶

import Effect (Effect)
import Effect.Class.Console (log) ❷

test :: Effect Unit
test = do
  log "placeholder" ❸
```

❶ Once again we are not writing anything from Prelude so we'll just import all of it.

❷ We're now using this version of `log` so we can use it in `Effect` and `Aff`.

❸ We'll delete this soon enough. It's just here to keep the compiler happy.

Next, change your `Main` module to import `Ch27a` and call its `test` Function:

```
module Main where

import Prelude
import Effect (Effect)

import Ch27a as Ch27a

main :: Effect Unit
main = Ch27a.test
```

If you run:

```
npx spago run
```

you should see it build and print:

```
[info] Build succeeded.
placeholder
```

27.1. Wrapping an NPM library

Make sure that you're in your project's directory and then install the `chalk` NPM library with the command:

```
npm install --save chalk
```

The `chalk` library adds ANSI Color escape sequences to `Strings` for displaying text color, background color and styles on the Terminal.

For testing, it's best to NOT use the Terminal built into VSCode since at the time of this writing it doesn't support all of the `chalk` features.

If you're on Windows, you should use Windows Terminal and NOT `cmd.exe`.

The next step is to look at some code examples from the NPM documentation on how to use the `chalk` package:

```
const chalk = require('chalk');
const log = console.log;

// Combine styled and normal strings
log(chalk.blue('Hello') + ' World' + chalk.red('!'));
```

This seems simple enough. But how do we do multiple styles, e.g. `blue` for text color and `bgRed` for background color?

Scrolling down on the NPM document page, I"ve found:

```
chalk.red.bold.underline('Hello', 'world');
```

Okay, so this is NOT simple. This is a Builder Pattern and makes our job more difficult.

`chalk` is a Function with an initial state of no styles.

`chalk.red` is a also Function but with `red` added to the internal state.

`chalk.red.bold` is a Function but with `bold` added to the internal state.

Javascript can do this kind of thing but PureScript cannot.

How can we write this line of code differently in Javascript such that it will make our lives much easier?

If you don't know Javascript well or you've forgotten, the following 2 lines are equivalent:

```
chalk.red.bold.underline('Hello', 'world');
chalk["red"]["bold"]["underline"]('Hello', 'world');
```

The Array syntax will allow our PureScript code to send an `Array` of `Strings` to our Javascript code where we can loop through the list building these styles up one at a time and then call the final Function.

Create a file called `Ch27a.js` in `src` since Foreign Functions must be implemented in a file of the same name as the PureScript file and in the same directory.

Start with the following code:

```
const chalk = require('chalk'); ❶

exports._chalk = (styles, str) => { ❷ ❸ ❹
  // finish coding here
}
```

❶ Include the `chalk` module.

❷ `styles` is an `Array String`.

❸ `str` is the `String` we're colorizing

❹ Notice that this Function is Uncurried.

Take the `styles` and combine them to create a Function that we can eventually call with `str`, for example, if `styles` is `["bold", "red"]`, then we can build a Function `chalk["bold"]["red"]`.

Finish coding this Function before you read on.

The first thing we're going to need to do is loop through the `styles`:

```
const chalk = require('chalk');

exports._chalk = (styles, str) => {
  for (var i = 0, c = chalk; i < styles.length; ++i) ❶
    // ???
}
```

❶ We start `c` equal to `chalk`, which is a Function that, if called, will have NO styling.

Next, we'll add the current style based on our index, `i`:

```
const chalk = require('chalk');

exports._chalk = (styles, str) => {
  for (var i = 0, c = chalk; i < styles.length; ++i)
    c = c[styles[i]]; ❶ ❷
    // ???
}
```

❶ `styles[i]` is the current style we're working with.

❷ We MUTATE `c` setting it equal to the OLD `c` with the current style applied.

At the end of our loop, `c` will contain all of the styles. So, all we'll need to do is call `c` with `str` and return that:

```
const chalk = require('chalk');

exports._chalk = (styles, str) => {
  for (var i = 0, c = chalk; i < styles.length; ++i)
    c = c[styles[i]];
  return c(str);
}
```

We now have our Javascript code that we'll want to call using `Data.Function.Uncurried`.

Return to `Ch27a.purs` and import that module and the appropriate versions of `Fn` and `runFn` Functions before you read further.

Since our Javascript Function takes 2 Parameters we'll import the following:

```
import Data.Function.Uncurried (Fn2, runFn2)
```

Add the following Type Alias to your code:

```
type Style = String
```

Write the `foreign import` declaration for `_chalk` and then return here.

Here's that declaration:

```
foreign import _chalk :: Fn2 (Array Style) String String ❶
```

❶ The first Parameter to `Fn2` matches the Type for `styles`. The second Parameter is the Type for `str`. The final Parameter is the Type of the return Value.

Now, write a PureScript friendly version of `_chalk`. Call it `chalk` and remember to NOT write it Point-free form so that the compiler can inline the code, i.e. optimize it by not having to make a call to the Function.

Do this before reading further.

We start with the Type Signature:

```
chalk :: Array Style -> String -> String
```

Notice how these Parameters correspond to the Parameters in `Fn2` used for `_chalk`.

Next we add the call to the Javascript code via `runFn2`:

```
chalk :: Array Style -> String -> String
chalk styles str = runFn2 _chalk styles str ❶
```

❶ NOT Point-free to allow inlining.

Notice how the first Parameter to `runFn2` is the Javascript Function. The remaining Parameters are the Parameters to pass to the Javascript Function.

Now, we can test our code. Add the following test code:

```
test :: Effect Unit
test = log $ chalk [ "red", "bold" ] "Test" <> chalk [ "red" ] "Test"
```

Build your code with:

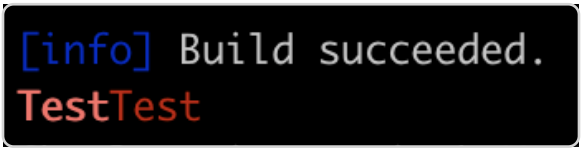
```
npx spago build
```

You should get:

```
[error] Some of your project files import modules from packages that are not
in the direct dependencies of your project.
To fix this error add the following packages to the list of dependencies in
your config:
- functions
```

Fix this error by adding the missing package to `spago.dhall`.

Now run the program in a Terminal Window, i.e. NOT in VSCode's terminal since it doesn't support all ANSI codes, it should produce:



Try it and see what you get. Your mileage may vary due to differing support for ANSI codes in Terminal. I'm run this on a Mac using `iTerm2`. If you are on a Mac and run this in the standard `Terminal`, it'll work but the bold won't be as obvious. In the VSCode Terminal, it looks like the non-bold.

Now, look at the code. Can you see any problems? Any place for future problems or improvements we can make?

If you cannot, I'll demonstrate what I see.

Change the code to be:

```
test :: Effect Unit
test = log $ chalk [ "red", "bold" ] "Test" <> chalk [ "red " ] "Test"
```

Can you see what I changed?

I added a space right after the second `red`. Can you imagine what's going to happen?

Try making this change and running your program and see if you're guess is right.

Your program should crash with a Stack Trace. Try it.

THIS IS A PROBLEM. We have to be careful that Javascript doesn't drag down our PureScript code.

We have a great language like PureScript with Type Safety and we're using `Strings`, which leaves us vulnerable to all the same problems we gladly left behind when we abandoned Javascript.

We should be leveraging the Type System to protect us from ourselves.

The first thing we could do is to create helper Functions so we didn't have to worry about mistyping:

```
red :: Style
red = "red"

bold :: Style
bold = "bold"

test :: Effect Unit
test = log $ chalk [ red, bold ] "Test" <> chalk [ red ] "Test"
```

While this is better, it doesn't prevent the following:

```
test :: Effect Unit
test = log $ chalk [ red, bold ] "Test"
  <> chalk [ red, "crash" ] "Test" ❶
```

❶ Added `"crash"`.

We need a Type that is not a `String`, but can be converted into a `String` for `chalk` to use. How might we accomplish this?

If you thinking about using `newtype`, let's see how that might work:

```
newtype Style = Style String

red :: Style
red = Style "red"

bold :: Style
bold = Style "bold"
```

This certainly protects us from random `Strings` thanks to the `newtype`, i.e. `Style`.

But how do we pass an `Array Style` to our Javascript Function that's expecting an `Array String`?

Way back when we learned about `newtype`, I mentioned that there was some optimizations that the compiler could make for `newtypes`. One of those optimizations is that the `newtype` is represented internally at runtime as the underlying Type.

We can see this by changing our test code to add the following:

```
import Debug (spy) ❶

newtype Style = Style String

red :: Style
red = Style "red"

test :: Effect Unit
test = do
  let x = spy "red" red ❷
  pure unit
```

❶ Import `spy` and add `debug` to `spago.dhall`.

❷ Print the internal representation of `red`.

This outputs:

```
red: 'red'
```

`red` is stored as a Javascript String. That means we don't have to do anything special with `Style` since it's just a `newtype`. This also means that if we pass an `Array Style` it's going to pass `Array String` to our Javascript Function.

I've setup a GitHub Gist of a PureScript module that contains all of the valid the Styles. You can find it here at this link, ChalkStyles.purs. Click it.

If you don't have access to GitHub, here's the code:

```
module ChalkStyles where

newtype Style = Style String

black :: Style
black = Style "black"

red :: Style
red = Style "red"

green :: Style
green = Style "green"

yellow :: Style
yellow = Style "yellow"
```

```
blue :: Style
blue = Style "blue"

magenta :: Style
magenta = Style "magenta"

cyan :: Style
cyan = Style "cyan"

white :: Style
white = Style "white"

blackBright :: Style
blackBright = Style "blackBright"

gray :: Style
gray = Style "gray"

grey :: Style
grey = Style "grey"

redBright :: Style
redBright = Style "redBright"

greenBright :: Style
greenBright = Style "greenBright"

yellowBright :: Style
yellowBright = Style "yellowBright"

blueBright :: Style
blueBright = Style "blueBright"

magentaBright :: Style
magentaBright = Style "magentaBright"

cyanBright :: Style
cyanBright = Style "cyanBright"

whiteBright :: Style
whiteBright = Style "whiteBright"

bgBlack :: Style
bgBlack = Style "bgBlack"
```

```
bgRed :: Style
bgRed = Style "bgRed"

bgGreen :: Style
bgGreen = Style "bgGreen"

bgYellow :: Style
bgYellow = Style "bgYellow"

bgBlue :: Style
bgBlue = Style "bgBlue"

bgMagenta :: Style
bgMagenta = Style "bgMagenta"

bgCyan :: Style
bgCyan = Style "bgCyan"

bgWhite :: Style
bgWhite = Style "bgWhite"

bgBlackBright :: Style
bgBlackBright = Style "bgBlackBright"

bgGray :: Style
bgGray = Style "bgGray"

bgGrey :: Style
bgGrey = Style "bgGrey"

bgRedBright :: Style
bgRedBright = Style "bgRedBright"

bgGreenBright :: Style
bgGreenBright = Style "bgGreenBright"

bgYellowBright :: Style
bgYellowBright = Style "bgYellowBright"

bgBlueBright :: Style
bgBlueBright = Style "bgBlueBright"

bgMagentaBright :: Style
bgMagentaBright = Style "bgMagentaBright"
```

```
bgCyanBright :: Style
bgCyanBright = Style "bgCyanBright"

bgWhiteBright :: Style
bgWhiteBright = Style "bgWhiteBright"

reset :: Style  -- Resets the current color chain.
reset = Style "reset"

bold :: Style  -- Make text bold.
bold = Style "bold"

dim :: Style  -- Emitting only a small amount of light.
dim = Style "dim"

italic :: Style  -- Make text italic. (Not widely supported)
italic = Style "italic"

underline :: Style  -- Make text underline. (Not widely supported)
underline = Style "underline"

inverse :: Style -- Inverse background and foreground colors.
inverse = Style "inverse"

hidden :: Style  -- Prints the text, but makes it invisible.
hidden = Style "hidden"

strikethrough :: Style  -- Puts a horizontal line through the center of the
text. (Not widely supported)
strikethrough = Style "strikethrough"

visible :: Style -- Prints the text only when Chalk has a color level > 0. Can
be useful for things that are purely cosmetic.
visible = Style "visible"
```

Add this file to your project. Make sure to name it `ChalkStyles.purs` and put it under `src`.

Then change your test code to use it before continuing.

First thing to do is remove the following from `Ch27a.purs`:

```
type Style = String
```

Then import `ChalkStyles`:

```
import ChalkStyles (Style, bold, red)
```

And finally, change our `Strings` to `Styles` in our test code:

```
test :: Effect Unit
test = log $ chalk [ red, bold ] "Test" <> chalk [ red ] "Test"
```

Now run it again and it should produce the same output.

Okay, now change your program in the following ways:

```
import ChalkStyles (Style(..), bold, red) ❶

test :: Effect Unit
test = log $ chalk [ red, bold ] "Test"
  <> chalk [ red, Style "crash" ] "Test" ❷
```

❶ Include Data Constructors for `Style`.

❷ Add our own `Style` that's unsupported and will crash our program.

Run the test code and see what happens.

It crashes.

The problem is that we have access to the `Style` Data Constructor and can create any `Style` we like, even ones that `chalk` doesn't support.

So, how do we prevent this?

To fix this, we need to export out the Type and all of the valid styles, but NOT the `Style` Data Constructor:

```
module ChalkStyles
(
  Style ❶
, black
, red
, green
, yellow
, blue
, magenta
```

```
  , cyan
  , white
  , blackBright
  , gray
  , grey
  , redBright
  , greenBright
  , yellowBright
  , blueBright
  , magentaBright
  , cyanBright
  , whiteBright
  , bgBlack
  , bgRed
  , bgGreen
  , bgYellow
  , bgBlue
  , bgMagenta
  , bgCyan
  , bgWhite
  , bgBlackBright
  , bgGray
  , bgGrey
  , bgRedBright
  , bgGreenBright
  , bgYellowBright
  , bgBlueBright
  , bgMagentaBright
  , bgCyanBright
  , bgWhiteBright
  , reset
  , bold
  , dim
  , italic
  , underline
  , inverse
  , hidden
  , strikethrough
  , visible
  )

where
```

❶ Export the Type but not the Data Constructor. If this line was `Style(..)`, then the Data Constructors

would also be exported.

I've set up another GitHub Gist for this improvement at this link, ChalkStylesExports.purs.

Click the link and copy paste the contents of `ChalkStylesExports.purs` into your `ChalkStyle.purs`.

If you don't have access to GitHub, then the above code is all that changed.

We now have a Type-safe way to call `chalk`. It is IMPOSSIBLE to use `chalk` in a way that will crash our program.

Make the following changes to your code:

```
import ChalkStyles (Style, bold, red, dim, strikethrough) ❶ ❷
import Data.Array ((:)) ❸

test :: Effect Unit
test = do
  let colorful styles = chalk $ red : styles ❹
  log $ colorful [ bold, strikethrough ] "Test" ❺
    <> colorful [] "Test" <> colorful [ dim ] "Test" ❻
```

❶ Removed the `(..)` from `Style` since we no longer have access to its Data Constructor.

❷ Added `dim` and `strikethrough`.

❸ Import `(:)`.

❹ Factored out common styles.

❺ Added strikethrough to the first `"Test"`. According the `chalk` documentation, strikethrough isn't widely supported. So, depending on what system you're running on, you might not see this style.

❻ Added an additional `"Test"` that's dim.

Your output should look like:



Don't worry if the `strikethrough` doesn't show up. Not all terminal programs support this. `iTerm2` does support this.

27.2. Reversing JSON Keys in Javascript

Now that we know how to call Javascript from PureScript, we're going to return to our Echo Server project. This time we're going to reverse the keys in the JSON that the Echo Server sends back to us by using

Javascript. We're going start with what we did last in Chapter 25.

So create a new file called `Ch27b.purs` and then copy and paste all of the code from `Ch25b.purs` into it. Then save it.

Now, CLOSE `Ch25b.purs` and make sure you're working in `Ch27.purs`.

Edit the module name to be `Ch27b`.

Then delete all of the `Argonaut` imports:

```
import Data.Argonaut (class DecodeJson, JsonDecodeError(..)
                    , decodeJson, (.:))

import Data.Argonaut.Decode.Decoders (decodeJObject)
```

Then import the `Decode` Typeclass:

```
import Foreign.Generic.Class (class Encode , class Decode) ❶
```

❶ Added `class Decode` to the import list.

You should now get errors on all of the `DecodeJson` Instance derivations, for example:

```
derive newtype instance decodeJsonCentimeters :: DecodeJson Centimeters
```

This can be fixed by simply changing it to be:

```
derive newtype instance decodeCentimeters :: Decode Centimeters ❶ ❷
```

❶ Renamed `decodeJsonCentimeters` to `decodeCentimeters`.

❷ Renamed `DecodeJson` to `Decode`.

Now, edit all of the other similar `derive` declarations.

The next bunch of errors we'll attack are the handwritten `Argonaut` Decoders, for example.

```
instance decodeJsonPersonal :: DecodeJson Personal where
...
instance decodeJsonGrade :: DecodeJson Grade where
...
instance decodeJsonStudent :: DecodeJson Student where
...
instance decodeJsonTeachingStatus :: DecodeJson TeachingStatus where
...
instance decodeJsonTeacher :: DecodeJson Teacher where
...
```

So delete the implementations.

Now we don't have decoders for the above Types, since we just deleted them. Create `Decode` Instance for them using `genericDecode`.

Once you're done, return here.

Here are the changes for `Personal`:

```
import Foreign.Generic (defaultOptions, encodeJSON, genericEncode,
genericDecode) ❶

instance decodePersonal :: Decode Personal where
  decode = genericDecode defaultOptions
```

❶ Added `genericDecode` to the import list.

The others are similar to this change:

```
instance decodeGrade :: Decode Grade where
  decode = genericDecode defaultOptions

instance decodeStudent :: Decode Student where
  decode = genericDecode defaultOptions

instance decodeTeachingStatus :: Decode TeachingStatus where
  decode = genericDecode defaultOptions

instance decodeTeacher :: Decode Teacher where
  decode = genericDecode defaultOptions
```

Turning our attention to the `test` Function, we can see that we have a compiler error on `decodeJson`, i.e.

the `Argonaut` decode Function. Go ahead and change that to `decodeJSON`, i.e. the `Foreign` decode Function:

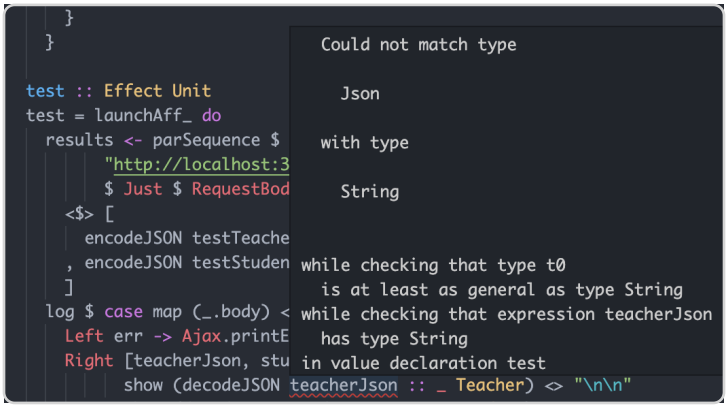
```
import Foreign.Generic (defaultOptions, encodeJSON, genericEncode
                      , genericDecode, decodeJSON) ❶

test :: Effect Unit
test = launchAff_ do
  results <- parSequence $ (\json -> Ajax.post ResponseFormat.json
        "http://localhost:3000/"
        $ Just $ RequestBody.String json)
    <$> [
      encodeJSON testTeacher
    , encodeJSON testStudent
    ]
  log $ case map (_.body) <$> sequence results of
    Left err -> Ajax.printError err
    Right [teacherJson, studentJson] ->
          -- COMPILER ERROR!!
          show (decodeJSON teacherJson :: _ Teacher) <> "\n\n" ❷
      <>  show (decodeJSON studentJson :: _ Student) ❷
    Right _ ->
      "The number of Ajax calls is different than what's being processed."
```

❶ Added `decodeJSON` to import list.

❷ Changed `deocodeJson` to `decodeJSON` in these 2 places.

Now, we're getting a compiler error:



Can you see what the problem is?

If it's not obvious, then it's time to ask that same old question, what are our Types.

Since the red underline is on `teacherJson` and it cannot match `Json` with `String`, it looks like we should

check the Type Signature for `decodeJSON`.

Remember, we changed it from `decodeJson`, which took a `Json`. So, we should check to see what `decodeJSON` takes:

```
decodeJSON :: ∀ a. Decode a => String -> F a
```

We can see that it takes a `String`, but we're giving it a `Json`. Can you see where we're getting a `Json` Value?

Look at `ResponseFormat.json`. This should be `ResponseFormat.string`.

Make that change to your code now.

Now, before we address the new compiler error, let's consider the return of `decodeJSON`. It's an `F a`, which is an `Except MultipleErrors a`.

When we were using `decodeJson`, it returned an `Either JsonDecodingError a` which we converted to a `String` with `show`.

So, if we can convert `Except MultipleErrors a` to `String`, we can leave the rest of the code alone.

We can use `runExcept` to get an `Either MultipleErrors a`. That's will get us a bit closer.

Then we need to make `MultipleErrors` a `String`. How can we do that?

The first thing I think of is to use `show`, which will only work if `MultipleErrors` has a `Show` Instance.

Go look it up on `Pursuit` and see if it does, then return here.

Since `MultipleErrors` is a `NonEmptyList ForeignError`, then we need to see if `ForeignError` has a `Show` Instance.

And when we click on `ForeignError` in `Pursuit`, and look at its Instance and we see that it does.

This means that `MultipleErrors` can be converted to a `String` using `show`.

That will require `lmap` since it's the left-hand side of an `Either`.

Go ahead and make these changes to convert `Except MultipleErrors a` to `Either String a`. Don't forget to use `lmap` and `show`. Do this before reading on.

Here are those changes:

```
import Data.Bifunctor (lmap) ❶
import Control.Monad.Except (runExcept)

test :: Effect Unit
test = launchAff_ do
  results <- parSequence $ (\json -> Ajax.post ResponseFormat.string
        "http://localhost:3000/"
        $ Just $ RequestBody.String json)
    <$> [
      encodeJSON testTeacher
    , encodeJSON testStudent
    ]
  log $ case map (_.body) <$> sequence results of
    Left err -> Ajax.printError err
    Right [teacherJson, studentJson] ->
          show (lmap show $ runExcept ❷
            $ decodeJSON teacherJson :: _ Teacher) <> "\n\n"
      <>  show (lmap show $ runExcept ❷
            $ decodeJSON studentJson :: _ Student)
    Right _ ->
      "The number of Ajax calls is different than what's being processed."
```

❶ Added these imports.

❷ Added `lmap show $ runExcept $` in these 2 places.

Now, I want factor out that common code that we just wrote. How could we do that?

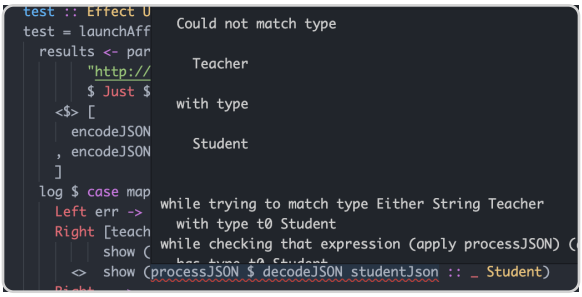
We could use a `let` or a `where` and make a local Function. Go ahead and change the code to factor out the code we just added into a Function called `processJSON` using a `where`. Write it Point-free while you're at it then return here.

Here's the test code with the common part factored out:

```
test :: Effect Unit
test = launchAff_ do
  results <- parSequence $ (\json -> Ajax.post ResponseFormat.string
        "http://localhost:3000/"
        $ Just $ RequestBody.String json)
    <$> [
      encodeJSON testTeacher
    , encodeJSON testStudent
    ]
  log $ case map (_.body) <$> sequence results of
    Left err -> Ajax.printError err
    Right [teacherJson, studentJson] ->
          show (processJSON ❶
            $ decodeJSON teacherJson :: _ Teacher) <> "\n\n"
          -- COMPILER ERROR!!
      <>  show (processJSON ❶
            $ decodeJSON studentJson :: _ Student)
    Right _ ->
      "The number of Ajax calls is different than what's being processed."
  where
    processJSON = lmap show <<< runExcept ❶
```

❶ Factored out the common code using Point-free notation.

Here's the unexpected compiler error:



This is a case of a type of error that we haven't encountered much.

Looking carefully at `processJSON`, figure out what Type it is. Spend a minute doing this before moving on.

It's hard to figure out, right?

We'll, the compiler figured out it's Type, but it's not telling us. So, we're going to cajole it to tell us.

Comment out the line with the compiler error. Then add a Type Hole as the whole Type Signature for `processJSON`.

Then see what you get. Do this before reading more.

Here's those changes:

```
test :: Effect Unit
test = launchAff_ do
  results <- parSequence $ (\json -> Ajax.post ResponseFormat.string
        "http://localhost:3000/"
        $ Just $ RequestBody.String json)
    <$> [
      encodeJSON testTeacher
    , encodeJSON testStudent
    ]
  log $ case map (_.body) <$> sequence results of
    Left err -> Ajax.printError err
    Right [teacherJson, studentJson] ->
          show (processJSON
            $ decodeJSON teacherJson :: _ Teacher) <> "\n\n"
      -- <>  show (processJSON
      --    $ decodeJSON studentJson :: _ Student) ❶
    Right _ ->
      "The number of Ajax calls is different than what's being processed."
  where
    processJSON :: ?x ❷
    processJSON = lmap show <<< runExcept
```

❶ Commented out the offending line of code.

❷ Type Hole for the Type Signature. Notice how I didn't use `_`. That's because using `_` means that the compiler will figure out what it is and it won't give us a compiler warning.

Here's the inferred Type Signature:

```
ExceptT (NonEmptyList ForeignError) Identity Teacher
  -> Either String Teacher
```

See the 2 places `Teacher` shows up? That's because the compiler has no way to know that we wanted `a's` in those positions.

It only knows how we've used it and, in this case, we've used it with a `Teacher`. When the compiler is inferring the Types for our first usage of `processJSON`, it doesn't know that we're going to use it with a totally different Type later in the code.

Because the inference process is localized like this, the compiler has ***overspecialized*** the Type.

So, here's a case where we must explicitly provide a more generalized Type Signature so we can use `processJSON` with Types other than just `Teacher`.

Go ahead and uncomment the offending code. Then replace the Type Hole with a generalized version of this inferred Type. Also, use `F` in your Type Signature.

Do this before moving on.

To generalize the Type Signature, we'll start with `F's` Type, i.e. the one we got from the Type Hole, and work toward what we have in the Type signature:

```
type F = Except MultipleErrors                          ❶
type Except e = ExceptT e Identity                      ❷
type MultipleErrors = NonEmptyList ForeignError         ❸
--------
ExceptT (NonEmptyList ForeignError) Identity Teacher    ❹
Except (NonEmptyList ForeignError) Teacher              ❺
Except MultipleErrors Teacher                           ❻
F Teacher                                               ❼
```

❶ Definition of `F`.

❷ Definition of `Except`.

❸ Definition of `MultipleErrors`.

❹ The first Parameter's inferred Type to `processJSON`.

❺ Substituted definition of `Except`.

❻ Substituted definition of `MultipleErrors`.

❼ Substituted definition of `F`.

So, the inferred Type is just `F Teacher` and if we generalize that it'll be `F a`:

```
import Foreign (F) ❶

test :: Effect Unit
test = launchAff_ do
  results <- parSequence $ (\json -> Ajax.post ResponseFormat.string
        "http://localhost:3000/"
        $ Just $ RequestBody.String json)
    <$> [
      encodeJSON testTeacher
    , encodeJSON testStudent
    ]
  log $ case map (_.body) <$> sequence results of
    Left err -> Ajax.printError err
    Right [teacherJson, studentJson] ->
          show (processJSON
            $ decodeJSON teacherJson :: _ Teacher) <> "\n\n"
      <>  show (processJSON
            $ decodeJSON studentJson :: _ Student) ❷
    Right _ ->
      "The number of Ajax calls is different than what's being processed."
  where
    processJSON :: ∀ a. F a -> Either String a ❸
    processJSON = lmap show <<< runExcept
```

❶ Import F.

❷ Uncommented.

❸ Replaced Type Hole with a generalized version of the inferred Type.

By the way, another way to quickly determine the input Type for `processJSON` is to consult the output Type for `decodeJSON` since we're calling `processJSON` with its output:

```
decodeJSON :: ∀ a. Decode a => String -> F a
```

We can see that it returns `F a`, which is the input to `processJSON`. This would have been the fastest way to figure this out. Next, time, maybe we'll try this approach first.

And now, for the first time, our new module compiles cleanly.

Now, we're ready to run the code.

First, open up another terminal window in VSCode or whatever terminal program you're using and run:

```
node src/echo-server.js
```

Then edit `Main.purs` to call `Ch27b` instead of `Ch27a`.

Then run the program and see what you get.

So, it looks like we need to add `foreign` back to our `spago.dhall`:

```
[error] Some of your project files import modules from packages that are not
in the direct dependencies of your project.
To fix this error add the following packages to the list of dependencies in
your config:
- foreign
```

Now when I fix this and rerun the code, I get:

```
(Left "(NonEmptyList (NonEmpty (ErrorAtProperty \"tag\" (TypeMismatch \"
String\" \"Undefined\")) Nil))")

(Left "(NonEmptyList (NonEmpty (ErrorAtProperty \"tag\" (TypeMismatch \"
String\" \"Undefined\")) Nil))")
```

First off, it doesn't work.

Second, do you notice the `\"` all over the place? That usually means we have an extra `show`. If you `show` a `String`, it'll add double quotes around that `String`.

Let's remove the `lmap show` from processJSON and see if it cleans up the embedded quotes:

```
import Foreign (F, MultipleErrors) ❶ ❷

test :: Effect Unit
test = launchAff_ do
  results <- parSequence $ (\json -> Ajax.post ResponseFormat.string
        "http://localhost:3000/"
        $ Just $ RequestBody.String json)
    <$> [
      encodeJSON testTeacher
    , encodeJSON testStudent
    ]
  log $ case map (_.body) <$> sequence results of
    Left err -> Ajax.printError err
    Right [teacherJson, studentJson] ->
          show (processJSON
            $ decodeJSON teacherJson :: _ Teacher) <> "\n\n"
      <>  show (processJSON
            $ decodeJSON studentJson :: _ Student)
    Right _ ->
      "The number of Ajax calls is different than what's being processed."
  where
    processJSON :: ∀ a. F a -> Either MultipleErrors a ❸
    processJSON = runExcept ❹
```

❶ Removed import of `Data.Bifunctor` since we're not using `lmap`.

❷ Added `MultipleErrors` to import list.

❸ Had to change the return Type since we're not mapping `show` over it anymore.

❹ Removed `lmap show`.

Technically, we no longer need `processJSON`. So, let's remove it. Yes, after all that work, we're removing it:

```
❶
test :: Effect Unit
test = launchAff_ do
  results <- parSequence $ (\json -> Ajax.post ResponseFormat.string
        "http://localhost:3000/"
        $ Just $ RequestBody.String json)
    <$> [
      encodeJSON testTeacher
    , encodeJSON testStudent
    ]
  log $ case map (_.body) <$> sequence results of
    Left err -> Ajax.printError err
    Right [teacherJson, studentJson] ->
          show (runExcept $ decodeJSON teacherJson :: _ Teacher) <> "\n\n"
      <>  show (runExcept $ decodeJSON studentJson :: _ Student)
    Right _ ->
      "The number of Ajax calls is different than what's being processed."
```

❶ Removed `import Foreign`.

If I had a dollar for every line of code I've written, debugged and eventually thrown out, I'd be rich.

Make these changes to your code and run it to see what it displays now.

This is what I get:

```
[warn] None of your project files import modules from some projects that are
in the direct dependencies of your project.
These dependencies are unused. To fix this warning, remove the following
packages from the list of dependencies in your config:
- foreign
```

Well, it turns out that `foreign` is once again no longer a direct dependency of our program. So remove it. Sorry.

Now, rerunning, this is what I get:

```
(Left (NonEmptyList (NonEmpty (ErrorAtProperty "tag" (TypeMismatch "String"
"Undefined")) Nil)))

(Left (NonEmptyList (NonEmpty (ErrorAtProperty "tag" (TypeMismatch "String"
"Undefined")) Nil)))
```

Okay, that's better. Well, it's prints a better message when it doesn't work.

But why doesn't it work?

The message should tell us if we take the time to examine it. It seems as if the decoder is expeciting a property (key) of `tag` of Type `String` but it's `Undefined`.

So that tells us that the key doesn't exist.

But why not?

If you're thinking that we haven't reversed the keys yet, you're right.

We need to do that. Think to yourself for a minute about how you'd accomplish this in Javascript. What approach would you take?

I know what I'd do. I'd plagerize the code from the Echo Server. It's taking a JSON Object and recursively reversing the keys to all of the Objects it encounters as it traverses its structure, which is exactly what we need.

So, create a new file called `Ch27b.js`. Copy and paste the code from the Echo Server that does the reversing of the JSON. Then return here.

Here's the code that should be copied to `Ch27b.js`:

```
const reverseArray = a => {
  var na = [];
  for (var i = 0; i < a.length; ++i) {
    const value = a[i];
    const nValue = value.constructor.name == 'Object'
      ? reverseKeys(value) : value;
    na.push(nValue);
  }
  return na;
}
const reverseKeys = o => {
  const no = {};
  for (const [key, value] of Object.entries(o)) {
    const nValue = value.constructor.name == 'Object'
      ? reverseKeys(value)
      : (value.constructor.name == 'Array'
        ? reverseArray(value) : value);
    no[key.split("").reverse().join("")] = nValue;
  }
  return no;
};
```

If this works for the Echo Server, it should work for us. The only thing we need to do is make it callable from PureScript.

Start by exporting `reverseKeys` in `Ch27b.js`. Look at `Ch27a.js` to see how we did that before if you need help.

Make this change, then come back here.

Here's the added code:

```
exports._reverseKeys = reverseKeys;
```

Now, we're ready to add the `foreign import` to our code like we've done before. Go ahead and give that a try and return here when you get stuck or finish.

Okay, so maybe we've jumped the gun here. We never considered the Type that `reverseKeys` takes. Look at that Function and see if you can tell what Type it is.

It's a Javascript Object. So, we need to pass a Javascript Object. But how do we do this in PureScript?

We have a `String` that we're passing to `decodeJSON`. So, we could just change our Javascript to do:

```
exports._reverseKeys s = reverseKeys(JSON.parse(s)); // NOT IDEAL!!
```

But this will throw an exception on a bad `String`. We don't want our beautiful PureScript code brought down by crappy Javascript code.

So, let's find a way to convert a `String` to a `Foreign`. Search `Pursuit` for `String -> Foreign`, since `Foreign` is the Type for Javascript data, and see what you find.

I don't know about you, but I get a big fat page full of nothing. Okay, so we need to think about this. What are we missing?

The `String` could be invalid JSON. So, whatever Type Signature we search for should take that into account. Let's look for `String -> Maybe Foreign`.

Well, I find `readJSON_` but it's in a different package and besides it doesn't return a `Foreign`. So that won't work.

Try `String -> Either e Foreign`.

Another big nothing. Okay, I don't know about you, but I'm sick of trying to guess the Type Signature. Let's try to guess the Function name.

What's a good name? Well, in Javascript we use `JSON.parse`, so look for `parse`.

What did you find?

I found a ton of `parse` Functions. So, I went to the bottom of the page and keep clicking the `Load more results` button until the button disappears, which means that there are not more search results.

Then I searched the page for `Foreign` and found:

```
parse :: String -> Either Error Foreign
```

This is in a module I've never seen before called `FFI.Foreign.JSON` and when I click on it I find:



Okay, so looking at the version, `0.0.2` doesn't make me feel very confident that I'm going to want to use this module.

But when I look at the page I see a Function called `stringify`. I wonder what we'll find if we search for that since the name is pretty unique.

Do it and see if you can find it in a library we trust.

Well, the only one I found in a trustworthy library is in `Argonaut`. But we're not using `Argonaut` in this solution.

So, let's go back to looking for `parse` again. See what you find when you search for `JSONParse`.

Okay, I got nothing. Try `parseJSON` and see what comes up.

FINALLY!!

```
parseJSON :: String -> F Foreign
```

And this Function is in `Foreign.JSON`, a module from a package we trust, i.e. `purescript-foreign-generic`.

Sometimes finding what you're looking for is like pulling teeth. You have keep at it trying anything you can think of until you finally find what you're looking for.

So, we'll want to use `parseJSON` in our PureScript code when we call `_reverseKeys`.

Now, we've knocked down that Technological Hurdle, we're ready to code the `foreign import` line.

Go ahead and use what we did in `Ch27a.purs` to help you write the PureScript Function that will call the Javascript Function, `_reverseKeys`.

Do this until the compiler underlines your whole file in red. This can happen before you're finished.

Here's how far I got:

```
import Foreign (Foreign) ❶
import Data.Function.Uncurried (Fn1, runFn1) ❷

foreign import _reverseKeys :: Fn1 Foreign String ❸
```

❶ Import `Foreign`.

❷ Import helper Functions for calling Uncurried Javascript Functions. Realize that a Function of 1 Parameter is, by definition, Curried.

❸ Define the Foreign Function Type Signature.

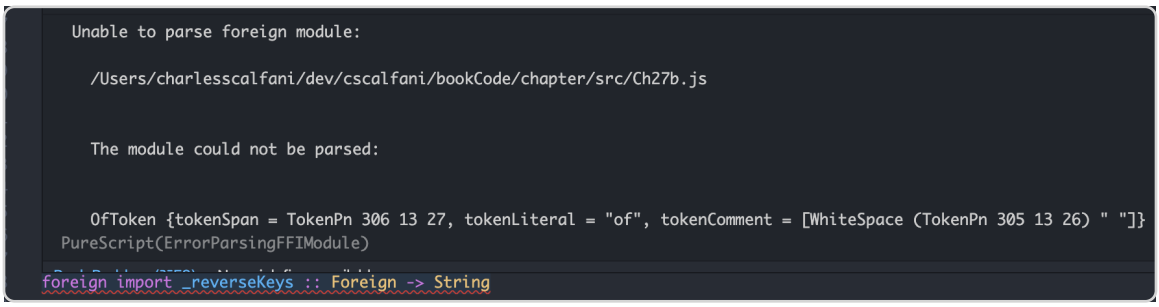
It's at this point, I got red underlines on every line.

But before we dive into the error, `_reverseKeys` only takes 1 Parameter. Let's delete the import on `Data.Function.Uncurried` and just make a normal Type Signature:

```
foreign import _reverseKeys :: Foreign -> String
```

Don't forget to remove `Data.Funtion.Uncurried`.

Now, for that error. Here's the compiler error:



Now, when you get these errors, it's means that the compiler cannot find your Foreign Function or that it's having trouble parsing your Javascript code.

Typically, this is because you're using more modern elements of Javascript. The error message has some parsing information that maybe we can use to help us:

```
OfToken {tokenSpan = TokenPn 306 13 27, tokenLiteral = "of", tokenComment =
[WhiteSpace (TokenPn 305 13 26) " "]}
```

The `tokenLiteral` is `"of"`. So let's search our code for that.

What did you find?

I found it on one line:

```
for (const [key, value] of Object.entries(o))
```

We're going to have to rewrite this. But what if you don't understand what this does?

The syntax is fairly new and the `Object.entries` is probably foreign to you. I'm no expert on this.

So, the approach I'm going to take is to understand what we get from `Object.entries` in the most expedient way possible.

Run the following commands in the Terminal:

```
$ node
> Object.entries({a: 1, b: 2}) ❶
[ [ 'a', 1 ], [ 'b', 2 ] ]
^D ❷
```

❶ Pass a simple example Object.

❷ Hit Ctrl-D to exit Node.

I've passed a dummy Javascript Object to `Object.entries` to see what it would return and we can see that it's an Array of pairs. Each pair is an Array of size 2 with the key in the first position and the value in the second.

Now, we know enough to use a traditional `for-loop` in place of the offending line.

If you know Javascript, give it go before reading on and see if the compiler error goes away. You'll probably have to manually save your PureScript file to trigger a recompile. If you don't know Javascript, continue reading now.

The first thing we should to do is comment out the offending `for`. We might want to use it for reference:

```
// for (const [key, value] of Object.entries(o)) {
// ???
```

Next, we'll call `Object.entries` to get the entries:

```
// for (const [key, value] of Object.entries(o)) {
const entries = Object.entries(o);
// ???
```

Next, we'll write a traditional, old school `for-loop`:

```
// for (const [key, value] of Object.entries(o)) {
  const entries = Object.entries(o);
  for (var i = 0; i < entries.length; ++i) {
    // ???
    // the rest of the old for-loop
  }
```

And finally, we need to get `key` from the first position of the current entry and `value` from the second:

```
// for (const [key, value] of Object.entries(o)) {
  const entries = Object.entries(o);
  for (var i = 0; i < entries.length; ++i) {
    const key = entries[i][0]; ❶ ❸
    const value = entries[i][1]; ❷ ❸
    // the rest of the old for-loop
  }
```

❶ Get the key from the first position of the pair.

❷ Get the value from the second position of the pair.

❸ Do not forget the `const`. The first time I coded this, I left them out which makes them global to this module. Since this Function is recursive, the recursive calls would step on these global variables. I hate Javascript.

When I return to my PureScript code and save it, the compiler error disappears. That tells us that our Javascript code is parsable by the PureScript Compiler.

Now, let's return to `test`:

```
test :: Effect Unit
test = launchAff_ do
  results <- parSequence $ (\json -> Ajax.post ResponseFormat.string
        "http://localhost:3000/"
        $ Just $ RequestBody.String json)
    <$> [
      encodeJSON testTeacher
    , encodeJSON testStudent
    ]
  log $ case map (_.body) <$> sequence results of
    Left err -> Ajax.printError err
    Right [teacherJson, studentJson] ->
          show (runExcept $ decodeJSON teacherJson :: _ Teacher) <> "\n\n"
      <>  show (runExcept $ decodeJSON studentJson :: _ Student)
    Right _ ->
      "The number of Ajax calls is different than what's being processed."
```

We need to call `parseJSON` to convert our `String` to a `Foreign` so that we call `_reverse`. And we're going to put the same code on 2 lines.

And looking at `parseJSON`, we can see that it's Type Signature is:

```
parseJSON :: String -> F Foreign
```

This is running in the `F` Monad. But then again, so is `decodeJSON`. So now, we're going to need to bind them.

I kind of wish I hadn't decided to throw out `processJSON`. Oh, well. We wrote in once. We can write it again.

Here's a starting point of what I'm thinking of:

```
test :: Effect Unit
test = launchAff_ do
  results <- parSequence $ (\json -> Ajax.post ResponseFormat.string
        "http://localhost:3000/"
        $ Just $ RequestBody.String json)
    <$> [
      encodeJSON testTeacher
    , encodeJSON testStudent
    ]
  log $ case map (_.body) <$> sequence results of
    Left err -> Ajax.printError err
    Right [teacherJson, studentJson] ->
          show (runExcept $ decodeJSON teacherJson :: _ Teacher) <> "\n\n"
      <>  show (runExcept $ decodeJSON studentJson :: _ Student)
    Right _ ->
      "The number of Ajax calls is different than what's being processed."
  where
    processJSON json = ???
```

I've only added the `where` section. We want to wind up with `runExcept`, `decodeJSON`, `parseJSON` and `_reverseKeys` in `processJSON`. We want to contain all this complexity inside of `processJSON`.

Go ahead and give it a shot remembering that `decodeJSON` and `parseJSON` run in the `F` Monad. When you finish or hit a brick wall, continue reading.

I know `runExcept` will take an `F a`, since it expands to `Except`. So, I'm going to start with that:

```
test :: Effect Unit
test = launchAff_ do
  results <- parSequence $ (\json -> Ajax.post ResponseFormat.string
        "http://localhost:3000/"
        $ Just $ RequestBody.String json)
    <$> [
      encodeJSON testTeacher
    , encodeJSON testStudent
    ]
  log $ case map (_.body) <$> sequence results of
    Left err -> Ajax.printError err
    Right [teacherJson, studentJson] ->
          show (runExcept $ decodeJSON teacherJson :: _ Teacher) <> "\n\n"
      <>  show (runExcept $ decodeJSON studentJson :: _ Student)
    Right _ ->
      "The number of Ajax calls is different than what's being processed."
  where
    processJSON json = runExcept do ❶
      ???
```

❶ `runExcept` gets passed the result of the `do` block meaning that the `do` must be an `Except` or an `F`.

Now, I need to first `parseJSON` before I can reverse the keys:

```
import Foreign.JSON (parseJSON) ❶

test :: Effect Unit
test = launchAff_ do
  results <- parSequence $ (\json -> Ajax.post ResponseFormat.string
        "http://localhost:3000/"
        $ Just $ RequestBody.String json)
    <$> [
      encodeJSON testTeacher
    , encodeJSON testStudent
    ]
  log $ case map (_.body) <$> sequence results of
    Left err -> Ajax.printError err
    Right [teacherJson, studentJson] ->
          show (runExcept $ decodeJSON teacherJson :: _ Teacher) <> "\n\n"
      <>  show (runExcept $ decodeJSON studentJson :: _ Student)
    Right _ ->
      "The number of Ajax calls is different than what's being processed."
  where
    processJSON json = runExcept do
      o <- parseJSON json ❷ ❸
      ???
```

❶ Add this import.

❷ `o` represents the JSON Object that we're going to give to `_reverseKeys`.

❸ If this step fails because we give it a bogus JSON `String`, we won't crash, but instead it'll be handled thanks to the underlying `ExceptT`. Contrast that to our earlier discussion about using `JSON.parse` in the Javascript code.

If you got stuck, stop reading and see if you can finish this now.

Next, I want to reverse the keys and give those results to `decodeJson`:

```
test :: Effect Unit
test = launchAff_ do
  results <- parSequence $ (\json -> Ajax.post ResponseFormat.string
        "http://localhost:3000/"
        $ Just $ RequestBody.String json)
    <$> [
      encodeJSON testTeacher
    , encodeJSON testStudent
    ]
  log $ case map (_.body) <$> sequence results of
    Left err -> Ajax.printError err
    Right [teacherJson, studentJson] ->
          show (runExcept $ decodeJSON teacherJson :: _ Teacher) <> "\n\n"
      <>  show (runExcept $ decodeJSON studentJson :: _ Student)
    Right _ ->
      "The number of Ajax calls is different than what's being processed."
  where
    processJSON json = runExcept do
      o <- parseJSON json
      decodeJSON $ _reverseKeys o ❶
```

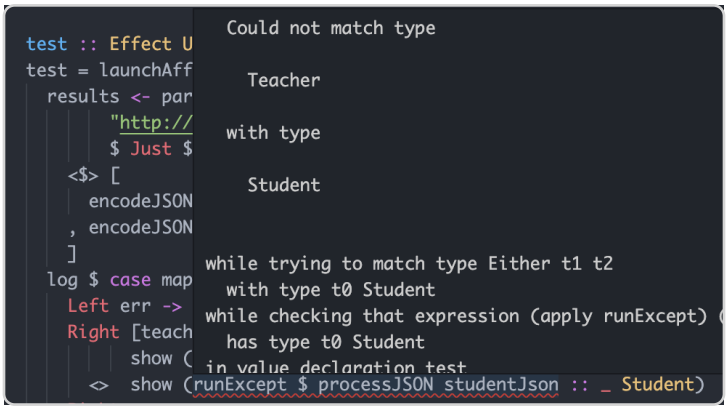
❶ `decodeJSON` returns an `F a` so there's no need for `pure` here.

And now, we can use `processJSON`:

```
test :: Effect Unit
test = launchAff_ do
  results <- parSequence $ (\json -> Ajax.post ResponseFormat.string
        "http://localhost:3000/"
        $ Just $ RequestBody.String json)
    <$> [
      encodeJSON testTeacher
    , encodeJSON testStudent
    ]
  log $ case map (_.body) <$> sequence results of
    Left err -> Ajax.printError err
    Right [teacherJson, studentJson] ->
          show (processJSON teacherJson :: _ Teacher) <> "\n\n" ❶
          -- COMPILER ERROR!!
      <>  show (processJSON studentJson :: _ Student) ❶
    Right _ ->
      "The number of Ajax calls is different than what's being processed."
  where
    processJSON json = runExcept do
      o <- parseJSON json
      decodeJSON $ _reverseKeys o
```

❶ Use `processJSON` in these 2 places.

Let's look at that pesky compiler error:



We've seen this error before. The compiler is overspecializing again.

Make sure your test code matches the above code.

Then comment out the offending line of code.

And finally, add a Type Signature to `processJSON` with a Type Hole like we did last time to see what Type the compiler infers.

Then take that Type and generalize it. Once that compiles, then you can uncomment out the offending line of code and, hopefully, everything will compile.

Give that a shot and return here when you're either done or stumped.

Okay, so here's all of those changes right up to the Type Hole:

```
test :: Effect Unit
test = launchAff_ do
  results <- parSequence $ (\json -> Ajax.post ResponseFormat.string
        "http://localhost:3000/"
        $ Just $ RequestBody.String json)
    <$> [
      encodeJSON testTeacher
    , encodeJSON testStudent
    ]
  log $ case map (_.body) <$> sequence results of
    Left err -> Ajax.printError err
    Right [teacherJson, studentJson] ->
          show (processJSON teacherJson :: _ Teacher) <> "\n\n"
      -- <>  show (processJSON studentJson :: _ Student) ❶
    Right _ ->
      "The number of Ajax calls is different than what's being processed."
  where
    processJSON :: ?x ❷
    processJSON json = runExcept do
      o <- parseJSON json
      decodeJSON $ _reverseKeys o
```

❶ Commented out the offending line.

❷ Added Type Signature with a Type Hole.

And here's the Type it inferred:

```
String -> Either (NonEmptyList ForeignError) Teacher
```

But since:

```
type MultipleErrors = NonEmptyList ForeignError
```

we can reduce it to:

```
String -> Either MultipleErrors Teacher
```

And now, we can generalize this as:

```
∀ a. String -> Either MultipleErrors a
```

Adding that yields:

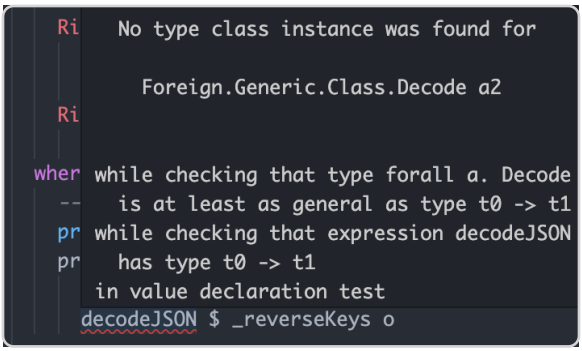
```
import Foreign (Foreign, MultipleErrors) ❶

test :: Effect Unit
test = launchAff_ do
  results <- parSequence $ (\json -> Ajax.post ResponseFormat.string
        "http://localhost:3000/"
        $ Just $ RequestBody.String json)
    <$> [
      encodeJSON testTeacher
    , encodeJSON testStudent
    ]
  log $ case map (_.body) <$> sequence results of
    Left err -> Ajax.printError err
    Right [teacherJson, studentJson] ->
          show (processJSON teacherJson :: _ Teacher) <> "\n\n"
      -- <>  show (processJSON studentJson :: _ Student)
    Right _ ->
      "The number of Ajax calls is different than what's being processed."
  where
    processJSON :: ∀ a. String -> Either MultipleErrors a ❷
    processJSON json = runExcept do
      o <- parseJSON json
      decodeJSON $ _reverseKeys o -- COMPILER ERROR!!
```

❶ Added `MultipleErrors` to import list.

❷ Added our generalized Type Signature.

And now, another compiler error:



Okay, so unlike our last `processJSON`, we're calling `decodeJSON` here, so we'll need to constrain `a` accordingly:

```
test :: Effect Unit
test = launchAff_ do
  results <- parSequence $ (\json -> Ajax.post ResponseFormat.string
        "http://localhost:3000/"
        $ Just $ RequestBody.String json)
    <$> [
      encodeJSON testTeacher
    , encodeJSON testStudent
    ]
  log $ case map (_.body) <$> sequence results of
    Left err -> Ajax.printError err
    Right [teacherJson, studentJson] ->
          show (processJSON teacherJson :: _ Teacher) <> "\n\n"
      <>  show (processJSON studentJson :: _ Student) ❶
    Right _ ->
      "The number of Ajax calls is different than what's being processed."
  where
    processJSON :: ∀ a. Decode a => String -> Either MultipleErrors a ❷
    processJSON json = runExcept do
      o <- parseJSON json
      decodeJSON $ _reverseKeys o
```

❶ Uncommented.

❷ Constrain `a` to have a `Decode` Instance.

Make these changes to your code (don't forget to uncomment) and run it to see if it works.

And now we have to re-add `foreign` back to our `spago.dhall` file:

```
[error] Some of your project files import modules from packages that are not
in the direct dependencies of your project.
To fix this error add the following packages to the list of dependencies in
your config:
- foreign
```

Once this is fixed, we run it to see if it works. And it does NOT:

```
(Left (NonEmptyList (NonEmpty (ForeignError "Unexpected token o in JSON at
position 1") Nil)))

(Left (NonEmptyList (NonEmpty (ForeignError "Unexpected token o in JSON at
position 1") Nil)))
```

But why not? Where do we even begin to look?

Well, we can see that the decoding is failing. So, let's see what we're giving `decodeJSON` using `spy`:

```
import Debug (spy)

  -- REDACTED test code above the where section
  where
    processJSON :: ∀ a. Decode a => String -> Either MultipleErrors a
    processJSON json = runExcept do
      o <- parseJSON json
      decodeJSON $ spy "json" $ _reverseKeys o ❶
```

❶ Added `spy` between `decodeJSON` and `_reverseKeys`. Talk about a man-in-the-middle attack.

Make this change and add `debug` to your `spago.dhall` (if it's not there).

Now run it and see what prints out in the console. From that, you have enough information to figure this problem out.

It may not be obvious at first. It took me a few minutes, but when it hit me, I felt pretty stupid that I coded this even though it's an easy mistake to make.

Give it try and see if you can see my mistake.

Here's a partial listing of the output:

```
json: { contents:
   { teacher:
      { contents:
         { status: { tag: 'NonTenured' },
           personal:
            { contents: { weight: 63.5, height: 162.56, age: 31 },
              tag: 'Personal' },
           numberOfStudents: 23,
           grades:
            [ { tag: 'Preschool' },
              { tag: 'Kindergarten' },
              { contents: 1, tag: 'Grade' } ] },
        tag: 'Teacher' },
     personal:
      { contents: { weight: 17.9, height: 107.9, age: 5 },
        tag: 'Personal' },
     grade: { contents: 1, tag: 'Grade' },
     gpa: 3.2 },
  tag: 'Student' }
```

Remember that this is what we're giving `decodeJSON`. Ask yourself if this looks like what `decodeJSON` wants:

```
decodeJSON :: ∀ a. Decode a => String -> F a
```

Can you see it now?

Okay, if you realized that this is NOT a `String`, then kudos to you. `_reverseKeys` works with a Javascript Object. If you remember, that's why we're calling `parseJSON`, i.e. to convert the `String` to a `Foreign`.

The problem is that we told the compiler that `_reverseKeys` would return a `String`:

```
foreign import _reverseKeys :: Foreign -> String
```

But it doesn't. I lied to the compiler and now I'm paying the price.

So, we need to `stringify` the Javascript Object before we return it from `_reverseKeys`. Note that we CANNOT make `reverseKeys` do this since it's recursive.

But we can make `_reverseKeys` do this:

```
exports._reverseKeys = o => JSON.stringify(reverseKeys(o));
```

Make this change and run it leaving in the `spy` code. See what you get.

I get this:

```
json:
'{"contents":{"status":{"tag":"NonTenured"},"personal":{"contents":{"weight":6
3.5,"height":162.56,"age":31},"tag":"Personal"},"numberOfStudents":23,"grades"
:[{"tag":"Preschool"},{"tag":"Kindergarten"},{"contents":1,"tag":"Grade"}]},"t
ag":"Teacher"}'
json:
'{"contents":{"teacher":{"contents":{"status":{"tag":"NonTenured"},"personal":
{"contents":{"weight":63.5,"height":162.56,"age":31},"tag":"Personal"},"number
OfStudents":23,"grades":[{"tag":"Preschool"},{"tag":"Kindergarten"},{"contents
":1,"tag":"Grade"}]},"tag":"Teacher"},"personal":{"contents":{"weight":17.9,"h
eight":107.9,"age":5},"tag":"Personal"},"grade":{"contents":1,"tag":"Grade"},"
gpa":3.2},"tag":"Student"}'
(Right (Teacher { grades: [Preschool,Kindergarten,(Grade 1)],
numberOfStudents: 23, personal: (Personal { age: (Years 31), height:
(Centimeters 162.56), weight: (Kilograms 63.5) }), status: NonTenured }))

(Right (Student { gpa: (GPA 3.2), grade: (Grade 1), personal: (Personal { age:
(Years 5), height: (Centimeters 107.9), weight: (Kilograms 17.9) }), teacher:
(Teacher { grades: [Preschool,Kindergarten,(Grade 1)], numberOfStudents: 23,
personal: (Personal { age: (Years 31), height: (Centimeters 162.56), weight:
(Kilograms 63.5) }), status: NonTenured }) }))
```

Notice that our `spy` displays a `String` now and our code works!!

Remove the `Debug` and the `spy`call before moving on.

Now, just to make sure we didn't break anything, run your program again.

Chapter 28. Writing a Backend using HTTPure

We're going to create a new project where we'll put the code for our backend server and, in a subsequent Chapter, our front-end program.

Go to parent directory that contains the project you've been working on.

If your code is kept in a folder called `/dev`, and you followed the directions exactly in Chapter 4 then you'd run:

```
cd /dev/purescript
```

If this is not where you put your project code, change to the appropriate directory now.

Next, create the `fullstack` project directory with a `server` directory under it:

```
mkdir -p fullstack/server # or some other name you like better
cd fullstack/server ❶
```

❶ We're going to make a directory called `client` when we write the front end.

Before we go further ***make sure that you're in the `server` directory***.

To initialize your project so that we can install `npm` packages and use `npx` to run the compiler and build tool from inside your project, run:

```
npm init -y
```

This will create a file call `package.json`. This will keep track of any `npm` packages we install.

To locally install the PureScript compiler, `purescript` and the build tool, `spago`, run:

```
npm install --save-dev spago@0.20.0 purescript@0.14.1
```

Notice the `--save-dev` option. This will tell `npm` that these packages are only for development.

To verify that we have installed the compiler locally, we'll use `npx` (NOTICE the `x`):

```
npx purs --version # should print out 0.14.1
```

`npx` looks in our local `node_modules` directory for `purs`, the name of the PureScript compiler, and then

runs it passing all other command line Parameters to it. The parameter `--version` tells `purs` to print out it's version.

To verify that we have installed the build tool locally, run:

```
npx spago version # should print out 0.20.0
```

Now, prepare the project for version control using `git`, run:

```
git init # should print `Initialized empty Git repository in <dir>`
```

The final step is to initialize this project for building a PureScript program using `spago`:

```
npx spago init
```

Build the starter app with:

```
npx spago build
```

Now, if you're using VSCode, you should be able type the following command to open up this directory in the editor:

```
code .
```

If this doesn't work or you're using a different editor, then simply open this directory manually from your editor.

28.1. A Brief Primer on HTTPure

`HTTPure` is an HTTP Server Framework that has the following features:

- Well tested
- Well documented
- Built to take advantage of PureScript language features for flexible and extensible routing
- Pure (no `set`, `get`, `use`, etc.)

28.1.1. Creating an HTTP Server

It's possible to create a insecure server, i.e. `HTTP` using `HTTPure.serve`. A secure server, i.e. `HTTPS`, can be

created with `HTTPure.serveSecure`.

Many times, servers are placed behind Proxies or Load Balancers, e.g. `nginx` or `haproxy`. When using this configuration, it makes sense to manage SSL Certificates at the this layer in the Enterprise and, in these sorts of scenarios, there isn't a need to create secure servers.

If you do need to manage the certificates at the server layer, you can provide certificate files to `HTTPure.serveSecure`.

We're not going to get bogged down in the hassle of creating self-signed certificates for testing. Instead, we'll just create an insecure server since we're just learning here. But it's always important to understand these issues before embarking on a new server project.

Let's create a very simple server. To start, we'll need to add the `purescript-httpure` package to `spago.dhall`:

```
{-
Welcome to a Spago project!
You can edit this file as you like.
-}
{ name = "my-project"
, dependencies =  ❶
  [ "console"
  , "effect"
  , "httpure" ❷
  , "psci-support"
  ]
, packages = ./packages.dhall
, sources = [ "src/**/*.purs", "test/**/*.purs" ]
}
```

❶ I reformatted the `dependencies` so I could manage them.

❷ Added `httpure`. Notice that we don't put the prefix `purescript-` from the package name. Also, notice that `httpure` is spelled with only 1 `p`.

Then run the following command to build our project:

```
npx spago build
```

We can ignore the following warning:

```
[warn] None of your project files import modules from some projects that are
in the direct dependencies of your project.
These dependencies are unused. To fix this warning, remove the following
packages from the list of dependencies in your config:
- httpure
```

Now edit your `Main.purs` to be the following:

```
module Main where

import Prelude

import Effect.Class.Console (log)
import HTTPure as HTTPure
import HTTPure.Request (Request)
import HTTPure.Response (ResponseM)
import HTTPure.Server (ServerM)

router :: Request -> ResponseM
router _ = HTTPure.ok "This works" ❶ ❷

port :: Int
port = 3000

main :: ServerM
main = HTTPure.serve port router
  $ log $ "Server up running on port: " <> show port ❸
```

❶ The `router` takes any request and determines the appropriate response. Our simple one here just replies with a constant text message.

❷ `HTTPure.ok` send a response with an HTTP Statue Code of `200`.

❸ `HTTPure.serve` starts an insecure server on the specified `port`. Once the server is up, it'll run the third Parameter, which is an `Effect Unit`. This could've been a `do` block, but we've only have 1 line of code.

Now run this server using:

```
npx spago run
```

Then open up a browser and put in the Address Bar, `localhost:3000`, then hit return. It should display `This works`.

Now, if you look closely at `main's` Type Signature, you may be wondering how we can have a `main` that

returns something other than an `Effect`. Well, it turns out that you cannot.

And now, you're probably wondering what `ServerM` is:

```
type ServerM = Effect (Effect Unit -> Effect Unit)
```

It's a Type Alias for `Effect`, which means that our `main` is returning an `Effect`. But what is the Function it returns?

Well, it's a Callback to shutdown the server. This shutdown Function can never be called since we're returning it from `main`.

Hit `Ctrl-C` in the Terminal window to kill the server.

Change your `main` to take advantage of this Callback as such:

```
import Data.Time.Duration (Milliseconds(..)) ❶
import Effect (Effect)
import Effect.Aff (launchAff_)
import Effect.Class (liftEffect)

main :: Effect Unit ❷
main = do
  shutdown <- HTTPure.serve port router
    $ log $ "Server up running on port: " <> show port ❸
  launchAff_ do ❹
    delay (Milliseconds 2000.0) ❺
    log "Shutting down server..."
    liftEffect $ shutdown $ log "Server shutdown" ❻
```

❶ Add these imports.

❷ Main now returns `Effect Unit`.

❸ Create the server and capture the `shutdow` Function.

❹ Start a separate process that runs under `Aff` to eventually shutdown the server.

❺ Delay 2 seconds.

❻ Shutdown the server.

The shutdown Function runs in `Effect` as we can see from its Type Signature, `Effect Unit -> Effect Unit`. But, since we're calling `shutdown` in `Aff`, we have to use `liftEffect`.

Then remove the following imports:

```
import HTTPure.Server (ServerM) ❶
```

❶ Delete this line.

`main` now will run your server for 2 seconds and then shut it down.

Hit `Contrl-C` to kill the running server and then run this code with `npx spago run` and see what happens.

I get:

```
To fix this error add the following packages to the list of dependencies in
your config:
- aff
- datetime
```

That means we need to add these dependencies to our `spago.dhall`:

```
{-
Welcome to a Spago project!
You can edit this file as you like.
-}
{ name = "my-project"
, dependencies =
  [ "aff" ❶
  , "console"
  , "datetime" ❶
  , "effect"
  , "httpure"
  , "prelude"
  , "psci-support"
  ]
, packages = ./packages.dhall
, sources = [ "src/**/*.purs", "test/**/*.purs" ]
}
```

❶ Added these dependencies.

28.2. Routers

Routers are Functions that take the `Request` and respond based on one or more its fields. Here's `HTTPure's` definition for `Request`:

```
type Request =
  { method :: Method.Method
  , path :: Path.Path
  , query :: Query.Query
  , headers :: Headers.Headers
  , body :: String
  , httpVersion :: Version.Version
  }
```

Routers have the following Type Signature:

```
router :: Request -> ResponseM
```

Since `router` is just a normal Function, we can use Guards or Pattern Matching to handle different requests.

Replace your `router` with the following:

```
router :: Request -> ResponseM
router { path: [ "hello" ] }   = HTTPure.ok "hello" ❶
router { path: [ "goodbye" ] } = HTTPure.ok "goodbye" ❷
router _                       = HTTPure.notFound ❸
```

❶ This Pattern Matches the `path` of the URL for `hello`, e.g. `localhost:3000/hello`.

❷ This also Pattern Matches the `path`, but for `goodbye`.

❸ All other cases are a Not Found error, i.e. `404`.

Next, the following change your `main`:

```
import HTTPure.Server (ServerM) ❶

main :: ServerM ❷
main = HTTPure.serve port router
  $ log $ "Server up running on port: " <> show port
```

❶ Added this import.

❷ Replace `main` with this.

Then remove the following imports:

```
import Data.Time.Duration (Milliseconds(..)) ❶
import Effect.Aff (launchAff_, delay)
import Effect (Effect)
import Effect.Class (liftEffect)
```

❶ Delete these lines.

Then run your server. You should get the following warning:

```
[warn] None of your project files import modules from some projects that are
in the direct dependencies of your project.
These dependencies are unused. To fix this warning, remove the following
packages from the list of dependencies in your config:
- aff
- datetime
```

We just added these and now we don't need them anymore. Let's remove just `datetime`. We're going to need `aff` eventually, so let's leave it.

Now, use your Browser and go to `localhost:3000/hello` and then go to `localhost:3000/goodbye`.

And then try `localhost:3000/anything` and you should get a `404` error, i.e. `HTTPure.notFound`.

Now, let's create a slightly more complex `router`:

```
import HTTPure.Method (Method(..)) ❶

router :: Request -> ResponseM
router { path: [ "hello" ], method } ❷
  | method == Get                = HTTPure.methodNotAllowed ❸
  | method == Post               = HTTPure.ok "hello"
router { path: [ "goodbye" ] }   = HTTPure.ok "goodbye"
router _                         = HTTPure.notFound
```

❶ Add this import.

❷ Pattern Match the `Request` binding `method` to a Variable called `method`.

❸ We're disallowing `GET` requests.

Rerun the server ignoring the warning message and then go to `localhost:3000/hello` in your Browser and you should get a `405` error, i.e. `HTTPure.methodNotAllowed`.

We're getting this error because Browsers make `HTTP GET` requests when you type a URL in the Address Bar and our `router` Function will reject that type of `Request`.

Now, we just need a way to test an `HTTP POST` request.

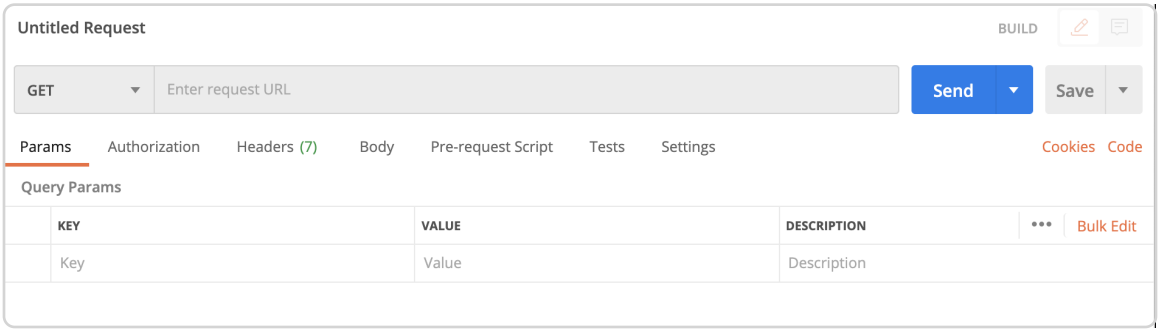
There's a very good tool called `Postman` that let's you send requests to servers. It's great for testing APIs.

Go to the Postman Download Page and download and install `Postman`.

Then run it. If it asks you to sign in, simply select the grey text to Skip. Now, click on the Plus Icon near the top of the window:

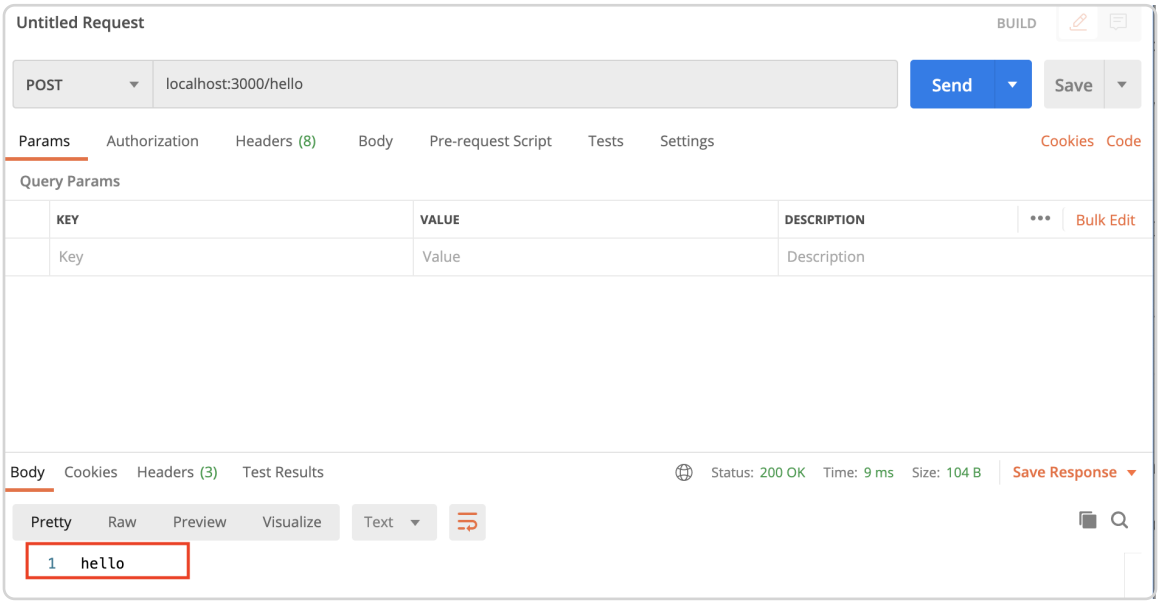


Then you'll get a new tabbed pane:



Pulldown the menu next to `GET` and choose `POST` and then enter `localhost:3000/hello` into the `Request URL Field`.

Then click `Send` and if your server is still running, you should see:



Down at the bottom is our response, `hello`.

Let's make a slight change to our `router` Function to parse path segments:

```
import Data.Array ((!!)) ❶
import Data.Maybe (fromMaybe)
import HTTPure.Lookup ((!@))

router :: Request -> ResponseM
router { path: [ "goodbye" ] }  = HTTPure.ok "goodbye"
router { path, method }
  | method == Get
      = HTTPure.methodNotAllowed ❷
  | method == Post, path !@ 0 == "this" ❸
      = HTTPure.ok $ fromMaybe "missing path[1]" $ path !! 1 ❹
  | method == Post, path !@ 0 == "that" ❺
      = HTTPure.ok $ fromMaybe "missing path[2]" $ path !! 2 ❻
router _                        = HTTPure.notFound
```

❶ Add these imports.

❷ `GET` is not supported.

❸ `POST` where the first path segment is `this` will respond with the second path segment or an error if missing.

❹ If the second path segment is missing, i.e. `path !! 1` is `Nothing`, then `missing path[1]` will be sent.

❺ `POST` where first path segment is `that` will respond with the third path segment or an error if missing.

❻ If the third path segment is missing, i.e. `path !! 2` is `Nothing`, then `missing path[2]` will be sent.

`!!` will return the Value of an Array at the specified index. It'll return `Nothing` if not found.

`!@` is from `HTTPure` and will return `mempty` if not found.

Now rerun your server and you should get the following error:

```
[error] Some of your project files import modules from packages that are not
in the direct dependencies of your project.
To fix this error add the following packages to the list of dependencies in
your config:
- arrays
- maybe
```

Update your `spago.dhall` to include these depedencies and then rerun the server.

Now go to `Postman` and `POST` to `localhost:3000/this/first/second` and see what you get.

You should get `first`. Now, change `this` to `that` and `POST` again.

Did you get `second`? Good.

The code for `router` is ugly and hard to read. Create a `postRouter` Function and factor out the `method == Post` cases before reading on.

Here's my take on it:

```
postRouter :: Request -> ResponseM
postRouter { path }
  | path !@ 0 == "this"
              = HTTPure.ok $ fromMaybe "missing path[1]" $ path !! 1
  | path !@ 0 == "that"
              = HTTPure.ok $ fromMaybe "missing path[2]" $ path !! 2
  | otherwise = HTTPure.notFound

router :: Request -> ResponseM
router { path: [ "goodbye" ] }  = HTTPure.ok "goodbye"
router request@{ method } ❶ ❷
  | method == Get               = HTTPure.methodNotAllowed
  | method == Post              = postRouter request
router _                        = HTTPure.notFound
```

❶ Added `request@` to name the Parameter while destructuring.

❷ We no longer need to destructure `path` here.

This cleans up the code a bit.

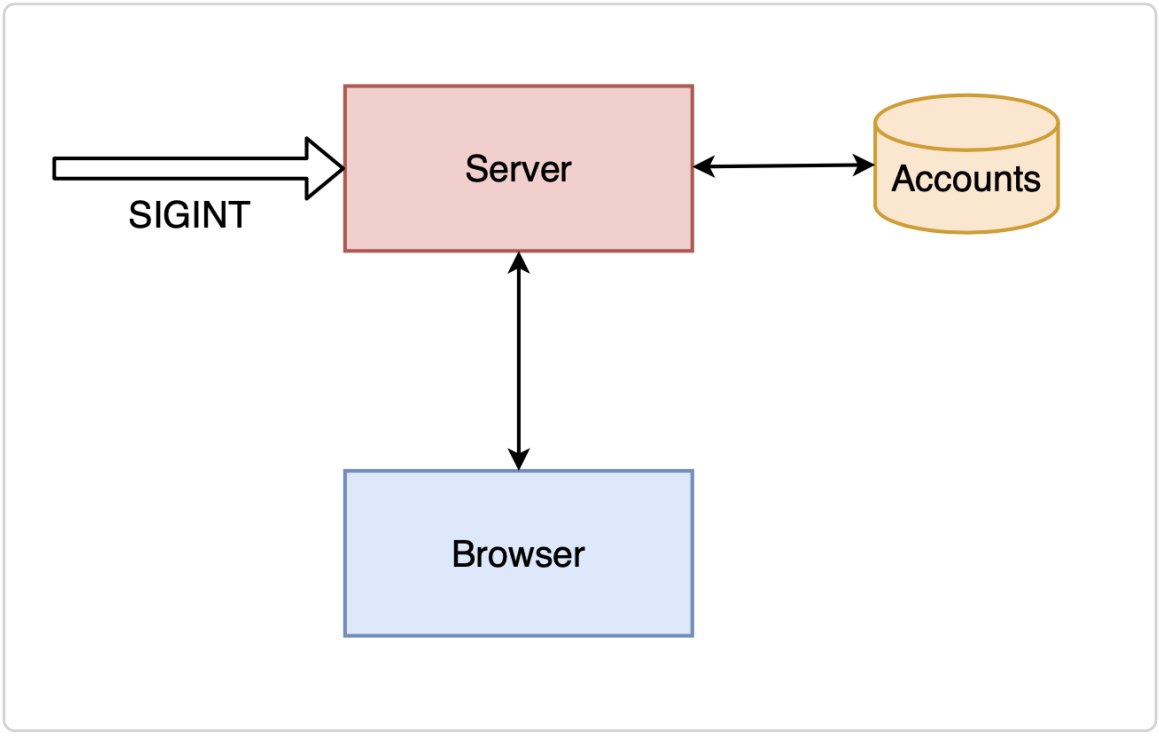
28.3. API For a Front-end Application

Now that we've cut our teeth on a few simple examples, let's create an API for the front-end application we're going to write in a later Chapter.

The application is going to do a few simple things:

- Logon a user
- Logoff a user
- Query for users
- Create a user

One of the first things I do when I'm starting to design something new is to draw a picture of what I'm going to build. I want to start this process as soon as possible since it helps me to understand what pieces are involved in the system:



All requirements must map somehow to this drawing, e.g. logging on a user will involve the `Browser`, `Server` and `Accounts`.

Another great thing about a drawing is it gives all those working on it a common model to work from. It's a lot easier for me to put a picture of what's in my head into your head by just showing you a drawing or diagram, as opposed to trying to do so with lots of words.

Too many people try to write words to convey what a picture can do in a fraction of the time and effort.

In more sophisticated systems, this initial drawing will change many times over as the system is reviewed by others and new issues come up that expose deficiencies in the original design. But this early design is important to get out as fast as possible so that you and your team have something to throw darts at.

I doubt we'll find too much wrong with this drawing since our application is going to be very simple, but who knows. Time will tell.

Notice the `SIGINT` in the diagram. This is a way to represent that we're going to handle `Ctrl-C` to do a clean shutdown of our server.

Also, notice the icon of a `disk`. It could be implemented as a File or Database. But at this level of detail, it's just there to show that the User Accounts are persisted, i.e. they're not transient and will be remembered upon a restart of our server.

Let's make some major implementation decisions. The Accounts will NOT be kept in a Database such as `Postgres` and it won't be stored in a Key/Value Store like `Redis`. These would all be perfectly good solutions.

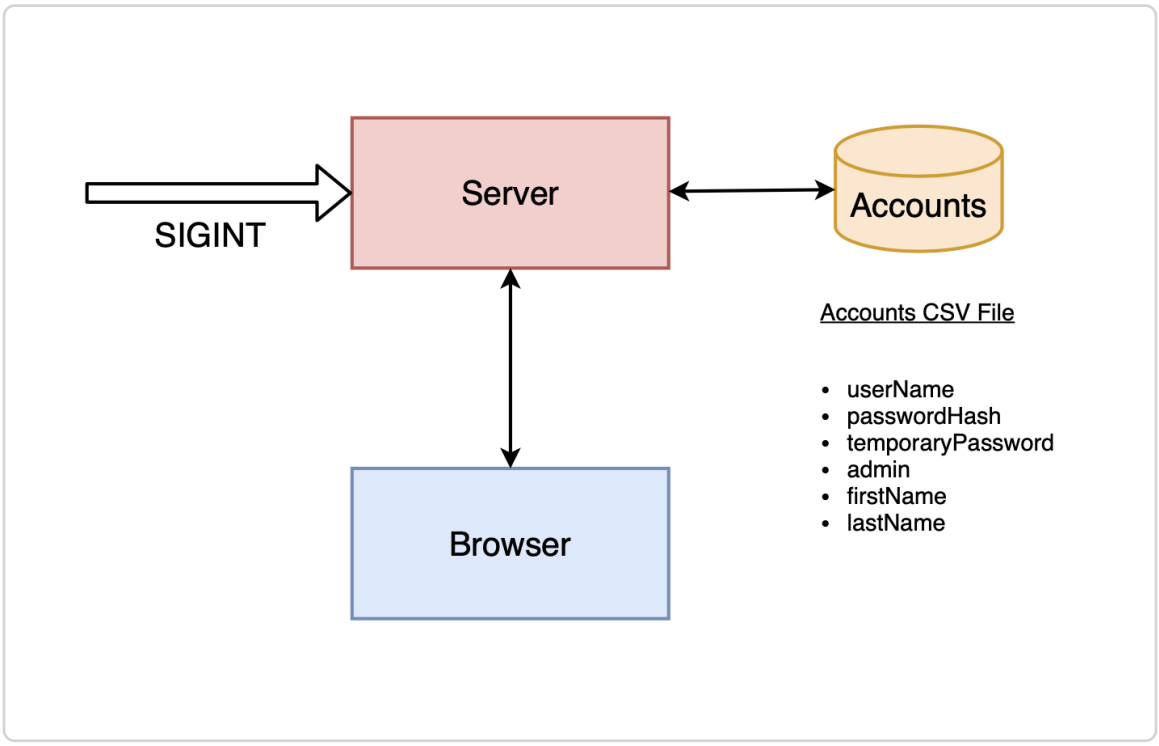
But, I don't want to get bogged down installing these tools onto your development system when we're trying

to learn PureScript and, more specifically, when we're learning how do we build a real application in a Purely Functional Programming Language.

So, instead, we're just going to read the data from a file. But, just because we're being expedient doesn't mean we won't try to do some best practices, e.g. we're not going to store passwords in the Accounts File in Plain Text. We'll actually hash them as is the norm in real-world systems.

I'd like to capture this new decision somewhere and I don't like it just buried here in the words of the previous paragraph because requirements like that tend to get lost or forgotten.

I want to note this decision on our diagram, which means I was wrong about the drawing not changing:



The elements of the CSV (Comma Separated Values) file are as follows:

- `userName` - A unique identifier for logging on to the system.
- `passwordHash` - The hash that's calculated from the password.
- `temporaryPassword` - `true` if the current password MUST be changed upon the next logon.
- `admin` - `true` if Administrative Account.
- `firstName` - User's first name.
- `lastName` - User's last name.

The next step is to write a formal definition for our API. We'll do that based on the list of functions that our application will perform.

The first API call will be to try to logon a user. Let's define what that JSON looks like, both the Request made by the `Browser` and all possible responses by the `Server`.

But before we do, let's decide that we're going to use the `Foreign` library to do our encoding and decoding. We're making this decision for 2 reasons.

First, we're coding both the Client and Server so we can completely control the API definition such that we can use this library. And second, this library is the easiest to use and can be mostly derived saving us unnecessary coding.

As we design the API, we'll keep in mind the behavior of this library so that we can leverage it to do most, if not all, of the encoding and decoding for us without requiring any handwritten code.

Here's the Logon Request:

```
{ "tag": "LogonRequest",
  "contents": {
    "userName": "joeuser",
    "password": "password"
  }
}
```

And the Logon Response for success:

```
{ "tag": "LogonResponse",
  "contents": {
    "tag": "LogonResultsSuccess",
    "contents": {
      "authToken": "e0fc09ad-894a-4399-9940-870895c27394", ❶
      "mustChangePassword": true
    }
  }
}
```

❶ All API calls except `Logon` will require an `authToken` to be honored. That's why `authToken` is returned upon a successful logon.

Logon Response for failure:

```
{ "tag": "LogonResponse",
  "contents": {
    "tag": "LogonResultsFailure" ❶
  }
}
```

❶ We're not providing any reason for the logon failure, e.g. `Invalid user name` or `Invalid password`

since we do not want to give the user, which could be a hacker, any information that they could use to guess a `userName` or `password`.

Next, let's design the Logoff Request:

```
{ "tag": "LogoffRequest",
  "contents": {
    "authToken": "e0fc09ad-894a-4399-9940-870895c27394" ❶
  }
}
```

❶ This is the `authToken` that the user got from the `LogonRequest`.

Logoff Response for success:

```
{ "tag": "LogoffResponse",
  "contents": {
    "tag": "LogoffResultsSuccess"
  }
}
```

Logoff Response for failure:

```
{ "tag": "LogoffResponse",
  "contents": {
    "tag": "LogoffResultsFailure" ❶
  }
}
```

❶ The only reason for a `Logoff` failure is that the `authToken` isn't valid, i.e. the user isn't logged on. This is why there is no `reason` field here.

Next, the Query Users Request:

```
{ "tag": "QueryUsersRequest",
  "contents": {
    "authToken": "e0fc09ad-894a-4399-9940-870895c27394"
  }
}
```

Query Users Response for success:

```
{ "tag": "QueryUsersResponse",
  "contents": {
    "tag": "QueryUsersResultsSuccess",
    "contents": {
      "users": [ ❶
        { "tag": "User",
          "contents": {
            "userName": "joeuser",
            "temporaryPassword": true,
            "admin": true,
            "firstName": "Joe",
            "lastName": "Mama"
          }
        },
        { "tag": "User",
          "contents": {
            "userName": "janeuser",
            "temporaryPassword": false,
            "admin": false,
            "firstName": "Jane",
            "lastName": "Mama"
          }
        }
      ]
    }
  }
}
```

❶ Notice that `users` is an Array of users.

Query Users Response for failure:

```
{ "tag": "QueryUsersResponse",
  "contents": {
    "tag": "QueryUsersResultsFailure",
    "contents": {
      "reason": { "tag": "NotAuthorized"} ❶ ❷
    }
  }
}
```

❶ `NotAuthorized` is if the user account making the request is NOT an Administrative Account.

❷ Another possible `tag` Values is `NotAuthenticated` when `authToken` is invalid.

Create User Request:

```
{ "tag": "CreateUserRequest",
  "contents": {
    "authToken": "e0fc09ad-894a-4399-9940-870895c27394",
    "user": {
      "userName": "janeuser",
      "password": "password", ❶
      "temporaryPassword": false,
      "admin": false,
      "firstName": "Jane",
      "lastName": "Mama"
    }
  }
}
```

❶ This field is in addition to our `User` Entity. We'll have to handle this specially somehow.

Create User Response for success:

```
{ "tag": "CreateUserResponse",
  "contents": {
    "tag": "CreateUserResultsSuccess"
  }
}
```

Create User Response for failure:

```
{ "tag": "CreateUserResponse",
  "contents": {
    "tag": "CreateUserResultsFailure",
    "contents": {
      "reason": { "tag": "AlreadyExists"} ❶ ❷
    }
  }
}
```

❶ `AlreadyExists` is if the user account already exists with the specified `userName`.

❷ Other possible `tag` Values are `NotAuthenticated` when `authToken` is invalid or `NotAuthorized` when the user isn't an admin.

28.4. Modeling the API

Now that we've designed our API, we should model its Types.

First thing to do is to create a Folder or Directory under the `src` Directory and call it `Api`.

Then create a file in the `Api` Directory called `Logon.purs`.

Then code the Types for the following JSON objects in `Logon.purs`:

```
{ "tag": "LogonRequest",
  "contents": {
    "userName": "joeuser",
    "password": "password"
  }
}

{ "tag": "LogonResponse",
  "contents": {
    "tag": "LogonResultsSuccess",
    "contents": {
      "authToken": "e0fc09ad-894a-4399-9940-870895c27394",
      "mustChangePassword": true
    }
  }
}

{ "tag": "LogonResponse",
  "contents": {
    "tag": "LogonResultsFailure"
  }
}
```

Go as far as you can before reading any further.

The first thing to do is to add the `module` definition:

```
module Api.Logon where
```

One very important thing to notice here is that the `module` definition is NOT `Logon`. It's `Api.Logon` since it's under the `Api` directory.

Next, we'll model the request:

```
newtype LogonRequest = LogonRequest ❶
  { ??? ❷

  }
```

❶ The Data Constructor will be the `tag` field in the JSON.

❷ The Record will be the `contents` field in the JSON.

We're NOT using Type Aliases so that we will get `tags` in the JSON. We want that so we'll get some Type safety in our JSON structures.

Next, we'll add the Record fields:

```
newtype LogonRequest = LogonRequest
  { userName :: String
  , password :: String
  }
```

We should stop here and think about the `password` here.

It's defined as a `String`. This is NOT secure, but not because it's a `String`. It's not secure since we're sending it from the Browser to the server in Plain Text.

That's because our server is using `HTTP`, not `HTTPS`. Since we're just writing a learning program, this will be fine, but we should always keep this in mind when designing any logon system.

By the way, you may think that encrypting the password on the client would solve this problem. But, it does not. The Browser is inherently insecure. We have debugging tools build right in to every Browser on every desktop. Basically, hacker tools have been built into every web application on the planet.

In fact, encrypting our password on the client and then sending it over an insecure channel is only mildly more secure than sending it in plain text.

In the plain-text scenario, a hacker just has to intercept the transmission and they can read the password.

When the password is encrypted on the client, a hacker has to first intercept the encrypted password. Then they'd have to understand how the code that encrypted the password works. And then they could decrypt it.

This is possible because the encryption code is available to the hacker since it's encrypted in the Browser and if the hacker can access the logon page, they'll have access to that code.

That code could then be put into a program to allow the hacker to use a brute-force attack by trying passwords until the stolen encryption code produces the encrypted password that the hacker intercepted.

The second scenario is much more work for the hacker and will keep some hackers out, but it's far from

secure and both should be considered dangerous.

In other words, all logon systems that use passwords should communicate over HTTPS. If you make a server that communicates over HTTP, then you should put it behind a Load Balancer or Reverse Proxy that does communicate over HTTPS.

Then you put the insecure server behind a firewall, i.e. you make sure that it's not accessible from the Internet. The only way to communicate with your insecure server is to go through the Reverse Proxy or Load Balancer.

With that out of the way, we'll model the Logon Response based on the following JSON:

```
{ "tag": "LogonResponse",
  "contents": {
    "tag": "LogonResultsSuccess",
    "contents": {
      "authToken": "e0fc09ad-894a-4399-9940-870895c27394",
      "mustChangePassword": true
    }
  }
}
```

Notice how `LogonResultsSuccess` is a `tag`. This is the antithesis of `LogonResultsFailure` in the failure case:

```
{ "tag": "LogonResponse",
  "contents": {
    "tag": "LogonResultsFailure"
  }
}
```

First thing to do then is to make a type for these `tags`, which are Data Constructors:

```
import Data.UUID (UUID) -- COMPILER ERROR!!

data LogonResults
  = LogonResultsSuccess ❶
    { authToken :: UUID ❷
    , mustChangePassword :: Boolean
    }
  | LogonResultsFailure
```

❶ Data Construtor is `tag` in the JSON.

❷ Record is `contents` in JSON.

Update you code to match this if you haven't already.

The compiler error is because this package is not part of our `spago.dhall`, so let's add it:

```
{-
Welcome to a Spago project!
You can edit this file as you like.
-}
{ name = "my-project"
, dependencies =
  [ "aff"
  , "arrays"
  , "console"
  , "effect"
  , "httpure"
  , "maybe"
  , "prelude"
  , "psci-support"
  , "uuid" ❶
  ]
, packages = ./packages.dhall
, sources = [ "src/**/*.purs", "test/**/*.purs" ]
}
```

❶ Keeping the list in alphabetical order.

Then kill the server (if it's still running) and run:

```
npx spago build
```

We'll still ignore the warning regarding `aff`.

Now if you save `Logon.purs`, the compiler error should disappear.

But there's one more thing to do. Go look up `UUID` on `Pursuit` and then click on the package name in the Search Results.

Read what it says in the package docs, then return here.

Here's what I found:

> Wrapper for the `uuid` npm package. This package is not installed automatically, as it comes from `npm`. Install it with `npm install -S uuid`.

To install the `npm` package and update your `package.json`, run:

```
npm install --save uuid@8.3.2 ❶
```

❶ In the docs, they say to use `-S` which is equivalent to `--save`.

Here are the response JSON structures for reference:

```
{ "tag": "LogonResponse",
  "contents": {
    "tag": "LogonResultsSuccess",
    "contents": {
      "authToken": "e0fc09ad-894a-4399-9940-870895c27394",
      "mustChangePassword": true
    }
  }
}

{ "tag": "LogonResponse",
  "contents": {
    "tag": "LogonResultsFailure"
  }
}
```

And finally, we need to define the Type for `LogonResponse`:

```
newtype LogonResponse = LogonResponse LogonResults
```

Add this Type and then create `Encode` and `Decode` Instances for `LogonRequest`, `LogonResults` and `LogonResponse`. Use the code you wrote in other Chapters as reference for how to do this.

Do this before reading on.

First, here are those changes for `LogonRequest`:

```
import Data.Generic.Rep (class Generic) ❶
import Foreign.Generic (genericEncode, genericDecode)
import Foreign.Generic.Class (class Encode, class Decode, defaultOptions)

newtype LogonRequest = LogonRequest ❷
  { userName :: String
  , password :: String
  }
derive instance genericLogonRequest :: Generic LogonRequest _

instance encodeLogonRequest :: Encode LogonRequest where
  encode = genericEncode defaultOptions

instance decodeLogonRequest :: Decode LogonRequest where
  decode = genericDecode defaultOptions
```

❶ Add these imports.

❷ This definition was already here, we just added the line to derive a `Generic` Instance.

If you didn't finish, please stop and see if you can finish the `Encode` and `Decode` Instance for `LogonResults` and `LogonResponse`, then read on.

Here are the changes for `LogonResults`:

```
data LogonResults ❶
  = LogonResultsSuccess
    { authToken :: UUID
    , mustChangePassword :: Boolean
    }
  | LogonResultsFailure
derive instance genericLogonResults :: Generic LogonResults _

instance encodeLogonResults :: Encode LogonResults where
  encode = genericEncode defaultOptions

instance decodeLogonResults :: Decode LogonResults where
  decode = genericDecode defaultOptions
```

❶ This definition was already here, we just added the line to derive a `Generic` Instance.

If you didn't finish, please stop and see if you can finish the `Encode` and `Decode` Instance for `LogonResponse`, then read on.

Here are the changes for `LogonResponse`:

```
newtype LogonResponse = LogonResponse LogonResults ❶
derive instance genericLogonResponse :: Generic LogonResponse _

instance encodeLogonResponse :: Encode LogonResponse where
  encode = genericEncode defaultOptions

instance decodeLogonResponse :: Decode LogonResponse where
  decode = genericDecode defaultOptions
```

❶ This definition was already here, we just added the line to derive a `Generic` Instance.

Now, before we rush off to model any other JSON Requests and Responses, we should write some test code to make sure we aren't going down a bad path.

Create a file called `Test.purs` under `src` and add the following code:

```
module Test where

import Prelude

import Api.Logon (LogonRequest(..), LogonResponse(..), LogonResults(..))
import Effect (Effect)
import Effect.Class.Console (log)

test :: Effect Unit
test = do
  log "placeholder" ❶
```

❶ We'll remove this later.

Don't worry too much about the compiler warnings regarding redundant imports. This is just test code that we'll discard when we're done with it.

Next, comment out `main` in `Main.purs` and replace it with:

```
import Effect (Effect) ❶
import Test (test)

main :: Effect Unit
main = test
```

❶ Added these imports.

Again, don't worry about redundant imports just yet.

Now run the server with:

```
npx spago run
```

And then fix the error:

```
[error] Some of your project files import modules from packages that are not
in the direct dependencies of your project.
To fix this error add the following packages to the list of dependencies in
your config:
- foreign-generic
```

Now, run your program again with `npx spago run` and when you do, it should output:

```
placeholder
```

Now, remove the `log "placeholder"` line from the `test` Function. Then add code to first create a `LogonRequest` and then encode it into JSON. And finally, display that JSON to console.

After you've finished, return here.

Here's that code:

```
import Foreign.Generic (encodeJSON) ❶

test :: Effect Unit
test = do
  log $ encodeJSON
    $ LogonRequest { userName: "joeuser", password: "password" } ❷
```

❶ Import the `encodeJSON` Function.

❷ Create a test `LogonRequest`.

Now run your program and see what you get. Then before reading on, compare that against the following to see if the structure is the same:

```
{ "tag": "LogonRequest",
  "contents": {
    "userName": "joeuser",
    "password": "password"
  }
}
```

Here's what the test code outputs:

```
{"contents":{"userName":"joeuser","password":"password"},"tag":"LogonRequest"}
```

Even though the top-level JSON keys are in a different order, this matches our API design and so we're good there.

Next, comment out the `log` line and add one just like it but for `LogonResponse` instead and see if its output matches the following:

```
{ "tag": "LogonResponse",
  "contents": {
    "tag": "LogonResultsSuccess",
    "contents": {
      "authToken": "00000000-0000-0000-0000-000000000000",
      "mustChangePassword": true
    }
  }
}
```

Use `emptyUUID` for a dummy `authToken` and return here once you're done.

Here's the code to add to `test`:

```
import Data.UUID (emptyUUID) ❶

test :: Effect Unit
test = do
  -- log $ encodeJSON
  --   $ LogonRequest { userName: "joeuser", password: "password" } ❷
  log $ encodeJSON $ LogonResponse
    $ LogonResultsSuccess
          { authToken: emptyUUID, mustChangePassword: true } ❸
```

❶ Import `emptyUUID` to get a test UUID.

❷ Commented out this code.

❸ Added this line of code.

Running this outputs:

```
{"contents":{"contents":{"mustChangePassword":true,"authToken":"00000000-0000-
0000-0000-000000000000"},"tag":"LogonResultsSuccess"},"tag":"LogonResponse"}
```

And finally, comment out that `log` line and add code to test `LogonResponse` for the failure case and see if it has the same structure as:

```
{ "tag": "LogonResponse",
  "contents": {
    "tag": "LogonResultsFailure"
  }
}
```

Do this before reading on.

Here are the changes to `test`:

```
test :: Effect Unit
test = do
  -- log $ encodeJSON
  --   $ LogonRequest { userName: "joeuser", password: "password" }
  -- log $ encodeJSON $ LogonResponse
  --  $ LogonResultsSuccess
  --        { authToken: emptyUUID, mustChangePassword: true }
  log $ encodeJSON $ LogonResponse $ LogonResultsFailure
```

And when run it outputs:

```
{ "contents": { "tag": "LogonResultsFailure" }, "tag": "LogonResponse" }
```

And this too matches.

Looks like we're moving in the right direction.

Next, we're going to work on the `Logoff` API. Here's the design for reference:

```
{ "tag": "LogoffRequest",
  "contents": {
    "authToken": "e0fc09ad-894a-4399-9940-870895c27394"
  }
}

{ "tag": "LogoffResponse",
  "contents": {
    "tag": "LogoffResultsSuccess"
  }
}

{ "tag": "LogoffResponse",
  "contents": {
    "tag": "LogoffResultsFailure"
  }
}
```

Now, create a file under the `Api` directory called `Logoff.purs` and do the same as we've done for `Logon`. Then return here to see how you did.

Here's the full implementation since it's pretty much the same as `Logon.purs`:

```
module Api.Logoff where ❶

import Data.Generic.Rep (class Generic)
import Data.UUID (UUID)
import Foreign.Generic (genericEncode, genericDecode)
import Foreign.Generic.Class (class Encode, class Decode, defaultOptions)

newtype LogoffRequest = LogoffRequest { authToken :: UUID }
derive instance genericLogoffRequest :: Generic LogoffRequest _

instance encodeLogoffRequest :: Encode LogoffRequest where
  encode = genericEncode defaultOptions

instance decodeLogoffRequest :: Decode LogoffRequest where
  decode = genericDecode defaultOptions

data LogoffResults = LogoffResultsSuccess | LogoffResultsFailure
derive instance genericLogoffResults :: Generic LogoffResults _

instance encodeLogoffResults :: Encode LogoffResults where
  encode = genericEncode defaultOptions

instance decodeLogoffResults :: Decode LogoffResults where
  decode = genericDecode defaultOptions

newtype LogoffResponse = LogoffResponse LogoffResults
derive instance genericLogoffResponse :: Generic LogoffResponse _

instance encodeLogoffResponse :: Encode LogoffResponse where
  encode = genericEncode defaultOptions

instance decodeLogoffResponse :: Decode LogoffResponse where
  decode = genericDecode defaultOptions
```

❶ Don't forget the `Api` prefix on the module name since this file is under the `Api` directory.

And here's the `QueryUsers` API design for reference:

```
{ "tag": "QueryUsersRequest",
  "contents": {
    "authToken": "e0fc09ad-894a-4399-9940-870895c27394"
  }
}

{ "tag": "QueryUsersResponse",
  "contents": {
    "tag": "QueryUsersResultsSuccess",
    "contents": {
      "users": [
        { "tag": "User",
          "contents": {
            "userName": "joeuser",
            "temporaryPassword": true,
            "admin": true,
            "firstName": "Joe",
            "lastName": "Mama"
          }
        },
        { "tag": "User",
          "contents": {
            "userName": "janeuser",
            "temporaryPassword": false,
            "admin": false,
            "firstName": "Jane",
            "lastName": "Mama"
          }
        }
      ]
    }
  }
}

{ "tag": "QueryUsersResponse",
  "contents": {
    "tag": "QueryUsersResultsFailure",
    "contents": {
      "reason": { "tag": "NotAuthorized"} ❶
    }
  }
}
```

❶ Don't forget that this can also be `NotAuthenticated`.

Now, create a file under the `Api` directory called `QueryUsers.purs` and code this as far as you can get. This will have some small stumbling blocks, but give it a try and see if you can overcome them. Then return here to see how you did.

This code has a few wrinkles compared to the others, so we'll take it a little bit slower. Here's the part that's pretty much the same as the rest:

```
module Api.QueryUsers where ❶

import Data.Generic.Rep (class Generic)
import Data.UUID (UUID)
import Foreign.Generic (genericEncode, genericDecode)
import Foreign.Generic.Class (class Encode, class Decode, defaultOptions)

newtype QueryUsersRequest = QueryUsersRequest { authToken :: UUID }
derive instance genericQueryUsersRequest :: Generic QueryUsersRequest _

instance encodeQueryUsersRequest :: Encode QueryUsersRequest where
  encode = genericEncode defaultOptions

instance decodeQueryUsersRequest :: Decode QueryUsersRequest where
  decode = genericDecode defaultOptions
```

❶ Don't forget the `Api` prefix on the module name since this file is under the `Api` directory.

Now writing the `QueryUsersResults` has a few wrinkles:

```
data QueryUsersResults
  -- COMPILER ERROR!!
  = QueryUsersResultsSuccess { users :: Array User } ❶
  | QueryUsersResultsFailure { reason :: QueryUsersFailureReason } ❷
derive instance genericQueryUsersResults :: Generic QueryUsersResults _

instance encodeQueryUsersResults :: Encode QueryUsersResults where
  encode = genericEncode defaultOptions

instance decodeQueryUsersResults :: Decode QueryUsersResults where
  decode = genericDecode defaultOptions
```

❶ First issue is that we need to define `User`.

❷ Second issue is that we need to create a Type for `QueryUsersFailureReason`.

You may have coding something different here and it could very well be just fine, but so you can continue to work through this Chapter, change your code to match this even though we have a few undefined items.

Next, we'll need to define `User`, but I'd like to keep the `User` definition out of the `Api` module. It really is an Entity in our system.

So, create a new Folder under `src` called `Entity` and then add a file called `User.purs` under it.

Then code up the `User` entity code and its `Encode` and `Decode` Instances using the following for reference:

```
{ "tag": "QueryUsersResponse",
  "contents": {
    "tag": "QueryUsersResultsSuccess",
    "contents": {
      "users": [
        { "tag": "User",
          "contents": {
            "userName": "joeuser",
            "temporaryPassword": true,
            "admin": true,
            "firstName": "Joe",
            "lastName": "Mama"
          }
        },
        { "tag": "User",
          "contents": {
            "userName": "janeuser",
            "temporaryPassword": false,
            "admin": false,
            "firstName": "Jane",
            "lastName": "Mama"
          }
        }
      ]
    }
  }
}
```

Finish the code for that file before reading on.

Here's the code for `User`:

```
module Entity.User where ❶

import Data.Generic.Rep (class Generic)
import Foreign.Generic (genericEncode, genericDecode)
import Foreign.Generic.Class (class Encode, class Decode, defaultOptions)

newtype User = User ❷
  { userName :: String
  , temporaryPassword :: Boolean
  , admin :: Boolean
  , firstName :: String
  , lastName :: String
  }
derive instance genericUser :: Generic User _

instance encodeUser :: Encode User where
  encode = genericEncode defaultOptions

instance decodeUser :: Decode User where
  decode = genericDecode defaultOptions
```

❶ Don't forget the `Entity` prefix on the module name since this file is under the `Entity` directory.

❷ Even though our diagram has `passwordHash` in it for the CSV, we should protect that information as much as possible. This is why it's missing from `User`. Otherwise, it could accidentally be sent to the client in `QueryUsersResponse`.

Now, import `Entity.User` into `Api.QueryUsers`. This should make the compiler error go away on `User`.

Then write the Type for `QueryUsersFailureReason` in `Api.QueryUsers` and make it encodable and decodable. Here is the JSON for reference:

```
{ "tag": "QueryUsersResponse",
  "contents": {
    "tag": "QueryUsersResultsFailure",
    "contents": {
      "reason": { "tag": "NotAuthorized"} ❶
    }
  }
}
```

❶ Don't forget that this can also be `NotAuthenticated`.

Once you're finished coding `QueryUsersFailureReason`, `Api.QueryUsers` should compile.

Stop reading and give that a go.

Here's the code for `QueryUsersFailureReason`:

```
data QueryUsersFailureReason = NotAuthorized | NotAuthenticated
derive instance genericQueryUsersFailureReason
    :: Generic QueryUsersFailureReason _

instance encodeQueryUsersFailureReason
    :: Encode QueryUsersFailureReason where
  encode = genericEncode defaultOptions

instance decodeQueryUsersFailureReason
    :: Decode QueryUsersFailureReason where
  decode = genericDecode defaultOptions
```

Next, add the code for `QueryUsersResponse` to `Api.QueryUsers` before moving on.

Here's the complete implementation:

```
module Api.QueryUsers where

import Data.Generic.Rep (class Generic)
import Data.UUID (UUID)
import Entity.User (User) ❶
import Foreign.Generic (genericEncode, genericDecode)
import Foreign.Generic.Class (class Encode, class Decode, defaultOptions)

newtype QueryUsersRequest = QueryUsersRequest { authToken :: UUID }
derive instance genericQueryUsersRequest :: Generic QueryUsersRequest _

instance encodeQueryUsersRequest :: Encode QueryUsersRequest where
  encode = genericEncode defaultOptions

instance decodeQueryUsersRequest :: Decode QueryUsersRequest where
  decode = genericDecode defaultOptions

data QueryUsersResults
  = QueryUsersResultsSuccess { users :: Array User }
  | QueryUsersResultsFailure { reason :: QueryUsersFailureReason }
derive instance genericQueryUsersResults :: Generic QueryUsersResults _

instance encodeQueryUsersResults :: Encode QueryUsersResults where
  encode = genericEncode defaultOptions
```

```
instance decodeQueryUsersResults :: Decode QueryUsersResults where
  decode = genericDecode defaultOptions

data QueryUsersFailureReason = NotAuthorized | NotAuthenticated
derive instance genericQueryUsersFailureReason :: Generic
QueryUsersFailureReason _

instance encodeQueryUsersFailureReason
    :: Encode QueryUsersFailureReason where
  encode = genericEncode defaultOptions

instance decodeQueryUsersFailureReason
    :: Decode QueryUsersFailureReason where
  decode = genericDecode defaultOptions

newtype QueryUsersResponse = QueryUsersResponse QueryUsersResults ❷
derive instance genericQueryUsersResponse :: Generic QueryUsersResponse _

instance encodeQueryUsersResponse :: Encode QueryUsersResponse where
  encode = genericEncode defaultOptions

instance decodeQueryUsersResponse :: Decode QueryUsersResponse where
  decode = genericDecode defaultOptions
```

❶ Here's where we imported `Entity.User` a few steps back.

❷ Here's the code for `QueryUserResponse`.

Make sure your code matches this.

Now, create a file under the `Api` directory called `CreateUser.purs`. We're going to code this API one piece at a time so don't be afraid to steal from what we've already done.

Here are the requirements for reference:

```
{ "tag": "CreateUserRequest",
  "contents": {
    "authToken": "e0fc09ad-894a-4399-9940-870895c27394",
    "user": {
      "userName": "janeuser",
      "password": "password",
      "temporaryPassword": false,
      "admin": false,
      "firstName": "Jane",
      "lastName": "Mama"
    }
  }
}

{ "tag": "CreateUserResponse",
  "contents": {
    "tag": "CreateUserResultsSuccess"
  }
}

{ "tag": "CreateUserResponse",
  "contents": {
    "tag": "CreateUserResultsFailure",
    "contents": {
      "reason": { "tag": "AlreadyExists"} ❶
    }
  }
}
```

❶ Other possible `tag` Values are `NotAuthenticated` when `authToken` is invalid or `NotAuthorized` when the user isn't an admin.

For now, only code up the `CreateUserRequest` and once you're done, return back here.

Here's the code for just `CreateUserRequest`:

```
module Api.CreateUser where

import Data.Generic.Rep (class Generic)
import Data.UUID (UUID)
import Entity.User (User)
import Foreign.Generic (genericEncode, genericDecode)
import Foreign.Generic.Class (class Encode, class Decode, defaultOptions)

newtype CreateUserRequest = CreateUserRequest
  { authToken :: UUID
  , user :: User (1)
  }
derive instance genericCreateUserRequest :: Generic CreateUserRequest _

instance encodeCreateUserRequest :: Encode CreateUserRequest where
  encode = genericEncode defaultOptions

instance decodeCreateUserRequest :: Decode CreateUserRequest where
  decode = genericDecode defaultOptions
```

(1) This is a problem since we need an additional field, `password`, that's not part of the `User` Entity.

Since, we have an additional field, i.e. `password`, to add to the `User` Entity, we're going to need to do some extra work here.

There are a few possible options. First, we add `password` to `User`. This is a bad option since we don't want to leak the `password` to the client when querying the `Users`.

Second, we create another data structure that's like `User` but has an extra field, `password`. If we do this correctly, we can avoid duplication of code, but it is a bit burdensome.

And finally, we simply change our API requirements to move `password` to the same level as `user`. We can do this since we're in charge of both the client and server.

Normally, I'd pick the final solution of moving `password` out, but we'd lose out on a learning opportunity if we did that. So, instead, we're going to create another data structure that has all of the things that `User` has plus `password`.

This approach will teach us how to handle this case when we're not able to change the server's API.

The first thing we'll need to do is return back to the `User` Entity:

```
newtype User = User
  { userName :: String
  , temporaryPassword :: Boolean
  , admin :: Boolean
  , firstName :: String
  , lastName :: String
  }
```

We need to factor out the fields from the Record. Do you remember how to do this?

We factor them out into a `Row` and then define the `Record` in terms of the `Row`. As a reminder, here's an example of a simple `Record` built from a `Row`:

```
type ARow r = ( a :: Int, b :: Int | r ) ❶
type ARecord = { | ARow ( c :: Int ) } ❷
```

❶ The `r` lets us use this Row and extend it by adding additional fields.

❷ Here we're extending `ARow` with `c`.

Factor out the fields from `User` into a `Row` called `UserRow` and then use that `Row` to define `User`. Do this in `Entity.User` before reading on.

The first thing to do is factor out the fields from `User` into a `Row` that can be extended with `r`:

```
type UserRow r =
  ( userName :: String
  , temporaryPassword :: Boolean
  , admin :: Boolean
  , firstName :: String
  , lastName :: String
  | r
  )
```

And next we build the `User` Record from this `Row Type`:

```
newtype User = User { | UserRow () } ❶
```

❶ We've provided an empty `Row` since we don't need to extend `UserRow`, since, by design, it has everything that `User` needs.

Another syntax for doing this is:

```
newtype User = User (Record (UserRow ()))
```

Personally, I prefer this syntax.

Now, that we have that, we can update our `CreateUserRequest` to use `UserRow` extending it to add the `password` field. Do that before reading on.

Here's that modification:

```
import Entity.User (UserRow) ❶

newtype CreateUserRequest = CreateUserRequest
  { authToken :: UUID
  , user :: Record (UserRow ( password :: String )) ❷
  }
```

❶ Replaced the `User` import with `UserRow`.

❷ Added the `password` field.

Next, write the Response code. Hint: You can steal code from `QueryUsers`. Here's the JSON for reference:

```
{ "tag": "CreateUserResponse",
  "contents": {
    "tag": "CreateUserResultsSuccess"
  }
}

{ "tag": "CreateUserResponse",
  "contents": {
    "tag": "CreateUserResultsFailure",
    "contents": {
      "reason": { "tag": "AlreadyExists"} ❶
    }
  }
}
```

❶ Other possible `tag` Values are `NotAuthenticated` when `authToken` is invalid or `NotAuthorized` when the user isn't an admin.

Do this before reading any further.

Here's the Response code:

```
-- COMPILER ERROR!!
newtype CreateUserResponse = CreateUserResponse CreateUserResults ❶
derive instance genericCreateUserResponse
  :: Generic CreateUserResponse _

instance encodeCreateUserResponse :: Encode CreateUserResponse where
  encode = genericEncode defaultOptions

instance decodeCreateUserResponse :: Decode CreateUserResponse where
  decode = genericDecode defaultOptions
```

❶ `CreateUserResults` hasn't been defined yet.

The compiler error is expected since we haven't defined `CreateUserResults`. Do that now, remembering that you can steal code from `QueryUsers`, then continue reading.

Here's that code:

```
data CreateUserResults
  = CreateUserResultsSuccess
  -- COMPILER ERROR!!
  | CreateUserResultsFailure { reason :: CreateUserFailureReason } ❶
derive instance genericCreateUserResults :: Generic CreateUserResults _

instance encodeCreateUserResults :: Encode CreateUserResults where
  encode = genericEncode defaultOptions

instance decodeCreateUserResults :: Decode CreateUserResults where
  decode = genericDecode defaultOptions
```

❶ `CreateUserFailureReason` hasn't been defined yet.

Once again, we are expecting this compiler error since we haven't defined `CreateUserFailureReason` yet.

Now, we have to define `CreateUserFailureReason`. There are 3 possible errors, `NotAuthorized`, `NotAuthenticated` and `AlreadyExists`. So add `CreateUserFailureReason` before moving on. Hint: Look at the code from `QueryUsers` for clues on how to write this.

Here's the code for that:

```
data CreateUserFailureReason
  = NotAuthorized
  | NotAuthenticated
  | AlreadyExists
derive instance genericCreateUserFailureReason
    :: Generic CreateUserFailureReason _

instance encodeCreateUserFailureReason
    :: Encode CreateUserFailureReason where
  encode = genericEncode defaultOptions

instance decodeCreateUserFailureReason
    :: Decode CreateUserFailureReason where
  decode = genericDecode defaultOptions
```

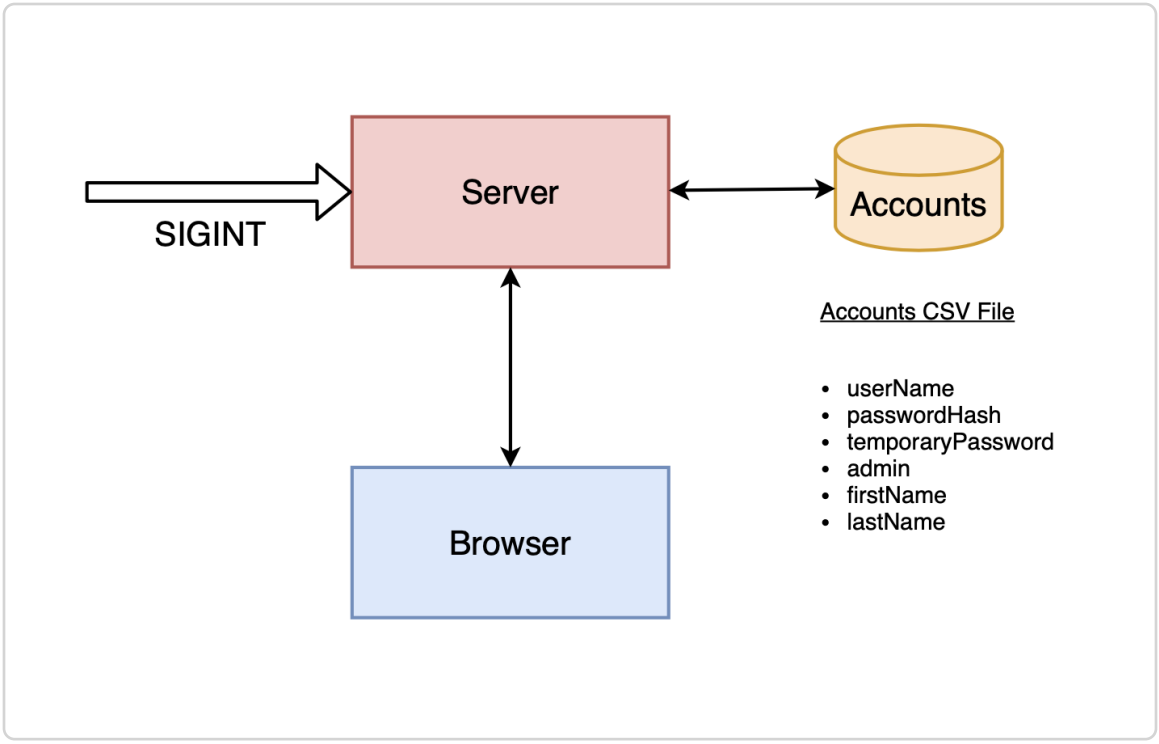
This module should now compile.

28.5. Coding the Server's Infrastructure

Now that we have our API modeled, some of which we've already tested, and we have all the Entities involved in that API, we're ready to build the plumbing of our server.

This is really the operational part of the server, i.e. it's infrastructure, e.g. the startup, shutdown, saving and loading accounts, and setting up the API handler.

Let's look back at our original diagram:



Do you see that red box labeled `Server`? It's time to drill down and design what's inside of that box.

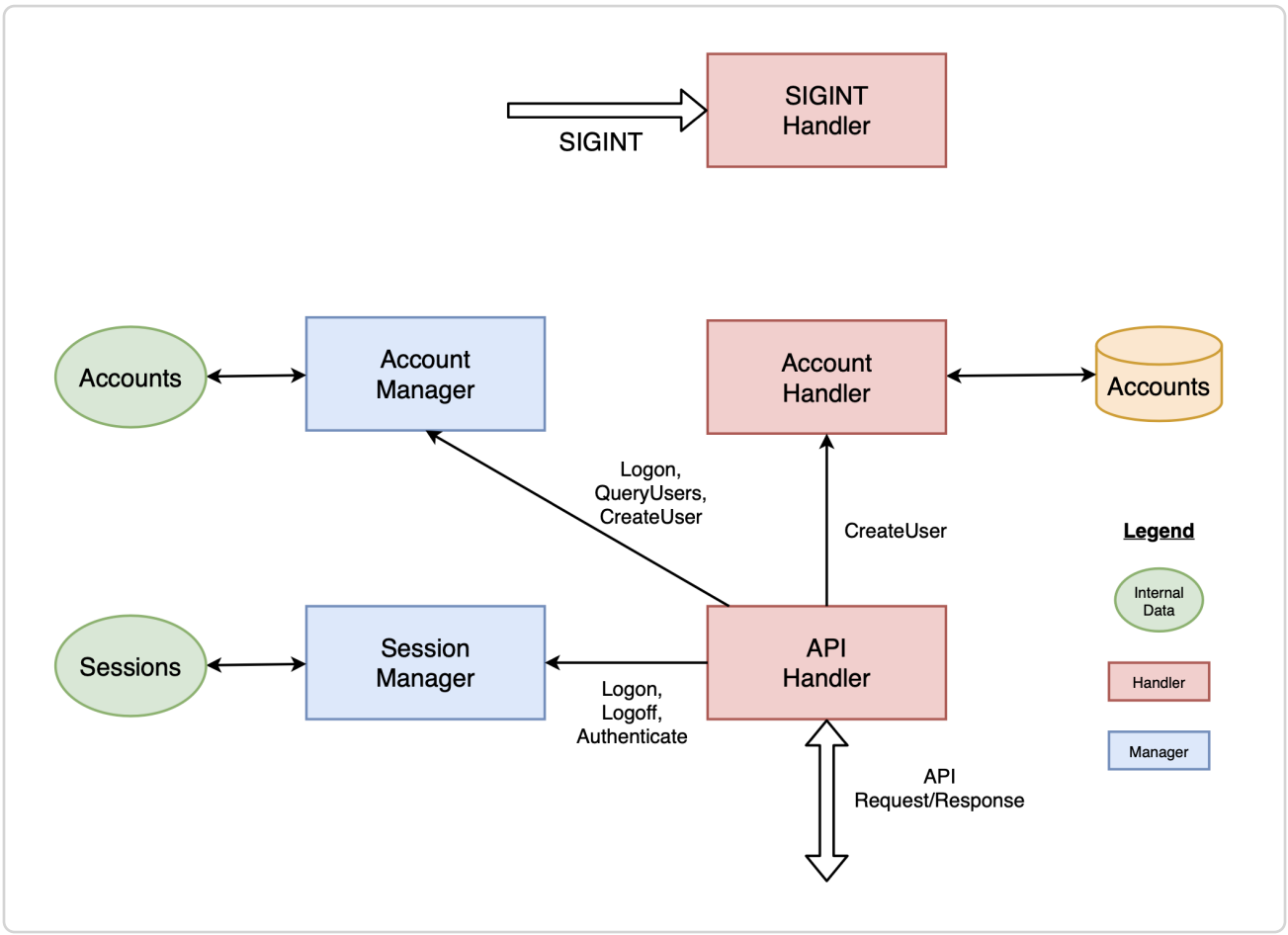
List out all of the moving parts to the server that you can think of before reading my list.

Here's my list:

- Account Management
  - Load ALL accounts
  - Save an account
- SIGINT Handler
- API Handler
  - Authorization
  - Logon/Logoff
  - Query Users
  - Create User
- Session Management
  - Logon
  - Logoff
  - Authentication

This is the best I can do at this point. There may be other things that we'll add later as we discover them, but this is a good starting point.

Here's a diagram that tries to capture the structure of our server:



In this diagram, the difference between a `Handler` and a `Manager` is whether it interfaces to the outside world or not.

While the nomenclature is arbitrary, the distinction is helpful because it tells us that the Managers should only interface with internal parts of our server, whereas the Handlers will interface to both internal parts of our server and external elements outside of our server, e.g. the File System and the client.

The two-way arrows between the boxes and ellipses shows the flow of information. The one-way arrows with text gives us an idea of which API calls will be making calls to the receiving box, i.e. where the arrow points, e.g. during the `Logon` API call, information regarding the `Accounts` will be needed to verify that the `Logon` request was made by a user of the system.

There is another one-way arrow labeled `Logon` that goes from the API Handler to the Session Manager. This is to show that after the user has been authenticated, that a new session needs to be created.

This is not meant to capture every last detail, but it's there to help us think and talk about our design, and subsequently our code. It's very difficult to build such a diagram like this from just looking at code. And having this diagram before we code, will make the process so much easier.

We'll modify this diagram if we discover some unforeseen issue that requires an architectural restructuring.

Looking at our design, what part would you do next?

Actually, the part I would do first isn't shown here, and that's the `main` of our program. And if we do that first, we should probably code the `SIGINT Handler` next.

First, delete or, better yet, comment out the follow from your `Main.purs`:

```
-- import Data.Array ((!!))
-- import Data.Maybe (fromMaybe)
-- import HTTPure.Lookup ((!@))
-- import HTTPure.Method (Method(..))
-- import HTTPure.Server (ServerM)
-- import Test (test)

-- postRouter :: Request -> ResponseM
-- postRouter { path }
--   | path !@ 0 == "this"
--               = HTTPure.ok $ fromMaybe "missing path[1]" $ path !! 1
--   | path !@ 0 == "that"
--               = HTTPure.ok $ fromMaybe "missing path[2]" $ path !! 2
--   | otherwise = HTTPure.notFound

-- router :: Request -> ResponseM
-- router request@{ method }
--   | method == Get               = HTTPure.methodNotAllowed
--   | method == Post              = postRouter request
-- router { path: [ "goodbye" ] }  = HTTPure.ok "goodbye"
-- router _                        = HTTPure.notFound

-- main :: Effect Unit
-- main = test
```

This should essentially leave the following code:

```
port :: Int
port = 3000
```

Next, create a new `main` that starts our server capturing the shutdown Function. When you do, you'll need a `router` so just create a `router` stub for now, i.e. a `router` Function that does nothing.

Also, don't worry about the compiler warnings about redundant imports for now. We'll clean them up at the end.

Do just this much for now and then return here once you're done.

First, we're going to make a `router` stub starting with the Type Signature:

```
router :: Request -> ResponseM
```

This takes a `Request` and then runs in or, more accurately, returns the `Response` Monad.

Next, we'll just ignore the `Request` and return `404`:

```
router :: Request -> ResponseM
router _ = HTTPure.notFound
```

That's our stub.

Next, we'll create our `main's` Type Signature:

```
main :: Effect Unit
```

And now we'll run in the `Effect` Monad:

```
main :: Effect Unit
main = do
  ???
```

And we'll start the server capturing the shutdown Callback:

```
main :: Effect Unit
main = do
  shutdown <- HTTPure.serve port router ❶
    $ log $ "Server up running on port: " <> show port ❷
```

❶ We'll use `shutdown` later.

❷ Don't miss the third Paramter to `HTTPure.server`, i.e. the Effect to run AFTER the server is running.

And finally, we'll return `pure unit` for now to keep the compiler happy:

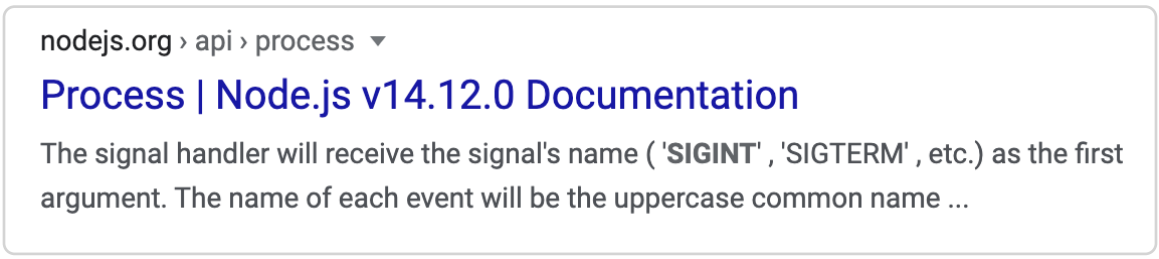
```
main :: Effect Unit
main = do
  shutdown <- HTTPure.serve port router
    $ log $ "Server up running on port: " <> show port
  pure unit
```

This should run so give it a try. Once it runs, you will have to kill the server with `Ctrl-C`.

Next, we can write the `Handler` for `SIGINT`. This is the signal that the OS sends to your program when you type `Ctrl-C`. But, how do we figure out how to do this?

Well, the first place I'd look is to see how `Node` handles SIGINT. To do that, we can use Google . Go ahead and search Google and see what you find.

I searched Google for the term `node SIGINT` and found:



When I click on that I get a big list of API Functions. So I search the page, via `Cmd-F` or `Ctrl-F` on Windows, for `SIGINT` and I find the example code:

```
// Begin reading from stdin so the process does not exit.
process.stdin.resume();

process.on('SIGINT', () => {
  console.log('Received SIGINT. Press Control-D to exit.');
});

// Using a single function to handle multiple signals
function handle(signal) {
  console.log(`Received ${signal}`);
}

process.on('SIGINT', handle);
process.on('SIGTERM', handle);
```

Unfortunately, this `on` function isn't something that I'd expect to be duplicated in PureScript. But just for fun, I'll check `Pursuit` for `on`.

When I do, I have to click on `Load More Results` at the bottom of the page over and over again until it disappears, which means that there are no more results. Then I search the page for `node` and I find nothing.

Just as I suspected, it's not called `on` in PureScript. Now what would you do at this point?

I'm going to search for `Node.Process` on `Pursuit` to see if there's a wrapper for `Node's process` module. When I do I find `Node.Process` as a module. After clicking on it, I search that page for `SIGINT` and nothing shows up.

But while I was typing and I got to `sig`, I noticed that it found `onSignal`:

```
onSignal :: Signal -> Effect Unit -> Effect Unit
```

The docs says:

```
Install a handler for a particular signal.
```

Now, this looks very promising. Go to `Pursuit` and search for `Node.Process` and then follow it until you get to `onSignal`.

Once you do, click on `Signal` in `onSignal's` Type Signature to see if it supports the `SIGINT`, i.e. when user types a `Ctrl-C`.

When I do this, I find a huge list of supported `Unix` Signals. I searched the page for `SIGINT` and I found it.

Next, go to `Pursuit` and look up `HTTPure`, then click on the package `purescript-httpure` and then click on `HTTPure.Server`. Once on that page look for the `serve` Function. I usually search the page with `Cmd-F` or `Ctrl-F` depending on the Operating System.

What did you find out about what `serve` returns?

I found that `serve` returns `ServerM`:

```
serve :: Int -> (Request -> ResponseM) -> Effect Unit -> ServerM
```

Now, click on `ServerM` in the Type Signature on `Pursuit` to see its definition.

Here's what I found:

```
type ServerM = Effect (Effect Unit -> Effect Unit)
```

Since we extracted `shutdown` from `Effect` via `bind`, or more specifically, via `<-` in the `do` block, its Type is:

```
shutdown :: Effect Unit -> Effect Unit
```

It's a Function that takes code written in `Effect` and produces an `Effect`. The means that `shutdown` can take a `do` block and that the return Value of that call can then be passed to `onSignal's` second Parameter:

```
onSignal :: Signal -> (Effect Unit -> Effect Unit) ❶
```

❶ Added redundant Parentheses to help us see that the Parenthetical is the same Type Signature for `shutdown`.

Okay, based on all of this research, add `SIGINT` code to `main` that calls `shutdown` with clean up logic in a `do` block. We don't have any clean up work to do yet, so we'll just log a message to the console that we've shutdown the server. Once you're done coding, return here.

The first thing to do is to include `onSignal`:

```
import Node.Process (onSignal) -- COMPILER ERROR!!
```

This compiler error lets us know that we're going to have to include some package in `spago.dhall`. Use `Pursuit` to see what package `Node.Process` is in, then add it to your `spago.dhall` if you haven't already. Then read on.

Here's the change to `spago.dhall`:

```
{-
Welcome to a Spago project!
You can edit this file as you like.
-}
{ name = "my-project"
, dependencies =
  [ "aff"
  , "arrays"
  , "console"
  , "effect"
  , "foreign-generic"
  , "httpure"
  , "maybe"
  , "node-process" ❶
  , "prelude"
  , "psci-support"
  , "uuid"
  ]
, packages = ./packages.dhall
, sources = [ "src/**/*.purs", "test/**/*.purs" ]
}
```

❶ Added here to maintain alpha order.

Run:

```
npx spago build
```

The next change to make is to setup the signal handler:

```
main :: Effect Unit
main = do
  shutdown <- HTTPure.serve port router
    $ log $ "Server up running on port: " <> show port
  onSignal SIGINT do ❶
    ???
  pure unit
```

❶ We pass a `do` block to `onSignal`.

We'll log that the server is shutting down. This is useful if the shutdown takes long:

```
main :: Effect Unit
main = do
  shutdown <- HTTPure.serve port router
    $ log $ "Server up running on port: " <> show port
  onSignal SIGINT do
    log "Shutting down server..." ❶
    ???
  pure unit
```

❶ Log before we call `shutdown`.

For now, we're just logging that we're shutting down. In the future, we may want to do some other clean up in the `do` block.

And finally, we'll call `shutdown` to shutdown the HTTP server:

```
main :: Effect Unit
main = do
  shutdown <- HTTPure.serve port router
    $ log $ "Server up running on port: " <> show port
  onSignal SIGINT do -- COMPILER ERROR!! ❶
    log "Shutting down server..."
    shutdown $ log "Server shutdown."
```

❶ We don't need `pure unit` anymore since `onSignal` returns `Effect Unit`.

From the compiler error, we can see that we need to import the `SIGINT` Data Constructor. If you haven't done this yet, figure out what to import from `Pursuit` before reading any more.

According to `Pursuit`, `Signal` is in `Data.Posix.Signal`:

```
import Data.Posix.Signal (Signal(SIGINT)) ❶
```

❶ Include `SIGINT` Data Constructor. We could have also imported `Signal(..)`.

Add this import and make sure you code matches `main` above.

Also, add `posix-types` to `spago.dhall`.

Now, before you run this, there's something you'll need to know. Our `SIGINT` handler will NOT work if we run our server with:

```
npx spago run
```

With this command, `npx` is run, which, in turn, runs `node` as a child process. So when `Ctrl-C` is pressed at the command line, it goes to the parent process, which is `npx` and not `node`. Upon receipt of `SIGINT`, `npx` will terminate causing our program to terminate. Before that happens though, your program will be sent `SIGTERM`, which we aren't handling at the moment.

We can get around this problem by running our program like:

```
npx spago build && node -e "require('./output/Main/index').main()" ❶ ❷
```

❶ The `&&` is a way to run a second command if the first one succeeds. You can also manually run these commands one right after the other.

❷ `node -e "require('./output/Main/index').main()"` came from the Spago documentation as the one extra command that's run for `spago run` compared with `spago build`.

Now, when you hit `Ctrl-C`, `node` will get the `SIGINT`, which will cause your signal handler to be called.

Go ahead and run your program this new way and once it's up and running, hit `Ctrl-C` and see what happens.

It should print out:

```
Server up running on port: 3000
^CShutting down server...
Sever shutdown.
```

We can also support `SIGTERM` instead of `SIGINT` or better yet, we support both since we're not always going to run our server using `npx`.

Another good reason to handle `SIGTERM` is if you're going to run your server in a Container like Docker. If you run your server as the main app in a Docker container, then, by default, Docker will send your server a SIGTERM before the Container is shutdown.

Many times, servers need to do real clean up tasks and trapping this signal is advantageous.

Change your code to handle the both the `SIGINT` and `SIGTERM` signals before reading any further.

First, we'll add `SIGTERM` to our imports:

```
import Data.Posix.Signal (Signal(SIGINT, SIGTERM)) ❶
```

❶ If we had included all Data Constructors with `(..)`, we wouldn't have to do this. But in this rare case, I like having them listed out explicitly since it lets us know that these are the ONLY 2 signals that are supported.

Next, we'll factor out the shutdown code:

```
main :: Effect Unit
main = do
  shutdown <- HTTPure.serve port router
    $ log $ "Server up running on port: " <> show port
  let shutdownServer = do ❶
        log "Shutting down server..."
        shutdown $ log "Server shutdown."
  onSignal SIGINT shutdownServer ❷
```

❶ Factored out the `do` block as `shutdownServer`.

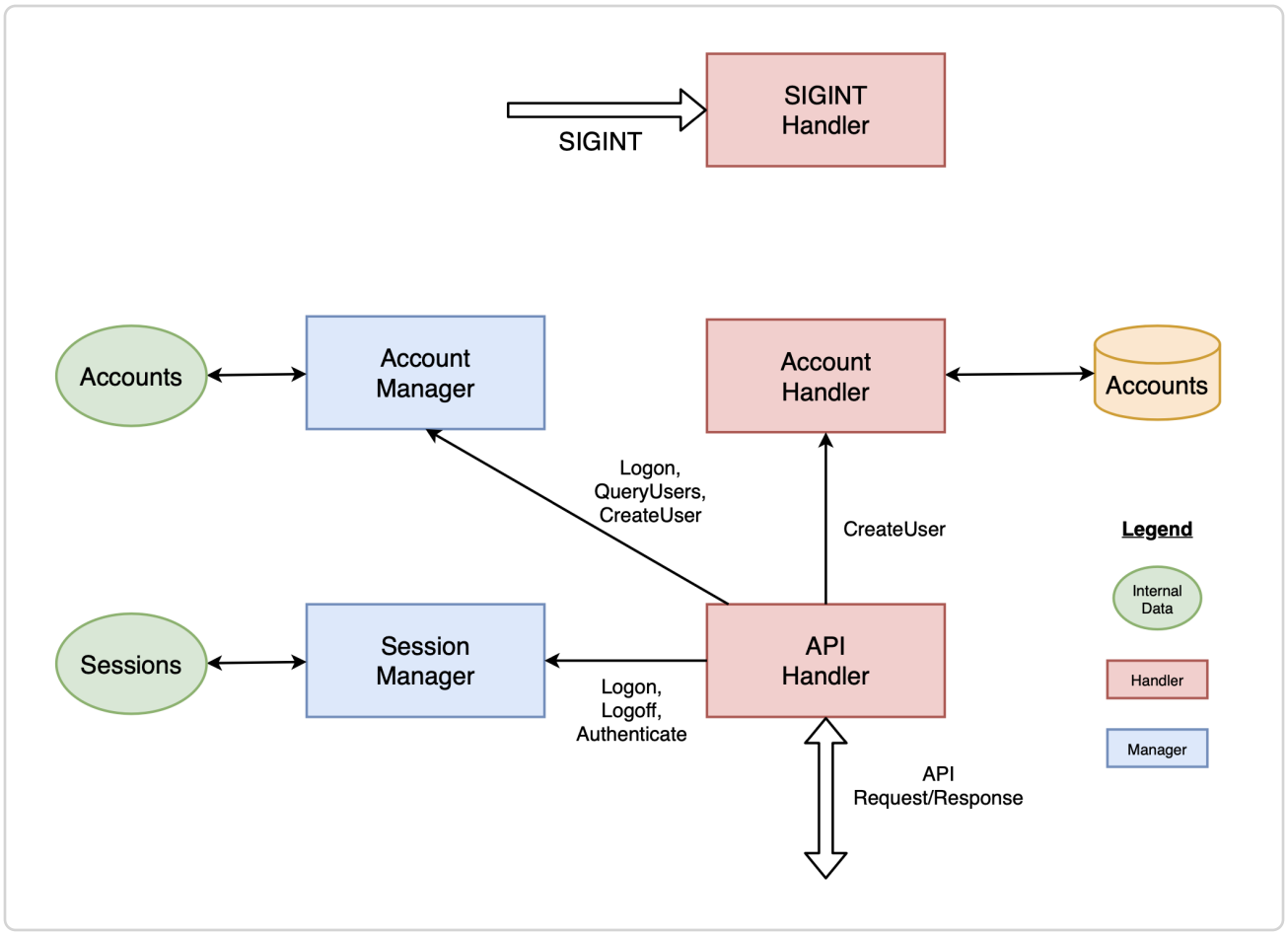
❷ Removed `do` block and using `shutdownServer`.

And finally, we handle `SIGTERM`:

```
main :: Effect Unit
main = do
  shutdown <- HTTPure.serve port router
    $ log $ "Server up running on port: " <> show port
  let shutdownServer = do
        log "Shutting down server..."
        shutdown $ log "Server shutdown."
  onSignal SIGINT shutdownServer
  onSignal SIGTERM shutdownServer ❶
```

❶ Added `SIGTERM` handler.

Now, we're ready to continue adding features. But before we do, we should consult our diagram:



Take a look and see what you'd work on next. Keep in mind the reasons you made that particular choice.

So, when I look at this diagram, I can see that I could write either a Manager or a Handler next. But, I feel I'm a bit too ignorant on what the Handlers are going to need from the Managers at this time.

Because of that, I'd like to wait and build the Managers as the Handlers need support.

That leaves the 2 Handlers. Which one would you start with and why? Take a minute to think about it and decide why you made that decision.

When I consider this, I feel that I understand more about what's needed by the API Handler than I do about the Account Handler. From the diagram, I can see that the API Handler will call Functions in the Account Handler. This means that, like the Managers, the API Handler will help dictate the detailed requirements for the Account Handler.

So, based on this line of thinking, my approach would be to start with the API Handler and stop from time to time to build out the parts that it'll depend on. Another approach one could take is to stub out these other parts, but to me that's requires a lot more knowledge about those parts than I have at the moment.

Also, stub code has to be coded, debugged and tested only to eventually be thrown out. I'd rather not do the extra work if I don't have to. Sometimes, this is unavoidable, but I don't think we're in that situation here.

The decision then is to start with the API Handler.

When I'm implementing an API, I like to build the code in the order that it'll most likely be called in, e.g. we should code `Logon` before `Logoff`. By doing it this way, I can test my code as I write it.

Imagine, instead, that we coded `Logoff` before `Logon`. In that case, we wouldn't be able to confidently test `Logoff` until `Logon` was done.

So, we might as well, code in the general order the API calls will be made. And that means that `Logon` is first.

We already have modules where we modeled our API Requests and Responses, e.g. `Api.Logon`. We could put our Handler code in those modules, or we could keep them as just our model definitions and put our Handler code in their own modules.

Which do you think we should do and why?

My thought is that if I put too much into those files, I may regret it in the future and need to refactor then. The Separation of Concerns up front is nice, but will require a little bit of rearranging at the moment.

I'm on the fence, but will error on the side of overly separating. So, let's do that.

Add a new folder under `src` called `Data` and then move the `Api` folder under it by dragging and dropping it.

Next change each file's module definition to start with `Data.Api` now that they're in a new location.

Then comment out every line of `Test.purs` except for the `module` declaration. `Test.purs` has outlived its usefullness and we don't want future changes we make to break our build since it most likely will no longer compile. We're commenting instead of deleting it so that we can keep if for reference.

Since, `Logon` is the first API call we will have to make, let's code that up and then we can test it.

First, create a Folder under `src` called `Handler` and then create a Folder under `src/Handler` called `Api`.

Then in that directory create a file called `Logon.purs` with the following:

```
module Handler.Api.Logon where
```

Before we start coding the `Logon` Handler, let's look at our `router`, which will need to decode the response and then dispatch it to the appropriate Function:

```
router :: Request -> ResponseM
router _ = HTTPure.notFound
```

Okay, so our `router` is a bit anemic at the moment. We need it to decode the Request and then based on what it decoded, call the appropriate Handler.

What's the best way to do this? What do you think?

We could start with `oneOf` on the decoder:

```
oneOf [
  decodeJSON request :: LogonReqeust
, decodeJSON request :: LogoffReqeust
-- etc.
]
```

The problem is that this won't compile. `Arrays` are Homogeneous in PureScript.

How might we get around this problem?

One way is to create a Sum Type to unify the Types:

```
data ApiRequest
  = LogonApiRequest LogonRequest
  | LogoffApiRequest LogoffRequest
```

But the downside to this approach is that we'll have to use a `case` expression that handles the dispatching:

```
case request of
  LogonApiRequest (LogonRequest _) -> LogonHandler.handle request
  LogoffApiRequest (LogoffRequest _) -> LogoffHandler.handle request
  -- etc.
```

This is just ugly boilerplate.

Another way is to create a Typeclass, e.g. `ApiRequest` and make `LogonRequest`, `LoggoffRequest`, etc. have an instance of this Typeclass:

```
class ApiRequest
```

This is what I call a Marker Class, i.e. it just marks that these Types are related.

But it too suffers from requiring a `case` expression.

I like Typeclasses for this, but the above solution is coming at it from the wrong direction. We should not make a Typeclass for the Requests, but, instead make a Typeclass for the Handlers:

```
class ApiHandler a where -- COMPILER WARNING!!
  handle :: String -> Either MultipleErrors Unit ❶
```

❶ The first Parameter is the `Request`.

The `handle` Method will error with `MultipleErrors`, which is the decoder's error Type, if it cannot decode the Request or return `Unit` if it successfully handles the `Request`.

Since `Either` has an `Alternative` Instance that picks the first `Right` it encounters, we can use `oneOf` in our `router`.

The following compiler warning is because our `a` here can be multiple Types, i.e. many Kinds or what is called `Polymophic Kinds` or `Polykinds` for short:

```
ApiHandler :: ∀ (k :: Type). k → Constraint

  The inferred kind for the class declaration ApiHandler contains polymorphic kinds.
  Consider adding a top-level kind signature as a form of documentation.

    class ApiHandler :: forall k. k -> Constraint


in type class declaration for ApiHandler
 PureScript(MissingKindDeclaration)
```

We can easily fix this by adding the following Type Signature:

```
class ApiHandler :: ∀ k. k -> Constraint ❶ ❷
class ApiHandler a where
  handle :: String -> Either MultipleErrors Handler
```

❶ Added the Type Signature to specify that `ApiHandler` takes a `Type` Parameter and returns a `Constraint`, which is the Type for a Typeclass.

❷ I left off the explicit Type for `k` which is shown in the compiler warning because it's implied in the context of `∀`.

But there's still a problem with this Typeclass definition. Can you see it?

There is a Type Parameter to the Typeclass, i.e. `a`, which doesn't show up in the `handle` Method's Type Signature:

```
class ApiHandler :: ∀ k. k -> Constraint
class ApiHandler a where ❶
  handle :: String -> Either MultipleErrors Unit
```

❶ The Type Parameter, `a`, doesn't show up in `handle's` Type Signature.

Because of this, there's no way for the compiler to know which version of `handle` to call. You'll need to fix this flaw when you code this.

Create a folder under `Handler` called `Class` and then create a file under `Handler/Class` called `ApiHandler.purs` and place the code for this Typeclass there.

Then fix the aforementioned problem and get it to compile.

Do that before reading on.

Here's the full file:

```
module Handler.Class.ApiHandler where ❶

import Data.Either (Either)
import Data.Unit (Unit) ❷
import Foreign (MultipleErrors)
import Type.Proxy (Proxy)

class ApiHandler :: ∀ k. k -> Constraint
class ApiHandler a where
  handle :: String -> Proxy a -> Either MultipleErrors Unit ❸
```

❶ Take note of the `module`'s name. See how it mirrors its path in the File System.

❷ You could also import `Prelude` to get `Unit`.

❸ We added a `Proxy` Parameter to get the Type Parameter `a` into `handle`'s Type Signature.

Now, return to `Handler.Api.Logon` and make a `Logon` Type and then make that an Instance of `ApiHandler` that decodes the `Request` into a `LogonRequest` and then handles that `Request`. Do this before moving on.

If you got stuck on creating the Type, here it is:

```
data Logon = Logon
```

It doesn't need to be anymore complicated than that. It just needs a Data Constructor.

If you got stuck on this Type, add it to your code and then finish by writing the `ApiHandler` Instance. Then continue reading.

We'll code this one line at a time starting with the instance definition:

```
import Handler.Class.ApiHandler (class ApiHandler) ❶

instance apiHandlerLogon :: ApiHandler Logon where
```

❶ Import `ApiHandler` Typeclass.

And now the left-hand side of the implementation:

```
instance apiHandlerLogon :: ApiHandler Logon where
  handle request _ = do ❶ ❷
    ???
```

❶ Don't care about the `Proxy` Parameter since its only there to inform the compiler which `handle` implementation to call.

❷ We start with a `do` block since we're going to code in the `Either` Monad. We code in `Either` because both `handle` and `decodeJSON` return `Either`.

Next, we'll decode the `request` using `decodeJSON`:

```
import Prelude ❶

import Control.Monad.Except (runExcept)
import Data.Api.Logon (LogonRequest)
import Foreign.Generic (decodeJSON)

instance apiHandlerLogon :: ApiHandler Logon where
  handle request _ = do
    logonReq <- runExcept (decodeJSON request :: _ LogonRequest) ❷
    ???
```

❶ Add these imports.

❷ Don't forget that we need to `runExcept` the result of `decodeJSON`.

And finally, we're going to want to call the Logon Handler, which we'll call `handler`:

```
instance apiHandlerLogon :: ApiHandler Logon where
  handle request _ = do
    logonReq <- runExcept (decodeJSON request :: _ LogonRequest)
    pure handler -- COMPILER ERROR!!
```

We have an expected compiler error since we haven't written `handler` yet.

Update your code to match this.

Next, we'll want to write a stubbed-out version of `handler`, i.e. one that does nothing. Remember, `handle` will return `Right unit` on success. This means that `handler` will return `Unit`.

Just write the Type Signature for `handler` before moving on.

Here's the Type Signature:

```
handler :: LogonRequest -> Unit
```

Okay, one thing we know about Pure Functions is that any Pure Function that returns `Unit` cannot do anything. However, a Monadic Function can return `Effect Unit`, which means it only has Side-effects.

But, I think we're going to want `handler` to return a response. That means we'll have to change our `handler` to return `ResponseM`. But if we do that, then our Typeclass Method `handle` will also need to return `ResponseM`.

Here's it current definition:

```
class ApiHandler :: ∀ k. k -> Constraint
class ApiHandler a where
  handle :: String -> Proxy a -> Either MultipleErrors Unit
```

How can we change it to do that? See if you can answer that before reading the answer.

We want to make sure that we're still working in the `Either` Monad so we still get the benefit of its `Alternative` Instance in `router` and it's Short-cicuiting effect when the request doesn't decode.

So, we won't change the `Either`, but instead we'll simply change `Unit` to `ResponseM` in `Either's` second Type Parameter:

```
import HTTPure.Response (ResponseM) ❶

class ApiHandler :: ∀ k. k -> Constraint
class ApiHandler a where
  handle :: String -> Proxy a -> Either MultipleErrors ResponseM ❷
```

❶ Import `ResponseM`.

❷ Changed the success Type from `Unit` to `ResponseM`.

Also, remove:

```
import Data.Unit (Unit)
```

Now, rewrite `handler's` Type Signature and finish coding the stub for `handler` before moving on.

Here's the new Type Signature:

```
import HTTPure.Response (ResponseM) ❶

handler :: LogonRequest -> ResponseM
```

❶ Import `ResponseM`.

And now we'll just stub out with a `do` block since we'll probably have many lines of code here eventually:

```
import HTTPure as HTTPure ❶

handler :: LogonRequest -> ResponseM
handler _ = do
  HTTPure.notFound ❷
```

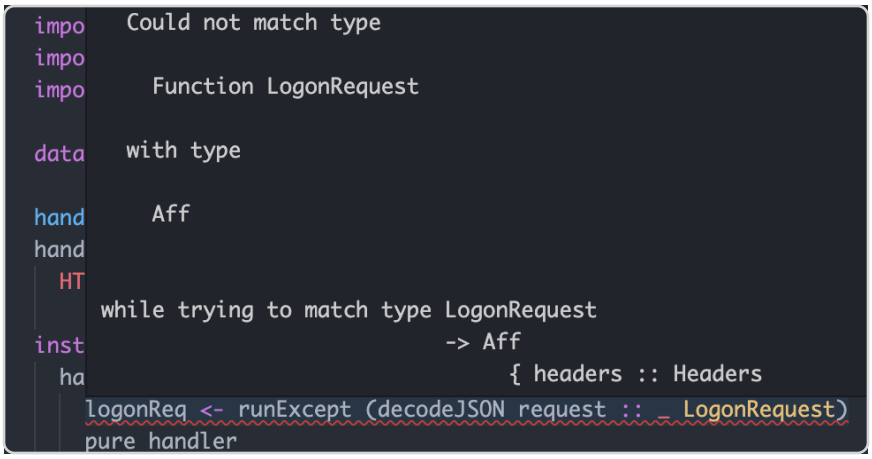
❶ Import `HTTPure` qualified.

❷ A `404` error for now.

Okay, so now we have a compiler error in our instance:

```
instance apiHandlerLogon :: ApiHandler Logon where
  handle request _ = do
    -- COMPILER ERROR!!
    logonReq <- runExcept (decodeJSON request :: _ LogonRequest)
    pure handler
```

Here's the error message:



Can you see my mistake?

Frankly speaking, this error message isn't helpful, but since we just wrote the `handler`, this is probably the problem.

And when I look at how I'm calling it, I'm NOT passing it all of its Parameters. This is probably why the compiler is complaining about a `Function` not matching an `Aff`. Remember, `ResponseM` runs in `Aff`.

So, we'll fix that:

```
instance apiHandlerLogon :: ApiHandler Logon where
  handle request _ = do
    logonReq <- runExcept (decodeJSON request :: _ LogonRequest)
    pure $ handler logonReq ❶
```

❶ Pass the `LogonRequest`.

And now it compiles.

So, what would you do next?

There are 2 tacts. We could stub out all of the other APIs, i.e. a breadth-first approach, and then flesh out a single API, e.g. Logon.

Or we can take a depth-first approach by finishing the `Logon` API.

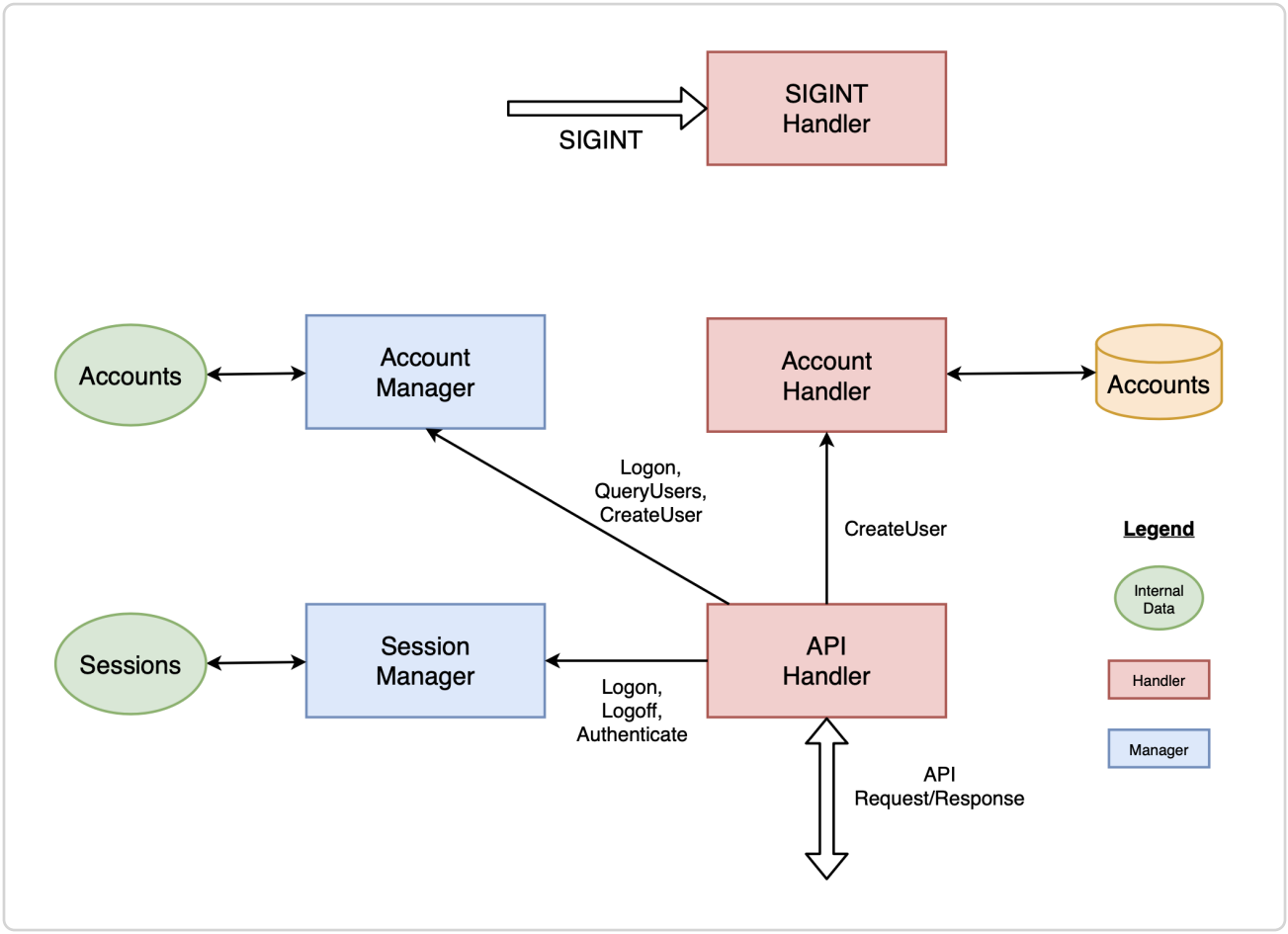
Which is better and why? Think about the why before reading on.

If we do the breath-first approach and we make a mistake, we'll have to fix it everywhere. Doing at least one API from start to finish will inform us on what we don't know at the moment and any mistakes we make along the way, will only need to be fixed once.

So, that's what we're going to do, finish the `handler` Function.

The next question is what does `handler` need to do?

If you're struggling with that answer, take a look back at our diagram:



From this, we can see that we need to consult with the Account Manager to see if the `userName` exist. And once we validate that the user's credentials are valid, we'll need to inform the Session Manager that we have a logged on user.

This means that we cannot continue with the API code until we first develop both the Account Manager and Session Manager.

But, before the Account Manager can be coded, we're going to need to have the Account Handler done otherwise we'll have no `Accounts`.

28.6. Coding the Account Handler

The Account Handler's job is to load `Accounts` from our file and to write a single `Account` to the file when we create a new `User`. That should be enough for what we'll need in this application. We don't really support deletion since this is just a learning application.

There is one thing else that we'll need to do and that's create the file with a Bootstrap Account. The first time we run our server, they will be NO `Accounts`. In that case, we'll first need to create a file with a Bootstrap Account and then load that.

Let's itemize the scope of the Account Handler:

- Create initial Accounts File with a Bootstrap Account with a temporary password
- Load all Accounts from the File
- Append a new Account to the Accounts File

Now, that we understand the scope of work, let's get started.

28.6.1. Coding `loadAccounts`

Make a file under the `Handler` Folder called `Account.purs`. Next, write a Function called `loadAccounts`. This will read a file called `accounts.csv`. This file will have the following format:

```
userName,passwordHash,temporaryPassword,admin,firstName,lastName
```

`loadAccounts` will parse this using `,` as the separator and create an `Array` of `Account` Entities. That means that you'll also need to create a file called `Account.purs` under the `Entity` directory. When you create the `Account` Type, you should leverage `UserRow` to do so.

There's a lot here, so let's take it slowly. First, create the `Account.purs` under the `Handler` directory and just write the Type Signature for `loadAccounts` before returning here.

We have to do File I/O, which means we have to write `loadAccounts` in either `Effect` or `Aff`. If you remember correctly, File I/O in Node that's written in `Effect` has a clunky Callback. So let's write it in `Aff` instead:

```
module Handler.Account where ❶

import Effect.Aff (Aff) ❷

loadAccounts :: Aff (Array Account)
```

❶ Important to get the module name correct. It follows the directory path starting from but not including `src`.

❷ Import `Aff`.

You may have wanted to pass a `String` to `loadAccounts` and that too would've been fine. But, I'm going to make a design decision here and have the name of the Accounts File known only to this module. I don't want the other parts of the program to have to know about this gory little detail that only the Account Handler cares about.

We can already see, that our code is not going to compile since we haven't written `Account`. Let's leave this code be for now, and write the code for the `Account` Entity.

To do that, make a file called `Account.purs` under the `Entity` directory and code up the Type for `Account`. Don't forget to leverage `UserRow`. Hint: `passwordHash` is of Type `String` since we're going to have to persist it to `accounts.csv`.

Return here when you've completed this task.

Here's the code:

```
module Entity.Account where ❶

import Entity.User (UserRow)

newtype Account = Account (Record (UserRow ( passwordHash :: String ))) ❷ ❸
```

❶ Important to get the module name correct. It follows the directory path starting from but not including `src`.

❷ We're using `newType` since there's only 1 Data Constructor and it has only 1 Type Parameter.

❸ `Account` has everything that `User` does with the addition of `passwordHash`.

Returning to our `loadAccounts` Function in `Handler.Account` we'll start coding a `do` block:

```
import Prelude ❶

import Entity.Account (Account) ❷

loadAccounts :: Aff (Array Account)
loadAccounts = do ❸
  ???
```

❶ Import `Prelude` since we're coding in a `do` block and we'll need `bind` and `discard`.

❷ Import `Account` Entity.

❸ Start with a `do` block since we're running under `Aff`.

What should we do next? Read the file, perhaps?

Actually, we'd like to check to see if the file exists first. If not, then we'll write a Bootstrap Account to it.

Then, when we load it, we can be guaranteed that it'll exist. It doesn't mean we will successfully load it, but we'll know that it exists AND there's at least one Account in it.

To check for a file's existence, you may need to consult `Node's` API documentation if `Pursuit` proves too difficult. I'd start with `Pursuit` though.

Before we write the rest of the Function, let's remind ourselves what `readTextFile` does. Search `Pursuit`

for `readTextFile` and see what you find.

When I do, it brings up 3 different versions. From the Type Signatures, we should be able to figure out which one to use.

Can you see which one to use?

If you can't, realize that we decided to do the File I/O in `Aff`. How about now?

We want the one from `Node.FS.Aff`:

```
readTextFile :: Encoding -> FilePath -> Aff String ❶
```

❶ From `Node.FS.Aff`.

Now, search `Pursuit` for `exists` and see what you find.

When I do that I find a lot of Functions, but one of them is also from the same module as `readTextFile`:

```
exists :: String -> Aff Boolean ❶
```

❶ From `Node.FS.Aff`.

Remember, when using Functions from `Node.FS.Aff`, we have to use `try` from `Control.Monad.Error.Class` so errors won't crash our program, but will, instead, return an `Either`:

```
try :: ∀ e m a. MonadError e m => m a -> m (Either e a)
```

Here's where we left off on `loadAccounts`:

```
loadAccounts :: Aff (Array Account)
loadAccounts = do
  ???
```

Using all of the information above, just write the code to check that the file `accounts.csv` exists. Make a Value that's global to the file called `accountsFile` that contains the Accounts File name.

Continue reading once you're done or get stuck.

We'll check the files existence using `exists` from `Node.FS.Aff`:

```
import Prelude ❶

import Control.Monad.Error.Class (try)
import Node.FS.Aff (exists)

accountsFile :: String ❷
accountsFile = "accounts.csv"

loadAccounts :: Aff (Array Account)
loadAccounts = do
  -- COMPILER WARNING!! ❸
  exists <- try $ exists accountsFile ❹
  pure [] ❺
```

❶ Add these imports.

❷ Define the file name so we don't have to keep typing the literal `String`, which are just more opportunities for us to mistype it.

❸ We'll ignore the compiler warning since we're not using `exists` yet.

❹ We `try` to check for existence. Note that Value `exists` is an `Either`.

❺ This is here to make the compiler happy. We're returning an empty `Array` for now.

We'll need to add `node-fs-aff` to `spago.dhall` since we're importing `Node.FS.Aff`:

```
{-
Welcome to a Spago project!
You can edit this file as you like.
-}
{ name = "my-project"
, dependencies =
  [ "aff"
  , "arrays"
  , "console"
  , "effect"
  , "foreign-generic"
  , "httpure"
  , "maybe"
  , "node-fs-aff" ❶
  , "node-process"
  , "posix-types"
  , "prelude"
  , "psci-support"
  , "uuid"
  ]
, packages = ./packages.dhall
, sources = [ "src/**/*.purs", "test/**/*.purs" ]
}
```

❶ Added here to maintain the alpha order.

Build your server with:

```
npx spago build
```

This will give you:

```
[error] Some of your project files import modules from packages that are not
in the direct dependencies of your project.
To fix this error add the following packages to the list of dependencies in
your config:
- either
- foreign
- transformers
```

Add these dependencies to your `spago.dhall` and then rebuild with `npx spago build`. It should now compile fine.

Next, we should write the Bootstrap Account to the file, which will create it, if it doesn't exist. So go ahead and check `exists`, remembering its Type is `Either` and write to the file the following Bootstrap Account if it doesn't exist:

```
-- userName,passwordHash,temporaryPassword,admin,firstName,lastName
bootstrapAccount :: String
bootstrapAccount = "admin,placeholder,true,true,Joe,Admin" ❶
```

❶ The `passwordHash` has `placeHolder` there as a placeholder until we know how we're going to calculate password hashes.

Go ahead and add this code to `Account.purs` and write to the file if it doesn't exist keeping in mind that if we cannot write to the Accounts File, then we'll want to crash the server. This is because something is seriously wrong and we cannot reliably run if we cannot initialize our Accounts File.

Get as far as you can then continue reading.

Here's the code to write the Bootstrap Account:

```
import Data.Either (isLeft) ❶
import Node.Encoding (Encoding(..)) ❷
import Node.FS.Aff (exists, writeTextFile) ❸

loadAccounts :: Aff (Array Account)
loadAccounts = do
  exists <- try $ exists accountsFile
  when (isLeft exists) ❹
    $ writeTextFile ASCII accountsFile bootstrapAccount ❺
  pure []
```

❶ Import `isLeft`.

❷ Import Encodings for `ASCII`.

❸ Add `writeTextFile` to import list.

❹ We don't care why it doesn't exist, just that it doesn't. That's why we're using `isLeft`.

❺ We WANT it to crash if we cannot write so there is no use of `try` here.

Make sure you add `node-buffer` to `spago.dhall` before moving on.

Now, we can read the file since we know it exists, because either we checked and it does exist, or we just created it.

Here's `readTextFile` again:

```
readTextFile :: Encoding -> FilePath -> Aff String ❶
```

❶ From `Node.FS.Aff`.

Now, read `accounts.csv` allowing the server to crash if we cannot read the file since we cannot startup without any accounts and we "know" the file exists. Hint: Take a look at a Function called `lines` in `Pursuit`. It will break up the file data from `readTextFile` into an `Array String` giving us one entry per line of the file.

To use `lines`, you'll need to add `stringutils` to your `spago.dhall`:

```
{-
Welcome to a Spago project!
You can edit this file as you like.
-}
{ name = "my-project"
, dependencies =
  [ "aff"
  , "arrays"
  , "console"
  , "effect"
  , "either"
  , "foreign-generic"
  , "foreign"
  , "httpure"
  , "maybe"
  , "node-buffer"
  , "node-fs-aff"
  , "node-process"
  , "posix-types"
  , "prelude"
  , "psci-support"
  , "stringutils" ❶
  , "uuid"
  , "transformers"
  ]
, packages = ./packages.dhall
, sources = [ "src/**/*.purs", "test/**/*.purs" ]
}
```

❶ Added `stringutils`.

Don't forget to run, ignoring the warnings:

```
npx spago build
```

Now, write the code to read the file and break it up into lines then return here.

We read the file then map `lines` over it:

```
import Data.String.Utils (lines) ❶
import Node.FS.Aff (exists, writeTextFile, readTextFile) ❷

loadAccounts :: Aff (Array Account)
loadAccounts = do
  exists <- try $ exists accountsFile
  when (isLeft exists)
    $ writeTextFile ASCII accountsFile bootstrapAccount
  -- COMPILER WARNING!! ❸
  accountLines <- lines <$> readTextFile ASCII accountsFile ❹
  pure []
```

❶ Import `lines`.

❷ Added `readTextFile` to the import list.

❸ We're not using `accountLinst` yet.

❹ Map `lines` over the `Aff String` we get back from `readTextFile`.

At this point, we should review our initial plan:

- Create initial Accounts File with a Bootstrap Account with a temporary password
- Load all Accounts from the File
- Append a new Account to the Accounts File

Looks like we're done with the first one and partially on the second.

Next, we'll need to parse each line into an Account. We're going to use a parser library found in the `purescript-parsing` package.

Add the `parsing` dependency to your `spago.dhall`. Then run:

```
npx spago build
```

You're going to get the following warning:

```
Warning found:
in module Text.Parsing.Parser
at .spago/parsing/v6.0.1/src/Text/Parsing/Parser.purs:117:1 - 117:64 (line
117, column 1 - line 117, column 64)

  A custom warning occurred while solving type class constraints:

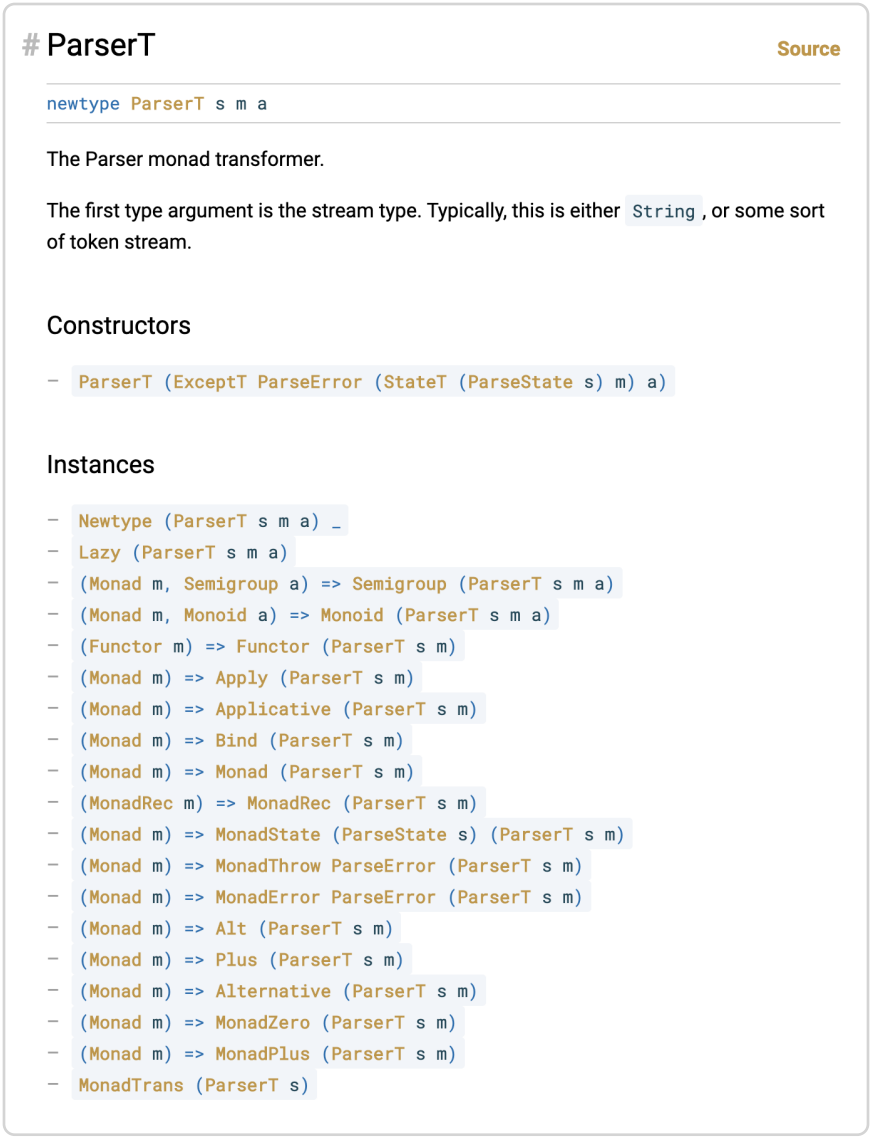
    'MonadZero' is deprecated, use 'Monad' and 'Alternative' constraints
instead


in value declaration monadZeroParserT

See
https://github.com/purescript/documentation/blob/master/errors/UserDefinedWarn
ing.md for more information,
or to contribute content related to this warning.
```

Turns out that in version 14 of the PureScript compiler and subsequent libraries, `MonadZero` has been marked as deprecated. The `purescript-parsing` library has until version 15 to remove such use but has yet to do so. This warning can be safely ignored.

Next, let's look at the Parser Monad from this library to better understand how we can use this:



ParserT Source

```
newtype ParserT s m a
```

The Parser monad transformer.

The first type argument is the stream type. Typically, this is either `String`, or some sort of token stream.

Constructors

- `ParserT (ExceptT ParseError (StateT (ParseState s) m) a)`

Instances

- `Newtype (ParserT s m a) _`
- `Lazy (ParserT s m a)`
- `(Monad m, Semigroup a) => Semigroup (ParserT s m a)`
- `(Monad m, Monoid a) => Monoid (ParserT s m a)`
- `(Functor m) => Functor (ParserT s m)`
- `(Monad m) => Apply (ParserT s m)`
- `(Monad m) => Applicative (ParserT s m)`
- `(Monad m) => Bind (ParserT s m)`
- `(Monad m) => Monad (ParserT s m)`
- `(MonadRec m) => MonadRec (ParserT s m)`
- `(Monad m) => MonadState (ParseState s) (ParserT s m)`
- `(Monad m) => MonadThrow ParseError (ParserT s m)`
- `(Monad m) => MonadError ParseError (ParserT s m)`
- `(Monad m) => Alt (ParserT s m)`
- `(Monad m) => Plus (ParserT s m)`
- `(Monad m) => Alternative (ParserT s m)`
- `(Monad m) => MonadZero (ParserT s m)`
- `(Monad m) => MonadPlus (ParserT s m)`
- `MonadTrans (ParserT s)`

Take a moment to study this. Notice how it's quite similar to our `Parser`.

Here's it's Type:

```
newtype ParserT s m a
  = ParserT (ExceptT ParseError (StateT (ParseState s) m) a)
```

The `s` is the Stream Type. In our case, it's just a `String`.

The `m` is the base Monad for the Monad Stack. We'll be using `Identity`.

`a` is the result of a successful `Parse`.

At this point, we should make a Type Alias for our Account Parser in `Handler.Account`:

```
import Data.Identity (Identity) ❶
import Text.Parsing.Parser (ParserT) ❷

type AccountParserT a = ParserT String Identity a
```

❶ Import `Identity`.

❷ Import `ParserT`.

Go ahead and add this to `Handler.Account`.

Next, we're going to write Parsers for each column of the CSV. We're going to start with `userName`. But we must first define what is a valid `userName`. Let's define it as starting with an `alpha` and then followed by zero or more `alphanumerics`.

With that out of the way, we're ready to code. We'll code this one together.

First, we'll write the Type Signature:

```
userName :: AccountParserT String
```

Next, we'll need to parse an `alpha` and after that as many `alphanumerics` that we can.

Look at `Pursuit` and the different modules in `purescript-parsing`. See if you find a module that looks like it may contain a parser we can use to accomplish this.

Did you find `Text.Parsing.Parser.String`? Look through that module and see if you can find a parser that we can use to parse `alpha` or `alphanumeric`.

When I did this, I found a familiar friend there:

```
satisfy
  :: ∀ s m
  .  StringLike s
  => Monad m
  => (Char -> Boolean)
  -> ParserT s m Char
```

We can use this along with `isAlpha` to get our first alpha:

```
import Data.CodePoint.Unicode (isAlpha) ❶
import Data.String.CodePoints (codePointFromChar)
import Text.Parsing.Parser.String (satisfy)

userName :: AccountParserT String
userName = do
  alpha <- satisfy (isAlpha <<< codePointFromChar) ❷
  ???
```

❶ Add these imports.

❷ `alpha's` Type is `Char`.

Next, we parse zero or more, i.e. `many`, alphaNumerics:

```
import Data.Array (many) ❶
import Data.CodePoint.Unicode (isAlpha, isAlphaNum) ❷

userName :: AccountParserT String
userName = do
  alpha <- satisfy (isAlpha <<< codePointFromChar)
  alphaNums <- many $ satisfy (isAlphaNum <<< codePointFromChar) ❸
  ???
```

❶ Import `many` from `Data.Array`.

❷ Add `isAlphaNum` to import list.

❸ `alphaNums's` Type is `Array Char`.

Now, we return `alpha` appended with `alphaNums`:

```
import Data.Array (many, (:)) ❶
import Data.String.CodeUnits (fromCharArray) ❷

userName :: AccountParserT String
userName = do
  alpha <- satisfy (isAlpha <<< codePointFromChar)
  alphaNums <- many $ satisfy (isAlphaNum <<< codePointFromChar)
  pure $ fromCharArray $ alpha : alphaNums ❸
```

❶ Added `(:)` to the import list.

❷ Import `fromCharArray`.

❸ We cons `alpha` onto `alphaNums`.

Now, on to `passwordHash`.

Take a look at `Node.Crypto.Hash` to see if you can find a way compute a `Hash` and then represent it as a `String`. When you do this, be prepared to justify why you chose what you did. Then read on to see my approach.

We'll for what we need, either `hex` or `base64` will work for us. But, we're going to use `hex` since it will easier to parser a Hex `String` compared to a Base-64 `String`. How does this match your choice?

So, now that we've done this analysis, We're ready to write a parser called `passwordHash`.

But first, to make that job easier, we'll need to write a parser to parse Hex `Strings`. It'll except `a` through `f`, inclusive, and `0` through `9`. Notice that I've made them lowercase. That's because `node` will render the `hex` in lowercase.

Code this parser calling it `hex` and return here when you've hit a stumbling block or have completed it.

Here's the code:

```
import Data.Array (many, some, (:)) ❶

hex :: AccountParserT String
hex = fromCharArray <$> (some $ satisfy isHex) ❷ ❸
  where isHex c = (c >= '0' && c <= '9') || (c >= 'a' && c <= 'f') ❹
```

❶ Added `some` to the import list.

❷ Mapping `fromCharArray` over the `Array Char` to get a `String`.

❸ We expect at least 1 Hex value, which is why we're using `some`.

❹ `isHex` is a Predicate that checks to make sure `c` is between `['0', '9']` or `['a', 'f']` inclusive.

Now write the `passwordHash` parser to use the `hex` parser before reading on.

Here the code:

```
passwordHash :: AccountParserT String
passwordHash = hex
```

Next, write a parser called `boolean` that will parse `"true"` or `"false"` and return a `Boolean`. Do this, then read on.

First, the Type Signature:

```
boolean :: AccountParserT Boolean
```

Next, we parse 1 or more characters:

```
import Text.Parsing.Parser.String (satisfy, anyChar) ❶

boolean :: AccountParserT Boolean
boolean = do
  bool <- fromCharArray <$> some anyChar ❷
```

❶ Added `anyChar` to import list.

❷ Get a `String` of any types of characters, 1 or more using `some`.

Now, we'll check `bool` for valid values:

```
boolean :: AccountParserT Boolean
boolean = do
  bool <- fromCharArray <$> some anyChar
  case bool of
    "true" -> pure true ❶
    "false" -> pure false ❶
    _ -> ???
```

❶ Check for valid `Boolean` as a `String`.

And now for the catch-all case, we'll `fail`:

```
import Text.Parsing.Parser (ParserT, fail) ❶

boolean :: AccountParserT Boolean
boolean = do
  bool <- fromCharArray <$> some anyChar
  case bool of
    "true" -> pure true
    "false" -> pure false
    _ -> fail "Invalid Boolean" ❷
```

❶ Added `fail` to the import list.

❷ Fail otherwise.

Now, write parsers `temporaryPassword` and `admin`, which are `Booleans`, then continue reading.

Here's their trivial implementations:

```
temporaryPassword :: AccountParserT Boolean
temporaryPassword = boolean

admin :: AccountParserT Boolean
admin = boolean
```

Next, write a parser called `properName` that takes an uppercase alpha and zero or more lowercase alphas. Then read on.

Here's the parser:

```
import Data.CodePoint.Unicode (isAlpha, isAlphaNum, isUpper, isLower) ❶

properName :: AccountParserT String
properName = do
  first <- satisfy (isUpper <<< codePointFromChar) ❷
  rest <- many $ satisfy (isLower <<< codePointFromChar) ❸
  pure $ fromCharArray (first : rest) ❹
```

❶ Added `isUpper` and `isLower` to the import list.

❷ Get first alpha as an uppercase.

❸ Get the rest as lowercase alpha, i.e. zero or more, which is why we're using `many`.

❹ Combine them and convert the `Array Char` to `String`.

Now, write `firstName` and `lastName` parsers.

Here are those parsers:

```
firstName :: AccountParserT String
firstName = properName

lastName :: AccountParserT String
lastName = properName
```

Before we go any further, I'm wondering if you're feeling like I am? We have an awful lot of parsing logic in this module. Seems worth separating.

Go ahead create a Folder under `src` called `Parser` and then create a file under `Parser` called `Account.purs` and factor out all the parsing Functions out into that file before reading further.

Here's that refactor:

```
module Parser.Account where ❶

import Prelude

import Data.Array (many, some, (:))
import Data.CodePoint.Unicode (isAlpha, isAlphaNum, isUpper, isLower)
import Data.Identity (Identity)
import Data.String.CodePoints (codePointFromChar)
import Data.String.CodeUnits (fromCharArray)
import Text.Parsing.Parser (ParserT, fail)
import Text.Parsing.Parser.String (satisfy, anyChar)

type AccountParserT a = ParserT String Identity a

userName :: AccountParserT String
userName = do
  alpha <- satisfy (isAlpha <<< codePointFromChar)
  alphaNums <- some $ satisfy (isAlphaNum <<< codePointFromChar)
  pure $ fromCharArray $ alpha : alphaNums

hex :: AccountParserT String
hex = fromCharArray <$> (some $ satisfy isHex)
  where isHex c = (c >= '0' && c <= '9') || (c >= 'a' && c <= 'f')

passwordHash :: AccountParserT String
passwordHash = hex

boolean :: AccountParserT Boolean
boolean = do
  bool <- fromCharArray <$> some anyChar
  case bool of
    "true" -> pure true
    "false" -> pure false
    _ -> fail "Invalid Boolean"

temporaryPassword :: AccountParserT Boolean
temporaryPassword = boolean

admin :: AccountParserT Boolean
admin = boolean

properName :: AccountParserT String
properName = do
  first <- satisfy (isUpper <<< codePointFromChar)
  rest <- many $ satisfy (isLower <<< codePointFromChar)
```

```
  pure $ fromCharArray (first : rest)

firstName :: AccountParserT String
firstName = properName

lastName :: AccountParserT String
lastName = properName
```

❶ Make sure the module name is correct.

Now, we can write a Function called `accountParser`, which will parse each element followed by a comma. Give it a try and see how far you get before reading on.

Here's our bootstrapAccount as reference:

```
-- userName,passwordHash,temporaryPassword,admin,firstName,lastName
bootstrapAccount :: String
bootstrapAccount = "admin,placeholder,true,true,Joe,Admin"
```

If you got stuck, here's the first part:

```
accountParser :: AccountParserT Account
accountParser = do
  userName' <- userName # comma ❶ ❷ ❸
  ???
```

❶ Here `comma` is going to be a parser, which we have yet to write, that will take a `AccountParserT` and combine it with a parser that parses a single comma. The comma, however, will NOT be returned from `comma`. Instead, it will return whatever the parser it was passed parses.

❷ Naming the result of the parse with a prime, i.e. `userName'` since the parser has the same name and we don't want to shadow it.

❸ Call `comma` at the end using `#` because the code will read in the same order as we parse, i.e. parse `userName` and then parse the `comma`.

Stop reading at this point and see how far you code before returning here.

Here's the rest of the individual columns:

```
accountParser :: AccountParserT Account
accountParser = do
  userName' <- userName # comma
  passwordHash' <- passwordHash # comma
  temporaryPassword' <- temporaryPassword # comma
  admin' <- admin # comma
  firstName' <- firstName # comma
  lastName' <- lastName ❶
  ???
```

❶ Notice there is NO `comma` call here. That because this is at the end of the line.

Now, add the code to return an `Account`. Consult the `Account` Data Constructor to see how to do this before reading any more.

Here's the construction of an `Account`:

```
import Entity.Account (Account(..)) ❶

accountParser :: AccountParserT Account
accountParser = do
  userName' <- userName # comma
  passwordHash' <- passwordHash # comma
  temporaryPassword' <- temporaryPassword # comma
  admin' <- admin # comma
  firstName' <- firstName # comma
  lastName' <- lastName
  pure $ Account ❷
    { userName: userName'
    , passwordHash: passwordHash'
    , temporaryPassword: temporaryPassword'
    , admin: admin'
    , firstName: firstName'
    , lastName: lastName'
    }
  where
    comma p = ???
```

❶ Include `Account` Data Constructor.

❷ Had to use the verbose syntax here since local names and field names collide with the parser names. We could rename the parser functions to shrink this, but probably six to one half dozen of the other.

Now, write the `comma` Parser Combinator, i.e. a Function that takes a parser and returns another parser.

Hint: It's going to take the parser it's given, i.e. `p`, and call that parser and then parse for a comma, returning the results of the original parser. Look in `Text.Parsing.Parser.String` for a parser that will parse a specific character.

Give it a try before moving on.

Let's write just the `comma` part one line at a time:

```
  where
    comma p = do
      v <- p
      ???
```

Here we capture the results of the parser, `p`, in `v`.

Now, we need to parse a specific character, i.e. a comma. We can use `char` from `Text.Parsing.Parser.String` for that:

```
  where
    comma p = do
      v <- p
      void $ char ',' ❶
      ???
```

❶ Single quotes are `Char` literals.

We throw away this second parse result.

And finally, we return the result of the first parser:

```
  where
    comma p = do
      v <- p
      void $ char ','
      pure v
```

Now, this works but is overkill. What we want to do is to apply the first. Do you remember `applyFirst's` Operator?

If not, look it up in `Pursuit` and rewrite our `do` block using `applyFirst's` Operator. Then read on.

Here the final result using the `applyFirst` Operator:

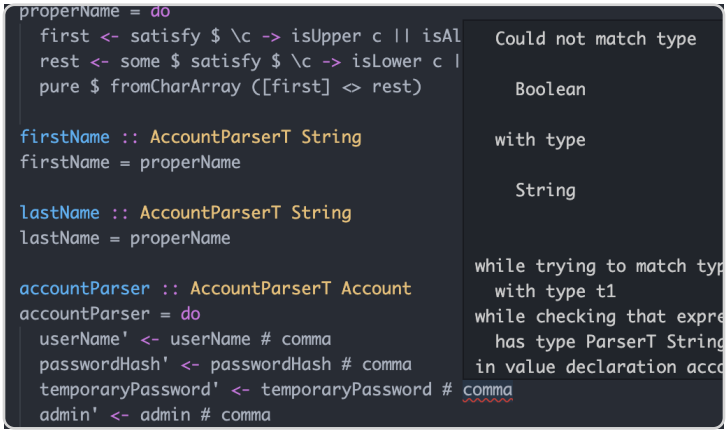
```
import Text.Parsing.Parser.String (satisfy, anyChar, char) ❶

accountParser :: AccountParserT Account
accountParser = do
  userName' <- userName # comma
  passwordHash' <- passwordHash # comma
  temporaryPassword' <- temporaryPassword # comma -- COMPILER ERROR!!
  admin' <- admin # comma
  firstName' <- firstName # comma
  lastName' <- lastName
  pure $ Account
    { userName: userName'
    , passwordHash: passwordHash'
    , temporaryPassword: temporaryPassword'
    , admin: admin'
    , firstName: firstName'
    , lastName: lastName'
    }
  where
    comma p = p <* char ',' ❷
```

❶ Added `char` to the import list.

❷ Using `applyFirst`, i.e. `<*`, to discard the results of the second computation.

Here's the compiler error:



Can you see my mistake?

It cannot match a `Boolean` with a `String` related to the call to `comma`. Does that help?

If not, then ask yourself what Type is `comma`. Can you write `comma's` Type Signature?

Let's think it through. `comma` takes a parser of Type `AccountParserT a`. The Polymorphic Type Parameter,

`a`, should be a huge clue to us. `comma` is a Polymorphic Function. We're going to need to add an explicit Type Signature otherwise the compiler is going to overspecialize its Type.

This overspecialization is fine if all of our parsers parsed `Strings`, but they don't. `temporaryPassword` parses a `Boolean`. And these are the 2 Types that the compiler is complaining about.

Fix my mistake by adding a Type Signature for `comma` then return here.

Here's the final code:

```
accountParser :: AccountParserT Account
accountParser = do
  userName' <- userName # comma
  passwordHash' <- passwordHash # comma
  temporaryPassword' <- temporaryPassword # comma
  admin' <- admin # comma
  firstName' <- firstName # comma
  lastName' <- lastName
  pure $ Account
    { userName: userName'
    , passwordHash: passwordHash'
    , temporaryPassword: temporaryPassword'
    , admin: admin'
    , firstName: firstName'
    , lastName: lastName'
    }
  where
    comma :: ∀ a. AccountParserT a -> AccountParserT a ❶
    comma p = p <* char ','
```

❶ `comma` is Polymorphic in `a` and therefore requires an explicit Type Signature to prevent the compiler from overspecializing.

We now have a parser that will parse a single line of our CSV file.

Having done that, we can return our attention back to `loadAccounts` in `Handler.Account`. Here's where we left off:

```
loadAccounts :: Aff (Array Account)
loadAccounts = do
  exists <- try $ exists accountsFile
  when (isLeft exists)
    $ writeTextFile ASCII accountsFile bootstrapAccount
  -- COMPILER WARNING!!
  accountLines <- lines <$> readTextFile ASCII accountsFile ❶
  pure []
```

❶ `accountLines` is the lines from the Accounts File.

We can apply our `accountParser` to each line of the file, i.e. to each entry of `accountLines`.

To do this, I'm going to suggest that you incubate this code in a separate line of code as follows:

```
loadAccounts :: Aff (Array Account)
loadAccounts = do
  exists <- try $ exists accountsFile
  when (isLeft exists)
    $ writeTextFile ASCII accountsFile bootstrapAccount
  accountLines <- lines <$> readTextFile ASCII accountsFile
  let x = accountLines :: ?x ❶ ❷
  pure []
```

❶ Here is our incubation line. Once the Type Hole gives us the Type we want, we'll integrate it into our Function.

❷ The Type Hole is `Array String`.

Notice the Type Hole is there to help us as we slowly build up the code for this line.

Now, we want to parse each line with `accountParser`. But how do we run our parser?

Go search `Pursuit` for `runParserT` and see what you get.

Here's what I get:

runParserT

```
runParserT :: forall m s a. Monad m => s -> ParserT s m a -> m (Either
    ParseError a)
```

Apply a parser, keeping only the parsed result.

P purescript-parsing M Text.Parsing.Parser

We can see that `runParserT` takes a `String` to parse and a `ParserT`. We have both of those things. We have lines from our Accounts File and we have `accountParser`.

Go ahead and add to our incubation line the code to run `accountParser` over each line in the `Array`, i.e. `accountLines`. Then continue reading.

We're going to map `runParserT` over `accountLines` to accomplish this:

```
import Parser.Account (accountParser) ❶
import Text.Parsing.Parser (runParserT)

loadAccounts :: Aff (Array Account)
loadAccounts = do
  exists <- try $ exists accountsFile
  when (isLeft exists)
    $ writeTextFile ASCII accountsFile bootstrapAccount
  accountLines <- lines <$> readTextFile ASCII accountsFile
  let x = flip runParserT accountParser <$> accountLines :: ?x ❷ ❸ ❹
  pure []
```

❶ Add these imports.

❷ We flipped `runParserT` to give it the parser Parameter before the `String` to parse.

❸ We map over each line in the `Array`.

❹ The Type Hole is `Array (Identity (Either ParseError Account))`.

We now have an `Array` of `Identity`, so go ahead and add code to `unwrap` the elements before reading on.

We're going to simply compose `unwrap` with `runParserT`. We can do this instead of calling `map` again thanks to the Functor Laws:

```
import Data.Newtype (unwrap) ❶

loadAccounts :: Aff (Array Account)
loadAccounts = do
  exists <- try $ exists accountsFile
  when (isLeft exists)
    $ writeTextFile ASCII accountsFile bootstrapAccount
  accountLines <- lines <$> readTextFile ASCII accountsFile
  let x = unwrap
        <<< flip runParserT accountParser <$> accountLines :: ?x ❷ ❸
  pure []
```

❶ Import `unwrap`.

❷ Compose `unwrap` with `flip runParserT accountParser` to eliminate `Identity`. Remember that `Identity` has a `Newtype` Instance, which is why we can call `unwrap`.

❸ The Type Hole is `Array (Either ParseError Account)`.

Now, we have an `Array` on the outside and an `Either` on the inside. But we'd actually like them the opposite way. Do you remember how to swap them?

Do you remember a Function that takes a `Traversable` of Type `t (m a)` and returns an `m (t a)`? If not, go look up `t (m a) -> m (t a)` on `Pursuit` and see what you find.

When I do this, I find `sequence`. So go ahead and add a call to `sequence` on the `Array` of `Eithers` and see what the Type Hole gives you. Then return back here.

Here's the change:

```
import Data.Traversable (sequence) ❶

loadAccounts :: Aff (Array Account)
loadAccounts = do
  exists <- try $ exists accountsFile
  when (isLeft exists)
    $ writeTextFile ASCII accountsFile bootstrapAccount
  accountLines <- lines <$> readTextFile ASCII accountsFile
  let x = sequence $ unwrap
        <<< flip runParserT accountParser <$> accountLines :: ?x ❷
  pure []
```

❶ Import `sequence`.

❷ The Type Hole is `Either ParseError (Array Account)`.

This line of code that we're incubating here has a Type that's almost what we're looking to return from this Function.

But the fact that we have an `Either` should clue us into the fact that I was myopic when I coded our Type Signature for `loadAccounts`. I forgot the fact that this could fail.

Fix my mistake by changing the Type Signature to reflect that we can fail with `ParseError`. Don't forget to also change the return Value so it matches our new return Type. Do this before reading any further.

Here's the Type Signature Change:

```
import Data.Either (Either, isLeft) ❶
import Text.Parsing.Parser (ParseError, runParserT) ❷

loadAccounts :: Aff (Either ParseError (Array Account)) ❸
loadAccounts = do
  exists <- try $ exists accountsFile
  when (isLeft exists)
    $ writeTextFile ASCII accountsFile bootstrapAccount
  accountLines <- lines <$> readTextFile ASCII accountsFile
  let x = sequence $ unwrap
        <<< flip runParserT accountParser <$> accountLines :: ?x
  pure $ pure [] ❹
```

❶ Added `Either` to import list.

❷ Added `ParseError` to import list.

❸ Changed Type Signature to reflect that we can fail with `ParseError`.

❹ Had to wrap our empty `Array` in a `Right` to match the new Type Signature. This line of code is only here to make the compiler happy for now.

Now looking at the Type Hole again, `Either ParseError (Array Account)`, we're just one step away from the Type that this Function now returns, i.e. `Aff (Either ParseError (Array Account))`.

Take our incubation line and replace the last line with it. Don't forget that it needs to still be wrapped in an `Aff`.

Here's that final change:

```
loadAccounts :: Aff (Either ParseError (Array Account))
loadAccounts = do
  exists <- try $ exists accountsFile
  when (isLeft exists)
    $ writeTextFile ASCII accountsFile bootstrapAccount
  accountLines <- lines <$> readTextFile ASCII accountsFile
  pure $ sequence ❶
    $ unwrap <<< flip runParserT accountParser <$> accountLines ❷
```

❶ Added `pure` to wrap this in an `Aff`.

❷ Replaced the last line with the incubation line minus the `let x=` and `:: ?x` parts.

Now that we've completed `loadAccounts`, let's look back at the list of things the Account Handler is supposed to do:

- Create initial Accounts File with a Bootstrap Account with a temporary password

- Load all Accounts from the File
- Append a new Account to the Accounts File

We've done the first 2 things on the list. All that's left is to append a new Account to the Accounts File.

What are our Technological Hurdles for this operation? Think about the operations that need to be done and then think about what you know how to do and what you don't.

Make a mental list of the hurdles you'll have to overcome before reading the next section.

28.6.2. Coding `createAccount`

The first thing I think about is the append operation. How do we append to the end of an existing file? That's Technological Hurdler number one.

Next, we need a way to take an `Account` and produce a CSV (Comma Separated Values) file line. That's not so much of a Technological Hurdle as it is a Function we'll need to write.

We have the following things to do:

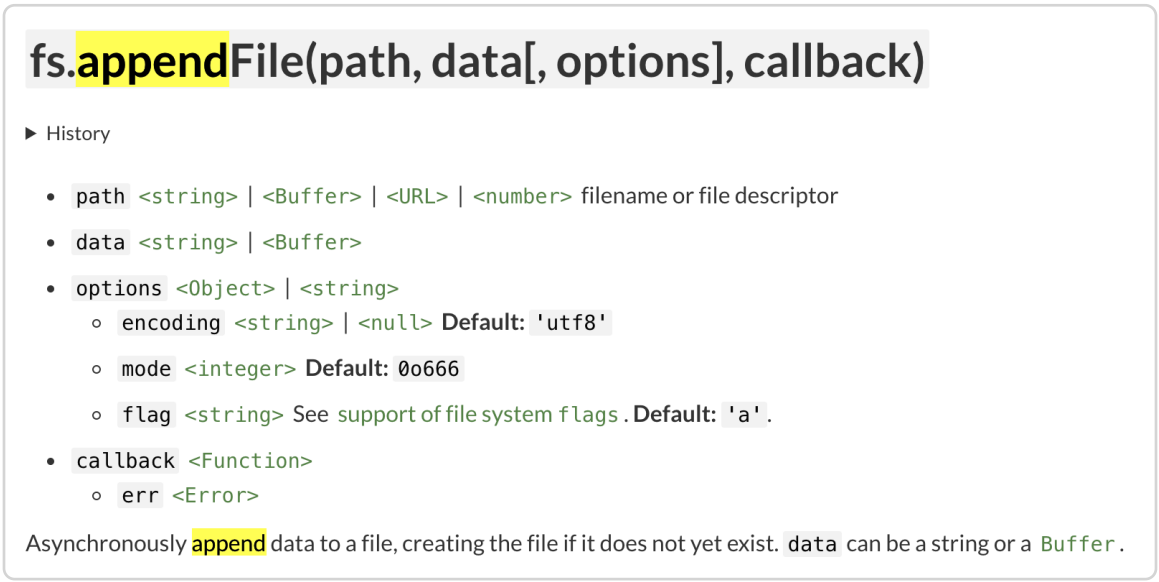
- Figure out how to append to an existing file
- Take an `Account` and produce a `String` that's a single line of the CSV Accounts File

Let's start with appending to an existing file.

Since we're running under Node, let's go look at how we do this in Node first. Search the Node API via Google with a search phrase like `Node API append file` or `Node append` or something like that.

Once you find out the name of the Function, then head on over to `Pursuit` and look up the same name and see if you can find anything in a `Node.FS.*` module. Then return back here.

Here's what I found:



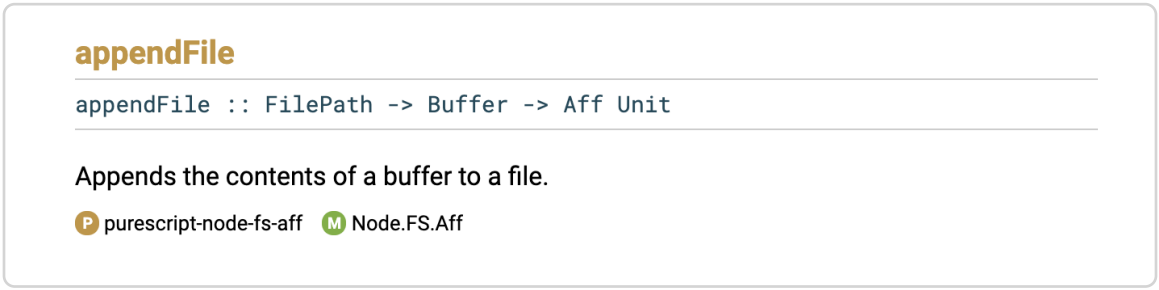
fs.appendFile(path, data[, options], callback)

▶ History

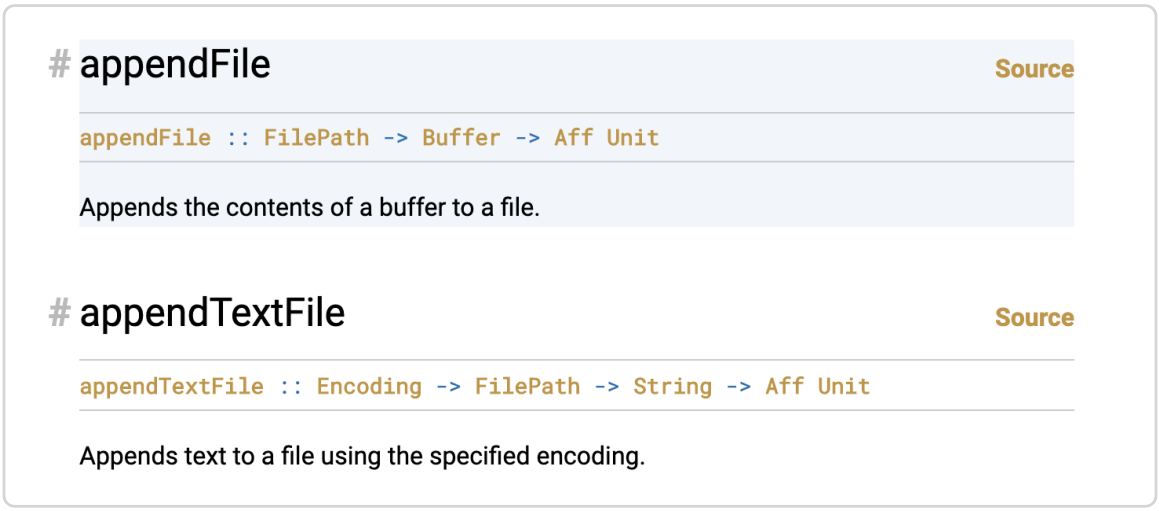
- `path` `<string>` | `<Buffer>` | `<URL>` | `<number>` filename or file descriptor
- `data` `<string>` | `<Buffer>`
- `options` `<Object>` | `<string>`
  - `encoding` `<string>` | `<null>` **Default:** `'utf8'`
  - `mode` `<integer>` **Default:** `0o666`
  - `flag` `<string>` See support of file system `flags`. **Default:** `'a'`.
- `callback` `<Function>`
  - `err` `<Error>`

Asynchronously append data to a file, creating the file if it does not yet exist. `data` can be a string or a `Buffer`.

And when I search `Pursuit` for `appendFile`, I find 3 different version, but I'm interested in the following one from `Node.FS.Aff`:



Clicking on this link takes me to:



Now, `appendFile` wants a `Buffer` and if you try to figure out how to take a `String` and turn it into a `Buffer`, you'll wind up down a rabbit hole. But, if we look right under `appendFile` we find `appendTextFile`:

```
appendTextFile :: Encoding -> FilePath -> String -> Aff Unit
```

This is exactly what we need.

Now, we've overcome our Technological Hurdle and we're ready to write a Function to create an Account. We're going to call it `createAccount` and it'll be put in the `Handler.Account` module.

Let's plan to delegate the process of turning an `Account` into a CSV `String` to another Function that we're going to call `accountToCSV`. We'll write this Function afterwards but for now, let's write this as a stand-in:

```
accountToCSV :: Account -> String
accountToCSV = const ""
```

Add this stand-in and then write `createAccount` in `Handler.Account` and when you're done or run into

difficulties, return here and continue reading.

Here's `createAccount's` Type Signature:

```
createAccount :: Account -> Aff (Either String Unit) ❶
```

❶ Runs in `Aff` since `appendTextFile` does.

We take an `Account` and return nothing on the success case and a File I/O Error on the error case, i.e. a `String`. This is a tell-tale sign that our Function has a Side-effect and it does. It's Side-effect is to write to a file.

One thing we're going to need to do is change `CreateUserFailureReason` to include this File I/O error. This way we can report back to the client why we're unable to write to the Accounts File.

Let's make the following change to `CreateUserFailureReason`:

```
data CreateUserFailureReason
  = NotAuthorized
  | NotAuthenticated
  | AlreadyExists
  | FileIOError String ❶
```

❶ File I/O error with a reason string.

One very important thing to point out is that we're returning a `String` and not a `CreateUserFailureReason` from this Function. This is a classical case of Separation of Concerns.

The Account Handler should NOT know anything about the API or the API Handler. The API Handler, however, MUST know about the Account Manager since it calls `createAccount`.

So, the API Handler is indirectly coupled to the Account Handler but the Account Handler isn't coupled to anything in our design and we'd like to keep it that way.

The `String` that is returned from `createAccount` will be converted into `CreateUserFailureReason` by the API Handler using our new Data Constructor, `FileIOError`.

Okay, now we're ready to continue with `createAccount`:

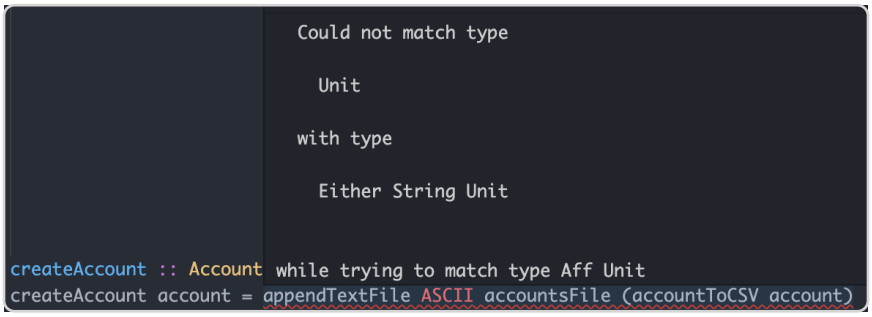
```
import Node.FS.Aff (exists, writeTextFile
                  , readTextFile, appendTextFile) ❶

createAccount :: Account -> Aff (Either String Unit)
-- COMPILER ERROR!!
createAccount account
  = appendTextFile ASCII accountsFile $ accountToCSV account ❷
```

❶ Added `appendTextFile` to import list.

❷ Append to CSV.

The compiler error is expected since we're not done, but it's also helpful:



So, `appendTextFile` is returning an `Aff Unit`, but `createAccount` is supposed to return `Aff (Either String Unit)`.

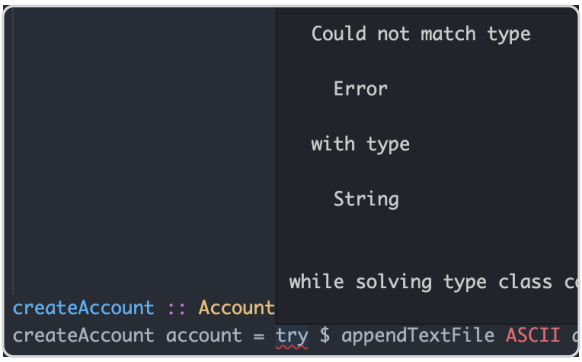
To fix that, we'll add `try` to catch the exception and turn it into an `Either`:

```
try :: ∀ e m a. MonadError e m => m a -> m (Either e a)
```

And here's that change:

```
createAccount :: Account -> Aff (Either String Unit)
-- COMPILER ERROR!!
createAccount account
  = try $ appendTextFile ASCII accountsFile $ accountToCSV account
```

And the new compiler error:



What do you think the compiler is complaining about now? If you don't know, then add a Type Hole in such a way that we can tell what Type we've written.

Then return here when you're done or you're stuck.

Here is the change:

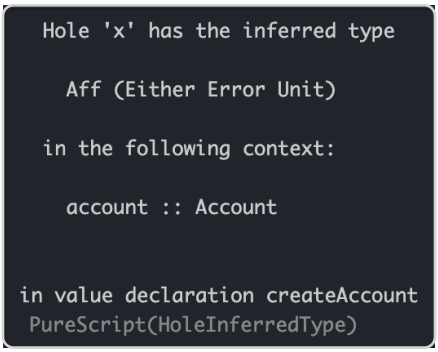
```
createAccount :: Account -> Aff (Either String Unit)
createAccount account =
  let x = try $ appendTextFile ASCII accountsFile
        $ accountToCSV account :: ?x ❶
  in pure $ pure unit ❷ ❸
```

❶ Set a dummy Variable `x` to our code with a Type Hole so we can see what Type this is.

❷ Note the `in` keyword. That's because we're not in a `do` block.

❸ `pure $ pure unit` is a dummy return Value that's of the correct Type.

The Type Hole is:



From this it should be clear what the compiler was complaining about earlier. We've written something of Type `Aff (Either Error Unit)` but `createAccount` is supposed to return `Aff (Either String Unit)`.

This is why we got a Type mismatch between `Error` and `String`. So, how do we convert an `Error` to a `String`?

We could look it up, but I'm not exactly sure of which `Error` this is. Probably the one in `Effect.Exception` and if I'm right, it has a `Show` Instance, which will turn it into a `String`.

But, honestly, it's actually faster to just `lmap show` over the `Aff` and if the compiler doesn't complain, then we're good:

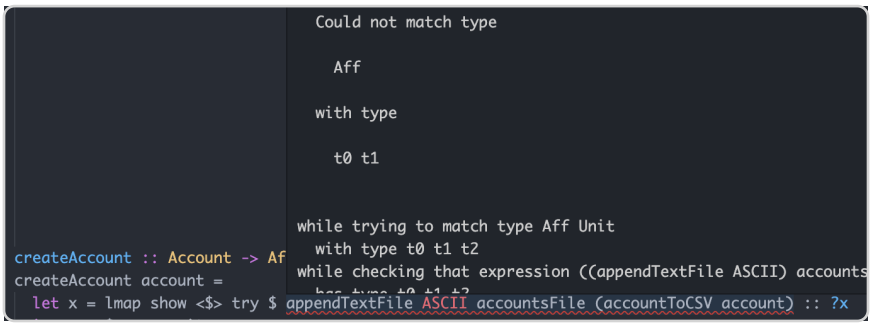
```
import Data.Bifunctor (lmap) ❶

createAccount :: Account -> Aff (Either String Unit)
createAccount account =
  -- COMPILER ERROR!!
  let x = lmap show <$> try $ appendTextFile ASCII
        accountsFile $ accountToCSV account :: ?x ❷
  in pure $ pure unit
```

❶ Import `lmap`.

❷ The map via `<$>` maps into the `Aff`, which contains an `Either`. Then the `lmap show` will map over the left side of the `Either`.

The compiler error:



When I see code with a lot of Binary Operators and an incomprehensible error like this, I always start adding Parentheses to see if I have a Precedence problem:

```
createAccount :: Account -> Aff (Either String Unit)
createAccount account =
  let x = lmap show <$> (try $ appendTextFile ASCII
            accountsFile $ accountToCSV account) :: ?x ❶
  in pure $ pure unit
```

❶ Added Parentheses left of `<$>`.

And now with the compiler error gone, we get the following Type Hole message:

```
Hole 'x' has the inferred type

  Aff (Either String Unit)

in the following context:

  account :: Account
```

And since this is exactly the Type we're looking for, we can just use this code and eliminate all the other cruft:

```
createAccount :: Account -> Aff (Either String Unit)
createAccount account = lmap show
  <$> (try $ appendTextFile ASCII accountsFile $ accountToCSV account)
```

Now, we're ready to write the real code for `accountToCSV`.

Here's a reminder of the order of the fields:

- `userName` - A unique identifier for logging on to the system.
- `passwordHash` - The hash that's calculated from the password.
- `temporaryPassword` - `true` if the current password MUST be changed upon the next logon.
- `admin` - `true` if Administrative Account.
- `firstName` - User's first name.
- `lastName` - User's last name.

Before you look at my code for it, code it yourself then read on.

We'll already have the Type Signature:

```
accountToCSV :: Account -> String
```

We turn an `Account` into a line for the CSV File, which will be a `String`.

Next the left-hand side of equal sign:

```
import Entity.Account (Account(..)) ❶

accountToCSV :: Account -> String
accountToCSV (Account  ❷ ❸
                { userName
                , passwordHash
                , temporaryPassword
                , admin
                , firstName
                , lastName }) = ???
```

❶ Added `(..)` to get the `Account` Data Constructor for Pattern Matching.

❷ Pattern Match to the Data Constructor and then destructure the contained Record.

❸ Notice the Parentheses. This are required when Pattern Matching. We're pretty used to this fact by now, but it can be easy to miss when your code is on multiple lines.

Next, I want to introduce you to a very useful Function that you probably have seen in other languages under the name `join`. It's called `intercalate` and its Dictionary definition is:

```
insert (something) between layers in a crystal lattice, geological formation,
or other structure.
```

The thing we're going to want to insert between things is a comma.

Here's it's Type Signature from `Data.Foldable`:

```
intercalate :: ∀ f m. Foldable f => Monoid m => m -> f m -> m ❶
```

❶ Here `m` stands for `Monoid` NOT `Monad`.

If we have a `Foldable` of `Monoids` this will append them together by placing the first Parameter, which is the same `Monoid`, between each element of the `Foldable`.

For example:

```
intercalate "," [ "a", "b", "c"] == "a,b,c"
```

Now, based on this information and the code we've written so far, please use `intercalate` to finish coding this. If you did finish this Function but didn't use `intercalate`, comment out your code and rewrite it using `intercalate`.

Once, you're done, keep reading.

We're going to want to take each element of the Account and turn it into a `String`, if it's not already, and then `intercalate` a comma between them:

```
import Data.Foldable (intercalate) ❶

accountToCSV :: Account -> String
accountToCSV (Account
                { userName
                , passwordHash
                , temporaryPassword
                , admin
                , firstName
                , lastName }) =
  intercalate ","
    [ userName
    , passwordHash
    , show temporaryPassword ❷
    , show admin ❷
    , firstName
    , lastName
    ]
```

❶ Import `intercalate`.

❷ Have to show these since they're `Booleans`.

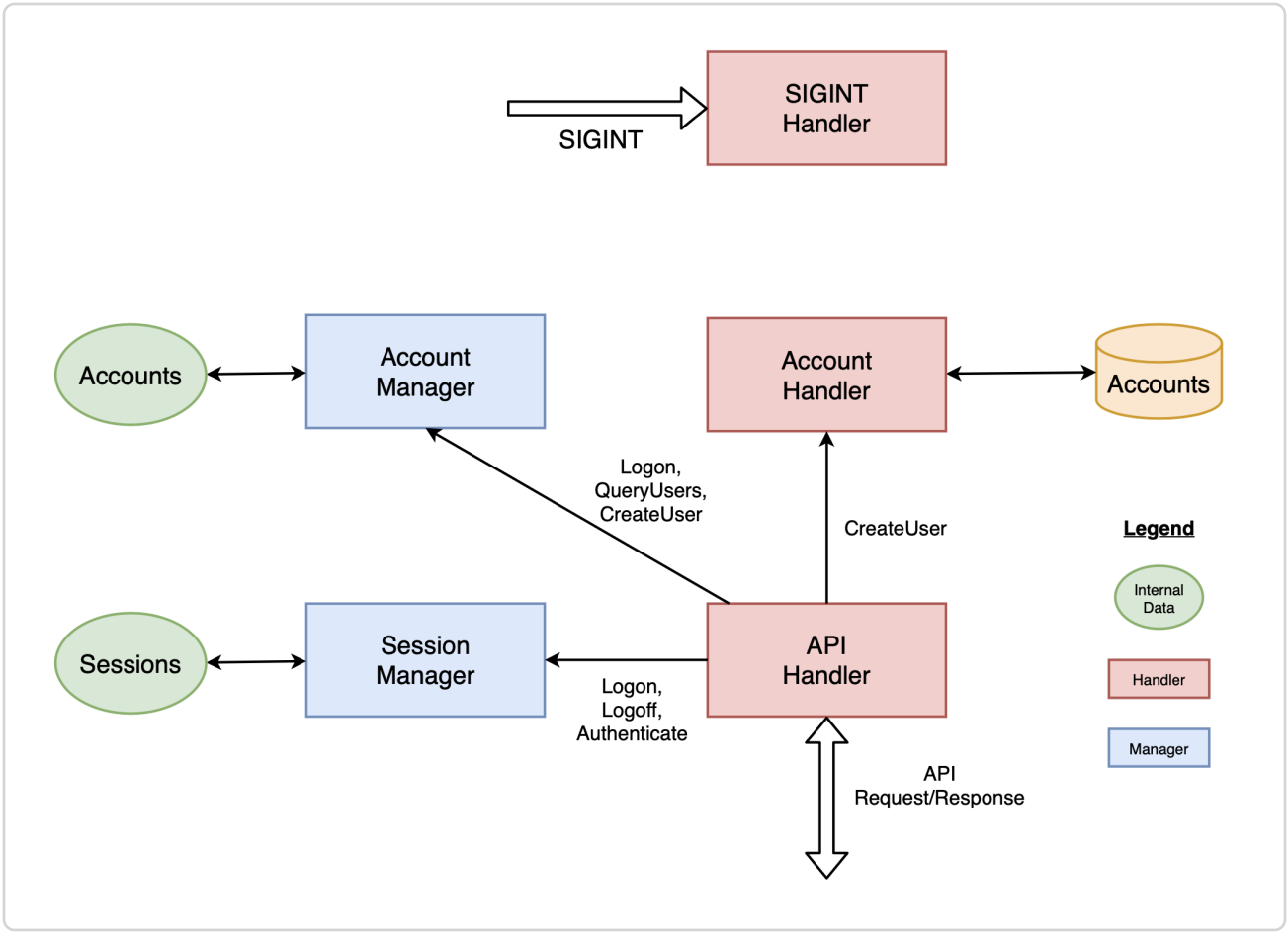
And the Account Handler is now complete. Well, for now. Who knows if we missed something or will want to augment something later. But for now, we're done.

Remember that we couldn't continue building the API until the Account Manager and Session Manager were done. But before we could embark on coding the Account Manager, we needed accounts, which is why we worked on the Account Handler next.

Now that the Account Handler is done, we can work on the Account Manager.

28.7. Coding the Account Manager

Let's return to our design diagram:



From this we see that the API Handler will call the Account Manager for:

- Logon Authentication
- Querying Users
- Create User

First thing to do is create a Folder under `src` called `Manager` and in that directory create a File called `Account.purs`. This is where we will be coding.

Don't forget to qualify your module name properly, i.e. it follows the path to the file from `src`:

```
module Manager.Account where
```

We'd like to start by writing a Function that will verify the logon credentials for the Logon Authentication. But before we can do that, we've got to write some code to compute the password hash.

28.7.1. Code for Password Hash

Let's see what the API has at its disposal during a logon request:

```
newtype LogonRequest = LogonRequest
  { userName :: String
  , password :: String
  }
```

So, we have a `userName` and a plain-text `password`. We're going to have these 2 items to produce a `passwordHash`. This is what's kept in the Accounts File and, hence, the `Account` Type:

```
newtype Account = Account (Record (UserRow ( passwordHash :: String )))
```

We'll have to first decide how we're going to take plain-text passwords and hash them. We're going to use a simple Salted Hash.

A Salted Hash helps protect Passwords from Dictionary Attacks by introducing into the plain-text password some random value. There are a few ways to do this. Two of which come to mind.

One way is to simply generate a random string, i.e. a GUID, and then append that to the password. But this means we'll be required to store that GUID in the `Account`, which, in our case, will require us to change our `Account` definition and the Accounts File.

Another way is to use the `userName` to generate a random String that we append to the password before we create the Hash.

Since, this Chapter is already quite large, we're going to take the latter approach since it'll require less work plus we'll learn more from it.

First, we'll need a Function that takes a `userName` and produces a Random `String`.

Second, we'll need a Function that takes a `password` and a `salt` and produces a `Hash`.

If you search on `Pursuit` for `random`, you'll find a package called `Random.PseudoRandom`. We cannot use this. It's too old and has fallen out of favor.

We have to use the `purescript-quickcheck` package. This package is for doing tests and it has to generate random values to be able to randomly do unit tests.

It relies on a package that implements a Linear Congruent Generator, which is the oldest and best-known pseudo-random number generator algorithm. There is one module in this package called `Random.LCG`.

Here's what we'll need:

```
import Random.LCG (Seed, mkSeed)
import Test.QuickCheck.Gen (sample)
import Test.QuickCheck.Arbitrary (arbitrary)
```

`Seed` is the Type of a Seed for the Generator.

`sample`, given a Seed, a count and a Generator, will produce a random `Array` of Values:

```
sample :: ∀ a. Seed -> Size -> Gen a -> Array a
```

`arbitrary` is part of the `Arbitrary` Typeclass. Any Type that can be generated, will have an instance of this class:

```
class Arbitrary t where
  arbitrary :: Gen t
```

It turns out that `String` has an `Arbitrary` Instance, but we want to control the length of the `String`. So instead, we'll use the `Arbitrary` Instance from `Char` along with `sample` to produce a `String` of a specified `Size`.

We can make a `Seed` using:

```
mkSeed :: Int -> Seed
```

If we give this Function the same `Int` it'll give us the same `Seed`, it's a Pure Function after all. And if we give the random generator the same `Seed`, it too will always produce the same result. That's because it too is Pure. And this predictability is paramount when hashing passwords.

Now all we have to do is figure out how we're going to convert our `userName` into an `Int`. How do you think we could do this?

One way is to use the `String` length. The problem with that is that most `userNames` are going to be pretty close to the same length and therefore not very random. In fact, all of the `userNames` with the same length will have the same seed and hence the same Salt.

Another way is to convert each character of the `userName` into an `Int` and then combine them somehow via Addition or Multiplication. This should work just fine for our needs.

One thing to note about deriving the Salt for password hashing from the `userName` is that changes to the `userName` will require the user to provide their `password`. This is so that the password can be rehashed with the new Salt since the Salt is derived from the `userName`. This will NOT be an issue for us since we're not supporting `userName` changes, but it's still good to note.

Before we move further, your're going to need to add a package `purescript-quickcheck` to your `spago.dhall`. When you rebuild the server, you'll get a warning:

```
[warn] None of your project files import modules from some projects that are
in the direct dependencies of your project.
These dependencies are unused. To fix this warning, remove the following
packages from the list of dependencies in your config:
- quickcheck
```

We an ignore that, but we shouldn't ignore the following error:

```
[error] Some of your project files import modules from packages that are not
in the direct dependencies of your project.
To fix this error add the following packages to the list of dependencies in
your config:
- bifunctors
- foldable-traversable
- identity
- newtype
- strings
- unicode
```

We're going to have to add these eventually. Let's not wait:

```
{-
Welcome to a Spago project!
You can edit this file as you like.
-}
{ name = "my-project"
, dependencies =
  [ "aff"
  , "arrays"
  , "bifunctors" ❷
  , "console"
  , "effect"
  , "either"
  , "foldable-traversable"❷
  , "foreign-generic"
  , "foreign"
  , "httpure"
  , "identity" ❷
  , "maybe"
  , "newtype" ❷
  , "node-buffer"
  , "node-fs-aff"
  , "node-process"
  , "parsing"
  , "posix-types"
  , "prelude"
  , "psci-support"
  , "quickcheck" ❶
  , "strings" ❷
  , "stringutils"
  , "transformers"
  , "unicode" ❷
  , "uuid"
  ]
, packages = ./packages.dhall
, sources = [ "src/**/*.purs", "test/**/*.purs" ]
}
```

❶ Added `quickcheck`.

❷ Added the packages from the error message.

Now rebuild should only produce the warning.

Next, we'll write a Function called `userNameSeed`:

```
import Random.LCG (Seed) ❶

userNameSeed :: String -> Seed
```

❶ Before you import this, add package `purescript-quickcheck` to your `spago.dhall` then build the server.

We'll convert each character into a number and then multiply them together to get an `Int`. Then we'll call `mkSeed` with that `Int` to get our `Seed`.

Finish the code for this Function before you continue reading. Hint: You're going to want to work with `Arrays` here. So convert the `String` into an `Array` and then use `maps` and `folds` to accomplish your task.

Once last hint is to look up `toCharCode` on `Pursuit`.

Once you're done or stuck, read on.

We'll start with the Type Signature, which was given:

```
userNameSeed :: String -> Seed
```

Next, we'll write the left-hand side of the equal sign:

```
userNameSeed :: String -> Seed
userNameSeed userName =
  ???
```

Now, we want to start by converting the `userName` into an `Array` of `Char`:

```
import Prelude ❶

import Data.String.CodeUnits (toCharArray) ❷

userNameSeed :: String -> Seed
userNameSeed userName =
  userName
    # toCharArray ❸
    ???
```

❶ Import `Prelude` to get `#`.

❷ Import `toCharArray`.

❸ Convert `userName` from a `String` to `Array Char`.

And now we want to convert each `Char` to its `Int` counterpart. Since all Unicode Characters are just Integers, we're going to use `toCharCode` to convert them:

```
import Data.Char (toCharCode) ❶

userNameSeed :: String -> Seed
userNameSeed userName =
  userName
    # toCharArray
    <#> toCharCode ❷
    ???
```

❶ Import `toCharCode`.

❷ Notice that we're mapping here, which means the result is an `Array Int`.

And now, we want to multiple all of the `Ints` with each other:

```
import Data.Foldable (foldl) ❶

userNameSeed :: String -> Seed
userNameSeed userName =
  userName
    # toCharArray
    <#> toCharCode
    # foldl (\prod n -> prod * n) 1 ❷ ❸
    ???
```

❶ Import `foldl`.

❷ Use Multiplicative Identity as our initial Value, i.e. `1`.

❸ `foldl's` leftmost Parameter in the Lambda is the Accumulator, i.e. `prod`, which is the running product.

Can you see how we can greatly simply our `foldl`?

We can write it Point-free:

```
userNameSeed :: String -> Seed
userNameSeed userName =
  userName
    # toCharArray
    <#> toCharCode
    # foldl (*) 1 ❶
    ???
```

❶ Written Point-free.

If this isn't obvious then here's the steps I skipped:

```
\prod n -> prod * n          (Given)
\prod n -> (*) prod n        (Infixed Notation)
\prod -> (*) prod            (ETA Reduce)
(*)                          (ETA Reduce)
```

And finally, we take our `Int`, i.e. the result of the multiplication, and make a `Seed` from it using `mkSeed`:

```
import Random.LCG (Seed, mkSeed) ❶

userNameSeed :: String -> Seed
userNameSeed userName =
  userName
    # toCharArray
    <#> toCharCode
    # foldl (*) 1
    # mkSeed ❷
```

❶ Added `mkSeed` to the import list.

❷ Turn our `Int` into a `Seed`.

And now, write a Function called `userNameSalt` that will first take a length, then a `userName` and generate a random `String`.

Use `userNameSeed` to generate a `Seed` and pass that to `sample` along with the supplied length to produce a random `Array` of `Char`. Then convert the `Array Char` to `String`.

Here are the imports you'll need:

```
import Test.QuickCheck.Arbitrary (arbitrary) ❶
import Test.QuickCheck.Gen (sample)
```

❶ This is your Generator, i.e. `Gen a`, for `sample` (see `sample's` Type Signature).

Use `Pursuit` to look up any Type Signatures that may you need. Once you've coded `userNameSalt` return here and continue reading.

Here's the Type Signature:

```
userNameSalt :: Int -> String -> String
```

It'll take a length parameter and the `userName`.

Next, we'll code the left-hand side of the implementation:

```
userNameSalt :: Int -> String -> String
userNameSalt saltLength userName =
  ???
```

We know that `sample` will give us an `Array` of random Values, so let's remind ourselves of its Type Signature:

```
sample :: ∀ a. Seed -> Size -> Gen a -> Array a
```

We need a seed. We can get that from `userNameSeed`.

Next, we need a `Size`. We can get this from `saltLength`.

Next, we need a generator. We can get this by using `arbitrary`. However, the compiler won't know which `arbitrary` to use until we use `fromCharArray` on the result of `sample`. Since `fromCharArray` converts a `Array Char` to `String`, the compiler can infer that `arbitrary` can only be the one from `Char`.

Knowing all of this, if you didn't finish coding, stop now and give it a try. Then move on.

Here's the code that implements all of the steps mentioned above:

```
import Data.String.CodeUnits (toCharArray, fromCharArray) ❶

userNameSalt :: Int -> String -> String
userNameSalt saltLength userName =
  fromCharArray
    $ sample (userNameSeed userName) saltLength arbitrary ❷
```

❶ Added `fromCharArray` to import list.

❷ The compiler knows that the `Array a` that `sample` returns must unify with `Array Char` since that's what `fromCharArray` wants. This means that it's going to use `Char's` version of `arbitrary`, since `a` unifies with `Char`.

Now, we can write a Function that we'll call `passwordHash`. But before we do, we'll need to look up `Node.Crypto.Hash` to search for a Function that will take a `String` and create a hash as a `String`.

Do this now and see what you find.

I found `hex` and `base64`. If you remember, we found these Functions before and decided that we'd use `hex`. That's why we wrote a `hex` Parser.

Here's hex's definition:

```
hex :: Algorithm -> String -> Effect String
```

Why is `hex` an `Effect`? I suspect, it has to do with how it's implemented in Node possibly using the Operating System to get its random value. A little bit of Googling reveals that Node uses OpenSSL and even further Googling reveals OpenSSL seeds the random number generator using a system-specific entropy source, e.g. in Linux it reads from `/dev/urandom`.

Since this is equivalent to reading from a File, we should move this Function to the Account Handler.

So, lift all of the code from the module `Manager.Account` (minus the import of `Prelude`) and put it in `Handler.Account`.

Here's what I moved:

```
import Data.Char (toCharCode)
import Data.Foldable (intercalate, foldl) ❶
import Data.String.CodeUnits (toCharArray, fromCharArray)
import Random.LCG (Seed, mkSeed)
import Test.QuickCheck.Arbitrary (arbitrary)
import Test.QuickCheck.Gen (sample)

userNameSeed :: String -> Seed
userNameSeed userName =
  userName
    # toCharArray
    <#> toCharCode
    # foldl (*) 1
    # mkSeed

userNameSalt :: Int -> String -> String
userNameSalt saltLength userName =
  fromCharArray $ sample (userNameSeed userName) saltLength arbitrary
```

❶ This import was already in `Handler.Account`, so I just added `foldl` to the import list.

Next, add `crypto` to your `spago.dhall` then build the server ignoring the expected warning.

We get the error:

```
[error] Some of your project files import modules from packages that are not
in the direct dependencies of your project.
To fix this error add the following packages to the list of dependencies in
your config:
- lcg
```

Add the `lcg` package to `spago.dhall` and rebuild.

In the module `Handler.Account`, write `passwordHash` using `hex` and the `Algorithm` of `SHA512`. Make the `saltLength` 3 times longer than the `userName`. Since `hex` runs in the `Effect` Monad, use `liftEffect` on `hex` to return `Aff String`.

Once you're done, come back here and keep reading.

We'll start with its Type Signature:

```
passwordHash :: String -> String -> Aff String
```

Takes a `userName` and a `password` and will produce a `Hash` as a `String` of hex characters. Notice that we return a `String` in the `Aff` Context. That's because of our use of `hex`.

Next, we'll write the left-hand side:

```
passwordHash :: String -> String -> Aff String
passwordHash userName password =
  ???
```

Next, step is to calculate the Salt using `userNameSalt`:

```
import Data.String.CodePoints (length) ❶

passwordHash :: String -> String -> Aff String
passwordHash userName password =
  let salt = userNameSalt (3 * length userName) userName in ❷ ❸
  ???
```

❶ Import `length`.

❷ Make the Salt 3 times longer than the `userName`.

❸ Note the `in` keyword. That's because we're not in a `do` block.

Finally, we'll call `hex` on the `password` concatenated with its `salt`:

```
import Node.Crypto.Hash (Algorithm(..), hex) ❶
import Effect.Class (liftEffect)

passwordHash :: String -> String -> Aff String
passwordHash userName password =
  let salt = userNameSalt (3 * length userName) userName in
  liftEffect $ hex SHA512 (password <> salt) ❷ ❸ ❹
```

❶ Add these imports.

❷ Use the `SHA512` hash algorithm.

❸ Append `salt` to `password` to add some randomness into the `password` hash.

❹ Use `liftEffect` to convert the `Effect` of `hex` to `Aff`.

Now, we're getting a shadowed variable warning on `passwordHash` in `accountToCSV`. Let's fix this by renaming our new Function to `passwordHashHex`:

```
passwordHashHex :: String -> String -> Aff String
passwordHashHex userName password =
  let salt = userNameSalt (3 * length userName) userName in
  liftEffect $ hex SHA512 (password <> salt)
```

Now that we have a way to hash passwords, let's use it on the `placeholder` text we have in the `bootstrapAccount`:

```
-- userName,passwordHash,temporaryPassword,admin,firstName,lastName
bootstrapAccount :: String
bootstrapAccount = "admin,placeholder,true,true,Joe,Admin" ❶
```

❶ `placeholder` on this line.

Keep in mind that adding a call to `passwordHashHex` here will change `bootstrapAccount's` Type Signature to `Aff String`. Change `placeholder` to the hash for a password of `"admin"`. Remember that case matters on passwords.

Make that change before reading on.

Let's change the Type Signature first:

```
bootstrapAccount :: Aff String
```

Next, let's use a `do` block:

```
bootstrapAccount :: Aff String
bootstrapAccount = do
  ???
```

Now, let's set `userName` and `password` Values:

```
bootstrapAccount :: Aff String
bootstrapAccount = do
  let userName = "admin"
      password = "admin"
  ???
```

Now, let's get the hash:

```
bootstrapAccount :: Aff String
bootstrapAccount = do
  let userName = "admin"
      password = "admin"
  passwordHash <- passwordHashHex userName password
  ???
```

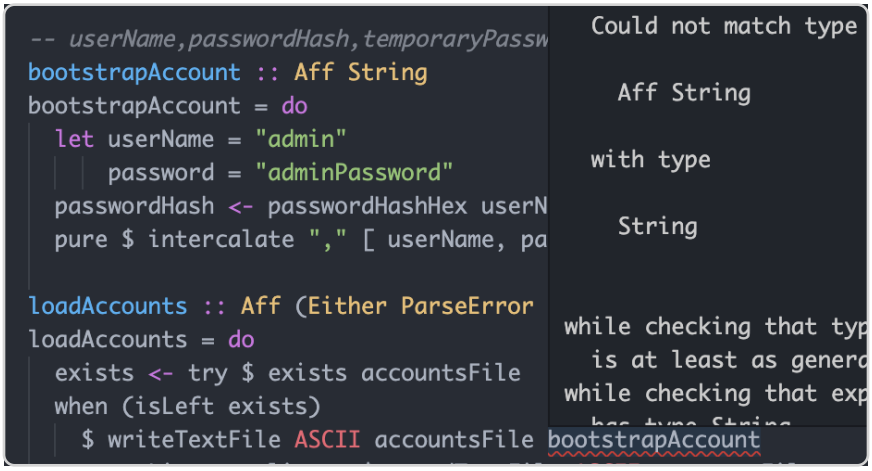
And finally, let's put it all together:

```
bootstrapAccount :: Aff String
bootstrapAccount = do
  let userName = "admin"
      password = "admin"
  passwordHash <- passwordHashHex userName password
  let true' = show true ❶
  pure $ intercalate "," ❷
    [ userName, passwordHash, true', true', "Joe", "Admin" ]
```

❶ `show true` is safer than `"true"` since typos will NOT be caught by the compiler.

❷ Using `intercalate` since we now have an `Array` of `String`.

Now, we're getting the following error where we used `bootstrapAccount` in `loadAccounts`:



Now, we're going to have to extract the `String` from the `Aff String` that `bootstrapAccount` now returns. Go ahead and modify `loadAccounts` before reading further.

Here's the change:

```
loadAccounts :: Aff (Either ParseError (Array Account))
loadAccounts = do
  exists <- try $ exists accountsFile
  when (isLeft exists) do ❶
    bsa <- bootstrapAccount ❷
    writeTextFile ASCII accountsFile bsa ❸
  accountLines <- lines <$> readTextFile ASCII accountsFile
  pure $ sequence
    $ unwrap <<< flip runParserT accountParser <$> accountLines
```

❶ Added a `do` block to do more than one computation in `Aff` than we were doing before, which was just to write the File.

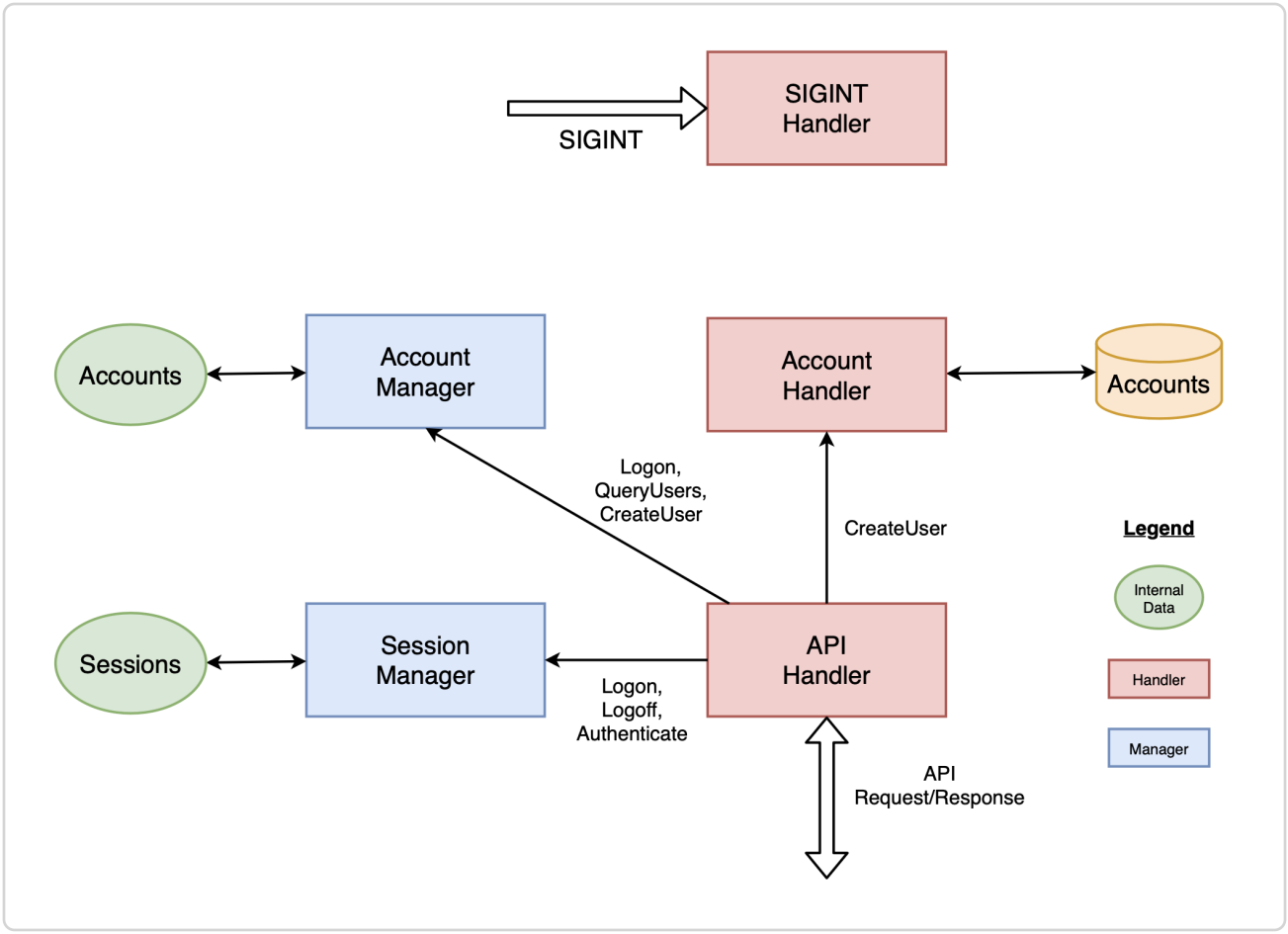
❷ Extract the `String` from the `bootstrapAccount`.

❸ Use the extracted `String` in place of where we were using the `bootstrapAccount`.

We need to do one more thing before we can finally verify the logon credentials and that's start up and shutdown the Account Manager.

28.7.2. Coding `startup` and `shutdown` for Account Manager

When we look back out our infrastructure diagram, we can see that `Accounts` is an in-memory, persisted Data Structure:



That means we're going to need to use an `AVar` since this data must be persisted across multiple HTTP Requests. We'll pass the `AVar` into the Account Manager Functions.

The `Accounts` Data Structure will be a Key-Value Store where the Key is the `userName` and the Value is the `Account`. We'll use a `Map` for this:

```
data Map k v
```

To see what API `Map` supports, go to `Pursuit` and search for `Map` and then browse the page with the above definition. Take note of `empty`, `insert`, `lookup` and `fromFoldable` to get acquainted.

Return to the `Manager.Account` file before reading on.

We'll start with defining the `Accounts` Type Alias:

```
import Data.Map (Map) ❶
import Entity.Account (Account) ❷

type Accounts = Map String Account ❸
```

❶ Import `Map`.

❷ Import `Account` from our program.

❸ The `key` is `userName` and the `value` is `Account`.

A Type Alias works well in this case since we already have a `Map` Type but don't want to type this out every time.

Add the `Accounts` Type and their corresponding imports (see above).

Next, we're going to code the startup and shutdown Functions. Start by writing a Function called `startup`. It will take an `AVar` of `Accounts`, an `Array Account` and return `Aff Unit`. It will initialize `Accounts` and save it in the `AVar`.

Give this a try before reading any further.

Let's start with the Type Signature and the necessary imports:

```
import Effect.Aff (Aff) ❶
import Effect.Aff.AVar (AVar)

startup :: AVar Accounts -> Array Account -> Aff Unit
```

❶ Add these imports.

Now, we'll start the implementation:

```
startup :: AVar Accounts -> Array Account -> Aff Unit
startup accountsAVar accounts = do
  ???
```

We've started a `do` block since we're coding in `Aff`, but we may delete this if it's unnecessary.

Next, we want to take the `accounts`, which is `Array Account`, and convert it into an `Array` where each element is a `Tuple` with `userName` and a single `account`. We'll `map` to accomplish this.

We're doing this is because we're going to turn the `Array` of `Tuples` into a `Map`, where the `key` is `userName` and the `value` is `Account`.

If you got stuck on this step, stop now and see how far you can get. Then read on.

Here's that change:

```
import Data.Tuple (Tuple(..)) ❶
import Entity.Account (Account(..)) ❷

startup :: AVar Accounts -> Array Account -> Aff Unit
startup accountsAVar accounts = do
  accounts
    <#> \account@(Account { userName }) -> Tuple userName account ❸ ❹
```

❶ Import `Tuple` and its Data Constructors.

❷ Added `(..)` to import `Account` Data Constructors for the Pattern Matching.

❸ Destructure `Account` to get to `userName`.

❹ Name the Variable `account` using the `@` syntax.

Now, take this `Array (Tuple String Account)` and use `fromFoldable` from `Map` to produce `Map String Account`, which is just `Accounts`. Then continue reading.

Here's that change:

```
import Data.Map as Map ❶

startup :: AVar Accounts -> Array Account -> Aff Unit
startup accountsAVar accounts = do
  (accounts
      <#> \account@(Account { userName }) -> Tuple userName account) ❷
    # Map.fromFoldable
```

❶ Add qualified import of `Map`.

❷ We must add Parentheses here otherwise, the `# Map.foldable` will be part of the Lambda.

And finally, we need to write this to the `AVar` using `put`:

```
import Effect.Aff.AVar as AVar ❶

startup :: AVar Accounts -> Array Account -> Aff Unit
startup accountsAVar accounts = ❷
  (accounts
      <#> \account@(Account { userName }) -> Tuple userName account)
    # Map.fromFoldable
    # flip AVar.put accountsAVar ❸
```

❶ Import `AVar` qualified.

❷ Removed the `do` since this is all in 1 line of code.

❸ We have to flip `AVar.put` to continue this kind of piping.

And now, that I look at this, I'm wondering if we should have this Function create the `AVar` instead of passing it in. Seems like it should. And if it creates it, then it should return it otherwise no one can use it.

Make those changes to your before reading on.

The Type Signature will change:

```
startup :: Array Account -> Aff (AVar Accounts) ❶
```

❶ We should return the `AVar` so it can be used by others.

Now we only have 1 Parameter:

```
startup :: Array Account -> Aff (AVar Accounts)
startup accounts = do ❶
  ???
```

❶ Added back the `do` since we're going to have a multi-step Monadic Computation.

We start with the code to create the `Map` that we had before:

```
startup :: Array Account -> Aff (AVar Accounts)
startup accounts = do
  (accounts
      <#> \account@(Account { userName }) -> Tuple userName account)
    # Map.fromFoldable
  ???
```

And finally, we create the `AVar` and return it:

```
startup :: Array Account -> Aff (AVar Accounts)
startup accounts = do
  (accounts
      <#> \account@(Account { userName }) -> Tuple userName account)
    # Map.fromFoldable
    # AVar.new
```

Next, let's write `shutdown`. But what should `shutdown` do? What clean up does it need to do?

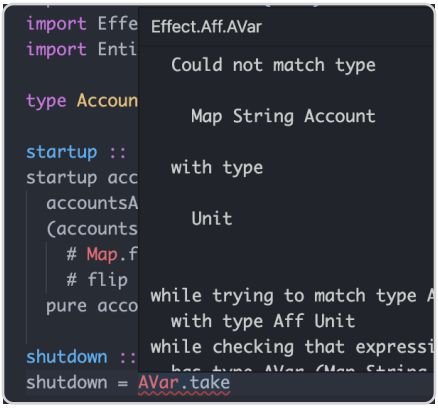
The only thing I can think of is to empty the `AVar` so no one else can get it. Write `shutdown` before reading on.

Here's `shutdown`:

```
shutdown :: AVar Accounts -> Aff Unit
shutdown = AVar.take -- COMPILER ERROR!! ❶
```

❶ Written Point-free where the `AVar` is the unnamed Parameter.

Here's the compiler's complaint:



Looks like `AVar.take` returns something other than `Unit`. Go look at `Pursuit` to see if that's right. Then return here.

Here's the Type Signature for `take`:

```
take :: ∀ a. AVar a -> Aff a
```

Now, that I look at this Type Signature, it makes sense that it would return the thing that we just took.

So, let's `void` that for our purposes:

```
shutdown :: AVar Accounts -> Aff Unit
shutdown = void <<< AVar.take
```

And now, we finally ready to write the verification of a logon.

28.7.3. Coding `verifyLogon`

Still working in the Account Manager, we want a Function that we'll call `verifyLogon` that takes the `userName` and a `password` and, first, makes sure that the `Account` exists and, second, that the credentials are correct, i.e. the `passwordHash`, which is calculated from `password`, matches what we have on record.

Stop here and code up just the Type Signature before reading on.

Here's the Type Signature:

```
verifyLogon :: AVar Accounts -> String -> String -> Aff Boolean
```

The first Parameter is `Accounts` via the `AVar`. The second Parameter is `userName` and the third is the `password`.

`verifyLogon` returns `true` if the credentials match. We're going to have to calculate the `passwordHash` by calling `passwordHashHex` in the Account Handler, i.e. in the `Handler.Account` module.

Stop reading here and code the rest of this Function. Then return here.

Here's the start of the implementation:

```
verifyLogon :: AVar Accounts -> String -> String -> Aff Boolean
verifyLogon accountsAVar userName password = do
  ???
```

Next, we should calculate the password `Hash`:

```
import Handler.Account (passwordHashHex) ❶

verifyLogon :: AVar Accounts -> String -> String -> Aff Boolean
verifyLogon accountsAVar userName password = do
  passwordHash' <- passwordHashHex userName password ❷
  ???
```

❶ Import `passwordHashHex`.

❷ Proactively naming this Variable with a Prime since it'll be compared with `passwordHash` from the Account and we don't want name collisions.

And now, we should read the `Accounts` from the `AVar`:

```
verifyLogon :: AVar Accounts -> String -> String -> Aff Boolean
verifyLogon accountsAVar userName password = do
  passwordHash' <- passwordHashHex userName password
  accounts <- AVar.read accountsAVar ❶
  ???
```

❶ We're using `read` instead of `take` since we aren't going to update them.

And then we should look up the `Account`:

```
verifyLogon :: AVar Accounts -> String -> String -> Aff Boolean
verifyLogon accountsAVar userName password = do
  passwordHash' <- passwordHashHex userName password
  accounts <- AVar.read accountsAVar
  let account' = Map.lookup userName accounts ❶ ❷
  ???
```

❶ Realize that `Map.lookup` can fail to find what it's looking for and, therefore, returns a `Maybe`.

❷ I like to name my Variables with a Prime when they return an `Either` or a `Maybe`. That frees up the non-Prime name in case I need it. It also reminds me that there's something up with this Variable.

And now, we'll return whether we found it and if the `Hashes` match or not:

```
import Data.Maybe (fromMaybe) ❶

verifyLogon :: AVar Accounts -> String -> String -> Aff Boolean
verifyLogon accountsAVar userName password = do
  passwordHash' <- passwordHashHex userName password
  accounts <- AVar.read accountsAVar
  let account' = Map.lookup userName accounts
  pure $ (account' <#> \(Account { passwordHash }) ❷ ❸
    -> passwordHash == passwordHash') # fromMaybe false ❹ ❺
```

❶ Import `fromMaybe`.

❷ We use `pure` to wrap the `Boolean` in `Aff`.

❸ We `map` into the `Maybe` with the Lambda to see if the `Hashes` match.

❹ We needed the Parentheses otherwise the `# fromMaybe false` will be part of the Lambda.

❺ If we don't find the `userName`, then it's NOT a valid logon, hence the `false`.

At this point, we could take one of two different paths. We could work on the Session Manager next, since after a logon, we'll need to create a Session.

Or we can hookup the rest of the plumbing to test a logon.

We have to do both, but since we've written a lot of code that we haven't tested yet, I think finishing off the logon logic would be best. Once we can do a verified logon, we'll immediately build the Session Manager since we need it to authenticate all subsequent requests to our server.

28.7.4. Finishing the `logon` Request

It's been a while since we've looked at our `Handler.Api.Logon` code.

Look it over and take time to refamiliarize yourself with it:

```
handler :: LogonRequest -> ResponseM
handler req = do
  HTTPure.notFound ❶

instance apiHandlerLogon :: ApiHandler Logon where
  handle request _ = do
    logonReq <- runExcept (decodeJSON request :: _ LogonRequest) ❷
    pure $ handler logonReq ❸
```

❶ This is a stand-in for the real handler. So, we'll need to finish this code.

❷ Attempt to decode the request as a `LogonRequest`.

❸ If the previous decode succeeds then we can call the handler. Otherwise, this will Short-circuit on the failed decode since `handle` runs in the `Either` Monad.

Now, let's work on the `handler`. Think about what it needs to do. Ask yourself, what happens when a `LogonRequest` is made.

Here's `LogonRequest` to help you answer that question:

```
newtype LogonRequest = LogonRequest
  { userName :: String
  , password :: String
  }
```

And here's the `LogonResponse`:

```
data LogonResults
  = LogonResultsSuccess
    { authToken :: UUID
    , mustChangePassword :: Boolean
    }
  | LogonResultsFailure

newtype LogonResponse = LogonResponse LogonResults
```

Think about what we need before moving on.

When I look at the request and response that our `handler` needs to work with, I can see that we're going to need to first call `verifyLogon` and then if that's good, we'll need to respond with an `authToken` and whether the password must be changed.

And that last part is a problem since we need `Account` information to determine if `mustChangePassword`. We aren't returning the `Account` information from `verifyLogon`:

```
verifyLogon :: AVar Accounts -> String -> String -> Aff Boolean
```

How should we change this so we can get back the `Account` information if it verifies? Think about how we should change our Type Signature. Then read on.

Maybe you thought of:

```
verifyLogon
  :: AVar Accounts
  -> String
  -> String
  -> Aff (Tuple Boolean Boolean) ❶ ❷ ❸
```

❶ The idea here is we'll return if `mustChangePassword` and if verified.

❷ It's hard to tell which `Boolean` is which here.

❸ This is a bad choice because we don't always get an `Account`. We could make it a `Maybe` or always `false` if we fail to get a password, but that's ugly too.

This approach is flawed for the aforementioned reasons. A much better approach that avoids all of these pitfalls is to return an `Account` in a `Maybe`:

```
verifyLogon :: AVar Accounts -> String -> String -> Aff (Maybe Account)
```

Return to `Manager.Account` and make this change to `verifyLogon` and change the code accordingly before reading further.

Here are those changes:

```
import Data.Maybe (Maybe(..)) ❶

verifyLogon :: AVar Accounts -> String -> String -> Aff (Maybe Account) ❷
verifyLogon accountsAVar userName password = do
  passwordHash' <- passwordHashHex userName password
  accounts <- AVar.read accountsAVar
  let account' = Map.lookup userName accounts
  pure $ (account' >>= \account@(Account { passwordHash }) ❸ ❹ ❺
      -> if passwordHash == passwordHash' then Just account else Nothing)
```

❶ Added `Maybe` and its Data Constructors and removed `fromMaybe`.

❷ Return Type is now `Aff (Maybe Account)`.

❸ Changed `map` to `bind` since the Lambda will now return a `Maybe`.

❹ Assign `account` using `@` syntax.

❺ Added `if` check and removed `fromMaybe`.

Now, that we're looking at `verifyLogon`, we can see that we're going to have to pass it an `AVar`. But where are we going to get this `AVar` from?

Let's think this through. When our `main` runs, it'll `startup` the Account Manager. This will return the `AVar` in question. We can provide this to our `router` so it can pass it along to the handlers.

I feel like this would best be done via a `ReaderT`. This means that we can add other things to `HandlerEnv`, e.g. other `AVars`, without having to pass a bunch of Parameters to all of our Functions.

So, let's change our `ApiHandler` definition to reflect that it runs in a `ReaderT`. Here's the current definition from `Handler.Class.ApiHandler`:

```
class ApiHandler :: ∀ k. k -> Constraint
class ApiHandler a where
  handle :: String -> Proxy a -> Either MultipleErrors ResponseM
```

Create a Type called `HandlerEnv` in `Handler.Class.ApiHandler` that's a Record that contains an `AVar Accounts`. This will be the read-only data that `ReaderT` passes along. We'll add another `AVar` for the Session Manager eventually.

Then change the Type Signature on the `handle` Function to add in the `ReaderT` for the success case of the `Either`.

Make these changes keeping in mind that `ResponseM` is just a Type Alias for `Aff`:

```
type ResponseM = Aff Response
```

Then read on.

We want to build a `ReaderT` Monad Stack with `Aff` at the bottom:

```
import Control.Monad.Reader (ReaderT) ❶
import Effect.Aff (Aff) ❶
import Effect.Aff.AVar (AVar) ❶
import HTTPure.Response (Response) ❷
import Manager.Account (Accounts) ❶

type HandlerEnv = ❸
  { accountsAVar :: AVar Accounts
  }

class ApiHandler :: ∀ k. k -> Constraint
class ApiHandler a where
  handle
    :: String
    -> Proxy a
    -> Either MultipleErrors (ReaderT HandlerEnv Aff Response) ❹ ❺
```

❶ Added these imports.

❷ Removed `ResponseM` and replaced it with `Response`.

❸ This is our Handler Environment. Remember that `HandlerEnv` is read-only, but the underlying Data Structure that the `AVar` references is not.

❹ Upon a successful decode, we return a `ReaderT`. This will need to be run using `runReaderT`.

❺ Our Monad Stack is a `ReaderT` on top and an `Aff` on the bottom.

Take a look at the Monad Stack we've built:

```
ReaderT HandlerEnv Aff Response
```

After `runReaderT` is called on our Stack, we'll be left with `Aff Response`. This is a `ResponseM`, which `HTTPure` Functions operate in. We're going to try and keep that in mind as we continue.

Monad Stacks are pretty verbose, so let's make the following change to simplify it:

```
type Handler = ReaderT HandlerEnv Aff Response ❶

class ApiHandler :: ∀ k. k -> Constraint
class ApiHandler a where
  handle :: String -> Proxy a -> Either MultipleErrors Handler ❶
```

❶ Factored out the `ReaderT` portion as `Handler`. This is so much nicer, especially since all of our handlers are going to use this definition and it'll clean up those Type Signatures a bit.

Now, return to `Handler.Api.Logon` and notice the compiler error. If it doesn't exist, then save the file if you're in `VSCode` and you'll see it.

This is because we've changed the Type Signature for `handle`. Fix this before moving on.

We just need to fix the `handle` Function for `Logon`:

```
import Handler.Class.ApiHandler (class ApiHandler, Handler) ❶ ❷

handler :: LogonRequest -> Handler
handler _ = do
  HTTPure.notFound
```

❶ Added `Handler` to import list.

❷ Deleted import of `HTTPure.Response` since we're not using `ResponseM` anymore.

Notice how we did NOT have to change the `HTTPure.notFound` line. That's because of `notFound's` Type Signature:

```
notFound :: ∀ m. MonadAff m => m Response
```

It runs in any Monad that has a `MonadAff` Instance. Both our old Monad, `Aff`, and our new one `ReaderT` have `MonadAff` Instances. So, there's no need to change that part of the code.

And now, that `handler` is a `ReaderT`, we can `ask` for the `accountsAVar` which `verifyLogon` needs. Add that code to `handler` before reading on.

Here's that added:

```
import Control.Monad.Reader.Class (ask) ❶

handler :: LogonRequest -> Handler
handler _ = do
  { accountsAVar } <- ask ❷ ❸
  HTTPure.notFound
```

❶ Import `ask`.

❷ Destructure the `HandlerEnv` to just take out `accountsAVar`. We'll need this for `verifyLogon`.

❸ The compiler will complain that we're not using `accountsAVar`. Don't worry. We will.

Next, add the call to `verifyLogon` to `handler` and then return here.

First, let's look at `verifyLogon':

```
verifyLogon :: AVar Accounts -> String -> String -> Aff (Maybe Account)
verifyLogon accountsAVar userName password = do
...
```

We should first destructure `handler's` first Parameter to expose `userName` and `password` that `verifyLogon` is going to need:

```
import Data.Api.Logon (LogonRequest(..)) ❶

handler :: LogonRequest -> Handler
handler (LogonRequest { userName, password }) = do ❷
  { accountsAVar } <- ask
  HTTPure.notFound
```

❶ Added import for Data Constructors.

❷ Destructure using Pattern Matching.

Next, we're going to call `verifyLogon` and respond accordingly:

```
import Manager.Account (verifyLogon) ❶
import Data.Maybe (Maybe(..))

handler :: LogonRequest -> Handler
handler (LogonRequest { userName, password }) = do
  { accountsAVar } <- ask
  case verifyLogon accountsAVar userName password of
    Nothing -> ???
```

❶ Add these 2 imports.

Stop here and handle both the error case and the success case of the call to `verifyLogon`. Then return here and continue reading.

For the error case, we'll respond with `LogonResultsFailure`:

```
import Data.Api.Logon (LogonRequest(..)
                     , LogonResponse(..), LogonResults(..)) ❶ ❷

handler :: LogonRequest -> Handler
handler (LogonRequest { userName, password }) = do
  { accountsAVar } <- ask
  case verifyLogon accountsAVar userName password of
    Nothing -> LogonResponse LogonResultsFailure ❸
    ???
```

❶ Added `(..)` to import Data Constructors for `LogonRequest`.

❷ Added `LogonResponse` and `LogonResults` with their Data Constructors.

❸ Failure case.

And now for the success case:

```
import Entity.Account (Account(..)) ❶
import Data.UUID (emptyUUID) ❷

handler :: LogonRequest -> Handler
handler (LogonRequest { userName, password }) = do
  { accountsAVar } <- ask
  case verifyLogon accountsAVar userName password of
    Nothing -> LogonResponse LogonResultsFailure
    Just (Account { temporaryPassword }) -> LogonResponse
      $ LogonResultsSuccess
        { authToken: emptyUUID -- TODO get from Session Mgr ❸
        , mustChangePassword: temporaryPassword ❹
        }
  ???
```

❶ Import `Account` Data Constructor for Pattern Matching.

❷ Temporarily import `emptyUUID`.

❸ We'll eventually call the Session Manager to create a `Session` for us which will return an `authToken`. We're using `emptyUUID` for now.

❹ If the password on file is temporary, then `mustChangePassword` is `true`.

At this point, we've coded the `case` expression, but we're not doing anything with it. This expression is the response. So, we can respond with it after it's been encoded.

Add the code to encode the response, i.e the `case` expression and then respond with it using `HTTPure.ok`. Return here when you've finished.

Here's that addition:

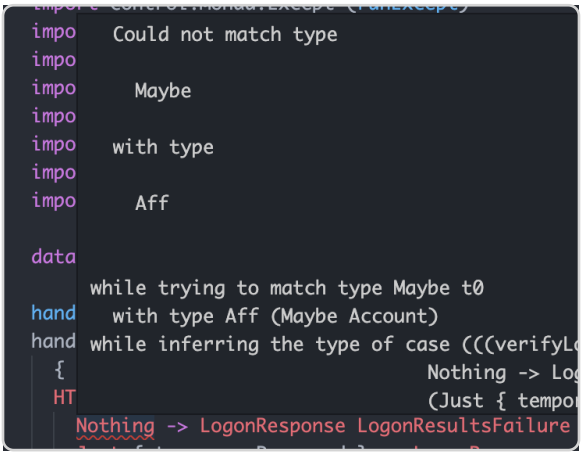
```
import Foreign.Generic (decodeJSON, encodeJSON) ❶

handler :: LogonRequest -> Handler
handler (LogonRequest { userName, password }) = do
  { accountsAVar } <- ask
  HTTPure.ok $ encodeJSON ❷
    $ case verifyLogon accountsAVar userName password of
      Nothing -> LogonResponse LogonResultsFailure -- COMPILER ERROR!!
      Just (Account { temporaryPassword }) -> LogonResponse
        $ LogonResultsSuccess
          { authToken: emptyUUID -- TODO get from Session Mgr
          , mustChangePassword: temporaryPassword
          }
```

❶ Added `encodeJSON` to the import list.

❷ First encode the response and then sent it to the client.

The compiler error is:



It's trying to Pattern Match a `Maybe` with an `Aff`. Can you see my mistake?

I forgot that `verifyLogon` runs in `Aff`. It's possible that you didn't forget this. If so, then good on you.

Let's fix my mistake:

```
import Control.Monad.Trans.Class (lift) ❶

handler :: LogonRequest -> Handler
handler (LogonRequest { userName, password }) = do
  { accountsAVar } <- ask
  verifiedAccount <- lift $ verifyLogon accountsAVar userName password ❷
  HTTPure.ok $ encodeJSON $ case verifiedAccount of ❸
    Nothing -> LogonResponse LogonResultsFailure
    Just (Account { temporaryPassword }) -> LogonResponse
      $ LogonResultsSuccess
        { authToken: emptyUUID -- TODO get from Session Mgr
        , mustChangePassword: temporaryPassword
        }
```

❶ Import `lift`.

❷ We need to `lift` here because `verifyLogon` runs in the base of our Monad Stack, i.e. in `Aff`. We only need to `lift` once since we only have 2 layers.

❸ `case` on the extracted results of `verifiedAccount`.

And now this compiles.

To be able to test this, we're going to need to go from the `main` Function to `handler`. That will mean that

we'll have to flesh out our `router`.

28.7.5. Fleshing out `router`

Here's where we've left off with `router` in the `Main` module:

```
router :: Request -> ResponseM
router _ = HTTPure.notFound
```

The `router` will need to try out each `ApiHandler` until it finds one that is successful and then it will call `runReaderT` on its `handle`. This will return a `ResponseM` or, more precisely, an `Aff Response`.

Let's start with the code to find a successful handler.

Before you code this, let's review a few things. First, `oneOf` from `Data.Foldable`:

```
-- Combines a collection of elements using the Alt operation.
oneOf :: ∀ f g a. Foldable f => Plus g => f (g a) -> g a ❶ ❷
```

❶ For us, `f` here will be `Array`.

❷ For us, `g` here will be `Either`.

So, now let's look at `Either's` Alt implementation:

```
-- | The `Alt` instance allows for a choice to be made between
-- | two `Either` values with the `<|>` operator,
-- | where the first `Right` encountered is taken.
-- |
-- | ``` purescript
-- | Right x <|> Right y == Right x
-- | Left x <|> Right y == Right y
-- | Left x <|> Left y == Left y
-- | ```
instance altEither :: Alt (Either e) where
  alt (Left _) r = r
  alt l        _ = l
```

This means that the first successful `Either` will be selected by `oneOf` since it'll use the `Alt` implementation from `Either`.

We can use `oneOf` to "try out" each handler and the one that encodes it will produce a `Right` that will be selected. In that `Right` will be our API Response.

And finally, let's look at an `HTTPure.Request`:

```
type Request =
  { method :: Method.Method
  , path :: Path.Path
  , query :: Query.Query
  , headers :: Headers.Headers
  , body :: String ❶
  , httpVersion :: Version.Version
  }
```

❶ Our API JSON will be in the `body`. This is the `String` each `Handler` will process.

Study these pieces of information before you begin adding code to `router` in the `Main` module. Then add a bit of code to use `oneOf` and get a Handler into a Variable called `handler`.

Don't go any further than that in your coding. Still return `HTTPure.notFound` for now so that our code will still compile. Then read on when you get stuck or hit an insurmountable compiler error.

The first step is to destructure the `Request` to get to the `body` since `handle` needs the JSON request:

```
router :: Request -> ResponseM
router { body } = HTTPure.notFound
```

Now, we're going to add `oneOf` with a list of `Handlers`, of which we only have 1 at the moment:

```
import Data.Foldable (oneOf) ❶
import Handler.Api.Logon (Logon)
import Handler.Class.ApiHandler (handle)
import Type.Proxy (Proxy(..))

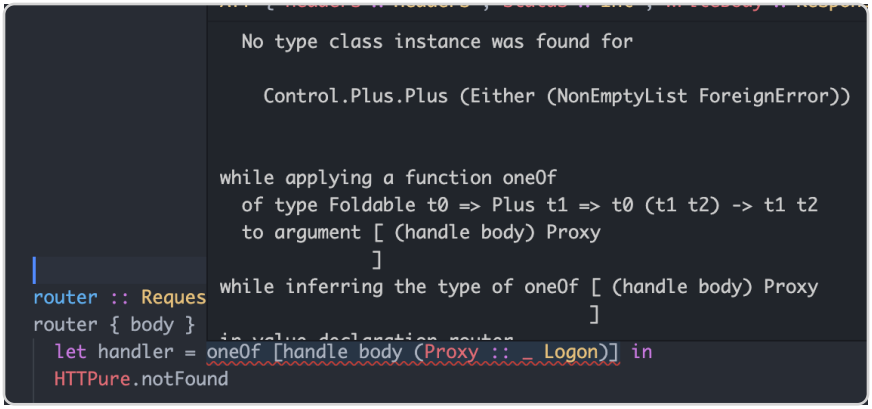
router :: Request -> ResponseM
router { body } =
  -- COMPILER ERROR!!
  let handler = oneOf [handle body (Proxy :: _ Logon)] in ❷
  HTTPure.notFound
```

❶ Added these imports.

❷ We use `Proxy` to define which `handle` Method to use.

Even though this has a compiler error, take the time to update your code to look like this.

Here's the error:



Can you see what's wrong? It's not obvious.

It turns out that `oneOf` from `Data.Foldable` has a `Plus` Constraint on our `Functor`, i.e. `Either`. Remember that `Plus` is just a `Monoid` for Types with 2 Type Parameters.

That means that it needs an `empty` value for `Either`. This is because its possible to pass `oneOf` an empty `Array`. In that case, it'll use `empty` from `Plus` as it's choice.

But, there is no `Plus` Instance for `Either`. When we talked about `Plus`, we saw how it's impossible to write a `Plus` instance for `Either`. That means we cannot use `oneOf`. Well, not this `oneOf`.

Instead, we'll have to use `oneOf` from `Data.NonEmpty`, which has no such restriction since it cannot get an empty `Array`. And since it cannot get an empty `Array`, this version of `oneOf` can always pick something:

```
oneOf :: ∀ f a. Alternative f => NonEmpty f a -> f a ❶
```

❶ Here `f` is the `Alternative`, which is a `Functor`. In our case, it's an `Array`.

Modify `router` to use `Data.NonEmpty's` version of `oneOf` before you read any further.

First, delete:

```
import Data.Foldable (oneOf)
```

And make these changes:

```
import Data.NonEmpty (oneOf, (:|)) ❶

router :: Request -> ResponseM
router { body } =
  let handler = oneOf $ (handle body (Proxy :: _ Logon)) :| [] in ❷ ❸
  HTTPure.notFound
```

❶ Replaced the import of `oneOf` from `Data.Foldable` with this one. Also, included the Binary Operator for constructing a `NonEmpty`.

❷ Construct a `NonEmpty Array` using `:|`.

❸ Ignore the warning about not using `handler`.

Now that we have `handler`, realize that it's going to be an `Either`. And when it's a `Left` it'll be the error from the last thing that is in the `Array`.

We don't really care about the errors of the last `Handler`. It's of no use to us or the client. We're just going to return an HTTP error code `400`, which is `Bad Request`.

So, change your code to deal with just the error case of `oneOf` call. Realize that your code won't compile after that.

Here's that change:

```
router :: Request -> ResponseM
router { body } =
  case oneOf $ (handle body (Proxy :: _ Logon)) :| [] of ❶
    Left _ -> HTTPure.badRequest body ❷
    ???
  HTTPure.notFound
```

❶ Changed the code from a `let` to a `case`.

❷ We don't care about the error details here. We just respond with `badRequest`, i.e. HTTP status of `400`.

The next question we should ask ourselves is what Type is in the `Right`? How can we figure that out?

Well, we can simply look at `Handler's` definition:

```
type Handler = ReaderT HandlerEnv Aff Response
```

This shows us that we have a `ReaderT` that needs to be run. But we don't have any read-only Value to pass to `runReaderT` yet.

We'll need to get that from `main`. So our code will compile while we work on `main`, comment out the `case` expression and we'll return to this in a minute:

```
router :: Request -> ResponseM
router { body } =
  -- case oneOf $ (handle body (Proxy :: _ Logon)) :| [] of ❶
  --   Left err -> HTTPure.badRequest body
  HTTPure.notFound
```

❶ Commented out just for now.

Now, let's turn our attention to `main`:

```
main :: Effect Unit
main = do
  shutdown <- HTTPure.serve port router
    $ log $ "Server up running on port: " <> show port
  let shutdownServer = do
        log "Shutting down server..."
        shutdown $ log "Server shutdown."
  onSignal SIGINT shutdownServer
  onSignal SIGTERM shutdownServer
```

We need to startup the Account Manager and then use the `accountsAVar` that it returns to pass to our `router` so it can call `runReaderT` with it.

Add the code to `main` to startup the Account Manager before reading on.

The first problem we encounter is that the code in `main` is running in `Effect` but not `Aff` like the rest of our program.

So, that's the first change to make:

```
import Effect.Aff (launchAff_) ❶
import Effect.Class (liftEffect)

main :: Effect Unit
main = launchAff_ do ❷
  liftEffect $ do ❸
    shutdown <- HTTPure.serve port router
      $ log $ "Server up running on port: " <> show port
    let shutdownServer = do
          log "Shutting down server..."
          shutdown $ log "Server shutdown."
    onSignal SIGINT shutdownServer
    onSignal SIGTERM shutdownServer
```

❶ Added these imports.

❷ Run this `do` block in `Aff`.

❸ Lift this `do` block, which runs in `Effect`, into `Aff`.

Now we make a call to `startup`

```
import Manager.Account as AccountManager ❶

main :: Effect Unit
main = launchAff_ do
  accountsAVar <- AccountManager.startup [] ❷
  liftEffect $ do
    shutdown <- HTTPure.serve port router
      $ log $ "Server up running on port: " <> show port
    let shutdownServer = do
          log "Shutting down server..."
          shutdown $ log "Server shutdown."
  onSignal SIGINT shutdownServer
  onSignal SIGTERM shutdownServer
```

❶ Full qualify the Account Manager.

❷ Used an empty account list for now. We really need to get this from the Account Handler.

Make sure your code is up to date with this and then add code to call the Account Handler to load the accounts. Don't forget that this will create the file if need be.

Once you do, take the accounts that it loaded and replace our `[]` with them. Give this a go before reading on.

Here's that change:

```
import Handler.Account as AccountHandler ❶

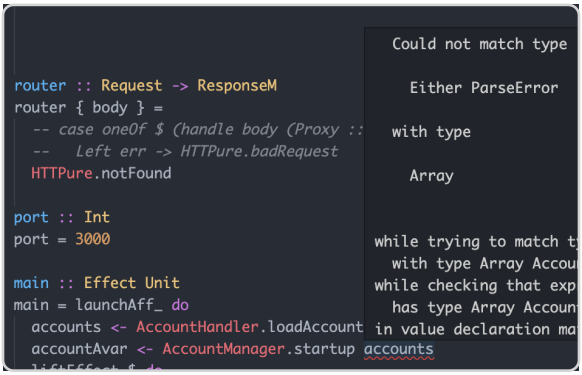
main :: Effect Unit
main = launchAff_ do
  accounts <- AccountHandler.loadAccounts ❷
  accountsAVar <- AccountManager.startup accounts -- COMPILER ERROR!! ❸
  liftEffect $ do
    shutdown <- HTTPure.serve port router
      $ log $ "Server up running on port: " <> show port
    let shutdownServer = do
          log "Shutting down server..."
          shutdown $ log "Server shutdown."
  onSignal SIGINT shutdownServer
  onSignal SIGTERM shutdownServer
```

❶ Import Handler.Account qualified as `AccountHandler`.

❷ Load the accounts from the Account Handler.

❸ Startup the Account Manager with `accounts`.

Here's the compiler error:



Looks like I forgot that `loadAccounts` can have errors:

```
loadAccounts :: Aff (Either ParseError (Array Account))
```

So, we'll fix my mistake:

```
import Data.Either (Either (..)) ❶

main :: Effect Unit
main = launchAff_ do
  loadResults <- AccountHandler.loadAccounts ❷
  case loadResults of
    Left err -> log $ "Cannot load accounts: " <> show err ❸
    Right accounts -> do ❹
      accountsAVar <- AccountManager.startup accounts
      liftEffect $ do
        shutdown <- HTTPure.serve port router
          $ log $ "Server up running on port: " <> show port
        let shutdownServer = do
              log "Shutting down server..."
              shutdown $ log "Server shutdown."
        onSignal SIGINT shutdownServer
        onSignal SIGTERM shutdownServer
```

❶ Added this import.

❷ Get loading results to be checked for errors in the following `case` expression.

❸ Handle the error case by logging and exiting.

❹ Here's the success case where we startup the Account Manager and run our Server.

Next, we'll need to build a `HandlerEnv` and pass it to `router` so it can call `runReaderT` with it. Do that before reading my solution.

First, we'll change `router` to expect a `HandlerEnv`:

```
import Handler.Class.ApiHandler (HandlerEnv, handle) ❶

router :: HandlerEnv -> Request -> ResponseM ❷
router env { body } =
  -- case oneOf $ (handle body (Proxy :: _ Logon)) :| [] of
  --   Left err -> HTTPure.badRequest body
  HTTPure.notFound
```

❶ Added `HandlerEnv` to import list. I like to keep my Types and Class first in the list, Functions next and Binary Operators last.

❷ Put `HandlerEnv` as first Parameter so we could Partially Apply it in `main`.

Now, for `main`:

```
main :: Effect Unit
main = launchAff_ do
  loadResults <- AccountHandler.loadAccounts
  case loadResults of
    Left err -> log $ "Cannot load accounts: " <> show err
    Right accounts -> do
      accountsAVar <- AccountManager.startup accounts
      liftEffect $ do
        shutdown <- HTTPure.serve port (router { accountsAVar }) ❶
          $ log $ "Server up running on port: " <> show port
        let shutdownServer = do
              log "Shutting down server..."
              shutdown $ log "Server shutdown."
        onSignal SIGINT shutdownServer
        onSignal SIGTERM shutdownServer
```

❶ Partially apply `HandlerEnv` to `router`, which returns a Function of Type `Request -> ResponseM`, which is exactly what `HTTPure.server` expects as its first Parameter.

At this point, `router` has enough information to call `runReaderT`. Uncomment the code in `router` and then finish the `case` expression by calling the `ReaderT` that the `Right` case gets back. Then read on.

Here's that change:

```
import Control.Monad.Reader (runReaderT) ❶
import Data.Either (Either(..))

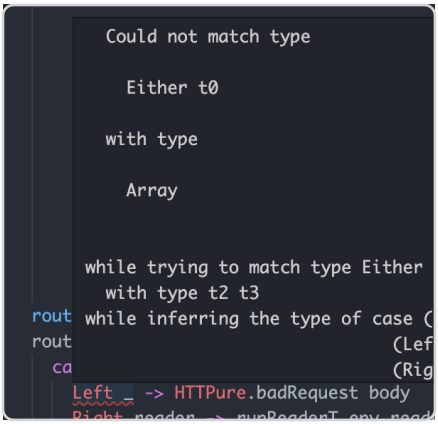
router :: HandlerEnv -> Request -> ResponseM
router env { body } =
  case oneOf $ (handle body (Proxy :: _ Logon)) :| [] of
    Left _ -> HTTPure.badRequest body -- COMPILER ERROR!!
    Right handler -> runReaderT handler env ❷ ❸
```

❶ Added these imports.

❷ Call `runReaderT` with the `HandlerEnv`. This will return an `Aff Response`, which is a `ResponseM`.

❸ Removed `HTTPure.notFound`.

And the compiler error is as follows:



Wait. I thought `oneOf` would return one item from the `NonEmpty Array` but it looks like that's not true. Let's look back at its Type Signature:

```
oneOf :: ∀ f a. Alternative f => NonEmpty f a -> f a
```

Oops. I didn't notice that it returned an `f a`. For us, `f` is an `Array`. So, we'll have to pull the head off of the return of `oneOf`.

But wait. If we use `head` to get the one element that `oneOf` selected, it'll give us a `Maybe (Either…)`. We can change the `Either` to a `Maybe` using `hush` since we don't care about the `ParseError`.

But mapping `hush` will give us `Maybe (Maybe…)`, so better to use `bind` instead to get a single `Maybe`. Make those changes before reading on.

Here are those changes:

```
import Data.Array (head) ❶
import Data.Either (Either(..), hush) ❷
import Data.Maybe (Maybe(..)) ❸

router :: HandlerEnv -> Request -> ResponseM
router env { body } =
  case hush =<< ❹
    (head $ oneOf $ (handle body (Proxy :: _ Logon)) :| []) of ❺
      Nothing -> HTTPure.badRequest body ❻
      Just reader -> runReaderT reader env ❼
```

❶ Import `head`.

❷ Added `hush` to import list.

❸ Added `Maybe` with Data Constructors to import list.

❹ The `case` expression REPLACES `HTTPure.notFound` as the last line of the `do` block.

❺ Get the `head` from `oneOf` and `hush` the contained `Either` to a `Maybe`.

❻ We check `Nothing` instead of `Left _` thanks to `hush`.

❼ We check `Just reader` instead of `Right reader` thanks to `hush`.

Now that it compiles, let's run it with:

```
npx spago run
```

When I do, I get:

```
[error] Some of your project files import modules from packages that are not
in the direct dependencies of your project.
To fix this error add the following packages to the list of dependencies in
your config:
- avar
- nonempty
- ordered-collections
- tuples
```

Add these packages to your `spago.dhall` and try to run it again and see what you get.

I get:

```
[info] Build succeeded.
/Users/charlesscalfani/dev/cscalfani/bookCode/fullstack/server14RT/output/Effe
ct.Aff/foreign.js:532
                throw util.fromLeft(step);
                ^

[Error: ENOENT: no such file or directory, open 'accounts.csv'] { ❶
  errno: -2,
  code: 'ENOENT',
  syscall: 'open',
  path: 'accounts.csv'
}
[error] Running failed; exit code: 1
```

❶ From this part of the error message, we can see that `accounts.csv` doesn't exist.

I thought we handled this case. Evidently not. Time to check out `loadAccounts` in `Handler.Account`. See if you can see where I went wrong before reading on.

Here's the problem:

```
loadAccounts :: Aff (Either ParseError (Array Account))
loadAccounts = do
  exists <- try $ exists accountsFile ❶
  when (isLeft exists) do ❷
    bsa <- bootstrapAccount
    writeTextFile ASCII accountsFile bsa
  accountLines <- lines <$> readTextFile ASCII accountsFile
  pure $ sequence
    $ unwrap <<< flip runParserT accountParser <$> accountLines
```

❶ `exists` is an `Either Error Boolean`, i.e. an `Error` if an I/O error occurred or a `Boolean` that tells us whether the file exists or not.

❷ We're only checking for `Left`. There's no check for `Right` case, which has our `Boolean` in it that tells us if the file doesn't exist.

Well, we need to check for both `Either` cases here. Fix this Function in our code to handle both the cases and then read on.

Here's the change:

```
import Data.Either (Either(..)) ❶

loadAccounts :: Aff (Either ParseError (Array Account))
loadAccounts = do
  exists' <- try $ exists accountsFile ❷
  let exists = case exists' of ❸
        Left _ -> false ❹
        Right b -> b ❺
  when exists do ❻
    bsa <- bootstrapAccount
    writeTextFile ASCII accountsFile bsa
  accountLines <- lines <$> readTextFile ASCII accountsFile
  pure $ sequence
    $ unwrap <<< flip runParserT accountParser <$> accountLines
```

❶ Removed `isLeft` and added `(..)`.

❷ Renamed `exists` to add a prime that tells me that it's not a Boolean but an `Either`.

❸ We compute whether it exists.

❹ If we had an error calling `exists` then we're assuming it does not exist.

❺ If we successfully called `exists`, then we will get a `Boolean`, i.e. `b`, that tells us if it does or does not

exist.

❻ Check computed `exists` now.

Now run it again and see what happens.

Did you get the same error as before. So did I. Looks like my assessment was wrong. We did, however, find a different bug, but not this runtime one.

Looking at `loadAccounts`, it seems that maybe the `writeTextFile` is not finishing before we try to `readTextFile`. Seems unlikely, but let's go with that theory for now.

Let's test this by making the following TEMPORARY changes:

```
loadAccounts :: Aff (Either ParseError (Array Account))
loadAccounts = do
  exists' <- try $ exists accountsFile
  let exists = case exists' of
        Left _ -> false
        Right b -> b
  when exists do
      bsa <- bootstrapAccount
      writeTextFile ASCII accountsFile bsa
  -- accountLines <- lines <$> readTextFile ASCII accountsFile ❶
  let accountLines = [] ❷
  pure $ sequence
    $ unwrap <<< flip runParserT accountParser <$> accountLines
```

❶ Comment out the line that's reading the file.

❷ Set `accountLines` to an empty `Array`.

Now run it and see what happens.

Did it run for you? It did for me too. Kill it with Ctrl-C.

So, that's the problem. But why? Think about it for a minute.

Well, I cannot see any reason that this is the case. The write should be finished... Unless, it's never written to.

Let's add some more test code:

```
import Effect.Class.Console (log) ❶

loadAccounts :: Aff (Either ParseError (Array Account))
loadAccounts = do
  exists' <- try $ exists accountsFile
  let exists = case exists' of
        Left _ -> false
        Right b -> b
  log $ "exists: " <> show exists ❷
  when exists do
      bsa <- bootstrapAccount
      writeTextFile ASCII accountsFile bsa
  -- accountLines <- lines <$> readTextFile ASCII accountsFile
  let accountLines = []
  pure $ sequence
    $ unwrap <<< flip runParserT accountParser <$> accountLines
```

❶ Import `log`.

❷ Log the value of `exists` to see if the file will be written to.

Now run it and see what it prints out.

Here's what I get:

```
exists: false
Server up running on port: 3000
```

So, it does not exist and the file isn't getting written to. Why? See if you can figure it out.

Looking at the logic, it's NOW obvious what's wrong:

```
loadAccounts :: Aff (Either ParseError (Array Account))
loadAccounts = do
  exists' <- try $ exists accountsFile
  let exists = case exists' of
        Left _ -> false
        Right b -> b
  log $ "exists: " <> show exists
  when exists do ❶
      bsa <- bootstrapAccount
      writeTextFile ASCII accountsFile bsa
  -- accountLines <- lines <$> readTextFile ASCII accountsFile
  let accountLines = []
  pure $ sequence
    $ unwrap <<< flip runParserT accountParser <$> accountLines
```

❶ This logic is backwards. It will only write to the file if it ALREADY exists.

Please remove our debug code including the `import Effect.Class.Console (log)` and fix my stupid bug before moving on.

The fix should look like:

```
loadAccounts :: Aff (Either ParseError (Array Account))
loadAccounts = do
  exists' <- try $ exists accountsFile
  let exists = case exists' of
        Left _ -> false
        Right b -> b
  unless exists do ❶
      bsa <- bootstrapAccount
      writeTextFile ASCII accountsFile bsa
  accountLines <- lines <$> readTextFile ASCII accountsFile ❷
  pure $ sequence
    $ unwrap <<< flip runParserT accountParser <$> accountLines
```

❶ Changed the logic to be correct.

❷ Uncommented.

Please check this code carefully to make sure your code matches.

Once you do, run the program and see what you get.

I get:

```
Cannot load accounts: (ParseError "Invalid Boolean" (Position { line: 1,
column: 155 })) ❶
```

❶ The `Boolean` parser is failing.

Okay, so this is good. We've gotten past that problem and have run head on into another. Go check your `accounts.csv` and see if you can figure out why column `155` is a bad `Boolean`. Then return here.

Well, here's my `accounts.csv`:

```
admin,f4cf17b2a9bae2e92974644a200eeee2b5a9096b1a502643cafe26b9e1fd21469d784e46
2e66290efe00ca9a1ca4da2767d0cce96372ab284100a221c79d3cce,true,true,Joe,Admin
```

The problem is that column `155` is at the end of the line.

That tells me that a parser before the `Boolean` parser already exhausted the `String`. This would be the `hex` parser since it's right before the first `true`.

Let's look at that parser in `Parser.Account` to see if it's bad:

```
hex :: AccountParserT String
hex = fromCharArray <$> (some $ satisfy isHex)
  where isHex c = (c >= '0' && c <= '9') || (c >= 'a' && c <= 'f')
```

Can you see anything wrong here?

I can't. This looks fine. Maybe our first `Boolean` parser consumes too many characters from the `String` leaving nothing left for our second one.

Let's look at that parser:

```
boolean :: AccountParserT Boolean
boolean = do
  bool <- fromCharArray <$> some anyChar
  case bool of
    "true" -> pure true
    "false" -> pure false
    _ -> fail "Invalid Boolean"
```

Can you see any issues here?

See that `some anyChar`. This is probably greedily consuming all of the characters remaining in the line.

This is happening long before we try to parse the `comma` in `accountParser`.

That was a big mistake on my part. Okay, no big deal. We'll just have to be smarter to request only alpha characters. So, go ahead and fix my mistake by only accepting alpha characters. Look around to see how we've done this before.

Then read on.

Here's the fix:

```
import Text.Parsing.Parser.String (satisfy, char) ❶

boolean :: AccountParserT Boolean
boolean = do
  bool <- fromCharArray <$> some (satisfy (isAlpha <<< codePointFromChar)) ❷
  case bool of
    "true" -> pure true
    "false" -> pure false
    _ -> fail "Invalid Boolean"
```

❶ Removed `anyChar`.

❷ Parse only alpha characters.

Now, that I'm looking at this code, it's clear to me that I should have used the `string` parser from our parsing library. Go look up `string` in the module `Text.Parsing.Parser.String`. Then return here.

See if you can rewrite this parser from scratch using `string`. Hint: Use `string` along with `<|>`. Give the rewrite a try before reading any more.

First, we'll start with this:

```
boolean :: AccountParserT Boolean
boolean = ???
```

Now, I'd like to have a parser that will parse `"true"`:

```
import Text.Parsing.Parser.String (satisfy, char, string) ❶

boolean :: AccountParserT Boolean
boolean = string "true" ???
```

❶ Added `string` to import list.

But I don't want the `String` that this returns. Instead I want a `true`. How do I take this `AccountParserT`

`String` and turn it into an `AccountParserT Boolean`?

If you said, `map`, you're not wrong, but there's a better way. Can you think of a different way?

If you cannot, here's the `map` implementation:

```
boolean :: AccountParserT Boolean
boolean = (string "true" <#> \_ -> true)
  <|> ???
```

Or we can use `const`:

```
boolean :: AccountParserT Boolean
boolean = (string "true" <#> const true)
  <|> ???
```

Both of these appraoches are valid and will work fine, but there's a more idiomatic approach. Using `*>`. See if you can figure out how before reading the answer.

Here's how:

```
boolean :: AccountParserT Boolean
boolean = (string "true" *> pure true)
  <|> ???
```

Next, we need to handle the `"false"` case:

```
boolean :: AccountParserT Boolean
boolean = (string "true" *> pure true)
  <|> (string "false" *> pure false)
  <|> ???
```

And finally we need to handle the failure case. Try to add that before reading on.

The failure case is handled by using `fail`:

```
import Control.Alt ((<|>)) ❶

boolean :: AccountParserT Boolean
boolean = (string "true" *> pure true)
  <|> (string "false" *> pure false)
  <|> fail "Invalid Boolean"
```

❶ Added import.

And this parser is much better than my first attempt. It's better because it leverages existing library Functions that have been tested and it reads better. Both good reasons for us to fixed this.

Running the server brings up:

```
[error] Some of your project files import modules from packages that are not
in the direct dependencies of your project.
To fix this error add the following packages to the list of dependencies in
your config:
- control
```

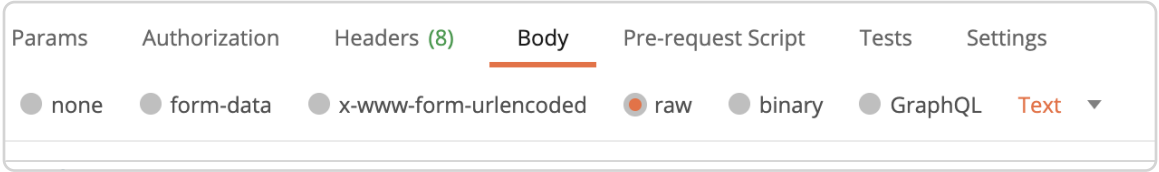
Fix this by adding the package to `spago.dhall` then rerun the server with our new and imporoved parser. What did you get?

If you did everything right, it should be running right now.

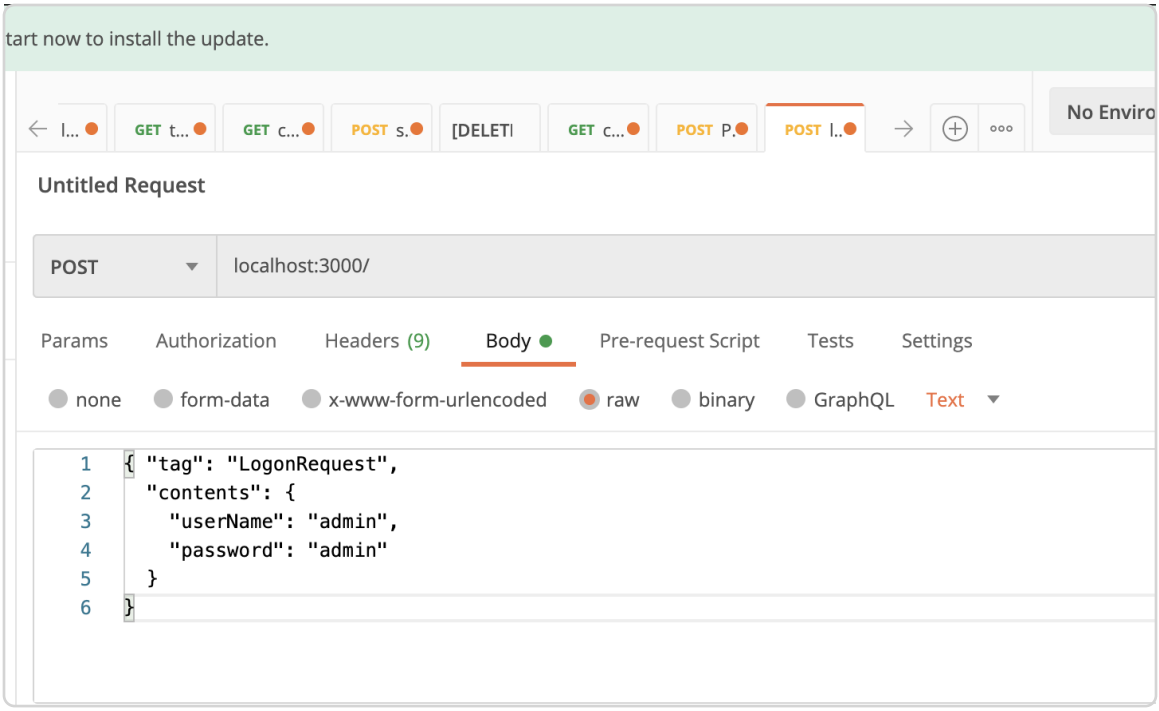
So, let's hop into `Postman` and test the `LogonRequest` using the following JSON body:

```
{ "tag": "LogonRequest",
  "contents": {
    "userName": "admin",
    "password": "admin"
  }
}
```

You'll have to click on the `Body` section and then click the Radio Button named `Raw` to enter in the above JSON:



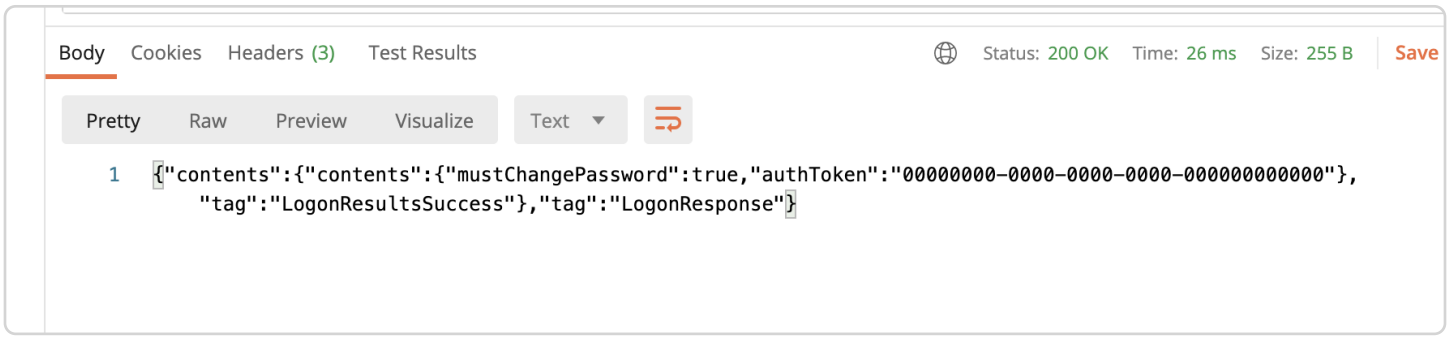
Here's what your `Postman` session should look like:



Make sure you're doing a `POST` and not a `GET`.

Now, click `Send`. What did you get?

I got:



So, it looks like that works. But my reminder to you to make sure you're making a `POST` requests makes me think about our `router`:

```
router :: HandlerEnv -> Request -> ResponseM
router env { body } =
  case hush =<<
    (head $ oneOf $ (handle body (Proxy :: _ Logon)) :| []) of
      Nothing -> HTTPure.badRequest body
      Just reader -> runReaderT reader env
```

It doesn't guard (pun intended) against `GET` requests. In fact, it doesn't check that we're making a `POST` request before it responds.

Change the `router` in `Main`, using Guards, to make sure that `method` from the `Request` is a `Post`. Use the code we commented out in `main` as a reference. Give this a go before reading any further.

Here's that fix:

```
router :: HandlerEnv -> Request -> ResponseM
router env { body, method } ❶ ❷
  | method == HTTPure.Post = ❸
      case hush =<<
        (head $ oneOf $ (handle body (Proxy :: _ Logon)) :| []) of
          Nothing -> HTTPure.badRequest body
          Just reader -> runReaderT reader env
  | otherwise = HTTPure.methodNotAllowed ❹
```

❶ Destructure `method` also.

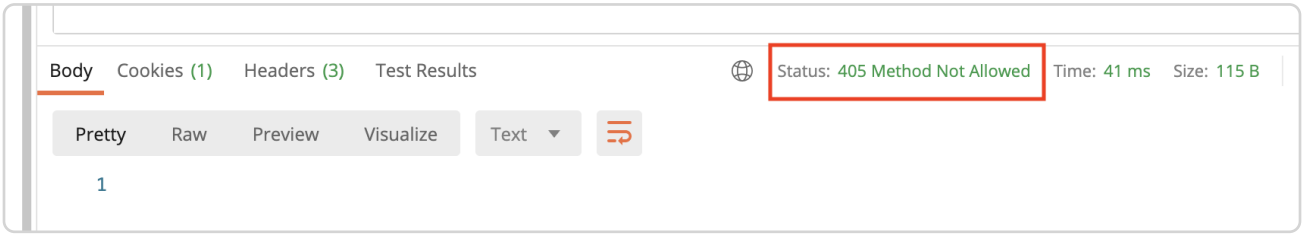
❷ Removed the equals sign since we're using Guards.

❸ Only respond to `POST` requests.

❹ Everything else is `405`, i.e. `Method Not Allowed`.

Rerun the server and then go to `Postman` and change `POST` to `GET` and send it and see if it returns an error status.

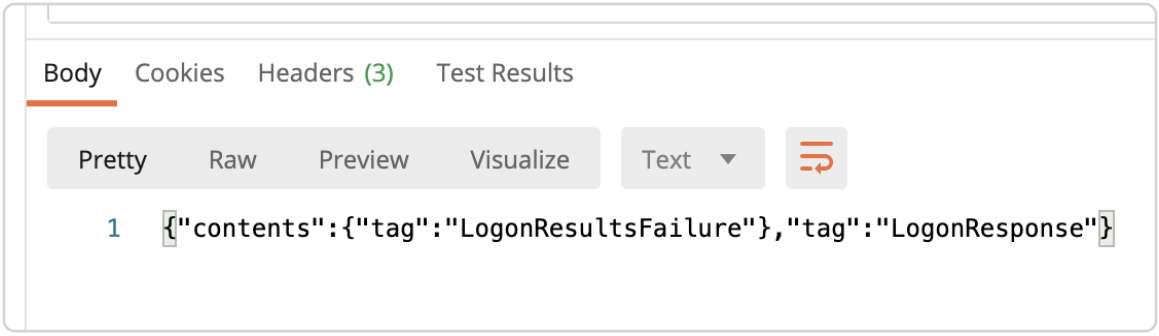
I get the following when I do that:



So our new code works. Now change `Postman` back to use `POST`.

Next, lets test our code for the invalid logon case by changing the `userName` in `Postman` to `"invalid"` and sending again. What do you get?

I got:



Now, that just tests the `userName` being wrong. So, let's change it back to `admin` and now lets make just the `password` wrong by making it `"invalid"`. Then test again.

When I do that I get the same response. So, it appears that we have a working server. Well, partially working.

Don't forget to change the password back to `admin`.

It's important to test both the working cases as well as the failure cases. Too often in our rush to get a working system we'll forget to test the cases that should fail.

It's possible to have a bug in our logon logic that would accept bad `userNames` or `passwords`. Without testing the failure cases, we'd have no way to know this.

28.8. Coding the Session Manager

The last thing we need to do is to create a Session when a Logon is successful. This way subsequent API calls can be verified.

The Session Manager needs the following functionality:

- Create a Session (on Logon)
- Delete a Session (on Logoff)
- Verify Auth Token (every non-Logon API call)

We're going to implement the Session Manager similar to how we did the Account Manager. We're going to have a `startup` and `shutdown`.

The `startup` will create a `sessionsAVar` that'll reference a `Map` of `authTokens` as the keys and sessions as the values. The session will have a UTC time that will allow us to remove expired Sessions.

So, let's code those Functions first.

28.8.1. Code for `startup`, `shutdown` and `verifySession`

Let's see what we did with Account Manager and steal what we can:

```
startup :: Array Account -> Aff (AVar Accounts)
startup accounts = do
  accountsAVar <- AVar.empty
  (accounts
      <#> \account@(Account { userName }) -> Tuple userName account)
    # Map.fromFoldable
    # AVar.new

shutdown :: AVar Accounts -> Aff Unit
shutdown = void <<< AVar.take
```

Looks like we can steal all of `shutdown`, which reminds me that we never called Account Manager's shutdown. We should add that now, before I forget again.

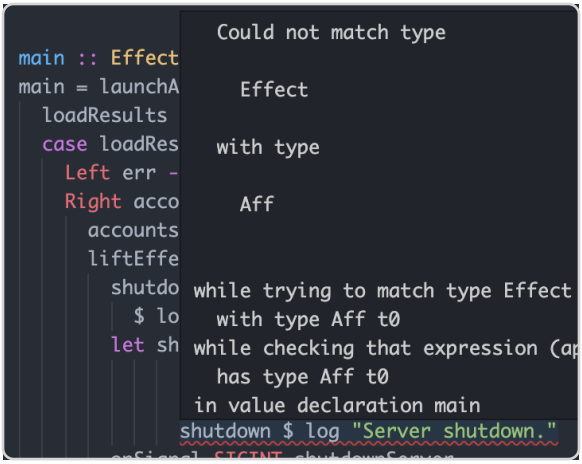
Go ahead and call `shutdown` in `main` BEFORE the server shuts down and then return here to compare your code with mine.

Here's my implementation of that:

```
main :: Effect Unit
main = launchAff_ do
  loadResults <- AccountHandler.loadAccounts
  case loadResults of
    Left err -> log $ "Cannot load accounts: " <> show err
    Right accounts -> do
      accountsAVar <- AccountManager.startup accounts
      liftEffect $ do
        shutdown <- HTTPure.serve port (router { accountsAVar })
          $ log $ "Server up running on port: " <> show port
        let shutdownServer = do
              log "Shutting down server..."
              AccountManager.shutdown accountsAVar ❶
              -- COMPILER ERROR!!
              shutdown $ log "Server shutdown."
        onSignal SIGINT shutdownServer
        onSignal SIGTERM shutdownServer
```

❶ Add the call to `shutdown`.

The compiler error:



Can you see why adding `AccountManager.shutdown` is giving us a Type mismatch?

This is because the `do` block for `shutdownServer` is running in `Effect` and `AccountManager.shutdown` is running in `Aff`.

Can you think of a way to fix this? If so, do so and when you have continue reading. And if not, please give yourself some time to think about it before giving up. Then read on.

`shutdownServer` needs to run in `Effect` since it's passed to `onSignal`, we have to keep it running in `Effect`. But the `do` block in `shutdownServer` can run in `Aff`.

We can accomplish this by using `launchAff_`. But then `shutdown` also runs in `Effect` and we can fix that by lifting into `Aff` with `liftEffect`:

```
main :: Effect Unit
main = launchAff_ do
  loadResults <- AccountHandler.loadAccounts
  case loadResults of
    Left err -> log $ "Cannot load accounts: " <> show err
    Right accounts -> do
      accountsAVar <- AccountManager.startup accounts
      liftEffect $ do
        shutdown <- HTTPure.serve port (router { accountsAVar })
          $ log $ "Server up running on port: " <> show port
        let shutdownServer = launchAff_ do ❶
              log "Shutting down server..." ❷
              AccountManager.shutdown accountsAVar
              liftEffect $ shutdown $ log "Server shutdown." ❸
        onSignal SIGINT shutdownServer
        onSignal SIGTERM shutdownServer
```

❶ Use `launchAff_` which runs an `Aff` in `Effect`. This keeps `shutdownServer` in `Effect`.

❷ `log` works in both `Effect` and `Aff` since we're using the `Effect.Class.Console` import.

❸ `shutdown` now runs in `Aff`. We can use `liftEffect` to lift `shutdown` into the current Monad, which is `Aff`.

Let's try to remember to return here once we've written `shutdown` for the Session Manager.

Now, create a File under `src/Manager` called `Session.purs` and call the module `Manager.Session`.

Then copy and paste `shutdown` from the Account Manager. Add whatever imports you need to make it compile. Right up until the point where `Accounts` is causing a compiler error. Then return here.

You're file should look like:

```
module Manager.Session where

import Prelude

import Effect.Aff (Aff)
import Effect.Aff.AVar (AVar)
import Effect.Aff.AVar as AVar

shutdown :: AVar Accounts -> Aff Unit -- COMPILER ERROR!! ❶
shutdown = void <<< AVar.take
```

❶ `Accounts` should be changed to `Sessions` as soon as we create the Entity.

We need to define `Sessions` like we did `Accounts`. Go see where `Account` is imported from in the Account Manager and then return here.

It's imported from `Entity.Account` which is in `src/Entity`. So create another File under the same directory called `Session.purs`. Then define a `newtype` for `Session` where it's a Record with `authToken`, i.e. `UUID`, the `userName` and UTC time called `lastTime`.

Use `Account.purs` in the same Folder as a template and go as far as you can until you finish or hit a brick wall.

Let's start with the module and it's imports:

```
module Entity.Session where

import Data.UUID (UUID)

newtype Session = Session
  { authToken :: UUID
  , userName :: String
  , lastTime :: ???
  }
```

At this point, we've got to figure out what Type is going to go where the `???` are. It's whatever Type the Function that gets the current system time returns. We're going to use this to timeout Sessions.

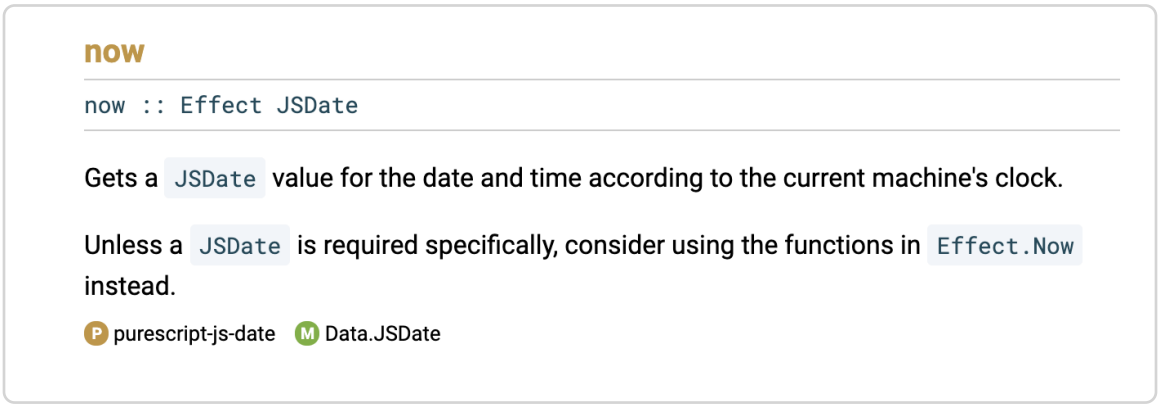
Go search `Pursuit` to see if you can figure out what Type to use here. Then return here and read on.

After a bunch of non-productive searches, on `Pursuit`, I finally gave up and searched for how to get the current time in Javascript and found `Date.now`. So I looked up `now` and after combing through a bunch of possible candidates, I gave up and did another search.

This time for `getTime`, which is another Javascript Function name. And I found:



This is great since it'll give us the number of milliseconds since the Epoch. But it wants a `JSDate`. Thanks, to my earlier perusal of the different `now` Functions, I remember seeing one:



Since `now` runs in `Effect`, we'll have to lift it to run it in `Aff`. But this should do nicely.

But what Type should `lastTime` be?

Looks like milliseconds from the Epoch will work just fine. So, it'll be a `Number`:

```
newtype Session = Session
  { authToken :: UUID
  , userName :: String
  , lastTime :: Number
  }
```

One thing to note here is that `userName` may seem a bit our of place here, but it's in here so we can cross reference a `Session` with its correpsonding `Account`, since `Accounts` have unique `userNames`. We're going to need this to check if the user of a particular `Session` is an `admin` or not since `admin` is stored in `Account`.

And now we can return to `Manager.Session` and fix the compiler error on `shutdown`:

```
shutdown :: AVar Accounts -> Aff Unit -- COMPILER ERROR!!
shutdown = void <<< AVar.take
```

First, add a Type Alias for `Sessions` using `Accounts` in `Manager.Account` as an example. Then replace `Accounts` with `Sessions` in `shutdown's` Type Signature in `Manager.Session`. Do this and then return here.

Here are those changes:

```
import Data.Map (Map) ❶
import Data.UUID (UUID)
import Entity.Session (Session)

type Sessions = Map UUID Session ❷

shutdown :: AVar Sessions -> Aff Unit ❸
shutdown = void <<< AVar.take
```

❶ Added these imports.

❷ Added this definition.

❸ Changed `Accounts` to `Sessions`.

Next, write `startup` for the Session Manager before continuing to read.

It's Type Signature is analogous to Account Manager's except that it doesn't start off with any existing `Sessions`. Granted, we could save out Sessions in a File upon a clean shutdown of the server and then read them in upon startup, but we're not going to in this design.

We'll start with the Type Signature:

```
startup :: Aff (AVar Sessions)
```

Then we need to create the empty AVar:

```
startup :: Aff (AVar Sessions)
startup = AVar.empty
```

And that's pretty much all we'll need to do. This is much simpler than the one in the Account Manager.

Looking at the Account Manager, we can see there's one more Function there that we don't have in Session Manager and that's a Verification Function, i.e. `verifyLogon`.

We're going to need a `verifySession` Function that will be called before any non-Logon API requests are honored.

Don't be afraid to steal from what we've already done with `verifyLogon`:

```
verifyLogon :: AVar Accounts -> String -> String -> Aff (Maybe Account)
verifyLogon accountsAVar userName password = do
  passwordHash' <- passwordHashHex userName password
  accounts <- AVar.read accountsAVar
  let account' = Map.lookup userName accounts
  pure $ (account' >>= \account@(Account { passwordHash })
      -> if passwordHash == passwordHash' then Just account else Nothing)
```

Take a second to review `verifyLogon`.

Now, go ahead and write `verifySession` in `Manager.Session`. Then return here when you're done or stuck.

First we start with the Type Signature. Like `verifyLogon`, it's going to need the `AVar`, but for the Session Store. Next, it'll need the `authToken` to verify, i.e. a Value of Type `UUID`. And just like `verifyLogon`, it may fail, so it'll return a `Maybe Session`:

```
import Data.Maybe (Maybe) ❶

verifySession :: AVar Sessions -> UUID -> Aff (Maybe Session)
```

❶ Import `Maybe`.

Next, we'll start the implementation:

```
verifySession :: AVar Sessions -> UUID -> Aff (Maybe Session)
verifySession sessionsAVar authToken = do
  ???
```

Next, we get the sessions from the `AVar`:

```
verifySession :: AVar Sessions -> UUID -> Aff (Maybe Session)
verifySession sessionsAVar authToken = do
  sessions <- AVar.read sessionsAVar ❶
  ???
```

❶ We're using `read` instead of `take` since we aren't going to update them.

Now, we can just look up the `Session` using the `authToken`:

```
import Data.Map as Map ❶

verifySession :: AVar Sessions -> UUID -> Aff (Maybe Session)
verifySession sessionsAVar authToken = do
  sessions <- AVar.read sessionsAVar
  pure $ Map.lookup authToken sessions ❷
```

❶ Import `Map`'s Functions qualified as `Map`.

❷ `Map.lookup` could fail with `Nothing`. This will get wrapped up in an `Aff` via `pure`.

Now that `verifySession` is done, we're going to need a way to add to sessions, i.e. create a `Session`.

We're going to write a Function called `createSession`. It'll create a new `Session` and add it to the `Sessions` store.

When we do, we want to make sure to use `take` and `put` so that no other `Fiber` can mess with the `Sessions` while we are.

Just write the Type Signature for `createSession` then return here when you're finished.

Here's the Type Signature:

```
createSession :: AVar Sessions -> String -> Aff UUID ❶ ❷ ❸
```

❶ We return the `UUID`, i.e. the `authToken`. This will ultimately be returned to the client to authenticate subsequent API calls.

❷ `String` is the `userName`.

❸ We're running in `Aff` since we are going to access the `AVar` among other `Aff` operations.

Then add the left-hand side of the equal sign starting a `do` block on the right:

```
createSession :: AVar Sessions -> String -> Aff UUID
createSession sessionsAVar userName = do
  ???
```

Since the order of what we do here doesn't matter too much, add code to get the current time. Look up the definitions of `getTime` and `now` on `Pursuit` to help you. Look up `getTime` first and then find the `now` that's compatible with it.

Add this code and then return here.

Here's that code:

```
import Data.JSDate (now, getTime) ❶
import Effect.Class (liftEffect)

createSession :: AVar Sessions -> String -> Aff UUID
createSession sessionsAVar userName = do
  lastTime <- getTime <$> liftEffect now ❷ ❸ ❹
  ???
```

❶ Added these imports.

❷ `now` runs in `Effect`, so we use `liftEffect` to run it in `Aff`.

❸ We map `getTime` over the `Aff JSDate` to get a `Aff Number` so that `lastTime` is the number of milliseconds since the 1970 Epoch.

❹ We call the Variable `lastTime` so it'll match the field in `Session` Record that we're going to create.

This one line could've been written as 2 lines in the `do` block, but coding with `<$>` eliminates having a Variable live for only one line. We don't have to name it nor do we have to maintain it unnecessarily.

Next, write the code to generate a `UUID` for the `authToken`. To do this, browse the `Data.UUID` module for a Function that will create a random `UUID`. Return here when your code is finished.

Here's that addition:

```
import Data.UUID (UUID, genUUID) ❶

createSession :: AVar Sessions -> String -> Aff UUID
createSession sessionsAVar userName = do
  lastTime <- getTime <$> liftEffect now
  authToken <- liftEffect genUUID ❷
  ???
```

❶ Added `genUUID` to import list.

❷ `genUUID` runs in `Effect` so we use `liftEffect`.

Now, write the code to add a new `Session` to `Sessions`. Remember to use `take` and `put` with `sessionsAVar`. Read on when that's complete.

First, we need the `Sessions` so we can `insert` a new Session:

```
createSession :: AVar Sessions -> String -> Aff UUID
createSession sessionsAVar userName = do
  lastTime <- getTime <$> liftEffect now
  authToken <- liftEffect genUUID
  sessions <- AVar.take sessionsAVar ❶
  ???
```

❶ Even though nothing is going to block us between the read and write of the `Sessions`, it's good to code as if something is. That's why we're calling `take`. This will block all other `Fibers`, i.e. other active API Requests to our server, until we're done modifying the `Sessions` structure.

And now we insert a new `Session` into `Sessions` and `put` that to the `AVar`:

```
import Entity.Session (Session(..)) ❶

createSession :: AVar Sessions -> String -> Aff UUID
createSession sessionsAVar userName = do
  lastTime <- getTime <$> liftEffect now
  authToken <- liftEffect genUUID
  sessions <- AVar.take sessionsAVar
  AVar.put (Map.insert authToken
    (Session { authToken, userName, lastTime }) sessions) sessionsAVar ❷
  ???
```

❶ Import `Session's` Data Constructors.

❷ Parenthesis are needed here around the `Map.insert` call. If you use `$` then `sessionAVar` will be passed

to `Map.insert` instead of `AVar.put`.

There's not much left to do, so finish the Function before reading on.

All that's left is to return the authToken:

```
createSession :: AVar Sessions -> String -> Aff UUID
createSession sessionsAVar userName = do
  lastTime <- getTime <$> liftEffect now
  authToken <- liftEffect genUUID
  sessions <- AVar.take sessionsAVar
  AVar.put (Map.insert authToken
    (Session { authToken, userName, lastTime }) sessions) sessionsAVar
  pure authToken ❶
```

❶ Return the `authToken` wrapped in `AFF`.

Now, we should write a Function called `expireSessions` to remove sessions that are past some timeout period:

```
sessionTimeout :: Number
sessionTimeout = 4.0 * 60.0 * 60.0 * 1000.0 -- 4 hours ❶
```

❶ Do not forget the decimal point since it's a `Number`.

Add this to your code.

Now, code `expireSessions` to remove any `Sessions` that have expired, i.e. when the difference between now and `lastTime` is more than `sessionTimeout`.

Consider using a `filter` from `Map` and don't forget to be safe and use `take` and `put` since we're going to be modifying the `AVar`.

Once you done, return here.

We'll start with the Type Signature:

```
expireSessions :: AVar Sessions -> Aff Unit
```

Notice that it returns `Unit`. That's because we're calling this only for its Side-effect of potentially deleting some old `Sessions`.

Next, we'll start the implementation:

```
expireSessions :: AVar Sessions -> Aff Unit
expireSessions sessionsAVar = do
  ???
```

And now we'll need the current time:

```
expireSessions :: AVar Sessions -> Aff Unit
expireSessions sessionsAVar = do
  now <- getTime <$> liftEffect now ❶
  ???
```

❶ Stole this from `createSession`.

Now, we need to `take` the `Sessions`:

```
expireSessions :: AVar Sessions -> Aff Unit
expireSessions sessionsAVar = do
  now <- getTime <$> liftEffect now
  sessions <- AVar.take sessionsAVar ❶
  ???
```

❶ Again, stole this from `createSession`.

We want to filter `sessions` when the difference between `now` and `lastTime` is greater than the `sessionTimeout`. If you haven't already, look up `filter` in `Data.Map` in `Pursuit`.

Then finish coding this unless you've already completed this, then just keep reading.

And finally, we filter and then put those results to the `AVar`:

```
expireSessions :: AVar Sessions -> Aff Unit
expireSessions sessionsAVar = do
  now <- getTime <$> liftEffect now
  sessions <- AVar.take sessionsAVar
  AVar.put
    (Map.filter
      (\(Session { lastTime }) -> now - lastTime > sessionTimeout) ❶ ❷
      sessions)
    sessionsAVar
```

❶ The filter Function.

❷ Destructure both `Session` and the underlying Record to get to `lastTime`.

Update your code to match this.

We should call this every time someone tries to verifies a `Session`. Also, every time someone successfully verifies a `Session`, we should update the `lastTime` since they clearly are still active. So essentially, API calls, will keep a `Session` alive.

First expire `Sessions` every time `verifySession` is called. Do this before moving on.

We'll expire sessions right before we read them:

```
verifySession :: AVar Sessions -> UUID -> Aff (Maybe Session)
verifySession sessionsAVar authToken = do
  expireSessions sessionsAVar ❶
  sessions <- AVar.read sessionsAVar
  pure $ Map.lookup authToken sessions
```

❶ Added call to `expireSessions`

Now, change your code to update the `lastTime` in the `Session` that was just verified. Since you're updating `Sessions`, you'll want to change `read` to `take` and when you've updated the `Session`, you'll want to use `put`.

Once you've done this or hit a road block, return back here.

We're going to rewrite `verifySession` staring with the first part:

```
verifySession :: AVar Sessions -> UUID -> Aff (Maybe Session)
verifySession sessionsAVar authToken = do
  expireSessions sessionsAVar
  sessions <- AVar.take sessionsAVar ❶
  ???
```

❶ Changed `read` to `take`.

Now, we'll get the current time:

```
verifySession :: AVar Sessions -> UUID -> Aff (Maybe Session)
verifySession sessionsAVar authToken = do
  expireSessions sessionsAVar
  sessions <- AVar.take sessionsAVar
  now <- getTime <$> liftEffect now ❶
  ???
```

❶ Stolen from `expireSessions`.

Next, we need to look up the `authToken` and then handle the error and success case separately. If you think you can finish from this point, please stop and do so. Return here when you're ready.

First, we'll look up the `authToken` and handle the error case:

```
import Data.Maybe (Maybe(..)) ❶

verifySession :: AVar Sessions -> UUID -> Aff (Maybe Session)
verifySession sessionsAVar authToken = do
  expireSessions sessionsAVar
  sessions <- AVar.take sessionsAVar
  now <- getTime <$> liftEffect now
  case Map.lookup authToken sessions of ❷
    Nothing -> AVar.put sessions sessionsAVar *> pure Nothing ❸ ❹
    ???
```

❶ Added `(..)` to include Data Constructors.

❷ Look up the `authToken`.

❸ If not found then simply put back the original `sessions`. Note the use of `*>`, aka `applySecond`.

❹ Return not valid, i.e. `Nothing`.

Next, we'll handle the success case by first computing the new Session with `lastTime` equal to `now`:

```
verifySession :: AVar Sessions -> UUID -> Aff (Maybe Session)
verifySession sessionsAVar authToken = do
  expireSessions sessionsAVar
  sessions <- AVar.take sessionsAVar
  now <- getTime <$> liftEffect now
  case Map.lookup authToken sessions of
    Nothing -> AVar.put sessions sessionsAVar *> pure Nothing
    Just (Session session) -> do ❶
      let newSession = Session $ session { lastTime = now } ❷
      ???
```

❶ Destructure using Pattern Matching to expose the underlying Record, i.e. `session`.

❷ Construct a new `Session` with `lastTime` set to current time, i.e. `now`. Note the use of `=` since we're using the Update Record syntax.

And now we need to update our `Map`, i.e. `Sessions`:

```
verifySession :: AVar Sessions -> UUID -> Aff (Maybe Session)
verifySession sessionsAVar authToken = do
  expireSessions sessionsAVar
  sessions <- AVar.take sessionsAVar
  now <- getTime <$> liftEffect now
  case Map.lookup authToken sessions of
    Nothing -> AVar.put sessions sessionsAVar *> pure Nothing
    Just (Session session) -> do
      let newSession = Session $ session { lastTime = now }
          newSessions = Map.insert authToken newSession sessions ❶
      ???
```

❶ Update the `Map` with `newSession`. This REPLACES the old Session Value since we're using the same `authToken` as the key.

And finally, we'll `put` the `newSessions` to the `AVar` and return the `newSession`:

```
verifySession :: AVar Sessions -> UUID -> Aff (Maybe Session)
verifySession sessionsAVar authToken = do
  expireSessions sessionsAVar
  sessions <- AVar.take sessionsAVar
  now <- getTime <$> liftEffect now
  case Map.lookup authToken sessions of
    Nothing -> AVar.put sessions sessionsAVar *> pure Nothing
    Just (Session session) -> do
      let newSession = Session $ session { lastTime = now }
          newSessions = Map.insert authToken newSession sessions
      AVar.put newSessions sessionsAVar ❶
      pure $ Just newSession ❷
```

❶ Update the `AVar` with the updated `Map`, i.e. `newSessions`.

❷ Return updated session, i.e. `newSession`.

Now, `verifySession` handles expiration and `Session` keep alives.

But, notice how we're calling `AVar.put` in 2 places. If we had forgotten to call it in the `Nothing` case, we'd have a pretty nasty bug. This bug could easily get past QA and wind up in Production.

Then on some fateful day months after we've deployed our server, a request from a session that expired would hang our server.

That's because we would have done an `AVar.take` WITHOUT a cooresponding `AVar.put`.

A much better approach would be to only have 1 `AVar.put`. One that both cases rely on.

I'm going to change the `case` expression to return a `Tuple` of our updated sessions along with the `Maybe` updated session we just verified:

```
import Data.Tuple (Tuple(..)) ❶

verifySession :: AVar Sessions -> UUID -> Aff (Maybe Session)
verifySession sessionsAVar authToken = do
  expireSessions sessionsAVar
  sessions <- AVar.take sessionsAVar
  now <- getTime <$> liftEffect now
  let Tuple newSessions newSession = ❷
        case Map.lookup authToken sessions of
          Nothing -> Tuple sessions Nothing ❸
          Just (Session session) -> ❹
            let newSession = Session $ session { lastTime = now }
                newSessions = Map.insert authToken newSession sessions in ❺
            Tuple newSessions (Just newSession) ❻
  AVar.put newSessions sessionsAVar ❼
  pure newSession ❽
```

❶ Added import.

❷ Our `case` expression returns a `Tuple` and is immediately destructured into `newSessions` and `newSession`.

❸ In the failure case, our old `sessions` is unchanged. We also return Nothing since we failed to find the `authToken`.

❹ We're no longer in a `do` block.

❺ Notice the use of the `in` keyword since we're no longer in a `do` block.

❻ In the success case, we've updated our `sessions` and our `session`. So we return those changed values in a `Tuple`.

❼ The ONE AND ONLY `AVar.put`. If we forget this line of code, we're going to catch our error every time we try do anything. This bug won't make it out of development and it definitely won't make it out of QA.

❽ Return our `Maybe Session`.

Change your `verifySession` to this far safer version before reading on.

Before we move on, we should run the server and make sure it still works:

```
npx spago run
```

I get this error:

```
[error] Some of your project files import modules from packages that are not
in the direct dependencies of your project.
To fix this error add the following packages to the list of dependencies in
your config:
- js-date
```

Fix this error and run it again.

Then go to `Postman` and send a `LogonRequest` again.

When I this, everything still works. Hopefully, yours does too.

28.9. Server Logging

Logging is the life blood of any Server. This is because they run unattended in a server room somewhere. We aren't going to be around when a problem rears its ugly head.

Logs lets us know what happened.

Granted in a real-world server, we'd spend far more time on Logging than we are going to do here, but this will still give you the basic idea.

Let's log each `Request` as it comes in. Take a look at the codebase and figure out the best place to do this before you read any further.

My first reaction was to modify the `router`, but then I realized making a Logging Router is so much better. This way we don't pollute our `router's` complex logic with logging.

A Logging Router will log the `Request` then call our current `router` and then log the `Response` before returning it. It's just a simple functional wrapper on `router`.

It also has to be a stand-in for our current `router` meaning that it'll have the same or, at least, a similar Type Signature.

Write as much of this as you can. Call it `loggingRouter` and put it in `Main`, then return when you're done or hit a brick wall.

Here's what I wrote:

```
loggingRouter :: HandlerEnv -> Request -> ResponseM
loggingRouter env req = do
  log $ "REQUEST: " <> show req ❶
  res <- router env req ❷
  log $ "RESPONSE: " <> show res -- COMPILER ERROR!! ❸
  pure res ❹
```

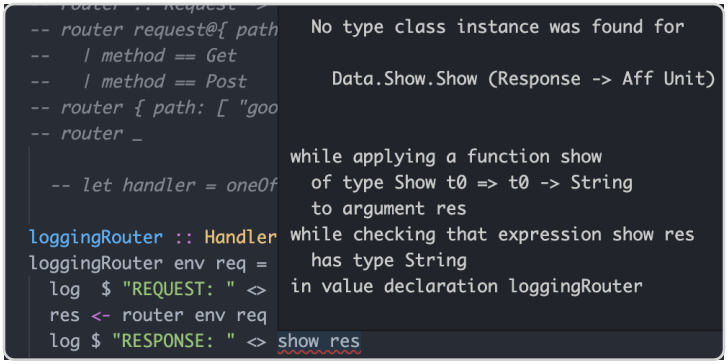
❶ Log the `Request`.

❷ Delegate to `router` capturing the `Response` in `res`.

❸ Log the `Response` (has compiler errors).

❹ Return `Response`.

Here's the compiler's complaint:



What do you think about this error? What should we do next?

When I see this, I ask the question, what Type is `Response`. When I look on `Pursuit` I find:

```
type Response =
  { status :: Status.Status
  , headers :: Headers.Headers
  , writeBody :: HTTP.Response -> Aff.Aff Unit ❶
  }
```

❶ This is our problem child.

That Function is NOT showable. What we'd like is a way to remove that field and then show the rest of the Record.

We can do this manually, but there's a pretty cool library that'll do this for any Record. It's called `purescript-record`. It has a module called `Record`. And there's a Function in that module called `delete` that'll create a new Record without the specified field.

This library can delete fields from a Record in a Type Safe manner. But it requires a bit of magic.

The first thing we're going to have to be able to do is specify the field name in the Types. We accomplish this by using a `Symbol` with `Proxy`.

Let's look at the kind `Symbol`:

Symbol

```
kind Symbol
```

`Symbol` is the kind of type-level strings.

Construct types of this kind using the same literal syntax as documented for strings.

`Symbol` lets us to use `String` literals in Type Definitions.

Let's first see how we use `Proxy` with `delete`:

```
log $ "RESPONSE: " <> (show $ delete (Proxy :: _ "writeBody") res) ❶ ❷
```

❶ `delete` field named `"writeBody"` from the `Response`, i.e. `res`.

❷ The `_` is figured out by the compiler to be the Type `Proxy`.

Here's a change to our logging using `delete` to pull out the non-existent field `writeBody___`. Since we specify the field name in the Type System, and that field doesn't exist, the following won't compile:

```
-- COMPILER ERROR!!
log $ "RESPONSE: " <> (show $ delete (Proxy :: _ "writeBody___") res)
```

It causes the following error:



This is because of the Constraints on `delete`:

```
delete
  :: ∀ proxy r1 r2 l a
  .  IsSymbol l ❶
  => Lacks l r1 ❶
  => Cons l a r1 r2 ❶
  => proxy l
  -> Record r2
  -> Record r1
```

❶ These Constraints are built into the compiler.

To understand this, first look at `proxy's` Type Parameter `l`, which stand for the `label` in Row `r2` that we're going to delete. It has an `IsSymbol` constraint, which means it must be a Literal String. You cannot use a Variable of Type `String` here.

The next 2 Constraints is where the Type-level magic happens.

First is `Lacks l r1`. This Constraint says that Row `r1`, which is the Row of the returned Record, must not contain the `Symbol`, `l`. This make sense since `delete` is supposed to remove that field and `Record r1` is returned. Remember, we can construct Records from Rows using the Data Constructor `Record`.

Second is `Cons l a r1 r2`. The `l` is our `Symbol` and represents a Label. The `a` is a Type and represents the Label's Type. Remember that Rows consists of Pairs of Labels and their corresponding Types:

```
type SomeRow = ( someLabel :: String, someOtherLable :: Int )
```

So, `Cons` is a Constraint that will make sure that Row `r1`, i.e. the labels after the deletion, cons'd with the Label, `l`, of Type `a` is equal to the Row, `r2`, i.e. the labels that we started with.

In other words, `Lacks` makes sure that the final returned Row, `r1`, doesn't already have the label we're going to delete and `Cons` makes sure that the only difference between the Rows `r1` and `r2` is that one label.

All this checking is done at the Type level, which is why it's Type Safe and why `writeBody___` fails to compile.

If you look at the implementation of `delete`, it simply calls Javascript to delete a field from a Javascript Object, which is the underlying representation of PureScript Records. This can be safely done because it's already been Type Checked by the compiler, so we know the key must be in the Object before Javascript tries to `delete` it.

For now, modify you `loggingRouter` to look like:

```
import Record (delete) ❶

loggingRouter :: HandlerEnv -> Request -> ResponseM
loggingRouter env req = do
  log $ "REQUEST: " <> show req
  res <- router env req
  log $ "RESPONSE: " <> (show $ delete (Proxy :: _ "writeBody") res) ❷
  pure res
```

❶ Add this import.

❷ This is the new line of code to log the `Response`.

Now, we could add lots of things to this Function. For example, we could add timestamps. When running in production with a Log Aggregator like Logify or DataDog, then you don't need to do this since those tools automatically timestamp your log lines for you.

In case you're not familiar with Log Aggregators, they typically collect the output your program writes to the console, i.e. `stdout` and `stderr` in Linux, and aggregate them along with the logs from other servers in your Enterprise into an online Data Store. Part of collecting the logs is to timestamp them upon collection. This allows them to present the logs in the proper order.

Another thing we could do is add a way to tie the `Request` to the `Response` so when we're digging though a huge log, with many simultaneous `Requests`, we'll be able to quickly find a `Request's` corresponding `Response`. We can do this pretty easily. So let's do it.

Generate a UUID and display it with both the `Request` and the `Response` log line. Look at the Session Manager where we generate a UUID and steal that code for this Function.

Return to reading when you've completed this task.

Here are the changes:

```
import Data.UUID (genUUID) ❶

loggingRouter :: HandlerEnv -> Request -> ResponseM
loggingRouter env req = do
  id <- liftEffect genUUID ❷
  let idStr = " (" <> show id <> ")" ❸
  log $ "REQUEST: " <> show req <> idStr ❹
  res <- router env req
  log $ "RESPONSE: "
    <> (show $ delete (Proxy :: _ "writeBody") res) <> idStr ❹
  pure res
```

❶ Added this import.

❷ `genUUID` runs in `Effect`, so we use `liftEffect`.

❸ We compute the identifying `String` once.

❹ Append identifying `String` to both log lines.

And the final thing that's useful is to know is how responsive your server is by timing how long it takes us to process the request. This is also very easy. We simply get the current time in milliseconds from the Epoch as soon as we get the request and right before we respond. We then calculate the difference and display the number of milliseconds in the `Response` log line.

Look to see how we got the current time in milliseconds in the Session Manager to help you add this feature to `loggingRouter`. Once you've done that, continue here.

Here are those changes:

```
import Data.JSDate (now, getTime) ❶

loggingRouter :: HandlerEnv -> Request -> ResponseM
loggingRouter env req = do
  start <- getTime <$> liftEffect now ❷
  id <- liftEffect genUUID
  let idStr = " (" <> show id <> ")"
  log $ "REQUEST: " <> show req <> idStr
  res <- router env req
  end <- getTime <$> liftEffect now ❸
  log $ "RESPONSE: "
    <> (show $ delete (Proxy :: _ "writeBody") res)
    <> " [" <> show (end - start) <> " ms]" ❹
    <> idStr
  pure res
```

❶ Import time Functions.

❷ Get the start time as soon as possible.

❸ Get the end time as late as possible.

❹ Add the duration for the `Request` to be serviced to the `Response` log line.

Take a good look at our `loggingRouter`. Aren't you glad we wrote it as a separate Function?

Now, change `main` to use `loggingRouter` instead of `router` and rerun the server and test it with `Postman`.

Well, when I do this, I get the error:

```
[error] Some of your project files import modules from packages that are not
in the direct dependencies of your project.
To fix this error add the following packages to the list of dependencies in
your config:
- record
```

Fix this and rerun the server and test it with Postman.

You should see something like:

```
REQUEST: { body: "{ \"tag\": \"LogonRequest\",\n  \"contents\": {\n
\"userName\": \"admin\",\n    \"password\": \"admin\"\n  }\n}", headers:
accept: */*
accept-encoding: gzip, deflate, br
cache-control: no-cache
connection: keep-alive
content-length: 95
content-type: text/plain
host: localhost:3000
postman-token: 297f753b-8fb0-47a3-8ee3-49e0fb3890c5
user-agent: PostmanRuntime/7.26.3

, httpVersion: HTTP/1.1, method: Post, path: [], query: (fromFoldable []) }
((UUID 02d93824-585b-45f4-84bd-2ce4531d8289))
RESPONSE: { headers: Content-Length: 154

, status: 200 } [20.0 ms] ((UUID 02d93824-585b-45f4-84bd-2ce4531d8289)) ❶
```

❶ Notice that the `body` of the response is missing. That's because of how `Response` encodes the `writeBody` as a Function, that will write the body as a Side-effect. Because of `HTPPure's` design choice, there's no easy way for us to gain access to the body. So for this learning example, we're going to leave it out.

Now this is a pretty ugly log, but not bad for how little work we did. And all of the information is there for us to debug most problems. Definitely all the ones we'll have here.

The only thing that's missing is the timestamp and since we now have time logic in the code, adding this should be pretty easy. So, let's do it together.

First, we'll change `start` and `end` into `startDate` and `endDate` and then we'll calculate `start` and `end` from those Values:

```
loggingRouter :: HandlerEnv -> Request -> ResponseM
loggingRouter env req = do
  startDate <- liftEffect now ❶
  id <- liftEffect genUUID
  let idStr = " (" <> show id <> ")"
  log $ "REQUEST: " <> show req <> idStr
  res <- router env req
  endDate <- liftEffect now ❶
  let duration = getTime endDate - getTime startDate ❷
  log $ "RESPONSE: "
    <> (show $ delete (Proxy :: _ "writeBody") res)
    <> " [" <> show duration <> " ms]" ❸
    <> idStr
  pure res
```

❶ Removed mapping with `getTime` and renamed to add `Date` to the end of the Variable names.

❷ Calculate the duration from `startDate` and `endDate` using `getTime` here now.

❸ Use `duration` in the log line.

Next, we'll create a little helper to format the timestamp, called `ts`:

```
import Data.JSDate (now, getTime, toUTCString) ❶

loggingRouter :: HandlerEnv -> Request -> ResponseM
loggingRouter env req = do
  startDate <- liftEffect now
  id <- liftEffect genUUID
  let idStr = " (" <> show id <> ")"
      ts dt = "(" <> toUTCString dt <> ") " ❷
  log $ "REQUEST: " <> show req <> idStr
  res <- router env req
  endDate <- liftEffect now
  let duration = getTime endDate - getTime startDate
  log $ "RESPONSE: "
    <> (show $ delete (Proxy :: _ "writeBody") res)
    <> " [" <> show duration <> " ms]"
    <> idStr
  pure res
```

❶ Added `toUTCString` to the import list.

❷ `ts` will take a `JSDate`, i.e. `dt`, and produce a human readable UTC date and time by leveraging `toUTCString`.

And now all we have to do is prefix the log lines with the appropriate timestamp:

```
loggingRouter :: HandlerEnv -> Request -> ResponseM
loggingRouter env req = do
  startDate <- liftEffect now
  id <- liftEffect genUUID
  let idStr = " (" <> show id <> ")"
      ts dt = "(" <> toUTCString dt <> ") "
  log $ ts startDate <> "REQUEST: " <> show req <> idStr ❶
  res <- router env req
  endDate <- liftEffect now
  let duration = getTime endDate - getTime startDate
  log $ ts endDate <> "RESPONSE: " ❷
    <> (show $ delete (Proxy :: _ "writeBody") res)
    <> " [" <> show duration <> " ms]"
    <> idStr
  pure res
```

❶ Added `ts startDate` for the `Request`.

❷ Added `ts endDate` for the `Response`.

Update your code to match this and then rerun your server and test it with `Postman`. You should get something like:

```
(Sat, 22 May 2021 20:06:44 GMT) REQUEST: { body: "{ \"tag\":
\"LogonRequest\",\n  \"contents\": {\n    \"userName\": \"admin\",\n
\"password\": \"admin\"\n  }\n}", headers: accept: */*
accept-encoding: gzip, deflate, br
cache-control: no-cache
connection: keep-alive
content-length: 95
content-type: text/plain
host: localhost:3000
postman-token: ed7a8a01-8345-4087-853c-67fc47b4e27a
user-agent: PostmanRuntime/7.26.3

, httpVersion: HTTP/1.1, method: Post, path: [], query: (fromFoldable []) }
((UUID 3ff66d80-ebbe-4b8a-9119-5de7d8e8020b))
(Sat, 22 May 2021 20:06:44 GMT) RESPONSE: { headers: Content-Length: 154

, status: 200 } [18.0 ms] ((UUID 3ff66d80-ebbe-4b8a-9119-5de7d8e8020b))
```

28.10. Coding the API

Along the way, we've coded the API for the `LogonRequest` to get all of our server parts working. But now, it's time to finish the API.

We have the following API requests left to code:

- Create User
- Query Users
- Logoff

Let's start with the easiest one, `LogoffRequest`. This is a good time to do this since we just finished `LogonRequest`.

28.10.1. Coding `LogoffRequest` API Handler

Before we code this, we should look back at what we did for `Logon` to see what we can steal:

```
data Logon = Logon

handler :: LogonRequest -> Handler
handler (LogonRequest { userName, password }) = do
  { accountsAVar } <- ask
  verifiedAccount <- lift $ verifyLogon accountsAVar userName password
  HTTPure.ok $ encodeJSON $ case verifiedAccount of
    Nothing -> LogonResponse LogonResultsFailure
    Just (Account { temporaryPassword }) -> LogonResponse
      $ LogonResultsSuccess
        { authToken: emptyUUID -- TODO get from Session Mgr
        , mustChangePassword: temporaryPassword
        }

instance apiHandlerLogon :: ApiHandler Logon where
  handle request _ = do
    logonReq <- runExcept (decodeJSON request :: _ LogonRequest)
    pure $ handler logonReq
```

Take a look at the `ApiHandler` Instance. And see what we'll have to change to implement `Logoff's` instance. Make a mental list of the changes you'd make before reading on.

I could only see 3:

```
data Logon = Logon ❶

instance apiHandlerLogon :: ApiHandler Logon where ❶
  handle request _ = do
    logonReq <- runExcept (decodeJSON request :: _ LogonRequest) ❷ ❸
    pure $ handler logonReq ❷
```

❶ Replace `Logon` with `Logoff`.

❷ Rename `logonReq` to `logoffReq`.

❸ Change `LogonRequest` to `LogoffRequest`.

How did this compare to your list?

Now, we could take this code and copy/paste it to `Handler.Logoff`, but then we'll be doing the same thing over and over again with each and every API implementation.

Instead, let's factor this out. We'll incubate it in `Handler.Logon` to start and then we'll put it in an API utility module once we're done.

Before we write a general `handle` Function, we should decide what to call it. What's a good name?

Naming is one of the hardest things in programming. Thinking about what our Function does can help us come up with a good name. It's a common `handle` function. It also decodes, so we might want that in the name. We have to be careful not to bloat the name though.

Also, naming it too generically is also bad practice because it will communicate too little information. Here are some candidates:

```
commonHandle ❶
apiHandler ❷
handleApi
```

❶ Conveys too little information.

❷ We already have a Typeclass with this name, `ApiHandler`.

I think I like `handleApi` because I feel like we're delegating our duties off to this to Function to handle the API call. Let's go with that. It's quite possible that you liked one of the others for perfectly good reasons or, perhaps, you came up with a much better name.

If you decide to use it, you'll have to remember that I called mine `handleApi`.

Now, you're ready to code this helper Function. Put it in `Handler.Logon` and call it `handleApi`. Keep in mind that we're going to have to pass a Type to `handleApi` since our code has an explicit Type declaration:

```
instance apiHandlerLogon :: ApiHandler Logon where
  handle request _ = do
    logonReq <- runExcept (decodeJSON request :: _ LogonRequest) ❶
    pure $ handler logonReq
```

❶ In `handleApi`, we're going to make this Type declaration Polymorphic.

Hopefully, you remember how we've done this in the past. See how far you can get before reading on.

So, we're going to start with the Type Signature:

```
import Data.Either (Either)
import Foreign (MultipleErrors)
import Type.Proxy (Proxy)

handleApi
  :: ∀ a
  .  Proxy a
  -> String
  -> Either MultipleErrors Handler
```

Next, we'll code the left-hand side:

```
handleApi
  :: ∀ a
  .  Proxy a
  -> String
  -> Either MultipleErrors Handler
handleApi _ request = ???
```

If you got stuck on the Type Signature and this has helped unstick you, stop now and see if you can continue coding.

Return here when you're ready to move on.

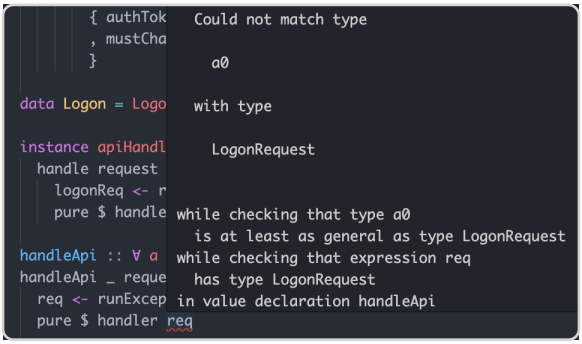
Now, we can just copy and paste the code from `Logon's` implementation for `handle` with some minor edits:

```
handleApi
  :: ∀ a
  . Proxy a
  -> String
  -> Either MultipleErrors Handler
handleApi _ request = do
  req <- runExcept (decodeJSON request :: _ a) ❶ ❷
  pure $ handler req -- COMPILER ERROR!! ❷
```

❶ We're using `a` since it's the Type of `Request` to decode.

❷ Renamed `logonReq` to a more generic `req`.

Now, the compiler error:



The problem is that we're incubating this Function in the same file with a `handler` Function. So the compiler is checking the Type for `req`, which is `a`, against the Type that `handler` has for its first Parameter, which is `LogonRequest`.

This is going to be troublesome. Let's move our code to a File caled `Common.purs` in `src/Handler/Api`.

Next, we'll copy the code we've written so far plus a few extra bits:

```
module Handler.Api.Common where

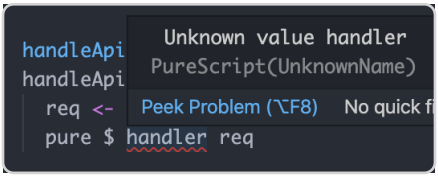
import Prelude

import Control.Monad.Except (runExcept)
import Data.Either (Either)
import Foreign (MultipleErrors)
import Foreign.Generic (decodeJSON)
import Handler.Class.ApiHandler (Handler)

handleApi
  :: ∀ a
  . Proxy a
  -> String
  -> Either MultipleErrors Handler
handleApi _ request = do
  req <- runExcept (decodeJSON request :: _ a)
  pure $ handler req -- COMPILER ERROR!!
```

Stop reading here and make sure you're code is caught up to this point and then return to keep reading.

We still have a compiler error but it's different in this file:



This tells us that `handler` is not defined. This handler Function will need to be passed in. Stop, reading here and change the Type Signature to include the `handler`. Then read on.

Here's the change:

```
handleApi
  :: ∀ a
  .  Proxy a
  -> (a -> Handler) ❶
  -> String
  -> Either MultipleErrors Handler
handleApi _ handler request = do ❶
  req <- runExcept (decodeJSON request :: _ a) -- COMPILER ERROR!!
  pure $ handler req
```

❶ Added `handler` as the second Parameter.

Realize that `handler` has the Type Signature `a -> Handler`. It's takes the Type of the request, e.g. `LogonRequest`, i.e. the same `a` we get from the `Proxy` Parameter. We can see that we're using that same `a` to help the compiler use the appropriate `decodeJSON` implementation.

If this still doesn't make sense, look at the Type Signature of `handler` in `Handler.Api.Logon`:

```
handler :: LogonRequest -> Handler ❶
```

❶ In this case, `a` is `LogonRequest`.

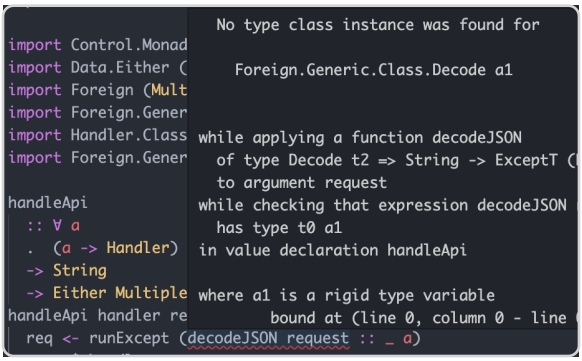
Now that we have an `a` in the Type Signature, we don't need `Proxy` anymore. Go ahead and remove it from the Type Signature before reading further.

Here's that change:

```
handleApi
  :: ∀ a
  .  (a -> Handler) ❶
  -> String
  -> Either MultipleErrors Handler
handleApi handler request = do ❶
  req <- runExcept (decodeJSON request :: _ a) -- COMPILER ERROR!!
  pure $ handler req
```

❶ Removed the `Proxy` Parameter.

Now, we can turn our attention to the compiler error:



This tells us that we need to constrain `a` to be decodable:

```
import Foreign.Generic.Class (class Decode) ❶

handleApi
  :: ∀ a
  .  Decode a ❷
  => (a -> Handler)
  -> String
  -> Either MultipleErrors Handler
handleApi handler request = do
  req <- runExcept (decodeJSON request :: _ a)
  pure $ handler req
```

❶ Import `Decode` Typeclass.

❷ Added Constraint.

And now it compiles.

Now, we can import our `handleApi` into `Handler.Api.Logon`:

```
import Handler.Api.Common (handleApi)
```

Add this to you code. Make sure you put it in the correct module.

Next, reimplement the `ApiHandler` instance using this Function. Then return back here.

Here's that rewrite:

```
import Foreign.Generic (encodeJSON) ❶

instance apiHandlerLogon :: ApiHandler Logon where
  handle request _ = handleApi handler request ❷
```

❶ Removed `decodeJSON` from import list.

❷ The Type Signature for `handler` dictates the Type that `handleApi` will use to decode the JSON.

Also, you can delete:

```
import Control.Monad.Except (runExcept)
```

Now, it's unfortunate that we cannot make this Point-free. We could just use `flip`, but that would be ugly. But since it's a minor change, we can change the order of the Typeclass Method Parameters.

But if we do that, we're going to impact `router`. So, make those changes and then rewrite `handle` to be Point-free. Then continue reading.

Here is the change to `Handler.Class.ApiHandler`:

```
class ApiHandler :: ∀ k. k -> Constraint
class ApiHandler a where
  handle :: Proxy a -> String -> Either MultipleErrors Handler ❶
```

❶ Flipped the Parameters from how they used to be.

Here is the change to `router` in `Main`:

```
router :: HandlerEnv -> Request -> ResponseM
router env { body, method }
  | method == HTTPure.Post =
      case hush =<<
        (head $ oneOf $ (handle (Proxy :: _ Logon) body) :| []) of ❶
          Nothing -> HTTPure.badRequest body
          Just reader -> runReaderT reader env
  | otherwise = HTTPure.methodNotAllowed
```

❶ Flipped the Parameters on the `handle` call.

And finally, here's the change to `Handler.Api.Logon`:

```
instance apiHandlerLogon :: ApiHandler Logon where
  handle _ = handleApi handler ❶
```

❶ This is now Point-free with `request` as the unnamed Parameter.

Now, let's look at what we can steal from `Logon` to code up `Logoff`:

```
data Logon = Logon

instance apiHandlerLogon :: ApiHandler Logon where
  handle _ = handleApi handler
```

This is very little code to copy/paste to our new `Logoff` module.

Create `Logoff.purs` under `src/Handler/Api` and then copy the above code from `Handler.Api.Logon` to our new file, which will be the module `Handler.Api.Logoff`. Don't forget to rename all instance of `Logon` to `Logoff` after you paste.

Then add imports until the compiler complains about `handler` not being defined, which is because we haven't written it yet. Do this before you read on.

Here's what I got:

```
module Handler.Api.Logoff where

import Prelude

import Handler.Class.ApiHandler (class ApiHandler)
import Handler.Api.Common (handleApi)

data Logoff = Logoff

instance apiHandlerLogff :: ApiHandler Logoff where
  handle _ = handleApi handler -- COMPILER ERROR!!
```

Now, we can write `handler` for the `LogoffRequest`. But before we do, we should refresh our memories on what `handler` does for `Logon`:

```
handler :: LogonRequest -> Handler
handler (LogonRequest { userName, password }) = do
  { accountsAVar } <- ask
  verifiedAccount <- lift $ verifyLogon accountsAVar userName password
  HTTPure.ok $ encodeJSON $ case verifiedAccount of
    Nothing -> LogonResponse LogonResultsFailure
    Just (Account { temporaryPassword }) -> LogonResponse
      $ LogonResultsSuccess
        { authToken: emptyUUID -- TODO get from Session Mgr ❶
        , mustChangePassword: temporaryPassword
        }
```

❶ This looks unfinished.

We need to finish this. Right now we're using an empty UUID for our authToken. Now that the Session Manager exists, we can call it to create a `Session` and return its `authToken` instead.

Before we delve into `Logoff`, change `Logon's` code in `Handler.Api.Logon` to call `createSession` from `Manager.Session` and use the `authToken` it gives us instead of `emptyUUID`. When you've completed that task, return here.

First, we add the import:

```
import Manager.Session (createSession)
```

Next, we need to look at the Type Signature of `createSession` to see what it needs:

```
createSession :: AVar Sessions -> String -> Aff UUID
```

It's clear that we need to an `AVar` for `Sessions` like we have for `Accounts`, which we can see in this line of code from `handler`:

```
  { accountsAVar } <- ask
```

`ask` gets a `HandlerEnv`, if my memory serves me. Do a global search of all the Files in your project in VSCode by using `Cmd-Shift-F` or `Ctrl-Shift-F` on Windows. And then type in `HandlerEnv` to find where it's defined. Then return here.

This is what I found in `Handler.Class.ApiHandler`:

```
type HandlerEnv =
  { accountsAVar :: AVar Accounts
  }
```

Looks like we never added a `sessionsAVar`. Do so now and then keep reading.

Here's that change:

```
import Manager.Session (Sessions) ❶

type HandlerEnv =
  { accountsAVar :: AVar Accounts
  , sessionsAVar :: AVar Sessions ❷
  }
```

❶ Import `Sessions`.

❷ Added `sessionsAVar`.

Now, we have to create this `AVar` like we did with `accountsAVar`. Search your project again for `accounstAVar` to see where we create it. Then keep reading.

I found it in `Main`:

```
main :: Effect Unit
main = launchAff_ do
  loadResults <- AccountHandler.loadAccounts
  case loadResults of
    Left err -> log $ "Cannot load accounts: " <> show err
    Right accounts -> do
      accountsAVar <- AccountManager.startup accounts ❶
      liftEffect $ do
        shutdown <- HTTPure.serve port (loggingRouter { accountsAVar }) ❷
          $ log $ "Server up running on port: " <> show port
        let shutdownServer = launchAff_ do
              log "Shutting down server..."
              AccountManager.shutdown accountsAVar
              liftEffect $ shutdown $ log "Server shutdown."
        onSignal SIGINT shutdownServer
        onSignal SIGTERM shutdownServer
```

❶ We get it from the `AccountManager`.

❷ And we pass it to our `loggingRouter`.

It's clear that we didn't startup or shutdown our Session Manager. I'm hoping that it gives us back an `AVar` like the Account Manager does. Go and check your Session Manager's `startup` code to see if it does and then return back here.

It sure does:

```
startup :: Aff (AVar Sessions)
startup = AVar.empty
```

Add the code to `main` to call `startup` and `shutdown` of the Session Manager before reading any further.

Here are those additions:

```
import Manager.Session as SessionManager

main :: Effect Unit
main = launchAff_ do
  loadResults <- AccountHandler.loadAccounts
  case loadResults of
    Left err -> log $ "Cannot load accounts: " <> show err
    Right accounts -> do
      accountsAVar <- AccountManager.startup accounts
      sessionsAVar <- SessionManager.startup ❶
      liftEffect $ do
        -- COMPILER ERROR!!
        shutdown <- HTTPure.serve port (loggingRouter { accountsAVar })
          $ log $ "Server up running on port: " <> show port
        let shutdownServer = launchAff_ do
              log "Shutting down server..."
              SessionManager.shutdown sessionsAVar ❷
              AccountManager.shutdown accountsAVar
              liftEffect $ shutdown $ log "Server shutdown."
        onSignal SIGINT shutdownServer
        onSignal SIGTERM shutdownServer
```

❶ Startup the Session Manager capturing the `sessionsAVar`.

❷ Shutdown the Session Manager. I like to shut things down in the opposite order than I start them up. In our case, order isn't important but it's a good habit to get into.

Now the compiler is complaining because the Record we're passing to `loggingReader` isn't the proper Type. That's becaues it's missing `sessionsAVar`, let's add it:

```
main :: Effect Unit
main = launchAff_ do
  loadResults <- AccountHandler.loadAccounts
  case loadResults of
    Left err -> log $ "Cannot load accounts: " <> show err
    Right accounts -> do
      accountsAVar <- AccountManager.startup accounts
      sessionsAVar <- SessionManager.startup
      liftEffect $ do
        shutdown <- HTTPure.serve port
          (loggingRouter { accountsAVar, sessionsAVar }) ❶
          $ log $ "Server up running on port: " <> show port
        let shutdownServer = launchAff_ do
              log "Shutting down server..."
              SessionManager.shutdown sessionsAVar
              AccountManager.shutdown accountsAVar
              liftEffect $ shutdown $ log "Server shutdown."
        onSignal SIGTERM shutdownServer
        onSignal SIGINT shutdownServer
```

❶ Added `sessionsAVar`.

Now, we can return to `handler` for `Logon` in `Handler.Api.Logon`:

```
handler :: LogonRequest -> Handler
handler (LogonRequest { userName, password }) = do
  { accountsAVar } <- ask
  verifiedAccount <- lift $ verifyLogon accountsAVar userName password
  HTTPure.ok $ encodeJSON $ case verifiedAccount of
    Nothing -> LogonResponse LogonResultsFailure
    Just (Account { temporaryPassword }) -> LogonResponse
      $ LogonResultsSuccess
        { authToken: emptyUUID -- TODO get from Session Mgr ❶
        , mustChangePassword: temporaryPassword
        }
```

We want to call `createSession` with the `sessionsAVar`. But how can we get that? If you're not sure then look to see how we got `accountsAVar` for a clue before reading on.

Update `handler` to create a `Session` on the successful verification of the `Account` information provided by the user. Here's the Type Signature for `createSessions` as reference:

```
createSession :: AVar Sessions -> String -> Aff UUID
```

Also, to avoid some odd compiler errors since we changed `HandlerEnv`, run:

```
npx spago build
```

This may have errors, but now you can save `Handler.Api.Logon`. Hopefully, this will prevent the weird errors I encountered.

Don't read any further until you've updated `handler` to call `createSession` and used the returned `authToken` in the `LogonResultsSuccess`. This will be a multi-step process requiring some reorgainzation of the existing code a bit. So, expect a few bumps along the way.

First, we'll `ask` for `sessionsAVar`:

```
handler :: LogonRequest -> Handler
handler (LogonRequest { userName, password }) = do
  { accountsAVar, sessionsAVar } <- ask ❶
  verifiedAccount <- lift $ verifyLogon accountsAVar userName password
  HTTPure.ok $ encodeJSON $ case verifiedAccount of
    Nothing -> LogonResponse LogonResultsFailure
    Just (Account { temporaryPassword }) -> LogonResponse
      $ LogonResultsSuccess
        { authToken: emptyUUID -- TODO get from Session Mgr
        , mustChangePassword: temporaryPassword
        }
```

❶ Added `sessionsAVar`. We will use this with `createSession`.

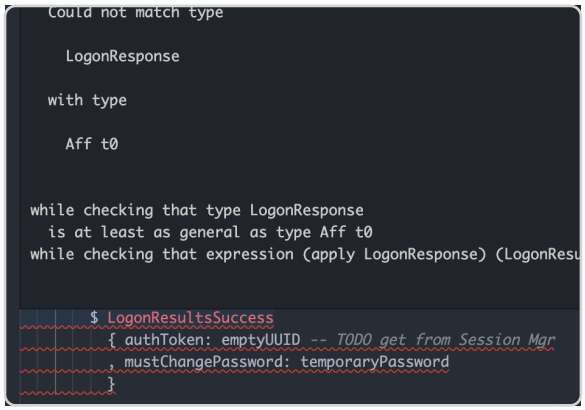
Next, we create a `Session` with `createSession`, which will return `Aff UUID`. We'll do this ONLY if the verification is successful:

```
handler :: LogonRequest -> Handler
handler (LogonRequest { userName, password }) = do
  { accountsAVar, sessionsAVar } <- ask
  verifiedAccount <- lift $ verifyLogon accountsAVar userName password
  HTTPure.ok $ encodeJSON $ case verifiedAccount of
    Nothing -> LogonResponse LogonResultsFailure
    Just (Account { temporaryPassword }) -> do ❶
      authToken <- createSession sessionsAVar userName ❷
      -- COMPILER ERROR!!
      LogonResponse
        $ LogonResultsSuccess
          { authToken: emptyUUID -- TODO get from Session Mgr
          , mustChangePassword: temporaryPassword
          }
```

❶ Add a `do` block to do Monadic Computations.

❷ Create the session capturing the `authToken`.

The compiler error:



Look carefully at the code we have so far. Notice how the `case` expression evaluates to a `Response` that's encoded and then sent to the client via `HTTPure.ok`.

But, we've just changed one of those `cases` to be an `Aff`. That's why the compiler is complaining.

The easiest way to fix this, is make BOTH `cases` an `Aff`. Stop here and fix this, then continue reading.

We'll fix this problem by adding `pure` to the right-hand side of the `Nothing` case and the last line of the `do` block in the `Just` case, since I forgot it earlier:

```
handler :: LogonRequest -> Handler
handler (LogonRequest { userName, password }) = do
  { accountsAVar, sessionsAVar } <- ask
  verifiedAccount <- lift $ verifyLogon accountsAVar userName password
  -- COMPILER ERROR!!
  HTTPure.ok $ encodeJSON $ case verifiedAccount of
    Nothing -> pure $ LogonResponse LogonResultsFailure ❶
    Just (Account { temporaryPassword }) -> do
      authToken <- createSession sessionsAVar userName
      pure $ LogonResponse ❷
        $ LogonResultsSuccess
          { authToken: emptyUUID -- TODO get from Session Mgr
          , mustChangePassword: temporaryPassword
          }
```

❶ Added `pure` here to make the `Nothing` case an `Aff LogonResonse`.

❷ I forgot to add `pure` here earlier. Now the `Just` case also returns an `Aff LogonResponse`.

The compiler error you're getting now should make sense without having to investigate the error message. We changed our `case` expression to be an `Aff`, which is NOT what `encodeJSON` expects.

To fix this, we're going to have to bind the `case` expression to `response`, then we can pass it to `HTTPure.ok $ encodeJSON`. Make this fix before reading the solution.

Here's that fix:

```
handler :: LogonRequest -> Handler
handler (LogonRequest { userName, password }) = do
  { accountsAVar, sessionsAVar } <- ask
  verifiedAccount <- lift $ verifyLogon accountsAVar userName password
  -- COMPILER ERROR!!
  response <- case verifiedAccount of ❶
    Nothing -> pure $ LogonResponse LogonResultsFailure
    Just (Account { temporaryPassword }) -> do
      authToken <- createSession sessionsAVar userName
      pure $ LogonResponse
        $ LogonResultsSuccess
          { authToken: emptyUUID -- TODO get from Session Mgr
          , mustChangePassword: temporaryPassword
          }
  HTTPure.ok $ encodeJSON response ❷
```

❶ Bind the results of the `case` expression to `response`.

❷ Moved the encoding of the `response` until after it's been bound. Then it's sent to the client via `HTTPure.ok`.

But we have a nasty compiler error with a useless error message:

```
Could not match type

  Aff

with type

  t0 Aff


while trying to match type Aff t2
  with type t0 Aff t1
while checking that expression (bind (case verification of
                                        Nothing -> ...
                                        (Just (Account { temp
  response <- case verification of
    Nothing -> pure $ LogonResponse LogonResultsFailure
    Just (Account { temporaryPassword }) -> do
      authToken <- createSession sessionsAVar
      pure $ LogonResponse
        $ LogonResultsSuccess
            { authToken: emptyUUID -- TODO get from Session Mgr
            , mustChangePassword: temporaryPassword
            }
  HTTPure.ok $ encodeJSON response
```

The only thing I can surmise is that we have a mismatch with our Monads. That tells me that I probably forgot to `lift` something. But what?

Let's look at `Handler`, which is the Monad we are running in, to see if we can see where I went wrong:

```
type Handler = ReaderT HandlerEnv Aff Response
```

So `Handler` is a Monad Stack with `ReaderT` on top and `Aff` on the bottom. We're calling `createSession`, which runs in `Aff`. That'll need to be lifted once to get it to the top of the Stack.

In fact, if we look back at the `verifyLogon` call, we can see that we're lifting it. Here's it's Type Signature:

```
verifyLogon :: AVar Accounts -> String -> String -> Aff (Maybe Account)
```

It too runs in `Aff`.

Let's fix my mistake by lifting `createSession`:

```
handler :: LogonRequest -> Handler
handler (LogonRequest { userName, password }) = do
  { accountsAVar, sessionsAVar } <- ask
  verifiedAccount <- lift $ verifyLogon accountsAVar userName password
  response <- case verifiedAccount of
    Nothing -> pure $ LogonResponse LogonResultsFailure
    Just (Account { temporaryPassword }) -> do
      authToken <- lift $ createSession sessionsAVar userName ❶
      pure $ LogonResponse
        $ LogonResultsSuccess
          { authToken: emptyUUID -- TODO get from Session Mgr
          , mustChangePassword: temporaryPassword
          }
  HTTPure.ok $ encodeJSON response
```

❶ Lifted `createSession`.

That made the compiler happy, but we're not done. We still need to use the returned `authToken` in our successful response to the client. If you haven't finished yet, please do so and then return back here.

Here's that final change:

```
handler :: LogonRequest -> Handler
handler (LogonRequest { userName, password }) = do
  { accountsAVar, sessionsAVar } <- ask
  verifiedAccount <- lift $ verifyLogon accountsAVar userName password
  response <- case verifiedAccount of
    Nothing -> pure $ LogonResponse LogonResultsFailure
    Just (Account { temporaryPassword }) -> do
      authToken <- lift $ createSession sessionsAVar userName
      pure $ LogonResponse
        $ LogonResultsSuccess
          { authToken ❶
          , mustChangePassword: temporaryPassword
          }
  HTTPure.ok $ encodeJSON response
```

❶ Use the `authToken` from the Session Manager.

Now delete:

```
import Data.UUID (emptyUUID)
```

Now, were we before all of this...

Oh, yeah. Working on `handler` for `Logoff`.

`Logon's` implementation may be useful as reference, but we really should have a plan. `Logoff` has to do the following:

- Verify authToken
- Reply with an error if invalid
- Otherwise, delete the Session
- And reply to the client that Logoff was successful

Are there any Technological Hurdles here?

Well, we can verify the `authToken` with `verifySession` in the Session Manager.

And we know how to reply to the client since we've done that in `Logon`, but how do we delete a `Session`?

Go look at the code for the Session Manager and see how, then return here.

It turns out that we cannot, yet. So, we'll need to write a Function in `Manager.Session` to do that. We're going to call it `deleteSession`. Hint: We're going to pass the `sessionsAVar` and an `authToken` to delete the entry from the `Map`.

Code this Function in `Manager.Session`, and when you're done, return here.

First we'll write the Type Signature:

```
deleteSession :: AVar Sessions -> UUID -> Aff Unit
```

We'll need 2 things to delete a `Session`. First, we need the `AVar` for `Sessions` and second we'll need to know the `authToken`, which is Type `UUID`, which is the key to the `Map`.

Also, note that `deleteSession` returns `Aff Unit`. This is a clue that it will only have Side-effects.

Next, we'll name our Parameters:

```
deleteSession :: AVar Sessions -> UUID -> Aff Unit
deleteSession sessionsAVar authToken = do ❶
  ???
```

❶ Start a `do` block to run in `Aff`.

Next, we get the `AVar`:

```
deleteSession :: AVar Sessions -> UUID -> Aff Unit
deleteSession sessionsAVar authToken = do
  sessions <- AVar.take sessionsAVar ❶
  ???
```

❶ We gain exlusive ownership to this via `take` since we're going to modify it.

Now, we delete from the `Map`. Do you know how to do that? If not, go look it up on Pursuit before reading on.

Let's add the deletion from the `Map`:

```
deleteSession :: AVar Sessions -> UUID -> Aff Unit
deleteSession sessionsAVar authToken = do
  sessions <- AVar.take sessionsAVar
  Map.delete authToken sessions ??? ❶
```

❶ We're not done with this line of code.

`Map.delete` returns a new `Map`, which means we can `put` that back in the now empty `AVar`:

```
deleteSession :: AVar Sessions -> UUID -> Aff Unit
deleteSession sessionsAVar authToken = do
  sessions <- AVar.take sessionsAVar
  AVar.put (Map.delete authToken sessions) sessionsAVar ❶
```

❶ Added the `AVar.put` with our updated `Map`.

And now, finally, we can write the `handler` for `Logoff`. Here's our plan again:

- Verify authToken
- Reply with an error if invalid
- Otherwise, delete the Session
- And reply to the client that Logoff was successful

Use the `handler` code from `Logon` as a reference and this plan to code up `handler` in `Handler.Api.Logoff` then return here.

We'll start with a similar Type Signature as `Logon's`:

```
import Data.Api.Logoff (LogoffRequest(..), LogoffResponse(..)
                      , LogoffResults(..)) ❶
import Handler.Class.ApiHandler (class ApiHandler, Handler) ❷

handler :: LogoffRequest -> Handler
```

❶ Import all of these from `Logoff`. This matches what we imported in `Logon` and it's currently more than what we need at the moment. We'll clean it up at the end if need be.

❷ Added `Handler` to the import list.

Next, we'll work on the left-hand side of the implementation:

```
handler :: LogoffRequest -> Handler
handler (LogoffRequest { authToken }) = do ❶
  ???
```

❶ Destructure the contained Record to gain access to `authToken`.

The next step is to `ask` for the `sessionsAVar` from the `ReaderT`:

```
import Control.Monad.Reader.Class (ask) ❶

handler :: LogoffRequest -> Handler
handler (LogoffRequest { authToken }) = do
  { sessionsAVar } <- ask ❷
  ???
```

❶ Import `ask`.

❷ Destructure to get just the `sessionsAVar`. We don't need anything else from `HandlerEnv`.

The very first thing we should do is to make sure that this request is authentic, i.e. is not coming from a Hacker:

```
import Control.Monad.Trans.Class (lift) ❶
import Manager.Session (verifySession)

handler :: LogoffRequest -> Handler
handler (LogoffRequest { authToken }) = do
  { sessionsAVar } <- ask
  verifiedSession <- lift $ verifySession sessionsAVar authToken ❶ ❷
  ???
```

❶ Import `lift` and `verifySession`.

❷ Do not forget to `lift` since `verifySession` runs in `Aff` and we're running in `Handler`, which is a 2-layer Monad Stack with `Aff` at the base.

And now, we should respond accordingly if the verification fails. If you didn't get this far, then take a look at how we handle the verification cases in `Logon` and implement just the `Nothing` case before reading any further.

Here's the `Nothing` case:

```
import Data.Maybe (Maybe(..)) ❶

handler :: LogoffRequest -> Handler
handler (LogoffRequest { authToken }) = do
  { sessionsAVar } <- ask
  verifiedSession <- lift $ verifySession sessionsAVar authToken
  response <- case verifiedSession of ❷
    Nothing -> pure $ LogoffResponse LogoffResultsFailure ❸ ❹
    ???
```

❶ Import `Maybe` with its Data Constructors.

❷ This line of code was inspired by how we did things in the `handler` for `Logon`. We may not need it to be a `bind` and a `let` might suffice. Time will tell.

❸ Our response will be failure if the verification failed, i.e. the `authToken` doesn't represent a valid `Session`.

❹ Note the use of `pure` here. That's because of the `bind` on the `case` expression.

Next, if you haven't finished this, write the success case, which still needs to delete the `Session` and reply with a success response. Then return back here.

Here's the success case coded one step at a time:

```
handler :: LogoffRequest -> Handler
handler (LogoffRequest { authToken }) = do
  { sessionsAVar } <- ask
  verifiedSession <- lift $ verifySession sessionsAVar authToken
  response <- case verifiedSession of
    Nothing -> pure $ LogoffResponse LogoffResultsFailure
    Just _ -> ??? ❶
```

❶ `verifySession's` return value, i.e. `Maybe Session`, isn't important to us here.

On the success case, we'll need to delete the `Session`, so we'll start a `do` block for the `deleteSession` call:

```
handler :: LogoffRequest -> Handler
handler (LogoffRequest { authToken }) = do
  { sessionsAVar } <- ask
  verifiedSession <- lift $ verifySession sessionsAVar authToken
  response <- case verifiedSession of
    Nothing -> pure $ LogoffResponse LogoffResultsFailure
    Just _ -> do
      ???
```

Now, we can delete the `Session` using our newly minted `deleteSession` Function:

```
import Manager.Session (verifySession, deleteSession) ❶

handler :: LogoffRequest -> Handler
handler (LogoffRequest { authToken }) = do
  { sessionsAVar } <- ask
  verifiedSession <- lift $ verifySession sessionsAVar authToken
  response <- case verifiedSession of
    Nothing -> pure $ LogoffResponse LogoffResultsFailure
    Just _ -> do
      lift $ deleteSession sessionsAVar authToken ❷
      ???
```

❶ Added `deleteSession` to the import list.

❷ Don't forget to add `lift` since we're operating in `Handler` and `deleteSession` is in `Aff`. We do this for all the same reasons we lifted `verifySession` (see above).

And now, we can return a successful response:

```
handler :: LogoffRequest -> Handler
handler (LogoffRequest { authToken }) = do
  { sessionsAVar } <- ask
  verifiedSession <- lift $ verifySession sessionsAVar authToken
  response <- case verifiedSession of
    Nothing -> pure $ LogoffResponse LogoffResultsFailure
    Just _ -> do
      lift $ deleteSession sessionsAVar authToken
      pure $ LogoffResponse LogoffResultsSuccess
  ???
```

If you haven't finished yet, finish off this Function by sending the `response` to the client. Then return here

and continue reading.

Our final step is to send the `response` to the client:

```
import Foreign.Generic (encodeJSON) ❶
import HTTPure as HTTPure

handler :: LogoffRequest -> Handler
handler (LogoffRequest { authToken }) = do
  { sessionsAVar } <- ask
  verifiedSession <- lift $ verifySession sessionsAVar authToken
  response <- case verifiedSession of
    Nothing -> pure $ LogoffResponse LogoffResultsFailure
    Just _ -> do
      lift $ deleteSession sessionsAVar authToken
      pure $ LogoffResponse LogoffResultsSuccess
  HTTPure.ok $ encodeJSON response ❷
```

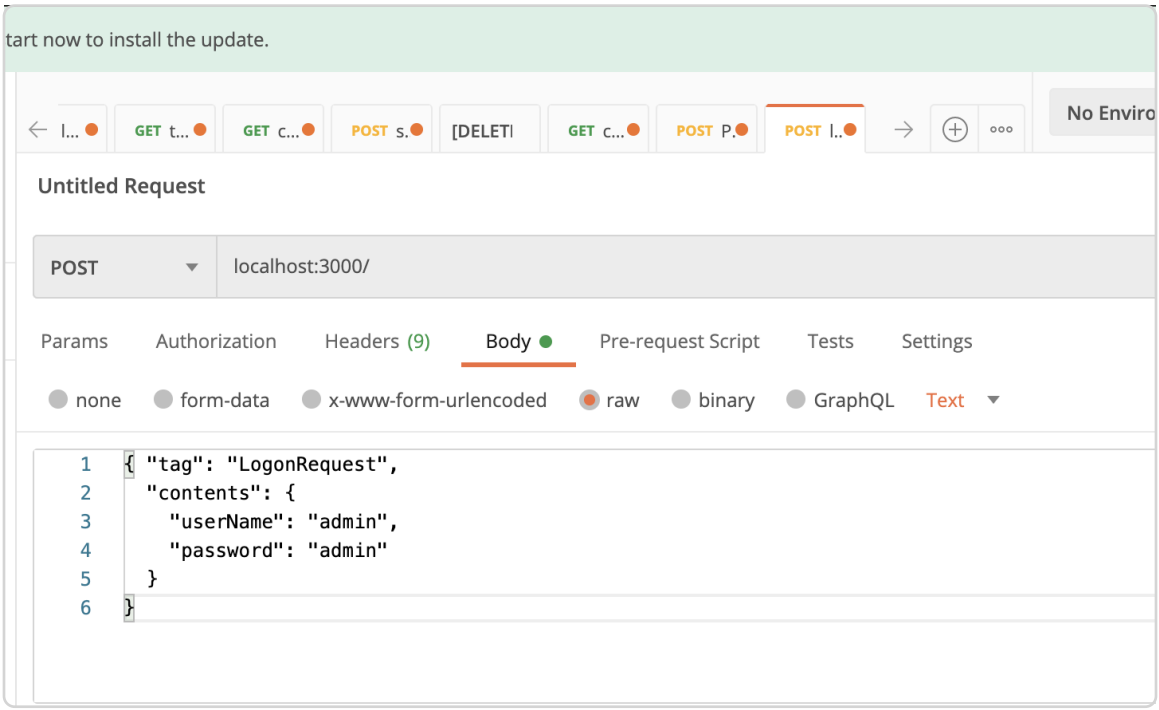
❶ Import `encodeJSON` and `HTTPure` qualified.

❷ We encode the response and reply with `HTTPure.ok`.

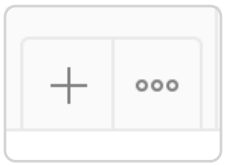
Now you can test this code using `Postman`. But first, we'll need to run the server:

```
npx spago run
```

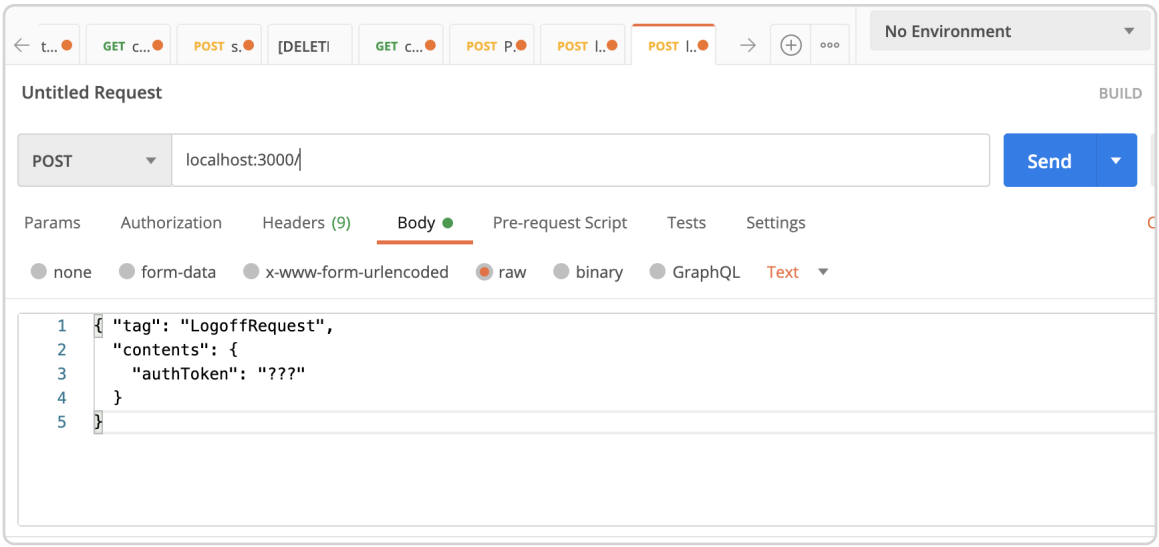
Before we can send a `Logofff`, we'll have to first send a `Logon` Request. Use the same settings from earlier:



Then click on the Plus Icon near the top of the window:



And create a `Logoff` Request:



The `???` is where we'll type in the `authToken` that our `Logon` request gets back.

So, send the Logon Request first and see what you get.

When I do that, `Postman` never gets a reponse. Looking at the output log of my server shows:

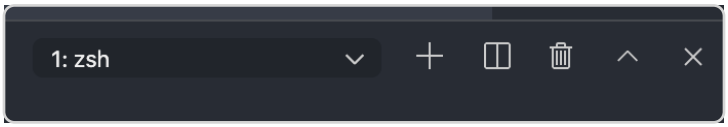
```
(Sat, 22 May 2021 20:06:44 GMT) REQUEST: { body: "{ \"tag\":
\"LogonRequest\",\n  \"contents\": {\n    \"userName\": \"admin\",\n
\"password\": \"admin\"\n  }\n}", headers: accept: */*
accept-encoding: gzip, deflate, br
cache-control: no-cache
connection: keep-alive
content-length: 95
content-type: text/plain
cookie:
_SESSION=t5EUhk1l5jzm09OhX7sFfbFRTnXOnhJfy/4tgHi6I5YNIKAIPpwAHtMcJrlvABVcCmNP4
fYouu+OitwDw6cs0evXSATaaxKBJTyzNxe9hT/oXJ0K51QlCMks5tgZCDklNPFHlP+AKJo=
host: localhost:3000
postman-token: 329006f6-ef6c-4f00-a57c-0c2eee27ad4e
user-agent: PostmanRuntime/7.26.3

, httpVersion: HTTP/1.1, method: Post, path: [], query: (fromFoldable []) }
((UUID b3839bb5-fdd2-4895-88e1-914168a2ac68))
```

Which shows a Request but no Response. What do you think I did wrong?

Well, knowing what our code does, I suspect I mishandled an `AVar` since it's the only thing that blocks the server.

By the way, killing the application with `Ctrl-C` doesn't work for me. I had to click on the Trash Can Icon in VSCode's Terminal window to kill the Terminal Session:



If you use `Ctrl-C` and it seems to kill the server but then you try rerunning the server and get an error that looks like:

```
[info] Build succeeded.
node:events:356
      throw er; // Unhandled 'error' event
      ^

Error: listen EADDRINUSE: address already in use 0.0.0.0:3000 ❶
    at Server.setupListenHandle [as _listen2] (node:net:1312:16)
    at listenInCluster (node:net:1360:12)
    at doListen (node:net:1498:7)
    at processTicksAndRejections (node:internal/process/task_queues:82:21)
Emitted 'error' event on Server instance at:
    at emitErrorNT (node:net:1339:8)
    at processTicksAndRejections (node:internal/process/task_queues:81:21) {
  code: 'EADDRINUSE',
  errno: -48,
  syscall: 'listen',
  address: '0.0.0.0',
  port: 3000
}
```

❶ The server port is still open, which means it's still running.

Then you can run the following command in Mac OSX or Linux:

```
ps -A | grep spago/run.js ❶
```

❶ In Windows, you'll have to kill the app using the Task Manager.

This will print out ALL of the apps that are running that have `spago` somewhere in the command line that it was run with.

You want to look for something like:

```
63392 ttys024    0:00.52 node .spago/run.js ❶
```

❶ You want the line that has `node .spago/run.js`.

The first number is the PID, i.e. the Process Id. Your number will be different. Using that PID, type the command:

```
kill -9 63392 ❶
```

❶ Your PID (Process Id), 63392 shown here, will be different. Use yours here.

Let's look at the `Logon` code to see where I may have messed up:

```
handler :: LogonRequest -> Handler
handler (LogonRequest { userName, password }) = do
  { accountsAVar, sessionsAVar } <- ask ❶
  verifiedAccount <- lift $ verifyLogon accountsAVar userName password
  response <- case verifiedAccount of
    Nothing -> pure $ LogonResponse LogonResultsFailure
    Just (Account { temporaryPassword }) -> do
      authToken <- lift $ createSession sessionsAVar userName
      pure $ LogonResponse
        $ LogonResultsSuccess
          { authToken
          , mustChangePassword: temporaryPassword
          }
  HTTPure.ok $ encodeJSON response
```

❶ We're using 2 different `AVars` in this Function.

Since either of the 2 `AVars` could be the problem, we'll just start with the `accountsAVar` first. We'll comment out the `verifyLogon` call, which uses the `accountsAVar` and replace it with a `let`:

```
handler :: LogonRequest -> Handler
handler (LogonRequest { userName, password }) = do
  { accountsAVar, sessionsAVar } <- ask
  let verifiedAccount = Nothing ❶
  -- verifiedAccount <- lift $ verifyLogon accountsAVar userName password ❷
  response <- case verifiedAccount of
    Nothing -> pure $ LogonResponse LogonResultsFailure
    Just (Account { temporaryPassword }) -> do
      authToken <- lift $ createSession sessionsAVar userName
      pure $ LogonResponse
        $ LogonResultsSuccess
          { authToken
          , mustChangePassword: temporaryPassword
          }
  HTTPure.ok $ encodeJSON response
```

❶ We always fail the verification.

❷ Commented out the suspected problem code.

Do this and run the server. Then send the `LogonRequest` again from `Postman` to see what happens.

I get:

```
{"contents":{"tag":"LogonResultsFailure"},"tag":"LogonResponse"}
```

This should tell us that there's something wrong with the way I'm handling the `AVars` in `verifyLogon`. Let's look at that code:

```
verifyLogon :: AVar Accounts -> String -> String -> Aff (Maybe Account)
verifyLogon accountsAVar userName password = do
  passwordHash' <- passwordHashHex userName password
  accounts <- AVar.read accountsAVar ❶
  let account' = Map.lookup userName accounts
  pure $ (account' >>= \account@(Account { passwordHash })
      -> if passwordHash == passwordHash' then Just account else Nothing)
```

❶ This is the only place where we're using the `AVar`.

Looks like `read` is the only thing that could possibly be blocking us. But when, if ever, does `read` block? Go look up `AVar` on `Pursuit` and then browse its page for `read` to see if it blocks.

When I did this, I found the sentence:

```
If the AVar is empty, this will queue until it is filled.
```

That is a veiled way of saying if the `AVar` is empty then we'll block.

So, why is our `accountsVar` empty? One possibility is that it's never filled, which we know it is since we ran our code earlier and the `LogonRequest` worked.

Another possibility is that it's emptied and it's never refilled. This could be because our code branches on some condition and I forgot to handle the refilling in one of the branches.

Search the codebase for everywhere that `accountsAVar` is used to see if you can find a `take` without a `put`. Then return here when you're done.

What did you find? I found nothing. Everything looks good. This is not too suprising since we've tested the Account Manager code in the past and it worked.

But, we've never tested the Session Manager. We should search the code for `sessionsAVar` and make sure it's created properly as well as handled properly in `verifySession` and `createSession`.

Return here after you're done investigating to see what I found.

Please make sure you take the time to do this investigation since it'll be good practice for debugging

problems.

Here's what I found:

```
startup :: Aff (AVar Sessions)
startup = AVar.empty
```

Unlike the Account Manager, which fills in its `AVar`, we have an empty `AVar`. This made sense when we wrote it but it's clear that my thinking was wrong.

What we want is an empty `Map` in the `AVar` NOT and empty `AVar`. So, look up `Map` on `Pursuit` and see if there's a way to create an empty `Map`. Then return here.

When I did this, I found `Map.empty`. So, now fix this code in `Manager.Session` before reading on.

If you wrote:

```
startup :: Aff (AVar Sessions)
startup = do
  aVar <- AVar.empty
  AVar.put Map.empty aVar
  pure aVar
```

That will work, but there's a more efficient way to do this by using `AVar.new`:

```
startup :: Aff (AVar Sessions)
startup = AVar.new Map.empty
```

Update your `startup` code to this.

Next, put `handler` in `Handler.Api.Logon` back to its original form, before we commented out the code:

```
handler :: LogonRequest -> Handler
handler (LogonRequest { userName, password }) = do
  { accountsAVar, sessionsAVar } <- ask
  verifiedAccount <- lift $ verifyLogon accountsAVar userName password ❶ ❷
  response <- case verifiedAccount of
    Nothing -> pure $ LogonResponse LogonResultsFailure
    Just (Account { temporaryPassword }) -> do
      authToken <- lift $ createSession sessionsAVar userName
      pure $ LogonResponse
        $ LogonResultsSuccess
          { authToken
          , mustChangePassword: temporaryPassword
          }
  HTTPure.ok $ encodeJSON response
```

❶ Deleted the `let` code to always fail the verification.

❷ Uncommented this line of code.

Now, rerun the server and try testing the `LogonRequest` again to see what happens now. Don't forget to re-build and run the server.

Here's what I get:

```
{"contents":{"contents":{"mustChangePassword":true,"authToken":"8b92d93d-4d5f-
484e-bd51-db21f7870750"},"tag":"LogonResultsSuccess"},"tag":"LogonResponse"}
```

Your `authToken` will be different than mine. Take this `authToken` and go back to the `LogoffRequest` tab in `Postman` and edit the `???` in the body replacing it with your `authToken`.

Here's my edit:

```
{ "tag": "LogoffRequest",
  "contents": {
    "authToken": "8b92d93d-4d5f-484e-bd51-db21f7870750"
  }
}
```

Now, click `Send` and see what happens.

I get:

```
{ "tag": "LogoffRequest",
  "contents": {
    "authToken": "8b92d93d-4d5f-484e-bd51-db21f7870750"
  }
}
```

Why is our server sending us back our Request?

Let's look at our `router` in `Main`:

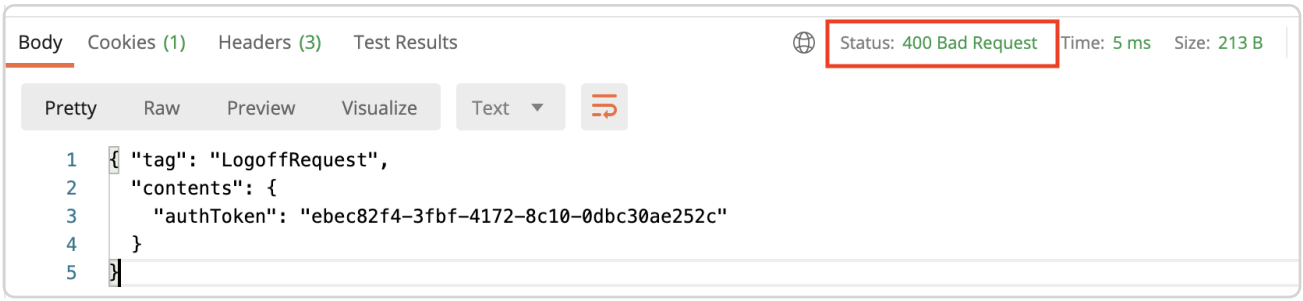
```
router :: HandlerEnv -> Request -> ResponseM
router env { body, method }
  | method == HTTPure.Post =
      case hush =<<
          (head $ oneOf $ (handle (Proxy :: _ Logon) body) :| []) of
        Nothing -> HTTPure.badRequest body ❶
        Just reader -> runReaderT reader env
  | otherwise = HTTPure.methodNotAllowed ❷
```

❶ Respond with `HTTP 400` when we cannot decode the `Request`.

❷ Respond with `HTTP 404` when not a `POST`.

We can see that we send `400` and the echo back the `Request` when we cannot decode the `Request`. It's clear we forgot to update this for `Logoff`, but how did we miss that this was an error?

Looks like it's there in `Postman` just hard to notice:



And if I would have looked at the logs I would have seen it there in our `Response`:

```
(Sat, 22 May 2021 22:02:55 GMT) RESPONSE: { headers: Content-Length: 103

, status: 400 } [1.0 ms] ((UUID 8b92d93d-4d5f-484e-bd51-db21f7870750))
```

Well, next time we'll know where to look.

Now, update `router` in `Main` to include `Logoff` by adding to the `NonEmptyList` before reading on.

Here's the change:

```
import Handler.Api.Logoff (Logoff) ❶

router :: HandlerEnv -> Request -> ResponseM
router env { body, method }
  | method == HTTPure.Post =
      let handlers = ❷
            handle (Proxy :: _ Logon) body :| [
              handle (Proxy :: _ Logoff) body ❸
            ] in ❹
      case hush =<< (head $ oneOf handlers) of ❷
        Nothing -> HTTPure.badRequest body
        Just reader -> runReaderT reader env
  | otherwise = HTTPure.methodNotAllowed
```

❶ Import `Logoff`.

❷ Factored out the `handlers` to help in the readability, especially since we're going to be adding more.

❸ Added `Logoff`.

❹ Don't forget the `in` keyword since we are NOT in a `do` block.

Before we rush off and run our code, let's factor out `body` from each `handler` entry, e.g. `handle (Proxy :: _ Logon) body`.

Try to factor it out by mapping over `handlers` with a Lambda or better yet an Anonymous Function, i.e. one with a wildcard. Return here when you're done.

Here's that improvement:

```
router :: HandlerEnv -> Request -> ResponseM
router env { body, method }
  | method == HTTPure.Post =
      let handlers =
            handle (Proxy :: _ Logon) :| [ ❶
            handle (Proxy :: _ Logoff) ❶
          ] <#> (_ $ body) in ❷
      case hush =<< (head $ oneOf handlers) of
        Nothing -> HTTPure.badRequest body
        Just reader -> runReaderT reader env
  | otherwise = HTTPure.methodNotAllowed
```

❶ Removed the second Parameter `body`.

❷ Using flipped `map`, i.e. `<#>`, to apply the Function from the `NonEmptyList` to `body`.

This improvement is a minor, but as we add more and more, the repetition of `body` would become noise.

A quick question before we run our program again: Why can't we factor out `handle`?

If we did, then we'd have a `NonEmptyList` with 2 `Proxies` of differing Types. `handle` homogenizes them.

Okay, run the server then send a `LogonRequest` and copy and paste the `authToken` from the `LogonResponse` to the `LogoffRequest` and send that to see what happens. Then return here.

I got the same thing, `Bad Request`. I suspect we are providing a malformed JSON structure for the `LogoffRequest`.

Add `debug` to your `spago.dhall` and run:

```
npx spago build
```

Ignore the warning printed:

```
[warn] None of your project files import modules from some projects that are
in the direct dependencies of your project.
These dependencies are unused. To fix this warning, remove the following
packages from the list of dependencies in your config:
- debug
```

Then make the following changes to `Main`:

```
import Debug (spy)

router :: HandlerEnv -> Request -> ResponseM
router env { body, method }
  | method == HTTPure.Post =
      let handlers =
              handle (Proxy :: _ Logon) :| [
              handle (Proxy :: _ Logoff)
            ] <#> (_ $ body) in
      case hush =<< (head $ spy "req" $ oneOf handlers) of ❶
        Nothing -> HTTPure.badRequest body
        Just reader -> runReaderT reader env
  | otherwise = HTTPure.methodNotAllowed
```

❶ Notice how the `spy "req"` is BEFORE the `head` call. If we put it after, then we'd only get the first decoding error printed in the console. That one would be the failed decoding for `Logon`, which we expect to fail when we send a `Logoff`. By putting it here, we will get all the errors.

Kill the server and rerun your program. This time just send the `LogoffRequest` from `Postman` since it won't matter what the `authToken` since we suspect that we're failing due to a decode issue. Then return here when you have your results.

Here's what I get:

```
req: [ Left {
    value0:
     NonEmpty {
       value0:
        ErrorAtProperty {
          value0: 'tag',
          value1: ForeignError { value0: 'Expected "LogonRequest" tag' } },
       value1: Nil {} } }, ❶
  Right { value0: [Function] } ] ❷
```

❶ We don't expect the `Logon` decoder to work and it did not.

❷ Looks like the `Logoff` decoder worked.

If the decoding worked then what's the problem. Take a look at the code for `router` and see if you can see anything else wrong with it.

After studying the `router` code, I can see that using `head` is the wrong approach. Can you see why?

Using `head` was fine when all we had was 1 `handler`, but now, we've got to look for one success case out of all of the possible ones. Here's the Type Signature of `oneOf` to see why this is the case:

```
oneOf :: ∀ f a. Alternative f => NonEmpty f a -> f a ❶
```

❶ `oneOf` returns `f a`, which for us is `Array (Either MutlipleErrors Handler)`. What we really wanted was just the first successful `Either MutlipleErrors Handler`. I think I mistakenly thought that `f` was an `Either`. Had I considered it carefully, I would've realized that `Either` can't be the `Alternative` since it can't have a `Plus`, which is `Alternative's` Superclass.

It's really clear that `oneOf` was the wrong approach. We did this because we didn't couldn't use `oneOf` from `Data.Foldable` because `Either` doesn't have a `Plus` Instance. We really wanted the behavior of this Function, not the one we settled on.

Where I went wrong is not realizing that `oneOf` uses `<|>` for `f`, in our case `Array`, and not `a`, which in our case is `Either`. `<|>` for `Array` is simply concatenation. This not what we want.

To fix this, we're going to have to write our own `oneOf` that give us the behavior of the `oneOf` from `Data.Foldable`, but will ONLY work with `NonEmpty` foldables. This way we don't have to use the `Plus` constraint since out list will never be empty.

Let's write that together. Before we do, comment out the router code that uses the old `oneOf` just for now since we're going to break it and we don't want the compiler errors that our new code generates to be masked by the broken code in `router`:

```
router :: HandlerEnv -> Request -> ResponseM
router env { body, method }
  -- | method == HTTPure.Post =
  --     let handlers =
  --             handle (Proxy :: _ Logon) :| [
  --             handle (Proxy :: _ Logoff)
  --           ] <#> (_ $ body) in
  --     case hush =<< (head $ spy "req" $ oneOf handlers) of
  --       Nothing -> HTTPure.badRequest body
  --       Just reader -> runReaderT reader env
  | otherwise = HTTPure.methodNotAllowed
```

Now, we'll code `oneOf` starting as always with the Type Signature:

```
import Data.NonEmpty (NonEmpty, (:|)) ❶ ❷

oneOf :: ∀ a e f. NonEmpty f (Either e a) -> Either e a
```

❶ Added `NonEmpty` to the import list.

❷ Removed `oneOf` from the import list.

We have a `NonEmpty f` of `Either`s. In our case, a `NonEmpty Array` of Partially Applied calls to `handle`:

```
let handlers =
        handle (Proxy :: _ Logon) :| [
        handle (Proxy :: _ Logoff)
      ] <#> (_ $ body) in
```

Then we map the application of `body` over the `NonEmpty Array` leaving us with the results of trying to handle the request by each handler. Only 1 handler will succeed, if any, since the requests are unique.

It's these `Either`s that we'll want to fold `<|>` over. This will result in the FIRST, `Right`, i.e. the first successful result, e.g.:

```
foldl (<|>) [Left 1, Right "yes", Left 2] (Left 99) -- Right "yes" ❶
```

❶ `Left 99` is our initial value for the `foldl`.

This is equivalent to:

```
Left 99 <|> Left 1 <|> Right "yes" <|> Left 2 -- Right "yes"
```

But if we're going to do a `foldl`, we're going to need an initial value. Luckily, we're using `NonEmpty` which always has at least a `head` value.

We're now going to Pattern Match the `NonEmpty` to get the head and tail:

```
oneOf :: ∀ a e f. NonEmpty f (Either e a) -> Either e a
oneOf (x :| xs) = ???
```

Now, we want to fold `<|>` over the `Eithers`. This will pick the first `Right`:

```
import Control.Alt ((<|>)) ❶
import Data.Foldable (class Foldable, foldl)

oneOf :: ∀ a e f. Foldable f => NonEmpty f (Either e a) -> Either e a ❷
oneOf (x :| xs) = foldl (<|>) x xs ❸ ❹
```

❶ Added these imports.

❷ Have to constrain `f` to be `Foldable` now that we're folding over `f` from `NonEmpty`.

❸ `x` is the initial value, i.e. the head of the `NonEmpty`.

❹ From this line of code, it should be very clear why we need a `NonEmpty`. With an `Either`, there's no `Plus`. We cannot `foldl` over an empty `Array` of `Either` since there's no `empty` from `Plus` to use as our initial Value for the `foldl` Function. With `NonEmpty`, we always have a head, in this case `x`, as the initial Value.

As always, just because this Function compiles, it doesn't mean that we're done. Can you see any improvements we can make on `oneOf`?

I think we should generalize it by abstracting the `Either e` to be `t`, which has an `Alt` Instance:

```
import Control.Alt (class Alt, (<|>)) ❶

oneOf :: ∀ a t f. Alt t => Foldable f => NonEmpty f (t a) -> t a ❷ ❸
oneOf (x :| xs) = foldl (<|>) x xs ❸
```

❶ Added the `Alt` class to the import list.

❷ Added `Alt t` Constraint.

❸ Replaced `Either e` with `t`.

And now you can remove the following imports:

```
import Data.Array (head)
import Debug (spy)
```

Then uncomment the code in `router` and change the router as such:

```
router :: HandlerEnv -> Request -> ResponseM
router env { body, method }
  | method == HTTPure.Post =
      let handlers =
              handle (Proxy :: _ Logon) :| [
              handle (Proxy :: _ Logoff)
            ] <#> (_ $ body) in
      case hush $ oneOf handlers of ❶ ❷
        Nothing -> HTTPure.badRequest body
        Just reader -> runReaderT reader env
  | otherwise = HTTPure.methodNotAllowed
```

❶ We're now using our `oneOf`, which simplifies this line of code since it does everything we need. We still `hush` the `Either`.

❷ Don't forget to change the `bind` to `apply`, i.e. `$`.

Now, run the server and send the `LogoffRequest` and see what happens. Then return here.

Here's what I get:

```
{"contents":{"tag":"LogoffResultsFailure"},"tag":"LogoffResponse"}
```

And that's great. This is to be expected since we never logged on.

Now, send the `LogonRequest` first and then copy the `authToken` out of the `Response` and into the

`LogoffRequest` to see if it works completely now. Then keep reading.

I still get:

```
{"contents":{"tag":"LogoffResultsFailure"},"tag":"LogoffResponse"}
```

So, what else could be wrong? Take a look at `handler` for `Logoff` and see if you can see what conditions cause a `LogoffResultsFailure`. Then return here and continue reading.

When I did this investigation, I found that the `LogoffResultsFailure` is only sent to the client when the `Session` does not verify.

That tells me that there might be something wrong with `verifySession`. Take a look at it and see if you can see anything that might be wrong with it. Then keep reading.

Let's look at the implementation together:

```
verifySession :: AVar Sessions -> UUID -> Aff (Maybe Session)
verifySession sessionsAVar authToken = do
  expireSessions sessionsAVar ❶
  sessions <- AVar.take sessionsAVar
  now <- getTime <$> liftEffect now
  case Map.lookup authToken sessions of
    Nothing -> AVar.put sessions sessionsAVar *> pure Nothing
    Just (Session session) -> do
      let newSession = Session $ session { lastTime = now }
          newSessions = Map.insert authToken newSession sessions
      AVar.put newSessions sessionsAVar
      pure $ Just newSession
```

❶ Expires any inactive `Sessions`.

The first thing it does is to expire old `Sessions`. I wonder if this is a problem. Maybe our `Session` is expiring too soon.

Add `Debug` to `Manager.Session` and then modify `expireSessions` accordingly:

```
import Debug (spy) ❶

expireSessions :: AVar Sessions -> Aff Unit
expireSessions sessionsAVar = do
  now <- getTime <$> liftEffect now
  sessions <- AVar.take sessionsAVar
  AVar.put
    (spy "expired" $ Map.filter ❷
      (\(Session { lastTime }) -> now - lastTime > sessionTimeout)
      sessions)
    sessionsAVar
```

❶ Added import.

❷ Print all the remaining `Sessions` AFTER we've filtered them to remove old ones.

Then run the server again sending first a `LogonRequest` then a `LogoffRequest`. It won't matter what `authToken` you send with the `LogoffRequest` since all we care about is dumping out all of the `Sessions` in `expireSessions`.

After you send the `LogoffRequest`, search the console for `expired` and continue reading once you find it.

Here's what I get:

```
expired: Leaf {} ❶
```

❶ `Leaf {}` is the internal representation of an empty `Map`.

Well, it looks like we're expiring too early. Look at the code and see if you can see my logic error. Then return here.

I see it. The logic is backwards. We want to KEEP sessions that have been idle less than the `sessionTimeout` NOT more:

```
expireSessions :: AVar Sessions -> Aff Unit
expireSessions sessionsAVar = do
  now <- getTime <$> liftEffect now
  sessions <- AVar.take sessionsAVar
  AVar.put
    (spy "expired" $ Map.filter
      (\(Session { lastTime }) -> now - lastTime < sessionTimeout) ❶
      sessions)
    sessionsAVar
```

❶ Changed `>` to `<`.

Rerun the server and once again just sent a `LogoffRequest` to see what you get now.

I get:

```
expired: Two {
  value0: Leaf {},
  value1: '6698cd9d-def3-4aa8-a3ca-c4bc8b087fc8',
  value2:
   { authToken: '6698cd9d-def3-4aa8-a3ca-c4bc8b087fc8',
     lastTime: 1602174302071 },
  value3: Leaf {} }
```

That's more like it. Now, remove `import Debug` and the `spy` code. Also, remove `debug` from `spago.dhall`.

Then rerun the server and do a proper test by copying the `authToken` and see what happens now.

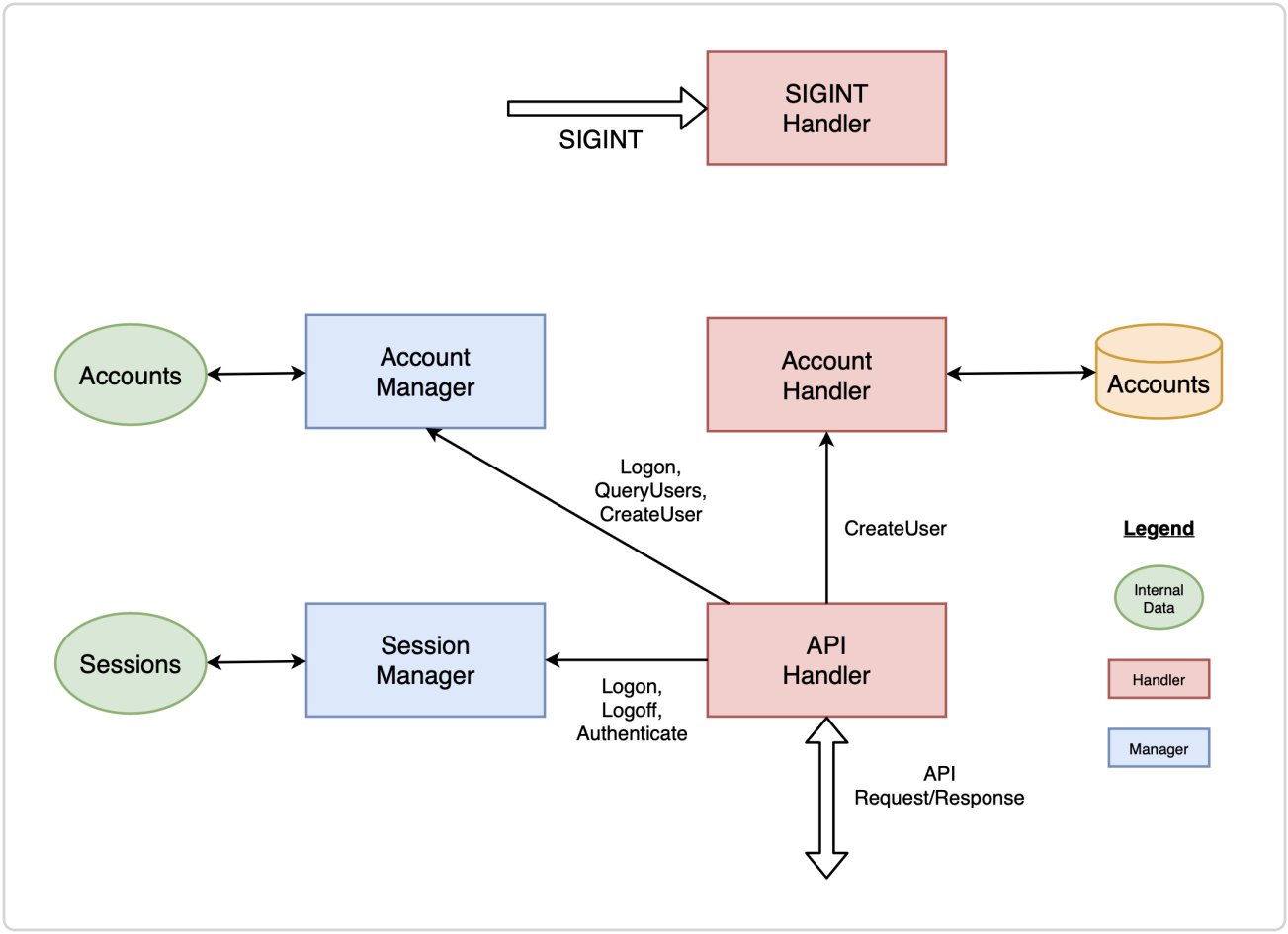
Here's what I get:

```
{"contents":{"tag":"LogoffResultsSuccess"},"tag":"LogoffResponse"}
```

Success!!

28.10.2. Coding `CreateUserRequest` API Handler

Let's look back at our Diagram paying particular attention to `CreateUser`:



We can see that `CreateUser` is marked on 2 Arrows. One from the API Handler to the Account Manager and one from the API Handler to the Account Handler.

The Account Handler will need to APPEND a record of the new `Account` to `accounts.csv`. And the Account Manager will need to add the new `User` to the `accountsAVar`.

Let's look at the differences between `Account` and `User` again:

```
type UserRow r =
  ( userName :: String
  , temporaryPassword :: Boolean
  , admin :: Boolean
  , firstName :: String
  , lastName :: String
   | r)

newtype User = User (Record (UserRow ()))

newtype Account = Account (Record (UserRow ( passwordHash :: String )))
```

So, `Account` is everything a `User` is plus a `passwordHash`.

Now, we should review our `CreateUserRequest`:

```
newtype CreateUserRequest = CreateUserRequest
  { authToken :: UUID
  , user :: Record (UserRow ( password :: String ))
  }
```

Here `user` is everything a `User` is plus a `password`.

This means that we can take the `User` fields from the `CreateUserRequest` and compute the `passwordHash` from the `password` to create an `Account` that we can send to `createAccount` for both the Account Manager and Account Handler.

Now that we have a plan, let's start by writing `createAccount` for the Account Manager, i.e. in the `Manager.Account` module.

Just write the Type Signature and then continue reading.

We're going to put this in `Manager.Account`:

```
createAccount :: AVar Accounts -> Account -> Aff Unit ❶
```

❶ We pass in the `accountsAVar` and an `account`.

Now, just write the left-hand side of the implementation, then read on.

Here's that addition:

```
createAccount :: AVar Accounts -> Account -> Aff Unit
createAccount accountsAVar account = do
  ???
  pure unit ❶
```

❶ Since we're coding this one line at a time, it can be helpful to keep the compiler happy. That's why we've added this here. Now, we'll get compiler errors as we code instead of getting them all at the very end. Do NOT forget to delete it.

One thing to note here is that we could've chosen to simply pass `CreateUserRequest` instead of `Account`.

Why do you think what we've done is better than doing that? Or do you think passing `CreateUserRequest` is better?

If we were to pass `CreateUserRequest` then API changes COULD also break our Account Manager. We want the knowledge of the API to be limited to the API Handler. This is what is meant by Separation of

Concerns.

Next, we'll get `accounts` from the `accountsAVar` before advancing.

Here's the code to get all of the `accounts`:

```
createAccount :: AVar Accounts -> Account -> Aff Unit
createAccount accountsAVar account = do
  accounts <- AVar.take accountsAVar ❶
  ???
  pure unit
```

❶ We `take` it since we're going to modify it.

Now, we're ready to add this new account to all the other accounts. But wait.

We should first check to make sure the `userName` doesn't already exist. But if we do that then that means that this Function can fail and `Aff Unit` is no longer the correct return Type.

Okay, we have our work cut out for us. Figure out how you want to change the Type Signature to reflect the fact that this can fail. Think carefully about the Type you want to return.

Do the analysis and then read on to see my solution.

My first thought is that I only need to know that it failed, so why not use a `Boolean`. The problem here is that I have no idea what `true` and `false` mean without looking at the code in detail.

This is known as ***Boolean Blindness***. There's no reason we should be flying blind when we can make a Type that fully describes our situation.

By the way, if you thought of changing `Aff Unit` to `Aff (Maybe Unit)`, then you have the same blindness. We have no idea what `Nothing` means in this particular situation. `Nothing's` meaning is obvious when we call Functions like `Map.lookup` or `findIndex`. But when we call a Function named `createAccount`, then `Nothing` is pretty ambiguous.

So, let's make a Type that explicitly captures the results of the `createAccount` Function:

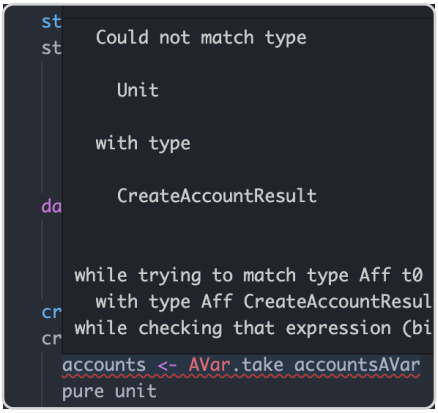
```
data CreateAccountResult
  = CreateAccountSuccess
  | CreateAccountAlreadyExists
```

The Data Constructors are named for exactly the situations that we expect to encounter.

Now, add `CreateAccountResult` and update your Type Signature to be:

```
createAccount :: AVar Accounts -> Account -> Aff CreateAccountResult
```

Now we have a compiler error:



Can you see what's wrong?

This isn't obvious from WHERE the red underline is. But we need to change `pure unit` to `pure CreateAccountSuccess`:

```
createAccount :: AVar Accounts -> Account -> Aff CreateAccountResult
createAccount accountsAVar account = do
  accounts <- AVar.take accountsAVar
  ???
  pure CreateAccountSuccess ❶
```

❶ Replaced `unit` with `CreateAccountSuccess`.

Now that our code compiles again, let's check to see if the account already exists. Add the code to check if the `userName` already exists. Then return here.

We have to destructure `account` to get the `userName` so that we can check to make sure that it doesn't already exist:

```
createAccount :: AVar Accounts -> Account -> Aff CreateAccountResult
createAccount accountsAVar account@(Account { userName }) = do ❶
  accounts <- AVar.take accountsAVar
  if Map.member userName accounts then
    pure CreateAccountAlreadyExists ❷
  else do ❸
    ???
    pure CreateAccountSuccess
```

❶ Destructure to get `userName`.

❷ Return an error if `userName` already exists. `Map.member` returns a `true` if it's already in the `Map`.

❸ Don't forget the `do` after the `else`.

Now we can safely add the `account` to our `accounts`. Do that and then return here.

Here we `insert` the `account` into `accounts` and then `put` it back to the `AVar`:

```
createAccount :: AVar Accounts -> Account -> Aff CreateAccountResult
createAccount accountsAVar account@(Account { userName }) = do
  accounts <- AVar.take accountsAVar
  if Map.member userName accounts then
    pure CreateAccountAlreadyExists
  else do
    AVar.put (Map.insert userName account accounts) accountsAVar ❶
    pure CreateAccountSuccess
```

❶ Add `account` to `accounts` and put it into the `AVar`.

And now, we've successfully created the account in memory. But wait... Look carefully at the code that I've written and see if you can see a bug, i.e. something we talked about when we wrote `verifySession`.

Can you see my mistake?

I've made the mistake of coding this such that we could do a `AVar.take` without an `AVar.put`. Can you see it now?

It happens in the failure case:

```
createAccount :: AVar Accounts -> Account -> Aff CreateAccountResult
createAccount accountsAVar account@(Account { userName }) = do
  accounts <- AVar.take accountsAVar
  if Map.member userName accounts then
    pure CreateAccountAlreadyExists ❶
  else do
    AVar.put (Map.insert userName account accounts) accountsAVar
    pure CreateAccountSuccess
```

❶ We've called `AVar.take` and are NOT calling `AVar.put` here.

Look at `verifySession` and see how we solved this problem there, then return back here.

Do you think we could solve our problem here in the same way?

I think we can. So, stop reading for a bit and fix my bad code to use the same technique to guarantee that we ALWAYS do an `AVar.put` before reading further.

Here's that change:

```
createAccount :: AVar Accounts -> Account -> Aff CreateAccountResult
createAccount accountsAVar account@(Account { userName }) = do
  accounts <- AVar.take accountsAVar
  let Tuple newAccounts result = ❶
        if Map.member userName accounts then
          Tuple accounts CreateAccountAlreadyExists ❷
        else
          Tuple (Map.insert userName account accounts) CreateAccountSuccess ❸
  AVar.put newAccounts accountsAVar ❹
  pure result ❺
```

❶ We capture the updated `accounts` in `newAccounts` and the `result` in a `Tuple`.

❷ If the `userName` already exists, then we return a `Tuple` of the unchanged `accounts` and a failure result.

❸ Otherwise, we updated the `accounts` by adding `account` and the `result` is success.

❹ We ALWAYS put the `newAccounts` back to the `AVar`.

❺ We return the `result`.

Now, we have a safe `createAccount` that won't leave us with an empty `AVar`.

But this feels like boilerplate to me. We did this same kind of code in `verifySession`.

Let's write a Function that captures this boilerplate and use it in both `createAccount` and `verifySession`. We'll write it together one line at a time.

We're going to call it `withAVar`. This Function will do the `AVar.take` and then GUARANTEE that the `AVar` is always put back so we won't have to think about it.

Here's it's Type Signature:

```
withAVar :: ∀ a b. AVar a -> (a -> Aff (Tuple a b)) -> Aff b
```

This will take an `AVar` of Type `a` and a Function that will consume that `a`. That Function will return some `Tuple a b` in `Aff`. To keep this Function as flexible as possible, it's important that we don't just return a plain `Tuple` so we can do effects in this Function.

The `Tuple's` first Value is the UPDATED `a`. This will be put back to the `AVar` by `withAVar`. The second part of the `Tuple` is Type `b` which is the result of the computation.

And finally, this Function returns that result in `Aff`.

Next, we'll code the left-hand side of the equal sign:

```
withAVar :: ∀ a b. AVar a -> (a -> Aff (Tuple a b)) -> Aff b
withAVar aVar f = ??? ❶
```

❶ We have the `aVar` and the Function, `f`.

Next, we'll get the `AVar` Value:

```
withAVar :: ∀ a b. AVar a -> (a -> Aff (Tuple a b)) -> Aff b
withAVar aVar f = do ❶
  value <- AVar.take aVar
  ???
```

❶ We start with a `do` block.

Now, we'll call our callback Function:

```
withAVar :: ∀ a b. AVar a -> (a -> Aff (Tuple a b)) -> Aff b
withAVar aVar f = do
  value <- AVar.take aVar
  Tuple newValue result <- f value ❶ ❷
  ???
```

❶ We destructure the `Tuple` to get the `newValue` of the `AVar` and the `result` of the computation.

❷ Since `f` runs in `Aff`, we use `bind` via the `<-` syntax. If it did not run in `Aff`, we'd have to use a `let`.

Now, we put the `newValue` back into the `AVar`:

```
withAVar :: ∀ a b. AVar a -> (a -> Aff (Tuple a b)) -> Aff b
withAVar aVar f = do
  value <- AVar.take aVar
  Tuple newValue result <- f value
  AVar.put newValue aVar ❶
  ???
```

❶ We ALWAYS call this. That's how we can guarantee that we'll never leave the `AVar` empty.

And finally, we'll return the result of the computation:

```
withAVar :: ∀ a b. AVar a -> (a -> Aff (Tuple a b)) -> Aff b
withAVar aVar f = do
  value <- AVar.take aVar
  Tuple newValue result <- f value
  AVar.put newValue aVar
  pure result ❶
```

❶ Return the computation result of the callback.

Looking at this Function, can you see how we can generalize it? Take a minute and see if you can figure a way to do that before continuing.

The `Aff` can be generalized to any Monad, `m`, that has a `MonadAff` Instance. But, we'll have to make sure we lift all `Aff` Functions into `m` using `liftAff`:

```
import Effect.Aff.Class (class MonadAff, liftAff) ❶

withAVar :: ∀ a b m. MonadAff m => AVar a -> (a -> m (Tuple a b)) -> m b ❷
withAVar aVar f = do
  value <- liftAff $ AVar.take aVar ❸
  Tuple newValue result <- f value
  liftAff $ AVar.put newValue aVar ❸
  pure result
```

❶ Added this import.

❷ Changed `Aff` to `m` and added the `MonadAff` Constraint on `m`.

❸ Lifted all `Aff` computations into `m` using `liftAff`.

Now, this Function can be used in `Aff` or any other Monad that can run in `Aff`, e.g. our `Handler`, which is a `ReaderT` Monad Stack with `Aff` on the bottom of the Stack. `ReaderT` delegates the `Aff` calls to its underlying Monad, which only works if `Aff` is in the Stack.

Add this to your code and then change `createAccount` to use it. When you're finished, continue reading.

Here's that change:

```
createAccount :: AVar Accounts -> Account -> Aff CreateAccountResult
createAccount accountsAVar account@(Account { userName }) = do
  withAVar accountsAVar ❶
    \accounts -> pure $ ❷
      if Map.member userName accounts then ❸
        Tuple accounts CreateAccountAlreadyExists
      else
        Tuple (Map.insert userName account accounts) CreateAccountSuccess
```

❶ We call `withAVar` passing in the `accountsAVar`.

❷ The original logic was a Pure Computation so we wrap it in `Aff` using `pure`.

❸ Our `if` logic doesn't change.

It's nice that we incubated `withAVar` in this module, but it really doesn't belong here. Besides, it would be nice to use it in `verifySession`.

So, we're going to move it to a File called `Utils.purs`. Create that File under the `src` Folder and move `withAVar` and the appropriate imports.

Then remove the following import from `Manager.Account`:

```
import Effect.Aff.Class (class MonadAff, liftAff)
```

And add:

```
import Utils (withAVar)
```

Return here when you've completed these tasks.

Here's `Utils.purs`:

```
module Utils where

import Prelude

import Data.Tuple (Tuple(..))
import Effect.Aff.AVar (AVar)
import Effect.Aff.AVar as AVar
import Effect.Aff.Class (class MonadAff, liftAff)

withAVar :: ∀ a b m. MonadAff m => AVar a -> (a -> m (Tuple a b)) -> m b
withAVar aVar f = do
  value <- liftAff $ AVar.take aVar
  Tuple newValue result <- f value
  liftAff $ AVar.put newValue aVar
  pure result
```

Now change `verifySession` in to use `withAVar` and then return here.

Here's that change:

```
import Utils (withAVar) ❶

verifySession :: AVar Sessions -> UUID -> Aff (Maybe Session)
verifySession sessionsAVar authToken = do
  expireSessions sessionsAVar
  withAVar sessionsAVar ❶
    \sessions -> do ❷
      now <- getTime <$> liftEffect now
      pure $ case Map.lookup authToken sessions of ❸
        Nothing -> Tuple sessions Nothing
        Just (Session session) ->
          let newSession = Session $ session { lastTime = now }
              newSessions = Map.insert authToken newSession sessions in
          Tuple newSessions (Just newSession)
```

❶ We're now using `withAVar` with `sessionsAVar`. It will do the `AVar.take` and `AVar.put` before and after calling our callback Function.

❷ Here's our callback. It runs in `Aff`, hence the `do` block.

❸ Our `case` expression is a `Pure` computation so we wrap it in `Aff` with `pure`.

Now by using `withAVar`, our code is much safer and we don't have to remember to call `AVar.put` on every possible path through our logic.

Next, we'll need to create an `Account` on disk. But, we already have `createAccount` in `Handler.Account`:

```
createAccount :: Account -> Aff (Either String Unit)
createAccount account = lmap show
  <$> (try $ appendTextFile ASCII accountsFile $ accountToCSV account)
```

Look at this Function and see if it will work for `CreateUserRequest's` requirements. Then continue reading.

It looks like it does, but I see a problem with it that we fixed in `createAccount` in `Manager.Account`. Compare the two and see if you can see what's wrong with this one. Then read on.

I can see that same Boolean Blindness we avoided in the Account Manager with the return Type `Either String Unit`. Technically, it's not Boolean Blindness but the `String` in `Either String Unit` is an implied File I/O error.

This is a case of something being implicit when explicit is not only possible but easy to implement and far better.

To remove the implicit nature of the return Type, we're going to make a Type like `CreateAccountResult` in `Manager.Account` for `Handler.Account` and modify `createAccount` to use it. Use the names `CreateAccountSuccess` and `CreateAccountFileError` for Data Constructors.

Once that's done, modify `createAccount` in `Handler.Account` to use our new Type.

Return back here once you've finished.

Here's that change:

```
data CreateAccountResult ❶
  = CreateAccountSuccess
  | CreateAccountFileError String

createAccount :: Account -> Aff CreateAccountResult ❷
createAccount account = lmap show
  <$> (try $ appendTextFile ASCII accountsFile $ accountToCSV account)
  <#> case _ of ❸ ❹
    Left err -> CreateAccountFileError err
    Right _ -> CreateAccountSuccess
```

❶ Created a Type for the result of calling `createAccount` that is far more informative than `Either String Unit`.

❷ Changed `Either String Unit` to `CreateAccountResult`. This probably broke whoever calls this.

❸ Using the flipped `map` via `<#>`.

❹ Using a `case` with a wildcard to process the `Either` from `try`.

This is a minor improvement but still worth it. Maybe not today but a month from now, we'll appreciate this.

Now, hunt down where `createAccount` was called and make it check the new return Type. Once that's fixed return here and continue reading.

Turns out that no one's calling this yet. That's good since our change would have broken our calling code.

We're now ready to work on `Handler.Api.CreateUser`.

Create a file called `CreateUser.purs` under `src/Handler/Api` and make sure you call the module `Handler.Api.CreateUser`.

Next, look at what we did with `Handler.Api.Logoff` and see what we can steal. Then return here.

The first thing I notice that we'll need is the data Type:

```
data Logoff = Logoff
```

Remember this is used to help the `router` to distinquish which handler to call. So, add this for `CreateUser` and then continue reading on.

Here's that Type:

```
module Handler.Api.CreateUser where

data CreateUser = CreateUser
```

The next thing we can copy from `Logoff` is the `ApiHandler` Instance code. So go ahead and write that next. Then return here to continue.

This is boilerplate code, but we've reduced it down to only 1 line of code:

```
instance apiHandlerCreateUser :: ApiHandler CreateUser where
  handle _ = handleApi handler
```

And now the compiler isn't happy because we haven't imported anything. Steal from imports from `Logoff` to make the errors disappear. Stop when you get the error regarding `handler` since we haven't written it yet. Do this before advancing forward.

Here are my imports:

```
import Prelude

import Data.Api.CreateUser (CreateUserRequest(..) ❶
                          , CreateUserResponse(..), CreateUserResults(..))
import Handler.Api.Common (handleApi)
import Handler.Class.ApiHandler (class ApiHandler, Handler) ❷
```

❶ I've brought in more than I need at the moment in anticipation that we'll need them when we write `handler`.

❷ This one has `Handler`, which is unnecessary at this point, but won't be for long.

Now, we have to tackle the `handler` Function.

But before we do, let's make a plan. Think through the steps that needs to be done to create a user in the system. Write them down if that helps. Then keep reading to see how your list compares with mine.

Here's my list (order matters):

- Verify `Session`.
- Make sure user is an `admin`.
- Create an `Account` from `user` in the `CreateUserRequest` using the `password` to calculate `passwordHash`.
- Call `createAccount` in the Account Manager to add it to our in memory accounts.
- If the previous call succeeded, then call `createAccount` in the Account Handler to append it to `accounts.csv`.

Well, that seems simple. But why this order?

Because the Account Handler doesn't know if the `Account` exists. It just does the File I/O. By calling the Account Manager first we can avoid writing to `accounts.csv` with a duplicate `userName` since `createAccount` in the Account Manager will give us an error if `userName` is a duplicate.

First, comment out the instance code to remove the compiler error:

```
-- instance apiHandlerCreateUser :: ApiHandler CreateUser where
--   handle _ = handleApi handler
```

So, code up `handler` by using `Logoff` and even `Logon` for reference. Return here when you're done or if you get stuck.

The first and easiest part is the Type Signature:

```
handler :: CreateUserRequest -> Handler
```

Next, easiest part is the left-hand side of the equal sign:

```
handler :: CreateUserRequest -> Handler
handler (CreateUserRequest { authToken, user }) = ???
```

Notice the destructuring. This is to make our code easier to read.

We should start with a do block since we're running under the Monad Stack of `Handler`:

```
handler :: CreateUserRequest -> Handler
handler (CreateUserRequest { authToken, user }) = do
  ???
```

The very first thing we must do is to validate the credentials of the caller. Here's `verifySessions's` Type Signature to help us:

```
verifySession :: AVar Sessions -> UUID -> Aff (Maybe Session)
```

We can see that it runs in `Aff` which is at the bottom of the `Handler` Monad Stack which means that it'll need a `lift`.

It will also need `sessionsAVar`.

Let's do both of those steps

```
import Control.Monad.Reader.Class (ask) ❶
import Control.Monad.Trans.Class (lift)
import Manager.Session (verifySession)

handler :: CreateUserRequest -> Handler
handler (CreateUserRequest { authToken, user }) = do
  { sessionsAVar } <- ask ❷
  verifiedSession <- lift $ verifySession sessionsAVar authToken ❸
  ???
```

❶ Add these imports.

❷ Get the `sessionsAVar`.

❸ Verify the session.

Next, we'll handle the 2 cases of `verifiedSession`. We'll start with the failure case:

```
import Data.Maybe (Maybe(..)) ❶

handler :: CreateUserRequest -> Handler
handler (CreateUserRequest { authToken, user }) = do
  { sessionsAVar } <- ask
  verifiedSession <- lift $ verifySession sessionsAVar authToken
  response <- case verifiedSession of ❷
    Nothing -> ???
    ???
```

❶ Import `Maybe` and its Data Constructors.

❷ This `case` expression will evaluate to a `response` like we've done in other `handlers`.

At this point, we'll want to look at `CreateUserResponse` and its related Types to see what to return:

```
data CreateUserFailureReason
  = NotAuthorized
  | NotAuthenticated
  | AlreadyExists
  | FileIOError String

data CreateUserResults
  = CreateUserResultsSuccess
  | CreateUserResultsFailure { reason :: CreateUserFailureReason }

newtype CreateUserResponse = CreateUserResponse CreateUserResults
```

Take a look at this to see what the response should be. Then read on.

Here's the response for the error case:

```
import Data.Api.CreateUser (CreateUserRequest(..)
                           , CreateUserResponse(..), CreateUserResults(..)
                           , CreateUserFailureReason(..)) ❶

handler :: CreateUserRequest -> Handler
handler (CreateUserRequest { authToken, user }) = do
  { sessionsAVar } <- ask
  verifiedSession <- lift $ verifySession sessionsAVar authToken
  response <- case verifiedSession of
    Nothing -> CreateUserResponse
      $ CreateUserResultsFailure { reason: NotAuthenticated }
    ???
```

❶ Added `CreateUserFailureReason` to import list.

Next, we'll deal with the success case:

```
import Entity.Session (Session(..)) ❶

handler :: CreateUserRequest -> Handler
handler (CreateUserRequest { authToken, user }) = do
  { sessionsAVar } <- ask
  verifiedSession <- lift $ verifySession sessionsAVar authToken
  response <- case verifiedSession of
    Nothing -> CreateUserResponse
      $ CreateUserResultsFailure { reason: NotAuthenticated }
    Just (Session { userName }) -> ??? ❷ ❸
```

❶ Import `Session` and its Data Constructors for Pattern Matching.

❷ `verifySession` returns the `Session` of the requestor but we're only concerned with `userName` to check for the `NotAuthorized` case, i.e. the user making this request is NOT an admin.

❸ We destructure the `Session` to gain access to `userName`.

Next, we should check to make sure the user is an admin. But we're going to have to write a Function called `findAccount` first. We're going to switch to the Account Manager, i.e. `Manager.Account`, and write `findAccount` there.

Write just the Type Signature before reading further.

Since `accounts` is stored in an `AVar`, we'll need to pass `AVar Accounts` to `findAccount`. We'll also have to pass the `userName`:

```
findAccount :: AVar Accounts -> String -> Aff (Maybe Account) ❶
```

❶ Returns a `Maybe Account` since it could fail.

For the next step, we're going to have to reminder ourselves what `Accounts` looks like:

```
type Accounts = Map String Account
```

Since it's in a `Map` and it's key is `userName`, writing the rest of the code should be pretty simple. Finish the code and then keep reading.

Let's start with the left-hand side:

```
findAccount :: AVar Accounts -> String -> Aff (Maybe Account)
findAccount accountsAVar userName = ???
```

Now, lets get the `Accounts`:

```
findAccount :: AVar Accounts -> String -> Aff (Maybe Account)
findAccount accountsAVar userName = do
  accounts <- AVar.read accountsAVar ❶
  ???
```

❶ We use `AVar.read` since we're not changing `accounts`.

And now, we'll look up the `userName` in the `Map` using `lookup`:

```
findAccount :: AVar Accounts -> String -> Aff (Maybe Account)
findAccount accountsAVar userName = do
  accounts <- AVar.read accountsAVar
  pure $ Map.lookup userName accounts ❶ ❷
```

❶ `Map.lookup` can fail and so it returns a `Maybe`.

❷ Don't forget to wrap the `Maybe` in `Aff` with `pure`.

Now, we could stop here, but as always, when I see a `do` block with only 2 lines, I always look to see how it would look on 1 line with `>>=`:

```
findAccount :: AVar Accounts -> String -> Aff (Maybe Account)
findAccount accountsAVar userName =
  AVar.read accountsAVar >>= pure <<< Map.lookup userName
```

This version eliminates the `accounts` Variable and is easier to read.

Now, we can use this in our `handler` code:

```
import Data.Maybe (Maybe(..), maybe) ❶
import Entity.Account (Account(..)) ❷
import Manager.Account (findAccount) ❷

handler :: CreateUserRequest -> Handler
handler (CreateUserRequest { authToken, user }) = do
  { sessionsAVar } <- ask
  verifiedSession <- lift $ verifySession sessionsAVar authToken
  response <- case verifiedSession of
    Nothing -> CreateUserResponse
      $ CreateUserResultsFailure { reason: NotAuthenticated }
    Just (Session { userName }) -> do
      admin <- lift $ findAccount accountsAVar userName ❸
        <#> maybe false (\(Account a) -> a.admin) ❹ ❺
      if not admin then CreateUserResponse
        $ CreateUserResultsFailure { reason: NotAuthorized } ❻
      else
        ???
```

❶ Added `maybe` to the import list.

❷ Add these imports.

❸ `lift` the call to `findAccount` since it runs in `Aff`.

❹ Use `maybe` to turn the not-found case into `false`.

❺ If found then make sure the `Account` is an admin account, i.e. `admin` is `true`. Notice that we're destructuring the Record to get at `admin`. In this case, this approach is more terse.

❻ Response is `NotAuthorized` if this request is from a non-admin account.

This is probably the first usage of `maybe` that we've come across. Here's its Type Signature:

```
maybe :: ∀ a b. b -> (a -> b) -> Maybe a -> b
```

`maybe` takes a default Value for the `Nothing`, i.e. `b`, case and then a Function for the `Just` case. Notice that

the Function must return a `b`, which is the same Type as the default Value.

Now, at this point, we're ready to implement the next step in our plan. Here's what's left to do:

- Create an `Account` from `user` in the `CreateUserRequest` using the `password` to calculate `passwordHash`.
- Call `createAccount` in the Account Manager to add it to our in memory accounts.
- If the previous call succeeded, then call `createAccount` in the Account Handler to append it to `accounts.csv`.

Search for how we calculate that `passwordHash` in the Account Handler then return here.

Did you find `passwordHashHex`?

We're going to need to call that Function. But it seems odd that Password Hashing is in the Account Handler. It doesn't deal with the outside world.

`passwordHashHex` and everything it calls and everything those Functions call should be moved into a file called `Crypto.purs`.

Create that file under the `src` Folder and then copy and paste those Functions from `Handler.Account` to `Crypto`. Then add all of the imports (or copy them) until the `Crypto` module compiles.

Make those change before advancing any further.

Here is `Crypto`:

```
module Crypto where

import Prelude

import Data.Char (toCharCode)
import Data.Foldable (foldl)
import Data.String.CodeUnits (toCharArray, fromCharArray)
import Effect.Aff (Aff)
import Random.LCG (Seed, mkSeed)
import Test.QuickCheck.Arbitrary (arbitrary)
import Test.QuickCheck.Gen (sample)
import Data.String.CodePoints (length)
import Node.Crypto.Hash (Algorithm(..), hex)
import Effect.Class (liftEffect)

userNameSeed :: String -> Seed
userNameSeed userName =
  userName
    # toCharArray
    <#> toCharCode
    # foldl (*) 1
    # mkSeed

userNameSalt :: Int -> String -> String
userNameSalt saltLength userName =
  fromCharArray
    $ sample (userNameSeed userName) saltLength arbitrary

passwordHashHex :: String -> String -> Aff String
passwordHashHex userName password =
  let salt = userNameSalt (3 * length userName) userName in
  liftEffect $ hex SHA512 (password <> salt)
```

Once that compiles, delete the Functions in `Handler.Account` that got moved and import `passwordHashHex` from `Crypto` so it'll compile again.

Then you'll wind up with a boatload of warning for redundant imports. So, clean those up.

Once you've done that refactor, continue reading.

Now, return back to `handler` in `Manager.Account`.

Import `passwordHashHex` from `Crypto` and remove the old import of `passwordHashHex` from `Handler.Account`.

Then add the code to `handler` in `Handler.Api.CreateUser` to compute the `passwordHash`. Then continue reading on.

Here's that change:

```
import Crypto (passwordHashHex) ❶

handler :: CreateUserRequest -> Handler
handler (CreateUserRequest { authToken, user }) = do
  { sessionsAVar } <- ask
  verifiedSession <- lift $ verifySession sessionsAVar authToken
  response <- case verifiedSession of
    Nothing -> CreateUserResponse
      $ CreateUserResultsFailure { reason: NotAuthenticated }
    Just (Session { userName }) -> do
      admin <- lift $ findAccount accountsAVar userName
        <#> maybe false (\(Account a) -> a.admin)
      if not admin then pure CreateUserResponse ❷
        $ CreateUserResultsFailure { reason: NotAuthorized }
      else do ❸
        passwordHash <- lift
          $ passwordHashHex user.userName user.password ❹
        ???
```

❶ Add this import.

❷ Added `pure` since `else` now has a `do` block.

❸ Added `do` block to `else` since we're calling `passwordHashHex` here.

❹ Lifting `passwordHashHex` since it runs in `Aff`.

Next, remove `password` from `user`. Use `delete` like we did back in the `loggingRouter`. Return here when you've finished.

Here are the changes:

```
import Record (delete) ❶
import Type.Proxy (Proxy(..))

handler :: CreateUserRequest -> Handler
handler (CreateUserRequest { authToken, user: user' }) = do ❷
  { sessionsAVar } <- ask
  verifiedSession <- lift $ verifySession sessionsAVar authToken
  response <- case verifiedSession of
    Nothing -> CreateUserResponse
      $ CreateUserResultsFailure { reason: NotAuthenticated }
    Just (Session { userName }) -> do
      admin <- lift $ findAccount accountsAVar userName
        <#> maybe false (\(Account a) -> a.admin)
      if not admin then pure CreateUserResponse
        $ CreateUserResultsFailure { reason: NotAuthorized }
      else do
        passwordHash <- lift
          $ passwordHashHex user'.userName user'.password ❸
        let user = delete (Proxy :: _ "password") user' ❹
        ???
```

❶ Add these imports.

❷ Renamed `user` to `user'` since it's not REALLY a `User`.

❸ Use `user'` instead of `user`.

❹ Delete the `password` field from `user'` to produce a real `User` called `user`.

Next, build an `Account` Value from the `user` and the `passwordHash` using `insert` from the `purescript-record` package. Then return here continue on.

Here is the code for creating an `Account` from `user` and `passwordHash`:

```
import Record (delete, insert) ❶

handler :: CreateUserRequest -> Handler
handler (CreateUserRequest { authToken, user: user' }) = do
  { sessionsAVar } <- ask
  verifiedSession <- lift $ verifySession sessionsAVar authToken
  response <- case verifiedSession of
    Nothing -> CreateUserResponse
      $ CreateUserResultsFailure { reason: NotAuthenticated }
    Just (Session { userName }) -> do
      admin <- lift $ findAccount accountsAVar userName
        <#> maybe false (\(Account a) -> a.admin)
      if not admin then pure CreateUserResponse
        $ CreateUserResultsFailure { reason: NotAuthorized }
      else do
        passwordHash <- lift
          $ passwordHashHex user'.userName user'.password
        let user = delete (Proxy :: _ "password") user'
            account = Account $
              insert (Proxy :: _ "passwordHash") passwordHash user ❷
        ???
```

❶ Added `insert` to import list.

❷ Create `account` by adding the `passwordHash` field to `user`.

We're ready to perform the next step of our plan, i.e. create the account in the Account Manager.

We'll start by looking at `createAccount` from the Account Manager:

```
createAccount :: AVar Accounts -> Account -> Aff CreateAccountResult
```

Take a look at the Type Signature and see if you can discern anything of importance regarding how we should proceed from it. Then read on.

I see 2 things. First, it needs the `accountsAVar` and we don't have it yet. And `createAccount` runs in `Aff` which is at the bottom of the `Handler` Monad Stack which means that it'll need a `lift`.

Let's add both:

```
import Manager.Account (findAccount, createAccount) ❶

handler :: CreateUserRequest -> Handler
handler (CreateUserRequest { authToken, user: user' }) = do
  { accountsAVar, sessionsAVar } <- ask ❷
  verifiedSession <- lift $ verifySession sessionsAVar authToken
  response <- case verifiedSession of
    Nothing -> CreateUserResponse
      $ CreateUserResultsFailure { reason: NotAuthenticated }
    Just (Session { userName }) -> do
      admin <- lift $ findAccount accountsAVar userName
        <#> maybe false (\(Account a) -> a.admin)
      if not admin then pure CreateUserResponse
        $ CreateUserResultsFailure { reason: NotAuthorized }
      else do
        passwordHash <- lift
          $ passwordHashHex user'.userName user'.password
        let user = delete (Proxy :: _ "password") user'
            account = Account $
              insert (Proxy :: _ "passwordHash") passwordHash user
        result <- lift $ createAccount accountsAVar account ❸
        ???
```

❶ Add `createAcccount` to import list.

❷ Also `ask` for `accountsAVar`.

❸ Create the account capturing the `result`.

Next, we need to handle the 2 cases from `createAccount`. Here's the `CreateAccountResults` definition for reference:

```
data CreateAccountResult
  = CreateAccountSuccess
  | CreateAccountAlreadyExists
```

Now we'll handle the error case first:

```
import Manager.Account (CreateAccountResult(..)
                       , findAccount, createAccount) ❶

handler :: CreateUserRequest -> Handler
handler (CreateUserRequest { authToken, user: user' }) = do
  { accountsAVar, sessionsAVar } <- ask
  verifiedSession <- lift $ verifySession sessionsAVar authToken
  response <- case verifiedSession of
    Nothing -> CreateUserResponse
      $ CreateUserResultsFailure { reason: NotAuthenticated }
    Just (Session { userName }) -> do
      admin <- lift $ findAccount accountsAVar userName
        <#> maybe false (\(Account a) -> a.admin)
      if not admin then pure CreateUserResponse
        $ CreateUserResultsFailure { reason: NotAuthorized }
      else do
        passwordHash <- lift
          $ passwordHashHex user'.userName user'.password
        let user = delete (Proxy :: _ "password") user'
            account = Account $
              insert (Proxy :: _ "passwordHash") passwordHash user
        result <- lift $ createAccount accountsAVar account
        case result of
          CreateAccountAlreadyExists -> CreateUserResponse
            $ CreateUserResultsFailure { reason: AlreadyExists } ❷
          ???
```

❶ Added `CreateAccountResults` to the import list.

❷ If `CreateAccountAlreadyExists` is returned then the reason this fails is `AlreadyExists`. You can see this in `Data.Api.CreateUser`.

And now for the success case:

```
handler :: CreateUserRequest -> Handler
handler (CreateUserRequest { authToken, user: user' }) = do
  { accountsAVar, sessionsAVar } <- ask
  verifiedSession <- lift $ verifySession sessionsAVar authToken
  response <- case verifiedSession of
    Nothing -> CreateUserResponse
      $ CreateUserResultsFailure { reason: NotAuthenticated }
    Just (Session { userName }) -> do
      admin <- lift $ findAccount accountsAVar userName
        <#> maybe false (\(Account a) -> a.admin)
      if not admin then pure CreateUserResponse
        $ CreateUserResultsFailure { reason: NotAuthorized }
      else do
        passwordHash <- lift
          $ passwordHashHex user'.userName user'.password
        let user = delete (Proxy :: _ "password") user'
            account = Account $
              insert (Proxy :: _ "passwordHash") passwordHash user
        result <- lift $ createAccount accountsAVar account
        case result of
          CreateAccountAlreadyExists -> CreateUserResponse
            $ CreateUserResultsFailure { reason: AlreadyExists }
          CreateAccountSuccess -> do ❶
            ???
```

❶ Start another `do` block that we may or may not need since we've only got to do one operation. We'll delete this if we're wrong.

At this point, we're ready to do step 3 in our plan, create the account in the Account Handler, i.e. write the new account to the Accounts File.

Here's its `createAccount` Type Signature:

```
createAccount :: Account -> Aff CreateAccountResult
```

We'll call the Account Handler to append the `Account` to `accounts.csv`:

```
import Handler.Account as AH ❶
import Manager.Account as AM

handler :: CreateUserRequest -> Handler
handler (CreateUserRequest { authToken, user: user' }) = do
  { accountsAVar, sessionsAVar } <- ask
  verifiedSession <- lift $ verifySession sessionsAVar authToken
  response <- case verifiedSession of
    Nothing -> CreateUserResponse
      $ CreateUserResultsFailure { reason: NotAuthenticated }
    Just (Session { userName }) -> do
      admin <- lift $ AM.findAccount accountsAVar userName ❷
        <#> maybe false (\(Account a) -> a.admin)
      if not admin then pure CreateUserResponse
        $ CreateUserResultsFailure { reason: NotAuthorized }
      else do
        passwordHash <- lift
          $ passwordHashHex user'.userName user'.password
        let user = delete (Proxy :: _ "password") user'
            account = Account $
              insert (Proxy :: _ "passwordHash") passwordHash user
        result <- lift $ AM.createAccount accountsAVar account ❷
        case result of
          AM.CreateAccountAlreadyExists -> CreateUserResponse ❷
            $ CreateUserResultsFailure { reason: AlreadyExists }
          AM.CreateAccountSuccess -> do ❷
            result <- lift $ AH.createAccount account ❸ ❹
            ???
```

❶ Had to change these to be imported qualified since we have 2 `createAccounts`.

❷ Change these lines of code to use qualified import of `AM`.

❸ Use qualified import of `AH`.

❹ Don't forget `lift`.

Now, we can handle the error and success cases:

```
handler :: CreateUserRequest -> Handler
handler (CreateUserRequest { authToken, user: user' }) = do
  { accountsAVar, sessionsAVar } <- ask
  verifiedSession <- lift $ verifySession sessionsAVar authToken
  response <- case verifiedSession of
    Nothing -> CreateUserResponse
      $ CreateUserResultsFailure { reason: NotAuthenticated }
    Just (Session { userName }) -> do
      admin <- lift $ AM.findAccount accountsAVar userName
        <#> maybe false (\(Account a) -> a.admin)
      if not admin then pure CreateUserResponse
        $ CreateUserResultsFailure { reason: NotAuthorized }
      else do
        passwordHash <- lift
          $ passwordHashHex user'.userName user'.password
        let user = delete (Proxy :: _ "password") user'
            account = Account $
              insert (Proxy :: _ "passwordHash") passwordHash user
        result <- lift $ AM.createAccount accountsAVar account
        case result of
          AM.CreateAccountAlreadyExists -> CreateUserResponse
            $ CreateUserResultsFailure { reason: AlreadyExists }
          AM.CreateAccountSuccess -> do
            result <- lift $ AH.createAccount account
            pure $ case result of ❶
              AH.CreateAccountFileError err -> CreateUserResponse
                $ CreateUserResultsFailure { reason: FileIOError err }
              AH.CreateAccountSuccess -> CreateUserResultsSuccess
  ???
```

❶ Handle both the success case and error case.

And finally, we can… Wait. Wait. Wait.

Before we go any further, just take a look at what we've written so far. How do you feel about this code?

My thoughts are that it's too complex. Frankly, this is a hot mess.

There's too much error checking. What happened to all that great Monadic Happy Path programming we did earlier in the book? I want that back!!

I want the whole `case` expression to run in `ExceptT` so I don't have to check for errors.

Our natural inclination may be to just keep forging ahead telling ourselves that we'll fix it later, but we won't.

That's because once this compiles we're going to want to test it. And once it's working, we're not going to want to touch it for fear that we'll break it. And there's way too much work here to just throw it away.

And if you're sort of feeling this right now, take a minute and look back at the code so far and realize that there are about another 2 dozen edits before it compiles and now you own this Function and must debug it when it breaks.

When writing production code, all too often we convince ourselves why this Function is just good enough. It works we tell ourselves.

We convince ourselves that the rewrite will probably not be that much better after all or that it might even be worse. And, if that's the case, then aren't we just saving time by not rewriting it?

These are all great excuses. But they're just reasons why we don't want to rewrite this awful mess.

I promise you, a rewrite is going to be so much better.

So, comment out this code. We may use it as reference. And when we're done with our rewrite, we'll compare our new Function with it to prove that it was worth rewriting.

We're going to start over from here:

```
import Control.Monad.Except (runExceptT) ❶

handler :: CreateUserRequest -> Handler
handler (CreateUserRequest { authToken, user: user' }) = do
  { accountsAVar, sessionsAVar } <- ask
  result <- lift $ runExceptT do ❷ ❸
    ???
```

❶ Add this import.

❷ The `do` block runs in `ExceptT e Aff a`. `Aff` is the underlying Monad since we're going to be calling Functions that run in `Aff`. We'll figure out what `e` and `a` are soon enough.

❸ After we call `runExceptT`, we'll have an `Aff a`, which we then `lift` into `Handler`.

Now, we need to know the Type Signature of the `do` block. Looking at our old implementation, we can see that we're returning a `result` of Type `CreateUserResults`. But look carefully at each return Value and see if you can see what's different each time. Then keep reading.

Here's what I see:

```
handler :: CreateUserRequest -> Handler
handler (CreateUserRequest { authToken, user: user' }) = do
  { accountsAVar, sessionsAVar } <- ask
  verifiedSession <- lift $ verifySession sessionsAVar authToken
  response <- case verifiedSession of
    Nothing -> CreateUserResponse
      $ CreateUserResultsFailure { reason: NotAuthenticated } ❶
    Just (Session { userName }) -> do
      admin <- lift $ AM.findAccount accountsAVar userName
        <#> maybe false (\(Account a) -> a.admin)
      if not admin then pure CreateUserResponse
        $ CreateUserResultsFailure { reason: NotAuthorized } ❶
      else do
        passwordHash <- lift
          $ passwordHashHex user'.userName user'.password
        let user = delete (Proxy :: _ "password") user'
            account = Account $
              insert (Proxy :: _ "passwordHash") passwordHash user
        result <- lift $ AM.createAccount accountsAVar account
        case result of
          AM.CreateAccountAlreadyExists -> CreateUserResponse
            $ CreateUserResultsFailure { reason: AlreadyExists } ❶
          AM.CreateAccountSuccess -> do
            result <- lift $ AH.createAccount account
            pure $ case result of
              AH.CreateAccountFileError err -> CreateUserResponse
                $ CreateUserResultsFailure { reason: FileIOError err } ❶
              AH.CreateAccountSuccess -> CreateUserResultsSuccess
  ???
```

❶ The only difference is the `CreateUserFailureReason`, e.g. `NotAuthenticated`, `NotAuthorized`, `AlreadyExists` and `FileIOError`.

All error conditions are simply setting a different `CreateUserFailureReason`. That means our `do` block Type Signature can be `ExceptT CreateUserFailureReason Aff Unit`.

The `a` is `Unit` since we only want the Side-effects of these Functions and at the end we'll just look at the `Either` that is `result` and decide whether to build the error response or the success one.

Here's our rewrite again so far:

```
handler :: CreateUserRequest -> Handler
handler (CreateUserRequest { authToken, user: user' }) = do
  { accountsAVar, sessionsAVar } <- ask
  result <- lift $ runExceptT do
    ???
```

The first thing we did in our old code was to call verifySession. Let's incubate this line of code starting with the Function call and modify it until it has the correct Type for our `do` block, i.e. `ExceptT CreateUserFailureReason Aff Session`.

Here's it's Type Signature:

```
verifySession :: AVar Sessions -> UUID -> Aff (Maybe Session)
```

We know we can use `note` to convert the `Maybe` we get from `verifySession` to an `Either`. We'll have to `map` since `Maybe` is in the `Aff` Context:

```
verifySession sessionsAVar authToken <#> note NotAuthenticated
  :: Aff (Either CreateUserFailureReason Session)
```

Now, all that we need is a way to convert `Aff (Either CreateUserFailureReason Session)` to `ExceptT CreateUserFailureReason Aff Session` so we can use it in our `do` block.

The first thing to notice is that we have an `Either` and that we should be able to convert that to an `ExceptT` since `ExceptT` contains an `Either`:

```
newtype ExceptT e m a = ExceptT (m (Either e a))
```

And it turns out there's a Function for that, but it's in Haskell and for some reason it's not in PureScript. But we're going to write it here together. We could just copy the code from Haskell, but then we wouldn't learn as much.

Let's call it what they did in Haskell, `liftEither`:

```
import Data.Either (Either) ❶

liftEither :: ∀ e m a. Either e a -> ExceptT e m a
```

❶ Add this import.

This will take an `Either` and return an `ExceptT`.

Next, we'll code the `Left` case:

```
import Data.Either (Either(..)) ❶

liftEither :: ∀ e m a. Either e a -> ExceptT e m a
liftEither (Left err) = ???
```

❶ Include `Either's` Data Constructors.

We need a way to take an `e` and turn it into an `ExceptT e m a`. If you search `Pursuit` directly for `e -> ExceptT e m a`, you'll get a ton of Functions none of which look right.

But if your realize that `ExceptT` is a specialized case of `MonadError`, then that might help you.

`MonadError` has 2 Type Parameters, `e` then `m`. So search `Pursuit` for `e -> m a` and see if you can find something that will work. Then return here.

Did you find `throwError`? Here's it's definition:

```
throwError :: ∀ a m e. MonadThrow e m => e -> m a ❶
```

❶ In our case, the `m` in `throwError` is `ExceptT e m`.

We can use this:

```
import Control.Monad.Error.Class (throwError) ❶

liftEither :: ∀ e m a. Either e a -> ExceptT e m a
liftEither (Left err) = throwError err
???
```

❶ Add this import.

But, now I want to generalize our Type Signature like `throwError's`:

```
import Control.Monad.Error.Class (class MonadError, throwError) ❶

liftEither :: ∀ e m a. MonadError e m => Either e a -> m a ❷
liftEither (Left err) = throwError err
???
```

❶ Added Typeclass `MonadError` to import list.

❷ Replaced `ExceptT e m` with `m`.

That's better.

Next, we'll do the `Right` case:

```
liftEither :: ∀ e m a. MonadError e m => Either e a -> m a
liftEither (Left err) = throwError err
liftEither (Right x)  = ???
```

We just want to wrap `x` up in the `MonadError`, in our case, `ExceptT`. We'll use `pure` for that since we generalized this Function:

```
liftEither :: ∀ e m a. MonadError e m => Either e a -> m a
liftEither (Left err) = throwError err
liftEither (Right x)  = pure x
```

Now, this may seem done and quite terse. But, there's a very useful helper that will make this far terser. It's called `either`:

```
either :: ∀ a b c. (a -> c) -> (b -> c) -> Either a b -> c ❶ ❷
```

❶ `a -> c` maps the `Left` to `c`.

❷ `b -> c` maps the `Right` to `c`.

Both `throwError` and `pure` are Functions that can be passed to `either`:

```
import Data.Either (Either(..), either) ❶

liftEither :: ∀ e m a. MonadError e m => Either e a -> m a
liftEither = either throwError pure
```

❶ Added `either` to import list.

That reads nicely. To me it says that `liftEither` will `throwError` on the `Left` and `pure` on the `Right`.

Add this Function to your module.

Now, returning to line of code we're incubating:

```
verifySession sessionsAVar authToken <#> note NotAuthenticated
  :: Aff (Either CreateUserFailureReason Session)
```

We can see that the `Either` is buried in `Aff`. But even worse than that is the fact that `Aff` doesn't have a `MonadError` instance. But `ExceptT` does.

So first, we need to convert `Aff` to `ExceptT`, which we can do by using `lift` since `Aff` is at the base of our `do` block's Stack, which is `ExceptT CreateUserFailureReason Aff Unit`:

```
verifySession sessionsAVar authToken <#> note NotAuthenticated # lift ❶
  :: ExceptT CreateUserFailureReason Aff
      (Either CreateUserFailureReason Session)
```

❶ Call `lift`.

That may not seem like it helped much since the `Either` is now buried in an `ExceptT`. Well, we wanted it buried in a `MonadError` instead `Aff`, because now we can use `bind` to get to the `Either`.

```
verifySession sessionsAVar authToken <#> note NotAuthenticated
  # lift >>= ??? ❶
  :: ExceptT CreateUserFailureReason Aff
      (Either CreateUserFailureReason Session)
```

❶ Bind to some Monadic Function.

Then we can send that `Either` to `liftEither`, which will wrap it up in an `ExceptT`. `liftEither` returns a Monadic Value, which is exactly what the Monadic Function passed to `bind` is supposed to return:

```
verifySession sessionsAVar authToken <#> note NotAuthenticated
  # lift >>= liftEither
  :: ExceptT CreateUserFailureReason Aff Session
```

`liftEither` will take a `Right` and simply wrap it in an `ExceptT` with `pure` or if it gets a `Left` it will call `throwError`, which Short-circuits. This Short-circuiting is the whole point of this rewrite.

Let's apply the line of code that we've just incubated here:

```
import Data.Either (Either(..), either, note) ❶
import HTTPure as HTTPure ❷

handler :: CreateUserRequest -> Handler
handler (CreateUserRequest { authToken, user: user' }) = do
  { accountsAVar, sessionsAVar } <- ask
  result <- lift $ runExceptT do
    verifySession sessionsAVar authToken
      <#> note NotAuthenticated # lift >>= liftEither ❸
    ???
    pure unit ❹
  HTTPure.notFound ❹
```

❶ Added `note` to the import list.

❷ Added this import.

❸ Added the call to `verifySession`.

❹ Put these 2 lines here to keep the compiler happy. This will make it so much easier to code since we'll find our errors as we code. There are a lot of Types here and we're going to want to debug the compiler errors one line at a time. We must remember to delete them.

Next, let's make sure the user making this request is an admin:

```
handler :: CreateUserRequest -> Handler
handler (CreateUserRequest { authToken, user: user' }) = do
  { accountsAVar, sessionsAVar } <- ask
  result <- lift $ runExceptT do
    Session { userName } <- verifySession sessionsAVar authToken ❶
      <#> note NotAuthenticated # lift >>= liftEither
    ???
    pure unit
  HTTPure.notFound
```

❶ Now we're extracting the `Session` and destructuring it to get to the Record that has `userName` in it.

Next, we'll call `findAccount`:

```
handler :: CreateUserRequest -> Handler
handler (CreateUserRequest { authToken, user: user' }) = do
  { accountsAVar, sessionsAVar } <- ask
  result <- lift $ runExceptT do
    Session { userName } <- verifySession sessionsAVar authToken
      <#> note NotAuthenticated # lift >>= liftEither
    AM.findAccount accountsAVar userName ❶
      ??? ❷
    pure unit
  HTTPure.notFound
```

❶ Returns an `Aff (Maybe Account)`.

❷ We're not done with this line of code.

We'll want to turn the result of `findAccount` into an `Either` just like we did with `verifySession` using `note`, but this time with `NotAuthorized` as the error if not found:

```
handler :: CreateUserRequest -> Handler
handler (CreateUserRequest { authToken, user: user' }) = do
  { accountsAVar, sessionsAVar } <- ask
  result <- lift $ runExceptT do
    Session { userName } <- verifySession sessionsAVar authToken
      <#> note NotAuthenticated # lift >>= liftEither
    AM.findAccount accountsAVar userName
      <#> note NotAuthorized ❶
      ??? ❷
    pure unit
  HTTPure.notFound
```

❶ Now this is an `Aff (Either CreateUserFailureReason Account)`.

❷ We're not done with this line of code.

We'll `lift` this call just like we did with `verifySession`:

```
handler :: CreateUserRequest -> Handler
handler (CreateUserRequest { authToken, user: user' }) = do
  { accountsAVar, sessionsAVar } <- ask
  result <- lift $ runExceptT do
    Session { userName } <- verifySession sessionsAVar authToken
      <#> note NotAuthenticated # lift >>= liftEither
    Account { admin } <- AM.findAccount accountsAVar userName ❶
      <#> note NotAuthorized # lift >>= liftEither ❷
    ???
    pure unit
  HTTPure.notFound
```

❶ Now, we're extracting and destructing the `Account` returned by `findAccount`.

❷ We lift using `# lift >>= liftEither`.

And now, we can check to see if this is an admin request:

```
handler :: CreateUserRequest -> Handler
handler (CreateUserRequest { authToken, user: user' }) = do
  { accountsAVar, sessionsAVar } <- ask
  result <- lift $ runExceptT do
    Session { userName } <- verifySession sessionsAVar authToken
      <#> note NotAuthenticated # lift >>= liftEither
    Account { admin } <- AM.findAccount accountsAVar userName
      <#> note NotAuthorized # lift >>= liftEither
    unless admin $ throwError NotAuthorized ❶
    ???
    pure unit
  HTTPure.notFound
```

❶ If not `admin` then Short-circuit with `NotAuthorized` via `throwError`.

We can steal the code we wrote from our old code to compute the `passwordHash` and create the `user` and `account`:

```
handler :: CreateUserRequest -> Handler
handler (CreateUserRequest { authToken, user: user' }) = do
  { accountsAVar, sessionsAVar } <- ask
  result <- lift $ runExceptT do
    Session { userName } <- verifySession sessionsAVar authToken
      <#> note NotAuthenticated # lift >>= liftEither
    Account { admin } <- AM.findAccount accountsAVar userName
      <#> note NotAuthorized # lift >>= liftEither
    unless admin $ throwError NotAuthorized ❶
    passwordHash <- lift $ passwordHashHex user'.userName user'.password
    let user = delete (Proxy :: _ "password") user'
        account = Account $
          insert (Proxy :: _ "passwordHash") passwordHash user
    ???
    pure unit
  HTTPure.notFound
```

Next, let's add the call to create the account in the Account Manager. But let's work on that line of code all by itself first.

Here's its Type Signature for `createAccount`:

```
createAccount :: AVar Accounts -> Account -> Aff CreateAccountResult
```

And here's the line of code we wrote in our old code:

```
AM.createAccount accountsAVar account :: Aff CreateAccountResult
```

We need that Function to return an `Either` like it used to. But this time, we're going to return an explicit error.

To do this, we're going to convert:

```
data CreateAccountResult
  = CreateAccountSuccess
  | CreateAccountAlreadyExists
```

into:

```
data CreateAccountError = CreateAccountAlreadyExists
```

Go to `Manager.Account` and replace the `CreateAccountResult` with `CreateAccountError`.

And now, we'll make `createAccount` in the same module return `Aff (Either CreateAccountError Unit)`:

```
import Data.Either (Either(..)) ❶

createAccount
  :: AVar Accounts
  -> Account
  -> Aff (Either CreateAccountError Unit) ❷
createAccount accountsAVar account@(Account { userName }) = do
  withAVar accountsAVar
    \accounts -> pure $
      if Map.member userName accounts then
        Tuple accounts (Left CreateAccountAlreadyExists) ❸
      else
        Tuple (Map.insert userName account accounts) (Right unit) ❹
```

❶ Added this import.

❷ Changed the return Type.

❸ Wrap in a `Left`.

❹ Wrap `unit` in a `Right`. Deleted the `CreateAccountSuccess`.

Make these changes to `Manager.Account`.

And now, while we're at it, let's do the same for the Account Handler's `createHandler` in `Handler.Account`:

```
data CreateAccountError = CreateAccountFileError String ❶ ❷

createAccount :: Account -> Aff (Either CreateAccountError Unit) ❸
createAccount account = lmap show
  <$> (try $ appendTextFile ASCII accountsFile $ accountToCSV account)
  <#> lmap CreateAccountFileError ❹ ❺
```

❶ New Type for errors.

❷ Don't forget to delete `CreateAccountResult`.

❸ Changed return Type.

❹ Removed the `case` expression.

❺ Take the `Either` that `try` returns and wrap the `String` on the `Left` with `CreateAccountFileError`.

Now, let's revisit that line of code and its new Type Signature:

```
AM.createAccount accountsAVar account
  :: Aff (Either CreateAccountError Unit)
```

Again, we'll need to `lift` and then `bind` to `liftEither` to get this into an `ExceptT`:

```
AM.createAccount accountsAVar account
  # lift >>= liftEither
  :: ExceptT CreateAccountError Aff Unit
```

Now, the only thing wrong here is that we have the wrong error Type, `CreateAccountError`. We need a Function to map the error. Search `Pursuit` for `ExceptT` and then browse the page and see if you can find something that'll take an `ExceptT e m a` to `ExceptT e' m a` or something like that. Then return here.

When I did this, I found:

```
withExceptT
  :: ∀ e e' m a
  .  Functor m
  => (e -> e')
  -> ExceptT e m a
  -> ExceptT e' m a
```

While `withExceptT` is a terrible name for something that maps the error, we can add it to our `createAccount` call to convert `CreateAccountError` to `CreateUserFailureReason`:

```
AM.createAccount accountsAVar account
  # lift >>= liftEither
  # (withExceptT $ const AlreadyExists) ❶
  :: ExceptT CreateUserFailureReason Aff Unit
```

❶ We map the error from the one that `createAccount` returns to the appropriate one for our `Response`. We're using `const` since there's only 1 possible error from `createAccount`, i.e. `CreateAccountAlreadyExists` and it doesn't have any Parameters.

Now, let's return to `Handler.Api.CreateUser` and add this line of code, sans the Type Signature, to our Function:

```
import Control.Monad.Except (runExcept, withExceptT) ❶
import Manager.Account (findAccount, createAccount) ❷

handler :: CreateUserRequest -> Handler
handler (CreateUserRequest { authToken, user: user' }) = do
  { accountsAVar, sessionsAVar } <- ask
  result <- lift $ runExceptT do
    Session { userName } <- verifySession sessionsAVar authToken
      <#> note NotAuthenticated # lift >>= liftEither
    Account { admin } <- AM.findAccount accountsAVar userName
      <#> note NotAuthorized # lift >>= liftEither
    unless admin $ throwError NotAuthorized
    passwordHash <- lift $ passwordHashHex user'.userName user'.password
    let user = delete (Proxy :: _ "password") user'
        account = Account $
          insert (Proxy :: _ "passwordHash") passwordHash user
    AM.createAccount accountsAVar account
      # lift >>= liftEither
      # (withExceptT $ const AlreadyExists)
    ???
    pure unit
  HTTPure.notFound
```

❶ Add `withExceptT` to import list.

❷ Removed `CreateAccountResult(..)` from import list.

Now that we've created a new `Account` in memory via `AM.createAccount`, we need to call `AH.createAccount` to create the `Account` on disk. But before we do, let's work on the line outside of the code like before:

```
AH.createAccount account
  :: Aff (Either CreateAccountError Unit)
```

Now, we'll `lift` it and `bind` it to `liftEither`:

```
AH.createAccount account
  # lift >>= liftEither
  :: ExceptT CreateAccountError Aff Unit
```

Now, we'll map `CreateAccountError` to `CreateUserFailureReason` using `withExceptT`:

```
AH.createAccount account
  # lift >>= liftEither
  # withExceptT \(AH.CreateAccountFileError err) -> FileIOError err ❶ ❷
  :: ExceptT CreateUserFailureReason Aff Unit
```

❶ We're fully qualifying the reference to `CreateAccountFileError` to avoid another import.

❷ This is a simple mapping from the one and only error from `createAccount` to our API failure reason.

Now we can add this to our code:

```
handler :: CreateUserRequest -> Handler
handler (CreateUserRequest { authToken, user: user' }) = do
  { accountsAVar, sessionsAVar } <- ask
  result <- lift $ runExceptT do
    Session { userName } <- verifySession sessionsAVar authToken
      <#> note NotAuthenticated # lift >>= liftEither
    Account { admin } <- AM.findAccount accountsAVar userName
      <#> note NotAuthorized # lift >>= liftEither
    unless admin $ throwError NotAuthorized
    passwordHash <- lift $ passwordHashHex user'.userName user'.password
    let user = delete (Proxy :: _ "password") user'
        account = Account $
          insert (Proxy :: _ "passwordHash") passwordHash user
    AM.createAccount accountsAVar account
      # lift >>= liftEither
      # (withExceptT $ const AlreadyExists)
    AH.createAccount account ❶
      # lift >>= liftEither
      # withExceptT \(AH.CreateAccountFileError err) -> FileIOError err
  HTTPure.notFound
```

❶ Removed `pure unit` since this is our last line of the `do` block.

Update your code to match this.

Before we handle the `result`, let's step back for a minute. Look at the code and see where we can improve the readability. Make a mental list before reading on.

The only real improvement I can see is that we can factor out our `# lift >>= liftEither` into a Function called `liftExcept`. If we factored this out and we change the Stack, we'll be glad we only have to make a change in one place.

Stop reading and write just the Type Signature for `liftExcept` using the following individual Type Signatures to help you:

```
lift :: ∀ m a t. MonadTrans t => Monad m => m a -> t m a
liftEither :: ∀ e m a. MonadError e m => Either e a -> m a
```

Return here when you've finished.

Okay, so this Type Signature can be a mind bender. Let's see if the compiler will help us out. Let's skip this for now.

Continue writing the code with NO Type Signature keeping in mind that `liftExcept` will get the same input as `lift` and then return here.

`liftExcept` will get a Monadic Value of some sort:

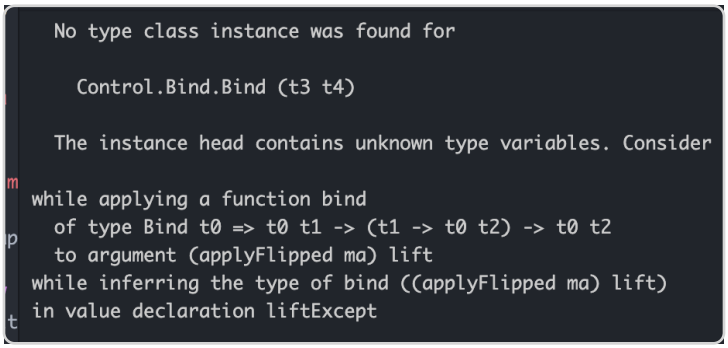
```
liftExcept ma = ???
```

And now we just feed it into our `# lift >>= liftEfect`:

```
liftExcept ma = ma # lift >>= liftEither -- COMPILER ERROR!!
```

Unfortunately, the compiler isn't going to just hand us our Type Signature on a Silver Platter. We're going to have to work for it.

Here's the error:



It needs something to have a `Bind` Instance. This make sense to us since we know we're working with `Monads`. But we also know that we're working with `Either` and we're also working with `ExceptT`, which is `MonadError` and `MonadTrans`.

This error doesn't help us because the Functions we're using are too general for it to infer anything of substance.

I'm going to show you a technique for sussing the Type Signature by feeding the compiler one thing at a time. We'll react to the errors and follow it, hopefully, to a pretty general Type Signature.

We have to give the Type Signature something. We know that `liftExcept` takes only 1 Parameter. We're going to give the compiler the most naïve Type Signature for Functions of 1 Parameter.

It's going to clearly object and give us errors that we'll use as clues to determine the Type Signature. Then as we add Constraints, we'll rename our Type Parameters so that they make more sense to us.

Please follow along one step at a time with me so you can truly understand how to do this.

Start with this code and this naïve Type Signature for a Function that takes 1 Parameter:

```
liftExcept :: ∀ x y. x -> y
liftExcept ma = ma # lift >>= liftEither
```

Now, check the compiler error. I get:

```
Could not match type x0 with type t2 t3
```

Seems as if `x` takes a Parameter. So, let's give it `a`:

```
liftExcept
  :: ∀ a x y
  .  x a
  -> y
```

Error:

```
Could not match type a0 with type Either t4 t5
```

Seems like `a` should be `Either e a`. Okay, let's change `x's` first Parameter from `a` to `Either e a`:

```
liftExcept
  :: ∀ e a x y
  .  x (Either e a)
  -> y
```

Error:

```
Could not match type t1 x2 a4 with type y0
```

Seems like `y` needs 2 Parameters where `x` is the first Parameter and `a` is the second:

```
liftExcept
  :: ∀ e a x y
  .  x (Either e a)
  -> y x a
```

Error:

```
Control.Bind.Bind (y3 x4)
```

So, `y x` needs a `Bind`. We know we're working with `Monads` so we'll add a `Monad` Constraint instead:

```
liftExcept
  :: ∀ e a x y
  .  Monad (y x)
  => x (Either e a)
  -> y x a
```

Error:

```
Control.Monad.Trans.Class.MonadTrans y4
```

Looks like `y` is a `MonadTrans`, i.e. a Monad Transformer. This should not be surprising since we're working with `ExceptT` and we have a `throwError` in `liftEither`.

Let's add the Constraint and rename it to `t`:

```
import Control.Monad.Trans.Class (class MonadTrans, lift) ❶

liftExcept
  :: ∀ e a x t
  .  MonadTrans t
  => Monad (t x)
  => x (Either e a)
  -> t x a
```

❶ Added `MonadTrans` to import list.

Error:

```
Control.Monad.Monad x0
```

Looks like `x` is a `Monad`. Let's add that Constraint and rename it to `m`:

```
liftExcept
  :: ∀ e a m t
  .  Monad m
  => MonadTrans t
  => Monad (t m)
  => m (Either e a)
  -> t m a
```

Error:

```
Control.Monad.Error.Class.MonadError e0 (t5 m3)
```

Looks like `t m` is a `MonadError`. Let's add that and REMOVE the `Monad (t m)` Constraint since it'll be covered by the `MonadError` Constraint:

```
liftExcept
  :: ∀ e a m t
  .  MonadError e (t m)
  => Monad m
  => MonadTrans t
  => m (Either e a)
  -> t m a
liftExcept ma = ma # lift >>= liftEither
```

And this compiles!!

When the compiler doesn't give us a Type Signature or if Type Holes in the Type Signature don't work because of missing Constraints, then this technique is pretty easy.

In our case, we know that `t` is `ExceptT e` which means that `t m a` is `ExceptT e m a` where `m` is a `Monad`. We could've use a Monomorphic Type for `t`, i.e. `ExceptT e`, but having this Polymorphic is always better since we could always use this elsewhere.

But now that we have this as a generalized Function with only `t` in the Type Signature, the name `liftExcept` seems wrong.

But what is a good name?

Naming is really hard but if we can think about what this Function does, we may come up with a better name for it.

When we look at the Type Signature for this Function, we can see that our input Type is `m (Either e a)` and our output Type is `t m a`. Notice that the `e` is missing. That's because the error case, i.e. the `e`, is thrown inside of `liftEither`.

We are only getting the success case back here. And because of that fact, I'm going to rename this Function, `liftSuccess`. This isn't a perfect name, but is far better than my first name since it doesn't have `Except` and it conveys that we're working with the success case, i.e. NOT the `e` but the `a`:

```
liftSuccess
  :: ∀ e a m t
  .  MonadError e (t m)
  => Monad m
  => MonadTrans t
  => m (Either e a)
  -> t m a
liftSuccess ma = ma # lift >>= liftEither
```

Before moving on, make sure your version matches this one with it's new name and change `handler` to use `liftSuccess` instead of `lift >>= liftEither` everywhere before continuing further.

Here are those changes:

```
handler :: CreateUserRequest -> Handler
handler (CreateUserRequest { authToken, user: user' }) = do
  { accountsAVar, sessionsAVar } <- ask
  result <- lift $ runExceptT do
    Session { userName } <- verifySession sessionsAVar authToken
      <#> note NotAuthenticated # liftSuccess ❶
    Account { admin } <- AM.findAccount accountsAVar userName
      <#> note NotAuthorized # liftSuccess ❶
    unless admin $ throwError NotAuthorized
    passwordHash <- lift $ passwordHashHex user'.userName user'.password
    let user = delete (Proxy :: _ "password") user'
        account = Account $
          insert (Proxy :: _ "passwordHash") passwordHash user
    AM.createAccount accountsAVar account
      # liftSuccess ❶
      # (withExceptT $ const AlreadyExists)
    AH.createAccount account
      # liftSuccess ❶
      # withExceptT \(AH.CreateAccountFileError err) -> FileIOError err
  HTTPure.notFound
```

❶ Made the edits in these 4 places.

And finally, we can remove `HTTPure.notFound` and replace it with a proper response. Use the helper, `either`, on `result` to accomplish this. Then continue reading.

I'm only going to show just the last part of `handler`, i.e. the code that's replacing `HTTPure.notFound`. Here's a template for what I'm going to code:

```
  -- ABOVE CODE REDACTED
  result # either
    (\reason -> ???) ❶
    (const $ ???) ❷
```

❶ This is the `Left` Function. This will build an error response from the error `reason`, which is of Type `CreateUserFailureReason`.

❷ This is the `Right` Function. When the operation is successful, we'll get a `Right unit`, so we're going to use `const` here to produce a success response.

If you got stuck, give it another try and see if you can finish. Stop reading and return here when you're ready to move on.

Let's work on the error case:

```
import Foreign.Generic (encodeJSON) ❶

  -- ABOVE CODE REDACTED
  result # either
    (\reason -> HTTPure.ok $ encodeJSON
      $ CreateUserResponse $ CreateUserResultsFailure { reason }) ❷
    (const $ ???)
```

❶ Added import.

❷ We construct the failure response and then encode it into JSON and then send it using `HTTPure.ok`.

Next, the success case:

```
  -- ABOVE CODE REDACTED
  result # either
    (\reason -> HTTPure.ok $ encodeJSON
      $ CreateUserResponse $ CreateUserResultsFailure { reason })
    (const $ HTTPure.ok $ encodeJSON
      $ CreateUserResponse CreateUserResultsSuccess)
```

Add this code to your handler in place of `HTTPure.notFound`. It should compile.

Now, I don't like the repetitive bit of code, i.e. `HTTPure.ok $ encodeJSON $ CreateUserResponse $`. Factor out the repetitive parts into a local Function called `ok`. Then return here.

First, we'll factor out the repetitive code on each line:

```
  -- ABOVE CODE REDACTED
  result # either
    (HTTPure.ok <<< encodeJSON <<< CreateUserResponse
      <<< \reason -> CreateUserResultsFailure { reason })
    (HTTPure.ok <<< encodeJSON <<< CreateUserResponse
      <<< const CreateUserResultsSuccess)
```

Now we can factor out the Function Composition and change it to Function Application and then call `HTTPure.ok $ encodeJSON $ CreateUserResponse` on `either`:

```
  -- ABOVE CODE REDACTED
  HTTPure.ok $ encodeJSON $ CreateUserResponse
    $ result # either
      (\reason -> CreateUserResultsFailure { reason })
      (const CreateUserResultsSuccess)
```

Here's the full Function:

```
handler :: CreateUserRequest -> Handler
handler (CreateUserRequest { authToken, user: user' }) = do
  { accountsAVar, sessionsAVar } <- ask
  result <- lift $ runExceptT do
    Session { userName } <- verifySession sessionsAVar authToken
      <#> note NotAuthenticated # liftSuccess
    Account { admin } <- AM.findAccount accountsAVar userName
      <#> note NotAuthorized # liftSuccess
    unless admin $ throwError NotAuthorized
    passwordHash <- lift $ passwordHashHex user'.userName user'.password
    let user = delete (Proxy :: _ "password") user'
        account = Account $
          insert (Proxy :: _ "passwordHash") passwordHash user
    AM.createAccount accountsAVar account
      # liftSuccess
      # (withExceptT $ const AlreadyExists)
    AH.createAccount account
      # liftSuccess
      # withExceptT \(AH.CreateAccountFileError err) -> FileIOError err
  HTTPure.ok $ encodeJSON $ CreateUserResponse
    $ result # either
      (\reason -> CreateUserResultsFailure { reason })
      (const CreateUserResultsSuccess)
```

Make sure that your code is up to date and compiles.

Then clean up the redundant imports we have thanks to the commented out version of `handler`.

Now, let's take a minute to compare it to our old code:

```
handler :: CreateUserRequest -> Handler
handler (CreateUserRequest { authToken, user: user' }) = do
  { accountsAVar, sessionsAVar } <- ask
  verifiedSession <- lift $ verifySession sessionsAVar authToken
  response <- case verifiedSession of ❶
    Nothing -> CreateUserResponse
      $ CreateUserResultsFailure { reason: NotAuthenticated }
    Just (Session { userName }) -> do
      admin <- lift $ AM.findAccount accountsAVar userName
        <#> maybe false (\(Account a) -> a.admin) ❶
      if not admin then pure CreateUserResponse ❶
        $ CreateUserResultsFailure { reason: NotAuthorized }
      else do
        passwordHash <- lift
          $ passwordHashHex user'.userName user'.password
        let user = delete (Proxy :: _ "password") user'
            account = Account $
              insert (Proxy :: _ "passwordHash") passwordHash user
        result <- lift $ AM.createAccount accountsAVar account
        case result of ❶
          AM.CreateAccountAlreadyExists -> CreateUserResponse
            $ CreateUserResultsFailure { reason: AlreadyExists }
          AM.CreateAccountSuccess -> do
            result <- lift $ AH.createAccount account
            pure $ case result of ❶
              AH.CreateAccountFileError err -> CreateUserResponse
                $ CreateUserResultsFailure { reason: FileIOError err }
              AH.CreateAccountSuccess -> CreateUserResultsSuccess
  ??? ❷
```

❶ These 5 places we're doing a check for errors.

❷ We never finished this.

Take a moment to look over the old code. Notice how many places we're explicitly doing some sort of `if` check for errors. I counted 5.

Now, look over the new code. In the new code, we're only explicitly doing 1 `if` check using `unless`.

Before we move on, let's factor out the helper Functions, `liftEither` and `liftSuccess`. Let's move them into our `Utils` module for a lack of a better location.

Return here when you've finished factoring them out.

Here's those additions to `Utils.purs`:

```
import Data.Either (Either, either)
import Control.Monad.Error.Class (class MonadError, throwError)
import Control.Monad.Trans.Class (class MonadTrans, lift)

liftEither :: ∀ e m a. MonadError e m => Either e a -> m a
liftEither = either throwError pure

liftSuccess
  :: ∀ e a m t
  .  MonadError e (t m)
  => Monad m
  => MonadTrans t
  => m (Either e a)
  -> t m a
liftSuccess ma = ma # lift >>= liftEither
```

Now, return to `Handler.Api.CreateUser` and add this import:

```
import Utils (liftSuccess)
```

Now's a good time to fix all of the import warnings before we test our code.

We're now ready to test. Run the server with:

```
npx spago run
```

Then go into `Postman`, open up a new tab and then click on the `Body` tab and then the `raw` radio button. Then enter:

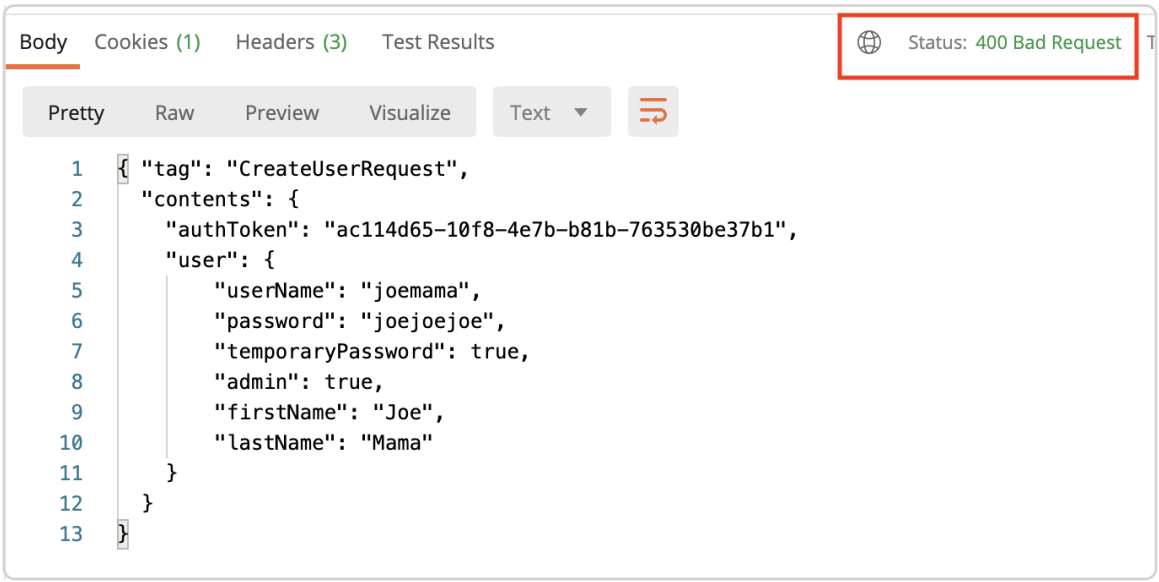
```
{ "tag": "CreateUserRequest",
  "contents": {
    "authToken": "AUTH_TOKEN_FROM_LOGON_RESPONSE", ❶
    "user": {
        "userName": "joemama",
        "password": "joejoejoe",
        "temporaryPassword": true,
        "admin": true,
        "firstName": "Joe",
        "lastName": "Mama"
    }
  }
}
```

❶ Put the `authToken` you get back from `LogonRequest` here.

Don't forget to do a `LogonRequest` first. Then take the `authToken` from the `LogonResponse` and edit the `CreateUserRequest`.

Make sure it's set to `POST` and that the URL is `localhost:3000`. Then click `Send` and see what you get back.

I got:



Why did we get this?

That's because I forgot to add `CreateUser` to the `router` in `Main`. Do that now and then return here.

Here's that change:

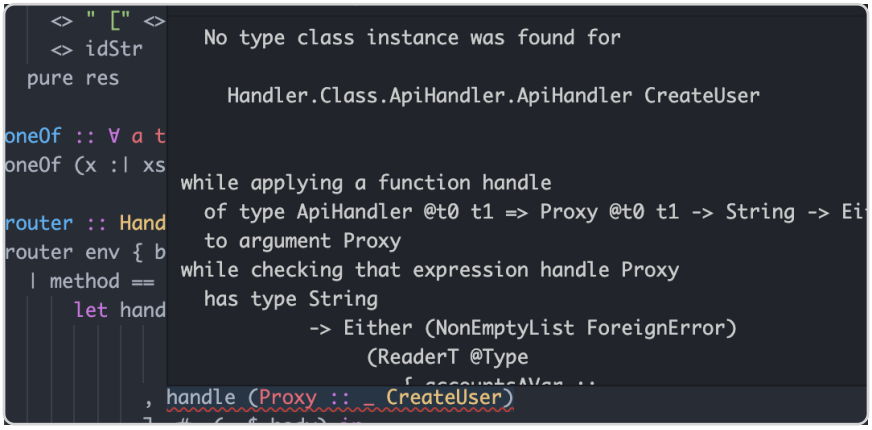
```
import Handler.Api.CreateUser (CreateUser) ❶

router :: HandlerEnv -> Request -> ResponseM
router env { body, method }
  | method == HTTPure.Post =
      let handlers =
              handle (Proxy :: _ Logon) :| [
              handle (Proxy :: _ Logoff)
              -- COMPILER ERROR!!
            , handle (Proxy :: _ CreateUser) ❷
            ] <#> (_ $ body) in
      case hush $ oneOf handlers of
        Nothing -> HTTPure.badRequest body
        Just reader -> runReaderT reader env
  | otherwise = HTTPure.methodNotAllowed
```

❶ Import `CreateUser`.

❷ Added this handler.

The compiler error is:



Go look at the code in `Handler.Api.CreateUser` to see if you can see the problem.

Turns out we forgot to uncomment out the `ApiHandler` instance code. Do that now:

```
import Handler.Api.Common (handleApi) ❶
import Handler.Class.ApiHandler (class ApiHandler, Handler) ❷

instance apiHandlerCreateUser :: ApiHandler CreateUser where
  handle _ = handleApi handler
```

❶ Added back this import. We removed it when we removed redundant imports while our instance code was commented out.

❷ Added `class ApiHandler` back to the import list for the same reason.

Now run the test again. Don't forget to send the `LogonRequest` first to get an `authToken`.

What did you get?

This is what I got:

```
{"contents":{"tag":"CreateUserResultsSuccess"},"tag":"CreateUserResponse"}
```

Now we should check to make sure that the data was written correctly to `accounts.csv`. Go look and see how it looks and then return here.

So, unfortunately, we missing a newline between the records. We'll need to fix that.

But wait... I just remembered that `accounts.csv` is not part of our code base. If we are going to manage this project with `git`, we should add this file to `.gitignore` so it doesn't get checked into our repository:

```
/bower_components/
/node_modules/
/.pulp-cache/
/output/
/generated-docs/
/.psc-package/
/.psc*
/.purs*
/.psa*
/.spago
accounts.csv
```

In VSCode, `accounts.csv` should be grayed out in the file explorer to the left.

Now let's fix our bug by simply adding a newline at the end of the CSV line. Go into `Handler.Account` and make that change before reading on.

Here's the one line code change:

```
accountToCSV :: Account -> String
accountToCSV (Account
                { userName
                , passwordHash
                , temporaryPassword
                , admin
                , firstName
                , lastName }) =
  intercalate ","
    [ userName
    , passwordHash
    , show temporaryPassword
    , show admin
    , firstName
    , lastName
    ] <> "\n" ❶
```

❶ Appended `\n` to the CSV line.

Now, we've got a bad `accounts.csv`. So delete it. Don't worry our server will recreate it if it's missing.

Now, kill your server and rerun it with:

```
npx spago run
```

Now, rerun the test then go check the file to see how it looks. Then return here.

Okay, that didn't work for me. It put a newline on the end of the `CreateUser` account, but it's missing from the first line, i.e. our `bootstrapAccount`.

Let's look at how the `bootstrapAccount` is created:

```
bootstrapAccount :: Aff String
bootstrapAccount = do
  let userName = "admin"
      password = "admin"
  passwordHash <- passwordHashHex userName password
  let true' = show true
  pure $ intercalate ","
    [ userName , passwordHash , true' , true' , "Joe", "Admin" ]
```

And now it's clear that this is NOT using the `accountToCSV` Function. It should be. Change

`bootstrapAccount` to use `accountToCSV` before reading further.

We need to construct an `Account` from the bootstrap data and then call `accountToCSV`:

```
bootstrapAccount :: Aff String
bootstrapAccount = do
  let userName = "admin"
      password = "admin"
  passwordHash <- passwordHashHex userName password
  pure $ accountToCSV $ Account ❶
    { userName ❷
    , passwordHash
    , temporaryPassword: true
    , admin: true
    , firstName: "Joe"
    , lastName: "Admin"
    }
```

❶ Pass an `Account` to `accountToCSV`.

❷ We don't need to use `userName: userName` here since the field and the Variable have the same name. Same goes for `passwordHash`.

Now, delete `accounts.csv`.

Rerun the server and retest and see how our file looks at the end. Then return here.

Well, that didn't go well. Here's what I got:

```
Cannot load accounts: (ParseError "Unexpected EOF" (Position { line: 1,
column: 1 }))
```

Take a look at `accounts.csv` and see if you can see what's wrong. Did you figure it out?

It's not very clear what's happening. But I suspect that 2 things are going on.

One, we're not ignoring empty lines. We have one of those now at the end of the file.

And two, we'll always get an error message with `line: 1` since we call the Parser once per line passing only the current line.

We could really improve our error messages if we wrote a parser to parse the whole file with the end of line as part of the parser. This way we'll get a proper line number when the parser fails.

28.10.3. Improving Our Parser

Here's the parser we're using:

```
accountParser :: AccountParserT Account
accountParser = do
  userName' <- userName # comma
  passwordHash' <- passwordHash # comma
  temporaryPassword' <- temporaryPassword # comma
  admin' <- admin # comma
  firstName' <- firstName # comma
  lastName' <- lastName
  pure $ Account
    { userName: userName'
    , passwordHash: passwordHash'
    , temporaryPassword: temporaryPassword'
    , admin: admin'
    , firstName: firstName'
    , lastName: lastName'
    }
  where
    comma :: ∀ a. AccountParserT a -> AccountParserT a
    comma p = p <* char ','
```

This will parse only 1 line. What we'd like is something that will parse the full file so we can get better error messages.

Take a look at `sepBy` in `Text.Parsing.Parser.Combinators`. There's a good example in the docs that we can steal from, but we're going to want to use `char` instead `string`.

`sepBy` takes 2 parsers. The first parser is the parser for anything, but the second parser parses the separator.

For us, this will be `accountParser` and our separation parser is a character parser for `\n`.

For now, just write the Type Signature for our new parser in `Parser.Account`, called `accountsParser` then return here:

Our Type Signature is:

```
accountsParser :: AccountParserT (Array Account) ❶
```

❶ `sepBy` returns a `List` but we'll convert it to an Array at the end.

`accountsParser` is a parser that returns an `Array` of `Accounts`.

And now write the rest of the parser until you get stuck or finish. Then keep reading.

The next part is to use `sepBy`:

```
import Text.Parsing.Parser.Combinators (sepBy) ❶

accountsParser :: AccountParserT (Array Account)
-- COMPILER ERROR!!
accountsParser = accountParser `sepBy` char '\n' ❷
```

❶ Added this import.

❷ We're using `sepBy` infixed for readability.

The compiler error is because we haven't converted the `List` that `sepBy` gives us to an `Array` to match our Type Signature.

If you didn't finish stop now and make this final change before reading on.

```
import Data.Array (many, some, (:), fromFoldable) ❶

accountsParser :: AccountParserT (Array Account)
accountsParser = accountParser `sepBy` char '\n' <#> fromFoldable ❷
```

❶ Added `fromFoldable` to import list.

❷ Map `fromFoldable` over the parser. This takes the `List` that `sepBy` gives us and maps it to an `Array`.

Now, we need to find where `accountParser` is currently called and replace it with `accountsParser`.

Search your code for where it's called, then return here.

I found it here in `Handler.Account`:

```
loadAccounts :: Aff (Either ParseError (Array Account))
loadAccounts = do
  exists' <- try $ exists accountsFile
  let exists = case exists' of
        Left _ -> false
        Right b -> b
  unless exists do
      bsa <- bootstrapAccount
      writeTextFile ASCII accountsFile bsa
  accountLines <- lines <$> readTextFile ASCII accountsFile ❶
  pure $ sequence
    $ unwrap <<< flip runParserT accountParser <$> accountLines ❷
```

❶ This line needs to be rewritten since we don't want to break the file up into lines anymore since we're going to parse the whole thing.

❷ This line needs to be rewritten to use our new parser.

First step is to comment out the lines we're going to rewrite and then add something to the end of this Function to keep the compiler happy.

Go ahead and do that before returning here.

Here's those changes:

```
loadAccounts :: Aff (Either ParseError (Array Account))
loadAccounts = do
  exists' <- try $ exists accountsFile
  let exists = case exists' of
        Left _ -> false
        Right b -> b
  unless exists do
      bsa <- bootstrapAccount
      writeTextFile ASCII accountsFile bsa
  -- accountLines <- lines <$> readTextFile ASCII accountsFile ❶
  -- pure $ sequence
  --   $ unwrap <<< flip runParserT accountParser <$> accountLines ❶
  pure $ Right [] ❷
```

❶ Commented this out since we're going to rewrite them.

❷ Added an empty `Array` wrapped in an `Either` using `Right` and then wrapped in `Aff` with `pure`.

Now, read the file data but this time don't break them up into lines using `lines`. Put it into a Variable called `fileData`. Make this one change and then return here.

Here's that change:

```
loadAccounts :: Aff (Either ParseError (Array Account))
loadAccounts = do
  exists' <- try $ exists accountsFile
  let exists = case exists' of
        Left _ -> false
        Right b -> b
  unless exists do
      bsa <- bootstrapAccount
      writeTextFile ASCII accountsFile bsa
  fileData <- readTextFile ASCII accountsFile ❶
  -- accountLines <- lines <$> readTextFile ASCII accountsFile
  -- pure $ sequence
  --  $ unwrap <<< flip runParserT accountParser <$> accountLines
  pure $ Right []
```

❶ Here we read the whole file into `fileData` with no post-processing.

Next, add the following to the code:

```
loadAccounts :: Aff (Either ParseError (Array Account))
loadAccounts = do
  exists' <- try $ exists accountsFile
  let exists = case exists' of
        Left _ -> false
        Right b -> b
  unless exists do
      bsa <- bootstrapAccount
      writeTextFile ASCII accountsFile bsa
  fileData <- readTextFile ASCII accountsFile
  let x = ??? :: ?x ❶
  -- accountLines <- lines <$> readTextFile ASCII accountsFile
  -- pure $ sequence
  --  $ unwrap <<< flip runParserT accountParser <$> accountLines ❷
  pure $ Right []
```

❶ Add this line minus the `???`.

❷ Use this line for reference.

The `???` is where you're going to incubate the parsing code we're going to eventually replace the `pure $ Right []` with.

The Type Hole will help us keep track of our Types.

Go ahead and add code where the `???` is to run `accountsParser` using `runParserT` before reading on.

Here's just that:

```
import Parser.Account (accountsParser) ❶

loadAccounts :: Aff (Either ParseError (Array Account))
loadAccounts = do
  exists' <- try $ exists accountsFile
  let exists = case exists' of
        Left _ -> false
        Right b -> b
  unless exists do
      bsa <- bootstrapAccount
      writeTextFile ASCII accountsFile bsa
  fileData <- readTextFile ASCII accountsFile
  let x = runParserT fileData accountsParser :: ?x ❷
  -- accountLines <- lines <$> readTextFile ASCII accountsFile
  -- pure $ sequence
  --  $ unwrap <<< flip runParserT accountParser <$> accountLines
  pure $ Right []
```

❶ Added `accountsParser` to the import list. Also, removed `accountParser`.

❷ The Type Hole is `Identity (Either ParseError (Array Account))`.

Now, add code to our test line to eliminate the `Identity`. Then return to reading.

Here's just that addition:

```
loadAccounts :: Aff (Either ParseError (Array Account))
loadAccounts = do
  exists' <- try $ exists accountsFile
  let exists = case exists' of
        Left _ -> false
        Right b -> b
  unless exists do
      bsa <- bootstrapAccount
      writeTextFile ASCII accountsFile bsa
  fileData <- readTextFile ASCII accountsFile
  let x = unwrap $ runParserT fileData accountsParser :: ?x ❶
  -- accountLines <- lines <$> readTextFile ASCII accountsFile
  -- pure $ sequence
  --  $ unwrap <<< flip runParserT accountParser <$> accountLines
  pure $ Right []
```

❶ The Type Hole is `Either ParseError (Array Account)`.

We almost have what we need. You can see this by looking at what `loadAccounts` is supposed to return.

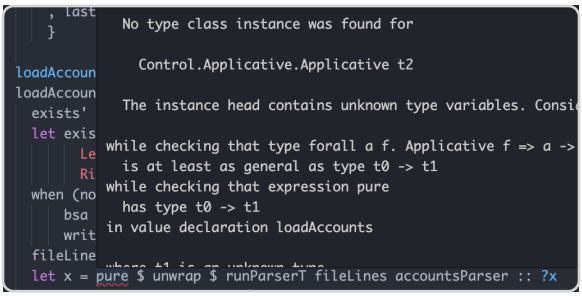
Add the last bit of code to make that happen and then keep reading.

Here's that change:

```
loadAccounts :: Aff (Either ParseError (Array Account))
loadAccounts = do
  exists' <- try $ exists accountsFile
  let exists = case exists' of
        Left _ -> false
        Right b -> b
  unless exists do
      bsa <- bootstrapAccount
      writeTextFile ASCII accountsFile bsa
  fileData <- readTextFile ASCII accountsFile
  --- COMPILER ERROR!!
  let x = pure $ unwrap $ runParserT fileData accountsParser :: ?x ❶
  -- accountLines <- lines <$> readTextFile ASCII accountsFile
  -- pure $ sequence
  --  $ unwrap <<< flip runParserT accountParser <$> accountLines
  pure $ Right []
```

❶ Added `pure`.

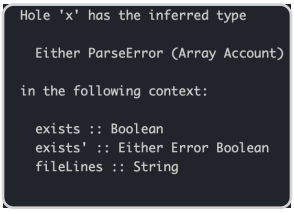
Here's the compiler error:



It underlines the `pure` since it cannot determine the `Applicative`. That's because we've put this in a `let` statement, so it could be any `Applicative`.

Let's tell the compiler which `Applicative` by changing the Type hole from `?x` to `Aff ?x`. Do this and see what you get.

Here's what I get for the Type Hole:



That means that our incubated line of code has Type: `Aff (Either ParseError (Array Account))`, which is exactly what we need.

Replace the final line with the incubated code and then remove the `let` line. Then return here and continue on.

Here's that final change:

```
loadAccounts :: Aff (Either ParseError (Array Account))
loadAccounts = do
  exists' <- try $ exists accountsFile
  let exists = case exists' of
        Left _ -> false
        Right b -> b
  unless exists do
      bsa <- bootstrapAccount
      writeTextFile ASCII accountsFile bsa
  fileData <- readTextFile ASCII accountsFile
  -- accountLines <- lines <$> readTextFile ASCII accountsFile
  -- pure $ sequence $ unwrap <<< flip runParserT accountParser <$>
accountLines
  pure $ unwrap $ runParserT fileData accountsParser ❶
```

❶ Here's the line we incubated in a `let` definition with a Type Hole.

At this point, you'll be able to remove the following imports:

```
import Data.String.Utils (lines)
import Data.Traversable (sequence)
```

Now run the server and see what the error message says now.

First I got:

```
[warn] None of your project files import modules from some projects that are
in the direct dependencies of your project.
These dependencies are unused. To fix this warning, remove the following
packages from the list of dependencies in your config:
- stringutils
```

Remove `stringutils` from `spago.dhall`.

After that I got:

```
Cannot load accounts: (ParseError "Unexpected EOF" (Position { line: 2,
column: 1 })) ❶
```

❶ Notice the line number is 2 now. That's because we're parsing all the lines not just one at a time. This is far better since we know which line in the file is a problem.

It was just as I suspected. Empty lines are a problem.

So, let's allow for an empty line in our parser.

Let's look back at our parser:

```
accountsParser :: AccountParserT (Array Account)
accountsParser = accountParser `sepBy` char '\n' <#> fromFoldable
```

We need to parse with `accountParser` OR a parser that parses nothing. Do you remember how to do an OR with parsers?

How about how to select between 2 parsers?

To do this we use `alt` or more commonly we use it's Binary Operator, `<|>`.

Now, we need a parser that doesn't consume anything from the `String`. Can you remember what we wrote like this when we wrote our `Parser`?

If you cannot remember, try to imagine that there is a parser that consume nothing from the `String`. What could that parser possibly return?

It can only return a Value of Type `Unit`. It cannot conjure up any other Type. We want this parser to always return `unit`. So all we have to do is wrap it up in a parser using `pure`.

If this doesn't readily make sense, here's our implementation for `pure` when we wrote our `Parser`:

```
instance applicativeParser :: Applicative (Parser e) where
  pure x = Parser \s -> pure $ Tuple s x ❶
```

❶ Notice how nothing is consumed from the `String`, `s`. And the Value`, `x`, is simply returned as the result of this `Parser`.

The only problem now is that `accountParser` is of Type `AccountParserT Account` and `pure unit` is of Type `AccountParserT Unit`. We cannot use these together with `<|>` since the Types don't match.

How can we unify these Types?

We could turn `accountParser` into an `AccountParserT Unit`, but then we'd lose the account information we want. And we cannot turn `pure unit` into an `AccountParserT Account`.

The best we can do is turn them both into `AccountParserT (Maybe Account)`, where `accountParser` will return a `Just` and `pure unit` a `Nothing`.

That means we need to map over each of our parsers before we use `<|>`.

Make this code change to `accountsParser` in `Parser.Account` even though it won't compile since we're not done. Then return here to continue.

Here's that change:

```
import Data.Maybe (Maybe(..)) ❶

accountsParser :: AccountParserT (Array Account)
accountsParser =
  -- COMPILER ERROR!!
  (Just <$> accountParser <|> const Nothing <$> pure unit) <2> ❸ ❹
    `sepBy` char '\n' <#> fromFoldable
```

❶ Added this import.

❷ We map `Just` over `accountParser` to get a parser of Type `AccountParserT (Maybe Account)`.

❸ We map `const Nothing` over `pure unit` to get a parser of Type `AccountParserT (Maybe a)`. The `a` will unify with `Account`.

❹ We using `const` to discard the `unit` returned from `pure unit`.

But wait. Look at `const Nothing <$> pure unit`. Can you see how we can reduce this?

Since `const Nothing <$> pure unit` applies `const Nothing` to the `unit` in the `Applicative`, we can just do `pure Nothing`:

```
accountsParser :: AccountParserT (Array Account)
accountsParser =
  -- COMPILER ERROR!!
  (Just <$> accountParser <|> pure Nothing) ❶
    `sepBy` char '\n' <#> fromFoldable
```

❶ Use `pure Nothing`, which is a `AccountParserT (Maybe a)`.

We expected the compiler error since now we have an `AccountParserT (Array (Maybe Account))`. We need to get rid of the `Maybes` in our `Array` while throwing away the `Nothings`.

Do you remember a Function that does that?

Go search `Pursuit` for `Array (Maybe a) -> Array a`.

Here's why I found:

```
catMaybes :: ∀ a. Array (Maybe a) -> Array a
```

This is exactly what we need. Go ahead add this to your code then keep reading.

Here's that final change:

```
import Data.Array (many, some, (:), fromFoldable, catMaybes) ❶

accountsParser :: AccountParserT (Array Account)
accountsParser =
  (Just <$> accountParser <|> pure Nothing)
    `sepBy` char '\n' <#> catMaybes <<< fromFoldable ❷
```

❶ Added `catMaybes` to the import list.

❷ `catMaybes` AFTER we've converted the `List` to an `Array`. Thanks to the Functor Laws, we can save a call to map by composing.

Now, run your server and see what happens.

If your code is like mine, then your server should be running.

Now, rerun the test, i.e. send `LogonRequest` first. Then take the `authToken` and copy and paste it into the `CreateUserRequest`. Then send that.

After that, check `accounts.csv`. It should have our new account on it's own line.

Now, before we shutdown the server, try to send the `CreateUserRequest` one more time and see what you get.

Here's what I got:

```
{"contents":{"contents":{"reason":{"tag":"AlreadyExists"}},"tag":"CreateUserResultsFailure"},"tag":"CreateUserResponse"}
```

Now, shutdown the server and restart it to make sure that it reads `accounts.csv` correctly.

And it does!

28.10.4. Coding `QueryUsers` API Handler

Querying the users should be pretty easy since we have everything in place for what we'll need. Our basic plan will be to get the Accounts from the Account Manager and convert them from `Accounts` to `Users` and then return that to the client.

To prepare to write the API handler, we'll first write a Function called `getAccounts` in `Manager.Account`.

Write only the Type Signature and then continue reading.

Here's the Type Signature:

```
getAccounts :: AVar Accounts -> Aff (Array Account)
```

That Type Signature says it all. We give the Function an `AVar` of `Accounts`, which, don't forget, is a `Map`, and it'll give us back just an `Array` of the `Accounts`.

Take the time to write this now. Then return here if you get stuck or when you're finished.

We'll write the left-hand side of the equal sign next:

```
getAccounts :: AVar Accounts -> Aff (Array Account)
getAccounts accountsAVar = ???
```

And now we can just use `AVar.read` to get the `Array` of `Accounts`:

```
getAccounts :: AVar Accounts -> Aff (Array Account)
getAccounts accountsAVar = AVar.read accountsAVar >>= ??? ❶
```

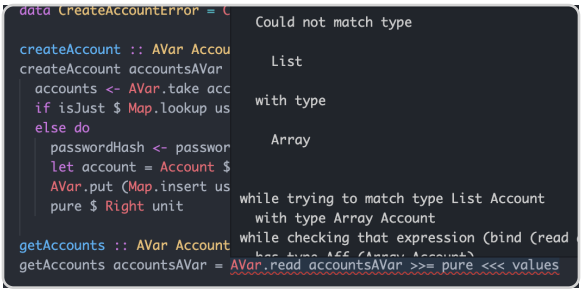
❶ We'll want to `bind` to a Function that will extract out just the Values of the `Map`.

And now, we'll use `values` to extract just the values:

```
getAccounts :: AVar Accounts -> Aff (Array Account)
getAccounts accountsAVar =
  AVar.read accountsAVar >>= pure <<< Map.values -- COMPILER ERROR!! ❶
```

❶ Don't forget to wrap the results in `Aff` using `pure`. We are using `bind` after all.

And now we have the following compiler error:



Can you see the problem?

Looks like `values` returns a `List` not an `Array`. If you like, you can look it up. But, I'm just going to add `fromFoldable` into the mix and see if that fixes it:

```
import Data.Array (fromFoldable) ❶

getAccounts :: AVar Accounts -> Aff (Array Account)
getAccounts accountsAVar =
  AVar.read accountsAVar >>= pure <<< fromFoldable <<< Map.values ❷
```

❶ Import `fromFoldable`.

❷ Convert the `List` that `Map.values` gives us to an `Array` using `fromFoldable`.

And this fixes it.

At this point you might be wondering why even write this Function. We could just call `AVar.read` in our code and you'd be right. But this approach is nice for 3 reasons.

First, it abstracts away the implementation details. We don't have to know how the `Accounts` are stored in memory.

Second, if we decided to change where we store the `Accounts`, we'll have minimal changes to make to our code.

And finally, we don't have to figure out how to write this complex line of code every time we want to get the accounts.

Next, create a File under `src/Handler/Api` called `QueryUsers.purs` and code up a the `ApiHandler` Instance stubbing out the `handler` Function for now. Use all of the code we've written so far as reference.

Then return here to see how you did.

We'll start with:

```
module Handler.Api.QueryUsers where

import Prelude

import Data.Api.QueryUsers (QueryUsersRequest(..), QueryUsersResponse(..)
                          , QueryUsersResults(..)
                          , QueryUsersFailureReason(..))
import Handler.Class.ApiHandler (class ApiHandler, Handler)
import Handler.Api.Common (handleApi)
import HTTPure as HTTPure

data QueryUsers = QueryUsers

handler :: QueryUsersRequest -> Handler
handler _ = HTTPure.notFound ❶

instance apiHandlerQueryUsers :: ApiHandler QueryUsers where
  handle _ = handleApi handler
```

❶ This is stubbed out.

The instance implementation was copy and pasted from `CreateUser's` with the obvious edits to change `CreateUser` to `QueryUsers`.

Next, the imports were stolen once again from `CreateUser` and edited where necessary.

The `QueryUsers` data Type is following the standard pattern we established for API Handlers.

And finally, `handler` is a dummy version that always sends a `404`. It's just here to keep the compiler happy.

We have some warnings about redundant imports, but we're going to ignore them for now because we plan on needing them eventually. And if not, we can clean them up in the end.

Now, flesh out `handler` by getting the `Accounts` from the Account Manager, converting them to `Users` by using `delete` from the module `Record` and finally responding back to the client.

Don't forget to handle all the possible errors, i.e. `NotAuthenticated` and `NotAuthorized`.

Make sure to look back at all of the other API Handlers to help you, especially `handler` from `CreateUser` since we have to do nearly all the same validation. Only return here for help if you absolutely need it.

Another suggestion is to get your Function to compile as soon as possible so you can debug the code as you write it one line at a time. We did this when writing `CreateUser's` Function by putting `HTTPure.notFound` as the last line of the `do` block.

You have more than enough experience at this point to write this completely on your own. If you do get stuck, return here and read only as much as you need to get over your stumbling block and then return to coding on your own.

We start, as always, with the Type Signature:

```
handler :: QueryUsersRequest -> Handler
```

Then we'll write the left-hand side of the equal sign:

```
handler :: QueryUsersRequest -> Handler
handler (QueryUsersRequest { authToken }) = ??? ❶
```

❶ Destructure to get the one and only Parameter from the request, i.e. `authToken`.

And then we'll start with a `do` block:

```
handler :: QueryUsersRequest -> Handler
handler (QueryUsersRequest { authToken }) = do
  ???
```

And now, we'll need to `ask` for the `accountsAVar` and the `sessionsAVar`:

```
import Control.Monad.Reader.Class (ask) ❶

handler :: QueryUsersRequest -> Handler
handler (QueryUsersRequest { authToken }) = do
  { accountsAVar, sessionsAVar } <- ask ❷
  HTTPure.notFound ❸
```

❶ Import `ask`.

❷ Ask for the `AVars` from `HandlerEnv`.

❸ This is here to make the compiler happy. This means that we can debug the compiler errors one line at time as we write them. Do NOT forget to remove this.

At this point, I'm going to rely heavily on what we did with `handler` from `CreateUser's`. And the first thing I notice with `CreateUser's` code is that we run in `ExceptT`.

Let's add that:

```
import Control.Monad.Except (runExceptT) ❶
import Control.Monad.Trans.Class (lift)

handler :: QueryUsersRequest -> Handler
handler (QueryUsersRequest { authToken }) = do
  { accountsAVar, sessionsAVar } <- ask
  result <- lift $ runExceptT do ❷
    ???
  HTTPure.notFound
```

❶ Added these imports.

❷ Run using `runExceptT` just like we did in `CreateUser`.

Next, I'm going to add a `pure` at the end to keep the compiler happy.

```
handler :: QueryUsersRequest -> Handler
handler (QueryUsersRequest { authToken }) = do
  { accountsAVar, sessionsAVar } <- ask
  result <- lift $ runExceptT do
    ???
    pure NotAuthorized ❶
  HTTPure.notFound
```

❶ This is just to make the compiler happy. Do NOT forget to remove this.

If you didn't get this far, catch your code up to this point and stop reading to see if you can continue plagiarizing from `CreateUser's` code to finish coding this. Return here if you get stuck.

Next, is to verify the session. I'm just going to steal from `CreateUser`:

```
import Data.Either (note) ❶
import Entity.Session (Session(..))
import Manager.Session (verifySession)
import Utils (liftSuccess)

handler :: QueryUsersRequest -> Handler
handler (QueryUsersRequest { authToken }) = do
  { accountsAVar, sessionsAVar } <- ask
  result <- lift $ runExceptT do
    Session { userName } <- verifySession sessionsAVar authToken
      <#> note NotAuthenticated # liftSuccess ❷
    ???
    pure NotAuthorized
  HTTPure.notFound
```

❶ Added these imports.

❷ This line was copied verbatim from `CreateUser`. Luckily, `QueryUsersFailureReason` also has `NotAuthenticated`.

Now, looking at `CreateUser`, the next step after this one gets the account so we could verify that this request is coming from an admin account. We also want to do that here.

Stealing once again, we get:

```
import Control.Monad.Except (runExceptT, throwError) ❶
import Entity.Account (Account(..)) ❷
import Manager.Account as AM ❷

handler :: QueryUsersRequest -> Handler
handler (QueryUsersRequest { authToken }) = do
  { accountsAVar, sessionsAVar } <- ask
  result <- lift $ runExceptT do
    Session { userName } <- verifySession sessionsAVar authToken
      <#> note NotAuthenticated # liftSuccess
    Account { admin } <- AM.findAccount accountsAVar userName ❸
      <#> note NotAuthorized # liftSuccess
    if not admin then throwError NotAuthorized ❹
    else do
      ???
      pure NotAuthorized ❺
  HTTPure.notFound
```

❶ Added `throwError` to the import list.

❷ Added these imports.

❸ Get the account using the `userName` we got from the `Session`.

❹ If the requestor is NOT an admin then respond with `NotAuthorized`.

❺ This got indented under the `do` block.

Now it's time to flesh out the rest of the `do` block under the `else`. This is the unique code for what our handler is going to do.

The next thing to do is get the accounts using `getAccounts`. If you haven't finished coding this, now may be a good spot to stop reading and see if you can finish from here. Give it a try.

We'll call `getAccounts` to get the accounts for sending back to the client, i.e. once we turn the `Accounts` into `Users`.

First, we'll get the accounts:

```
handler :: QueryUsersRequest -> Handler
handler (QueryUsersRequest { authToken }) = do
  { accountsAVar, sessionsAVar } <- ask
  result <- lift $ runExceptT do
    Session { userName } <- verifySession sessionsAVar authToken
      <#> note NotAuthenticated # liftSuccess
    Account { admin } <- AM.findAccount accountsAVar userName
      <#> note NotAuthorized # liftSuccess
    if not admin then throwError NotAuthorized
    else do
      accounts <- lift $ AM.getAccounts accountsAVar ❶
      ???
      pure NotAuthorized
  HTTPure.notFound
```

❶ Have to `lift` here since `getAccounts` runs in `Aff` and our `do` block is in `ExceptT`.

Now, for the tricky part. We need to map a Function over the `Array` we just got back from `getAccounts`. That Function will destructure an `Account` to get to its contained Record, then delete the extra field that `Account` has when compared to `User`. And finally, it'll construct a `User`.

We should first look to `Account` to see what the name of that extra field is again:

```
newtype Account = Account (Record (UserRow ( passwordHash :: String )))
```

And now we should be able to use `delete` from the module `Record` to remove this.

If you got stuck and you think you now have enough information to keep coding, please stop now and give it a try.

Now, we can convert the `Accounts` to `Users`:

```
import Entity.User (User(..)) ❶
import Record (delete)
import Type.Proxy (Proxy(..))

handler :: QueryUsersRequest -> Handler
handler (QueryUsersRequest { authToken }) = do
  { accountsAVar, sessionsAVar } <- ask
  result <- lift $ runExceptT do
    Session { userName } <- verifySession sessionsAVar authToken
      <#> note NotAuthenticated # liftSuccess
    Account { admin } <- AM.findAccount accountsAVar userName
      <#> note NotAuthorized # liftSuccess
    if not admin then throwError NotAuthorized
    else do
      accounts <- lift $ AM.getAccounts accountsAVar
      let users = accounts <#> \(Account r) -> ❷
            User $ delete (Proxy :: _ "passwordHash") r ❸
      ???
      pure NotAuthorized
  HTTPure.notFound
```

❶ Added these imports.

❷ This is a Pure Computation, which calls for a `let`.

❸ The Lambda is mapped over `accounts`. First, it destructures the `Acccount` to expose the contained Record, `r`, and then deletes the `passwordHash` from the Record. And finally, it constructs a `User` using the modified Record.

Now, if you didn't get this far, stop and replace the `pure NotAuthorized` with a successful `result`, i.e. `QueryUsersResultsSuccess`. Look up it's definition to help you. Then read on.

We're going to remove our stand-in code in the `do` block under the `else` and we're going to replace it with a successful response:

```
handler :: QueryUsersRequest -> Handler
handler (QueryUsersRequest { authToken }) = do
  { accountsAVar, sessionsAVar } <- ask
  result <- lift $ runExceptT do
    Session { userName } <- verifySession sessionsAVar authToken
      <#> note NotAuthenticated # liftSuccess
    Account { admin } <- AM.findAccount accountsAVar userName
      <#> note NotAuthorized # liftSuccess
    if not admin then throwError NotAuthorized
    else do
      accounts <- lift $ AM.getAccounts accountsAVar
      let users = accounts <#> \(Account r) ->
            User $ delete (Proxy :: _ "passwordHash") r
      pure $ QueryUsersResultsSuccess { users } ❶
  HTTPure.notFound
```

❶ Replaced `pure NotAuthorized` with this line.

You may be surprised that there's only 1 `pure` here. I was too at first. But it became clear once I saw the Type for the `do` block that's passed to `runExceptT`.

To find out, add a Type Hole to the `pure $ QueryUsersResultsSuccess { users }` line and see what Type it's returning. Then return back here.

Here's the Type:

```
ExceptT QueryUsersFailureReason Aff QueryUsersResults
```

Keep this Type in mind when you steal the last bit of code from `CreateUser's` implementation where it responds to the client. Stop reading and make that final change replacing `HTTPure.notFound`.

Here's that final change:

```
import Data.Either (note, either) ❶
import Foreign.Generic (encodeJSON) ❷

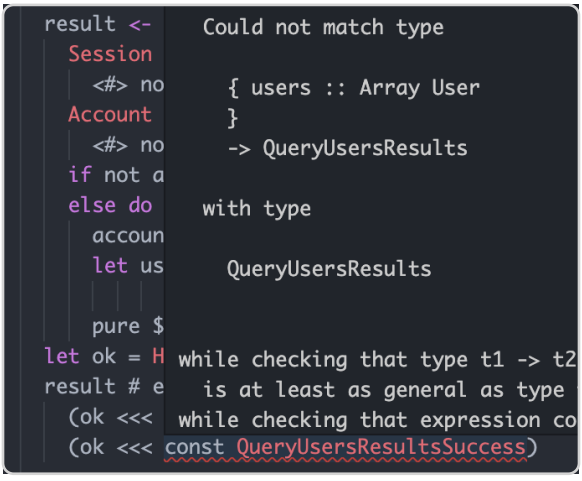
handler :: QueryUsersRequest -> Handler
handler (QueryUsersRequest { authToken }) = do
  { accountsAVar, sessionsAVar } <- ask
  result <- lift $ runExceptT do
    Session { userName } <- verifySession sessionsAVar authToken
      <#> note NotAuthenticated # liftSuccess
    Account { admin } <- AM.findAccount accountsAVar userName
      <#> note NotAuthorized # liftSuccess
    if not admin then throwError NotAuthorized
    else do
      accounts <- lift $ AM.getAccounts accountsAVar
      let users = accounts <#> \(Account r) ->
            User $ delete (Proxy :: _ "passwordHash") r
      pure $ QueryUsersResultsSuccess { users }
  let ok = HTTPure.ok <<< encodeJSON <<< QueryUsersResponse ❸
  result # either
    (ok <<< \reason -> QueryUsersResultsFailure { reason }) ❸
    -- COMPILER ERROR!!
    (ok <<< const QueryUsersResultsSuccess) ❸
```

❶ Added `either` to the import list.

❷ Added this import.

❸ Copied this code from `CreateUser's` implementation and changed `CreateUser` to `QueryUsers` in these 3 places.

Here's our compiler error:



Can you see where I've gone wrong?

It appears to me that `const QueryUsersResultsSuccess` is expecting `{ users :: Array User } -> QueryUsersResults` but got `QueryUsersResults` instead.

Can you see why that is?

It's because of this line:

```
pure $ QueryUsersResultsSuccess { users } ❶
```

❶ In the success case, we return `QueryUsersResults` unlike what we did in `CreateUsers` where we've been stealing code from. In `CreateUsers`, we simply returned `unit` via the `AH.createAccount` call.

Because we're returning something different than we did in `CreateUsers`, this line will need to change:

```
(ok <<< const QueryUsersResultsSuccess)
```

We can simply respond using `ok` since we already have a `QueryUsersResultsSuccess`:

```
handler :: QueryUsersRequest -> Handler
handler (QueryUsersRequest { authToken }) = do
  { accountsAVar, sessionsAVar } <- ask
  result <- lift $ runExceptT do
    Session { userName } <- verifySession sessionsAVar authToken
      <#> note NotAuthenticated # liftSuccess
    Account { admin } <- AM.findAccount accountsAVar userName
      <#> note NotAuthorized # liftSuccess
    if not admin then throwError NotAuthorized
    else do
      accounts <- lift $ AM.getAccounts accountsAVar
      let users = accounts <#> \(Account r) ->
            User $ delete (Proxy :: _ "passwordHash") r
      pure $ QueryUsersResultsSuccess { users }
  let ok = HTTPure.ok <<< encodeJSON <<< QueryUsersResponse
  result # either
    (ok <<< \reason -> QueryUsersResultsFailure { reason })
    ok ❶
```

❶ Removed `const` Function.

Now, that compiles.

That's the danger of copying code. It turns out that we used `const` in `CreateUser` since `CreateUserResultsSuccess` doesn't take any Parameters, hence the use of `const` to throw away

whatever we got back from `AH.createAccount` and just reply with `CreateUserResultsSuccess`.

But here, we need to return `QueryUsersResultsSuccess { users }` and that's why we needed to change the code a bit.

Take the time to update your code with this Function.

Next, we'll update `router` in `Main` to include our new API Handler. Do that now before reading any further.

Here's that change:

```
import Handler.Api.QueryUsers (QueryUsers) ❶

router :: HandlerEnv -> Request -> ResponseM
router env { body, method }
  | method == HTTPure.Post =
      let handlers =
              handle (Proxy :: _ Logon) :| [
              handle (Proxy :: _ Logoff)
            , handle (Proxy :: _ CreateUser)
            , handle (Proxy :: _ QueryUsers) ❷
            ] <#> (_ $ body) in
      case hush $ oneOf handlers of
        Nothing -> HTTPure.badRequest body
        Just reader -> runReaderT reader env
  | otherwise = HTTPure.methodNotAllowed
```

❶ Added this import.

❷ Added this entry.

Now, I'm not liking how the handlers list is polluting our `router` logic.

Factor `handlers` out of `router` renaming it to `apiHandlers` leaving `router` in tact for the moment, and then return here to continue.

First, we'll just copy and paste out `handlers` with its new name:

```
-- COMPILER WARNING!!
apiHandlers =
    handle (Proxy :: _ Logon) :| [
    handle (Proxy :: _ Logoff)
  , handle (Proxy :: _ CreateUser)
  , handle (Proxy :: _ QueryUsers)
  ]
```

The compiler is telling us what our Type Signature should be in this warning:

```
NonEmpty Array (String -> Either (NonEmptyList ForeignError) (ReaderT ... Aff
...))
```

Unfortunately, the … means that it's leaving stuff out to make it more readable. But `ReaderT … Aff …` looks a lot like `Handler`, so let's assume that. Also, `NonEmptyList ForeignError` is just `MultipleErrors`, so we'll make that substitution too:

```
import Foreign (MultipleErrors) ❶
import Handler.Class.ApiHandler (Handler, HandlerEnv, handle) ❷

apiHandlers :: NonEmpty Array (String -> Either MultipleErrors Handler) ❸
apiHandlers =
    handle (Proxy :: _ Logon) :| [
    handle (Proxy :: _ Logoff)
  , handle (Proxy :: _ CreateUser)
  , handle (Proxy :: _ QueryUsers)
  ]
```

❶ Added this import.

❷ Added `Handler` to the import list.

❸ Here's our Type Signature thanks to a little help from the compiler.

This Type Signature makes sense since we have a `NonEmpty Array` of Functions. These Functions take a `String`, which is the `body` of the `Request` and will call the `handle` Method of `APIHandler` returning either decoding errors or a `Handler`.

Now, it's time to attack `router`. Use `apiHandlers` in `router` before reading on.

Here's that change:

```
router :: HandlerEnv -> Request -> ResponseM
router env { body, method }
  | method == HTTPure.Post =
      let handlers = apiHandlers <#> (_ $ body) in ❶
      case hush $ oneOf handlers of
        Nothing -> HTTPure.badRequest body
        Just reader -> runReaderT reader env
  | otherwise = HTTPure.methodNotAllowed
```

❶ Only this line changed.

Now, our `router` won't get more and more unreadable as we add API Handlers.

And now, it's time to test. Rerun the server.

Then create another tab in `Postman`. Make sure to set it to `POST` and make the URL `localhost:3000`.

Then click on the `Body` tab and then the `raw` radio button. Then enter:

```
{ "tag": "QueryUsersRequest",
  "contents": {
    "authToken": "AUTH_TOKEN_FROM_LOGON_RESPONSE" ❶
  }
}
```

❶ Put the `authToken` you get back from `LogonRequest` here.

Now, run a test by first sending `LogonRequest`, then copying the `authToken` to `QueryUsersRequest` and then send that.

What did you get?

Here's what I get:

```
{"contents":{"contents":{"users":[{"contents":{"userName":"admin","temporaryPa
ssword":true,"lastName":"Admin","firstName":"Joe","admin":true},"tag":"User"},
{"contents":{"userName":"joemama","temporaryPassword":true,"lastName":"Mama","
firstName":"Joe","admin":true},"tag":"User"}]},"tag":"QueryUsersResultsSuccess
"},"tag":"QueryUsersResponse"}
```

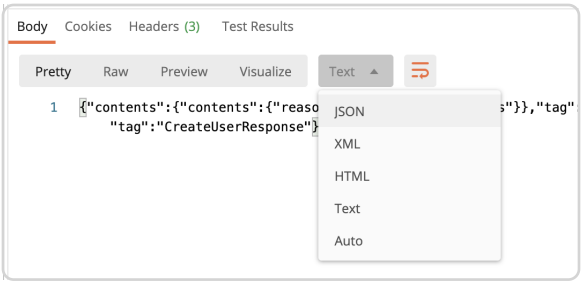
Create a new File in VSCode. Then click the `Select a language` link and then type `JSON` and hit enter.

Now paste the JSON response. It should format it to look like:

```
{
  "contents": {
    "contents": {
      "users": [
        {
          "contents": {
            "userName": "admin",
            "temporaryPassword": true,
            "lastName": "Admin",
            "firstName": "Joe",
            "admin": true
          },
          "tag": "User"
        },
        {
          "contents": {
            "userName": "joemama",
            "temporaryPassword": true,
            "lastName": "Mama",
            "firstName": "Joe",
            "admin": true
          },
          "tag": "User"
        }
      ]
    },
    "tag": "QueryUsersResultsSuccess"
  },
  "tag": "QueryUsersResponse"
}
```


And this looks exactly what we want.

We can also format directly in `Postman` by clicking on the `Pretty` tab in the response section and then selecting the Text pulldown and selecting JSON:



28.11. Servers in PureScript

We've seen that we can write a server in PureScript. If you would have no problems with putting a Node server in production then it would be so much better to write that server in PureScript instead of Javascript.

However, I would encourage you to write it Haskell to be able to take advantage of true multi-threading. The number of libraries for backend work in PureScript is limited compared with Haskell, although there are many wrappers around Javascript libraries for interfacing with Postgres, Redis, etc.

It doesn't mean that you cannot wrap some missing Node or NPM library using what we've learned in the FFI Chapters. But that can be a lot of tedious work.

Your choice will involve the following:

- How comfortable you are with Haskell compared with PureScript
- How many other developers know Haskell compared with PureScript
- What are the requirements of the server
- What are the business needs, e.g. time to deliver, technical debt, etc.

Whatever you choose in production and for whatever reasons, you now have a good starting point when writing a server.

We're not done with this server. We're going to use it as we write a front-end web application using the Halogen framework.

Chapter 29. Building Front-ends using Halogen

Halogen is a front-end framework that models Web GUIs as Components. It's written in PureScript and it's performance and size are comparable with React or Angular.

Each component maintains its own State and can be composed with other Components to produce complex user interfaces.

Like many libraries today, Halogen uses a Virtual DOM, which is also written in PureScript.

The Virtual DOM allows the developer to simply re-render the full page whenever the State changes and the Virtual DOM will effeciently update the real DOM by only updating the differences between the real DOM and the Virtual one. This keeps the rendering logic very simple for Halogen Components.

29.1. Halogen Component Overview

Components are at the heart of the Halogen Architecture. Components can be composed, i.e. they can contain zero or more other Components known as Children or Child Components.

There are different names given Components:

- ***Parent*** is a Component that has Children. A Component can be both a Parent and a Child.
- ***Child*** is a Component that has a Parent. A Component can be both a Child and a Parent.
- ***Top-level*** is a Component that is a Component that's created by the Application. You can have many Top-level Components, where the Application is the de facto Parent.

The code for a Component is usually self-contained in a single module. There are 4 typical Functions in a Component:

- ***Component Constructor*** that creates the Component
- ***Action Handler*** that handles different Actions (more on Actions later)
- ***Query Handler*** that responds to Parent Queries (more on Queries later)
- ***Renderer*** that generates the HTML for the Component

29.1.1. State

Components have ***State*** that is accessed via `MonadState`, e.g. `get`, `put`, `modify`, etc. Changes to State will trigger the rendering of the Component since State dictates the Component's look and behavior. The State is initialized when the Component is Initialized and deleted when the Component is Finalized.

Any State that must be persisted beyond the Lifecycle of the Component must be managed manually by the developer.

29.1.2. Actions

An ***Action*** is a single operation that a Component can do. These are atomic Operations that operate in the `HalogenM` Monad. The Action Handler is one of 2 places where a Component can do certain `HalogenM` operations:

- Modify State
- Perform Effects
- Send data to Parent
- Query Children (more on Query later)
- Subscribe/Unsubscribe to Emitters (more on Emitters later)
- Interact directly with the DOM
- Fork/Kill `HalogenM` processes

29.1.3. Queries

A ***Query*** is a request made from a Parent Component to a Child for the transfer of information. That transfer of information can either be transferred from the Child to the Parent or vice versa.

The Query Handler, like the Action Handler, operates in the `HalogenM` Monad, which means that it can perform all of the same `HalogenM` operations that are listed in the Actions section.

29.1.4. Inputs

Each time a Parent Component renders, it sends an ***Input*** Value to a Child Component. This will cause the associated Action to fire in the Child Component if the Child supports Inputs. If that Action modifies the Child's State then the Child Component will re-render.

29.1.5. Outputs

An ***Output*** is data that a Child Component sends to its Parent. Unlike Queries, this data is unsolicited.

29.1.6. Emitters

Emitters are subscribed to by Components and turned into Actions for the Component to handle at a later time.

Examples of Emitters:

- Halogen rendered DOM elements, e.g. `button`
- Non-Halogen rendered DOM elements, e.g. `document`
- User Emitters, i.e. Actions emitted from any other part of the Application, e.g., when data is Read from a Bus or when a timer expires or any other Asynchronous Event

Except for Halogen rendered DOM element events, all other Events must be turned into Actions manually by the developer.

29.1.7. Lifecycle

The ***Lifecycle*** of a Component is as follows:

- Component is Constructed at which point the State is initialized
- Initializer for the Component is called
- Component operates handling Actions and Queries
- Finalizer for the Component is called
- Component State is deleted

29.1.8. Slots

A ***Slot*** is a place where Child Component information is kept. The following information regarding a Child Component is kept in a Slot:

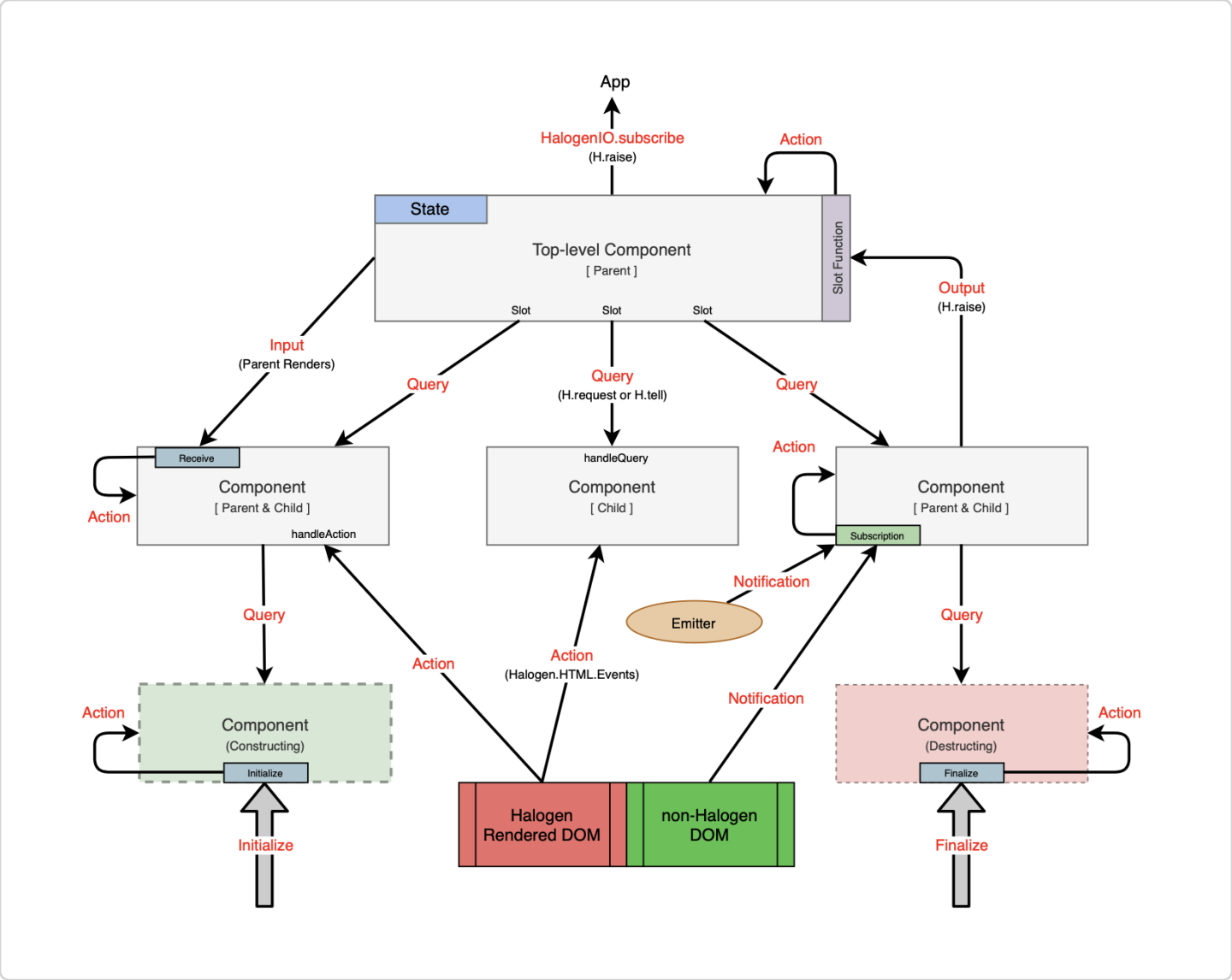
- Type of the Queries a Parent can send the Child
- Type of the Output a Parent can receive from the Child
- Type of the Input a Parent can give the Child when rendering
- Type of a Unique Identifier with an `Ord` Instance to distinguish multiple instances of the same Child Component
- Unique Identifier for the Child
- A mapping Function to map Output from the Child to an Action

The Slot information is kept in 2 places. The Type information is kept in a Row Type and the rest is specified when the Child is rendered in the Parent.

29.1.9. Halogen Model

The following is a diagram that models the Halogen Architecture capturing most, if not all, of the aforementioned items, i.e. Components (Parent, Child, Top-level), Component State, Actions, Queries, Inputs, Outputs, Emitters, Lifecycle (Initialize, Finalize) and Child Slots.

There's a lot of detail in this diagram, so please do not be intimidated by its initial complexity. We're going to go over every aspect of this.



Take a minute to study this diagram. It's going to be your cheatsheet. If you ever forget how things work, you can always come back to this.

29.2. Halogen Example Application

Before we write a program to interface to our server, we're going to build an Application to learn every aspect of this diagram. The surprising thing is how small our program will be when it's completed.

29.2.1. Creating New Project

Go to parent directory that contains the project you've been working on.

If your code is kept in a folder called `/dev`, and you followed the directions exactly in Chapter 4 then you'd run:

```
cd /dev/purescript
```

If this is not where you put your project code, change to the appropriate directory now.

Next, create the `halo` project directory:

```
mkdir -p halo # or some other name you like better
cd halo
```

Before we go further ***make sure that you're in the `halo` directory***.

To initialize your project so that we can install `npm` packages and use `npx` to run the compiler and build tool from inside your project, run the following command:

```
npm init -y
```

This will create a file call `package.json`. This will keep track of any `npm` packages we install.

To locally install the PureScript compiler, `purescript` and the build tool, `spago`, run:

```
npm install --save-dev spago@0.20.0 purescript@0.14.1
```

Notice the `--save-dev` option. This will tell `npm` that these packages are only for development.

To verify that we have installed the compiler locally, we'll use `npx` (NOTICE the `x`):

```
npx purs --version # should print out 0.14.1
```

`npx` looks in our local `node_modules` directory for `purs`, the name of the PureScript compiler, and then runs it passing all other command line Parameters to it. The parameter `--version` tells `purs` to print out it's version.

To verify that we have installed the build tool locally, run:

```
npx spago version # should print out 0.20.0
```

Now, prepare the project for version control using `git`, run:

```
git init # should print `Initialized empty Git repository in <dir>`
```

The final step is to initialize this project for building a PureScript program using `spago`:

```
npx spago init
```

Since we're creating a Halogen Application, we'll need to install the necessary packages. We're going to use `spago` to do this instead of manually editing our `spago.dhall` file like we've been doing:

```
npx spago install halogen
```

This will install `purescript-halogen`.

Let's build our application next:

```
npx spago build
```

You will get a bunch of warnings like:

```
Warning 1 of 4:

  in module Data.Ord.Generic
  at .spago/prelude/v5.0.0/src/Data/Ord/Generic.purs:22:3 - 22:41 (line 22,
column 3 - line 22, column 41)

    Name b1 was introduced but not used.

  in value declaration genericCompare'

  See
https://github.com/purescript/documentation/blob/master/errors/UnusedName.md
for more information,
  or to contribute content related to this warning.
```

This is because a bunch of libraries haven't caught up to version 14 of the compiler. Version 14 of the PureScript compiler gives you warnings now for unused variables. This change was made and not all libraries have quite caught up yet.

There are also a bunch of deprecated features from version 13 that will give you warnings. They will be permanently removed in version 15, but for now we get warnings. These can be ignored when they come from libraries. The library owners will take care of fixing them in due time.

We only have to worry about these warnings in our own code. But for now all the warnings are for libraries, since we haven't written any code just yet.

Now, if you're using VSCode, you should be able type the following command to open up this directory in the editor:

```
code .
```

If this doesn't work or you're using a different editor, then simply open this directory manually from your editor.

Next, we're going to install `purescript-halogen-css` by hand. We have to do this because it's not in the spago package-set at the time of this writing.

First, locate the following line of code in your `packages.dhall`:

```
in  upstream ❶
```

❶ Should be the last line in the file.

Now, change that line to the following lines:

```
let additions =
  { halogen-css =
    { dependencies =
       [ "css"
       , "halogen"
       ]
    , repo =
       "https://github.com/purescript-halogen/purescript-halogen-css"
    , version =
        "v8.0.0"
    }
  }
in  upstream // additions
```

This will add the `halogen-css` library to our project. This library is version `8.0.0` and works with version 14 of the PureScript compiler. For some reason, it hasn't been added to spago's package set, so we must do so manually.

Next, we add this module to our `spago.dhall` file:

```
{-
Welcome to a Spago project!
You can edit this file as you like.
-}
{ name = "halo" ❶
, dependencies =
  [ "console"
  , "effect"
  , "halogen"
  , "halogen-css" ❷
  , "prelude"
  , "psci-support"
  ]
, packages = ./packages.dhall
, sources = [ "src/**/*.purs", "test/**/*.purs" ]
}
```

❶ Renamed `my-project` to `halo`.

❷ Added `halogen-css` to the list of dependencies (after reformatting).

And finally, we can rebuild the app code:

```
npx spago build
```

Once again, ignore all the library warnings. If you run the build command, you'll see that there are no warnings since there is no need to rebuild the libraries.

You will get the following warning that you should pay attention to:

```
[info] Build succeeded.
[warn] None of your project files import modules from some projects that are
in the direct dependencies of your project.
These dependencies are unused. To fix this warning, remove the following
packages from the list of dependencies in your config:
- halogen-css ❶
```

❶ This warning normally is to be heeded, but in our case we just added this and we're planning on using this so we'll leave it.

29.2.2. Writing a Counter Component

We're going to create a very simple component that is a `Counter`. It'll have controls for incrementing and

decrementing its count.

Start by making a Folder under `src` called `Component`. Then create a file under Component called `Counter.purs`.

Let's start with:

```
module Component.Counter where ❶

import Prelude

import Halogen as H
import Halogen.HTML as HH
import Halogen.HTML.CSS as HC
```

❶ Make sure the module name mirrors the File path from `src`.

Add this code to your module.

We're importing some standard modules that nearly all Halogen Components need. `Halogen` is where general Halogen Functions are, e.g. `H.mkComponent` to make a Component.

`Halogen.HTML` is where all of the HTML Functions are to create Virtual DOM Elements, e.g. `HH.div`, `HH.text`, `HH.button`, etc.

`Halogen.HTML.CSS` is where the CSS style Function is, i.e. `HC.style`, which let's you define styles for your Virtual DOM Elements.

Let's start by adding the `component` Function. This will create an `HH.HTMLComponent`, which is a Halogen Component:

```
import Effect.Aff.Class (class MonadAff) ❶

component
  :: ∀ m
  .  MonadAff m ❷
  => H.Component Query Input Output m ❸
component = H.mkComponent ❹
  ???
```

❶ Add this import.

❷ We're operating in some Monad `m` that can operate in `Aff`. In this application, it'll be `Aff`, but it could be your own Monad Stack with `Aff` at the bottom, i.e. `ReaderT Env Aff a`.

❸ This returns a Halogen Component that renders to `HH.HTML`. `Query`, `Input` and `Output` are Component-

specific Types. `m` is the effect Monad that's used during the Evaluation. Your component is ***Evaluated*** each time an event occurs that warrants running your component code.

❹ `H.mkComponent` makes a Component. It takes some Parameters, which are currently missing.

Let's look at `H.mkComponent` in detail:

```
mkComponent
  :: ∀ state query action slots input output m
  .  ComponentSpec state query action slots input output m
  -> Component query input output m
```

This may seem intimidating at first glance, but these Type Parameters are used over and over again throughout Halogen, so you'll get quite familiar with them soon enough.

Here are their definitions:

- `state` - This is the Type that is your State. Components maintain their own internal State.
- `action` - This is the Type for Actions that your Component can take. In our `Counter` Component, we will have the Actions, `Increment` and `Decrement`, which will be triggered by the user clicking on the appropriate buttons.
- `slots` - This is the Row Type that defines your Children. When your Component has no Child Components, you can simply just make this an Empty Row, i.e. `()`.
- `input` - This is the Type for the Input you'll get from your Parent Component or from the Application if you're a Top-level Component.
- `output` - This is the Type for the messages you send back to your Parent or the Application if you're a Top-level Component.
- `query` - This is what's called the ***Query Algebra***, i.e. operations you can perform on your Component. If you've programmed in Object Oriented Languages you can think of this as the Public Interface to your Component.

Now let's look at `ComponentSpec` in detail:

```
import Halogen.HTML.Core as HC

type ComponentSpec state query action slots input output m =
  { initialState :: input -> state
  , render
      :: state
      -> HC.HTML (ComponentSlot slots m action) action
  , eval
      :: HalogenQ query action input
      ~> HalogenM state action slots output m
  }
```

- `initialState` - This Function gets a single Parameter of the `input` Type and will initialize the `state`.
- `render` - This Function creates the Virtual DOM, using Functions like `HH.div`, `HH.input`, `HH.text`, etc. It takes a single Parameter of the `state` Type and uses this data to render the Virtual DOM Elements.
- `eval` - This is a Function that Evaluates your Component. We don't ever write this Function, but instead use a helper called `mkEval` where we pass an `EvalSpec`.

Let's add the `ComponentSpec` to our code:

```
component
  :: ∀ m
  .  MonadAff m
  => H.Component Query Input Output m
component = H.mkComponent
  { initialState: ??? ❶
  , render ❷
  , eval: H.mkEval ??? ❸
  }
```

❶ We haven't defined `Input` or `State` yet, so we cannot code this just yet.

❷ `render` it typically a fairly large Function and is usually in the same module as the component code.

❸ `H.mkEval` wants an `EvalSpec`, but we don't know anything about `EvalSpec` yet.

Since we're writing a simple Counter Component, let's define the following:

```
type Input = Int
type Output = Int
type State = { count :: Int }
```

The `Input` and `Output` are the same Type as the `count` Value in our State. This means that we can receive `Ints` from our Parent and we can send `Ints` to our Parent.

Add these Types to your code.

Now, we can write `initialState` since we know both the `input` and `state` Types:

```
component
  :: ∀ m
  .  MonadAff m
  => H.Component Query Input Output m
component = H.mkComponent
  { initialState: \i -> { count: i } ❶
  , render
  , eval: H.mkEval ???
  }
```

❶ We simply take the `Input` and initialize our counter Value with it.

To continue coding, we'll need to understand `EvalSpec`:

```
type EvalSpec state query action slots input output m =
  { handleAction :: action -> HalogenM state action slots output m Unit
  , handleQuery
      :: ∀ a. query a
      -> HalogenM state action slots output m (Maybe a)
  , receive :: input -> Maybe action
  , initialize :: Maybe action
  , finalize :: Maybe action
  }
```

- `handleAction` - This Function is where your Actions are handled. Most Events regarding the Component get turned into Actions and this is where you can actually perform Effects.
- `handleQuery` - This Function handles requests from the Parent or the Application if you're a Top-level Component. Like `handleAction`, it is the only other Function in your Component where Effects can be performed.
- `receive` - This Function is called whenever your Parent renders. It takes a single Parameter of Type `input` and then optionally turns it into an Action. That Action will eventually be processed by `handleAction`.
- `initialize` - This Function is called when your Component is first created. It can return an optional Action to be subsequently processed by `handleAction`.
- `finalize` - This Function is called when your Component is being destroyed. This can happen if your

Parent's render Functon no longer renders you or, if you're a Top-level Component, when the Application explicitly calls `dispose` on you.

There is a default `EvalSpec`:

```
defaultEval
  :: ∀ state query action slots input output m
  .  EvalSpec state query action slots input output m
defaultEval =
  { handleAction: const (pure unit)
  , handleQuery: const (pure Nothing)
  , receive: const Nothing
  , initialize: Nothing
  , finalize: Nothing
  }
```

All of the defaults do nothing. So, this makes overriding easy. Let's use the default and just override `initialize` and `finalize`:

```
import Data.Maybe (Maybe(..)) ❶

data Action ❷
  = Initialize
  | Finalize

component
  :: ∀ m
  .  MonadAff m
  => H.Component Query Input Output m
component = H.mkComponent
  { initialState: \i -> { count: i }
  , render
  , eval: H.mkEval H.defaultEval {
      initialize = Just Initialize ❸
    , finalize = Just Finalize ❹
    }
  }
  where
  ???
```

❶ Import `Maybe` and its Data Constructors.

❷ Define 2 Actions to handle the initialization and finalization of our Component.

❸ We're overriding `initialize` from `defaultEval` to be the `Initialize` Action. Since this is optional, it's a `Maybe`.

❹ We're overriding `finalize` from `defaultEval` to be the `Finalize` Action.

Notice the `where` keyword at the bottom of the above code block. I like to put all of my Component Functions, e.g. `render`, `handleAction`, etc. under a `where`.

You don't have to do this, but if you pass any Parameters to the `component` Function when you construct your Component, those Parameters will be in scope in every Function under the `where`.

I've found that I rely on this more than not, which is why I always code my components this way.

Now, let's write a stub for the `render` Function:

```
  -- ABOVE CODE REDACTED
  where
  render :: State -> H.ComponentHTML Action Slots m
  render _ = HH.text "Stubbed out" ❶
```

❶ A single Text Node.

All we need now is to add Type Aliases for `Query` and `Slots`:

```
type Query = Void ❶

type Slots :: ∀ k. Row k ❷
type Slots = () ❸
```

❶ For now, there are no Queries. So we use `Void`.

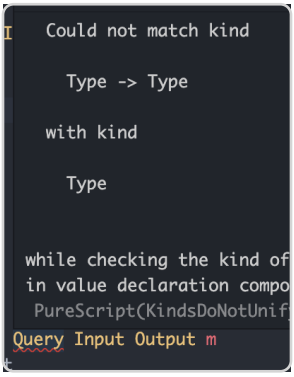
❷ Since we're specifying an empty Row, the compiler will complain that `The inferred kind for the type declaration Slots contains polymorphic kinds`. And it will suggest you add a Type Signature since this definition is a top-level one. Remember, polymorphic kinds means that we can specify any Kind. Some examples of Kinds are `Type`, `Constaint`, `Symbol`, and `Row`. Since the `Row` is empty, it could be anything.

❸ For now, there are no Child Components.

We have a compiler error on the Type Signature of `component`:

```
component
  :: ∀ m
  .  MonadAff m
  -- COMPILER ERROR!!
  => H.Component Query Input Output m
```

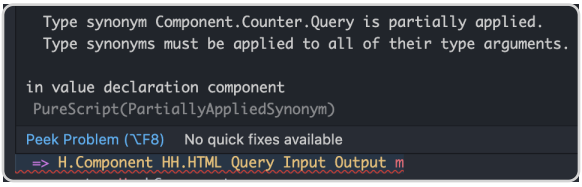
Here's the error:



Looks like `Query` takes a Parameter, so let's give it one:

```
type Query a = Void -- COMPILER ERROR!!
```

Now, we have a different error:



Looks like we cannot have a Phantom on Type Aliases. We can change it to a real Type or we can use a Type called `Const`:

```
newtype Const a b = Const a
```

This lets us do:

```
import Data.Const (Const)

type Query :: ∀ k. k -> Type ❶
type Query = Const Void
```

❶ `Const` takes 2 parameter. We are only Partially Applying `Void` to `Const`. Because of this, the compiler

doesn't know what Kind the second Parameter is, which is why we need this Type Signature. It basically says, we take any Kind, `k`, and return a `Type`.

We could've made `Query` a real Type, which wouldn't require the use or import of `Const`:

```
newtype Query :: ∀ k. k -> Type ❶
newtype Query a = Query Void
```

❶ The Type Signature is unchanged and still required since `a` is unknown and a Phantom Parameter.

The choice is yours. Pick your poison. I'm going to choose the use of `Const`.

Now, you should have a single compiler WARNING since we're not using `Halogen.HTML.CSS` yet:

```
module Component.Counter where

import Prelude

import Data.Const (Const)
import Data.Maybe (Maybe(..))
import Effect.Aff.Class (class MonadAff)
import Halogen as H
import Halogen.HTML as HH
import Halogen.HTML.CSS as HC -- COMPILER WARNING!!

type Input = Int
type Output = Int
type State = { count :: Int }

type Slots :: ∀ k. Row k
type Slots = ()

type Query :: ∀ k. k -> Type
type Query = Const Void

data Action
  = Initialize
  | Finalize

component
  :: ∀ m
  .  MonadAff m
  => H.Component Query Input Output m
component = H.mkComponent
  { initialState: \i -> { count: i }
  , render
  , eval: H.mkEval H.defaultEval {
      initialize = Just Initialize
    , finalize = Just Finalize
    }
  }
  where
  render :: State -> H.ComponentHTML Action Slots m
  render _ = HH.text "Stubbed out"
```

Now that this compiles, we should add a `handleAction` Function so we can log when the Component is initialized and finalized:

```
import Effect.Class.Console (log) ❶

  -- ABOVE CODE REDACTED
  where
  handleAction
    :: Action
    -> H.HalogenM State Action Slots Output m Unit
  handleAction = case _ of ❷
    Initialize -> log "Initialize"
    Finalize -> log "Finalize"

  render :: State -> H.ComponentHTML Action Slots m
  render _ = HH.text "Stubbed out"
```

❶ Import `log` that works in both `Aff` and `Effect`.

❷ Using wildcard `case` expression here. This is idiomatic.

Make sure that your code is up to date and compiles before advancing on.

Except for the compiler warning that we haven't used `Halogen.HTML.CSS`, did you get a warning on `component`. Hover your mouse over the warning and see what you get.

Here's what I get:



That's because I forgot to add `handleAction` to the `EvalSpec`. Let's rectify that:

```
component
  :: ∀ m
  .  MonadAff m
  => H.Component Query Input Output m
component = H.mkComponent
  { initialState: \i -> { count: i }
  , render
  , eval: H.mkEval H.defaultEval {
      initialize = Just Initialize
    , finalize = Just Finalize
    , handleAction = handleAction ❶
    }
  }
  where
  -- BELOW CODE REDACTED
```

❶ Added `handleAction` function that we just wrote so Halogen knows about it.

Now, we should code the `main` to run our UI. Change you `Main.purs` to look like:

```
module Main where

import Prelude

import Component.Counter as Counter ❶
import Effect (Effect)
import Halogen.Aff as HA ❷
import Halogen.VDom.Driver (runUI) ❸

main :: Effect Unit
main = do
  HA.runHalogenAff do ❹
    body <- HA.awaitBody ❺
    runUI Counter.component 0 body ❻
```

❶ Import our `Counter` Component.

❷ Import `Halogen.Aff`, for `awaitBody`.

❸ Import `Halogen.VDom.Driver` to run our UI via `runUI`.

❹ `runHalogenAff` is nothing more than a simple helper on `runAff` that will throw exceptions on any uncaught errors.

❺ This waits for the body of the DOM to be ready before we try to add anything to it. This is basically the Halogen equivalent of waiting for the `DOMContentLoaded` event in Javascript.

❻ This runs our Top-level Component in an HTML Element. We happen to be using `body` since our whole program is written in Halogen. It also passes the `Input` Value to `initialState`. Here we're starting our `Counter` at `0`.

Make sure your `main` matches and compiles before reading on.

Next, we'll need to create a simple HTML Document. Create a File called `index.html` OUTSIDE of the `src` Folder, i.e. as a sibling to the `src` Folder, with the following:

```
<!DOCTYPE html>
<html lang="en" dir="ltr">

<head>
  <meta charset="utf-8">
  <title>HALO - Learning</title>
</head>

<body>
  <script src="/index.js" charset="utf-8"></script> ❶
</body>

</html>
```

❶ When we build our Application, we'll tell `spago` to Bundle our Application. This will create a file called `index.js` in the build directory.

And finally, we're going to create a `Makefile` also outside of the `src` directory to use `make` to build our Application. We're no longer running in Node, so we'll need some different commands:

```
build:
    npx spago build ❶

run:
    npx spago bundle-app ❷
    python -m SimpleHTTPServer ❸

.PHONY: build run
```

❶ Normal build command. This is useful for checking to make sure you code compiles.

❷ We're now using `bundle-app` for Browser development.

❸ Using Python's `SimpleHTTPServer` to serve up our code. It defaults to port `8000`. If you'd like to run on a different port, simply add the port number as the final argument to the command, e.g. `python -m SimpleHTTPServer 8080`.

If you're on Windows and cannot or do not want to install GNU `make`, you can use `nmake` and the following `Makefile`:

```
build: .phony
    npx spago build

run: .phony
    npx spago bundle-app
    python -m SimpleHTTPServer

.phony:
```

Note that I have no way to test this make file so your mileage may vary. You can also create a batch file or simply type these command by hand.

If you're on a Mac, you should already have `Python` installed. On Windows, you can install it if it's not already. But, basically, any simple HTTP server you have at your disposal will do here. We just need something to server our files.

You could actually just write one in short order using PureScript and Node.

Next, we're going to run our program with:

```
make run
```

And unfortunately, we get the following error:

```
[error] Some of your project files import modules from packages that are not
in the direct dependencies of your project.
To fix this error add the following packages to the list of dependencies in
your config:
- aff
- const
- maybe
```

We need to add these DIRECT dependencies to our `spago.dhall`:

```
{-
Welcome to a Spago project!
You can edit this file as you like.
-}
{ name = "halo"
, dependencies =
  [ "aff" ❶
  , "console"
  , "const" ❶
  , "effect"
  , "halogen"
  , "halogen-css"
  , "maybe" ❶
  , "prelude"
  , "psci-support"
  ]
, packages = ./packages.dhall
, sources = [ "src/**/*.purs", "test/**/*.purs" ]
}
```

❶ Added these dependencies keeping the list sorted.

Let's try again:

```
make run
```

Now it should be running:

```
[info] Build succeeded.
[info] Bundle succeeded and output file to index.js
python -m SimpleHTTPServer
Serving HTTP on 0.0.0.0 port 8000 ...
```

If you're using Python's `SimpleHTTPServer`, you can launch your browser and go to `localhost:8000`. I'd suggest using Chrome since it has great developer tools.

When I do this, the only thing on the page is the text, `Stubbed out`. That's because all we did was to render a placeholder Text Node. Let's change that:

```
render :: State -> H.ComponentHTML Action Slots m
render { count } = HH.text $ show count
```

Now, kill the server with `Ctrl-C` and run it again with:

```
make run
```

When I do this and refresh my browser page, I get `0` on the page. If you don't then make sure you're doing a Hard Reload on Chrome with `Shift-Cmd-R` on Mac or `Ctrl-F5` on Windows. Don't forget this won't work if the Developer Tools are not currently visible.

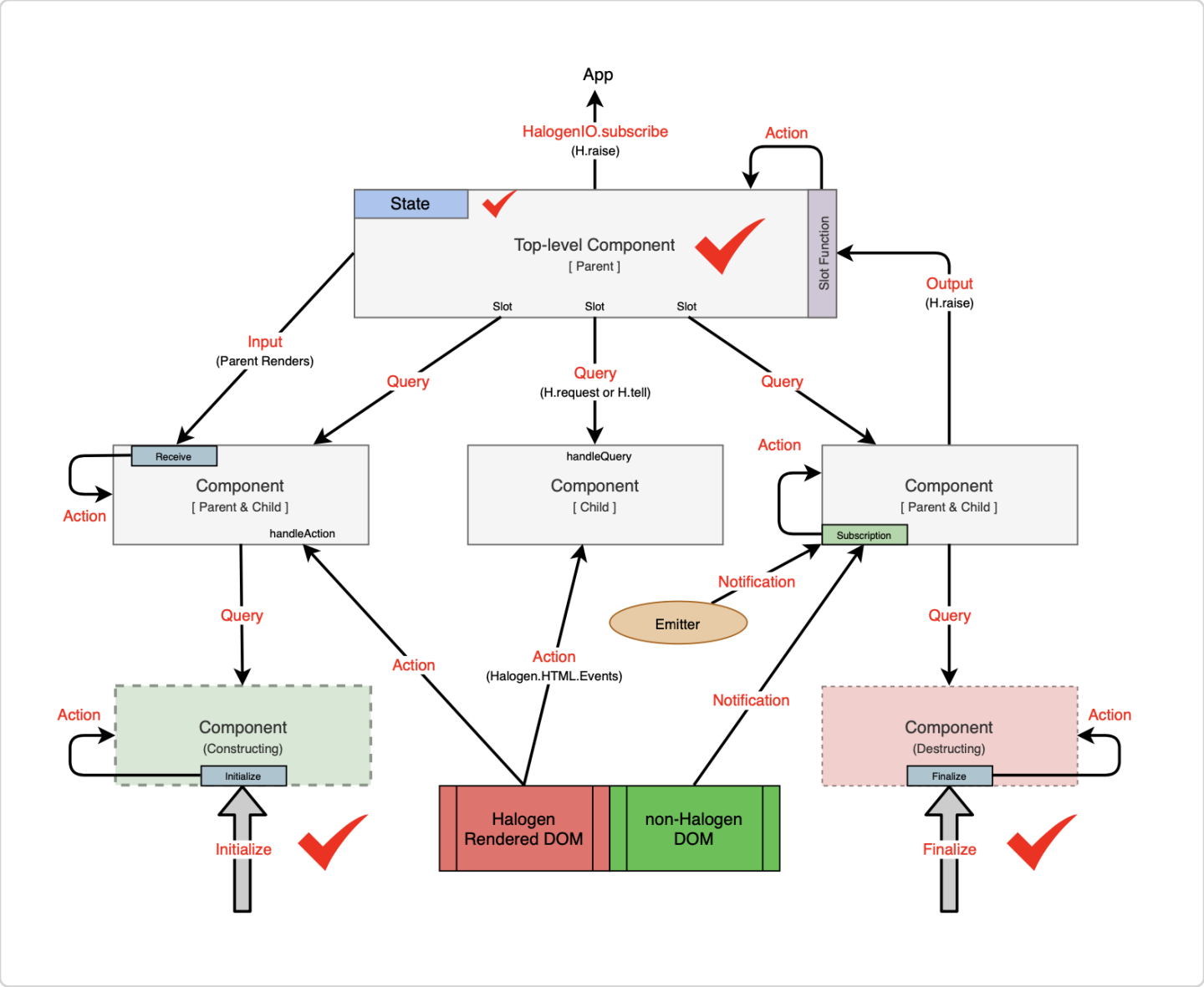
Now, that we see the `0`, we know our component is working.

Now, open up your Console in the Developer Tools. In Chrome, hit `F12` in Mac or Windows or `Cmd-Option-J` on the Mac.

When I do this, I can see that `Initialized` is printed on the Console. This happens when the `Initialize` Action is performed.

At the moment, there's no easy way for us to test `Finalize`, so we'll leave that for later.

Let's revisit our diagram and mark off what we've accomplished:



In this section, we've accomplished the following:

- Created a Top-level Component
- Created a Component with State
- Handled Initialize
- Handled Finalize

29.2.3. Rendering the Counter Component

Next, we're going to flesh out the visual aspect of our Component. We'll start simply with no CSS and then style it at the end.

Let's start by modifying our `render` Function to be the following:

```
import Halogen.HTML.Events as HE ❶

render :: State -> H.ComponentHTML Action Slots m
render { count } =
  HH.div [] ❷
  [
    HH.div []
    [
      HH.button [ HE.onClick $ const Decrement ] [ HH.text "-" ] ❸
    , HH.text $ show count ❹
    , HH.button [ HE.onClick $ const Increment ] [ HH.text "+" ] ❸
    ]
  ]
```

❶ Import Halogen Events so we can handle the Events from user interactions with our UI.

❷ The first Parameter to `HH.div` is where we're going to place our CSS code. But for now, it's an empty `Array`.

❸ These buttons have `onClick` handlers that create `Actions` to be handled in `handleAction`.

❹ We show the `count` in between the buttons.

One thing to note is that `HH.div_` is the same as `HH.div` minus the first Parameter, i.e. its Properties. If you have a `div` that has no styling or other Properties, then using `HH.div_` is more terse and preferable.

We, however, will be adding styling soon, so we're using `HH.div`. Many of the common HTML Element Functions have an underscore equivalent for an Element with no Properties, e.g. `HH.section_`, `HH.table_`, `HH.p_`, etc.

Looking at `HE.onClick`, we can see the use of `const`. That's because this normally takes a Function that expects an `Event` as its first Parameter. In this case, the Event is a `MouseEvent`:

```
onClick
  :: ∀ r i
  .  (MouseEvent -> Maybe i)
  -> IProp (onClick :: MouseEvent | r) i
```

Since we don't care about the `MouseEvent`, we're using `const` to dispose of it.

Now, that I look at our repetitive code for `HE.onClick`, I think it's worth factoring out:

```
render :: State -> H.ComponentHTML Action Slots m
render { count } = let onClick = HE.onClick <<< const in ❶
  HH.div []
  [
    HH.div []
    [
      HH.button [ onClick Decrement ] [ HH.text "-" ] ❷
    , HH.text $ show count
    , HH.button [ onClick Increment ] [ HH.text "+" ] ❷
    ]
  ]
```

❶ Factored out repetitive code into `onClick`. Note the change from `$` to `<<<` making it Point-free.

❷ These lines read much nicer now.

Take the time to update your `render` Function.

Next, we'll have to add the 2 new Actions to our `Action` Type:

```
data Action
  = Initialize
  | Finalize
  | Increment ❶
  | Decrement ❶
```

❶ Added these lines.

And finally, we can handle the Actions:

```
handleAction
  :: Action
  -> H.HalogenM State Action Slots Output m Unit
handleAction = case _ of
  Initialize -> log "Initialize"
  Finalize -> log "Finalize"
  Increment -> H.modify_ \s -> s { count = s.count + 1 } ❶
  Decrement -> H.modify_ \s -> s { count = s.count - 1 } ❷
```

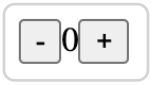
❶ Add 1 to the State and modify it with `H.modify_`.

❷ Subtract 1 from the State and modify it with `H.modify_`.

Make sure your code is up to date with mine and rerun the server and refresh your browser to see what

you get.

When I do this, I get:



This is pretty ugly but if you click on the buttons, the Counter value should change appropriately.

Now, let's fix the ugly UI by styling our Counter:

```
import CSS.Display (display, flex) ❶
import CSS.Flexbox (flexDirection, row, justifyContent)
import CSS.Geometry (width)
import CSS.Size (rem)

render :: State -> H.ComponentHTML Action Slots m
render { count } = let onClick = HE.onClick <<< const <<< Just in
  HH.div []
  [
    HH.div
    [ HC.style do ❷
        display flex
        flexDirection row
        justifyContent spaceEvenly -- COMPILER ERROR!!
        width (rem 6.0)
    ]
    [
      HH.button [ onClick Decrement ] [ HH.text "-" ]
    , HH.text $ show count
    , HH.button [ onClick Increment ] [ HH.text "+" ]
    ]
  ]
```

❶ Added these imports.

❷ Using `HC.style` which expects a single Parameter of Type `StyleM`, a Monad.

There's a lot here, so let's break it down one part at a time. First, the CSS imports are all standard PureScript CSS imports. There's nothing special about these for Halogen.

Another thing to note about the CSS modules is their categories. Whoever put this library together tried to categorize CSS in a logical fashion, a tall order at best.

I think they did a pretty good job, but in the beginning this partitioning can prove difficult when searching for the correct module to import from. Pursuit is your friend here.

The next thing to note in the above code is that `HC.style` is written using a `do` block. The `StyleM` Monad is nothing more than a lightweight wrapper on the `Writer` Monad, which simply appends these CSS Rules together into an `Array`.

This is just syntactical sugar to avoid having to use the `Semigroup` operator, `<>`. You can, however, code with `<>` if you like:

```
-- ABOVE CODE REDACTED
  HH.div [
    HC.style $
          display flex
      <>  flexDirection row
      <>  justifyContent spaceEvenly
      <>  width (rem 6.0)
  ]
-- BELOW CODE REDACTED
```

Knowing this fact can be very useful if and when you are programmatically adding styles to an Element and cannot use the `do` block to do it.

Idiomatic Halogen uses the `do` block for most situations.

And the final thing to note is the compiler error. It turns out that `spaceAround` and `spaceBetween` are supported in the CSS libary, but `spaceEvenly` is not for some strange reason.

I could have changed my code to use one of those, but what if that's exactly what my UI needs. How do you go about adding this feature?

To figure that out, I simply looked at the implementation of `spaceAround`. Follow with me as we unearth the code for `spaceAround`.

First, go to `Pursuit` and search for `spaceAround`. Then click on the one from `CSS.Flexbox`.

You should find that it's a Method to a Typeclass called `SpaceAround`:



This Typeclass has Instances for `Value`, `AlignContentValue` and `JustifyContentValue`. That last one is how we want to use it.

Seems like we'll have to create a similar Typeclass with those 3 Instances. Now, click on the Source link to `SpaceAround`. You should find the Typeclass definition:

```
class SpaceAround a where
  spaceAround :: a
```

Next, search for `instance spaceAround` on the page (note the space between the words). You should find the following 3 instance definitions:

```
instance spaceAroundValue :: SpaceAround Value where
  spaceAround = fromString "space-around"

instance spaceAroundAlignContentValue :: SpaceAround AlignContentValue where
  spaceAround = fromString "space-around"

instance spaceAroundJustifyContentValue :: SpaceAround JustifyContentValue
  where spaceAround = fromString "space-around"
```

These instances limit where we can use the text `space-around` in our HTML that we generate.

Seems like we'll want to limit `space-evenly` in the exact same way. So, add the following code to your Counter Component:

```
import CSS.Flexbox (AlignContentValue, JustifyContentValue ❶
                  , flexDirection, row, justifyContent)
import CSS.Property (Value) ❷
import CSS.String (fromString) ❷

class SpaceEvenly a where
  spaceEvenly :: a

instance spaceEvenlyValue :: SpaceEvenly Value where
  spaceEvenly = fromString "space-evenly"

instance spaceEvenlyAlignContentValue
    :: SpaceEvenly AlignContentValue where
  spaceEvenly = fromString "space-evenly"

instance spaceEvenlyJustifyContentValue
    :: SpaceEvenly JustifyContentValue where
  spaceEvenly = fromString "space-evenly"
```

❶ Added `AlignContentValue` and `JustifyContentValue` to import list.

❷ Added these imports.

We probably should move this to a library of missing CSS features or even better, we should put in a Pull-request on Github, but since we're just learning here, we'll just put it in the Counter module.

Make sure your code is up to date.

Now, when you rerun the server you get:

```
[info] Build succeeded.
[error] Some of your project files import modules from packages that are not
in the direct dependencies of your project.
To fix this error add the following packages to the list of dependencies in
your config:
- css
```

Add `css` to `spago.dhall`.

Rerun the server and refresh the browser and you should see that the Counter looks so much nicer:



29.2.4. Counter Component Query Algebra

We're going to add a Query Algebra to our Component. Change `Query` from:

```
import Data.Const (Const) ❶

type Query :: ∀ k. k -> Type
type Query = Const Void
```

❶ This import should be removed.

to:

```
data Query a
  = SetCount Int a ❶
  | GetCount (Int -> a) ❷
```

❶ This query will set the current `count` Value. This is an `H.tell` type Query.

❷ This query will retrieve the current `count` Value and return it to the caller. This is an `H.request` type Query, i.e. it has a return Value.

When your Component is ***evaluated***, it's done using something called a `Free Monad`. This is an interesting technique and beyond the scope of this book, although you understand enough at this point to study it on your own.

But, the only issue to keep in mind is that with the `Free Monad`, it performs computations in a linear fashion with nested operations.

Take the following definition for a set of Operations that can be nested to form a Program:

```
data Op next ❶
  = Output String next  -- writes the message to the console
  | Warning next        -- writes a fixed warning message to the console
  | Done                -- terminates Program ❷

program :: Op
program =
  Warning (Output "This is a test" (Done)) ❸
```

❶ These are the operations that we can perform in our program.

❷ This operation allows our Program to terminate, i.e. it is the base-case for the recursive `Program` definition since it has NO `next`.

❸ Notice how our Program is a nested set of `Ops`.

Now, when this program is interpreted:

```
interpret :: Op -> Aff Unit
interpret prog = case _ of
  Output _ -> do
    next <- output prog ❶
    interpret next
  Warning _ -> do
    next <- warning prog ❶
    interpret next
  Done -> pure unit

output :: Op -> Aff (Op)
output (Output msg next) = log msg *> pure next ❷

warning :: Op -> Aff (Op)
warning (Warning next) = log "WARNING!!" *> pure next ❷
```

❶ Notice how we recurse with the rest of the Program. Each operation returns the next thing to interpret.

❷ Here we perform the operation and then we return the next thing to interpret.

This is an oversimplified version of what's going on when we write our Query Algebras. We have to remember to always return what's next. That's why our Algebra has Type `a` in each Data Constructor:

```
data Query a
  = SetCount Int a
  | GetCount (Int -> a)
```

We could have referred to `a` as `next` as a reminder that we have to make sure to return `next` from `handleQuery`:

```
data Query next
  = SetCount Int next
  | GetCount (Int -> next)
```

Typically, `a` is used and understood to be `next` when working with Queries.

In `handleQuery`, it's possible to terminate the evaluate of a Query, which is why we return a `Maybe a` instead of `a`:

```
handleQuery
  :: ∀ a
  .  Query a
  -> H.HalogenM State Action Slots Output m (Maybe a)
```

In some rare cases where Queries fail, you can return `pure Nothing` from `handleQuery`.

So, now, we're ready to code `handleQuery`. First, we'll start with the Type Signature and the start of the implementation:

```
handleQuery
  :: ∀ a
  .  Query a
  -> H.HalogenM State Action Slots Output m (Maybe a)
handleQuery = case _ of ❶
  ???
```

❶ Using wildcard `case` expression here. This is idiomatic.

Next, we'll handle `SetCount`:

```
handleQuery
  :: ∀ a
  .  Query a
  -> H.HalogenM State Action Slots Output m (Maybe a)
handleQuery = case _ of
  SetCount c a -> H.modify_ _ { count = c } *> pure (Just a) ❶ ❷
  ???
```

❶ Notice the somewhat confusing double underscores. The first is part of the Function name, `modify_`, and the second is a wildcard for the `State`, which we are updating with the remainder of the line.

❷ `count` in our `State` is set to the Value passed from the Query, i.e. `c`, and then we return `a`, i.e. the next thing to evaluate.

And finally, we'll handle `GetCount`:

```
handleQuery
  :: ∀ a
  .  Query a
  -> H.HalogenM State Action Slots Output m (Maybe a)
handleQuery = case _ of
  SetCount c a -> H.modify_ _ { count = c } *> pure (Just a)
  GetCount reply -> H.get >>= \{ count } -> pure $ Just $ reply count ❶
```

❶ Don't forget that the `reply` Function will return `a` back to you so that you can return it as the next thing to evaluate.

Update your code to add `handleQuery` under the `where` just like `handleAction`.

Don't forget to also add `handleQuery` to your `EvalSpec` before reading on.

Here's that final change:

```
  -- ABOVE CODE REDACTED
  , eval: H.mkEval H.defaultEval {
      initialize = Just Initialize
    , finalize = Just Finalize
    , handleAction = handleAction
    , handleQuery = handleQuery ❶
    }
  -- BELOW CODE REDACTED
```

❶ Added this line.

It would be nice if we could test this, but since we're a Top-level Component and not a Child Component and we're not going to have the Application query this component, we'll have to create a Counter as a Child to test `Query`.

29.2.5. Counter Component as a Child Component

We're going to change `Counter` to allow it to have Child Components that are also Counter Components.

To keep from infinitely adding `Counter` children, we'll pass the number of children we want a `Counter` to have:

```
component
  :: ∀ m
  .  MonadAff m
  => Int ❶ ❸
  -> H.Component Query Input Output m ❸
component numChildren = H.mkComponent ❷
  -- BELOW CODE REDACTED
```

❶ Added `Int` Type to the Type Signature.

❷ Added `numChildren` Parameter. Note, because we've added all of our other Functions under a `where`, they too have access to the Value of this Parameter.

❸ Make sure that the first Parameter is preceded by `=>` and return Parameter is preceded by `->`. It's easy to mess this up when editing to add another Parameter.

You're going to get a big compiler warning. That's because we introduced `numChildren` but haven't used it yet.

Next, we're going change the `Slots` Type to reflect that we can have children now:

```
-- type Slots :: ∀ k. Row k ❶
type Slots = ( counter :: H.Slot Query Output Int )
```

❶ Remove this line.

`H.Slot` is a Type Helper for defining slot Types. It takes the `query` Type, the `output` Type and then a `slot` Type that distinguishes similar Child Components from one another. Note that `query` and `output` are the Child's Types, i.e. the Child's `Query` and `Output`. It turns out that we are our own children, so that's why these are also our Types. This is not the norm.

And since we'll have 1 or more `Counter` Child Components, we're using `Int` to uniquely distinguish the Children from one another.

The only requirement for the `slot` Type is that must have an `Ord` Instance, so if you were to create a Sum Type to represent the unique Children, you'd have to derive `Ord` for it and, of course, `Eq` since it's `Ord's` Superclass.

Next, we'll create a `Proxy` for the `counter` label that's in the `Slots` Row Type:

```
import Type.Proxy (Proxy(..)) ❶

type Slots = ( counter :: H.Slot Query Output Int )

_counter = Proxy :: Proxy "counter" ❷
```

❶ Import `Proxy`.

❷ Normally, we write the Type Signature on a separate line, but this is idiomatic Halogen and terse.

We'll use this when we add our Children to the Virtual DOM in `render`.

Before we embark on rendering our Children, make sure that your code is caught up.

Next, we'll add the Children to the Virtual DOM in `render`:

```
import CSS.Geometry (width, paddingLeft, paddingTop, paddingBottom) ❶
import Data.Array (range) ❷

  render :: State -> H.ComponentHTML Action Slots m
  render { count } = let onClick = HE.onClick <<< const <<< Just in
    HH.div []
    ([ ❸
      HH.div [
        HC.style do
          display flex
          flexDirection row
          justifyContent spaceEvenly
          width (rem 6.0)
      ]
      [
        HH.button [ onClick Decrement ] [ HH.text "-" ]
      , HH.text $ show count
      , HH.button [ onClick Increment ] [ HH.text "+" ]
      ]
    ] <> children) ❸
    where
    children = if numChildren == 0 then [] else ❹
      [ HH.div [
          HC.style do
            display flex
            paddingLeft (rem 1.0)
            paddingTop (rem 1.0)
            paddingBottom (rem 1.0)
      ] $ range 0 (numChildren - 1) <#> \n -> ❺
        -- COMPILER ERROR!!
        HH.slot _counter n (component 0) (n + 1) FromChild] ❻
```

❶ Added `paddingLeft`, `paddingTop` and `paddingBottom` to import list.

❷ Added this import.

❸ Had to add Parentheses to be able to append `children` to the top `div`.

❹ We compute the `children` based on `numChildren`. It's an empty `Array` if no Children.

❺ Using `range` to create an `Array` to map over. With `numChildren` set to `3`, `range 0 (numChildren - 1)` will produce `[0, 1, 2]`.

❻ Create a slot with `HH.slot`.

Make sure that you update your code accordingly before advancing any further.

We'll ignore the compiler error for now and concentrate on the details of the `HH.slot` call:

```
HH.slot _counter n (component 0) (n + 1) FromChild
```

The first Parameter is the `Proxy` that refers to the label in the `Slots` Type.

The second Parameter uniquely identifies this Child from all others of the same Type, i.e. with the same label in the `Slots` Type, which, in our case, is `_counter`.

The third Parameter is the Component. We're creating it with NO children.

The fourth Parameter is the `Input` to the Component. This will be passed to `initialState` the first time and `receive` subsequent times and set the Counters to initially be one more than their unique id, i.e. 1, 2, 3, etc.

The fifth Parameter is the Slot Function that takes `Output` from the Child and converts it into an `Action` to be handled in `handleAction`.

Next, we should add `FromChild` to `Action`:

```
data Action
  = Initialize
  | Finalize
  | Increment
  | Decrement
  | FromChild Int ❶
```

❶ Added Data Constructor.

This fixes our old compiler error and we have another since we're not done yet.

So, let's finish by handling this new `Action` in `handleAction`:

```
handleAction
  :: Action
  -> H.HalogenM State Action Slots Output m Unit
handleAction = case _ of
  Initialize -> log "Initialize"
  Finalize -> log "Finalize"
  Increment -> H.modify_ \s -> s { count = s.count + 1 }
  Decrement -> H.modify_ \s -> s { count = s.count - 1 }
  FromChild c -> log $ "Received from Child: " <> show c ❶
```

❶ Added this case to just log what we've received from our Child.

Make sure you code is totally up to date and then rerun your server.

Did you get a compiler error?

That's because our Top-level Counter isn't getting it's new Parameter yet. Change `main` as follows:

```
main :: Effect Unit
main = do
  HA.runHalogenAff do
    body <- HA.awaitBody
    runUI (Counter.component 3) 0 body ❶
```

❶ Pass `3` for `numChildren`.

Try again.

Did you get:

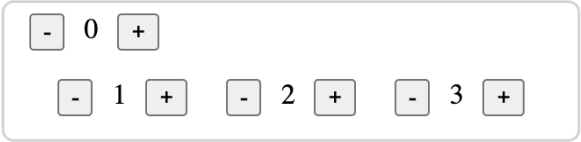
```
[warn] None of your project files import modules from some projects that are
in the direct dependencies of your project.
These dependencies are unused. To fix this warning, remove the following
packages from the list of dependencies in your config:
- const ❶
[error] Some of your project files import modules from packages that are not
in the direct dependencies of your project.
To fix this error add the following packages to the list of dependencies in
your config:
- arrays ❷
```

❶ No longer needed.

❷ We need this.

Update your `spago.dhall` to add `arrays` and remove `const`.

Now run the server and refresh your browser. You should get:



Currently, we have no way of calling `H.raise` to send `Output` to our Parent. So, let's add a button under the Counter, whose `Action` will call `H.raise` :

```
import CSS.Common (center)

render :: State -> H.ComponentHTML Action Slots m
render { count } = let onClick = HE.onClick <<< const <<< Just in
  HH.div []
  ([
    HH.div
      [ HC.style do
          display flex
          flexDirection row
          justifyContent spaceEvenly
          width (rem 6.0)
      ]
      [
        HH.button [ onClick Decrement ] [ HH.text "-" ]
      , HH.text $ show count
      , HH.button [ onClick Increment ] [ HH.text "+" ]
      ]
  , HH.div ❶
      [ HC.style do
          display flex
          justifyContent center
          width (rem 6.0)
          paddingTop (rem 0.5)
      ]
      -- COMPILER ERROR!!
      [ HH.button [ onClick RaiseParent ] [ HH.text "Raise" ] ] ❷
  ] <> children)
  where
  -- BELOW CODE REDACTED
```

❶ Added this `div`.

❷ We'll send an `Action` of `RaiseParent` when we click this button.

Update your `render` Function with this `div` before reading on.

Now, add the code to handle `RaiseParent` that calls `H.raise` with our current `count` Value. This should remove the compiler error. Then return here.

Here are those changes:

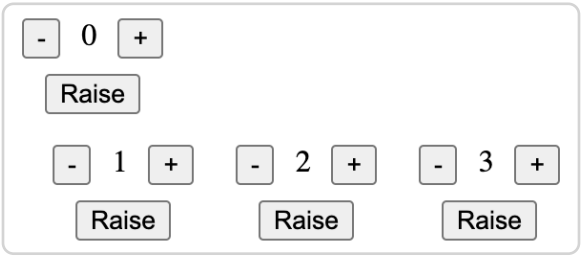
```
data Action
  = Initialize
  | Finalize
  | Increment
  | Decrement
  | FromChild Int
  | RaiseParent ❶

handleAction
  :: Action
  -> H.HalogenM State Action Slots Output m Unit
handleAction = case _ of
  Initialize -> log "Initialize"
  Finalize -> log "Finalize"
  Increment -> H.modify_ \s -> s { count = s.count + 1 }
  Decrement -> H.modify_ \s -> s { count = s.count - 1 }
  FromChild c -> log $ "Received from Child: " <> show c
  RaiseParent -> H.get >>= \{ count } -> H.raise count ❷
```

❶ Added this Data Constructor.

❷ Handle `RaiseParent` by getting the State via `H.get` and then binding to a Function that'll destructure the `count` and send it to the Parent.

Now, rerun your server, refresh your browser and you should have a screen like:



Now, make sure your console is open. You should have the following:



This shows that all 4 Counters initialized. Chrome will display a count next to lines that have been contiguously logged multiple times with the exactly same message.

Now click `Raise` under the Counter that has a value of 2. It should log:

```
Received from Child: 2
```

Now click on the `+` button 3 times on that same Counter. It should now read 5. Now click again on the `Raise`

button and you should now see:

```
Received from Child: 5
```

Now, click on `Raise` for the Top-level Counter. What do you see in the log?

I see nothing displayed when I do that. But why?

It's because a Top-level Component, by definition, does not have a Parent. So, the App will have to play the role of the Parent.

To support this, we're going to have to look at what `runUI` returns:

```
runUI
  :: ∀ query input output
  .  Component HTML query input output Aff
  -> input
  -> HTMLElement
  -> Aff (HalogenIO query output Aff)
```

It returns a `HalogenIO`, which has the following definition:

```
type HalogenIO query output m =
  { dispose :: m Unit
  , messages :: Emitter output ❶
  , query :: ∀ a. query a -> m (Maybe a)
  }
```

❶ This `Emitter` emits Values of Type `output`.

We can subscribe to the `Emitter` that we get from `messages`.

We're going to use a helper Function from `Halogen.Subscription` called `subscribe` to subscribe to Values of Type `output`:

```
import Effect.Class (liftEffect) ❶
import Effect.Class.Console (log)
import Halogen.Subscription as HS

main :: Effect Unit
main = do
  HA.runHalogenAff do
    body <- HA.awaitBody
    io <- runUI (Counter.component 3) 0 body ❷
    void $ liftEffect $ HS.subscribe io.messages \currentCount -> ❸
      log $ "Received Count from Top-level Counter: "
        <> show currentCount ❹
```

❶ Added these imports.

❷ Capture the return from `runUI`, i.e. `io`, which is of Type `HalogenIO`.

❸ Subscribe to an `Emitter` that takes a Function whose first and only Parameter is a Value of Type `output` from a Top-level Component. In this case, the Top-level Component is a `Counter` and it's `Output` Type is an `Int`.

❹ Log the current `Counter` Value.

Update your `main` accordingly and rerun the server. Ooops:

```
[info] Build succeeded.
[error] Some of your project files import modules from packages that are not
in the direct dependencies of your project.
To fix this error add the following packages to the list of dependencies in
your config:
- halogen-subscriptions
```

We need to add `halogen-subscriptions` to `spago.dhall`

Rerun the server, refresh the browser and then click the Top-level Counter's `Raise` button and see what is printed in the console.

Here's what I get:

```
Received Count from Top-level Counter: 0
```

Now, let's set a Child Counter to `123` whenever it sends us `Output`. To query a Child, we use either `H.tell` or `H.request`.

We use `H.tell` when we're simply telling the Child to do something. We use `H.request` when we expect something to be returned from the Child.

Since we're going to set the Child's Value using `SetValue`, we're going to use `H.tell`:

```
tell
  :: ∀ state action output m label slots query output' slot _1
   . Row.Cons label (Slot query output' slot) _1 slots
  => IsSymbol label
  => Ord slot
  => Proxy label
  -> slot
  -> Tell query
  -> HalogenM state action slots output m Unit
```

This is a daunting Type Signature. But, we just care about the Parameters, which are `Proxy label`, `slot` and `Tell query`.

That means we'll write something like:

```
H.tell _counter someSlotNumber ...
```

Here `someSlotNumber` is the slot number that identifies our Child.

And `_counter` is our `Proxy` for the label in our `Slots` Type:

```
type Slots = ( counter :: H.Slot Query Output Int ) ❶
```

❶ Here `Int` is the Type of the Value that uniquely identifies a particular `slot` that a Child occupies.

`H.tell` is how you tell a Child to perform a Procedure, i.e. do something with no return Value, as opposed to `H.request` that tells the Child to perform a Function, i.e. compute something returning a Value.

Here's the rest of the `H.tell` call:

```
H.tell _counter someSlotNumber $ SetCount 123
```

The problem is that we still have to figure out what `slot` the Child is in that called `H.raise`, i.e. what number to put in place of `someSlotNumber`.

Looking at the `FromChild` Data Constructor, we can see that we have no information regarding that:

```
data Action
  = Initialize
  | Finalize
  | Increment
  | Decrement
  | FromChild Int ❶
  | RaiseParent
```

❶ This `Int` is the count Value NOT which Child.

We'll need to add another Parameter to `FromChild` that contains the `slot` Value.

Go ahead and change the `FromChild` Data Constructor to have another Parameter for `slot`, then change `handleAction` code to set the Child's count Value to `123` in `FromChild` keeping the existing `log` functionality and finally update the `render` Function where `FromChild` is used to map Child `Outputs` to `Actions` to include the Child's `slot` identifier, i.e. it's slot number.

Then return here.

First, we'll add another `Int` to `FromChild`.

```
data Action
  = Initialize
  | Finalize
  | Increment
  | Decrement
  | FromChild Int Int ❶
  | RaiseParent
```

❶ The first `Int` is the slot number since we're going to Partially Apply this in the `render` Function.

Those 2 `Int` Parameters to `FromChild` are confusing. Which `Int` is which again? Let's fix that:

```
type SlotNum = Int ❶
type CounterValue = Int

data Action
  = Initialize
  | Finalize
  | Increment
  | Decrement
  | FromChild SlotNum CounterValue ❷
  | RaiseParent
```

❶ Create Type Aliases to distinguish the different `Int`s.

❷ Use the new Type Aliases.

Next, we'll update `FromChild` in `handleAction`:

```
handleAction
  :: Action
  -> H.HalogenM State Action Slots Output m Unit
handleAction = case _ of
  Initialize -> log "Initialize"
  Finalize -> log "Finalize"
  Increment -> H.modify_ \s -> s { count = s.count + 1 }
  Decrement -> H.modify_ \s -> s { count = s.count - 1 }
  FromChild slot c -> do ❶ ❷
    void $ H.tell _counter slot $ SetCount 123 ❸
    log $ "Received from Child: " <> show c ❹
  RaiseParent -> H.get >>= \{ count } -> H.raise count
```

❶ Added `slot` Parameter as the first Parameter.

❷ Added a `do` block since we're doing multiple steps now.

❸ We `query` the `slot` of the Child who sent us `Output` and `tell` it to `SetCount` to `123`. We had do use `void` since this returns a `Maybe Unit`.

❹ We still log like we did before.

And finally, we need to Partially Apply `FromChild` in `render` with the `slot` number:

```
  -- ABOVE CODE REDACTED
    where
    children = if numChildren == 0 then [] else
      [ HH.div [
        HC.style do
          display flex
          paddingLeft (rem 1.0)
          paddingTop (rem 1.0)
          paddingBottom (rem 1.0)
      ] $
        range 0 (numChildren - 1) <#> \n ->
          HH.slot _counter n (component 0)
           (n + 1) (FromChild n) ] ❶
```

❶ Pass the slot number, i.e. `n`, as the first Parameter to `FromChild`.

Now, rerun your server and refresh your Browser. Then click on the third Child Counter's `Raise` Button. It should change its value to `123` and log the following:

```
Received from Child: 3
```

Fix this message so it also displays which Child it received from, e.g. `Received from Child 2: 5`. Then continue reading.

Here's that change:

```
  -- ABOVE CODE REDACTED
    FromChild slot c -> do
      void $ H.tell _counter slot $ SetCount 123
      log $ "Received from Child " <> show slot <> ": " <> show c ❶
  -- BELOW CODE REDACTED
```

❶ Added the slot info to the log output.

Re-testing should now log:

```
Received from Child 2: 3
```

That can be confusing since humans don't count starting at zero. Change the code to add `1` to the `slot`:

```
  -- ABOVE CODE REDACTED
    FromChild slot c -> do
      void $ H.tell _counter slot $ SetCount 123
      log $ "Received from Child " <> show (slot + 1) <> ": " <> show c ❶
  -- BELOW CODE REDACTED
```

❶ Added `1` to `slot`.

Now it displays:

```
Received from Child 3: 3
```

Now, request the counter Value in `FromChild` right AFTER we change it to `123`. Use `H.request` and `GetCount`. Realize that `H.request` will return a `HalogenM state action slots output m (Maybe a)`.

Once you have that value, then log it to the console.

Add that code before returning here.

Here's that addition:

```
  -- ABOVE CODE REDACTED
    FromChild slot c -> do
      void $ H.tell _counter slot $ SetCount 123
      count <- H.request _counter slot GetCount ❶
      log $ "Received from Child " <> show (slot + 1) <> ": " <> show c
      log $ "Queried from Child " <> show (slot + 1) <> ": " <> show count ❷
  -- BELOW CODE REDACTED
```

❶ Query Child for it's current `count` value, which will always be `Just 123` since we just set it.

❷ Here's where we log the requested value to the console.

And when the server is rerun and the browser refreshed, clicking on the third Child's `Raise` button logs:

```
Received from Child 3: 3
Queried from Child 3: (Just 123)
```

`H.tell` and `H.request` are the 2 different ways to query a Child Component.

One last thing to do is to implement `receive` in the `EvalSpec` so Parent re-renders will send Children their Input data. But before we do this, we should pay close attention to our current Counter behavior and in particular the interaction between the Parent and Children Counters.

Make sure your server is running and go back and refresh your browser. The first Child Counter should have a value of 1 to begin with. Now click the `+` button once. It's value is now 2.

Now, click on the `+` button of the Parent Counter, whose value is currently `0`. It should change to `1` and the first Child Counter should still be `2`.

Note that re-rendering the Parent did NOT affect the Child Counter's State. Keep this in mind as we move forward.

We're going to implement `receive` and update our State when we get the `Input` Value from our Parent. This will happen each time our Parent re-renders.

First, add to the `Action`:

```
data Action
  = Initialize
  | Finalize
  | Increment
  | Decrement
  | FromChild SlotNum CounterValue
  | RaiseParent
  | Received Int ❶
```

❶ Added `Received` Data Constructor with the `Input` Type as its one and only Parameter.

Next, add `receive` to your `EvalSpec` to return our new `Action` remembering that the return Value is optional, i.e. a `Maybe`.

Then return back here.

Here's that change:

```
  -- ABOVE CODE REDACTED
  , eval: H.mkEval H.defaultEval {
      initialize = Just Initialize
    , finalize = Just Finalize
    , handleAction = handleAction
    , handleQuery = handleQuery
    , receive = Just <<< Received ❶
    }
  -- BELOW CODE REDACTED
```

❶ `receive` is a Function that takes 1 Parameter, i.e. the `Input` Value. That's why it's composed using `<<<`

And finally, change `handleAction` to handle this new `Action` to set the `count` to the `Input` Value from `receive`. Then return back here.

Here's that change:

```
handleAction
  :: Action
  -> H.HalogenM State Action Slots Output m Unit
handleAction = case _ of
  Initialize -> log "Initialize"
  Finalize -> log "Finalize"
  Increment -> H.modify_ \s -> s { count = s.count + 1 }
  Decrement -> H.modify_ \s -> s { count = s.count - 1 }
  FromChild slot c -> do
    void $ H.tell _counter slot $ SetCount 123
    count <- H.request _counter slot GetCount
    log $ "Queried from Child " <> show (slot + 1) <> ": " <> show count
    log $ "Received from Child " <> show (slot + 1) <> ": " <> show c
  RaiseParent -> H.get >>= \{ count } -> H.raise count
  Received c -> H.modify_ _ { count = c } ❶
```

❶ We replace the count with the Value we received from our Parent.

Now, rerun the server and refresh the browser. The first Child Counter should have a value of 1 to begin with. Now click the `+` button once. It's value should now be 2.

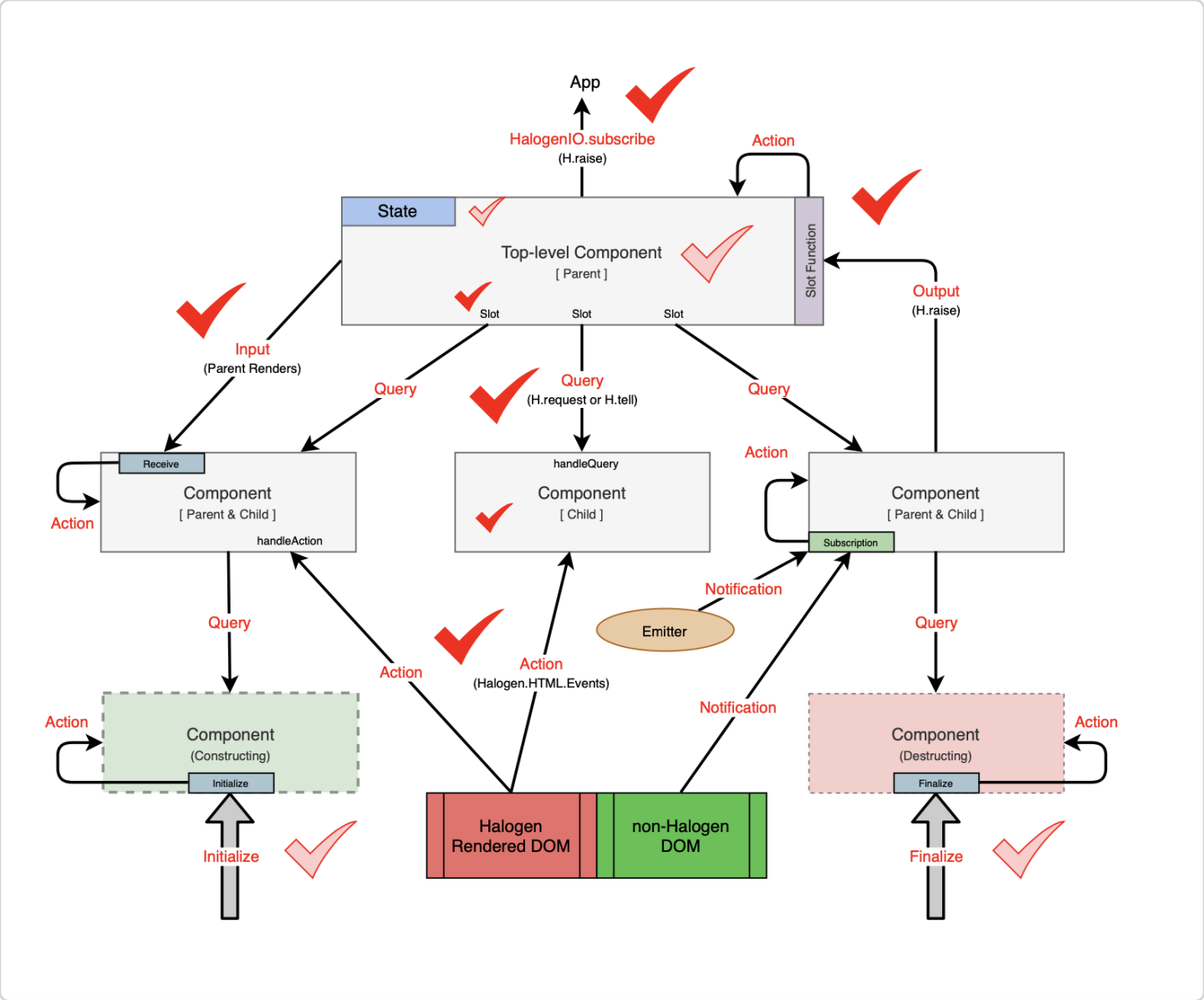
Now, click on the `+` button of the Parent Counter whose value is currently `0`. It should change to `1` and the first Child Counter should now be BACK to `1`.

This is because our Parent re-rendered when we clicked its `+` button and, currently, the Parent is rendering its Children with their INITIAL count values.

You may be wondering when to use `receive` and when not to. I would suggest using it when the Parent maintains all or some of the Child's State. The Child will still have its own State but it will copy all or some of its State from the Parent.

Since Components re-render on a State change, a change to the Parent's State will cause the Parent to re-render causing `receive` to be called in each Child. When the Child handles `receive`, the Child updates its State causing it to re-render.

Now, let's take a look at our diagram to see how much we've accomplished:



In this section, we've accomplished the following:

- Subscribe to Top-level Component's Output
- Added Children to Slot
- Rendered Children
- Send Output from Child to Parent that's Handled by Parent
- Queried Child with `H.tell` and `H.request`
- Halogen Rendered DOM Events are handled as Actions
- Receive Input from Parent when Parent re-renders

29.2.6. Emitters

Next, we're going to work with `Emitters`. `Emitters` emit Values of a particular Type. An `Emitter` is

defined as:

```
newtype Emitter a
```

The `a` here is the Type of Value it'll emit.

`Emitters` emit and `Listeners` listen for those emission. We create these 2 using a single Function, `HS.create`:

```
import Halogen.Subscription as HS

{ emitter, listener } <- H.liftEffect HS.create
```

Now we can emit things using `HS.notify`:

```
import Halogen.Subscription as HS

HS.notify listener someValue ❶
```

❶ We `notify` the `listener` with `someValue`, which is of Type `a` in `Emitter a`.

When used with HQ.subscribe, we can send `Actions` to our Component:

```
import Halogen.Query.HalogenM as HQ

HQ.subscribe emitter ❶

HS.notify listener $ DoAction value ❷
```

❶ Subscribe to an `Emitter Action`.

❷ Emit an `Action` that will trigger `handleAction` in our Component.

We can also attach to the DOM to listen to events like non-Halogen DOM events, e.g. `document` Events, or the resize Event on the Browser Window. We use `eventListener` from `Halogen.Query.Event` to accomplish that:

```
eventListener
  :: ∀ a
   . Event.EventType
  -> EventTarget.EventTarget
  -> (Event.Event -> Maybe a)
  -> HS.Emitter a
```

To leverage these features, we're going to first create a `CountdownTimer` Component that will be another Top-level Component in our Application. When the timer expires, it'll send `Output` via `H.raise` to the Application letting it know that it's time to dispose of the Top-level Counter Component. This will in turn, remove all of its Child Counters.

Start by creating a file under `src/Component` called `CountdownTimer.purs`.

Here are the specs for `CountdownTimer`:

- Keeps track of the current count value
- Initializes its count value from its `Input`
- Will count from its initial value down to `0`
- Has an EventSource that's based on a 1 second timer `Fiber`
- Once the current count hits `0` the Fiber will be killed and the Parent notified

Next, code the Types for `Input`, `Output`, `Slots` and `State`. Then return here.

Here's our module so far:

```
module Component.CountdownTimer where

import Prelude

type Input = Int ❶
type Output = Unit ❷

type State = { count :: Int } ❸

type Slots :: ∀ k. Row k
type Slots = () ❹
```

❶ The `Input` is a starting `count` Value.

❷ The `Output` is just a signal to the Parent and contains no other information, hence the use of `Unit`.

❸ The `State` only needs its current `count`.

❹ There are no Child Components, hence the empty Row Type.

Next, code the Types for `Query` and `Action` before reading any further.

Here are those Types:

```
import Data.Const (Const) ❶

type Query :: ∀ k. k -> Type
type Query = Const Void ❷

data Action
  = Initialize ❸
  | Tick ❹
```

❶ Add this import.

❷ We have no `Queries`.

❸ The `Initialize` Action will start our timer on .

❹ We'll emit an `Tick` Action every second until our counter expires, i.e. reaches `0`.

Next, write as much of the `component` Function as you can using `Counter` as an example. Then return here and we'll move forward together.

First, we'll write the Type Signature:

```
import Effect.Aff.Class (class MonadAff) ❶
import Halogen as H
import Halogen.HTML as HH

component
  :: ∀ m
  .  MonadAff m
  => H.Component Query Input Output m
component = ???
```

❶ Added these imports.

Now, we'll call `H.mkComponent`:

```
component
  :: ∀ m
  .  MonadAff m
  => H.Component Query Input Output m
component = H.mkComponent
  { initialState: \i -> { count: i } ❶
  , render ❷
  , eval: ???
  }
```

❶ Initialize our State.

❷ `render` still needs to be written.

Next, we'll call `mkEval` using `defaultEval` as our default `EvalSpec`:

```
import Data.Maybe (Maybe(..)) ❶

component
  :: ∀ m
  .  MonadAff m
  => H.Component Query Input Output m
component = H.mkComponent
  { initialState: \i -> { count: i }
  , render -- COMPILER ERROR!!
  , eval: H.mkEval H.defaultEval {
      initialize = Just Initialize ❷
    , handleAction = handleAction ❸
    }
  }
```

❶ Added this import.

❷ We return the `Initialze` Action when we're constructed. This will initialize our `EventSource` in `handleAction`.

❸ We need to handle `Actions`. Note the use of the `=`. That's because we're updating the default field.

Our `EvalSpec` is much simpler for this Component since it's functionality is limited when compared with `Counter`.

The compiler error is expected at this point since we haven't written `render`. So, write a stub to keep the compiler happy. Look back at `render` from `Counter` to help you. Then return here and continue reading.

Here's that addition:

```
component
  :: ∀ m
  .  MonadAff m
  => H.Component Query Input Output m
component = H.mkComponent
  { initialState: \i -> { count: i }
  , render
  , eval: H.mkEval H.defaultEval {
      initialize = Just Initialize
    , handleAction = handleAction -- COMPILER ERROR!!
    }
  }
  where
  render :: State -> H.ComponentHTML Action Slots m
  render _ = HH.text ""
```

Now, write a stub for `handleAction` to eliminate our new error. Don't forget to steal the Type Signature from `handleAction` in `Counter`. Then return back here.

Here is the stub for `handleAction`:

```
  -- ABOVE CODE REDACTED
  where
  handleAction
    :: Action
    -> H.HalogenM State Action Slots Output m Unit
  handleAction = case _ of
    Initialize -> pure unit
    Tick -> pure unit

  render :: State -> H.ComponentHTML Action Slots m ❶
  render _ = HH.text ""
```

❶ I like to have `render` at the bottom of my `where` since it's typically verbose. This way I don't have to scroll past a bunch of `render` code just to change `handleAction` or `handleQuery`.

Now that we have the skeleton of our Component done, it's time to flesh out the rest (pun intended).

We'll start with creating the `Emitter` and `Listener`. We'll code this together since it's new:

```
import Halogen.Subscription as HS ❶

  -- ABOVE CODE REDACTED
  where
  handleAction
    :: Action
    -> H.HalogenM State Action Slots Output m Unit
  handleAction = case _ of
    Initialize -> do
      { emitter:tickEmitter, listener:tickListener } <- H.liftEffect HS.create
❷

      ???
    Tick -> pure unit
```

❶ Added this import.

❷ We start by creating an `Emitter` and `Listener` renaming them to `tickEmitter` and `tickListener` respectively.

Next, we'll subscribe to the `Emitter` using `H.subscribe`. This returns the `SubscriptionId` so we can `unsubscribe` if we choose.

Subscription are automatically unsubscribed by Halogen when a Component is Finalized. But in our code, we're going to want to unsubscribe once the `CountdownTimer` has reached `0`. If we didn't care about the `SubscriptionId`, we could've called the Prime version of this Function, `H.subscribe'`.

Here's the code for subscribing:

```
import Halogen.Query.HalogenM as HQ ❶

  -- ABOVE CODE REDACTED
  where
  handleAction
    :: Action
    -> H.HalogenM State Action Slots Output m Unit
  handleAction = case _ of
    Initialize -> do
      { emitter:tickEmitter, listener:tickListener } <- H.liftEffect HS.create
      tickSubscription <- HQ.subscribe tickEmitter ❷
      ???
    Tick -> pure unit
```

❶ Added this import.

❷ Added `H.subscribe'`.

Next, we'll create a `Fiber` with `forkAff` to `emit` a `Tick` every second:

```
import Control.Monad.Rec.Class (forever) ❶
import Data.Time.Duration (Milliseconds(..))
import Effect.Aff (forkAff, delay)

  -- ABOVE CODE REDACTED
  where
  handleAction
    :: Action
    -> H.HalogenM State Action Slots Output m Unit
  handleAction = case _ of
    Initialize -> do
      { emitter:tickEmitter, listener:tickListener } <- H.liftEffect HS.create
      tickSubscription <- HQ.subscribe tickEmitter
      tickFiber <- H.liftAff $ forkAff $ forever do ❷
        delay $ Milliseconds 1000.0 ❸
        H.liftEffect $ HS.notify tickListener $ Tick tickSubscription ❹
        ???
    Tick -> pure unit
```

❶ Added these import.

❷ Loop `forever`.

❸ Delay 1 second.

❹ Emit the `Tick` Action. Notice the `sid` Parameter. This is so we can `unsubscribe` once our counter reaches `0`. This means that our `Tick` definition needs to change.

Let's add the `SubscriptionId` to our Tick definition next:

```
data Action
  = Initialize
  | Tick H.SubscriptionId ❶
```

❶ We added a Parameter to the `Tick` Action.

And now we'll add the Parameter to `Tick` in `handleAction`:

```
-- ABOVE CODE REDACTED
where
handleAction
  :: Action
  -> H.HalogenM State Action Slots Output m Unit
handleAction = case _ of
  Initialize ->
    H.subscribe' \sid -> EventSource.affEventSource \emitter -> do
      fiber <- forkAff $ forever do
        delay $ Milliseconds 1000.0
        EventSource.emit emitter (Tick sid)
      ???
  Tick sid -> pure unit ❶
```

❶ Added `SubscriptionId` Parameter called `sid`.

Next, we need to save `fiber` to our State so we can kill it on Finalize or when the counter reaches `0`:

```
-- ABOVE CODE REDACTED
where
handleAction
  :: Action
  -> H.HalogenM State Action Slots Output m Unit
handleAction = case _ of
  Initialize -> do
    { emitter:tickEmitter, listener:tickListener } <- H.liftEffect HS.create
    tickSubscription <- HQ.subscribe tickEmitter
    tickFiber <- H.liftAff $ forkAff $ forever do
      delay $ Milliseconds 1000.0
      H.liftEffect $ HS.notify tickListener $ Tick tickSubscription
    H.modify_ _ { tickFiber' = Just tickFiber } ❶
  Tick sid -> pure unit
```

❶ Save `tickFiber'` in State so we can stop it later with `killFiber`.

We now need to add `tickFiber'` to our State:

```
import Effect.Aff (Fiber, forkAff, delay) ❶

type State =
  { count :: Int
  , tickFiber' :: Maybe (Fiber Unit) ❷
  }
```

❶ Added `Fiber` to the import list.

❷ Added `tickFiber'` to `State`. Note the Prime in the name. That helps me remember it's a Maybe

And now we have to update `initalState` to account for the change in our `State`:

```
component
  :: ∀ m
  .  MonadAff m
  => H.Component Query Input Output m
component = H.mkComponent
  { initialState: \i -> { count: i, tickFiber': Nothing } ❶
  , render
  , eval: H.mkEval H.defaultEval {
      initialize = Just Initialize
    , handleAction = handleAction
    }
  }
  where
  -- BELOW CODE REDACTED
```

❶ Initialize to NO fiber running.

Now, we're ready to work on the `Tick` Action:

```
-- ABOVE CODE REDACTED
where
handleAction
  :: Action
  -> H.HalogenM State Action Slots Output m Unit
handleAction = case _ of
  Initialize -> do
    { emitter:tickEmitter, listener:tickListener } <- H.liftEffect HS.create
    tickSubscription <- HQ.subscribe tickEmitter
    tickFiber <- H.liftAff $ forkAff $ forever do
      delay $ Milliseconds 1000.0
      H.liftEffect $ HS.notify tickListener $ Tick tickSubscription
    H.modify_ _ { tickFiber' = Just tickFiber }
  Tick sid -> do
    { count } <- H.get ❶
    ???
```

❶ Get the count from the our State.

Next, we'll countdown the `count` as long as it's a Positive Integer:

```
-- ABOVE CODE REDACTED
  Tick sid -> do
    { count } <- H.get
    unless (count <= 0) $ H.modify_ _ { count = count - 1 } ❶
    ???
```

❶ We decrement our `count` as long as it's not `0` or less.

And finally, when `count` reaches `0`, we'll inform our Parent and unsubscribe to the `EventSource` causing it's `Finalizer` to run, which will kill the timer `Fiber`:

```
-- ABOVE CODE REDACTED
  Tick sid -> do
    { count } <- H.get
    unless (count <= 0) $ H.modify_ _ { count = count - 1 }
    when (count == 1) do
      H.raise unit ❶
      H.unsubscribe sid ❷
      killTickFiber ❸
```

❶ Tell Parent of the timer expiration via `H.raise`. We're just sending `unit` since there's no real information to pass other than an event has occurred.

❷ Unsubscribe from the `EventSource` using the passed in `sid` Value.

❸ Kill the fiber (we need to write this).

Now, let's write `killTickFiber`:

```
import Data.Maybe (Maybe(..), maybe) ❶
import Effect.Aff (Fiber, forkAff, delay, killFiber) ❷
import Effect.Exception (error) ❸

  -- ABOVE CODE REDACTED
    Tick sid -> do
      { count } <- H.get
      unless (count <= 0) $ H.modify_ _ { count = count - 1 }
      when (count == 1) do
        H.raise unit
        H.unsubscribe sid
        killTickFiber
    where
      killTickFiber = do
        { tickFiber' } <- H.get
        tickFiber' # maybe (pure unit) \tickFiber -> do
          H.liftAff $ killFiber (error "Event source finalized") tickFiber ❹
          H.modify_ _ { tickFiber' = Nothing } ❺
```

❶ Added `maybe` to import list.

❷ Added `killFiber` to import list.

❸ Added this import.

❹ Kill fiber.

❺ Clear `tickFiber'` so no one else tries to kill it again. BTW, you're program won't crash if you try to kill a non-running fiber, but this is just good practice just in case one day the `Fiber` implementation changes.

We should probably kill the tick `Fiber` if we `Finalize` the Component and the counter hasn't gotten to zero:

```
data Action
  = Initialize
  | Finalize ❶
  | Tick H.SubscriptionId

-- ABOVE CODE REDACTED
Finalize -> killTickFiber ❷
Tick sid -> do
  { count } <- H.get
  unless (count <= 0) $ H.modify_ _ { count = count - 1 }
  when (count == 1) do
    H.raise unit
    H.unsubscribe sid
    killTickFiber
-- BELOW CODE REDACTED
```

❶ Added `Finalize` Action.

❷ Kill the fiber if it's still running when we get finalized.

Make sure your code is up to date and compiles before reading on.

The last thing we need to do is finish coding `render`:

```
import CSS.Display (display, flex) ❶
import CSS.Flexbox (flexDirection, column)
import CSS.Geometry (padding)
import CSS.Size (rem)
import Halogen.HTML.CSS as HC

  -- ABOVE CODE REDACTED
  render :: State -> H.ComponentHTML Action Slots m
  render { count } =
    HH.div
      [ HC.style do
          display flex
          flexDirection column
          padding (rem 1.0) (rem 1.0) (rem 1.0) (rem 1.0)
      ]
      [ HH.text $ "COUNTDOWN: " <> show count ]
```

❶ Add these imports.

Take a minute to understand the `render` code before you update your code with it. You've seen all this

before except for maybe `padding`, which takes padding Values for `Size` in the same order as `padding` does in CSS, i.e. top, right, bottom and left.

Now, we're going to add the `CountdownTimer` as another Top-level Component. We'll add it to the `body`, which means it'll get appended to the `body` we already have with our `Counters`, i.e. assuming we add it AFTER the `Counter`.

Returning to our `Main` module, we'll make the following changes:

```
import Component.CountdownTimer as CountdownTimer ❶

main :: Effect Unit
main = do
  HA.runHalogenAff do
    body <- HA.awaitBody
    io <- runUI (Counter.component 3) 0 body
    io.subscribe $ CR.consumer \currentCount -> do
      log $ "Received Count from Top-level Counter: " <> show currentCount
      pure Nothing
    void $ runUI CountdownTimer.component 15 body ❷ ❸
    ???
```

❶ Added this import.

❷ Start the `CountdownTimer` with a `15` second countdown.

❸ We're discarding the `HalogenIO` Value for the timer Component since we're not going to use it.

Now, we're going to subscribe to the timer expiration event and when it fires, we'll `dispose` of the `Counter` Component:

```
import Effect.Aff (launchAff_) ❶

main :: Effect Unit
main = do
  HA.runHalogenAff do
    body <- HA.awaitBody
    io <- runUI (Counter.component 3) 0 body
    io.subscribe $ CR.consumer \currentCount -> do
      log $ "Received Count from Top-level Counter: " <> show currentCount
      pure Nothing
    ioTimer <- runUI CountdownTimer.component 15 body
    void $ liftEffect $ HS.subscribe ioTimer.messages ❷
      \_ -> launchAff_ io.dispose ❸ ❹
```

❶ Added this import.

❷ We subscribe to second Top-level Component via `ioTimer`.

❸ We throw away the `unit` that the Lambda is passed from the `CountdownTimer's` call to `H.raise`.

❹ `HalogenIO` has a `dispose` that we're calling to dispose of the other Top-level Component, `Counter`.

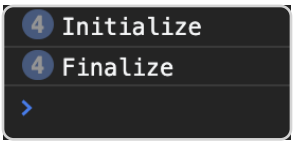
Make sure that your code is up to date and re-build the app and run the server. You'll see:

```
[info] Build succeeded.
[error] Some of your project files import modules from packages that are not
in the direct dependencies of your project.
To fix this error add the following packages to the list of dependencies in
your config:
- const
- datetime
- exceptions
- tailrec
```

Add these to the `spago.dhall` file.

Now, re-build the app and refresh your browser and watch the `COUNTDOWN`. When it hits `0`, you should see two things.

First, the `Counter` and its Children disappear. Second, you should see in the console that the `finalize` for all 4 `Counters` got called:



So, that worked and that's great, but the fact we have to do the clean-up ourselves of the `tickFiber` in our `Finalize` is problematic. We could easily forget this. Halogen already unsubscribes from the emitter when we're finalized, so we don't have to remember. But our emitter has a `Fiber` that we really should kill. If we don't kill it, the `notify` call will do nothing since there are no subscribers, but we have a `Fiber` that's running and doing nothing but generating heat.

If our page was in a larger application, then a new `Fiber` would be created every single time the user navigated to our page. So it would be possible to have dozens and dozens of these dead `Fibers`. Not a good situation.

This is why we explicitly handled the killing of the `Fiber`. But there must be a better way to handle this. And it took me a bit to figure out one, but there is. I'm simply going to just give you the code and then I'll explain each and every line so you can understand how it works.

First, comment out the following code:

```
    -- Initialize -> do
    --   -- timer setup
    --   { emitter:tickEmitter, listener:tickListener } <- H.liftEffect
HS.create
    --   tickSubscription <- HQ.subscribe tickEmitter
    --   tickFiber <- H.liftAff $ forkAff $ forever do
    --     delay $ Milliseconds 1000.0
    --     H.liftEffect $ HS.notify tickListener $ Tick tickSubscription
    --   H.modify_ _ { tickFiber' = Just tickFiber }
```

And add the following in its place:

```
    Initialize -> do
      -- timer setup
      { emitter:tickEmitter, listener:tickListener } <- H.liftEffect HS.create
      tickFiber <- H.liftAff $ forkAff $ forever do -- 0.14
        log "Sleeping..."
        delay $ Milliseconds 1000.0
        H.liftEffect $ HS.notify tickListener Tick
      let fiberEmitter = HS.makeEmitter \_ -> pure $ launchAff_ do
            log "Killing tick fiber..."
            killFiber (error "Event source finalized") tickFiber
      HQ.subscribe' \sid -> (tickEmitter <|> fiberEmitter) <#> (_ $ sid)
```

Now, comment out our the call to `killTickFiber` in our `Tick` handler and the `killTickFiber` Function:

```
    Tick sid -> do
      { count } <- H.get
      unless (count <= 0) $ H.modify_ _ { count = count - 1 }
      when (count == 1) do
        H.raise unit
        H.unsubscribe sid
    --     killTickFiber
    -- where
    --   killTickFiber = do
    --     { tickFiber' } <- H.get
    --     tickFiber' # maybe (pure unit) \tickFiber -> do
    --       H.liftAff $ killFiber (error "Event source finalized") tickFiber
    --       H.modify_ _ { tickFiber' = Nothing }
```

You'll also need to modify your imports:

```
import Control.Alt ((<|>)) ❶
import Effect.Aff (Fiber, forkAff, delay, killFiber, launchAff_) ❷
```

❶ Added this import. This will need the `control` package in `spago.dhall`.

❷ Added `launchAff_` to import list.

Then add `control` to your `spago.dhall`:

```
{-
Welcome to a Spago project!
You can edit this file as you like.
-}
{ name = "halo"
, dependencies =
  [ "aff"
  , "arrays"
  , "console"
  , "const"
  , "control" ❶
  , "css"
  , "datetime"
  , "effect"
  , "exceptions"
  , "halogen-css"
  , "halogen-subscriptions"
  , "halogen"
  , "maybe"
  , "prelude"
  , "psci-support"
  , "tailrec"
  , "web-dom"
  , "web-events"
  , "web-html"
  , "web-uievents"
  ]
, packages = ./packages.dhall
, sources = [ "src/**/*.purs", "test/**/*.purs" ]
}
```

❶ Added this package since we added `import Control.Alt`.

And now comment out:

```
data Action
  = Initialize
-- | Finalize ❶
  | Tick H.SubscriptionId
  | HandleDocClick HTMLDocument MouseEvent

component
  :: ∀ m
  .  MonadAff m
  => H.Component Query Input Output m
component = H.mkComponent
  { initialState: \i -> { count: i, tickFiber': Nothing }
  , render
  , eval: H.mkEval H.defaultEval {
      initialize = Just Initialize
    -- , finalize = Just Finalize ❷
    , handleAction = handleAction
    }
  }
  where
  -- BELOW CODE REDACTED
```

❶ We don't need to manually kill our `Fiber`, so we don't need to handle `Finalize`.

❷ This too can go.

And now remove from `handleAction` the following:

```
  -- ABOVE CODE REDACTED

      -- Finalize -> killTickFiber ❶
    HandleDocClick _ event -> do
    -- BELOW CODE REDACTED
```

❶ We don't need to handle an Action that doesn't exist.

We don't need `tickFiber` in our State:

```
type State =
  { count :: Int
  -- , tickFiber' :: Maybe (Fiber Unit) ❶
  }

component
  :: ∀ m
  .  MonadAff m
  => H.Component Query Input Output m
component = H.mkComponent
  -- { initialState: \i -> { count: i, tickFiber': Nothing } ❷
  { initialState: \i -> { count: i }
  -- BELOW CODE REDACTED
```

❶ We no longer have to keep track of the `tickFiber`.

❷ And we don't need to initialize it.

And finally, let's clean up the compiler warnings for our imports:

```
import Data.Maybe (Maybe(..))
-- import Data.Maybe (Maybe(..), maybe) ❶
import Effect.Aff (forkAff, delay, killFiber, launchAff_)
-- import Effect.Aff (Fiber, forkAff, delay, killFiber, launchAff_) ❷
```

❶ Our code doesn't use `maybe` anymore.

❷ Our code doesn't use the Type `Fiber` anymore.

Your code should compile at this point.

Now, let's look back in detail at our new approach. First, we'll start with `Tick`:

```
    Tick sid -> do
      { count } <- H.get
      unless (count <= 0) $ H.modify_ _ { count = count - 1 }
      when (count == 1) do
        H.raise unit
        H.unsubscribe sid ❶
```

❶ The unsubscribe will kill our `tickFiber`.

We removed the `killTickFiber` call and now are relying on `H.unsubscribe` to do this for us. To understand how that works, we need to dig into the new `Initialize` code.

We're going to look at the code over and over again with a particular focus each time. Here we're looking at some debugging code I've added to prove to us that our code is working. This code can be removed once we've proven that it works:

```
    Initialize -> do
      -- timer setup
      { emitter:tickEmitter, listener:tickListener } <- H.liftEffect HS.create
      tickFiber <- H.liftAff $ forkAff $ forever do
        log "Sleeping..." ❶ ❷
        delay $ Milliseconds 1000.0
        H.liftEffect $ HS.notify tickListener Tick
      let fiberEmitter = HS.makeEmitter \_ -> pure $ launchAff_ do
            log "Killing tick fiber..." ❶ ❸
            killFiber (error "Event source finalized") tickFiber
      HQ.subscribe' \sid -> (tickEmitter <|> fiberEmitter) <#> (_ $ sid)
```

❶ These 2 lines of code were added to let us know that our solution works.

❷ Each time our `tickFiber` sleeps, we log. This will let us know when our `Fiber` is running and more importantly, when it is NOT running anymore.

❸ This is logged when our `Fiber` is killed. More on how this works in a minute.

Now, let's look at what we're doing differently from our previous implementation. There are 2 things we're doing differently here. First, we've created another `Emitter`, which will be used to kill the `Fiber`.

And second, we're using the Primed version of `H.subscribe` since that variant takes a Function that gets the `SubscriptionId` and returns the `Emitter Action`.

```
subscribe :: ∀ state action slots output m. HS.Emitter action -> HalogenM
state action slots output m SubscriptionId

subscribe' :: ∀ state action slots output m. (SubscriptionId -> HS.Emitter
action) -> HalogenM state action slots output m Unit
```

The non-primed version returns the `SubscriptionId`, but since we have to create the `tickFiber` BEFORE we subscribe, and our `Fiber` needs to pass the `SubscriptionId` to the `Tick` Data Constructor so the `Tick` handler can unsubscribe, we have a chicken-and-egg problem, if we use the non-primed version of `H.subscribe`:

```
    Initialize -> do
      -- timer setup
      { emitter:tickEmitter, listener:tickListener } <- H.liftEffect HS.create
      tickFiber <- H.liftAff $ forkAff $ forever do
        log "Sleeping..."
        delay $ Milliseconds 1000.0
        H.liftEffect $ HS.notify tickListener Tick ❶
      let fiberEmitter = HS.makeEmitter \_ -> pure $ launchAff_ do ❷ ❸
            log "Killing tick fiber..."
            killFiber (error "Event source finalized") tickFiber
      HQ.subscribe' \sid -> (tickEmitter <|> fiberEmitter) <#> (_ $ sid) <4>
❺ ❻
```

❶ `Tick` expects a `SubscriptionId` as it's first Parameter. And since we don't have it yet, since we haven't subscribed yet, we'll have to get from `H.subscribe'`, i.e. the Primed version.

❷ We are making an `Emitter` for our `Fiber`, but ONLY so we can control the clean up upon an unsubscribe. We use `makeEmitter` to do this.

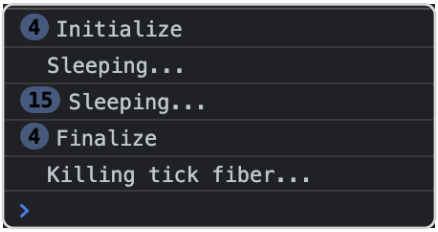
❸ We don't care about the callback that's passed as the First Parameter to the Function that's passed to `makeEmitter` since it's used for notifications. We already have the `tickEmitter/tickListener` pair for that. We just return the clean up code to be executed upon an unsubscribe. Since we only have a single subscriber, we don't have to keep track of the subscribers like `create` does. We just kill the `tickFiber`.

❹ We are using the Primed version os `subscribe` to get the `SubscriptionId` that we're going to pass to our Action, i.e. `Tick`, using `map`.

❺ We use `<|>` to combine `tickEmitter` and `fiberEmitter` into a single `Emitter`. When the Halogen Engine finalizes our component, it'll `unsubscribe` from our combined `Emitter` triggering the clean up code for BOTH to be run.

❻ An `Emitter` is a `Functor`, so we can `map` over it. Our `Emitter` emits Actions that are short a single Parameter, viz. the `SubscriptionId`, hence its Type is `Emitter (H.SubscriptionId -> Action)`. We're mapping a Function over the `Emitter` which will apply `sid` to the `Tick` Data Constructor creating an `Action`. This makes the final result an `Emitter Action`, which is exactly what `H.subscribe'` expects the Function to return.

Go ahead an rerun the app and let the `CountdownTimer` run down to `0`. You should see the following:



We get our `Sleeping...` message each time it sleeps, i.e. as long as the `tickFiber` is running. And once we

kill it, as witnessed by the `Killing tick fiber…` message, we no longer get `Sleeping…` messages.

This proves our approach is working and now we don't have to remember to clean up the timer `Fiber`.

The final part to be added to our program is an `Emitter` that's tied to a non-Halogen DOM Node to emit an `Action`.

Let's imagine that you have a Component that wants to know when you click on the `document`. We're going to use the `CountdownTimer` for this even though it really doesn't belong in that component. But that's fine since we're just creating a test application for learning purposes.

First thing to do is to create an `Emitter`. We'll start by adding it to the `Initialize` where we created the timer `Emitter`:

```
import Halogen.Query.Event as HQE ❶
import Web.HTML.HTMLDocument as HTMLDocument
import Web.HTML (window)
import Web.HTML.Window as Window
import Web.UIEvent.MouseEvent as ME
import Web.UIEvent.MouseEvent (MouseEvent)
import Web.UIEvent.MouseEvent.EventTypes (click)

  -- ABOVE CODE REDACTED
  where
  handleAction
    :: Action
    -> H.HalogenM State Action Slots Output m Unit
  handleAction = case _ of
    Initialize -> do
      -- timer setup ❷
      { emitter:tickEmitter, listener:tickListener } <- H.liftEffect HS.create
      tickSubscription <- HQ.subscribe tickEmitter
      tickFiber <- H.liftAff $ forkAff $ forever do
        delay $ Milliseconds 1000.0
        H.liftEffect $ HS.notify tickListener $ Tick tickSubscription
      H.modify_ _ { tickFiber' = Just tickFiber }
      -- document click setup ❷
      document <- H.liftEffect $ Window.document =<< window ❸
      HQ.subscribe' $ const $ HQE.eventListener ❹
        click ❺
        (HTMLDocument.toEventTarget document) ❻
        -- COMPILER ERROR!!
        (map (HandleDocClick document) <<< ME.fromEvent) ❼ ❽
-- BELOW CODE REDACTED
```

❶ Added these imports

❷ Added these comments to differentiate the code blocks.

❸ We get the `document` starting from the Browser `window`.

❹ We subscribe using `HQ.subscribe'` and `HQE.eventListener`. Note we're using the Prime version os `HS.subscribe` since we don't need to unsubscribe. We'll let Halogen do that when we're Finalized.

❺ This is the `MouseEvent` we're going to listen for.

❻ `toEventTarget` in `Web.HTML.HTMLDocument` will take an `HTMLDocument` and create an `EventTarget`.

❼ This is a callback Function when the `click` Event occurs. It takes an `Event` and returns a `Maybe Action`. Since `ME.fromEvent` returns a `Maybe`, we `map` over that `Maybe` with our `Action` Data Constructor.

❽ Our soon-to-be-created Data Constructor, `HandleDocClick`, takes 3 Parameters. First is the `HTMLDocument`, second is the `SubscriptionId` and third is the `MouseEvent`.

The compiler error lets us know that we need to add the Action definition for `HandleDocClick`. It'll take the `HTMLDocument`, `SubscriptionId` and the `MouseEvent`:

```
import Web.HTML.HTMLDocument (HTMLDocument) ❶

data Action
  = Initialize
  | Tick H.SubscriptionId
  | HandleDocClick HTMLDocument MouseEvent
```

❶ Added this import.

For now, we're just going to log to the console when the `document` is clicked on:

```
import Effect.Class.Console (log) ❶

    -- ABOVE CODE REDACTED
    Finalize -> killTickFiber
    HandleDocClick _ _ -> log "Document clicked on"
```

❶ Added this import.

Add this line of code to `handleAction`.

Now, rerun the app and refresh the browser. Now click around and see what gets logged in the Console.

And new build errors:

```
[info] Build succeeded.
[error] Some of your project files import modules from packages that are not
in the direct dependencies of your project.
To fix this error add the following packages to the list of dependencies in
your config:
- web-html
- web-uievents
```

Add these to `spago.dhall`.

Once again, rerun the app and refresh the browser. Now click around and see what gets logged in the Console.

Did you notice how everything you clicked on will display the `Document clicked on` message?

That's because Events in the DOM bubble up from the element that got clicked on all the way up the DOM tree until it finally reaches the `document`.

That's what `stopPropagation` is for. But it must be used at the Target of the click. That means that if we want to use `stopPropagation`, we'll have to add it to everything on our page.

A better solution is to get the actual Target Element of the Event, i.e. whatever Element the user clicked on. Then compare it with `document` to see if it was the thing that was actually clicked on and then and only then will we consider the `document` clicked on.

Dealing with the DOM is never easy. So, we'll take this one step at a time.

First, we want to take the `event` we get and turn it into a DOM `Node`. Then we'll use `isEqualNode` to compare it with the `Node` from the `document`. We have to do this since there's no way to compare `Elements`.

Here's our plan:

```
MouseEvent --> Event --> EventTarget --> Element --> Node
```

We'll start with a `MouseEvent` and turn it into an `Event` and then to an `EventTarget` and so on until we have the `Node`.

We're going to incubate one line of code, modifying it one step at a time until we have a `Node` following the above plan:

```
import Web.Event.Event as WebEvent
import Web.DOM.Element as DomElement
import Web.UIEvent.MouseEvent as ME

event :: MouseEvent

ME.toEvent event :: Event

WebEvent.target $ ME.toEvent event :: Maybe EventTarget

DomElement.fromEventTarget
  =<< (WebEvent.target $ ME.toEvent event) :: Maybe Element

DomElement.toNode <$>
  (DomElement.fromEventTarget
    =<< (WebEvent.target $ ME.toEvent event)) :: Maybe Node ❶
```

❶ We're going to use this line of code.

Okay, so we got to a `Maybe Node` instead of `Node`. We're going to have to deal with that later.

Next, is to do the same with the `document`. Here's our plan:

```
HTMLDocument --> Document --> Element --> Node
```

Once again, we'll incubate one line of code, modifying it one step at a time until we have a `Node`. We will follow the above plan:

```
import Web.DOM.Document as DomDocument
import Web.HTML.HTMLDocument as HTMLDocument

document :: HTMLDocument

HTMLDocument.toDocument document :: Document

DomDocument.documentElement $ HTMLDocument.toDocument document
  :: Effect (Maybe Element)

do
  docElem' <- DomDocument.documentElement
    $ HTMLDocument.toDocument document
      :: Effect (Maybe Element)

do
  docNode' <- map DomElement.toNode <$>
    (DomDocument.documentElement $ HTMLDocument.toDocument document)
      :: Effect (Maybe Node) ❶
```

❶ We're going to use this line of code. We'll have to lift it with `H.liftEffect`.

Here's the code change:

```
import Web.DOM.Document as DomDocument ❶
import Web.DOM.Element as DomElement
import Web.DOM.Node as DomNode
import Web.Event.Event as WebEvent

    -- ABOVE CODE REDACTED
    HandleDocClick document event -> H.liftEffect do ❷ ❸
      docNode' <- map DomElement.toNode <$>
        (DomDocument.documentElement
          $ HTMLDocument.toDocument document) ❹
      let clickNode' = DomElement.toNode <$>
            (DomElement.fromEventTarget
              =<< (WebEvent.target $ ME.toEvent event)) ❺
      case docNode', clickNode' of ❻
        Just dn, Just cn -> do ❼
          eq <- DomNode.isEqualNode dn cn ❽
          when eq $ log "Document clicked on" ❾
        _, _ -> pure unit ❿
```

❶ Added these imports.

❷ We now give `HandleDocClick's` Parameters names.

❸ We lift all the `Effect` calls in the `do` block.

❹ The line we incubated above to get the `Node` for the `document`.

❺ The line we incubated above to get the `Node` for the `Element` we clicked on.

❻ Using two Values separated by commas in the `case` expression.

❼ Pattern match for both to be a `Just`. Notice the Pattern Matches are also separated by commas.

❽ Compare `Nodes` for equality.

❾ Only log when what we clicked on was the `document`, i.e. the clicked on `Node` is equal to the `Node` that is the `document`.

❿ The catch-all case must have the same number of don't-care Variables separated by commas.

Now, rerun the app and…. Oh, no. More missing dependencies:

```
[info] Build succeeded.
[error] Some of your project files import modules from packages that are not
in the direct dependencies of your project.
To fix this error add the following packages to the list of dependencies in
your config:
- web-dom
- web-events
```

Update your `spago.dhall`.

Now, rerun the app and refresh the Browser and click around the page. You should notice that when you click BELOW all of the controls, `Document clicked on` gets logged to the console (or in Chrome, the number gets incremented next to text).

You should also notice that clicking to the RIGHT of the `Counters` or the `CountdownTimer` Components does NOT log that the document was clicked.

That's because those Components have `divs` that span the width of the page. If we wanted to change this behavior, we could consider clicking on a `div` the same as clicking on the `document`. This clearly would take a bit more logic.

We could also consider clicking on `buttons` as having NOT clicked on the `document` and everything else is like clicking on the `document`.

Comment out your current `HandleDocClick` code and replace it with this:

```
import Data.Array (elem) ❶
-- import Web.DOM.Document as DomDocument ❷

    -- ABOVE CODE REDACTED
    HandleDocClick _ event -> do ❸
      let clickNode' = DomElement.toNode <$>
            (DomElement.fromEventTarget
              =<< (WebEvent.target $ ME.toEvent event)) ❹
      case clickNode' of
        Just node ->
          unless (DomNode.nodeName node `elem` [ "BUTTON" ]) ❺
            $ H.liftEffect $ log "Document clicked on"
        _ -> pure unit
```

❶ Added this import.

❷ Comment out this import.

❸ We don't need the `document` Parameter for this version of `HandleDocClick`.

❹ Same code to get the clicked `Node` as we had before.

❺ `unless` the `Node` name is an element of the `Array` then we log `Document clicked on`.

Rerun the app and refresh the Browser and see how it works. Then return here.

You should have seen that clicking on anything OTHER than a `BUTTON` Node will display `Document clicked on`.

Now, let's say you want to know when you've click on a very specific `Node`. In that case, you can use ***Reference Labels***.

These are created with `H.refLabel`. Using `HP.ref` will make a Property from the label that can be associated with an Element.

Make the following changes to your code:

```
import Halogen.HTML.Properties as HP ❶

  -- ABOVE CODE REDACTED
  theCountdownDiv :: H.RefLabel
  theCountdownDiv = H.RefLabel "the-countdown-div" ❷

  render :: State -> H.ComponentHTML Action Slots m
  render { count } =
    HH.div
      [ HP.ref theCountdownDiv ❸
      , HC.style do
          display flex
          flexDirection column
          padding (rem 1.0) (rem 1.0) (rem 1.0) (rem 1.0)
      ]
    [ HH.text $ "COUNTDOWN: " <> show count ]
```

❶ Added this import.

❷ Create a Reference Label for the Countdown `div`.

❸ Add a property to the `div` with the Label.

Now, we can use the Reference Label with `H.getHTMLElementRef` to get the `Element` for the `div` that contains the `CountdownTimer` Component. We can then take that `Element` and convert it to a `Node` to be able to compare it against the clicked `Node`:

Comment out your current `HandleDocClick` code and replace it with this:

```
-- import Data.Array (elem) ❶
import Web.HTML.HTMLElement as HTMLElement ❷

    -- ABOVE CODE REDACTED
    HandleDocClick _ event -> do ❸
      let clickNode' = DomElement.toNode <$>
            (DomElement.fromEventTarget
              =<< (WebEvent.target $ ME.toEvent event)) ❹
      divNode' <- map HTMLElement.toNode
        <$> H.getHTMLElementRef theCountdownDiv ❺
      case clickNode', divNode' of ❻
        Just cn, Just dn -> H.liftEffect do
          eq <- DomNode.isEqualNode cn dn
          when eq $ log "The CountdownTimer DIV clicked on" ❼
        _, _ -> pure unit
```

❶ Added this import.

❷ Comment out this import.

❸ We don't need the `document` Parameter for this version of `HandleDocClick`.

❹ Same code as before to get the clicked `Node`.

❺ Look up `theCountdownDiv` Label and then convert the `Element` to a `Node`.

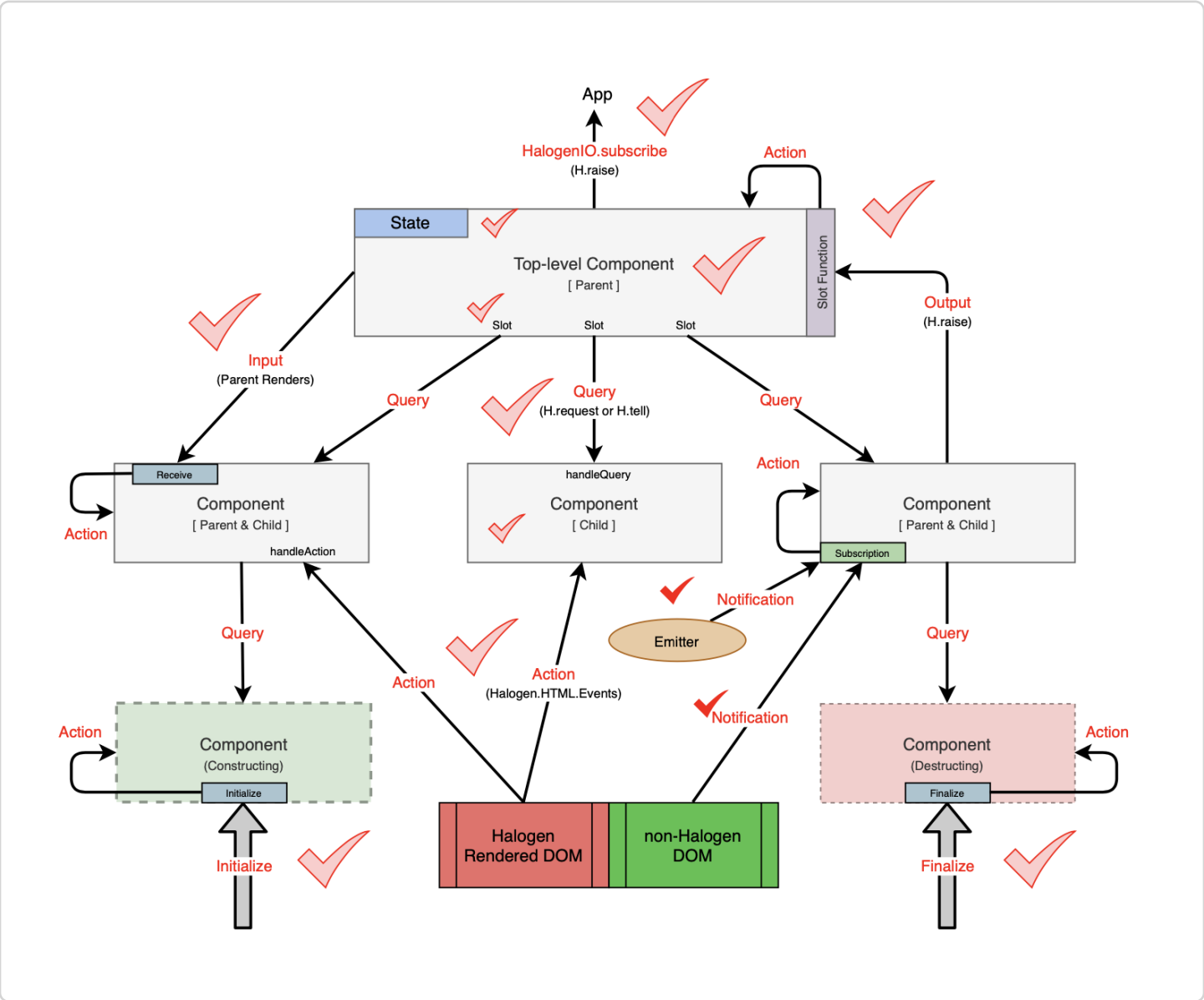
❻ This code is very similar to an earlier example to compare `Nodes`.

❼ We log a different message since we're targeting clicking on a specific `div`.

Rerun the app, refresh the Browser and then click around and see how this works. Then return here.

You should see that `The CountdownTimer DIV clicked on` is logged every time you click on anything in the `div` for the `CountdownTimer`.

Let's look at diagram and see what we've accomplished:



In this section, we've accomplished the following:

- Create an EventSource
- Subscribe to non-Halogen DOM `EventSource`, e.g. a `click` Event on the `document`

At this point, we've accomplished everything in this diagram, which means that we now have a great reference Halogen Application.

This will be a great reference when we build any Halogen Application, e.g. when we build a Front-end Application for the Backend Server that we built earlier.

Chapter 30. Writing a Front End using Halogen

In the previous Chapter, we explored all of the major parts of Halogen when building UIs. But there's more than just the UI when it comes to building a real ***Single Page Application*** (SPA).

We need to handle ***Hash Routing***, i.e. route on the client where the Hash portion of the URL, i.e. the portion after the `#`, determines which page to display and, optionally, what content.

By the way, the technical term for everything past the `#` is ***URI Fragment***.

And we'll need to design our ***Application Monad***, i.e. the Monad Stack that our Application runs in. In the last Chapter, we always ran our Halogen UIs using `runUI`, which evaluates our Components in `Aff`. For our Application, we're going to want to run using our own Stack.

Before we delve into these 2 areas, let's first add a sibling directory to our `server`.

If your code is kept in a folder called `/dev`, and you followed the directions exactly in Chapter 4 then you'd run:

```
cd /dev/purescript
```

If this is not where you put your project code, change to the appropriate directory now.

Now create the client:

```
cd fullstack
mkdir client
cd client
```

Before we go further ***make sure that you're in the `client` directory***.

To initialize your project so that we can install `npm` packages and use `npx` to run the compiler and build tool from inside your project, run:

```
npm init -y
```

This will create a file call `package.json`. This will keep track of any `npm` packages we install.

To locally install the PureScript compiler, `purescript` and the build tool, `spago`, run:

```
npm install --save-dev spago@0.20.0 purescript@0.14.1
```

Notice the `--save-dev` option. This will tell `npm` that these packages are only for development.

To verify that we have installed the compiler locally, we'll use `npx` (NOTICE the `x`):

```
npx purs --version # should print out 0.14.1
```

`npx` looks in our local `node_modules` directory for `purs`, the name of the PureScript compiler, and then runs it passing all other command line Parameters to it. The parameter `--version` tells `purs` to print out it's version.

To verify that we have installed the build tool locally, run:

```
npx spago version # should print out 0.20.0
```

Now, prepare the project for version control using `git`, run:

```
git init # should print `Initialized empty Git repository in <dir>`
```

The final step is to initialize this project for building a PureScript program using `spago`:

```
npx spago init
```

Build the starter app with:

```
npx spago build
```

You will get a bunch of warnings like:

```
Warning 1 of 4:

  in module Data.Ord.Generic
  at .spago/prelude/v5.0.0/src/Data/Ord/Generic.purs:22:3 - 22:41 (line 22,
column 3 - line 22, column 41)

    Name b1 was introduced but not used.

  in value declaration genericCompare'

  See
https://github.com/purescript/documentation/blob/master/errors/UnusedName.md
for more information,
  or to contribute content related to this warning.
```

This is because a bunch of libraries haven't caught up to version 14 of the compiler. Version 14 of the PureScript compiler gives you warnings now for unused variables. This change was made and not all libraries have quite caught up yet.

There are also a bunch of deprecated features from version 13 that will give you warnings. They will be permanently removed in version 15, but for now we get warnings. These can be ignored when they come from libraries. The library owners will take care of fixing them in due time.

We only have to worry about these warnings in our own code. But for now all the warnings are for libraries, since we haven't written any code just yet.

Now, if you're using VSCode, you should be able type the following command to open up this directory in the editor:

```
code .
```

If this doesn't work or you're using a different editor, then simply open this directory manually from your editor.

30.1. Halogen CSS Library Setup

Unfortunately, version 14 of the compiler is a transitionary version. Not all libraries have kept up. It may be that by the time you read this, that last statement will no longer be true.

But as of the time of this writing, it's imperative that you use a version of `halogen-css` that is NOT part of the standard `package-set`. If you are explicitly specifying version when installing `spago` and `purescript`, then this will always be the case.

To use the correct version of `halogen-css`, you'll need to edit the very BOTTOM of your `packages.dhall` file to look like:

```
let upstream =
      https://github.com/purescript/package-sets/releases/download/psc-0.14.1-
20210506/packages.dhall

sha256:d199e142515f9cc15838d8e6d724a98cd0ca776ceb426b7b36e841311643e3ef ❶

let additions = ❷
  { halogen-css =
    { dependencies =
       [ "css"
       , "halogen"
       ]
    , repo =
       "https://github.com/purescript-halogen/purescript-halogen-css"
    , version =
        "v8.0.0"
    }
  }

in  upstream // additions ❸
```

❶ This line will already be there.

❷ You'll need to add this block to use `v8.0.0` of the `halogen-css` library.

❸ You'll need to edit this line to match.

30.2. Hash Routing

There are 2 libraries that we're going to use for Routing, `purescript-routing` and `purescript-routing-duplex`.

We'll use `purescript-routing` to handle changes in the URL `Hash` and we'll use `purescript-routing-duplex` for Route Parsing and Printing.

Create a Folder under `src` called `Data` and then under that Folder create a file called `Route.purs`. We're going to put our Route Codec in here.

Next add the following:

```
module Data.Route where

import Prelude hiding ((/)) ❶
```

❶ We're hiding the Divide Operator since `Routing.Duplex.Generic.Syntax` has a similar operator for defining URL Paths that we're going to want to use.

Next, go to `Pursuit` and look up `RouteDuplex` and read the description and note the one and only Data Constructor then return here.

Let's look carefully at `RouteDuplex` definition, in particular, its Data Constructor:

```
data RouteDuplex i o = RouteDuplex (i -> RoutePrinter) (RouteParser o)
```

Its first Parameter is a Function that takes an `i` and produces a `RoutePrinter`, whatever that is. And the second Parameter is a parser for `o`.

In the docs, it says that we'll want to use `RouteDuplex'` since the inputs and the outputs are the same Type with that definition:

```
type RouteDuplex' a = RouteDuplex a a ❶
```

❶ Input and Outputs are the same Type.

The `Duplex` part of the name of this Type represents the fact that we can parse a URL from a `String` to some output Type, the second `a`, and then take that output Type, the first `a`, as an input and create a `RoutePrinter`, which can produce a `String`.

This library will give us the ability to define a Type for Routing and a Duplex Parser/Printer for our `Route` Type.

Let's return to `RouteDuplex's` Data Constructor:

```
RouteDuplex (i -> RoutePrinter) (RouteParser o)
```

At this point, we're pretty familiar with parsers. But what is exactly is a `RoutePrinter`. Return to `Pursuit` and click on the `RoutePrinter` to investigate. Then keep reading.

When I do this I find that `RoutePrinter` contains a Function from `RouteState` to `RouteState`. Now click on `RouteState` in `Pursuit` and see what you find. Then continue reading.

This is what I found looking directly at the source code:

```
type RouteState =
  { segments :: Array String
  , params :: RouteParams
  , hash :: String
  }
```

`RouteState` seems to contain the parts of a parsed URL. Can you figure out what these different parts are?

I cannot. To figure this out, return to `RoutePrinter` on `Pursuit` to see if you can find a Function that will help us understand this better. Look for a Function that takes or returns a `RouteState`.

Once you find something that you think will work, you'll have to look at code for that Function since none of these Functions are documented. See if you can figure out what `segments`, `params` and `hash` represent before reading on.

When I do this, I find `printPath`:

```
printPath :: RouteState -> String
printPath { segments, params, hash: hash' } =
  printSegments segments <> printParams params <> printHash hash'
  where
  printSegments = case _ of
    [""] -> "/"
    xs -> joinWith "/" $ map unsafeEncodeURIComponent xs ❶

  printParams [] = ""
  printParams ps = "?" <> joinWith "&" (uncurry printParam <$> ps) ❷

  printParam key ""  = unsafeEncodeURIComponent key
  printParam key val = unsafeEncodeURIComponent key <> "="
    <> unsafeEncodeURIComponent val

  printHash "" = "" ❸
  printHash h  = "#" <> h
```

❶ Prints `segments`.

❷ Prints `params`.

❸ Prints `hash`.

From this, we can see that `segments` are printed with `/` in between them. The `segments` must make up a path of a URL, e.g. `/logon`.

Next, we can see that `params` are printed with a preceeding `?` and `&` in between. We can conclude that

`params` are Query Parameters in the URL.

And finally, the `hash` is printed with a preceeding `#`, which lets us know that its the URI Fragment or what's commonly referred to as the Hash.

So, I suspect that `RouteParser` takes a `String` and produces a `RouteState`. Doing a little investigation work, I find `runRouteParser` in `Routing.Duplex.Parser`:

```
runRouteParser :: ∀ a. RouteState -> RouteParser a -> RouteResult a
```

We can see that it returns `RouteResult`, which is:

```
data RouteResult a
  = Fail RouteError
  | Success RouteState a
```

This contains a `RouteState` on the success case.

So it looks like `RouteParser` will parse the segments of the URL Path, the Query Parameters and the URI Fragment (Hash).

But, the nice thing about URI Fragments, is that they too can have path segments, since the `/` character is allowed in a Fragment. It can also have the equivalent of Query Strings or Query Parameters since `=`, `&` and `?` are also allowed in Fragments.

So, if we use `Route.Duplex` for just parsing URI Fragments, then we'll be able to parse both `segments` and `params`. We cannot parse `hash` from a URI Fragment since `#` is not allowed in a URI Fragment.

30.2.1. Defining Routes

Before we define our Routes, let's install `purescript-routing-duplex` and `purescript-routing` by running:

```
npx spago install routing-duplex routing
npx spago build
```

You will get the following warning:

```
[warn] None of your project files import modules from some projects that are
in the direct dependencies of your project.
These dependencies are unused. To fix this warning, remove the following
packages from the list of dependencies in your config:
- routing
- routing-duplex
```

That's because we aren't using the library anywhere in our code. Not yet that is.

Now, let's define a Route Type that corresponds with our backend server API. But let's start simply by just including logon and logoff:

```
data Route
  = Logon
  | Logoff
```

Next, we'll need to make our `Route` Type have a `Generic` Instance so `Route.Duplex` can use it:

```
import Data.Generic.Rep (class Generic) ❶

data Route
  = Logon
  | Logoff
derive instance genericRoute :: Generic Route _ ❷
```

❶ Added this import.

❷ Derive `Generic`.

Now, we'll build our Codec, i.e. Coder-Decoder. Here's it is in full:

```
import Routing.Duplex (RouteDuplex', root, path) ❶
import Routing.Duplex.Generic (sum, noArgs)

routeCodec :: RouteDuplex' Route
routeCodec = root $ sum ❷ ❸
  { "Logon": path "logon" noArgs ❹ ❺ ❻
  , "Logoff": path "logoff" noArgs
  }
```

❶ Added these imports.

❷ `root` represents the root of the URL path, i.e. `/`.

❸ `sum` is to represent Sum Types. We're using it here since our `Route` Type is a Sum Type.

❹ Each field name `String` must match exactly the name of the Data Constructor, hence the use of double quotes to use uppercase characters in the Record field names.

❺ `path` defines a Path of a particular Value, e.g. `logon`.

❻ `noArgs` is when the particular route has no Parameters.

The Record passed to `sum` defines routes that have field names that must match EXACTLY the Data Constructors in the `a` of `RouterDuplex' a`. In our case, the field names match the Data Constructors of `Route`.

There is also a `product` Function that can be used for Product Types.

So far, we've defined the following valid Path Segments:

```
/logon
/logoff
```

These are Static Segments. We'll see Variable Segments in a moment.

Next, install the `purescript-uuid` package with:

```
npx spago install uuid
npx spago build
```

Once again, we can ignore the following warning:

```
[warn] None of your project files import modules from some projects that are
in the direct dependencies of your project.
These dependencies are unused. To fix this warning, remove the following
packages from the list of dependencies in your config:
- routing
- uuid
```

Next, create a File under `src/Data` called `UserId.purs`. Add the following code:

```
module Data.UserId where

import Data.UUID (UUID)

newtype UserId = UserId UUID
```

This is a `UserId` that we'll use in a URL when we want to view User information.

Now, let's add a route for the users page. To do this, we'll first need to update our `Route` Type:

```
import Data.Maybe (Maybe) ❶
import Data.UserId (UserId(..))

data Route
  = Logon
  | Logoff
  | Users (Maybe UserId) ❷
derive instance genericRoute :: Generic Route _
```

❶ Added these imports.

❷ We've added the `UserId` as a `Maybe` so we can see all users when it's missing.

Notice the compiler error on sum when we make this change:

```
Could not match type

  ( "Users" :: RouteDuplex t2 t2
  ...
  | t3
  )

with type

  ( "Logoff" :: RouteDuplex NoArguments NoArguments
  , "Logon" :: RouteDuplex NoArguments NoArguments
  ...
  , "Logoff": path "logoff" noArgs
  }
```

This is because our definition is made Type safe by `Route.Duplex` library's use of `Generics`. The compiler will catch when we're missing a field.

Notice here that it's complaining about a Row with `Users` in it not matching the Row with only `Logon` and `Logoff` in it.

The compiler will, however, NOT complain if we have an EXTRA field in the Record we pass to `sum`.

We'll need to add a route definition to our Record. But before we do we should look at how we can define a Path Segment using `segment`:

```
-- Allows you to capture a variable segment. For example, the path
-- /user/5 has one static segment and one variable segment. This always
-- reads and writes a string, but when used with other combinators, it
-- can transform that string into a type of your choice. For example:
-- `path "user" (int segment)` parses to `User Int`.
segment :: RouteDuplex' String
```

Notice how `segment` defines Variable Segments. This can be seen in the documented example of `/user/5`. `user` is a Static Path Segment, whereas, `5` is a Variable Path Segment.

We can use the following modifiers to change the `String` Type that `segment` gives us to other Types:

```
int :: RouteDuplex' String -> RouteDuplex' Int
boolean :: RouteDuplex' String -> RouteDuplex' Boolean
string :: RouteDuplex' String -> RouteDuplex' String ❶
```

❶ Just identity.

While these combinators convert to standard Types, we'll need to convert our Path Segment into a `UserId`. This can be accomplished with another more powerful combinator:

```
as
  :: ∀ s a b
  .  (a -> s) ❶
  -> (String -> Either String b) ❷
  -> RouteDuplex s String
  -> RouteDuplex a b
```

❶ This is the custom Printer.

❷ This is the custom Parser.

`as` is written here in the more general form where the input and output Types could be different. But they're not for us, so here's what it looks like in that case:

```
as
  :: ∀ a
  .  (a -> String)
  -> (String -> Either String a)
  -> RouteDuplex' String
  -> RouteDuplex' a
```

We'll need to write a Function to convert a `RouteDuplex' String` to `RouteDuplex' UserId`. Write just the Type Signature of a Function called `userId` that does this before reading on.

Here's just that portion of `userId`:

```
userId :: RouteDuplex' String -> RouteDuplex' UserId
```

Next, finish the code leveraging `as` to provide a custom printer and parser. Then return here.

We're going to call `as` with yet undetermined `printer` and `parser`:

```
import Routing.Duplex (RouteDuplex', root, path, as) ❶

userId :: RouteDuplex' String -> RouteDuplex' UserId
userId = as printer parser where ❷
  printer :: UserId -> String ❸
  printer = ???
  parser :: String -> Either String UserId ❸
  parser = ???
```

❶ Added `as` to the import list.

❷ Written Point-free where the `RouteDuplex' String` is unnamed.

❸ These Type Signatures are here to help us write these Functions. We'll remove them at the end.

If you didn't finish, but think that you can now, stop reading here and give a try.

The next part is the code for the custom `printer`:

```
import Data.UUID as UUID ❶

userId :: RouteDuplex' String -> RouteDuplex' UserId
userId = as printer parser where
  printer :: UserId -> String
  printer (UserId uuid) = UUID.toString uuid ❷
  parser :: String -> Either String UserId
  parser = ???
```

❶ Added this import.

❷ Destructure `UserId` to get the underlying `UUID` then convert it to a `String`. Note, we're not using `show` here since it includes the `UUID` prefix, e.g. `(UUID dc64cf80-f663-436a-a290-3cac4acf279f)`.

And finally, we'll write the custom `parser`:

```
import Data.Either (Either, note) ❶

userId :: RouteDuplex' String -> RouteDuplex' UserId
userId = as printer parser where ❷
  printer :: UserId -> String
  printer (UserId uuid) = UUID.toString uuid
  parser :: String -> Either String UserId
  parser  = UUID.parseUUID
    >>> map UserId
    >>> note "Invalid UserId"
```

❶ Added this import.

This uses `parseUUID` to parse a `String` into a UUID. This can fail and returns a `Maybe UUID` which is why we map over the `Maybe UUID` with the Data Constructor `UserId`.

We have to take that resulting `Maybe UserId` and apply `note` to convert it into the `Either` that `as` expects its parser to return.

We can remove the Type Signatures (you can also leave them if you like):

```
import Data.Either (note) ❶

userId :: RouteDuplex' String -> RouteDuplex' UserId
userId = as printer parser where
  printer (UserId uuid) = UUID.toString uuid
  parser  = UUID.parseUUID
    >>> map UserId
    >>> note "Invalid UserId"
```

❶ Removed `Either` from import list since it's no longer needed since we removed the Type Signatures.

Next, we'll use `userId` to define the `Users` Route:

```
import Routing.Duplex (RouteDuplex', root, path, as, optional, segment) ❶
import Routing.Duplex.Generic.Syntax ((/)) ❷

routeCodec :: RouteDuplex' Route
routeCodec = root $ sum
  { "Logon": path "logon" noArgs
  , "Logoff": path "logoff" noArgs
  , "Users": "users" / (optional $ userId segment) ❸ ❹
  }
```

❶ Added `optional` and `segment` to the import list.

❷ Added this import.

❸ Using `/` to separate Path Segments.

❹ Using `optional` combinator since the second path segment is optional.

Now, that you've seen how to write a customer Codec for your own Types, I have some bad news. We don't have a `UserId` for our users. We only have a unique `userName`. That means that we cannot use the code we just wrote.

If this was a real Application, then this is how you would probably do it. But since this is just a learning Application, not having a unique `UserId` is fine.

So, we're going to have to make the following change to our Route Type:

```
data Route
  = Logon
  | Logoff
  | Users (Maybe String) ❶
-- | Users (Maybe UserId)
```

❶ Changed `UserId` to `String`.

Then change our Codec as follows:

```
routeCodec :: RouteDuplex' Route
routeCodec = root $ sum
  { "Logon": path "logon" noArgs
  , "Logoff": path "logoff" noArgs
  , "Users": "users" / optional segment ❶
  -- , "Users": "users" / (optional $ userId segment)
  }
```

❶ `segment` parses and prints a `String`.

This Router Codec looks complete enough for us to continue building the rest of our Application.

30.3. Application Monad

When we wrote our Halogen Application in the previous Chapter, we ran it in `Aff`. But normally, we're going to want to run our components in a Monad Stack.

And this Application is no different. We're going to need an Application Monad that we can run our Component Computations in.

There are a few things that we must do if we're going to run in our custom Application Monad, which we'll call `AppM`.

First, we need to decide what our Monad Stack, if any, will look like. Based on experience, I can safely say that we'll need at least a `ReaderT`. The reason for this is because we need some Global State to keep track of the logged-on user.

So, let's capture our decisions by first creating a File `AppM.purs` under `src` that has the following:

```
module AppM where

import Control.Monad.Reader.Trans (ReaderT)
import Effect.Aff (Aff)
import Env (Env) -- COMPILER ERROR!!

newtype AppM a = AppM (ReaderT Env Aff a) ❶
```

❶ Our Application Stack is a `ReaderT` with `Aff` at the bottom, i.e. this eventually runs in `Aff`. `Env` is our reader's read-only Type.

You will get a compiler error on the import of `Env` since we haven't created it yet. We'll do that now.

Create a File called `Env.purs` and put it under `src` as well. Add the following code to it:

```
module Env where

import Effect.Ref (Ref)
import Data.LoggedOnUser (LoggedOnUser) -- COMPILER ERROR!!
import Data.Maybe (Maybe)

type Env =
  { userRef :: Ref (Maybe LoggedOnUser) ❶
  }
```

❶ The `Ref` contains a `Maybe` since there are times where the user is NOT logged on.

You'll get an error on importing `Data.LoggedOnUser` since we haven't created it yet. We'll do that soon enough, but in the meantime, let's look at the details of `Env`.

We're using something we haven't seen before, `Ref`, short for Reference, which is a Globally Mutable Variable. This differs from `AVar` in a couple of ways.

First, unlike `AVar`, it can never be empty and therefore never blocks a `Fiber`.

Second, it runs in `Effect` instead of `Aff`, which means that we'll have to lift calls to its API using

`liftEffect` since our Halogen components will be running in `MonadAff`, i.e. some Monad that ultimately runs in `Aff`.

Third, its NOT thread-safe, or more accurately, it's not fiber-safe. That means 2 `Fibers` can both modify the `Ref`. There is no `take` and `put` like with `AVars`. This is not going to be much of a problem for us since our Logon Page will write a `Just` Value to it and our Logoff will write a `Nothing` to it.

All other pages will ONLY read it. So, we don't need the kind of protection we'd normally get with `AVars`

Now, we should define the `LoggedOnUser`. Create a file under `src/Data` called `LoggedOnUser.purs` with the followng:

```
module Data.LoggedOnUser where

import Data.UUID (UUID)

type LoggedOnUser =
  { authToken :: UUID
  }
```

This captures the logged on user's `authToken`. We'll need this to make API calls to our `server`.

Now we should be able to compile all of our files with:

```
npx spago build
```

And we get:

```
[warn] None of your project files import modules from some projects that are
in the direct dependencies of your project.
These dependencies are unused. To fix this warning, remove the following
packages from the list of dependencies in your config:
- routing
[error] Some of your project files import modules from packages that are not
in the direct dependencies of your project.
To fix this error add the following packages to the list of dependencies in
your config:
- aff
- either
- maybe
- refs
- transformers
```

We're going to ignore the warning for now but we should add these new direct dependencies to our `spago.dhall` to address the error:

```
{-
Welcome to a Spago project!
You can edit this file as you like.
-}
{ name = "my-project"
, dependencies =
  [ "aff" ❶
  , "console"
  , "effect"
  , "either" ❶
  , "maybe" ❶
  , "prelude"
  , "psci-support"
  , "refs" ❶
  , "routing-duplex"
  , "routing"
  , "transformers" ❶
  , "uuid"
  ]
, packages = ./packages.dhall
, sources = [ "src/**/*.purs", "test/**/*.purs" ]
}
```

❶ These were added in these locations to keep the list in alphabetical order.

Now we can try to build again:

```
npx spago build
```

Now that we can build with a warning, we should turn our attention back to our Application Monad. Before advancing any further, use `derive newtype` to derive `Monad` for `AppM`. Don't forget that you'll need to derive a few Typeclasses before `Monad`.

Then return here.

Here are those changes:

```
import Prelude ❶

newtype AppM a = AppM (ReaderT Env Aff a)
derive newtype instance functorAppM :: Functor AppM
derive newtype instance applyAppM :: Apply AppM
derive newtype instance applicativeAppM :: Applicative AppM
derive newtype instance bindAppM :: Bind AppM
derive newtype instance monadAppM :: Monad AppM
```

❶ Import `Prelude` to get `Functor`, `Apply`, etc.

Now use `derive newtype` to get `MonadEffect` and `MonadAff`. These instances will let us call Functions in `Effect` using `liftEffect` and Functions in `Aff` by using `liftAff`.

Code this before reading further.

Here are those changes:

```
import Effect.Aff.Class (class MonadAff) ❶
import Effect.Class (class MonadEffect)

newtype AppM a = AppM (ReaderT Env Aff a)
derive newtype instance functorAppM :: Functor AppM
derive newtype instance applyAppM :: Apply AppM
derive newtype instance applicativeAppM :: Applicative AppM
derive newtype instance bindAppM :: Bind AppM
derive newtype instance monadAppM :: Monad AppM
derive newtype instance monadEffectAppM :: MonadEffect AppM ❷
derive newtype instance monadAffAppM :: MonadAff AppM ❷
```

❶ Added these imports.

❷ Added these lines.

Next, we need a way to run our `AppM` Stack. So write a Function called `runAppM` to run our Stack. Remember that `AppM` is just a `newtype` wrapper around a `ReaderT` Monad Stack. Return here once you're done.

First, we'll start with the Type Signature:

```
runAppM :: Env -> AppM ~> Aff ❶
```

❶ Notice the use of Natural Transformation Type Operator, `~>`. That saves us from specifying the `a`.

Next, we'll write the left-hand side of the implementation:

```
runAppM :: Env -> AppM ~> Aff
runAppM env (AppM reader) = ??? ❶
```

❶ We destructure the `AppM` to get to the contained `ReaderT`.

And now we can just run the underlying `ReaderT`:

```
import Control.Monad.Reader.Trans (ReaderT, runReaderT) ❶

runAppM :: Env -> AppM ~> Aff
runAppM env (AppM reader) = runReaderT reader env
```

❶ Added `runReaderT` to the import list.

Now I think about it, we're going to need a `MonadAsk` Instance so we can use `ask` in our Halogen Components. Go ahead and use `derive newtype` to implement `MonadAsk`. Don't forget that `MonadAsk` has 2 Type Parameters.

When you've coded this, return here.

Here's that change:

```
import Control.Monad.Reader.Class (class MonadAsk) ❶

newtype AppM a = AppM (ReaderT Env Aff a)
derive newtype instance functorAppM :: Functor AppM
derive newtype instance applyAppM :: Apply AppM
derive newtype instance applicativeAppM :: Applicative AppM
derive newtype instance bindAppM :: Bind AppM
derive newtype instance monadAppM :: Monad AppM
derive newtype instance monadEffectAppM :: MonadEffect AppM
derive newtype instance monadAffAppM :: MonadAff AppM
derive newtype instance monadAskAppM :: MonadAsk Env AppM ❷
```

❶ Added this import.

❷ Added this derivation.

That's concludes all of the derivations we'll need for our Application Monad. Next, we're going to have to explictly write our `instance` for all other functionality that `AppM` supports.

30.4. Capabilities

To help us write more generalized Halogen Components, we're going to factor out things like Error Logging

and Page Navigation. We don't want our Components tightly coupled to the implementation of our Application.

We'd like them to be as general as is reasonably possible.

Since our Logon Page will be a Component, we want it to be decoupled from the Application's navigational mechanics. This will allow us to put our Logon Component into a separate library if we want to.

We're also going to add Logging, another implementation detail we'd like to reserve for the Application Developer and not the Component Developer. In our case, we will be both, but we'll still make the separation to reduce coupling.

Both Navigation and Logging and other such functionalities are often referred to as ***Capabilities***. We can think of them as what our Application is capable of. These Capabilities will be implemented using Typeclasses and dictated as requirements by the Component using Constrants.

This will make far more sense once you see them in action. So, let's start with `Navigation`:

```
class Monad m <= Navigate m route where ❶
  navigate :: route -> m Unit ❷ ❸
```

❶ `Navigate` is a Typeclass of 2 Parameters, which means that all Methods must have both Parameters in their Type Signature.

❷ Given a `route`, which is an Application-specific `Route`, `navigate` will change the Hash in the current URL to cause a new client page to be displayed.

❸ Notice both Typeclass Type Parameters are in this Type Signature, i.e. `m` and `route`.

Our Logon Page cannot possibly know what Type `route` is going to be since it's Polymorphic. What we're going to need is a different interface that will allow the Logon Page to take the `mustChangePassword` it gets back from the `server` and use it to compute a `route`.

For that, we're going to create another Capability called `LogonRoute`:

```
class LogonRoute route where
  logonRoute :: UUID -> PasswordType -> route ❶
```

❶ The `UUID` is the `authToken` and the `PasswordType` says whether it's `PasswordPermanent` or `PasswordTemporary`, i.e. a Data Type we haven't defined yet, but will soon.

The Logon Page will first use `logonRoute` to compute the proper `route` and then use that `route` to navigate to the landing page for the user.

Let's stop here and add these Capabilites to our client. Create a Folder called `Capability` under `src`. Then add a File called `Navigate.purs` under `src/Capability`. Then add our `Navigate` Typeclass and the appropriate imports.

Once you've done that, create a File called `LogonRoute.purs` under `src/Capability` and add the Typeclass and the appropriate imports adding the `PasswordType` definition:

```
data PasswordType = PasswordPermanent | PasswordTemporary ❶
```

❶ Temporary passwords are used when the user forgets their password and requests a password reset.

Then return back here.

Here is `Navigate.purs`:

```
module Capability.Navigate where

import Prelude

class Monad m <= Navigate m route where
  navigate :: route -> m Unit
```

And here is `LogonRoute.purs`:

```
module Capability.LogonRoute where

import Data.UUID (UUID)

data PasswordType = PasswordPermanent | PasswordTemporary

class LogonRoute route where
  logonRoute :: UUID -> PasswordType -> route
```

One thing to keep in mind is that we're going to be calling `navigate` from inside the `HalogenM` Monad. That means that we're going to have to `lift` it every time we use it since `AppM` is the underlying Monad for the `HalogenM` Monad Stack.

We remove the need for lifting by making a `Navigate` instance for `HalogenM` which will do the lifting for us.

To do this, we'll have to look up all of the Type Parameters to `HalogenM`. Do this by looking up `HalogenM` on `Pursuit`, then return here.

Here's what I find:

```
HalogenM state action slots output m a
```

Before we implement this, we should install `purescript-halogen` and `purescript-halogen-css`:

```
npx spago install halogen ❶ ❷
npx spago build ❸
```

❶ Install both packages.

❷ Once `halogen-css` gets updated for version 14 of the compiler, we could install it here as well using: `npx spago install halogen halogen-css` instead.

❸ Build project so we won't get errors in the editor when we import Halogen modules.

Ignore the warnings for now.

Next, make sure you're working in `Capability.Navigate` and then write the code for the `Navigate` Instance for `HalogenM`.

Remember that `HalogenM state action slots output m` is the Monad, which `Navigate` calls `m`.

Once your done, continue reading.

First, we'll start coding the instance defintion:

```
import Halogen (HalogenM) ❶

instance navigateHalogenM
    :: Navigate m route ❷
    => Navigate (HalogenM state action slots output m) route where ❸
```

❶ Added this import.

❷ `HalogenM's` underlying Monad must support `Navigate`, hence, the Constraint. In our case, this will be `AppM`.

❸ Notice that we're not using `a`, i.e. the final Type Parameter to `HalogenM`. That's because everything that preceeds it is a Monad.

The Type Signature says that as long as the underlying Monad, in our case `AppM`, has a `Navigate` Instance, then `HalogenM` will.

Next, we'll add the `navigate` implementation by simply lifting the underlying Monad's `navigate`:

```
import Control.Monad.Trans.Class (lift) ❶

instance navigateHalogenM
    :: Navigate m route
    => Navigate (HalogenM state action slots output m) route where
  navigate = lift <<< navigate
```

❶ Added this import.

Next, we're going to create a `Log` Capability.

There's a lot of possible approaches we could take to do logging like we did in the server. But for this implementation, we're going to keep it simple and just write a simple JSON structure to the Console:

```
module Capability.Log where

import Data.DateTime (DateTime)

data LogLevel = Debug | Info | Warning | Error

type LogEntry =
  { level :: LogLevel
  , message :: String
  , timestamp :: DateTime
  }
```

Create a File called `Log.purs` under `src/Capability` and add the above definitions.

Next, write the code for the Typeclass called `Log`.

It'll have a Method called `logEntry` that will produce a `LogEntry`. Remember that we'll need to timestamp the `LogEntry`, so `logEntry` will run in some Monad. That should tell us that our `Log` Typeclass will have an `m` Type Parameter.

It will have another Method called `log` that will take a `LogEntry` and log it to the console. Keep in mind that `DateTime` has a `Show` Instance, so we can just use `show` on the `LogEntry` Record.

Once you've done that, return here and continue reading.

We'll start by writing the Typeclass definition:

```
import Prelude ❶

class Monad m <= Log m where
```

❶ Added this import to get `Monad` definition.

And next, the `logEntry` Type Signature:

```
class Monad m <= Log m where
  logEntry :: LogLevel -> String -> m LogEntry ❶ ❷
```

❶ The `timestamp` will be calculated in the implementation of this Method.

❷ Since getting the `timestamp` will involve running in `Effect`, this is a Monadic computation, hence `m LogEntry`.

And finally, we'll add the `log` Type Signature:

```
class Monad m <= Log m where
  logEntry :: LogLevel -> String -> m LogEntry
  log :: LogEntry -> m Unit ❶
```

❶ By design, the expectation for implementations of this Typeclass is that `log` will leverage the `log` that runs in `Aff`.

And now, add the `HalogenM` instance for this Typeclass like we did for `Navigate` so we won't have to `lift` calls to `log` or `logEntry` in `HalogenM` Functions, i.e. `handleAction` and `handleQuery`.

Then return here when that's complete.

Here's that implementation:

```
import Control.Monad.Trans.Class (lift) ❶
import Halogen (HalogenM)

instance logHalogenM
    :: Log m
    => Log (HalogenM state action slots output m) where
  logEntry level = lift <<< logEntry level ❷
  log = lift <<< log
```

❶ Added these imports.

❷ This is written as Point-free as is possible since these Functions take 2 Parameters.

These Capabilities seem sufficient for our Application at the moment. We may need to add more as we go along, but this is a good starting point.

30.5. Capability Implementations

We defined our Capabilities and now it's time to implement them. They will be implemented by the Monad that our Components run in. This is typically referred to as `m`, for example:

```
component
  :: ∀ m
   . MonadAff m
  => MonadAsk Env m
  => H.Component Query Input Output m
```

In our Application the `m` here is `AppM`, which means that we're going to have to make the following `AppM` Instances:

```
LogonRoute
Navigate
Log
```

We'll start with `LogonRoute`. Since our learning application is so limited, we don't really have too many places to go when a user logs on.

In a real application, we may decide that administrators route first to the admin section of the app, whereas, normal users would go to a dashboard or a some special landing page.

For us though, we're going to route everyone to `Users` with no `User` selected EXCEPT when the password needs to be changed. In that case, we'll route them to a dummy page where they COULD change their password.

Since our overly simplified server doesn't support password changes, we'll just have a button on the dummy page that when clicked will navigate the user to the Users page.

First, add `ChangePassword` as a `Route` to `Route.purs` and then write the instance for `LogonRoute` in `App.purs` to abide by the above specifications before reading further.

Here's the change to `Route`:

```
data Route
  = Logon
  | Logoff
  | Users (Maybe String)
  -- | Users (Maybe UserId)
  | ChangePassword ❶
```

❶ Added new Route Data Constructor for the Change Password Page.

And now, the compiler is telling us that something is wrong with `routeCodec`. It should be clear that we're missing the entry for `ChangePassword`:

```
routeCodec :: RouteDuplex' Route
routeCodec = root $ sum
  { "Logon": path "logon" noArgs
  , "Logoff": path "logoff" noArgs
  , "Users": "users" / (optional $ userId segment)
  , "ChangePassword": path "change-password" noArgs ❶
  }
```

❶ This `Route` entry is simply `/change-password` with no arguments.

Now, here's the instance implementation in `AppM`:

```
import Capability.LogonRoute (class LogonRoute, PasswordType(..)) ❶
import Data.Maybe (Maybe(..))
import Data.Route (Route)
import Data.Route as Route

-- COMPILER ERROR!!
instance logonRouteAppM :: LogonRoute Route where
  logonRoute _  PasswordPermanent = Route.Users Nothing ❷
  logonRoute _  PasswordTemporary = Route.ChangePassword ❸
```

❶ Added these imports.

❷ Users with permanent passwords will route to the Users Page.

❸ User with temporary passwords will route to the Change Password Page.

Now there are 2 issues with this. First, we have a compiler error. We'll fix that soon enough. But notice how we're not using the `UUID` Parameter.

Seems as if that was a bad initial design decision. Let's fix that by removing it from the Typeclass and our

Instance implementation.

Make those changes before reading further.

Here are those changes. First, the one in `Capability.LogonRoute`:

```
class LogonRoute route where
  logonRoute :: PasswordType -> route ❶
```

❶ Removed `UUID` Parameter, which warranted the removal of the import of `Data.UUID`.

Also, removed:

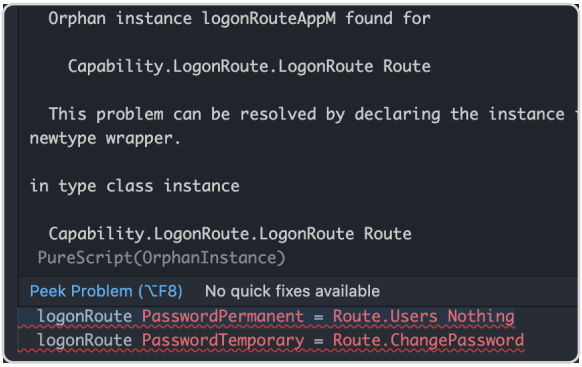
```
import Data.UUID (UUID)
```

And now, here's the change to our instance implementation:

```
-- COMPILER ERROR!!
instance logonRouteAppM :: LogonRoute Route where
  logonRoute PasswordPermanent = Route.Users Nothing ❶
  logonRoute PasswordTemporary = Route.ChangePassword ❶
```

❶ Removed the first Parameter.

Okay, now let's look at that pesky compiler error:

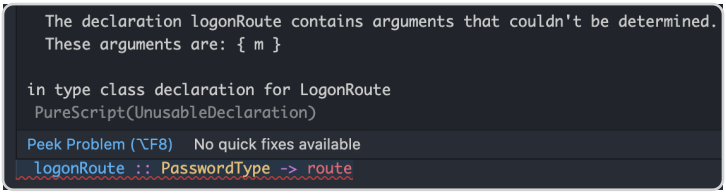


We have an Orphaned Instance since we're coding in `AppM.purs`, which is not where `Route` is defined.

My first mistake was not putting an `m` in the Typeclass. We can fix that by doing the following:

```
-- COMPILER ERROR!!
class LogonRoute m route where
  logonRoute :: PasswordType -> route
```

But now the compiler is complaining that `logonRoute` doesn't have a an `m`:



We could simply fix this with a Functional Dependency:

```
-- NOT A GOOD IDEA!!
class LogonRoute m route | route -> m where
  logonRoute :: PasswordType -> route
```

This compiles but still requires us to put the Typeclass instance in `Route`. This is bad since we want to add the Capability Constraint to the Component for `m`. There is no `Route` Parameter for our Components.

So, we could reverse the Functional Dependency:

```
-- COMPILER ERROR!!
class LogonRoute m route | m -> route where
  logonRoute :: PasswordType -> route
```

And now the compiler is complaining about not have an `m` again in the Type Signature of `logonRoute`.

So, we'll simply have to change this Pure Function into a Monadic one:

```
import Prelude ❶

class Monad m <= LogonRoute m route where ❷
  logonRoute :: PasswordType -> m route ❸
```

❶ Added to get `Monad`.

❷ Made `Monad` a Superclass.

❸ Changed return Type from `route` to `m route`.

Update your definition. Then change your `AppM` Instance to add `pure` to the return Value:

```
instance logonRouteAppM :: LogonRoute AppM Route where ❶
  logonRoute PasswordPermanent = pure $ Route.Users Nothing
  logonRoute PasswordTemporary = pure Route.ChangePassword
```

❶ Added `AppM` for `m`.

Now add a Halogen Instance for `LogonRoute` using what we did with `Navigate` and `Log` as references, then return here when you're done.

Here's that change:

```
import Control.Monad.Trans.Class (lift) ❶
import Halogen (HalogenM)

instance logonRouteHalogenM
    :: LogonRoute m route
    => LogonRoute (HalogenM state action slots output m) route where
  logonRoute = lift <<< logonRoute
```

❶ Added these imports.

The next Capability to implement in `AppM` is `Navigate`. Look up `setHash` from `Routing.Hash` in `Pursuit` before attempting to code this Instance. Also, look up `print` in `Routing.Duplex`. Give it a try before reading on.

Here's the implementation:

```
import Capability.Navigate (class Navigate) ❶
import Effect.Class (class MonadEffect, liftEffect)
import Routing.Duplex as RouteDuplex
import Routing.Hash (setHash)

instance navigateAppM :: Navigate AppM Route where
  navigate = liftEffect <<< setHash
    <<< RouteDuplex.print Route.routeCodec ❷ ❸ ❹
```

❶ Added these imports.

❷ Written Point-free with the `Route` being the unnamed Parameter.

❸ First we take our `Route` and run it through our Codec to produce a `String` that we then give to `setHash` to change the Browser's Hash value.

❹ Had to `liftEffect` since `setHash` runs in `Effect`.

And finally, write the code for the `Log` Instance. Look up, in `Pursuit`, `Effect DateTime` to find a Function that will get the current time into a `DateTime`.

When you implement `log` make sure you log the whole Record to the console.

Return here when you're done.

Here's that implementation:

```
import Capability.Log (class Log) ❶
import Effect.Class.Console as Console ❷
import Effect.Now (nowDateTime)

instance logAppM :: Log AppM where
  logEntry level message =
    liftEffect nowDateTime >>= pure <<< { level, message, timestamp: _ } ❸ ❹
  log = Console.log <<< show -- COMPILER ERROR!! ❺
```

❶ Added these imports.

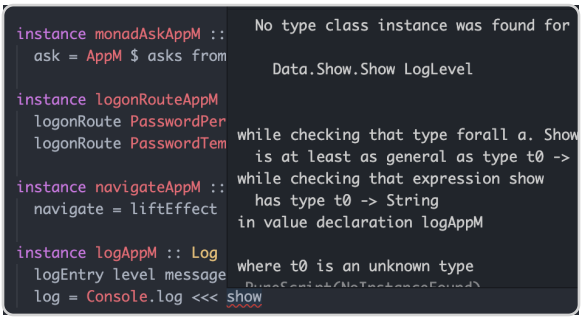
❷ Fully qualify as `Console` since `log` will conflict with our Instance Method, `log`.

❸ `liftEffect` on `nowDateTime` so it'll run in `AppM`,

❹ `bind` to an Anonymous Function to create `LogEntry` and then compose with `pure` to wrap it in `AppM`. You can look at the `_` as the space where the timestamp goes. It the Value returned from `nowDateTime`.

❺ `show` the `LogEntry` Record.

And we have a compiler error:



Looks like `LogLevel` isn't showable. Add the code to make it so before reading any further.

Here are the lines of code to add to `Capability.Log`:

```
import Data.Generic.Rep (class Generic)
import Data.Show.Generic (genericShow)

derive instance genericLogLevel :: Generic LogLevel _

instance showLogLevel :: Show LogLevel where
  show = genericShow
```

And now return to `AppM.purs` and save the File to see if it compiles.

And it does.

30.6. Router Component

Let's look back on our implementation of `Navigate`:

```
instance navigateAppM :: Navigate AppM Route where
  navigate = liftEffect <<< setHash
    <<< RouteDuplex.print Route.routeCodec
```

We're setting the Browser's Hash value, but no one is listening to the change.

We could use `addEventListener`:

```
addEventListener
  :: EventType
  -> EventListener
  -> Boolean
  -> EventTarget
  -> Effect Unit
```

And then we'd have to use the following Functions to be able to handle Hash change events:

```
hashchange :: EventType
eventListener :: ∀ a. (Event -> Effect a) -> Effect EventListener
toEventTarget :: Window -> EventTarget
```

But, as it turns out `Routing.Hash` has already done all of the hard work for us in a Function called `matchesWith`, which allows us to pass in our own parser:

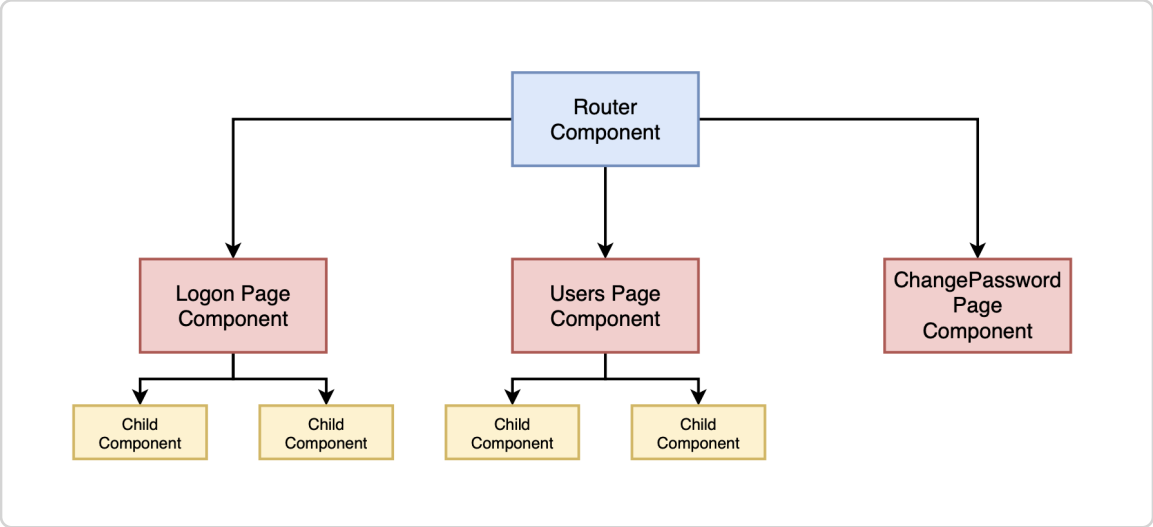
```
matchesWith
  :: ∀ f a
   . Foldable f
  => (String -> f a) ❶
  -> (Maybe a -> a -> Effect Unit)
  -> Effect (Effect Unit)
matchesWith parser cb = ... ❷
```

❶ The parser must return a `Foldable`. This may seem to be a problem since `RouteDuplex's` parser returns an `Either`, but as it turns out, `Either` has a `Foldable` Instance.

❷ The implementation is not important here.

Our plan will be to use `matchesWith` in our `main` and the use `H.mkTell` to let our Top-level Component know that the Hash has changed. `H.mkTell` is used when telling a Top-level Component.

This Top-level Component will be our Router Component:



Each Page will be a Child Component of the Router Component. It will decide which Child to render depending on the current `Route`.

For now, let's build the Router Component. We'll stub out the `render` Function to display a message based on the page that's supposed to be displayed for now.

Make a Folder called `Component` under `src` then make a File called `Router.purs` under `src/Component`.

Use your `halo` Application as a reference guide to help you as you write the code for `Router`.

As with any Halogen Component, we'll have to define Types for the following:

- `Input` (from Parent)
- `Output` (to Parent)
- `State`
- `Action` (events from EventListeners)
- `Query` (`H.tell` or `H.request` from Parent)
- `Slots` (for Children)

Define `Input` for `Router` before reading on.

There isn't any input from the Application, i.e. the Top-level's "Parent", so we'll make it `Unit`:

```
import Prelude ❶

type Input = Unit
```

❶ Added import for `Unit`.

Nothing in our specification for Router tells us that we need `Input` yet. We may discover that this needs to change, but for now, we'll just make it `Unit`. Remember, Parent Components always provide `Input` to their Child Components. This is why we cannot make this `Void`.

Next, define `Output`, then read on.

There is no useful information to communicate to the Application so we'll just make it `Void`:

```
type Output = Void
```

Notice how we made `Output` equal to `Void` but we didn't make `Input` a `Void`. Why not?

It turns out that we always have to pass an `Input` to the Top-level or Child Component, so it must exist. Since we're never going to call `H.raise`, we have no such requirement for the `Output`.

Next, we should define the `State`. This will help us define `Action` and `Query`. Do this and then keep reading.

At the moment there's no real need to keep track of anything in our `Router`:

```
type State = {}
```

And now, define `Action` and then return back here.

Once again, there's no will be no `EventSources` in our Router. There will only be Child Components which are the Pages of our Application:

```
type Action = Void
```

And now define `Query` keeping in mind that we're going to get route changes from the Application. This will be done with `H.mkTell` to let us know that the Hash changed requiring us to Route to the new page.

Also, refer to our `halo` Application, especially the `Query` definition in the Counter Component.

Once you're done, continue reading.

Here's our definition:

```
import Data.Route (Route) ❶

data Query a = Navigate Route a ❷ ❸
```

❶ Added this import.

❷ The `a` is the rest of the computation and is required of the Query API by Halogen.

❸ `Route` is the route to navigate to.

And finally, we have to define the `Slots` for all of the Child Components which will be our individual Pages. Once again refer to the `halo` Application. Return back here once you're done.

Here's the Row definition:

```
import Data.Const (Const) ❶
import Halogen as H

type PageSlot = H.Slot (Const Void) Void Unit ❷ ❸

type Slots =
  ( logon :: PageSlot
  , logoff :: PageSlot
  , users :: PageSlot
  , changePassword :: PageSlot
  )
```

❶ Added imports.

❷ Convenience Type for a Slot that's just a Page. Note the `Unit` to uniquely distinguish Children of the same Type. There will never be more than one Page as a Child so `Unit` is enough to uniquely identify the one and only.

❸ Since there is no communication from the Pages to the Top-level Component, the `query` is set to `Const Void` and `output` is `Void`.

Next, let's add the `Symbol Proxies` for the Child Slots:

```
import Type.Proxy (Proxy(..)) ❶

_logon = Proxy :: Proxy "logon"
_logoff = Proxy :: Proxy "logoff"
_users = Proxy :: Proxy "users"
_changePassword = Proxy :: Proxy "changePassword"
```

❶ Added this import.

At this point, we're ready to create the Component. Write the `component` Function and just stub out the `render` and `handleQuery` Functions. Once again, use the `halo` Application as a reference.

Return here when you're ready to move on.

First, we'll code the `component` Function:

```
import Effect.Aff.Class (class MonadAff) ❶
import Halogen.HTML as HH

component
  :: ∀ m
  .  MonadAff m ❷
  => H.Component Query Input Output m
component = H.mkComponent
  { initialState: \_ -> {} ❸
  , render
  , eval: H.mkEval H.defaultEval {
      handleQuery = handleQuery ❹
    }
  }
  where
  ???
```

❶ Added these imports.

❷ The Monad `m` runs in `Aff`. If you look at `AppM.purs`, you'll see that we derived `MonadAff` for `AppM`.

❸ Our input to the Lambda is `Unit`, so we use `_`. We return an empty Record for our `State`.

❹ We only need to implement `handleQuery` all other values are defaulted thanks to `H.defaultEval`. This is an easy thing to forget. From time to time, you will test your Component and it won't work. And you'll look at your code and scratch your head. When that happens, check here to make sure you've added `handleAction`, `handleQuery`, `receive`, etc.

Next, we'll add our stubbed out Functions:

```
import Data.Maybe (Maybe(..)) ❶

-- ABOVE CODE REDACTED
  where
  handleQuery
    :: ∀ a
    .  Query a
    -> H.HalogenM State Action Slots Output m (Maybe a)
  handleQuery _ = pure Nothing

  render :: State -> H.ComponentHTML Action Slots m
  render _ = HH.text ""
```

❶ Added this import.

This should compile.

The next step is to flesh out our stubbed out Functions. We'll start with `handleQuery`.

Ultimately, we're going to want to make sure the user's logged on before we honor `Navigate`. If the user is NOT logged on, then we'll redirect them to the Logon Page.

But for now, we're ONLY going to save the `route` and let the `render` Function render the appropriate Child Component.

That last statement should make it clear that our `State` definition is deficient. We'll need to save the `route` in our `State`.

So first, add `route` to `State`. Then finish `handleQuery` to save the `route` to the `State`. Later on, we'll check to make sure that the user is logged on.

Don't forget to use the `halo` Application code we wrote to remind yourself how to modify `State` in our Component code. Also, make sure you look at how to properly return a value from `handleQuery`.

Once you've completed that task, return here.

First step is to add `route` to `State`:

```
type State = { route :: Route }
```

Note that this is not a `Maybe Route`. We have a reasonable inital Value to give this when we initialize the `State`. We'll set it to `Logon` since the user can't possibly be logged on when the Router Component is initialized.

Next, we need to change `initialize`:

```
import Data.Route (Route(..)) ❶
import Capability.Navigate (class Navigate) ❷

component
  :: ∀ m
  .  MonadAff m
  => Navigate m Route ❸
  => H.Component Query Input Output m
component = H.mkComponent
  { initialState: \_ -> { route: Logon } ❹
  , render
  , eval: H.mkEval H.defaultEval {
      handleQuery = handleQuery
    }
  }
```

❶ Added `(..)` to get the Data Constructors.

❷ Added this import.

❸ This Component will do navigation so we'll need that Capability from our Monad, `m`.

❹ Initialize to `Logon`.

Now, we're ready to code `handleQuery`:

```
  handleQuery
    :: ∀ a
    .  Query a
    -> H.HalogenM State Action Slots Output m (Maybe a)
  handleQuery = case _ of ❶
    Navigate route a -> H.modify_ _ { route = route } *> pure (Just a) ❷ ❸
```

❶ Removed the first Parameter and replaced with a wildcard `case` expression.

❷ Modify the `State` with the new `Route`.

❸ Using `applySecond` to return the next computation to evaluate, i.e. `pure (Just a)`.

We're not really ready to write the full `render` Function, so just put an appropriate text Node on the page depending on the `Route` in the `State`, e.g. `HH.text "Logon"` for the Logon Page, `HH.text Logoff` for Logoff, etc.

Once you done, return here.

Here's that change:

```
  render :: State -> H.ComponentHTML Action Slots m
  render { route } = case route of
    Logon -> HH.text "Logon"
    Logoff -> HH.text "Logoff"
    Users _ -> HH.text "Users"
    ChangePassword -> HH.text "ChangePassword"
```

This is pretty good but, it's going to be hard to be read against our background, which will be dark. Let's fix that:

```
import CSS (color, white) ❶
import Halogen.HTML.CSS as HC

  render :: State -> H.ComponentHTML Action Slots m ❷
  render { route } = case route of
    Logon -> HH.span [ HC.style $ color white ] [ HH.text "Logon" ]
    Logoff -> HH.span [ HC.style $ color white ] [ HH.text "Logoff" ]
    Users _ -> HH.span [ HC.style $ color white ] [ HH.text "Users" ]
    ChangePassword ->
      HH.span [ HC.style $ color white ] [ HH.text "ChangePassword" ]
```

❶ Added these imports to be eventually be removed.

❷ Changed `render` to put the text in a `<span>` where the text is `white`.

We'll replace the stand-in `HH.text` with the Children to `Router` as we develop them.

Once you save this, you'll note the compiler error on the 2 new imports.

We'll need to add `css` to the `spago.dhall` for `import CSS`.

Also, I forgot to add `halogen-css` to `spago.dhall` when we added it to `package.dhall`. Let's fix that:

```
{-
Welcome to a Spago project!
You can edit this file as you like.
-}
{ name = "my-project"
, dependencies =
  [ "aff"
  , "console"
  , "css" (1)
  , "effect"
  , "either"
  , "halogen"
  , "halogen-css" (1)
  , "maybe"
  , "prelude"
  , "psci-support"
  , "refs"
  , "routing"
  , "routing-duplex"
  , "transformers"
  , "uuid"
  ]
, packages = ./packages.dhall
, sources = [ "src/**/*.purs", "test/**/*.purs" ]
}
```

(1) Added these.

Now build the app with:

```
npx spago build
```

This command installed the new packages into our `.spago` directory, but gives us the following errors:

```
[error] Some of your project files import modules from packages that are not
in the direct dependencies of your project.
To fix this error add the following packages to the list of dependencies in
your config:
- const
- datetime
- now
```

Go ahead and add these to your `spago.dhall` and rebuild. It should build with no error and no warnings.

At this point, our code is complete enough for us to test, but we'll have to create a `Halogen` application. We'll do this by modifying our `Main.purs`.

Now, let's change our `main` Function to run our Top-level Component in the `AppM` Monad. We'll code this together since there are some new aspects to running in a Monad other than `Aff`.

In our `halo` Application, we used `runUI` since all component evaluations were run in `Aff`. But our Application is running in `AppM`, which has a `MonadAff` instance. That means that we can use a Halogen helper Function called `hoist`:

```
hoist
  :: ∀ query input output m m'
  .  Functor m'
  => (m ~> m')
  -> Component query input output m
  -> Component query input output m'
```

This can be quite daunting when you first look at it. So, let's take it one step at a time.

Forgetting all of the Constraints for a moment, the first Parameter is `m ~> m'`, which is a Natural Transformation from one `Functor` to another. In our case, we have `AppM` that we want to transform into `Aff` so we can use it with `runUI`.

We have such a Function:

```
runAppM :: Env -> AppM ~> Aff ❶
```

❶ Once we provide an `Env`, this will do as our first Parameter to `hoist`, i.e. it's a Natural Transformation from `AppM` to `Aff` unifying `AppM` with `m` and `Aff` with `m'` in `hoist's` Type Signature.

Next we need a Component that runs in `m`, which, for us, is `AppM`. We have such a Component. Our `Router` was written with the idea that it'll run in `AppM`.

And finally, `hoist` will return us a Component that runs in `Aff`. This component is what `runUI` wants.

Our Router Component has the following definition:

```
H.Component Query Input Output m ❶
```

❶ This is the return Type of `component` in `Component.Router`.

There's a Constraint that `m'` must be a `Functor` and, in our case, `m'` is `Aff`, which is a `Monad` and therefore a

`Functor`.

So, after all of that, we make our `Main` module look like:

```
module Main where

import Prelude

import AppM (runAppM)
import Component.Router as Router
import Data.Maybe (Maybe(..))
import Effect (Effect)
import Effect.Ref as Ref
import Halogen as H
import Halogen.Aff as HA
import Halogen.VDom.Driver (runUI)

main :: Effect Unit
main = do
  userRef <- Ref.new Nothing ❶
  HA.runHalogenAff do ❷
    body <- HA.awaitBody ❸
    let router = H.hoist (runAppM { userRef }) Router.component ❹
    runUI router unit body ❺
```

❶ Create a `userRef` of `Nothing` since no user has logged on yet.

❷ Run Halogen code with this. Don't forget this is just `runAff` with unhandled errors thrown as exceptions.

❸ Wait for the `body` to be rendered by the Browser.

❹ Hoist our `router`, which runs in `AppM`, to now run in `Aff`.

❺ Run our Top-level Component passing `Unit` as `Input`. This will go to the `initialize` Function in the Component that will build the inital `State`.

Now, the only thing that we're missing is the initial HTML. Create a file called `index.html` under the root directory of your project and steal all of the HTML code from the `halo` Application and paste it here:

```
<!DOCTYPE html>
<html lang="en" dir="ltr">

<head>
  <meta charset="utf-8">
  <title>HALO - Learning</title>
</head>

<body>
  <script src="/index.js" charset="utf-8"></script>
</body>

</html>
```

Then add the following edit to the `body` tag:

```
<body style="margin:0; background-color:#383333"> ❶ ❷
```

❶ Many browsers have a `margin` of `8` for the `body` as a default, so this will remove that.

❷ We're going to want this background color.

And then make the edit on the `title` tag:

```
  <title>Functional Programming Made Easier</title>
```

Copy and paste the `Makefile` from the `halo` Application as well:

```
build:
    npx spago build

run:
    npx spago bundle-app
    python -m SimpleHTTPServer

.PHONY: build run
```

Make sure you put `Makefile` under the root directory of your project.

Now, we're finally ready to run our Application:

```
make run
```

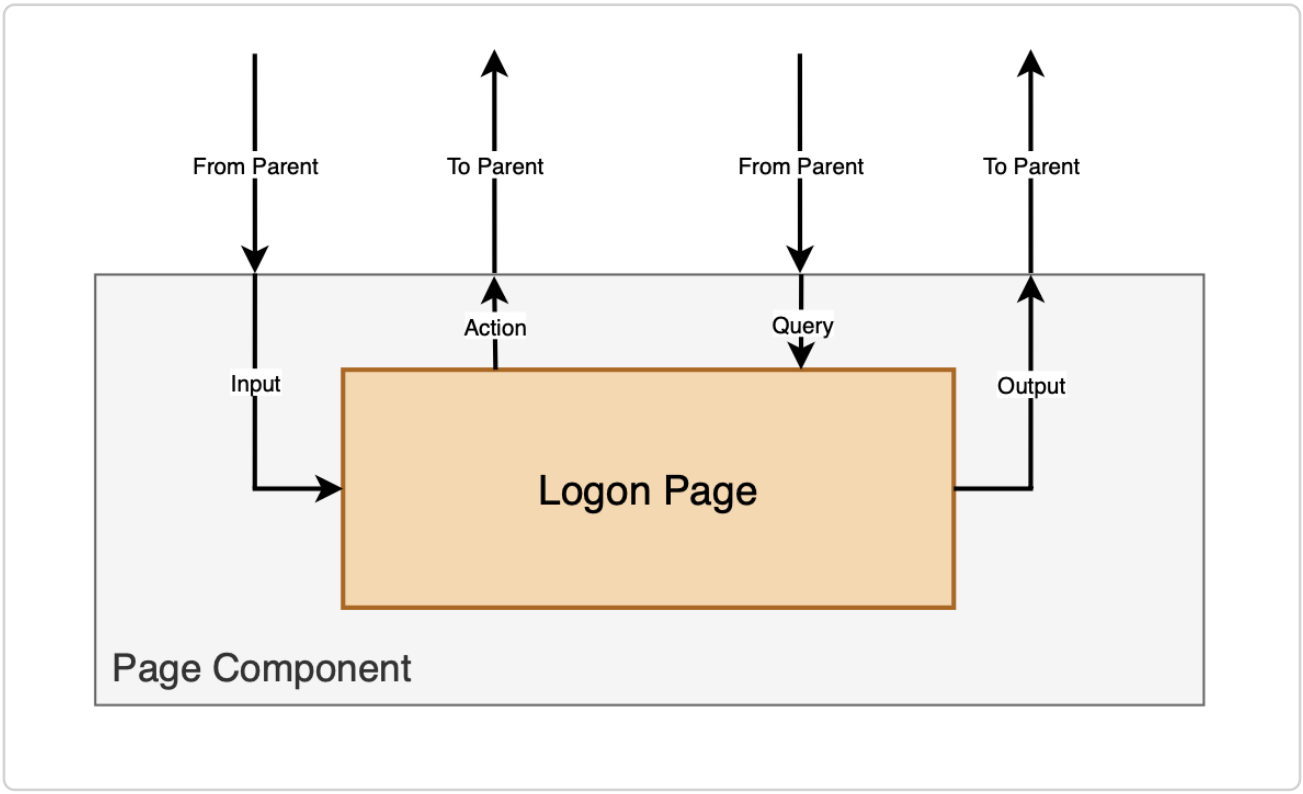
Once it builds and is running, go you your browser at `localhost:8000` and you should see the text, `Logon` in `white` on the dark background we specified in `index.html`.

30.7. Page Component

We're going to want to build a common Page Component that will contain the header of our application with an always present `Logoff` button.

We're going to build it as a Container for any Page Component, e.g. the Logon Page. It will forward all Inputs it receives to the Inner Component. It will also forward any Output it gets from the Inner Component to the Page Component's Parent.

One of it's goals is to inject some layout and functionality while appearing invisible in the traditional Parent/Child Hierarchy:



`Input`, `Output`, `Action` and `Query` are all redirected to and from the Parent.

We're going to code this together since it's a new concept.

First, create a File `Page.purs` under `src/Component` with:

```
module Component.Page where

import Prelude
```

We're are NOT going to define `Input` or `Output` or `Query` for this Component. Those Types will be taken from the Inner Component.

We will, however, define `State` and `Action`. We'll start with `State`:

```
type State iInput = { iInput :: iInput }
```

`iInput` is the Inner Component's `Input` Type. We need to store this in our State because the only way we send `Input` to our Children is by rendering them.

By storing the `Input` we get from our Parent into our `State`, our `render` Function will be called, since `State` changes trigger re-renders.

This will allow us to pass this new `Input` to the Inner Component. We'll see this in action once we write the `render` Function.

Next, we're going to want the following `Action` definition:

```
data Action iInput iOutput ❶
  = Input iInput ❷
  | Output iOutput ❸
  | Logoff ❹
```

❶ `iInput` and `iOutput` are the Types for the Inner Component's `Input` and `Output` respectively.

❷ This `Action` will handle receiving an `Input` from our Parent. This will be sent to the Inner Component in `render`.

❸ This `Action` will handle receiving an `Output` from the Inner Component. This will be sent to the Parent using `H.raise`.

❹ When the user clicks on the `Logoff` button, this `Action` will `navigate` to the `Logoff` Route.

Next, we'll define the Child Slots:

```
import Halogen as H ❶

type Slots iQuery iOutput = ( inner :: H.Slot iQuery iOutput Unit )
```

❶ Added this import.

`iQuery` and `iOutput` are the Types for the Inner Component's `Query` and `Output` respectively.

There is only 1 Child Component, hence the use of `Unit` for the `slot` Type Parameter.

Now, we'll create a `Symbol Proxy` to refer to the one and only label in the Row Type:

```
import Type.Proxy (Proxy(..)) ❶

_inner = Proxy :: Proxy "inner"
```

❶ Added this import.

Now, we can define the `component`. This will be a little different than the usual since our Types are Polymorphic, e.g. `iInput`, `iOutput` and `iQuery`.

Let's start with just the Type Signature:

```
import Capability.Navigate (class Navigate) ❶
import Data.Route (Route)
import Effect.Aff.Class (class MonadAff)
import Halogen.HTML as HH

component
  :: ∀ iQuery iInput iOutput m ❷
   . MonadAff m ❸
  => Navigate m Route ❹
  => H.Component iQuery iInput iOutput m ❺
  -> H.Component iQuery iInput iOutput m ❻
```

❶ Added these imports.

❷ Notice that we have additional Polymorphic Parameters for `iQuery`, `iInput` and `iOutput`.

❸ We always need this since our Monad, `m`, eventually runs in `Aff`.

❹ We'll need this Capability to be able to navigate to the Logoff page.

❺ This is the Inner Component. It's Types are the Polymorphic Types, `iQuery`, `iInput` and `iOutput`.

❻ Notice how this Component mirrors the Types of its Inner Component. This is how it looks invisible to Parent.

Next, we'll add the implementation:

```
import Data.Maybe (Maybe(..)) ❶

component
  :: ∀ iQuery iInput iOutput m
   . MonadAff m
  => Navigate m Route
  => H.Component iQuery iInput iOutput m
  -> H.Component iQuery iInput iOutput m
component innerComponent = H.mkComponent
  { initialState: \iInput -> { iInput } ❷
  , render                                -- COMPILER ERROR!!
  , eval: H.mkEval $ H.defaultEval
      { handleAction = handleAction ❸
      , handleQuery = handleQuery ❹
      , receive = Just <<< Input ❺
      }
  }
```

❶ Added this import.

❷ When we initially get our `Input`, it's used to initialize the `State` so when `render` gets called, the `Input` can be passed the Inner Component.

❸ `handleAction` is needed for our `Actions`.

❹ `handleQuery` is needed to delegate `Queries` from the Parent.

❺ When we `receive` Input from our Parent, we turn it into an `Action` that will store it in the `State` for the `render` Function to pass to our Inner Component.

It's no surprise that we have a compiler error since we're not done yet.

Let's add `handleAction`. First, the Type Signature:

```
-- ABOVE CODE REDACTED
  where
  handleAction
    :: Action iInput iOutput
    -> H.HalogenM (State iInput) (Action iInput iOutput)
                  (Slots iQuery iOutput) iOutput m Unit
```

This seems a lot more confusing than our usual `handleAction` Type Signature, but if look carefully, you'll see that it's not that much different from the more familiar:

```
-- OURS
  handleAction
    :: Action iInput iOutput
    -> H.HalogenM (State iInput) (Action iInput iOutput)
                  (Slots iQuery iOutput) iOutput m Unit
-- COMMON
handleAction
    :: Action
    -> H.HalogenM State           Action
                  Slots                 Output  m Unit
```

Now, we'll code the Handler, one `Action` at a time staring with `Input`:

```
-- ABOVE CODE REDACTED
  where
  handleAction
    :: Action iInput iOutput
    -> H.HalogenM (State iInput) (Action iInput iOutput)
                  (Slots iQuery iOutput) iOutput m Unit
  handleAction = case _ of ❶
    Input input -> H.modify_ _ { iInput = input } ❷
    ???
```

❶ Here's the canonical wildcard `case` expression.

❷ We modify the `State` to add the new `Input` Value.

The `Input` is stored in the `State` triggering a re-render. It's in `render` that the Inner Component will get this Value.

Next, we're going to handle `Output`:

```
-- ABOVE CODE REDACTED
  where
  handleAction
    :: Action iInput iOutput
    -> H.HalogenM (State iInput) (Action iInput iOutput)
                  (Slots iQuery iOutput) iOutput m Unit
  handleAction = case _ of
    Input input -> H.modify_ _ { iInput = input }
    Output output -> H.raise output ❶
```

❶ An `Output`, which we received when the Inner Component called `H.raise`, is sent to the Parent using

`H.raise`.

And finally, we're going to handle when the user clicks on `Logoff`:

```
import Capability.Navigate (class Navigate, navigate) ❶
import Data.Route as Route ❷

-- ABOVE CODE REDACTED
  where
  handleAction
    :: Action iInput iOutput
    -> H.HalogenM (State iInput) (Action iInput iOutput)
                  (Slots iQuery iOutput) iOutput m Unit
  handleAction = case _ of
    Input input -> H.modify_ _ { iInput = input }
    Output output -> H.raise output
    Logoff -> navigate Route.Logoff ❸
```

❶ Added `navigate` to the import list.

❷ Fully qualify `Route` so `Logoff` won't be ambiguous, since it could represent a `Route` or an `Action`.

❸ Navigate to the Logoff Page.

Now, we're ready to code `handleQuery`. This just needs to delegate the call to the Inner Component.

Take a look at the `halo` Application to see how we wrote the Type Signature for `handleQuery` there. Remember, our Types have Polymorphic Parameters, e.g. `State` in the `halo` Application should be replaced with `State iInput` here.

Stop reading and see if you can write just the Type Signature. Return here once you've completed that task to see how you did.

Here's the Type Signature for `handleQuery`:

```
-- ABOVE CODE REDACTED
  where
  -- handleAction REDACTED
  handleQuery
    :: ∀ a
    .  iQuery a
    -> H.HalogenM (State iInput) (Action iInput iOutput)
                  (Slots iQuery iOutput) iOutput m (Maybe a)
```

Now, write the implementation, but delegate it to the Inner Component using `H.query`. Once again, look to

`halo` for help.

Return to here once you're ready to move on.

Here's that line of code:

```
-- ABOVE CODE REDACTED
  handleQuery
    :: ∀ a
    .  iQuery a
    -> H.HalogenM (State iInput) (Action iInput iOutput)
                  (Slots iQuery iOutput) iOutput m (Maybe a)
  handleQuery = H.query _inner unit ❶ ❷ ❸
```

❶ This is written Point-free where `Query` is unnamed.

❷ We call `H.query` on the Inner Component by using the `Symbol Proxy`, `_inner`, which is defined using `Proxy`. We're using `H.query` here since we don't know if it's a Request or a Tell Query.

❸ The second Parameter to `H.query` uniquely identifies which Child and since there's only one, we're using `unit`.

And now, I'm just going to give you the code for `render`. It's mostly HTML and CSS stuff here so I don't want to waste time coding it incrementally. As you type it in, please try to pay attention to what the code is doing.

We will review in detail any groundbreaking code in `render` below.

Make sure, like `handleAction` and `handleQuery`, these Functions are tabbed in under the `where`:

```
import CSS (CSS)
import CSS as CSS
import CSS.Background (backgroundColor)
import CSS.Box (boxShadow)
import CSS.Color (Color, rgba, rgb, white)
import CSS.Common (center)
import CSS.Cursor (cursor, pointer)
import CSS.Display (display, zIndex, position, fixed, flex)
import CSS.Font ( FontWeight(..), sansSerif, fontFamily, color, fontSize
                , fontWeight)
import CSS.Flexbox (flexDirection, row, flexStart, flexEnd, flexBasis
                   , flexShrink, flexGrow, alignItems, justifyContent)
import CSS.Geometry ( padding, paddingTop, paddingLeft, paddingRight
                    , width, height, minHeight)
import CSS.Property (value)
import CSS.Size (rem, px, pct, vh)
import CSS.Text (letterSpacing)
```

```
import CSS.Text.Shadow (textShadow)
import Data.NonEmpty ((:|))
import Halogen.HTML.CSS as HC
import Halogen.HTML.Events as HE
import Halogen.HTML.Properties as HP

-- ABOVE CODE REDACTED
  where
-- handleAction and handleQuery REDACTED
  paperColor :: Color ❶
  paperColor = rgb 0xd9 0xd9 0xd9

  themeColor :: Color
  themeColor = rgb 0x00 0x66 0x75

  themeFont :: CSS
  themeFont = fontFamily [ "Verdana" ] $ sansSerif :| []

  render
    :: State iInput
    -> H.ComponentHTML (Action iInput iOutput) (Slots iQuery iOutput) m
  render { iInput } =
    HH.div [
      HC.style do
        paddingTop $ rem 5.0
    ]
    [ HH.header [
        HC.style do
          position fixed
          CSS.top $ px 0.0
          width $ pct 100.0
          height $ rem 4.0
          boxShadow (px 10.0) (px 10.0) (px 20.0) (rgba 0 0 24 0.75)
          display flex
          alignItems center
          justifyContent flexStart
          backgroundColor themeColor
          zIndex 1
      ]
      [ HH.div [
          HC.style do
            paddingLeft $ rem 1.0
            color paperColor
            display flex
```

```
            flexDirection row
            alignItems center
            flexBasis $ px 0.0
            flexGrow 1.0
            flexShrink 1.0
        ]
        [ HH.img [
            HC.style do
              width $ px 40.0
          , HP.src bookCover -- COMPILER ERROR!!
          ]
        , HH.span [
            HC.style do
              paddingLeft $ rem 1.0
              fontSize $ rem 1.3
              themeFont
              textShadow (px 0.0) (px 0.0) (px 5.0) (rgba 0 0 0 1.0)
              fontWeight $ FontWeight $ value "500"
              letterSpacing $ rem 0.05
          ]
          [ HH.text "Functional Programming Made Easier" ]
        , HH.div [
            HC.style do
              display flex
              flexGrow 1.0
              justifyContent flexEnd
              paddingRight $ rem 1.0
          ]
          [ HH.span [
              HC.style do
                themeFont
                fontWeight $ FontWeight $ value "500"
                color white
                padding (rem 1.0) (rem 1.0) (rem 1.0) (rem 1.0)
                cursor pointer
            , HE.onClick $ const Logoff ❷
            ] [ HH.text "Logoff" ]
          ]
        ]
      ]
    , HH.div [
        HC.style do
          display flex
          alignItems center
```

```
          justifyContent center
          minHeight $ vh 90.0
      ]
      [ HH.slot _inner unit innerComponent iInput Output ] ❸
    ]
```

❶ The small Functions here are general styling constants that are used in `render`. This will eventually be moved out so it can be used by other Components.

❷ We handle the `Logoff` click here.

❸ Here's where the Inner Component finally gets its `Input`. Remember that `render` is passed the `State` where `iInput` is stored in preparation for this call.

The compiler error is because we don't have an image for the website. First, create a new Folder called `Image` under `src`.

Then go to my GitHub Gist to get a PureScript Module that contains the Data URL that is the image. You can find it here at this link, BookCover.purs. Click it.

Now, copy the code into a new File called `BookCover.purs` and place it in the `src/Image` Folder. The easiest way to do copy this code from Github is to click the `Raw` button and then select all of the text via `Cmd-A` or `Ctrl-A` on Windows then copy the text.

Save that file and then return back to `Component.Page` and add:

```
import Image.BookCover (bookCover)
```

It should now compile.

30.8. Logon Page

We would like to test our code sooner than later. So, we're just going to make a fully stubbed out Logon Page so we can test.

Create a File called `Logon.purs` under `src/Component` and add the following code:

```
module Component.Logon where

import Prelude

import CSS (color, white)
import Data.Const (Const)
import Effect.Aff.Class (class MonadAff)
import Halogen as H
import Halogen.HTML.CSS as HC
import Halogen.HTML as HH

type Input = Unit ❶
type Output = Void
type State = {}
type Action = Void

type Query :: ∀ k. k -> Type ❷
type Query = Const Void

type Slots :: ∀ k. Row k ❸
type Slots = ()

component
  :: ∀ m
  .  MonadAff m
  => H.Component Query Input Output m
component = H.mkComponent
  { initialState: \_ -> {}
  , render
  , eval: H.mkEval H.defaultEval ❹
  }
  where
  render :: State -> H.ComponentHTML Action Slots m
  render {} = HH.span [ HC.style $ color white ] [ HH.text "Logon" ] ❺
```

❶ All the Component Types are trivially defined.

❷ Don't forget the Type Signature here. If you do, the compiler will remind you.

❸ We add a Type Signature here just like we did `Query` since the compiler cannot figure out our kind.

❹ We're just using the `defaultEval`, which means we don't have to implement any handlers.

❺ We render a simple Text Node.

Let's add this new Component as a Child to our Top-level Component, i.e. the Router Component:

```
import Component.Logon as Logon ❶
import Component.Page as Page

-- ABOVE CODE REDACTED
  render :: State -> H.ComponentHTML Action Slots m
  render { route } = case route of
    Logon ->
      HH.slot _logon unit (Page.component Logon.component) unit absurd ❷
    Logoff -> HH.span [ HC.style $ color white ] [ HH.text "Logoff" ]
    Users _ -> HH.span [ HC.style $ color white ] [ HH.text "Users" ]
    ChangePassword ->
      HH.span [ HC.style $ color white ] [ HH.text "ChangePassword" ]
```

❶ Added these imports.

❷ Render `Logon` Child Component.

Now, we should be able to run the Application again:

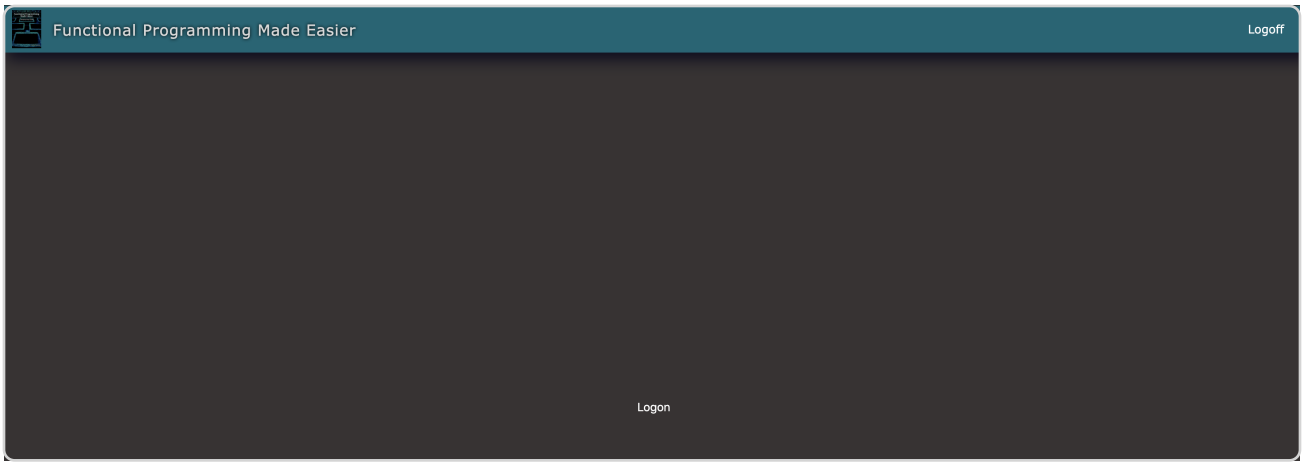
```
make run
```

However, we get this error:

```
[error] Some of your project files import modules from packages that are not
in the direct dependencies of your project.
To fix this error add the following packages to the list of dependencies in
your config:
- nonempty
```

Add this package to `spago.dhall` and then use `make run` to run it.

Now, navigate to `localhost:8000` and refresh the page. You should see:



Now, click the `Logoff` button and see what happens.

I get a new URL:

```
localhost:8000/#/logoff
```

But the page still says `Logon`. Why is that?

It turns out that we talked about using `matchesWith` for routing, but we never added it to our Application.

Let's add that now:

```
import Data.Route (routeCodec) ❶
import Effect.Aff (launchAff_)
import Effect.Class (liftEffect)
import Routing.Hash (matchesWith)
import Routing.Duplex as RouteDuplex

main :: Effect Unit
main = do
  userRef <- Ref.new Nothing
  HA.runHalogenAff do
    body <- HA.awaitBody
    let router = H.hoist (runAppM { userRef }) Router.component
    io <- runUI router unit body ❷
    liftEffect $ matchesWith (RouteDuplex.parse routeCodec) ❸ ❹
      \old' new -> when (old' /= Just new) $ -- COMPILER ERROR!! ❺
        launchAff_ $ io.query $ H.mkTell $ Router.Navigate new ❻ ❼
```

❶ Added these imports.

❷ Capture the `HalogenIO` from `runUI` so we can `Query` the Router to let it know that the Hash changed.

❸ We have to `liftEffect` since we're in `Aff` but `matchesWith` runs in `Effect`.

❹ We provide our custom parser, i.e. `RouteDuplex.parse routeCodec`.

❺ When the old hash is not the same as the new one, then consider it changed.

❻ We have to run `io.query` in `Aff` but this Lambda returns an `Effect`. We use `launchAff_` to run it in `Aff`.

❼ We `tell` the Router Component, which is our Top-level Component, to `Navigate`. This will change its `State` so that it'll render the correct Child Page.

`matchesWith` listens for the `hashChange` Event and then calls our custom parser and if it is successful it'll call the callback with `Maybe a`, i.e. `Maybe` the previous `Route` and the current `Route`.

If they are different, then our callback will `Query` our Router Component telling it, via `H.mkTell`, to Navigate to a different `Route`. The Router Component stores that in its `State` and `renders` the appropriate Page depending on the current `Route`.

Update you code to this before we look at the compiler error.

The compiler error is because `Route` doesn't have an `Eq` Instance. Add that Instance using `derive` before reading any further.

Here's that addition added to `Data.Route`:

```
derive instance eqRoute :: Eq Route
```

And now, our `Main` module should compile.

Run the program with:

```
make run
```

And then go to `localhost:8000`. Make sure that you're NOT at the `Logoff` URL, i.e. `localhost:8000/#/logoff`.

Then refresh the page and click on the `Logoff` button and see what happens.

When I do it, all I get is a page with `Logoff` text, which is exactly what we should expect.

Now, we're ready to start building out the Logon Page.

But before we do, let's do a little housekeeping. First, we're going to move those styling Functions out of `Component.Page` into their own Module.

Create a File under `src` called `AppTheme.purs` and move `paperColor`, `themeColor` and `themeFont` from `Component.Page`.

You should have:

```
module AppTheme where

import Prelude

import CSS (CSS, fontFamily)
import CSS.Color (Color, rgb)
import CSS.Font (sansSerif)
import Data.NonEmpty ((:|))

paperColor :: Color
paperColor = rgb 0xd9 0xd9 0xd9

themeColor :: Color
themeColor = rgb 0x00 0x66 0x75

themeFont :: CSS
themeFont = fontFamily [ "Verdana" ] $ sansSerif :| []
```

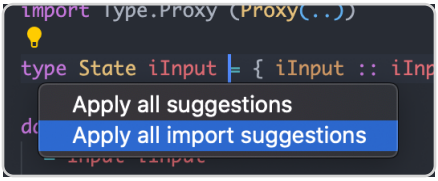
Now, clean up the redundant imports and add the following import to `Component.Page`:

```
import AppTheme (themeColor, paperColor, themeFont)
```

Quick side note: In VSCode, once your PureScript file compiles and if you have a bunch of redundant imports, you'll notice a Light Bulb Icon pop up near your cursor:



You can click the Light Bulb Icon and you'll get a menu:



Picking the `Apply all import suggestions` will clean up your redundant imports automatically. I haven't had great luck with the other menu items, but your mileage may vary.

Next, create a Folder under `src` called `CSS`. Then create a File under that called `Missing.purs`.

Put the following into that File:

```
module CSS.Missing where

import CSS.Flexbox (AlignContentValue, JustifyContentValue)
import CSS.Property (Value)
import CSS.String (fromString)

class SpaceEvenly a where
  spaceEvenly :: a

instance spaceEvenlyValue :: SpaceEvenly Value where
  spaceEvenly = fromString "space-evenly"
instance spaceEvenlyAlignContentValue :: SpaceEvenly AlignContentValue where
  spaceEvenly = fromString "space-evenly"
instance spaceEvenlyJustifyContentValue :: SpaceEvenly JustifyContentValue
where
  spaceEvenly = fromString "space-evenly"
```

This was all stolen from `halo`. We're going to want this for our Logon Page.

Now, return to `Component.Logon` and delete the following:

```
import CSS (color, white)
```

Now add the following imports:

```
import AppTheme (paperColor, themeColor, themeFont)
import CSS.Background (backgroundColor)
import CSS.Border (borderRadius)
import CSS.Box (boxShadow)
import CSS.Color (rgba, white)
import CSS.Common (center)
import CSS.Cursor (cursor, pointer)
import CSS.Display (display, flex)
import CSS.Flexbox (flexDirection, column, row, flexGrow, alignItems
                  , justifyContent)
import CSS.Font (FontWeight(..), color, fontSize , fontWeight)
import CSS.Geometry ( paddingTop, paddingBottom, paddingRight, paddingLeft
                    , width, height)
import CSS.Missing (spaceEvenly)
import CSS.Property (value)
import CSS.Size (rem, px, pct, vw)
import DOM.HTML.Indexed.InputType (InputType(..))
import Halogen.HTML.Events as HE
import Halogen.HTML.Properties as HP
import Image.BookCover (bookCover)
```

Next, replace the Action Definition with:

```
data Action
  = Logon
```

And now replace the `render` Function with the following:

```
  render :: State -> H.ComponentHTML Action Slots m
  render {} =
    HH.div [
      HC.style do
        display flex
        flexDirection column
        width $ vw 22.0
        height $ vw 25.0
        paddingTop $ vw 1.25
        paddingBottom $ vw 1.25
        paddingLeft $ vw 1.50
        paddingRight $ vw 1.50
        backgroundColor paperColor
```

```
        borderRadius (px 7.0) (px 7.0) (px 7.0) (px 7.0)
        boxShadow (px 10.0) (px 10.0) (px 20.0) (rgba 0 0 24 0.75)
    ]
    [ HH.div[
        HC.style do
          paddingRight $ vw 1.0
          display flex
          flexDirection column
          alignItems center
          justifyContent center
          flexGrow 10.0
      ]
      [ HH.img [
          HC.style do
            width $ pct 40.0
        , HP.src bookCover
        ]
      ]
    , HH.div [
        HC.style do
          display flex
          flexDirection column
          justifyContent spaceEvenly
          alignItems center
          flexGrow 3.0
      ]
      [ HH.div [
          HC.style do
            display flex
            flexDirection row
            alignItems center
            justifyContent center
            themeFont
        ]
        [ HH.label [
            HC.style do
              width (rem 6.0)
          ] [ HH.text "Username: " ]
        , HH.input [
            HC.style do
              backgroundColor paperColor
              width (vw 8.3)
              paddingLeft (rem 0.5)
              paddingRight (rem 0.5)
```

```
            fontSize (vw 1.0)
        ]
      ]
    , HH.div [
        HC.style do
          display flex
          flexDirection row
          alignItems center
          justifyContent center
          themeFont
      ]
      [ HH.label [
          HC.style do
            width (rem 6.0)
        ] [ HH.text "Password: " ]
      , HH.input [
          HP.type_ InputPassword
        , HC.style do
            backgroundColor paperColor
            width (vw 8.3)
            paddingLeft (rem 0.5)
            paddingRight (rem 0.5)
            fontSize (vw 1.0)
        ]
      ]
  ]
, HH.div [
    HC.style do
      display flex
      flexDirection row
      alignItems center
      justifyContent center
      paddingTop $ vw 0.65
  ]
  [ HH.button
    [
      HC.style do
        backgroundColor themeColor
        themeFont
        fontWeight $ FontWeight $ value "500"
        fontSize $ vw 1.0
        width (rem 20.0)
        height $ vw 3.0
        color white
```

```
            cursor pointer
        , HE.onClick $ const Logon
        ]
        [ HH.text "LOGON" ]
      ]
    ]
```

Most of this code is styling, but there are a few things to take note of, e.g. how we specify that an Input field is a Password:

```
import DOM.HTML.Indexed.InputType (InputType(..)) ❶

-- ABOVE CODE REDACTED
      , HH.input [
          HP.type_ InputPassword ❷
          , HC.style do
              backgroundColor paperColor
              width (vw 8.3)
              paddingLeft (rem 0.5)
              paddingRight (rem 0.5)
              fontSize (vw 1.0)
-- BELOW CODE REDACTED
```

❶ We need to import the `InputType` Type, which has Data Constructors for types of `input` fields.

❷ Set the `type` Property to `InputPassword`.

Another notable item is the button click handling:

```
import Halogen.HTML.Events as HE ❶

-- ABOVE CODE REDACTED
      [ HH.button
        [
          HC.style do
            backgroundColor themeColor
            themeFont
            fontWeight $ FontWeight $ value "500"
            fontSize $ vw 1.0
            width (rem 20.0)
            height $ vw 3.0
            color white
            cursor pointer
        , HE.onClick $ const Logon ❷
        ]
        [ HH.text "LOGON" ]
      ]
-- BELOW CODE REDACTED
```

❶ We import `Halogen.HTML.Events` to get the standard DOM events, e.g. `onChange`, `onBlur`, `onKeyUp`, etc.

❷ `onClick` takes a Function that consumes the `Event`. We don't care about the details of the click so we use `const` to dispose of it.

Run your application with:

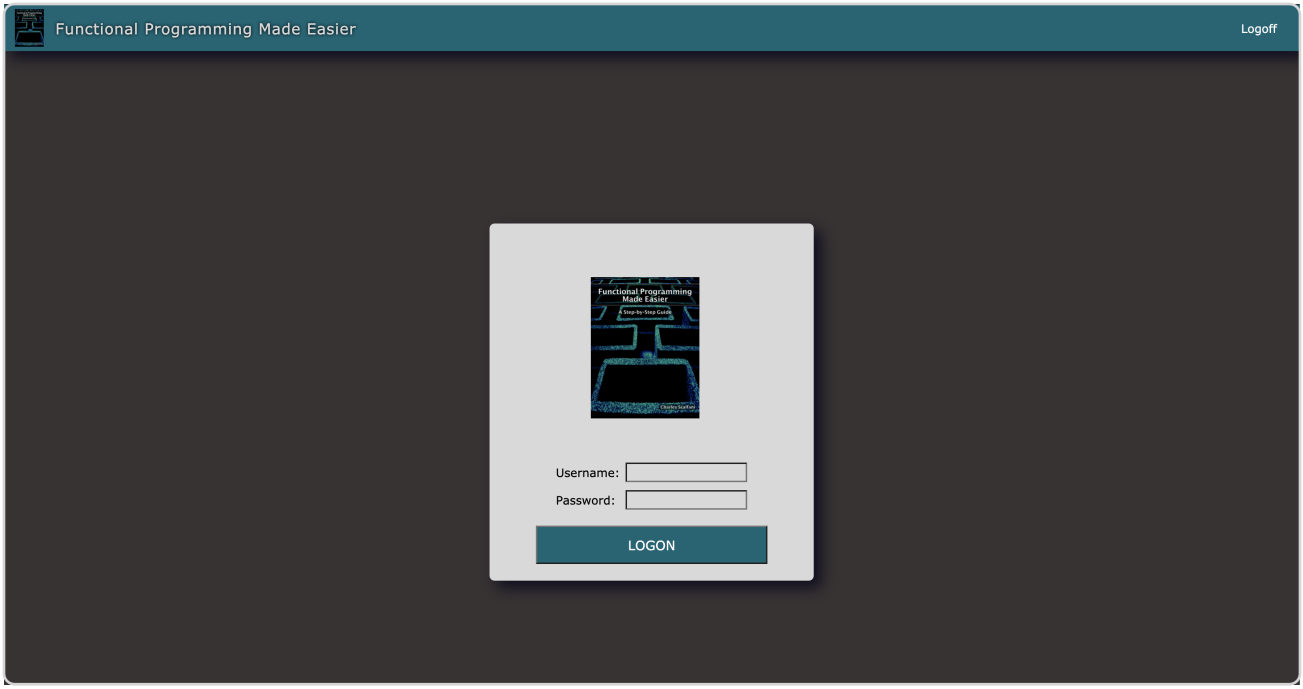
```
make run
```

And you'll get this error:

```
[error] Some of your project files import modules from packages that are not
in the direct dependencies of your project.
To fix this error add the following packages to the list of dependencies in
your config:
- dom-indexed
```

Update your `spago.dhall` file accordingly, run the program and navigate to `localhost:8000`. Make sure that you don't just refresh the browser since the URL is at the logoff screen.

When you do, you should see the following:



Now, we have some decisions to make. We want to have the user type in their Username and Password then click on the Logon Button. But what should we do if they haven't done so and they simply click the Logon Button?

There are a few things we could do. We could not enable the Logon Button when there is no data in the required fields. This will require us to monitor every change to Username and Password so we can enable the Logon Button as soon as every required field has data in it.

Another approach we could take is to give an error message when a required field is empty and they've clicked the Logon Button. The only question is how do we display this error message?

There are 2 ways that come to mind. First, we display an error message next to the offending fields, i.e. the required fields that are empty. Another way is to display a pop-up Dialog Box with the error message in it.

Which approach would you take and why do you think that your approach is better than the others? Think about this for a moment before reading my opinion.

My favorite approach is to disable the button and only display it when it's valid since I think it provides for a cleaner UI. The only downside to this approach is when the user doesn't understand what they need to enter to enable the button. But I think we don't run that risk here with this form since any user who has spent any time on the Internet has encountered many Logon Screens.

So, for this scenario, I'm going to choose that approach. However, if this were a more complex form, then I'd choose to add error messages next to the missing fields.

And while the Dialog Box isn't a terrible idea here, we're going to have plenty of opportunities for Dialog Boxes in this app.

Dialog boxes aren't my favorite solution, but sometimes they are necessary to reduce screen clutter and it'll

be good for us to see how to develop such a Component and how it will interact with our Page.

Now that we have a plan we're ready to complete the Logon Page.

The next thing we'll need to do to is keep track of the data in the fields in our `State` so that we have the values once the user clicks the Logon Button.

This is normally done with `onChange`, which is fired when the user exits a field and that field has been modified. But we're going to want to check on every change. For that, we're going to want to use `onValueInput`.

Add an `Action` called `Input` and then add `onValueInput` from `Halogen.HTML.Events` to both the Username and Password fields to trigger that new `Action`. Use the code for `HE.onClick` in this file as reference along with `Pursuit`.

Return here when you're ready to move on.

Here are those changes:

```
data Action
  = Logon
  | Input String ❶

-- ABOVE CODE REDACTED
        [ HH.label [
            HC.style do
              width (rem 6.0)
          ] [ HH.text "Username: " ]
        , HH.input [
            HE.onValueInput Input ❷
          , HC.style do
              backgroundColor paperColor
              width (vw 8.3)
              paddingLeft (rem 0.5)
              paddingRight (rem 0.5)
              fontSize (vw 1.0)
          ]
        ]
-- CODE REDACTED
        [ HH.label [
            HC.style do
              width (rem 6.0)
          ] [ HH.text "Password: " ]
        , HH.input [
            HP.type_ InputPassword
          , HE.onValueInput Input ❷
          , HC.style do
              backgroundColor paperColor
              width (vw 8.3)
              paddingLeft (rem 0.5)
              paddingRight (rem 0.5)
              fontSize (vw 1.0)
          ]
-- BELOW CODE REDACTED
```

❶ Added `Input` Data Constructor. The `String` Parameter is given to us by `onValueInput`.

❷ Added the Event Handler for both fields.

Note that `onValueInput` takes a Function of Type `String -> Maybe a`. This allows for us to reject the input in the `render` logic. We could also reject it `handleAction`.

Let's look at our component creation code:

```
component
  :: ∀ m
  .  MonadAff m
  => H.Component Query Input Output m
component = H.mkComponent
  { initialState: \_ -> { }
  , render
  , eval: H.mkEval H.defaultEval
  }
```

Can you see what we're missing?

We are using all of the defaults. We need to override them so we can add `handleAction`. Make that change to add ONLY `handleAction` using what we did in `halo` or in `Component.Page` as a example.

Return back here when you're finished coding.

Here is that change:

```
component
  :: ∀ m
  .  MonadAff m
  => H.Component Query Input Output m
component = H.mkComponent
  { initialState: \_ -> { }
  , render
  , eval: H.mkEval H.defaultEval
      { handleAction = handleAction -- COMPILER ERROR!!
      }
  }
```

No, surprise at the compiler error since we haven't written `handleAction` yet. We'll do that soon enough, but for now, we need to add the Value of the Username and Password fields into our State so that we can keep track of their values as they change.

Then when the user clicks on the Logon Button, we'll have them at the ready and can use them to call our backend server.

Update `State` to include this information and then continue reading once you're done.

Here is that change:

```
type State =
  { userName :: String
  , password :: String
  }
```

Next, update `initialState` to initialize these Values to an empty `String` then return here.

Here's that change:

```
component
  :: ∀ m
  .  MonadAff m
  => H.Component Query Input Output m
component = H.mkComponent
  { initialState: \_ -> { userName: "", password: "" }
  , render
  , eval: H.mkEval H.defaultEval
      { handleAction = handleAction -- COMPILER ERROR!!
      }
  }
```

Now write `handleAction` to maintain these newly added Values to `State`, but stub out the `Logon` Action for now. When you're finished, continue on.

Here's `handleAction`:

```
-- ABOVE CODE REDACTED
  where
  handleAction :: Action -> H.HalogenM State Action Slots Output m Unit
  handleAction = case _ of
    Input value -> ???
    Logon -> pure unit
```

I only got this far and then I realized that there's no way to know which field to update.

There are 2 ways to solve this that come to mind. First, we create a `UsernameInput` and a `PasswordInput`. But this approach doesn't scale well. Imagine a large form that has dozens and dozens of fields. We'd have to have a bunch of boilerplate in our Action Type, for example:

```
data Action
  = Field1Input
  | Field2Input
  | Field3Input
  | Field4Input
  | Field5Input
  | Field6Input
  | Field7Input
  | Field8Input
```

The other approach is to pass a Function to update the field in `State`. Then we can write pretty generic handling code and only require a single Data Constructor.

Comment out the following lines so we can get compiler errors that are useful as we add code:

```
  , eval: H.mkEval H.defaultEval
      -- { handleAction = handleAction
      -- }
-- CODE REDACTED
  where
  -- handleAction :: Action -> H.HalogenM State Action Slots Output m Unit
  -- handleAction = case _ of
  --   Input value ->
  --   Logon -> pure unit
```

Now, change the `Input` Data Constructor to take a Parameter that's a Function which updates `State` for the field in question. Return here once that's done.

Here's that change:

```
data Action
  = Logon
  | Input (State -> State) ❶
```

❶ We will pass the input `String` and the current `State` and produce a new `State` with the appropriate field updated. Notice how the `String` Parameter is BEFORE the first Parameter. That's because we want to Partially Apply this Function with the input `String` leaving us a Function that will modify our `State`, i.e. `State -> State`.

Now, change the 2 places we call `HE.onValueInput` to add this new Parameter to `Input`. Once you're done, continue on.

Here's that change:

```
-- ABOVE CODE REDACTED
        [ HH.label [
            HC.style do
              width (rem 6.0)
          ] [ HH.text "Username: " ]
        , HH.input [
            HE.onValueInput
              $ Input <<< \s -> _ { userName = s } ❶ ❷
          , HC.style do
              backgroundColor paperColor
              width (vw 8.3)
              paddingLeft (rem 0.5)
              paddingRight (rem 0.5)
              fontSize (vw 1.0)
          ]
        ]
-- CODE REDACTED
        [ HH.label [
            HC.style do
              width (rem 6.0)
          ] [ HH.text "Password: " ]
        , HH.input [
            HP.type_ InputPassword
          , HE.onValueInput
              $ Input <<< \s -> _ { password = s } ❶ ❸
          , HC.style do
              backgroundColor paperColor
              width (vw 8.3)
              paddingLeft (rem 0.5)
              paddingRight (rem 0.5)
              fontSize (vw 1.0)
          ]
-- BELOW CODE REDACTED
```

❶ I wrote the Type Signature for `Input` to be `String -> State -> State` so I could code these Functions using a wildcard for the `State`.

❷ Update `userName`.

❸ Update `password`.

This should compile.

Now, uncomment:

```
component = H.mkComponent
  { initialState: \_ -> { userName: "", password: "" }
  , render
  , eval: H.mkEval H.defaultEval
      -- { handleAction = handleAction ❶
      -- }
  }
```

❶ Uncomment these lines.

And then uncomment and finish `handleAction`. Return here when you're ready to move forward.

Here's the completed `handleAction`:

```
-- ABOVE CODE REDACTED
  where
  handleAction :: Action -> H.HalogenM State Action Slots Output m Unit
  handleAction = case _ of
    Input f -> H.modify_ f ❶
    Logon -> pure unit
```

❶ `H.modify_` will update our `State`. It calls a Function of Type `State -> State`.

Now modify `render` to disable the Logon Button if and only if either `userName` or `password` are empty. Search `Pursuit` for `disabled`. There will be many, but we want the Halogen version. Keep in mind that we're looking for a way to set the Properties of the button.

Continue reading once you've finished.

Here is the first part of the change to `render`:

```
  render :: State -> H.ComponentHTML Action Slots m
  render { userName, password } = ❶
-- BELOW CODE REDACTED
```

❶ Destructure `State` to get `userName` and `password`.

And here's the other change to `render`:

```
-- ABOVE CODE REDACTED
      [ HH.button
        [
          HC.style do
            backgroundColor themeColor
            themeFont
            fontWeight $ FontWeight $ value "500"
            fontSize $ vw 1.0
            width (rem 20.0)
            height $ vw 3.0
            color white
            cursor pointer
        , HE.onClick $ const Logon
        , HP.disabled logonDisabled ❶
        ]
        [ HH.text "LOGON" ]
      ]
    ]
    where logonDisabled = userName == "" || password == "" ❷
```

❶ Conditionally disable button.

❷ Button should be disabled if either `userName` or `password` are empty.

Now that I typed this, I realize that we should check not for an empty `String` but for a blank string, i.e. a `String` with empty spaces is equivalent to an empty `String` for this logic.

Make that change to `logonDisabled` then return here.

Here's that change:

```
import Data.String.Common (trim) ❶

-- ABOVE CODE REDACTED
    where logonDisabled = trim userName == "" || trim password == "" ❷
```

❶ Added this import. `trim` will trim whitespaces and line termination characters from the beginning and ending of a `String`.

❷ `trim` both `Strings` before checking for empty.

Now, run your application and see how our new code works.

When I test the application, I can't tell when our button is disabled. I checked the state of our button in the Developer Console and the button is disabled, which means our code is working, but it's not obvious to the

user.

Change the `render` code to display the text in gray when the button is disabled. Then rerun the app and see how you like that change.

Here's that change:

```
import CSS.Color (rgba, white, gray) ❶

-- ABOVE CODE REDACTED
      [ HH.button
        [
          HC.style do
            backgroundColor themeColor
            themeFont
            fontWeight $ FontWeight $ value "500"
            fontSize $ vw 1.0
            width (rem 20.0)
            height $ vw 3.0
            color if logonDisabled then gray else white ❷
            cursor pointer
        , HE.onClick $ const Logon
        , HP.disabled logonDisabled
        ]
        [ HH.text "LOGON" ]
-- BELOW CODE REDACTED
```

❶ Added `gray` to the import list.

❷ Conditionally set the color.

When I run this and use it as a user, it's a bit better, but I notice that the cursor still changes to a `pointer` when the button is disabled giving me the wrong impression that I can click it.

Change the `render` code to have the `cursor` be a pointer over the button ONLY when it's NOT disabled. When it is disabled, pick a more appropriate `cursor` type from `CSS.Cursor`.

Run the app and see how you like your change before reading on.

Here's that change:

```
import CSS.Cursor (cursor, pointer, notAllowed) ❶

-- ABOVE CODE REDACTED
      [ HH.button
        [
          HC.style do
            backgroundColor themeColor
            themeFont
            fontWeight $ FontWeight $ value "500"
            fontSize $ vw 1.0
            width (rem 20.0)
            height $ vw 3.0
            color if logonDisabled then gray else white
            cursor if logonDisabled then notAllowed else pointer ❷
        , HE.onClick $ const Logon
        , HP.disabled logonDisabled
        ]
        [ HH.text "LOGON" ]
-- BELOW CODE REDACTED
```

❶ Added `notAllowed` to the import list.

❷ Conditionally set `cursor` type.

This change makes our UI much more obvious. Give it a try.

Oh no, another build error:

```
[error] Some of your project files import modules from packages that are not
in the direct dependencies of your project.
To fix this error add the following packages to the list of dependencies in
your config:
- strings
```

Add this to `spago.dhall` and then try to run the app again.

The button should be grayed out and when you hover your cursor over it, you should see the `notAllowed` cursor.

30.9. Calling the Backend

Next thing to getting the Logon Page to work is to call the Backend once the user clicks on the Logon Button.

But we have a problem. The API code is in the `server` and not available to our `client` program.

To fix that, we're going to change the `sources` setting in `spago.dhall` file in the `client` to reference our the `Data/Api` and the `Entity` directories in the `server`:

```
, sources = [
  "src/**/*.purs"
  , "../server/src/Data/Api/**/*.purs" ❶
  , "../server/src/Entity/**/*.purs" ❷
  , "test/**/*.purs"
  ]
```

❶ `"../server/src/Data/Api/*/.purs"` was added to include the API code.

❷ `"../server/src/Entity/*/.purs"` was added to include the Entity code, which is needed by the API code.

And now rebuild the `client` to compile these addition source modules:

```
make build
```

This gives us the following error:

```
[error] Some of your project files import modules from packages that are not
in the direct dependencies of your project.
To fix this error add the following packages to the list of dependencies in
your config:
- foreign-generic
```

Changing `spago.dhall` to include this package will build successfully.

When I originally did this, I forgot the `Entity` directory. When I built the `client` without it, I got the following error:

```
  in module Data.Api.CreateUser
  at ../server/src/Data/Api/CreateUser.purs:5:1 - 5:29 (line 5, column 1 -
line 5, column 29)

    Module Entity.User was not found.
    Make sure the source file exists, and that it has been provided as an
input to the compiler.
```

This told me that the `Data.Api` code referred to `Entity.User` and that I needed to add the `Entity` directory to my `spago.dhall` under the `sources` setting.

Now that we can access the API code in the server, we're ready to call the server's `Logon` API. But before we do, we're going to need to install the Ajax package:

```
npx spago install affjax
make build
```

We're going to ignore the warning for now:

```
[warn] None of your project files import modules from some projects that are
in the direct dependencies of your project.
These dependencies are unused. To fix this warning, remove the following
packages from the list of dependencies in your config:
- affjax
```

Let's return to `Logon.purs` and start with `handleAction`, which currently is as follows:

```
-- ABOVE CODE REDACTED
  where
  handleAction :: Action -> H.HalogenM State Action Slots Output m Unit
  handleAction = case _ of
    Input f -> H.modify_ f
    Logon -> pure unit
```

Add a `do` block under the `Logon` case and leave the `pure unit`:

```
-- ABOVE CODE REDACTED
  where
  handleAction :: Action -> H.HalogenM State Action Slots Output m Unit
  handleAction = case _ of
    Input f -> H.modify_ f
    Logon -> do
      pure unit
```

Now, just add code to get the `userName` and `password` then return here.

Here's that call:

```
-- ABOVE CODE REDACTED
  where
  handleAction :: Action -> H.HalogenM State Action Slots Output m Unit
  handleAction = case _ of
    Input f -> H.modify_ f
    Logon -> do
      { userName, password } <- H.get ❶
      pure unit
```

❶ Get `State` destructuring to get `userName` and `password`.

Next, make the Ajax call to the `server` to attempt to `Logon`. If you forget how to do this, refresh your memory by looking at the `test` Function in `Ch25b.purs`.

Return here when that's complete.

Here's that addition:

```
import Affjax as Ajax ❶
import Affjax.ResponseFormat as ResponseFormat
import Affjax.RequestBody as RequestBody
import Data.Api.Logon (LogonRequest(..)
                     , LogonResponse(..), LogonResults(..)) ❷
import Foreign.Generic (encodeJSON)

-- ABOVE CODE REDACTED
  where
  handleAction :: Action -> H.HalogenM State Action Slots Output m Unit
  handleAction = case _ of
    Input f -> H.modify_ f
    Logon -> do
      { userName, password } <- H.get
      ajaxResult <- H.liftAff $ Ajax.post ResponseFormat.string ❸ ❹
        "http://localhost:3000/"
        $ Just $ RequestBody.String ❺
        $ encodeJSON $ LogonRequest { userName, password } ❻
      pure unit
```

❶ Added these imports.

❷ There's more imported here than we need at the moment, but we'll need them eventually.

❸ Note that `ajaxResult` is an `Either`.

❹ The `ResponseFormat` is a `String` since we're going to use `decodeJSON` on it. This is unlike `test` in

`Ch25.purs`. That code uses `decodeJson` from `Argonaut`, which wants a `JSON`.

❺ The `RequestBody` is also a `String`, which is the output Type of `encodeJSON`.

❻ We're encoding `LogonRequest`.

Now at this point, we have an `Either`. We could start using a `case` expression but then we're going to need to decode the `String` we got back from a successful Ajax call, which will also be an `Either` requiring yet another `case` expression.

Instead, let's code in the `Either` Monad for just a minute:

```
-- ABOVE CODE REDACTED
  where
  handleAction :: Action -> H.HalogenM State Action Slots Output m Unit
  handleAction = case _ of
    Input f -> H.modify_ f
    Logon -> do
      { userName, password } <- H.get
      ajaxResult <- H.liftAff $ Ajax.post ResponseFormat.string
        "http://localhost:3000/"
        $ Just $ RequestBody.String
        $ encodeJSON $ LogonRequest { userName, password }
      let logonResponse = do ❶
        ???
      pure unit
```

❶ `logonResponse` will be an `Either` since the `do` block will be running in the `Either` Monad.

Now, write code to extract the `body` from the `Response`. Remember that we're coding in the `Either` Monad. That means that we can code as if we're coding just the Happy Path. So, you can get the `Response` from the `ajaxResult` and not have to check for the error case.

Put that line of code where the `???` is and return back here when you're ready to move on.

Here's that addition:

```
import Data.Bifunctor (lmap) ❶

-- ABOVE CODE REDACTED
  where
  handleAction :: Action -> H.HalogenM State Action Slots Output m Unit
  handleAction = case _ of
    Input f -> H.modify_ f
    Logon -> do
      { userName, password } <- H.get
      ajaxResult <- H.liftAff $ Ajax.post ResponseFormat.string
        "http://localhost:3000/"
        $ Just $ RequestBody.String
        $ encodeJSON $ LogonRequest { userName, password }
      let logonResponse = do
            { body } <- lmap Ajax.printError ajaxResult ❷ ❸ ❹
        ???
      pure unit
```

❶ Added this import.

❷ Mapping over left side of `ajaxResult` to convert the Ajax `Error` to a `String`. We're going to want to display this error at some point in our code.

❸ Remember, we're coding in the `Either` Monad. If `ajaxResult` is a `Left`, this code will Short-circuit and return immediately. We're writing our code as the Happy Path here thanks to all the error checking code in `Either's` implemenation of `bind`.

❹ Don't forget to indent enough to get past the Variable name.

Next, decode the `body` of the `Response`. If you cannot remember how to do that, look at `processAjaxResult` in `Ch25a.purs` to refresh your memory.

Continue reading once you're done.

Here's that addition:

```
import Control.Monad.Except (runExcept) ❶
import Foreign.Generic (encodeJSON, decodeJSON) ❷

-- ABOVE CODE REDACTED
  where
  handleAction :: Action -> H.HalogenM State Action Slots Output m Unit
  handleAction = case _ of
    Input f -> H.modify_ f
    Logon -> do
      { userName, password } <- H.get
      ajaxResult <- H.liftAff $ Ajax.post ResponseFormat.string
        "http://localhost:3000/"
        $ Just $ RequestBody.String
        $ encodeJSON $ LogonRequest { userName, password }
      let logonResponse = do
            { body } <- lmap Ajax.printError ajaxResult
            response <- lmap show $ runExcept ❸
              $ decodeJSON body :: _ LogonResponse ❹ ❺
        ???
      pure unit
```

❶ Added this import.

❷ Added `decodeJSON` to the import list.

❸ Mapping over the left side to convert the decode `Error` to `String` for the same reason we did with the Ajax `Error`.

❹ Don't forget that we need to explcitly specify the Type that we're decoding. There's no way for the compiler to figure this out since `decodeJSON` is consuming a `String`.

❺ The result of `decodeJSON` is `F a`, where `F` is Type Alias for Foreign Errors. We're not specifying the `F` here to avoid an unnecessary import.

And now we can simply return the response as the final result of this computation. Add that and then keep reading.

Here's that addition:

```
-- ABOVE CODE REDACTED
  where
  handleAction :: Action -> H.HalogenM State Action Slots Output m Unit
  handleAction = case _ of
    Input f -> H.modify_ f
    Logon -> do
      { userName, password } <- H.get
      ajaxResult <- H.liftAff $ Ajax.post ResponseFormat.string
        "http://localhost:3000/"
        $ Just $ RequestBody.String
        $ encodeJSON $ LogonRequest { userName, password }
      let logonResponse = do
            { body } <- lmap Ajax.printError ajaxResult
            response <- lmap show $ runExcept
              $ decodeJSON body :: _ LogonResponse
            pure response ❶
      ???
      pure unit
```

❶ Return `response`.

We can reduce this by not naming the `response` at all and just returning from the last line we coded. Make that improvement then read on.

Here's that improvement:

```
-- ABOVE CODE REDACTED
  where
  handleAction :: Action -> H.HalogenM State Action Slots Output m Unit
  handleAction = case _ of
    Input f -> H.modify_ f
    Logon -> do
      { userName, password } <- H.get
      ajaxResult <- H.liftAff $ Ajax.post ResponseFormat.string
        "http://localhost:3000/"
        $ Just $ RequestBody.String
        $ encodeJSON $ LogonRequest { userName, password }
      let logonResponse = do
            { body } <- lmap Ajax.printError ajaxResult
            lmap show $ runExcept $ decodeJSON body :: _ LogonResponse
      ???
      pure unit
```

Now, we're going to have to use a `case` expression, but at least it'll be only one.

In the error case, we're going to pop-up the standard Browser `alert` dialog box for now. Eventually, we'll build a Dialog Component and we'll use that instead.

But for now, add the following to your code:

```
import Web.HTML.Window (alert) ❶
import Web.HTML (window)

-- ABOVE CODE REDACTED
  where
  handleAction :: Action -> H.HalogenM State Action Slots Output m Unit
  handleAction = case _ of
    Input f -> H.modify_ f
    Logon -> do
      { userName, password } <- H.get
      ajaxResult <- H.liftAff $ Ajax.post ResponseFormat.string
        "http://localhost:3000/"
        $ Just $ RequestBody.String
        $ encodeJSON $ LogonRequest { userName, password }
      let logonResponse = do
            { body } <- lmap Ajax.printError ajaxResult
            lmap show $ runExcept $ decodeJSON body :: _ LogonResponse
      ???
      pure unit
    where ❷
    alertError :: String -> H.HalogenM State Action Slots Output m Unit
    alertError msg = H.liftEffect $ window >>= alert msg ❸
```

❶ Add these imports.

❷ Add this `where` and the `alertError` Function. Make sure you indent it properly. It's supposed to be the `where` for `handleAction`.

❸ Since `alert` runs in `Effect`, we use `H.liftEffect` to lift it into `HalogenM`.

Now, add the `case` expression and just the `Left` case code that calls `alertError` before reading any further.

Here's that addition:

```
import Data.Either (Either(..)) ❶

-- ABOVE CODE REDACTED
  where
  handleAction :: Action -> H.HalogenM State Action Slots Output m Unit
  handleAction = case _ of
    Input f -> H.modify_ f
    Logon -> do
      { userName, password } <- H.get
      ajaxResult <- H.liftAff $ Ajax.post ResponseFormat.string
        "http://localhost:3000/"
        $ Just $ RequestBody.String
        $ encodeJSON $ LogonRequest { userName, password }
      let logonResponse = do
            { body } <- lmap Ajax.printError ajaxResult
            lmap show $ runExcept $ decodeJSON body :: _ LogonResponse
      case logonResponse of ❷
        Left err -> alertError err ❸
        ???
      pure unit
    where
    alertError :: String -> H.HalogenM State Action Slots Output m Unit
    alertError msg = H.liftEffect $ window >>= alert msg
```

❶ Added this import.

❷ Remember that `logonResponse` is an `Either`.

❸ We're made sure that `err` is a `String`. This is what the 2 `lmaps` in the computation for `logonResponse` are doing.

Now, there are 2 cases for `Right`, i.e. the failure case and the success case. Instead of adding another `case` expression, use Pattern Matching to explicitly match the `Right` with the failure case of the API call.

Look back at the `Logon` API code in the `server` to refresh your memory. For the failure case, simply alert with "Invalid Logon Credentials". Remember, we don't want to help our a hacker by giving any more information than this.

Then, once you're done, continue reading.

Here's the API failure case:

```
-- ABOVE CODE REDACTED
  where
  handleAction :: Action -> H.HalogenM State Action Slots Output m Unit
  handleAction = case _ of
    Input f -> H.modify_ f
    Logon -> do
      { userName, password } <- H.get
      ajaxResult <- H.liftAff $ Ajax.post ResponseFormat.string
        "http://localhost:3000/"
        $ Just $ RequestBody.String
        $ encodeJSON $ LogonRequest { userName, password }
      let logonResponse = do
            { body } <- lmap Ajax.printError ajaxResult
            lmap show $ runExcept $ decodeJSON body :: _ LogonResponse
      case logonResponse of
        Left err -> alertError err
        Right (LogonResponse LogonResultsFailure) -> ❶
          alertError "Invalid Logon Credentials" ❷
        ???
      pure unit
    where
    alertError :: String -> H.HalogenM State Action Slots Output m Unit
    alertError msg = H.liftEffect $ window >>= alert msg
```

❶ Pattern match `Right`, i.e. the successful Ajax call, with the unsuccessful API response.

❷ Let the user know they provided invalid credentials.

And finally, we need to handle the success case. First, just write the Pattern Match, i.e. the left side of the `->`. Then return here.

Here's just that part:

```
-- ABOVE CODE REDACTED
      case logonResponse of
        Left err -> alertError err
        Right (LogonResponse LogonResultsFailure) ->
          alertError "Invalid Logon Credentials"
        Right (LogonResponse (LogonResultsSuccess
          { authToken, mustChangePassword })) -> do ❶ ❷
            ???
      pure unit
-- BELOW CODE REDACTED
```

❶ Destructure to get the `authToken` and `mustChangePassword`.

❷ Started a `do` block that will run in `HalogenM`. Probably. Hard to tell at this point.

We need to save the `authToken` in `userRef`. We can get `userRef` from our `ReaderT` using `ask`. And we can use `write` from `Effect.Ref` to update the `userRef`.

Write both of those lines of code before reading on.

Here are those additions:

```
import Control.Monad.Reader.Class (class MonadAsk, ask) ❶
import Data.Maybe (Maybe(..))
import Effect.Ref as Ref
import Env (Env)

component
  :: ∀ m
  .  MonadAff m
  => MonadAsk Env m ❷
  => H.Component Query Input Output m
-- BELOW CODE REDACTED

  handleAction :: Action -> H.HalogenM State Action Slots Output m Unit
  handleAction = case _ of
-- CODE REDACTED
      case logonResponse of
        Left err -> alertError err
        Right (LogonResponse LogonResultsFailure) ->
          alertError "Invalid Logon Credentials"
        Right (LogonResponse (LogonResultsSuccess
          { authToken, mustChangePassword })) -> do
            { userRef } <- ask ❸
            H.liftEffect $ Ref.write (Just { authToken }) userRef ❹ ❺
            ???
      pure unit
-- BELOW CODE REDACTED
```

❶ Added these imports.

❷ Added `MonadAsk` Constraint.

❸ Get `Env`, i.e. the `ReaderT` environment, and destructure it to get the `userRef`.

❹ Write `authToken` to `userRef`. Don't forget it's a `Maybe`.

❺ We need to `liftEffect` this since the `do` block is `HalogenM` and `Ref.write` is in `Effect`.

And finally, if `mustChangePassword` is `true` then we want to navigate to `ChangePassword` otherwise we'll navigate to `Users`. But we want to decouple the Logon Page from knowing anything about our `Routes`.

That's why we made `Capability.LogonRoute`. With the idea that maybe our Logon Page would exist in a library.

Look at the `LogonRoute` Instance implementation that we did in `AppM.purs` to help you add this final step before reading on.

Here's the final piece:

```
import Capability.LogonRoute (class LogonRoute, PasswordType(..) ❶
                             , logonRoute)
import Capability.Navigate (class Navigate, navigate)

component
  :: ∀ m route ❷
  .  MonadAff m
  => MonadAsk Env m
  => Navigate m route ❸
  => LogonRoute m route ❹
  => H.Component Query Input Output m
component = H.mkComponent
-- BELOW CODE REDACTED

  handleAction :: Action -> H.HalogenM State Action Slots Output m Unit
  handleAction = case _ of
-- CODE REDACTED
      case logonResponse of
        Left err -> alertError err
        Right (LogonResponse LogonResultsFailure) ->
          alertError "Invalid Logon Credentials"
        Right (LogonResponse (LogonResultsSuccess
          { authToken, mustChangePassword })) -> do
            { userRef } <- ask
            H.liftEffect $ Ref.write (Just { authToken }) userRef
            if mustChangePassword then
              -- COMPILER ERROR!!
              navigate =<< logonRoute PasswordTemporary ❺ ❻
            else navigate =<< logonRoute PasswordPermanent ❼ ❽
-- BELOW CODE REDACTED
```

❶ Added these imports.

❷ Added `route`.

❸ Added `Navigate` Capability.

❹ Added `LogonRoute` Capability.

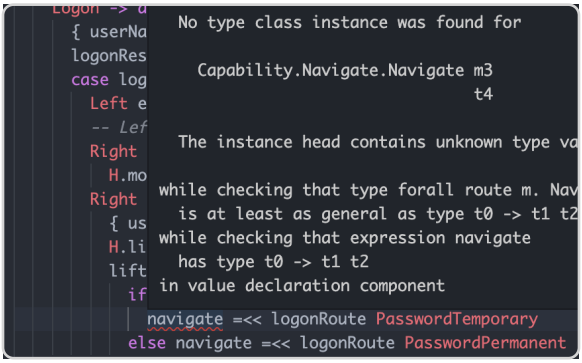
❺ If `mustChangePassword` then we route to where we should for a temporary password, which is our Change Password Page.

❻ We're using `=<<` since both `navigate` and `logonRoute` are Monadic Functions.

❼ Otherwise we navigate to where we should route for permanent passwords, i.e. the users page.

❽ Removed `pure unit`.

Here's our compiler error:



The compiler cannot figure out what the second Type is for `Navigate`. This can be seen by the `t4` in the error message. It can, however, figure out the first Type has a `Navigate` Instance since it knows of the `HalogenM` instance for `Navigate`, which constrains `m` to have a `Navigate` instances as well.

But there's nothing of Type `route` for it to latch onto and there's no way to get from `m` to `route`.

We COULD fix that by telling the compiler what the second Type is:

```
-- ABOVE CODE REDACTED
            if mustChangePassword then
              -- NOT IDEAL!!
              navigate =<< (logonRoute PasswordTemporary :: _ route)
              -- NOT IDEAL!!
            else navigate =<< (logonRoute PasswordPermanent :: _ route)
-- BELOW CODE REDACTED
```

But what would be even better is if we didn't have to do this. And we can do that by create a Functional Dependency between the `m` that the compiler can figure out and the `route` it cannot:

```
class Monad m <= Navigate m route | m -> route where ❶
  navigate :: route -> m Unit
```

❶ Added `| m -> route`. This tells the compiler that if you can figure out `m` then it'll imply one and only

one `route`.

Make this change so we don't need to provide the explicit Type Annotations.

Let's also make the same change to `LogonRoute` just in case we use `logonRoute` without `navigate`. We probably aren't going to do so in this Application, but this is still a good practice:

```
class Monad m <= LogonRoute m route | m -> route where ❶
  logonRoute :: PasswordType -> m route
```

❶ Added Functional Dependency here too, just in case we have a similar problem as we did with `Navigate`.

Now, our compiler error is gone.

```
  handleAction :: Action -> H.HalogenM State Action Slots Output m Unit
  handleAction = case _ of
-- CODE REDACTED
      case logonResponse of
        Left err -> alertError err
        Right (LogonResponse LogonResultsFailure) ->
          alertError "Invalid Logon Credentials"
        Right (LogonResponse (LogonResultsSuccess
          { authToken, mustChangePassword })) -> do
            { userRef } <- ask
            H.liftEffect $ Ref.write (Just { authToken }) userRef
            if mustChangePassword then
              navigate =<< logonRoute PasswordTemporary
            else navigate =<< logonRoute PasswordPermanent
-- BELOW CODE REDACTED
```

Step back from this code change and see if you can see how we can improve the `if` expression before reading on.

The first thing I can see is that we can factor out `navigate`:

```
  handleAction :: Action -> H.HalogenM State Action Slots Output m Unit
  handleAction = case _ of
-- CODE REDACTED
      case logonResponse of
        Left err -> alertError err
        Right (LogonResponse LogonResultsFailure) ->
          alertError "Invalid Logon Credentials"
        Right (LogonResponse (LogonResultsSuccess
          { authToken, mustChangePassword })) -> do
            { userRef } <- ask
            H.liftEffect $ Ref.write (Just { authToken }) userRef
            navigate =<< if mustChangePassword then
              logonRoute PasswordTemporary
            else logonRoute PasswordPermanent
-- BELOW CODE REDACTED
```

It would also be nice if we could also factor out `logonRoute`:

```
-- ABOVE CODE REDACTED
            navigate <=< logonRoute
              $ if mustChangePassword then PasswordTemporary
                else PasswordPermanent
-- BELOW CODE REDACTED
```

Notice how the `bind` changed to Kleisli Composition using the flipped `fish` operator, `<=<`. We could have kept the `bind`, but Function Composition is preferable to Function Application.

Make this change to your code.

Next, let's log to the Browser's console every time the user logs on successfully. But, we're not going to use `log`. We're going to use the `Log` Capability:

```
import Capability.Log (class Log, LogLevel(..), log, logEntry) ❶

component
  :: ∀ m route
  .  MonadAff m
  => MonadAsk Env m
  => Navigate m route
  => LogonRoute m route
  => Log m ❷
  => H.Component Query Input Output m
-- BELOW CODE REDACTED

  handleAction :: Action -> H.HalogenM State Action Slots Output m Unit
  handleAction = case _ of
-- CODE REDACTED
        Right (LogonResponse (LogonResultsSuccess
          { authToken, mustChangePassword })) -> do
            log =<< logEntry Info "User logged on" ❸ ❹
            { userRef } <- ask
            H.liftEffect $ Ref.write (Just { authToken }) userRef
            navigate <=< logonRoute
              $ if mustChangePassword then PasswordTemporary
                else PasswordPermanent
-- BELOW CODE REDACTED
```

❶ Added this import.

❷ Added this Constraint, i.e. the Logging Capability.

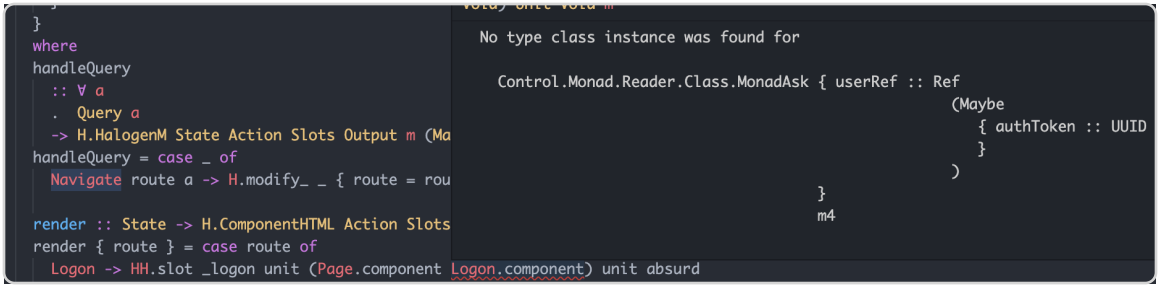
❸ Added logging on success case.

❹ We're using `=<<` since both `log` and `logEntry` are Monadic Functions.

We're ready to test. You can run it with:

```
make run
```

When I do this, I get an error in the Router Component:



This is because the Logon Component's `m` has a `MonadAsk` Constraint on `m` but the Router Component does not.

We can add this easily to `Component.Router`:

```
import Control.Monad.Reader.Class (class MonadAsk) ❶
import Env (Env)

component
  :: ∀ m
  .  MonadAff m
  => MonadAsk Env m ❷
  => H.Component Query Input Output m
-- BELOW CODE REDACTED
```

❶ Added these imports.

❷ Added `MonadAsk` Constraint.

It's clear that every single Constraint that any of our Pages have, will bubble up to require that the `Route` Component also constrains `m` accordingly.

That means we'll need to add `Log` and `LogonRoute` since the Logon Page has those Constraints:

```
import Capability.Log (class Log) ❶
import Capability.LogonRoute (class LogonRoute)

component
  :: ∀ m
  .  MonadAff m
  => MonadAsk Env m
  => Navigate m Route ❸
  => LogonRoute m Route ❷ ❸
  => Log m ❷
  => H.Component Query Input Output m
-- BELOW CODE REDACTED
```

❶ Added these imports.

❷ Added these constraints.

❸ Notice how we're using a Monomorphic Type, `Route`, here. That's because our Router Component is using one and only one `Route`, whereas, the Logon Component is written to work with any `route` allowing us to place it in a library.

Now run with:

```
make run
```

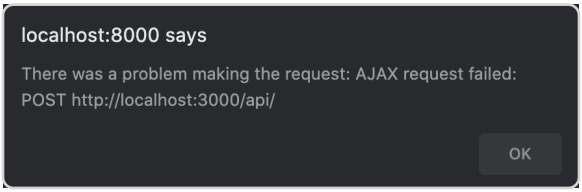
And now it compiles but is missing a direct dependency:

```
[error] Some of your project files import modules from packages that are not
in the direct dependencies of your project.
To fix this error add the following packages to the list of dependencies in
your config:
- bifunctors
- web-html
```

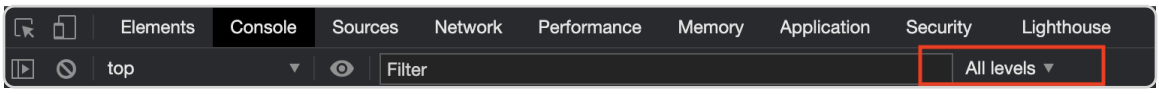
Add these to `spago.dhall` and run the app again.

Next, refresh the browser and then logon using `admin` for **both** the `userName` and `password`.

You should get the alert dialog box:



Open up the console in the Browser and check the error message. Make sure that you're not filtering error message and that all messages are being shown:



If you get:

```
Failed to load resource: net::ERR_CONNECTION_REFUSED
```

then you forgot to run your server. So do that now and retry to logon.

If you forgot to run your server and now it's running, you will see the same alert dialog box. And this time you should see the following in your Console:

```
Access to XMLHttpRequest at 'http://localhost:3000/' from origin
'http://localhost:8000' has been blocked by CORS policy: No 'Access-Control-
Allow-Origin' header is present on the requested resource.
```

The are 2 ways to fix this. The first way is to support `CORS` in the `server` and the second is to have our API Server serve up our client files.

Since the second is the way we're probably going to want to run in production, we'll make that change.

We'll look at both solutions, but we'll only implement the latter.

30.9.1. CORS Solution

`CORS` stands for ***Cross-Origin Resource Sharing***. This is a security issue where the user is logged on to one website, but the browser code is accessing another. Generally, this can be a very bad thing since the user is unaware they are interacting with a different website.

`localhost:3000` is considered a different website than `localhost:8000`. The ports are different, which makes the Browser consider them different sites.

This can be fixed by handling the `OPTIONS` pre-flight request that the Browser will make to the `server` before issuing the real request.

`OPTIONS` is an HTTP Verb just like `GET` and `POST`. This gives us an opportunity to let the Browser know which websites we'll accept requests from. We can respond by placing that information in the HTTP Headers, viz. `Access-Control-Allow-Origin`, `Access-Control-Allow-Headers` and `Access-Control-Allow-Methods` to name a few.

One of the many oddities with `CORS` is that it doesn't require a pre-flight `OPTIONS` request for a `POST` unless `Content-Type` is `application/json`. This appears to be an historical issue to maintain behavior similar to before `CORS`.

We could force a `CORS` pre-flight for `POST` requests, by making the following changes in our `Logon` code in the `client`:

```
import Affjax.RequestHeader (RequestHeader(..)) ❶
import Data.HTTP.Method (Method(..))

-- ABOVE CODE REDACTED
      ajaxResult <- H.liftAff $ Ajax.request Ajax.defaultRequest ❷ ❸
        { content = Just $ RequestBody.String
            $ encodeJSON $ LogonRequest { userName, password }
        , headers = [ RequestHeader "Content-Type" "application/json" ] ❹
        , method = Left POST
        , url = "http://localhost:3000/"
        , responseFormat = ResponseFormat.string
        }
      -- ajaxResult <- H.liftAff $ Ajax.post ResponseFormat.string
      --   "http://localhost:3000/"
      --   $ Just $ RequestBody.String
      --   $ encodeJSON $ LogonRequest { userName, password }
-- BELOW CODE REDACTED
```

❶ Added these imports.

❷ Using `Ajax.request` instead of the simplified helper `Ajax.post`.

❸ Using Record Update syntax here to modify the `Ajax.defaultRequest`.

❹ Setting the header `Content-Type` to `application/json` to force the Browser to pre-flight the `POST` method.

Turns out that this isn't necessary, but seeing this alternative way to make a request is useful.

All we need to do is set the `Access-Control-Allow-Origin` to the appropriate value upon responding to `POST` requests and the Browser will be happy.

Here's the code change to make in the `server` IF we were going to do that:

```
import Data.Either (Either(..), hush) ❶

router :: HandlerEnv -> Request -> ResponseM
router env req@{ body, method }
  | method == HTTPure.Post =
      let handlers = apiHandlers <#> (_ $ body) in
      case hush $ oneOf handlers of
        Nothing -> HTTPure.badRequest body
        Just reader -> runReaderT reader env
          >>= \res@{ headers } -> pure $ res { headers = headers <>
                    HTTPure.header "Access-Control-Allow-Origin" "*" } ❷
  | otherwise = HTTPure.notFound
```

❶ Added `hush` to the import list.

❷ Take the `Aff Response` we get back from `runReaderT`, i.e. running our `Handler`, and add the `Access-Control-Allow-Origin` header to allow every origin, i.e. `*`. It would've been better to use `origin` from the `Request` header and to make sure it's from `localhost` and then return just the value of the `origin` header. But since we're just learning here, we're being a bit sloppy.

And just for fun, here's the code change to support `OPTIONS` in our `server`:

```
import HTTPure.Status as HTTPStatus ❶

router :: HandlerEnv -> Request -> ResponseM
router env req@{ body, method }
  | method == HTTPure.Post =
      let handlers = apiHandlers <#> (_ $ body) in
      case hush $ oneOf handlers of
        Nothing -> HTTPure.badRequest body
        Just reader -> runReaderT reader env
          >>= \res@{ headers } -> pure $ res { headers = headers <>
                    HTTPure.header "Access-Control-Allow-Origin" "*" }
  | method == HTTPure.Options = ❷
    let headers = HTTPure.headers [
        Tuple "Access-Control-Allow-Origin" "*"
      , Tuple "Access-Control-Allow-Methods" "*"
      , Tuple "Access-Control-Allow-Headers" "*"
      ] in
    HTTPure.emptyResponse' HTTPStatus.ok headers ❸
  | otherwise = HTTPure.notFound
```

❶ Added this import.

❷ When the browser sends `OPTIONS`, we're going to respondng to allow all `Methods` and `Headers` from any `origin`.

❸ Responding with an empty body using `emptyResponse'` since `OPTIONS` only needs to return headers.

But, we're not going to do any of this. I've only put it here for future reference, i.e. if you find yourself in a situation where you need to support CORS on the server.

30.9.2. Static File Server Solution

Instead of dealing with `CORS`, we're going to have our `server` serve our `index.html` and other files to the Browser instead of relying on Python's `SimpleHTTPServer`.

This means that the files and the API calls will be from the exact same server avoiding the CORS issue altogether.

To do this, we're going to have to support the `GET` method since that's what the Browser uses to retrieve files from a web server.

But, we're going to want to also make sure we don't serve up just any files. So, we'll support a ***Whitelist*** of acceptable file types, i.e. file extensions, that can be safely served up.

Another thing we'll do is to sequester the files that can be served up when we build the client. These files will be served up by our `server`.

First, let's change our build to copy the appropriate files to a `dist`, or distribution Folder. To do this, we'll need to modify our `Makefile` in our client:

```
build:
    npx spago bundle-app ❶
    mkdir -p dist ❷
    cp {index.html,index.js} dist ❸

.PHONY: build ❹
```

❶ Changed `build` to `bundle-app`. `bundle-app` is the build option in `spago` to build all of your code into a single `index.js` file for the Browser.

❷ The `-p` option will not error if the directory already exists.

❸ Copy both `index.html` and `index.js` to `dist`. Make sure there are NO spaces in the list of files.

❹ Removed the `run` target from the file and from this line.

Make the above change to your client's `Makefile`.

Next, add `index.js` to your `.gitignore`:

```
/bower_components/
/node_modules/
/.pulp-cache/
/output/
/generated-docs/
/.psc-package/
/.psc*
/.purs*
/.psa*
/.spago
index.js ❶
```

❶ This is the output of the `bundle-app` command.

Now run to create a distribution, i.e. release:

```
make build
```

You should now see a `dist` Folder with the appropriate files in it.

Remember that every time we change the client, we'll have to run `make build` to BUILD and COPY the files so that our `server` can serve the updated files to the Browser.

Next, add the following line to the `.gitignore` file that `spago init` created:

```
dist
```

This will EXCLUDE this directory from the `git` repository.

If you're coding in VSCode, you should notice that `index.js` and the `dist` directory are grayed out.

Now, switch to your `server` project.

We're going to support serving up files, but before we do let's make a plan:

- Maintain a Whitelist of file extensions that are allowed to be retrieved
- Return the appropriate MIME Type based on the file extension
- Read files from `../client/dist`, which is relative to the `server` directory

First, we'll create a Whitelist in `Main.purs`:

```
whitelistExtensions :: Array String
whitelistExtensions = [ "js", "html", "jpg", "png" ]
```

Add this to `server's` project in `Main` now.

We can add to this list if we have a need for a different file type. This is just an added precaution in case we accidentally put a file of a different extension into the `client/dist` folder. At least our `server` won't deliver it.

Next, we're going to write a Function that takes a File Extension and will return the appropriate MIME Type. The MIME Type will tell us what to put in the `Content-Type` HTTP Header in the response. This information will tell the Browser how to treat the file.

First, create a `Map` of File Extensions and their corresponding MIME Types. Use `Data.MediaType` and `Data.MediaType.Common` to help you. Call this map `mimeTypes`. Return here when you're ready to move on.

First, we need to add `media-types` to our `spago.dhall` file so we can use this library:

```
npx spago install media-types
```

Then build:

```
npx spago build
```

And then we can add the following code to `Main`:

```
import Data.Map (Map) ❶
import Data.Map as Map
import Data.MediaType (MediaType)
import Data.MediaType.Common as MIME
import Data.Tuple (Tuple(..))

mimeTypes :: Map String MediaType
mimeTypes = Map.fromFoldable [ ❷
    Tuple "js" MIME.applicationJavascript ❸
  , Tuple "html" MIME.textHTML
  , Tuple "jpg" MIME.imageJPEG
  , Tuple "png" MIME.imagePNG
  ]
```

❶ Added these imports.

❷ Using `fromFoldable` from `Map` to take an `Array` of `Tuples` of keys and values to create our `Map`.

❸ Each entry in our `Array` is a `Tuple` where the first position has the key and the second has the value.

Now, that I'm looking at this, it seems that we've got 2 sources of truth. The `whitelistExtensions` and `mimeTypes`. Imagine adding to `mimeTypes` but forgetting to add it to `whitelistExtensions` or vice versa.

Delete `whitelistExtensions` and we'll use `mimeTypes` as our Whitelist. When we cannot find a MIME Type then we'll treat that file as being blacklisted.

Next, add a Function called `mimeType` (no `s`) that takes a `fileName` and returns its MIME Type. Use `mimeTypes` to look up its type. Return here to continue reading when you're done.

First, we start with the Type Signature:

```
mimeType :: String -> Maybe MediaType ❶
```

❶ Returns a `Maybe` since we could fail to find the MIME Type. In that case, we'll consider this file request as invalid since it's not part of our Whitelist.

Next, we'll do the start of the implementation:

```
mimeType :: String -> Maybe MediaType
mimeType fileName = do ❶
  ???
```

❶ Coding in the `Maybe` Monad.

Next step is to get the File Extension:

```
import Data.Array (last, tail) ❶
import Data.String.Common (split)
import Data.String.Pattern (Pattern(..))

mimeType :: String -> Maybe MediaType
mimeType fileName = do
  ext <- last =<< (tail $ split (Pattern ".") fileName) ❷ ❸
  ???
```

❶ Added these imports.

❷ Split the `fileName` on `.` and throw away the head, i.e. part before the file extension, then get the `last` item in the `Array`.

❸ Either extract the `Just` result or Short-circuit returning `Nothing` to the caller. We can do this because we're in the `Maybe` Monad.

And finally, we'll look up the File Extension in the `mimeTypes` Map:

```
import Data.String.Common (split, toLower) ❶

mimeType :: String -> Maybe MediaType
mimeType fileName = do
  ext <- last =<< (tail $ split (Pattern ".") fileName)
  Map.lookup (toLower ext) mimeTypes ❷
```

❶ Added `toLower` to input list.

❷ `Map.lookup` will return a `Maybe`.

And since we have a `do` block with only 2 lines of code, we can rewrite this as:

```
mimeType :: String -> Maybe MediaType
mimeType fileName = (last =<< (tail $ split (Pattern ".") fileName)) >>=
  flip Map.lookup mimeTypes <<< toLower
```

I'm not sure I like this version better. You can choose for yourself. I'm going to keep the `do` block version.

Now, add the following to your code:

```
staticRoot :: String
staticRoot = "../client/dist"
```

This is the root directory for all static file requests. The files are considered static since they are not dynamically generated files by our server, i.e. it's not Server Rendered content as one might find in technologies such as `JSP`, `ASP.NET`, `WebForms`, `Next.js`, etc. or when running `React` or `Angular` on the server.

Next, write a Function called `staticFiles` that will take a `Request` and return a `ResponseM`. From that request it'll use the `path`, which is an `Array`, and `staticRoot` to build a path to the file. If the `path` is empty then `index.html` will be the default.

Then that file will be read and its contents returned to the client if and only if it's in the Whitelist, i.e. we can find a MIME Type for its File Extension.

See how we did `readTextFile` in the Account Manager in our `server` as reference, but, in this case, use `try` from `Control.Monad.Error.Class` and when you cannot read the file send `HTTPure.notFound`, i.e. `404`.

And finally, you're going to want to send both `Content-Length` and `Content-Type` in the HTTP Headers. This can be accomplished with `HTTPure.headers` and by using the Primed version of `ok`, i.e. `HTTPure.ok'`.

There a lot here, so take the time to re-read the specifications before delving into the code. And then take each piece one step at a time. Then return here once you're ready to move forward.

The first step, as always, is the Type Signature:

```
staticFiles :: Request -> ResponseM
```

And we just care about the `path` portion of the `Request`:

```
staticFiles :: Request -> ResponseM
staticFiles { path } =
  ???
```

Since `path` is an `Array`, we need to put the path together to form the `fileName`:

```
import Data.Array (last, tail, null) ❶
import Data.Foldable (class Foldable, foldl, intercalate) ❷

staticFiles :: Request -> ResponseM
staticFiles { path } =
  let fileName =
        if null path then "index.html" ❸
        else intercalate "/" path in ❹
  ???
```

❶ Added `null` to the import list.

❷ Added `intercalate` to the import list.

❸ Handle default case.

❹ Combine Path Segments to produce a File Path. Note, if you're on Windows and Node doesn't normalize the path separator as I expect it should, you'll have to change the `/` to be `\\`.

Next, we'll look up the MIME Type:

```
import Data.Maybe (Maybe(..), maybe) ❶

staticFiles :: Request -> ResponseM
staticFiles { path } =
  let fileName =
        if null path then "index.html"
        else intercalate "/" path in
  mimeType fileName # maybe HTTPure.forbidden \mime -> do ❷
    ???
```

❶ Added `maybe` to the import list.

❷ Use our `mimeType` Function to look up the MIME Type for the `fileName`. We'll respond with `forbidden` or `403` if not found. Otherwise the Lambda is called with the MIME Type, `mime`.

And now, we're ready to read the whole file:

```
import Control.Monad.Error.Class (try) ❶
import Node.Encoding (Encoding(..))
import Node.FS.Aff (readTextFile)

staticFiles :: Request -> ResponseM
staticFiles { path } =
  let fileName =
        if null path then "index.html"
        else intercalate "/" path in
  mimeType fileName # maybe HTTPure.forbidden \mime -> do
    fileData' <- try $ readTextFile ASCII $ staticRoot <> "/" <> fileName ❷
    ???
```

❶ Added these imports.

❷ Reading file with prepended `staticRoot` on the front of the `fileName`. Use `try` so we can properly handle exceptions. Same caveat with Windows as with any File Path.

And now we handle the error case:

```
import Data.Either (Either(..), hush, either) ❶

staticFiles :: Request -> ResponseM
staticFiles { path } =
  let fileName =
        if null path then "index.html"
        else intercalate "/" path in
  mimeType fileName # maybe HTTPure.forbidden \mime -> do
    fileData' <- try $ readTextFile ASCII $ staticRoot <> "/" <> fileName
    fileData' # either (const HTTPure.notFound) \fileData -> ❷
      ???
```

❶ Added `either` to import list.

❷ If we have a `Left` then `notFound`, i.e. `404` error. Otherwise, we call the Lambda with `fileData`.

Now, we need to respond with `Content-Type` and `Content-Length`. So, we'll build the headers first:

```
import Data.String (length) ❶
import Data.Newtype (unwrap)

staticFiles :: Request -> ResponseM
staticFiles { path } =
  let fileName =
        if null path then "index.html"
        else intercalate "/" path in
  mimeType fileName # maybe HTTPure.forbidden \mime -> do
    fileData' <- try $ readTextFile ASCII $ staticRoot <> "/" <> fileName
    fileData' # either (const HTTPure.notFound) \fileData ->
      let headers = HTTPure.headers
            [ Tuple "Content-Length" (show $ length fileData) ❷
            , Tuple "Content-Type" $ unwrap mime ❸
            ] in
      ???
```

❶ Added these imports.

❷ Set `Content-Length` base on the length of `fileData`.

❸ Unwrap `MediaType` to get to its underlying `String`. We can use `unwrap` since `MediaType` has a `Newtype` Instance.

And now we can response with the Primed Version of `ok`:

```
staticFiles :: Request -> ResponseM
staticFiles { path } =
  let fileName =
        if null path then "index.html"
        else intercalate "/" path in
  mimeType fileName # maybe HTTPure.forbidden \mime -> do
    fileData' <- try $ readTextFile ASCII $ staticRoot <> "/" <> fileName
    fileData' # either (const HTTPure.notFound) \fileData ->
      let headers = HTTPure.headers
            [ Tuple "Content-Length" (show $ length fileData)
            , Tuple "Content-Type" $ unwrap mime
            ] in
      HTTPure.ok' headers fileData ❶
```

❶ `ok'` takes our `headers`, whereas `ok` does not.

The last thing we'll need to do is call this function. So go change `router` to handle the HTTP `GET` and call `staticFiles`. Then return here.

Here's that code change:

```
router :: HandlerEnv -> Request -> ResponseM
router env req@{ body, method } ❶
  | method == HTTPure.Post =
      let handlers = apiHandlers <#> (_ $ body) in
      case hush $ oneOf handlers of
        Nothing -> HTTPure.badRequest body
        Just reader -> runReaderT reader env
  | method == HTTPure.Get = staticFiles req ❷
  | otherwise = HTTPure.methodNotAllowed
```

❶ Aliased the full `Request` to be `req`.

❷ Added this Guard.

Now, we're ready to run our server and test it:

```
npx spago run
```

Now, go to `Postman` and create a new Tab and enter in the URL, `localhost:3000`, leave the method as `GET` and see if you get `index.html` returned when you click `Send`.

Click on the Headers tab in the Response area and check that `Content-Type` and `Content-Length` are

properly set:



Now change the URL to `localhost:3000/index.js` and click `Send`:



Now change the `.js` to `.pdf` in the URL, hit `Send` and you should get:



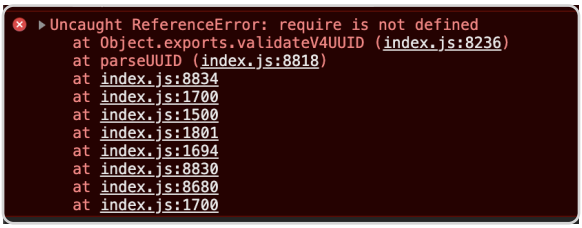
Seems as if our Static File Server is working.

Now, go to the Browser and enter in the URL, `localhost:3000`. This should show the Logon Screen as it used to when we went to port `8000`.

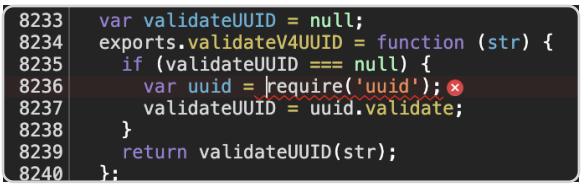
Once you have that up, you can try to logon and see what you get.

When I try this, nothing happens. Check the Console to see if there's any errors besides the fact that we don't support a `favicon.ico`.

Here's what I see:



Click on the `index.js:8236` or whatever link is shown at the top of your Stack Trace. That should take you to something like:



This is trying to `require` the UUID Node module. But we're running in the Browser not Node. We're going to have to use `Parcel` or `Webpack` to convert our `index.js` to something that will run in the Browser.

These Bundlers will put all of our code along with all Node modules code that's required via `require` into a single Javascript file.

We're going to use `Parcel` for this project because it's simple. But `Webpack` is what I'd use for a Production Server because it's more configurable.

Return to your client project.

The first step is to install `Parcel` for the client project:

```
npm install --save-dev parcel@1.12.3 ❶
```

❶ `--save-dev` will save this installation under `devDependencies` in our `package.json`. Those are dependencies for development only.

Now, modify your `Makefile` to match:

```
build:
    npx spago bundle-app
    rm -f dist/*.js dist/*.map
    exec npx --node-arg=--max-old-space-size=4096 parcel build index.html ❶
    # for Node 15+
    # NODE_OPTIONS='--max-old-space-size=4096' exec npx parcel build --no
-minify index.html
```

❶ When our program gets large, `Parcel` will need more heap, hence the `4096` value. This probably won't happen for us, but we're going to put this here as a precaution.

Notice how we no longer need to create the `dist` Folder nor do we need to copy any files there. `Parcel` does all of that for us.

We do, however, need to ignore the `.cache` directory, which `Parcel` creates to speed up build times. So, add the following line to `.gitignore`:

```
.cache
```

If you're in VSCode, you should see `.cache` grayed out, i.e. once it's created.

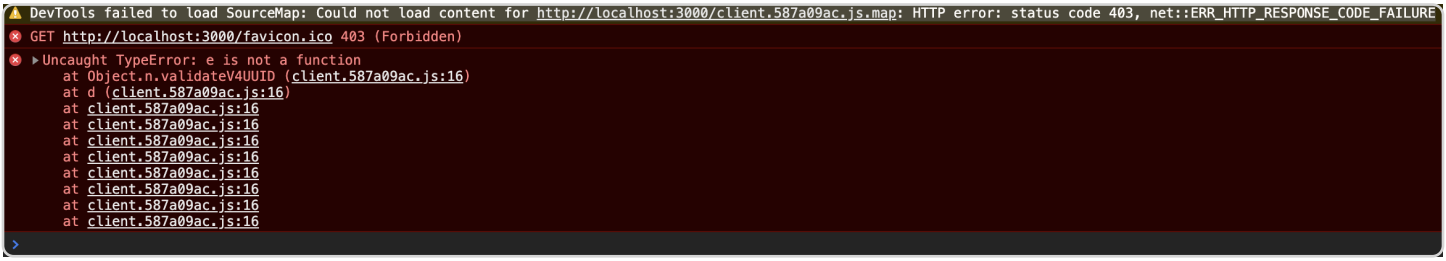
Now, rebuild the client with:

```
make build ❶
```

❶ This build will take longer since Parcel needs to run.

Refresh the Browser and try to logon again and see what happens now.

When I do this, I get nothing. The Console reveals a few useful pieces of information for us:



The first thing I notice is that we aren't returning the Source Map file that `Parcel` created and is used by the Browser's Debugging Tools to help us make sense of our Javascript code if, and when, we ever get desperate enough to debug at the Javascript level.

Here's the details of that error:

```
DevTools failed to load SourceMap: Could not load content for
http://localhost:3000/client.587a09ac.js.map: HTTP error: status code 403,
net::ERR_HTTP_RESPONSE_CODE_FAILURE
```

This isn't a terrible issue, but we can fix it so easily and so we should just in case someday we'll need this.

Let's return to the `server` code and change `mimeTypes` as follows:

```
mimeTypes :: Map String MediaType
mimeTypes = Map.fromFoldable [
    Tuple "js" MIME.applicationJavascript
  , Tuple "html" MIME.textHTML
  , Tuple "jpg" MIME.imageJPEG
  , Tuple "png" MIME.imagePNG
  , Tuple "map" MIME.textPlain ❶
  ]
```

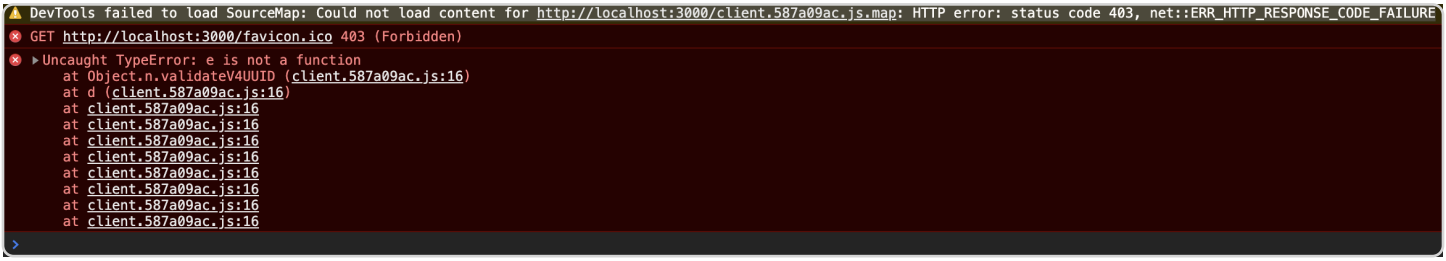
❶ Source Maps end in `.map` and are just plain text.

Kill the server with `Ctrl-C` and rerun it:

```
npx spago run
```

Then refresh the Browser and see if the `403` error goes away on our Source Map.

It did for me. But we still have to deal with the other error. Look at the error we got in our last test:



The `e is not a function` is pretty useless, but clicking on the top link of the Stack Trace takes us to `validateV4UUID` a Node Function. This makes me wonder if we installed the packages we need for `Data.UUID`.

Go look up `Data.UUID` on `Pursuit` and click on its package link, then read the docs to see what packages we should install. Then check `package.json` in the `client` code to see if they're there under the `dependencies` section. Return here when you're done.

When I do this I find that the NPM package called `uuid` needs to be installed. And when I checked `package.json`, I didn't find it.

That means we should run the following in the `client` directory:

```
npm install --save uuid@8.3.2
```

Now we need to rebuild our `client` since `Parcel` bundles all of our code into one massive Javascript file and the newly installed `uuid` code is not there since we just installed it:

```
make build
```

Now, refresh the browser and try to logon on once more and see what happens now.

When I do this, I'm redirected to the ChangePassword page.

By the way, if you're using Chrome and it tries to save your password and then it immediately warns you about a data breach with your password on `localhost` and you'd like this to go away, you can do so by disabling the saving of passwords.

Since Chrome changes things constantly, you may be better served by searching Google for how to turn this

off, but here's how to do this as of the time of this writing.

First, enter `chrome://settings/passwords` into the Address Bar of Chrome. Then turn off the following:



If this isn't at this location in Chrome when you try, I'd suggest doing a Google search for how to turn it off.

And realize that if you do this, it'll turn off asking to remember passwords for ALL websites, so you may want to turn it back on later.

Also, keep in mind that this error is benign, although annoying.

30.10. ChangePassword Page

We're going to make the ChangePassword Page just have a button on it that we'll click and it'll redirect us to the Users page since we don't have an API for setting a permenant password.

We should fomulate a plan for coding this.

First, we'll use the code from the Logon Page as a template.

Second, we'll start with defining:

- `Input` (from Parent)
- `Output` (to Parent)
- `State`
- `Action` (events from EventListeners)
- `Query` (`H.tell` or `H.request` from Parent)
- `Slots` (for Children)

Then write:

- `component`
- `handleAction` (for the button click)
- `render` (put the button in the center of the screen using Flex Grid)

If you don't already know how to center a button in a `div` using `display flex` using `align-items` and `justify-content`, then take the time know to Google it. I'd suggest using the MDN (Mozilla Developer Network) link that shows up.

First thing to do is to create a File under `src/Component` called `ChangePassword.purs` and write just the Type Definitions. Then return here and reading on.

Here that file so far:

```
module Component.ChangePassword where

import Prelude

import Data.Const (Const)
import Halogen as H ❶
import Halogen.HTML.CSS as HC
import Halogen.HTML as HH
import Halogen.HTML.Events as HE

type State = {} ❷
type Input = Unit ❸
type Output = Void ❹
data Action
  = Click ❺

type Query :: ∀ k. k -> Type
type Query = Const Void ❻

type Slots :: ∀ k. Row k
type Slots = () ❼
```

❶ All of the Halogen imports are a bit premature, but we'll need at least these, so might as well add them now.

❷ No `State`.

❸ No `Input`, but it must be a Value that we can create since it must be sent from the Parent. This is why we use `Unit` and NOT `Void`.

❹ No `Output`. This can be `Void` since we never create an `Output` and our Parent can use `absurd` for the Output Handler in the `HH.slot` call. We'll need to do this once we add this Page to the Router Component.

❺ The one and only `Action` is clicking the button to redirect.

❻ Remember that `Query` is of Kind `Type -> Type`. That's why `Const Void` works.

❼ No Child Components.

Next, write `component` stealing liberally from the Logon Page. Stub out `handleAction` and `render` for now. Then return here.

Here's the `component's` Type Signature:

```
import Capability.Navigate (class Navigate) ❶
import Data.Route (Route)
import Effect.Aff.Class (class MonadAff)

component
  :: ∀ m
  .  MonadAff m ❷
  => Navigate m Route ❸
  => H.Component Query Input Output m
```

❶ Added these imports.

❷ We always need this so we can eventually run in `Aff`.

❸ We're going to need this Capability from our Monad, `m`, to navigate away from this Page.

And now the implementation with stubbed out `handleAction` and `render`:

```
component = H.mkComponent
  { initialState: \_ -> {} ❶
  , render
  , eval: H.mkEval H.defaultEval
      { handleAction = handleAction ❷
      }
  }
  where
  handleAction :: Action -> H.HalogenM State Action Slots Output m Unit
  handleAction _ = pure unit

  render :: State -> H.ComponentHTML Action Slots m
  render _ = HH.text ""
```

❶ Since we have no State, there's nothing to do here.

❷ We're only supporting `handleAction`.

And now write the code for `handleAction` then return here when that's complete.

Here's the implementation:

```
import Capability.Navigate (class Navigate, navigate) ❶
import Data.Maybe (Maybe(..)) ❷
import Data.Route as Route ❷

-- ABOVE CODE REDACTED
  where
  handleAction :: Action -> H.HalogenM State Action Slots Output m Unit
  handleAction = case _ of ❸
    Click -> navigate $ Route.Users Nothing ❹
-- BELOW CODE REDACTED
```

❶ Added `navigate` to the import list.

❷ Added these imports.

❸ Removed Parameter from left-hand side of equal sign and used wildcard `case` instead.

❹ We pass `Nothing` to `Route.Users` since we have no particular user in mind.

Now, add the code to the Router Component to render our component like we did with `Logon` before reading any further.

Here's that change:

```
import Component.ChangePassword as ChangePassword ❶

-- ABOVE CODE REDACTED
  render :: State -> H.ComponentHTML Action Slots m
  render { route } = case route of
    Logon -> HH.slot _logon unit (Page.component Logon.component) unit absurd
    Logoff -> HH.span [ HC.style $ color white ] [ HH.text "Logoff" ]
    Users _ -> HH.span [ HC.style $ color white ] [ HH.text "Users" ]
    ChangePassword -> HH.slot _changePassword unit ❷
      (Page.component ChangePassword.component) unit absurd ❸
```

❶ Added this import.

❷ We use `_changePassword` to identify the correct Component.

❸ We pass `unit` for `Input` and use `absurd` as a Function that takes the `Output`, which is a `Void`, and converts it to an `Action`.

We can use a Function called `HH.slot_` to eliminate the need to pass `absurd` as the final parameter when we don't have any `Output` from our Child Component:

```
-- ABOVE CODE REDACTED
  render :: State -> H.ComponentHTML Action Slots m
  render { route } = case route of
    Logon -> HH.slot_ _logon unit (Page.component Logon.component) unit ❶
    Logoff -> HH.span [ HC.style $ color white ] [ HH.text "Logoff" ]
    Users _ -> HH.span [ HC.style $ color white ] [ HH.text "Users" ]
    ChangePassword -> HH.slot_ _changePassword unit
      (Page.component ChangePassword.component) unit ❶
```

And finally, code up the `render` Function in `Component.ChangePassword` to place a button that's in the middle of the page with the text `Users`.

Use everything that we've coded so far in both `Component.Page` and `Component.Logon` to help you. Both Google and StackOverflow are your friends for figuring out why your CSS isn't working.

Use Flex Grids since they are pretty simple to use and don't forget about `align-items` and `justify-content`.

Once you've got something that you think will work well, return here.

I mostly stole the code from Logon's button:

```
import AppTheme (themeColor, themeFont) ❶
import CSS.Background (backgroundColor)
import CSS.Color (white)
import CSS.Common (center)
import CSS.Cursor (cursor, pointer)
import CSS.Display (display, flex)
import CSS.Flexbox (flexDirection, row, justifyContent, alignItems)
import CSS.Font (FontWeight(..), fontWeight, fontSize, color)
import CSS.Geometry (width, height, paddingTop)
import CSS.Property (value)
import CSS.Size (vw, rem)

-- ABOVE CODE REDACTED
  render :: State -> H.ComponentHTML Action Slots m
  render _ =
    HH.div [
      HC.style do
        display flex
        flexDirection row
        alignItems center
        justifyContent center
        paddingTop $ vw 0.65
    ]
    [ HH.button
      [
        HC.style do
          backgroundColor themeColor
          themeFont
          fontWeight $ FontWeight $ value "500"
          fontSize $ vw 1.0
          width (rem 20.0)
          height $ vw 3.0
          color white ❷
          cursor pointer ❷
      , HE.onClick $ const Click
      ]
      [ HH.text "Users" ]
    ]
```

❶ Added these imports. With the `CSS` imports, we could've included most of these imports with `import CSS` since it re-exports most common CSS Functions except for `CSS.Cursor`, which we wrote. But I like having them separated since it let's me know what CSS Functionality I have instead of having this massive import list in no particular grouping or order.

❷ Removed the disabled checks here from what we did on the Logon Page.

Now, build the client:

```
make build
```

Make sure your `server` is still running and then go to your Browser and make sure the URL is `localhost:3000/#/change-password`. If it already is then simply reload the page, otherwise, type this URL into the Address Bar and then hit Return.

When I do either of these, I get a `Users` button in the middle of my page.

But we aren't logged on any more. We refreshed our page and the authToken is no longer in memory.

We need to have the `Router` be smarter. If there's an empty `userRef`, then the `Router` must redirect the user to the Logon Page no matter what the URL Hash value is.

Take a stab at making that modification before reading my solution.

Here's my solution:

```
import Control.Monad.Reader.Class (class MonadAsk, ask) ❶
import Data.Maybe (Maybe(..), isNothing) ❷
import Effect.Ref as Ref ❸

-- ABOVE CODE REDACTED
  where
  handleQuery
    :: ∀ a
    .  Query a
    -> H.HalogenM State Action Slots Output m (Maybe a)
  handleQuery = case _ of
    Navigate route a -> do ❹
      { userRef } <- ask ❺
      ref <- H.liftEffect $ Ref.read userRef ❻
      H.modify_ _ { route = if isNothing ref then Logon else route } ❼
      pure (Just a) ❽
-- BELOW CODE REDACTED
```

❶ Added `ask` to the import list.

❷ Added `isNothing` to the import list.

❸ Added this import.

❹ Added a `do` block to do multiple steps.

❺ `ask` for the `ReaderT` environment, i.e. `Env`, destructing to get just `userRef`

❻ Read from the `Ref` so we can see if its `Nothing` or not. We use `liftEffect` since `Ref.read` runs in `Effect` and we're in `HalogenM`.

❼ Always force the `route` to be `Logon` if we haven't logged on. Otherwise, abide by the `route` we got from the `Navigate` Action.

❽ No need for `applySecond` since we're now in a `do` block.

Now, rebuild the `client`:

```
make build
```

Now, refresh the browser and see if it redirects us to the Logon Page.

And it does, but it doesn't change the Hash in the URL. That's because we routed INTERNALLY.

There are 2 different things we can do here. We can leave this and then when the user logs on successfully, we can query the Hash to see where the user was trying to go to and go there instead of the Logon Page's default location.

To do this, we'd use `getHash` from `Routing.Hash` and `Route.parse` from `Routing.Duplex`. Once the user logs on, we'd get the Hash and try to parse it. If it parses and it's NOT `Route.Logon`, then we'd honor it and redirect. We're going to rely on each Page to validate that the user is Authorized to view that Page, so this doesn't pose a security risk.

If the Hash doesn't parse or it parses but is just `Route.Logon`, then we'd keep the current logic. Of course, this logic would be slightly complicated if we had a real, working ChangePassword Page.

But, we don't want that behavior for this Application even though that's the norm. For us, we're just going to navigate using `navigate` to set the Hash accordingly instead of just routing internally like before:

```
import Capability.Navigate (class Navigate, navigate) ❶

-- ABOVE CODE REDACTED
  where
  handleQuery
    :: ∀ a
    .  Query a
    -> H.HalogenM State Action Slots Output m (Maybe a)
  handleQuery = case _ of
    Navigate route a -> do
      { userRef } <- ask
      ref <- H.liftEffect $ Ref.read userRef
      if isNothing ref then navigate Logon ❷
      else H.modify_ _ { route = route } ❸
      pure (Just a)
-- BELOW CODE REDACTED
```

❶ Added `navigate` to the import list.

❷ Now we explicitly `navigate` to `Logon`. This will change the Hash in the Address Bar of the Browser.

❸ Otherwise, we just abide by the requested `route`.

Now, rebuild the `client`:

```
make build
```

And refresh the Browser.

When I do this, my Hash is `/logon`.

Log back on and try out the `Users` button to see what happens.

It goes to the Users Page when I click it. Now, refresh the page and see if you go back to the Logon Page with the correct Hash.

When I did that, it does. So, it looks like our Hash Routing is working much better now.

30.11. Users Page

The Users Page is will contain a list of users to the left and when the user clicks on a User from the list, we will route to the Users Page, but this time with a `User` selected.

This routing will allow the Browser back buttons to work in a more natural way that users have come to expect.

When a User is selected, the right side of the Page will display fields that will contain details regarding that user.

This page will also contain a Create User button that will allow us to create a new user.

Start by creating a File under `src/Component` called `Users.purs`.

Next, code up the following Types as we did on the ChangePassword Page keeping in mind that we may or may not have a currently selected `User` and that we need to display a message on the screen when a user is not authorized to view this screen since they're not an admin:

- `Input` (from Parent)
- `Output` (to Parent)
- `State`
- `Action` (events from EventListeners)
- `Query` (`H.tell` or `H.request` from Parent)
- `Slots` (for Children)

Use what we've written so far as your cheatsheet in coding this and then return here when you're ready to move forward.

Here's the start of `Component.Users`:

```
module Component.Users where

import Prelude

import Data.Const (Const) ❶
import Data.Maybe (Maybe)
import Entity.User (User)
import Halogen as H
import Halogen.HTML.CSS as HC
import Halogen.HTML as HH
import Halogen.HTML.Events as HE
import Halogen.HTML.Properties as HP

type State =
  { authorized :: Boolean ❷
  , selectedUser :: Maybe User ❸
  , users :: Array User ❹
  }
type Input = Unit
type Output = Void
data Action
  = Initialize ❺

type Query :: ∀ k. k -> Type
type Query = Const Void

type Slots :: ∀ k. Row k
type Slots = ()
```

❶ Some of the imports are needed now and others are in anticipation of code to come.

❷ `authorized` is a flag we'll use to keep track of whether the user is an `admin` or not. This must be stored in the `State` since `render` is a Pure Function and cannot read from the `userRef`. `render` will NOT display the page when the user is not authorized to view this page.

❸ `selectedUser` will come from the Hash Route. We'll want to render differently when this is the case so that the selected user is highlighted in the list of users.

❹ `users` is the list of `Users` that we got back from our `server`.

❺ We have to call our `server` using `QueryUsers` API call to get a list of all of the users. We'll do this as soon as possible, which is going to be the first `Action` we handle after our Component is initialized.

This is a good start based on what we know up front. We'll make any necessary changes here as we learn more about what we need to support our desired behavior for this Page.

Next, write the `component` Function, a stubbed-out version of `render` and stub out the `Initialize` case in

`handleAction`. Return here when that's all done.

Here's the Type Signature for `component`:

```
import Capability.Navigate (class Navigate) ❶
import Data.Route (Route)
import Effect.Aff.Class (class MonadAff)

component
  :: ∀ m
  .  MonadAff m ❷
  => Navigate m Route ❸
  => H.Component Query Input Output m
```

❶ Added these imports.

❷ Our Monad, `m`, will eventually run in `Aff`.

❸ We know that we're going to have to navigate when the user picks a different user, so we'll need this Capability from our Monad, `m`.

Now for the implementation:

```
import Data.Maybe (Maybe(..)) ❶

component
  :: ∀ m
  .  MonadAff m
  => Navigate m Route
  => H.Component Query Input Output m
component = H.mkComponent
  { initialState: \_ ->
      { authorized: false ❷
      , selectedUser: Nothing
      , users: []
      }
  , render
  , eval: H.mkEval H.defaultEval
      { handleAction = handleAction
      , initialize = Just Initialize ❸
      }
  }
  where
  ???
```

❶ Added `(..)` to include Data Constructors.

❷ We're initializing our `State` to NOT authorized, NO selected user and NO users for now. `Initialize` will set these correctly since it can perform effects to get the logged on user and query the backend for `users`.

❸ Handle the `initialize` with an `Action` of `Initialize`.

And now the stubbed out code:

```
-- ABOVE CODE REDACTED
  where
  handleAction :: Action -> H.HalogenM State Action Slots Output m Unit
  handleAction = case _ of
    Initialize -> pure unit

  render :: State -> H.ComponentHTML Action Slots m
  render _ = HH.text ""
```

Let's make a plan for how we're going to code up `Initialize`:

- Get `authToken` from `LoggedOnUser`
- Call `server` to query users if and only if user is `admin`
- WAIT. WAIT.

We don't know if the user is an admin or not. How can we get this information?

Let's look at `LoggedOnUser`:

```
type LoggedOnUser =
  { authToken :: UUID
  }
```

Currently, we don't have this information.

Next, let's look at where we got this information from, i.e. `LogonResponse` in our `server`:

```
data LogonResults
  = LogonResultsSuccess
    { authToken :: UUID
    , mustChangePassword :: Boolean
    }
  | LogonResultsFailure
```

It should be clear that we're missing a crucial piece of information to complete the code for this Page.

Return to the `server` code and where we put `authToken` into the `LogonResponse`, let's also add `admin`. Make this change before reading further.

First thing to do is to add an `admin` field to `LogonResponse`:

```
data LogonResults
  = LogonResultsSuccess
    { authToken :: UUID
    , admin :: Boolean ❶
    , mustChangePassword :: Boolean
    }
  | LogonResultsFailure
```

❶ Added `admin` to Record.

Next, we'll update the Logon API Handler in `Handler.Api.Logon`:

```
handler :: LogonRequest -> Handler
handler (LogonRequest { userName, password }) = do
  { accountsAVar, sessionsAVar } <- ask
  verification <- lift $ verifyLogon accountsAVar userName password
  response <- case verification of
    Nothing -> pure $ LogonResponse LogonResultsFailure
    Just (Account { admin, temporaryPassword }) -> do ❶
      authToken <- lift $ createSession sessionsAVar userName
      pure $ LogonResponse
        $ LogonResultsSuccess
          { authToken
          , admin ❷
          , mustChangePassword: temporaryPassword
          }
  HTTPure.ok $ encodeJSON response
```

❶ Added `admin` to the destructuring from the `Account`.

❷ Added `admin` to the response.

Now, kill the server with `Ctrl-C` and rerun it with:

```
npx spago run
```

Once the server is running, return to the `client` project and build it so the editor will see the changes we

made to `LogonResultsSuccess` from the `server` code:

```
make build
```

Now add the `admin` flag to `LoggedOnUser`, then return here.

Here's that change:

```
type LoggedOnUser =
  { authToken :: UUID
  , admin :: Boolean ❶
  }
```

❶ `admin` was added.

And finally, add the `admin` Value when we initialize `LoggedOnUser`. Return here when that's complete.

Here's that change in `Component.Logon` in `handleAction`:

```
-- ABOVE CODE REDACTED
      case logonResponse of
        Left err -> alertError err
        Right (LogonResponse LogonResultsFailure) ->
          alertError "Invalid Logon Credentials"
        Right (LogonResponse (LogonResultsSuccess
          { authToken, admin, mustChangePassword })) -> do ❶
            log =<< logEntry Info "User logged on"
            { userRef } <- ask
            H.liftEffect $ Ref.write (Just { admin, authToken }) userRef ❷
            navigate <=< logonRoute
              $ if mustChangePassword then PasswordTemporary
                else PasswordPermanent
-- BELOW CODE REDACTED
```

❶ Added `admin` the destructuring of `LogonResultsSuccess`.

❷ Added `admin` to `LoggedOnUser`.

Now, let's return to making our plan for coding `Initialize` in `Component.Users` with a slightly different perspective:

- Get `authToken` from `LoggedOnUser`
- If `LoggedOnUser` is NOT authorized to view this page do NOTHING

- Call `server` to query users
- Update `State` accordingly

Now, write the code for `Initialize` to query the backend for users and initialize the `State`. Steal code from what we did on the Logon Page to make API calls to the backend.

Use the same `alertError` Function for now for errors. We'll replace this in the next section when we create a Modal Dialog Component.

Then return here when you're stuck or ready to move on.

We'll start coding with:

```
  handleAction :: Action -> H.HalogenM State Action Slots Output m Unit
  handleAction = case _ of
    Initialize -> do
      ???
```

And now, we need to check the `LoggedOnUser`, which we can get from the `ReaderT` environment:

```
import Control.Monad.Reader.Class (class MonadAsk, ask) ❶
import Effect.Ref as Ref
import Env (Env)

component
  :: ∀ m
  .  MonadAff m
  => MonadAsk Env m ❷
  => Navigate m Route
  => H.Component Query Input Output m
-- BELOW CODE REDACTED

  handleAction :: Action -> H.HalogenM State Action Slots Output m Unit
  handleAction = case _ of
    Initialize -> do
      { userRef } <- ask ❸
      loggedOnUser' <- H.liftEffect $ Ref.read userRef ❹
      ???
```

❶ Added these imports.

❷ Added this Constraint so we can call `ask`.

❸ Get `userRef` from environment.

❹ Read the `Ref` to get the Value stored there, i.e. `Maybe LoggedOnUser`. Note that it's a `Maybe`. This is why I've named it with a Prime to remind myself that something's up with this Variable.

Now, we need to handle the `Maybe`:

```
import Data.Maybe (Maybe(..), maybe) ❶

  handleAction :: Action -> H.HalogenM State Action Slots Output m Unit
  handleAction = case _ of
    Initialize -> do
      { userRef } <- ask
      loggedOnUser' <- H.liftEffect $ Ref.read userRef
      loggedOnUser' # maybe (pure unit) \{ authToken, admin } -> ❷ ❸
        ???
```

❶ Added `maybe` to import list.

❷ Do nothing if there is no logged on user. This should never happen since our Router Component redirects to the Logon Page when this is the case.

❸ In the `Just` case, our Lambda destructures `LoggedOnUser`.

If you got stuck and feel you can finish, stop here and give it a try. Return here when you're ready to continue.

We only want to query for users if the user is authorized, i.e. an admin:

```
  handleAction :: Action -> H.HalogenM State Action Slots Output m Unit
  handleAction = case _ of
    Initialize -> do
      { userRef } <- ask
      loggedOnUser' <- H.liftEffect $ Ref.read userRef
      loggedOnUser' # maybe (pure unit) \{ authToken, admin } ->
        when admin do
          ????
```

And now we simply copy a block of code from what we did on Logon Page's `handleAction`:

```
import Affjax as Ajax ❶
import Affjax.ResponseFormat as ResponseFormat
import Affjax.RequestBody as RequestBody
import Control.Monad.Except (runExcept)
import Data.Api.QueryUsers (QueryUsersRequest(..), QueryUsersResponse(..)
                          , QueryUsersFailureReason(..)
                          , QueryUsersResults(..))
import Data.Bifunctor (lmap)
import Data.Either (Either(..))
import Foreign.Generic (encodeJSON, decodeJSON)

  handleAction :: Action -> H.HalogenM State Action Slots Output m Unit
  handleAction = case _ of
    Initialize -> do
      { userRef } <- ask
      loggedOnUser' <- H.liftEffect $ Ref.read userRef
      loggedOnUser' # maybe (pure unit) \{ authToken, admin } ->
        when admin do
          ajaxResult <- H.liftAff $ Ajax.post ResponseFormat.string ❷
            "http://localhost:3000/"
            $ Just $ RequestBody.String
            $ encodeJSON $ QueryUsersRequest { authToken } ❸
          let queryResponse = do ❹
                { body } <- lmap Ajax.printError ajaxResult
                lmap show $ runExcept
                  $ decodeJSON body :: _ QueryUsersResponse ❺
          case queryResponse of ❹
            Left err -> alertError err
            ???
```

❶ Added these imports.

❷ From here down, the code was copied from the Logon Page's `handleAction`.

❸ Changed the request from `LogonRequest` to `QueryUsersRequest`.

❹ Renamed `logonResponse` to `queryResponse` in these places.

❺ Changed `LogonResponse` to `QueryUsersResponse`.

This kind of coding, i.e. copy/paste coding, is always a red flag. We'll factor this bit of code out once we're done with this Function.

Now, all we have to do handle the 2 Response cases:

```
handleAction :: Action -> H.HalogenM State Action Slots Output m Unit
handleAction = case _ of
  Initialize -> do
    { userRef } <- ask
    loggedOnUser' <- H.liftEffect $ Ref.read userRef
    loggedOnUser' # maybe (pure unit) \{ authToken, admin } ->
      when admin do
        ajaxResult <- H.liftAff $ Ajax.post ResponseFormat.string
          "http://localhost:3000/"
          $ Just $ RequestBody.String
          $ encodeJSON $ QueryUsersRequest { authToken }
        let queryResponse = do
              { body } <- lmap Ajax.printError ajaxResult
              lmap show $ runExcept
                $ decodeJSON body :: _ QueryUsersResponse
        case queryResponse of
          Left err -> alertError err
          Right (QueryUsersResponse
              (QueryUsersResultsFailure { reason })) ->
            ???
          Right (QueryUsersResponse
              (QueryUsersResultsSuccess { users })) -> do
            ???
```

And now, for the failure case, we'll alert with an error:

```
-- ABOVE CODE REDACTED
        case queryResponse of
          Left err -> alertError err
          Right (QueryUsersResponse
              (QueryUsersResultsFailure { reason })) ->
            alertError $ "Query Users: " <> case reason of ❶
              NotAuthorized -> "Not Authorized"
              NotAuthenticated -> "Not Authenticated"
          Right (QueryUsersResponse
              (QueryUsersResultsSuccess { users })) -> do
            ???
```

❶ We have 2 possible Values for `QueryUsersFailureReason`.

Next, we update the `State` on the success case:

```
-- ABOVE CODE REDACTED
          case queryResponse of
            Left err -> alertError err
            Right (QueryUsersResponse
                (QueryUsersResultsFailure { reason })) ->
              alertError $ "Query Users: " <> case reason of
                NotAuthorized -> "Not Authorized"
                NotAuthenticated -> "Not Authenticated"
            Right (QueryUsersResponse
                (QueryUsersResultsSuccess { users })) -> ❶
              H.modify_ _ { authorized = true, users = users } ❷ ❸
```

❶ Removed the `do` since this is only 1 line of code.

❷ We already checked to make sure they were authorized to see this page, so `authorized` is `true`.

❸ Set `users`.

And finally, we copy and paste `alertError` since we're going to ultimately throw it away:

```
import Web.HTML.Window (alert) ❶
import Web.HTML (window)

  handleAction :: Action -> H.HalogenM State Action Slots Output m Unit
  handleAction = case _ of
    Initialize -> do
      { userRef } <- ask
      loggedOnUser' <- H.liftEffect $ Ref.read userRef
      loggedOnUser' # maybe (pure unit) \{ authToken, admin } ->
        when admin do
          ajaxResult <- H.liftAff $ Ajax.post ResponseFormat.string ❷
            "http://localhost:3000/"
            $ Just $ RequestBody.String
            $ encodeJSON $ QueryUsersRequest { authToken }
          let queryResponse = do
                { body } <- lmap Ajax.printError ajaxResult
                lmap show $ runExcept
                  $ decodeJSON body :: _ QueryUsersResponse ❸
          case queryResponse of
            Left err -> alertError err
            Right (QueryUsersResponse
                (QueryUsersResultsFailure { reason })) ->
              alertError $ "Query Users: " <> case reason of
                NotAuthorized -> "Not Authorized"
                NotAuthenticated -> "Not Authenticated"
            Right (QueryUsersResponse
                (QueryUsersResultsSuccess { users })) ->
              H.modify_ _ { authorized = true , users = users }
    where
    alertError :: String -> H.HalogenM State Action Slots Output m Unit ❹
    alertError msg = H.liftEffect $ window >>= alert msg
```

❶ Added these imports.

❷ The beginning of our copy/paste block.

❸ The ending of our copy/paste block.

❹ Copied from 'Logon'.

Now, let's address our aggregious copy/paste operation of a large portion of this Function. Factor out the code that we copied from the Logon Page into a Function called `apiCall`.

Write `apiCall` without a Type Signature and let the compiler help you determine it. Return here when you're ready to move forward.

We'll start by just copying and pasting the code block from `handleAction`:

```
apiCall = do
  ajaxResult <- H.liftAff $ Ajax.post ResponseFormat.string
    "http://localhost:3000/"
    $ Just $ RequestBody.String
    $ encodeJSON $ QueryUsersRequest { authToken }
  -- COMPILER ERROR!!
  let queryResponse = do
        { body } <- lmap Ajax.printError ajaxResult
        lmap show $ runExcept $ decodeJSON body :: _ QueryUsersResponse
```

The compiler error tells us that we need to return something:

```
apiCall = do
  ajaxResult <- H.liftAff $ Ajax.post ResponseFormat.string
    "http://localhost:3000/"
    $ Just $ RequestBody.String
    $ encodeJSON $ QueryUsersRequest { authToken } -- COMPILER ERROR!!
  let queryResponse = do
        { body } <- lmap Ajax.printError ajaxResult
        lmap show $ runExcept $ decodeJSON body :: _ QueryUsersResponse
  pure queryResponse ❶
```

❶ We're returning the response for now.

The new compiler error tells us that we need to pass in `authToken`:

```
-- COMPILER WARNING!!
apiCall authToken = do ❶
  ajaxResult <- H.liftAff $ Ajax.post ResponseFormat.string
    "http://localhost:3000/"
    $ Just $ RequestBody.String
    $ encodeJSON $ QueryUsersRequest { authToken }
  let queryResponse = do
        { body } <- lmap Ajax.printError ajaxResult
        lmap show $ runExcept $ decodeJSON body :: _ QueryUsersResponse
  pure queryResponse
```

❶ Added `authToken` Parameter.

Now, this compiles. But before we work on the Type Signature, we can simplify the return code. Notice that

we're defining a Variable called `queryResponse` only to return it via `pure queryResponse`. Let's eliminate that Variable:

```
-- COMPILER WARNING!!
apiCall authToken = do
  ajaxResult <- H.liftAff $ Ajax.post ResponseFormat.string
    "http://localhost:3000/"
    $ Just $ RequestBody.String
    $ encodeJSON $ QueryUsersRequest { authToken }
  pure do ❶
    { body } <- lmap Ajax.printError ajaxResult
    lmap show $ runExcept $ decodeJSON body :: _ QueryUsersResponse
```

❶ Remember that this `do` block is an `Either`.

The compiler warning is telling us what our Type Signature should be, so we'll add it:

```
import Data.UUID (UUID) ❶

apiCall :: ∀ t132
  .  Bind t132
  => MonadAff t132
  => UUID
  -> t132 (Either String QueryUsersResponse)
-- BELOW CODE REDACTED
```

❶ Added this import.

Looks like `t132` is a `Monad`. Your number may vary. We'll remove `Bind` since `MonadAff` requires `t132` to be a `Monad`. We'll also rename it to `m`:

```
apiCall :: ∀ m
  .  MonadAff m
  => UUID
  -> m (Either String QueryUsersResponse)
```

At this point, we can see that this function is not Polymorphic enough. It'll only work with `QueryUsersRequest`. Let's fix that:

```
apiCall :: ∀ request response m ❶
  .  MonadAff m
  => UUID
  -> request ❷
  -> m (Either String response) ❸
apiCall authToken request = do ❷
  ajaxResult <- H.liftAff $ Ajax.post ResponseFormat.string
    "http://localhost:3000/"
    $ Just $ RequestBody.String
    $ encodeJSON request -- COMPILER ERROR!! ❹
  pure do
    { body } <- lmap Ajax.printError ajaxResult
    lmap show $ runExcept $ decodeJSON body :: _ response ❺
```

❶ Added `request` and `response`.

❷ Added `request` Parameter.

❸ Changed the Monomorphic Type Parameter from `QueryUsersResponse` to the Polymorphic Type `response`.

❹ Encode the passed in `request`.

❺ The decode Type is now the Polymorphic Type, `response`.

The compiler error is because we don't have an `Encode` Instance for `request`. So, we'll require it by using a Constraint:

```
import Foreign.Generic.Class (class Encode) ❶

apiCall :: ∀ request response m
  .  MonadAff m
  => Encode request ❷
  => UUID
  -> request
  -> m (Either String response)
apiCall authToken request = do
  ajaxResult <- H.liftAff $ Ajax.post ResponseFormat.string
    "http://localhost:3000/"
    $ Just $ RequestBody.String
    $ encodeJSON request
  pure do
    { body } <- lmap Ajax.printError ajaxResult
    -- COMPILER ERROR!!
    lmap show $ runExcept $ decodeJSON body :: _ response
```

❶ Added this import.

❷ Added `Encode` Constraint on `request`.

And now we need to constrain `response` for `Decode`:

```
import Foreign.Generic.Class (class Encode, class Decode) ❶

apiCall :: ∀ request response m
  .  MonadAff m
  => Encode request
  => Decode response ❷
  => UUID
  -> request
  -> m (Either String response)
-- COMPILER WARNING!!
apiCall authToken request = do
  ajaxResult <- H.liftAff $ Ajax.post ResponseFormat.string
    "http://localhost:3000/"
    $ Just $ RequestBody.String
    $ encodeJSON request
  pure do
    { body } <- lmap Ajax.printError ajaxResult
    lmap show $ runExcept $ decodeJSON body :: _ response
```

❶ Added `class Decode` to import list.

❷ Added `Decode` Constraint on `response`.

The compiler is telling us that `authToken` isn't used. We needed it to construct `QueryResultRequest`, but now we're just passing in the full request, so we can remove this Parameter:

```
apiCall :: ∀ request response m
  .  MonadAff m
  => Encode request
  => Decode response
  => request ❶ ❷
  -> m (Either String response)
apiCall request = do ❸
  ajaxResult <- H.liftAff $ Ajax.post ResponseFormat.string
    "http://localhost:3000/"
    $ Just $ RequestBody.String
    $ encodeJSON request
  pure do
    { body } <- lmap Ajax.printError ajaxResult
    lmap show $ runExcept $ decodeJSON body :: _ response
```

❶ Removed the `UUID` Type Parameter.

❷ Changed `->` to `=>`.

❸ Removed `authToken` Parameter.

We also can delete the import:

```
import Data.UUID (UUID)
```

Looking at `apiCall`, I'm wondering if the explicit Type Signature on the following line is really necessary anymore:

```
    lmap show $ runExcept $ decodeJSON body :: _ response
```

Let's take it out and see if the compiler can figure it out without our help.

Turns out that it can:

```
apiCall :: ∀ request response m
  .  MonadAff m
  => Encode request
  => Decode response
  => request
  -> m (Either String response)
apiCall request = do
  ajaxResult <- liftAff $ Ajax.post ResponseFormat.string
    "http://localhost:3000/"
    $ Just $ RequestBody.String
    $ encodeJSON request
  pure do
    { body } <- lmap Ajax.printError ajaxResult
    lmap show $ runExcept $ decodeJSON body ❶
```

❶ Removed explicit Type Signature.

Now that this compiles, create a File `Utils.purs` under the `src` directory.

Move `apiCall` to `Utils` along with the necessary imports. Note, that `H.liftAff` can be changed to plain old `liftAff` once `apiCall` is moved. `Halogen` module just re-exports `liftAff` for sake of convenience.

Here's the changes to `Utils`:

```
module Utils where

import Prelude

import Affjax as Ajax ❶
import Affjax.ResponseFormat as ResponseFormat
import Affjax.RequestBody as RequestBody
import Control.Monad.Except (runExcept)
import Data.Bifunctor (lmap)
import Data.Either (Either)
import Data.Maybe (Maybe(..))
import Effect.Aff.Class (class MonadAff, liftAff) ❷
import Foreign.Generic (encodeJSON, decodeJSON)
import Foreign.Generic.Class (class Encode, class Decode)

apiCall :: ∀ request response m
  .  MonadAff m
  => Encode request
  => Decode response
  => request
  -> m (Either String response)
apiCall request = do
  ajaxResult <- liftAff $ Ajax.post ResponseFormat.string ❸
    "http://localhost:3000/"
    $ Just $ RequestBody.String
    $ encodeJSON request
  pure do
    { body } <- lmap Ajax.printError ajaxResult
    lmap show $ runExcept $ decodeJSON body :: _ response
```

❶ Added these imports.

❷ Using `liftAff` from `Effect.Aff.Class` instead of the one that's re-exported from `Halogen`.

❸ Changed `H.liftAff` to `liftAff`.

Now, return to the Users Page and import `Utils` and use `apiCall` in `handleAction` before reading on.

Here's that change:

```
import Utils (apiCall) ❶

  handleAction :: Action -> H.HalogenM State Action Slots Output m Unit
  handleAction = case _ of
    Initialize -> do
      { userRef } <- ask
      loggedOnUser' <- H.liftEffect $ Ref.read userRef
      loggedOnUser' # maybe (pure unit) \{ authToken, admin } ->
        when admin do
          queryResponse <-
            apiCall (QueryUsersRequest { authToken }) ❷
          case queryResponse of
            Left err -> alertError err
            Right (QueryUsersResponse
                (QueryUsersResultsFailure { reason })) ->
              alertError $ "Query Users: " <> case reason of
                NotAuthorized -> "Not Authorized"
                NotAuthenticated -> "Not Authenticated"
            Right (QueryUsersResponse
                (QueryUsersResultsSuccess { users })) ->
              H.modify_ _ { authorized = true , users = users }
```

❶ Import `apiCall`.

❷ Using `apiCall` in place of copy/pasted code.

Delete the following imports:

```
import Affjax as Ajax
import Affjax.RequestBody as RequestBody
import Affjax.ResponseFormat as ResponseFormat
import Control.Monad.Except (runExcept)
import Data.Bifunctor (lmap)
import Foreign.Generic (encodeJSON, decodeJSON)
import Foreign.Generic.Class (class Encode, class Decode)
```

Now do the same on the Logon Page, i.e. `Component.Logon` and then continue reading.

Here's that change:

```
import Utils (apiCall) ❶

  handleAction :: Action -> H.HalogenM State Action Slots Output m Unit
  handleAction = case _ of
    Input f -> H.modify_ f
    Logon -> do
      { userName, password } <- H.get
      logonResponse <-
        apiCall (LogonRequest { userName, password }) ❷
      case logonResponse of
        Left err -> alertError err
        Right (LogonResponse LogonResultsFailure) ->
          alertError "Invalid Logon Credentials"
        Right (LogonResponse (LogonResultsSuccess
          { authToken, admin, mustChangePassword })) -> do
            log =<< logEntry Info "User logged on"
            { userRef } <- ask
            H.liftEffect $ Ref.write (Just { admin, authToken }) userRef
            navigate <=< logonRoute
              $ if mustChangePassword then PasswordTemporary
                else PasswordPermanent
```

❶ Import `apiCall`.

❷ Using `apiCall`. Removed code that's duplicated in `apiCall`.

And don't forget to remove the following imports:

```
import Affjax as Ajax
import Affjax.ResponseFormat as ResponseFormat
import Affjax.RequestBody as RequestBody
import Data.Bifunctor (lmap)
import Control.Monad.Except (runExcept)
import Foreign.Generic (encodeJSON, decodeJSON)
```

Step back for a moment and notice how much nicer `handleAction` looks with `apiCall`. Refactoring is a pain in the moment but it always leave our codebase looking so much better.

This will also make calling other APIs on other Pages so much easier.

Before we can start coding the `render` Function for the Users Page, I'd like to make some changes to our Page Component that'll give us some more control over styling.

When we originally created the Page Component, we centered the content:

```
-- ABOVE CODE REDACTED
    , HH.div [
        HC.style do
          display flex
          alignItems center ❶
          justifyContent center ❷
          minHeight $ vh 90.0
      ]
      [ HH.slot _inner unit innerComponent iInput Output ]
-- BELOW CODE REDACTED
```

❶ We align the items centered on the Vertical.

❷ We justify the content centered on the Horizontal.

That's fine for the pages we've made so far, but it would be nice if we could have more control over the styling of the `div` that holds the unique portion of the Page, i.e. what we called the Inner Component.

We're going to modify the code in the Page Component, i.e. `Component.Page` module, to make this possible.

First thing we'll do is pass a `StyleM` to the Component upon creation:

```
import CSS.Stylesheet (StyleM) ❶

component
  :: ∀ iQuery iInput iOutput m
   . MonadAff m
  => Navigate m Route
  => StyleM Unit ❷
  -> H.Component iQuery iInput iOutput m ❸
  -> H.Component iQuery iInput iOutput m
component style innerComponent = H.mkComponent ❷ ❹
-- BELOW CODE REDACTED
```

❶ Added this import.

❷ Added the `style` Parameter of Type `StyleM`, which is the Monad that is represented by the `do` we see after the `HC.style`.

❸ Changed `=>` to `->`.

❹ The fact that `render` is under the `where` of `component` means that we don't have to pass `style` to it since these Parameters are in scope. This is why I always put `handleAction`, `handleQuery` and `render` under the `where`.

Next, we're going to use `style` in `render` to allow us to override the styling:

```
-- ABOVE CODE REDACTED
    , HH.div [
        HC.style $ (do ❶
          display flex
          alignItems center
          justifyContent center
          minHeight $ vh 90.0) *> style ❶ ❷
      ]
      [ HH.slot _inner unit innerComponent iInput Output ]
    ]
```

❶ You need to add both the `$` and the Parentheses.

❷ We `applySecond` our `style` overrides LAST so that conflicts will choose ours over the default, e.g. when we specify `justifyContent flexStart`, which conflicts with what's specified here, ours will be the last CSS `Rule` that Halogen sees and, hence, uses.

Make sure your code is up to date with this.

Now, we're going to modify the `Router`, i.e. `Component.Router`, to pass `styles` to the Page Component and while we're here, we're going to call `Users.component` even though it's `render` Function isn't complete:

```
import Component.Users as Users ❶
import CSS.Flexbox (flexStart, alignItems, stretch, justifyContent)
import CSS.Geometry (padding)
import CSS.Size (rem)

  render :: State -> H.ComponentHTML Action Slots m
  render { route } = case route of
    Logon -> HH.slot_ _logon unit (defaultPage Logon.component) unit ❷
    Logoff -> HH.span [ HC.style $ color white ] [ HH.text "Logoff" ]
    Users _ -> HH.slot_ _users unit (wholePage Users.component) unit ❸
    ChangePassword -> HH.slot_ _changePassword unit
      (defaultPage ChangePassword.component) unit ❷
    where
      defaultPage = Page.component $ pure unit ❹
      wholePage = Page.component do ❺
        alignItems stretch
        justifyContent flexStart
        padding (rem 2.0) (rem 2.0) (rem 2.0) (rem 2.0)
```

❶ Added these imports.

❷ Using default Page settings with no overrides.

❸ Using whole Page settings with some overrides.

❹ `pure unit` is an empty `StyleM`.

❺ The `do` block is the first Parameter to `Page.component` of Type `StyleM`. We override the Horizontal justification and put a `2 rem` padding all around the `div` that contains the Inner Component.

Now, with those improvements out of the way, we're ready to code the `render` Function for the Users Page, i.e. `Component.Users`.

So, we don't waste a bunch of time with layout and styling, I'm just going to give you a starting point for `render`. As you're typing it in, try to pay attention to what the code's doing:

```
import AppTheme (paperColor) ❶
import CSS.Background (backgroundColor)
import CSS.Cursor (cursor, pointer)
import CSS.Display (display, flex)
import CSS.Flexbox (flexDirection, row, column, flexGrow)
import CSS.Geometry (minWidth, paddingRight)
import CSS.Size (rem)
import Entity.User (User(..))
import Halogen.HTML.Core (ClassName(..))

  render :: State -> H.ComponentHTML Action Slots m
  render { authorized, users } = ❷
    if not authorized then HH.text "NOT AUTHORIZED" else ❸
    HH.div [
      HC.style do
        display flex
        flexDirection row ❹
        flexGrow 1.0
    ]
    [ HH.div [ ❺
      HC.style do
        display flex
        flexDirection column
        minWidth (rem 20.0)
        paddingRight (rem 2.0)
      ]
      [ HH.ul_ ❻
        (users <#> \(User { userName }) -> ❼
          HH.li [
            HP.class_ $ ClassName "list-group-item" ❽
          , HC.style do
              backgroundColor paperColor
              cursor pointer
          ]
          [ HH.text userName ])
      ]
    ]
```

❶ Added these imports except for `Entity.User`, which `(..)` was added.

❷ Destructure `State`.

❸ We first make sure that the the user is authorized to view this page and that they didn't just type in the URL to this location in the Address Bar or use a link they got from someone who does have authorization.

❹ Our main layout for this page will be a `row`.

❺ This is our first `div` in this `row`. It's the list of `users`. The second `div` isn't coded yet, but will display the selected user's information.

❻ Using `ul_` variant since it has NO Properties.

❼ We map over the `users` to produce `li` tags.

❽ We're using Bootstrap's `class` (more on this soon).

Now, we'd like to leverage Bootstrap's styling. At the time of this writing, version 4.5.3 was the latest version of Bootstrap. The CDN location changes from time to time, so you may need to go to the Bootstrap web site to get the latest link.

You can also just search via Google for `bootstrap cdn link`.

We're going to modify our `index.html` to get Bootstrap from a CDN (Content Delivery Network):

```
<!DOCTYPE html>
<html lang="en" dir="ltr">

<head>
  <meta charset="utf-8">
  <title>Functional Programming Made Easier</title>
  <link rel="stylesheet"
    href=
    "https://stackpath.bootstrapcdn.com/bootstrap/4.3.1/css/bootstrap.min.css"
    integrity=
    "sha384-ggOyR0iXCbMQv3Xipma34MD+dH/1fQ784/j6cY/iJTQUOhcWr7x9JvoRxT2MZw1T"
    crossorigin="anonymous"> ❶
</head>

<body style="margin:0; background-color: #383333">
  <script src="/index.js" charset="utf-8"></script>
</body>

</html>
```

❶ Add this `link` tag. Search Goolge for `bootstrap cdn link` if you don't want to type this.

And with that, you should be able to build the `client`:

```
make build
```

We're going to ignore the warning that we're not using `Halogen.HTML.Events as HE` in `Users.pur` since

we're going to use it soon enough.

Now, go to `localhost:3000` in your Browser, refresh the page, logon and click on the `Users` button on the ChangePassword Page. You should see something like:



We are getting the benefits of `Bootstrap's` styling for our list of users.

Next, step is to handle the selection of users on the Users Page. We need to create an `Action` that will be called `UserSelected` which takes a `User` as its one and only Parameter.

Add this `Action` and the code in `render` of `Component.Users` to add the click event Function. Just stub out the code in `handleAction` for now. Then return here when you're ready to move on.

Here is the change to `Action`:

```
data Action
  = Initialize
  | UserSelected User ❶
```

❶ Added this.

Here's the stubbed out code in `handleAction`:

```
-- ABOVE CODE REDACTED
    UserSelected _ -> pure unit
-- BELOW CODE REDACTED
```

And now here's the code for adding the click Function to each list item:

```
-- ABOVE CODE REDACTED
      [ HH.ul_
          (users <#> \user@(User { userName }) -> ❶
            HH.li [
              HP.class_ $ ClassName "list-group-item"
            , HC.style do
                backgroundColor paperColor
                cursor pointer
            , HE.onClick $ const $ UserSelected user ❷
            ]
            [ HH.text userName ])
      ]
    ]
```

❶ Named the Parameter `user` using the `@` syntax.

❷ Added `HE.onlick`. The `const` is to throw away the `MouseEvent` and just return an `Action`.

Now, go back to `handleAction` and write the code to handle the selection of a `User` by navigating to `Route.User`. Don't forget that `Route.User's` first Parameter is a `String`, i.e. the `userName`, which uniquely identifies the user that is selected.

Then when you're done, return here and continue reading.

First, we'll make the change to `handleAction` to `navigate`:

```
import Capability.Navigate (class Navigate, navigate) ❶
import Data.Route as Route ❷

-- ABOVE CODE REDACTED
    UserSelected (User { userName }) -> ❸
      navigate $ Route.Users $ Just userName ❹
-- BELOW CODE REDACTED
```

❶ Added `navigate` to the import list.

❷ Added this import.

❸ Destructure `User` to get at `userName`.

❹ Navigate to the `Users` route with the `userName`, which will dictate which user in the list of users is highlighted and whose information will be displayed to the right of the list of users.

Next, we have to handle the `userName` from the `Route` in the `render` Function in `Component.Router`.

Currently, we don't do anything with it. Here's the current code for reference:

```
-- ABOVE CODE REDACTED
  render :: State -> H.ComponentHTML Action Slots m
  render { route } = case route of
    Logon -> HH.slot_ _logon unit (defaultPage Logon.component) unit
    Logoff -> HH.span [ HC.style $ color white ] [ HH.text "Logoff" ]
    Users _ -> HH.slot_ _users unit ❶
      (wholePage Users.component) unit
    ChangePassword -> HH.slot_ _changePassword unit
      (defaultPage ChangePassword.component) unit
-- BELOW CODE REDACTED
```

❶ Notice how we throw away the first Parameter of the Pattern Match.

Next, change this line of code in the `render` Function in `Component.Router` so it passes `userName` as `Input` to our Users Component. Then return here.

Here's the change to `render`:

```
-- ABOVE CODE REDACTED
    Users userName' -> HH.slot_ _users unit ❶ ❷
      -- COMPILER ERROR!!
      (wholePage Users.component) userName' ❸
-- BELOW CODE REDACTED
```

❶ I like to name my Variables with Primes to remind me that they're a `Maybe` or an `Either`.

❷ Remember that `userName'` comes from the parsing of the Hash Route in the URL.

❸ Pass `userName'` as `Input` to `Users.component` instead of `unit`.

Notice how we pass the `userName'` as `Input` to the component and NOT as a Parameter to the `Users.component` Function. This is because the component will not be Finalized when we change `Routes` due to selecting a `User` from the list.

The only time your Component gets Finalized is when it currently exists in the DOM and upon the next render of your Parent, it doesn't exist in the VDOM (Virtual DOM). This will not be the case when we click on a new `User`, which is good since we don't want to query for all of the users every time we select a different user in the UI.

If we passed the `userName` as a Parameter to the `Users.component` Function, the `initialize` would be called when the component was initially created. This would give us a chance to initialize our `State` with `userName`. But the problem is that subsequent calls to `Users.component` will NOT call `initialize` since the component already exists. And passing the `userName` as a Parameter will NOT allow us to update our `State` meaning that our `render` Function will never be called to display the newly selected user.

Understanding these Lifecycle issues is very important for creating both a performant Application, as well as a properly functioning one.

To fix the compiler error, the Users Component needs to accept `Input` from its Parent, the Router Component. Remember this is handled by the `receive` Function in the `EvalSpec`.

First, create an `Action` in `Component.Users` called `SelectedUserName` that takes a `Maybe String` and then stub out the handling of it in `handleAction`. And second, implement `receive` in the `EvalSpec`. Then return here and continue reading.

Here's the first part:

```
type Input = Maybe String ❶

data Action
  = Initialize
  | UserSelected User
  | SelectedUserName (Maybe String) ❷

-- ABOVE CODE REDACTED
  handleAction :: Action -> H.HalogenM State Action Slots Output m Unit
  handleAction = case _ of
-- CODE REDACTED
    UserSelected (User { userName }) -> navigate $ Route.Users $ Just userName
    SelectedUserName _ -> pure unit ❸
```

❶ Change from `Unit` to `Maybe String`.

❷ Added `Action` to handle `Input` from our Parent.

❸ Stubbed out code for handling our new `Action`.

Here's the second part:

```
component
  :: ∀ m
  .  MonadAff m
  => MonadAsk Env m
  => Navigate m Route
  => H.Component Query Input Output m
component = H.mkComponent
  { initialState: \_ -> ❶
      { authorized: false
      , selectedUser: Nothing
      , users: []
      }
  , render
  , eval: H.mkEval H.defaultEval
      { handleAction = handleAction
      , receive = Just <<< SelectedUserName ❷
      , initialize = Just Initialize
      }
  }
```

❶ We're currently throwing away the `Input` we get initially since we have no users. We'll need to fix this.

❷ `Input` from our Parent produces the `Action`, `SeletedUserName`.

Notice that `initialState` isn't using the `userName'` that it gets from its Parent, i.e. the Router Component.

We cannot look up the `user` here since we don't have any users yet since we haven't called the backend yet to get them.

We'll need to store this Value in the `State` so when we do get the `users` from our server API, we can then select the current user, if there is one.

Make that change to the code to simply store the initial `username` calling our new field `initUserName` before reading on.

Here's that change:

```
type State =
  { authorized :: Boolean
  , selectedUser :: Maybe User
  , users :: Array User
  , initUserName :: Maybe String ❶
  }

-- ABOVE CODE REDACTED
component = H.mkComponent
  { initialState: \initUserName -> ❷
      { authorized: false
      , selectedUser: Nothing
      , users: []
      , initUserName ❸
      }
-- BELOW CODE REDACTED
```

❶ Added initial user name field that might be `Nothing` depending on the initial URL Hash path.

❷ We no longer throw away the `Input` Parameter.

❸ We initialize our new field with `initUserName` that we get from the Router Component, i.e. our Parent.

We now need to change the `Initialize` case in `handleAction` to update `selectedUser` AFTER it successfully gets the `users` back from the `QueryUsersRequest` API call.

Let's start by coding:

```
-- ABOVE CODE REDACTED
            Right (QueryUsersResponse
                (QueryUsersResultsSuccess { users })) ->
              H.modify_ _
                { authorized = true
                , users = users
                , selectedUser = initUserName <#> \userName -> ??? ❶
                }
-- BELOW CODE REDACTED
```

❶ We need to find the user with the specified `userName`.

It should be obvious that I made a mistake when I made `users` an `Array`:

```
type State =
  { authorized :: Boolean
  , selectedUser :: Maybe User
  , users :: Array User
  , initUserName :: Maybe String
  }
```

It would be so much better if this was a `Map` whose key was `userName`.

Comment out this unfinished line of code and fix my mistake by changing `users` in `State` to be a `Map` and then fix the code anywhere that this change breaks by using `Map` instead `Array`.

Then return here and continue reading.

First, we'll change `State`:

```
import Data.Map (Map) ❶

type State =
  { authorized :: Boolean
  , selectedUser :: Maybe User
  , users :: Map String User ❷
  , initUserName :: Maybe String
  }
```

❶ Added this import.

❷ Changed from `Array` to `Map`.

Next, we'll fix `render` to work with the Values of the `Map` of `users`:

```
import Data.Array (fromFoldable) ❶
import Data.Map as Map

  render :: State -> H.ComponentHTML Action Slots m
  render { authorized, users, selectedUser } = ❷
    -- CODE REDACTED
          (Map.values users # fromFoldable ❸
            <#> \(user@(User { userName })) ->
              HH.li [
                HP.class_ $ ClassName "list-group-item"
              , HC.style do
                  backgroundColor paperColor
                  cursor pointer
              , HE.onClick $ const $ UserSelected user
              ]
              [ HH.text userName ])
-- BELOW CODE REDACTED
```

❶ Added these imports.

❷ Destructure `selectedUser` also.

❸ Use just the `values` from the `Map`. We have to convert the `List` that we get back from `Map.values` into `Array` using `fromFoldable`.

Next, we'll fix `initialize`:

```
-- ABOVE CODE REDACTED
component = H.mkComponent
  { initialState: \initUserName ->
      { authorized: false
      , selectedUser: Nothing
      , users: Map.empty ❶
      , initUserName
      }
  , render
  , eval: H.mkEval H.defaultEval
      { handleAction = handleAction
      , receive = Just <<< SelectedUserName
      , initialize = Just Initialize
      }
  }
```

❶ Changed from empty `Array` to empty `Map`.

And finally, we'll convert the `Array` of `users` we get back from `QueryUsersRequest` into a `Map`:

```
import Data.Foldable (foldl) ❶
import Data.Newtype (unwrap)

-- ABOVE CODE REDACTED
            Right (QueryUsersResponse
                (QueryUsersResultsSuccess { users })) ->
              let mkMap f =
                    foldl (\m r -> Map.insert (f r) r m) Map.empty in ❷
              H.modify_ _
                { authorized = true
                -- COMPILER ERROR!!
                , users = mkMap (_.userName <<< unwrap) users ❸
                -- , selectedUser = initUserName <#> \userName ->
                }
-- BELOW CODE REDACTED
```

❶ Added these imports.

❷ `mkMap` will take an Array of Records and use `f` to produce a Key from each Record. It will then add that Key and the Record as the Value to the `Map`.

❸ Using `unwrap` to get contained Record from `User`.

We'll get a compiler error since we've used `unwrap` in the above code. This is because `User` doesn't have a `Newtype` Instance.

So, we'll have to go the the `server` code where `User` is defined and make a it has a `Newtype` Instance:

```
import Prelude ❶
import Data.Newtype (class Newtype)

newtype User = User (Record (UserRow ()))
derive instance genericUser :: Generic User _
derive instance newtypeUser :: Newtype User _ ❷
```

❶ Added these imports.

❷ Derive `Newtype`.

Now return to the `client` code and run:

```
make build
```

This will build the code from our newly changed code from the `server`.

But this will produce an error:

```
[error] Some of your project files import modules from packages that are not
in the direct dependencies of your project.
To fix this error add the following packages to the list of dependencies in
your config:
- arrays
- foldable-traversable
- newtype
- ordered-collections
```

Add these to spago and then re-build:

```
make build
```

Then return to `Component.Users` and save the file and the compiler error should disappear.

Now that my bad decision to make `users` an `Array` is fixed, we can return to that line of code we commented out:

```
-- ABOVE CODE REDACTED
            Right (QueryUsersResponse
                (QueryUsersResultsSuccess { users })) ->
              let mkMap f = foldl (\m r -> Map.insert (f r) r m) Map.empty in
              H.modify_ _
                { authorized = true
                , users = mkMap (_.userName <<< unwrap) users
                -- , selectedUser = initUserName <#> \userName -> ❶
                }
-- BELOW CODE REDACTED
```

❶ This line is unfinished.

Uncomment the unfinished line of code and finish coding it to look up the `initUserName` in the `Map` of `users`. Once you've done that, return back here and continue.

Here's that change:

```
          Right (QueryUsersResponse
              (QueryUsersResultsSuccess { users })) -> do ❶
            let mkMap f = foldl (\m r -> Map.insert (f r) r m) Map.empty ❷
                usersMap = mkMap (_.userName <<< unwrap) users ❸
            { initUserName } <- H.get ❹
            H.modify_ _
              { authorized = true
              , users = usersMap ❺
              , selectedUser = initUserName
                  >>= \userName -> Map.lookup userName usersMap ❻ ❼
              }
```

❶ Had to add a `do` block here since we're now calling `H.get` along with `H.modify_`.

❷ Removed the `in` keyword since we're in a `do` block now.

❸ Created Variable `usersMap` since we need this in 2 computations.

❹ Get the `initUserName` from our `State`.

❺ Set users to our `Map` of `users`.

❻ Look up the `initUserName` to see if it exists and if so, then that's our selected user.

❼ Notice the use of `bind` here. That's because `Map.lookup` can return a `Maybe`. This is why `map` wasn't used.

Now, we're ready to write the implementation for `SelectedUserName` in `handleAction`.

It should set `selectedUser` appropriately in the `State`. Make this change before reading on.

Here's that change:

```
-- ABOVE CODE REDACTED
    SelectedUserName userName' -> do
      { users } <- H.get ❶
      H.modify_ _ { selectedUser = userName'
        >>= \userName -> Map.lookup userName users } ❷
```

❶ Get `users` from the `State` for the `Map.lookup`.

❷ Set `selectedUser` in the `State` accordingly.

And now, we'll make the selected `User` render differently.

Let's start with `selectedColor` in `AppTheme.purs`. We'll use `themeColor` with a `25%` transparency, i.e. `0.75` opacity Value for the ***Alpha Channel***:

```
import CSS.Color (Color, rgb, rgba) ❶

selectedColor :: Color
selectedColor = rgba 0x00 0x66 0x75 0.75 ❷
```

❶ Added `rgba` to import list. The `a` in the name stands for `Alpha` Channel. It represents an opacity level.

❷ We stole the `rgb` Values from `themeColor` but if we change `themeColor`, this won't change.

It would be nice if we could compute `selectedColor` from `themeColor` instead of copying and pasting the raw Values.

Let's use `toRGBA` to extract the `rgb` Values from `themeColor`:

```
toRGBA :: Color -> { a :: Number, b :: Int, g :: Int, r :: Int } ❶
```

❶ `toRGBA` takes a `Color` and gives us its `RGBA` components. For use, we'll use this to get just `RGB` from `themeColor`.

And now, here's that change:

```
import CSS.Color (Color, rgb, rgba, toRGBA) ❶

selectedColor :: Color
selectedColor = themeColor # toRGBA # \{ r, g, b } -> rgba r g b 0.75
```

❶ Added `toRGBA` to import list.

That's better. Now if someone changes `themeColor`, `selectedColor` will also change accordingly.

Now, we'll change render to use the `selectedColor`:

```
import AppTheme (paperColor, selectedColor) ❶

  render :: State -> H.ComponentHTML Action Slots m
  render { authorized, selectedUser, users } = ❷
-- CODE REDACTED
      [ HH.ul_
          (Map.values users # fromFoldable
            <#> \(user@(User { userName })) ->
              HH.li [
                HP.class_ $ ClassName "list-group-item"
              , HC.style do
                  backgroundColor
                    -- COMPILER ERROR!!
                    if Just user == selectedUser then selectedColor ❸
                    else paperColor
                  cursor pointer
              , HE.onClick $ const $ UserSelected user
              ]
              [ HH.text userName ])
      ]
    ]
```

❶ Added `selectedColor` to import list.

❷ Added `selectedUser` to the destructuring Pattern.

❸ The Parameter to `backgroundColor` is conditionally `paperColor` or this new, yet-to-be-defined `selectedColor`.

To fix the compiler error, we need to make `User` have and `Eq` instance. Remember, `User` is defined in the `server` code.

Change to your `server` code and derive `Eq` for `User` then return here.

Here's that changes:

```
import Prelude ❶

newtype User = User (Record (UserRow ()))
derive instance genericUser :: Generic User _
derive instance newtypeUser :: Newtype User _
derive instance eqUser :: Eq User ❷
```

❶ Added to get `Eq` class.

❷ Derive `Eq` Instance.

Return to the `client` project and rebuild with:

```
make build
```

Save the `Users.purs` file to make the compiler error go away (assuming your in VSCode).

Now refresh the browser, logon and click on the `Users` button on the ChangePassword Page. Then select a `User` and see how it looks.

When I do this, the text is really hard to read. Change the `render` code to change the font color from `black` to `white` when a `User` is selected.

Return here once you're are ready to continue.

Here's that change:

```
import CSS.Color (white, black) ❶
import CSS.Font (color)

-- ABOVE CODE REDACTED
      [ HH.ul_
          (Map.values users # fromFoldable
            <#> \(user@(User { userName })) ->
              let isSelected = Just user == selectedUser in ❷
              HH.li [
                HP.class_ $ ClassName "list-group-item"
              , HC.style do
                  backgroundColor
                    if isSelected then selectedColor ❸
                    else paperColor
                  color ❹
                    if isSelected then white
                    else black
                  cursor pointer
              , HE.onClick $ const $ UserSelected user
              ]
              [ HH.text userName ])
      ]
    ]
```

❶ Added these imports.

❷ Factored out this common logic.

❸ Changed this to use `isSelected`.

❹ Added this style to change the text `color` from `black` to `white` when selected.

Now, rebuild the Application and retest and see how it looks.

I think our change looks much better.

And the final step for our Users Page is to display the `User` information to the right. Since this part of the code is pretty much just HTML and CSS, I'm just going to give it to you whole.

Please take the time to try to understand what it's doing:

```
import CSS.Geometry (minWidth, padding, paddingBottom, paddingRight) ❶
import CSS.Flexbox (flexDirection, row, column, flexGrow, flexBasis
                  , justifyContent, alignItems, flexStart) ❷
import CSS.Size (rem, pct) ❸

-- ABOVE CODE REDACTED
      [ HH.ul_
          (Map.values users # fromFoldable
            <#> \(user@(User { userName })) ->
              let isSelected = Just user == selectedUser in
              HH.li [
                HP.class_ $ ClassName "list-group-item"
              , HC.style do
                  backgroundColor
                    if isSelected then selectedColor
                    else paperColor
                  color
                    if isSelected then white
                    else black
                  cursor pointer
              , HE.onClick $ const $ UserSelected user
              ]
              [ HH.text userName ])
      ]
    , selectedUser # maybe (HH.text "") \(User user) -> ❹
        let item h = HH.div [ HC.style $ paddingBottom (rem 0.5) ] [ h ] in ❺
        HH.div [
            HC.style do
              display flex
              flexDirection row
              flexBasis (pct 100.0)
              backgroundColor paperColor
```

```
                padding (rem 2.0) (rem 2.0) (rem 2.0) (rem 2.0)
            ]
            [ HH.div [
                HC.style do
                  display flex
                  flexDirection column
                  justifyContent flexStart
                  alignItems flexStart
                  minWidth (rem 10.0)
              ]
              [
                item $ HH.text "User Name:"
              , item $ HH.text "Name:"
              , item $ HH.text "Administrator:"
              ]
            , HH.div [
                HC.style do
                  display flex
                  flexDirection column
                  justifyContent flexStart
                  alignItems flexStart
              ]
              [
                item $ HH.text user.userName
              , item $ HH.text $ user.firstName <> " " <> user.lastName
              , item $ HH.text if user.admin then "Yes" else "No"
              ]
          ]
    ]
```

❶ Added `padding` and `paddingBottom` to import list.

❷ Added `flexBasis`, `justifyContent`, `alignItems` and `flexStart` to import list.

❸ Added `pct` to import list.

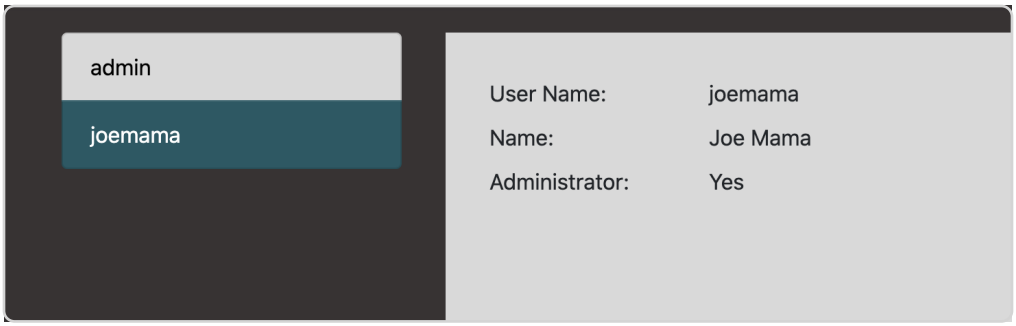
❹ Only display the right-hand side if there's a `selectedUser`. Otherwise, put an empty Text Node via `HH.text ""`.

❺ `item` is a simple helper Function for defining items.

Note that this code is added to the END of the `render` Function.

Make sure you update your code and then rebuild and test your Application.

Here's what it should look like when you run it:

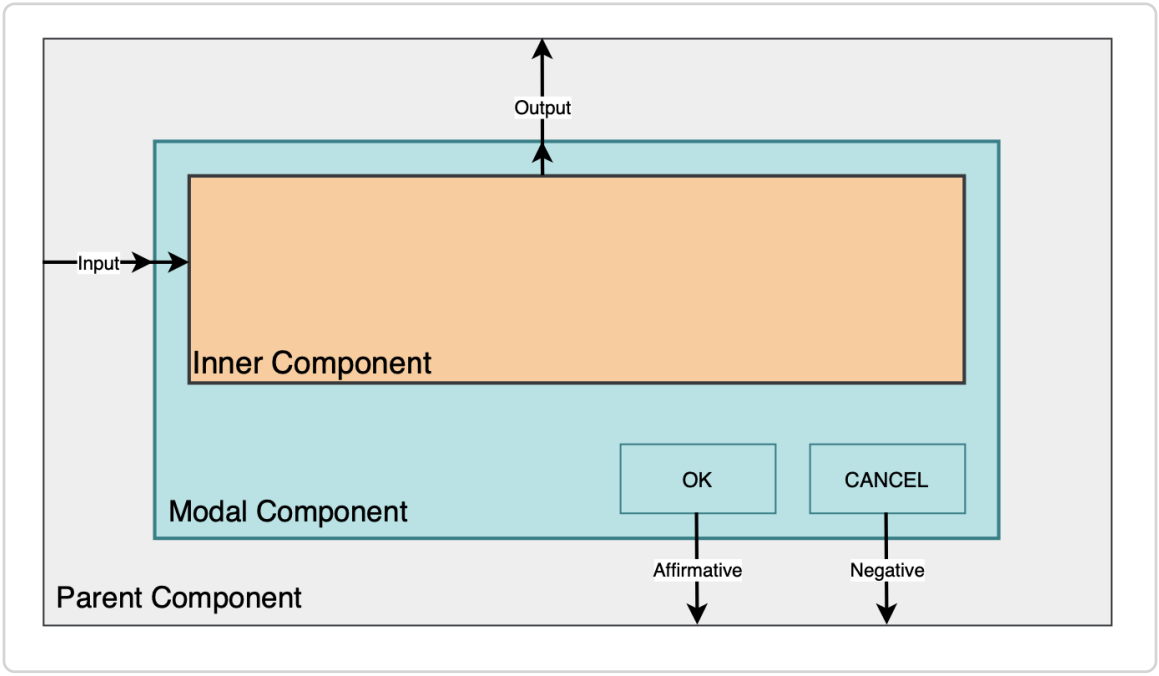


The last thing we need to do on this page is to Create a new `User`. But to do that, we're going to need a Modal Dialog Box first.

30.12. Modal Dialog Component

We're going to build a Modal Dialog Component called `Modal`. It will work similar to our Page Component in that it'll have an Inner Component. The Inner Component can be anything, e.g. a Form like we'll need for the Create User Modal or just Text as we'll need for our error Dialogs that will replace the use of `alertError`.

Let's start with a design:



Input from the Parent to the Modal Component will be passed directly to the Inner Component. And Output from the Inner Component will be passed directly to the Parent.

The Modal Component will have 2 buttons with default labels `OK` and `CANCEL`, which will produce the Outputs `Affirmative` and `Negative` respectively. We'll allow some configuration on our Modal Component, i.e. re-labeling the buttons and disabling the Cancel Button.

This is going to be pretty complicated compared to other Components that we've done so far, and more

complicated than the average Component. Because of that, we'll code the Modal Component and the Message Component together. This should be enough experience to allow you to code the Create User Modal on your own.

Let's start by creating a File under `src/Component` called `Modal.purs` with the following:

```
module Component.Modal where

import Prelude

import Effect.Aff.Class (class MonadAff)
import Halogen as H ❶
import Halogen.HTML as HH
import Halogen.HTML.CSS as HC
import Halogen.HTML.Events as HE

type State iInput = { iInput :: iInput } ❷
```

❶ We'll need these Halogen imports eventually.

❷ Our `State` is similar to `Page's` for all the same reasons. We need to save the `Input` in the `State` so we can pass it to the Inner Component in `render` as one of the `HH.slot` Parameters.

Consulting our design diagram, it's clear that there are 3 `Outputs`, so we'll define those:

```
data Output iOutput ❶
  = Affirmative
  | Negative
  | InnerOutput iOutput ❷
```

❶ `iOutput` is the Type for the Inner Component's `Output`.

❷ `InnerOutput` wraps the Inner Component's `Output` in our `Output`.

`Affirmative` and `Negative` are to let our Parent know that the buttons have been clicked.

Returning again to our diagram, we can see that there are 4 `Actions`, one for each arrow:

```
data Action iInput iOutput
  = Input iInput ❶
  | Output iOutput ❷
  | AffirmativeClicked ❸
  | NegativeClicked ❸
```

❶ `Input` from Parent.

❷ `Output` from Inner Component.

❸ Button clicks.

We're reacting to `Input` from the Parent, `Output` from the Inner Component and the user clicking on buttons.

Next, we define the Child Slots Type and Symbol Proxies for the labels:

```
import Type.Proxy (Proxy(..)) ❶

type Slots iQuery iOutput = ( inner :: H.Slot iQuery iOutput Unit )

_inner = Proxy :: Proxy "inner"
```

❶ Added this import.

This code is like what we did for the Page Component.

And now we're ready to code the `component` creation Function. We'll start with the Type Signature:

```
component
  :: ∀ iQuery iInput iOutput m
   . MonadAff m ❶
  => H.Component iQuery iInput iOutput m ❷
  -> H.Component iQuery iInput (Output iOutput) m ❸
```

❶ We always need this when we run in a Monad that eventually runs in `Aff`.

❷ The Inner Component.

❸ Notice how our `Query` and `Input` Type Parameters are exactly the same Type as our Inner Component's. That's why we didn't define them in this module.

And now, we call `H.mkComponent` to make the Component:

```
component
  :: ∀ iQuery iInput iOutput m
   . MonadAff m
  => H.Component iQuery iInput iOutput m
  -> H.Component iQuery iInput (Output iOutput) m
component innerComponent = H.mkComponent
  { initialState: \iInput -> { iInput } ❶
  , render
  , eval: H.mkEval $ H.defaultEval
      { handleAction = handleAction ❷
      , handleQuery = handleQuery ❸
      , receive = Just <<< Input ❹
      }
  }
  where
  ???
```

❶ We initialize our `State` with the Input from the Parent.

❷ We're going to need to handle our `Actions`.

❸ Queries will simply be forwarded to the Inner Component like we did in the Page Component.

❹ We will handle Input from the Parent the same way we did in the Page Component, i.e. Inputs are passed to the Inner Component by `render`, which were put into the `State` by `handleAction`.

Now, we ready to write `handleAction's` Type Signature:

```
-- ABOVE CODE REDACTED
  where
  handleAction
    :: Action iInput iOutput
    -> H.HalogenM (State iInput) (Action iInput iOutput)
                  (Slots iQuery iOutput) (Output iOutput) m Unit ❶
```

❶ This Type differs only slightly from what we did in the Page Component. In `Page`, our Output was the SAME as our Inner Component. In `Modal`, we have our OWN `Output` so we define the Output as `Output iOutput`, whereas `Page` defined it as just `iOutput`.

Next, we'll write the boilerplate part of `handleAction`:

```
-- ABOVE CODE REDACTED
  where
  handleAction
    :: Action iInput iOutput
    -> H.HalogenM (State iInput) (Action iInput iOutput)
                  (Slots iQuery iOutput) (Output iOutput) m Unit
  handleAction = case _ of
    Input input -> ???
    Output output -> ???
    AffirmativeClicked -> ???
    NegativeClicked -> ???
```

For `Input`, i.e. Input from our Parent, we'll just update our `State` and let `render` pass it to the Inner Component in the `HH.slot` call:

```
-- ABOVE CODE REDACTED
  where
  handleAction
    :: Action iInput iOutput
    -> H.HalogenM (State iInput) (Action iInput iOutput)
                  (Slots iQuery iOutput) (Output iOutput) m Unit
  handleAction = case _ of
    Input input -> H.modify_ _ { iInput = input }
    Output output -> ???
    AffirmativeClicked -> ???
    NegativeClicked -> ???
```

For `Output`, i.e. Output from the Inner Component, we'll just forward it our Parent:

```
-- ABOVE CODE REDACTED
  where
  handleAction
    :: Action iInput iOutput
    -> H.HalogenM (State iInput) (Action iInput iOutput)
                  (Slots iQuery iOutput) (Output iOutput) m Unit
  handleAction = case _ of
    Input input -> H.modify_ _ { iInput = input }
    Output output -> H.raise $ InnerOutput output ❶
    AffirmativeClicked -> ???
    NegativeClicked -> ???
```

❶ We wrap `output` in `InnerOutput` since we have other `Outputs`.

For `AffirmativeClicked` and `NegativeClicked`, we will Output `Affirmative` and `Negative` respectively:

```
-- ABOVE CODE REDACTED
  where
  handleAction
    :: Action iInput iOutput
    -> H.HalogenM (State iInput) (Action iInput iOutput)
                  (Slots iQuery iOutput) (Output iOutput) m Unit
  handleAction = case _ of
    Input input -> H.modify_ _ { iInput = input }
    Output output -> H.raise $ InnerOutput output
    AffirmativeClicked -> H.raise Affirmative
    NegativeClicked -> H.raise Negative
```

For `handleQuery`, we'll just steal it from `Component.Page` since they both just forward the Query:

```
import Data.Maybe (Maybe) ❶

-- ABOVE CODE REDACTED
  handleQuery
    :: ∀ a
    .  iQuery a
    -> H.HalogenM (State iInput) (Action iInput iOutput)
                  (Slots iQuery iOutput) (Output iOutput) m (Maybe a) ❷
  handleQuery = H.query _inner unit
```

❶ Added this import.

❷ Once again, the Output Type differs from the Page Components in the exact same way.

And now, for the `render` Function. There is nothing difficult here if you've done HTML and CSS before. So, I'm going to just give you the whole thing. As you type it in, take time to note what's being done:

```
import Prelude hiding (top) ❶

import AppTheme (paperColor, themeColor, themeFont)
import CSS.Background (backgroundColor)
import CSS.Color (rgba, white)
import CSS.Common (center)
import CSS.Display (display, position, fixed, zIndex, flex)
```

```
import CSS.Flexbox (alignItems, justifyContent
                  , flexDirection, row, column, flexEnd, spaceAround)
import CSS.Font (FontWeight(..), fontWeight, color, fontSize)
import CSS.Geometry (top, left, padding, width, height ❶
                   , marginLeft, minWidth, maxWidth, minHeight)
import CSS.Overflow (overflow, overflowAuto)
import CSS.Property (value)
import CSS.Size (Size(..), pct, rem)
import Data.Maybe (Maybe (..))

  render
    :: State iInput
    -> H.ComponentHTML (Action iInput iOutput) (Slots iQuery iOutput) m
  render { iInput } = ❷
    HH.div [
      HC.style do
        display flex
        alignItems center
        justifyContent center
        position fixed ❸
        top $ Size $ value 0.0
        left $ Size $ value 0.0
        width (pct 100.0) ❹
        height (pct 100.0)
        overflow overflowAuto
        backgroundColor (rgba 0 0 0 0.4) ❺
        zIndex 1 ❻
    ]
    [ HH.div [
        HC.style do
          display flex
          flexDirection column
          justifyContent spaceAround
          padding (rem 1.0) (rem 1.0) (rem 1.0) (rem 1.0)
          backgroundColor paperColor
          minWidth (rem 40.0)
          maxWidth (pct 80.0)
          minHeight (rem 10.0)
      ]
      [ HH.div [
          HC.style do
            display flex
            flexDirection column
            padding (rem 1.0) (rem 1.0) (rem 1.0) (rem 1.0)
```

```
        ]
        [ HH.slot _inner unit innerComponent iInput Output ] ❼
      , HH.div [
          HC.style do
            display flex
            flexDirection row
            alignItems center
            justifyContent flexEnd
            width (pct 100.0)
            backgroundColor paperColor
        ]
        [ HH.button
          [
            buttonStyle
          , HE.onClick $ const AffirmativeClicked
          ]
          [ HH.text "OK" ] ❽
        , HH.button [
            buttonStyle
          , HE.onClick $ const NegativeClicked
          ]
          [ HH.text "CANCEL" ] ❽
        ]
    ]
  where
    buttonStyle = HC.style do ❾
      themeFont
      fontWeight $ FontWeight $ value "500"
      color white
      padding (rem 0.5) (rem 0.5) (rem 0.5) (rem 0.5)
      backgroundColor themeColor
      width (rem 8.0)
      marginLeft (rem 2.0)
      fontSize (rem 0.9)
```

❶ Had to hide `top` from Prelude since we're using it from `CSS.Geometry`.

❷ Destructure `State` to get the Inner Component's `Input` Value, i.e. `iInput`.

❸ We position ourselves as `fixed` so we won't move when the user scrolls the page.

❹ `width` and `height` are `100%` to cover the whole page.

❺ Background color as black with `40%` opactity. This "dims" the background.

❻ `zIndex` of `1` is just in case anything else is in our Stacking Order.

❼ Here's where the Inner Component is passed it's Input.

❽ These are hardcoded for now. We're going to make them configurable.

❾ Helper for styling both buttons.

We are displayed on top of all elements on the page due to many factors:

- `position fixed` makes us a Positioned Element, which display on top of non-Positioned Elements (`static`, `relative` and `absolute` don't behave this way)
- Specifying an Opacity treats us as `z-index 0`, which will displays on top of non-Position Elements, i.e. Elements with NO `z-index` specified
- `z-index` of `1` will make us a Positioned Element
- when the page that includes us in their `render` Function puts us as the LAST Element since all things being equal later Elements are displayed on top of previous

We now have a Modal Component, but without an Inner Component, we cannot use it.

30.13. Message Modal Component

To be able to remove the standard Browser `alert` Dialog Box, we're going to need to make an Inner Component that we can use to just display a message.

This will be a very simple Component with no `Outputs`, `Actions` or `Queries` and definitely no Children. It will, however, have an `Input`, viz. the message to display.

So create a File called `Message.purs` under the `src/Component` Folder and add the Following:

```
module Component.Message where

import Prelude

import Effect.Aff.Class (class MonadAff) ❶
import Halogen as H
import Halogen.HTML as HH
import Halogen.HTML.CSS as HC
```

❶ The usual suspects.

Now define the trivial Types for `Output`, `Action`, `Query` and `Slots`. Look to other Components as references if you forget how to do this.

Once you're done, keep reading.

Here are those Types:

```
import Data.Const (Const)

type Output = Void
type Action = Void

type Query :: ∀ k. k -> Type
type Query = Const Void

type Slots :: ∀ k. Row k
type Slots = ()
```

Now define the `Input` Type keeping in mind that our Input is the message to display. Then return here.

Here's that Type:

```
type Input = String
```

And finally, define the `State` before reading any further.

We need to save the message `String` so that `render` can have access to it:

```
type State = { message :: String }
```

Next, write the `component` Function. Use all of our other Components as reference. Then return here to see how you did.

Here's the code:

```
component
  :: ∀ m
  .  MonadAff m
  => H.Component Query Input Output m ❶
component = H.mkComponent
  { initialState: \message -> { message } ❷
  , render ❸
  , eval: H.mkEval H.defaultEval ❹
  }
  where
  ???
```

❶ This is standard Type Signature for `component`.

❷ We save our inital `Input` in our `State`

❸ We still need to write this.

❹ We use all defaults, i.e. no `handleAction`, `handleQuery`, `initialize`, `finalize` or `recieve`.

Note that we could've coded for `receive` to accept changes to the `message`. We're not going to do this, however, since we're just going to use this for Static Messages.

But, one could imagine a Modal Dialog that has a Progress Bar in it with a `Cancel` button. In that case, we'd want to process our `Input`.

And finally, we can write `render`. We're going to want to use `themeFont` and `paperColor` in the Styling of our `div` that holds our `message`.

Finish `render` before reading any further.

Here's that code:

```
import AppTheme (themeFont, paperColor) ❶
import CSS.Background (backgroundColor)

-- ABOVE CODE REDACTED
  where
  render :: State -> H.ComponentHTML Action Slots m
  render { message } =
    HH.div [
      HC.style do
        themeFont
        backgroundColor paperColor
    ] [ HH.text message]
```

❶ Added these imports.

Now that we have that completed, let's replace `alertError` in `Component.Logon` first and then `Component.Users`.

We're going to code the Logon Page changes together and then you'll use what you've learned in that process to make similar changes to the Users Page.

We'll start by modifying the `State` in `Component.Logon` to have an optional `errorMessage`:

```
type State =
  { userName :: String
  , password :: String
  , errorMessage :: Maybe String ❶
  }
```

❶ When `Nothing` then the `Modal` is hidden.

Next, we'll have to redefine the `Slots` Row Type since we now have `Modal` as a Child Component:

```
import Component.Message as Message ❶
import Component.Modal as Modal
import Type.Proxy (Proxy(..))

type Slots =
  ( modal :: H.Slot Message.Query (Modal.Output Message.Output) Unit ) ❷ ❸

_modal = Proxy :: Proxy "modal" ❹
```

❶ Added these imports.

❷ `Modal` doesn't define its own `Query` but instead defers to its Inner Component's `Query`. This is why we're using `Message.Query` here.

❸ Remember that `Modal.Output` takes a single Parameter of the Inner Component's `Output` Type. It's called `iOutput` in `Modal's` code.

❹ We define the label for the Child.

And now we should change `initialState` in `component` to handle the changes that we've made to `State`:

```
component
  :: ∀ m route
  .  MonadAff m
  => MonadAsk Env m
  => Navigate m route
  => LogonRoute m route
  => Log m
  => H.Component Query Input Output m
component = H.mkComponent
  { initialState: \_ ->
    { userName: ""
    , password: ""
    , errorMessage: Nothing ❶
    }
  , render
  , eval: H.mkEval H.defaultEval
      { handleAction = handleAction
      }
  }
  where
-- BELOW CODE REDACTED
```

❶ Initialize to NO error. The Modal will not be displayed by our `render` Function if this is `Nothing`.

Now, we'll simply set the `errorMessage` in the `State` when an error occurs. This will trigger a re-render and our `render` Function will see that `errorMessage` is NOT `Nothing` and display the Modal.

Here are the changes to `handleAction`:

```
-- ABOVE CODE REDACTED
  where
  handleAction :: Action -> H.HalogenM State Action Slots Output m Unit
  handleAction = case _ of
    Input f -> H.modify_ f
    Logon -> do
      { userName, password } <- H.get
      logonResponse <- apiCall (LogonRequest { userName, password })
      case logonResponse of
        Left err -> H.modify_ _ { errorMessage = Just err } ❶
        Right (LogonResponse LogonResultsFailure) ->
          H.modify_ _ { errorMessage = Just "Invalid Logon Credentials" } ❶
        Right (LogonResponse (LogonResultsSuccess
          { authToken, admin, mustChangePassword })) -> do
            log =<< logEntry Info "User logged on"
            { userRef } <- ask
            H.liftEffect $ Ref.write (Just { admin, authToken }) userRef
            navigate <=< logonRoute
              $ if mustChangePassword then PasswordTemporary
                else PasswordPermanent
-- BELOW CODE REDACTED
```

❶ Set `errorMessage` where we used to call `alertError` in these 2 places

We can now delete the following lines:

```
import Web.HTML.Window (alert)
import Web.HTML (window)

    where
    alertError :: String -> H.HalogenM State Action Slots Output m Unit
    alertError msg = H.liftEffect $ window >>= alert msg
```

Next, we'll add an `Action` to encapsulate `Output` from the `Modal`:

```
data Action
  = Logon
  | Input (State -> State)
  | Modal (Modal.Output Message.Output) ❶
```

❶ We pass the `Modal's` Output to the `Action`.

We'll need to handle the `Modal` Action in `handleAction`:

```
  handleAction :: Action -> H.HalogenM State Action Slots Output m Unit
  handleAction = case _ of
-- CODE REDACTED
    Modal output -> case output of
      Modal.Affirmative -> H.modify_ _ { errorMessage = Nothing } ❶
      Modal.Negative -> H.modify_ _ { errorMessage = Nothing } ❶
      Modal.InnerOutput _ -> pure unit ❷
-- BELOW CODE REDACTED
```

❶ Hide the Modal Dialog whether the user clicked `Affirmative` or `Negative` button.

❷ Our Message Modal Component doesn't output anything.

And finally we're going to change `render`, first to destructure `errorMessage`:

```
  render :: State -> H.ComponentHTML Action Slots m
  render { userName, password, errorMessage } =
-- BELOW CODE REDACTED
```

And then to add the Modal Component as the LAST Element:

```
import Data.Maybe (Maybe(..), maybe) ❶

-- ABOVE CODE REDACTED
    , (errorMessage # maybe (HH.text "") \message -> ❷
      HH.slot _modal unit
        (Modal.component Message.component) message Modal) <3> ❹ ❺
    ]
    where logonDisabled = trim userName == "" || trim password == ""
```

❶ Added `maybe` to the import list.

❷ When we do NOT have an error message we use an empty Text Node. Otherwise, we create the Modal Child Component.

❸ We pass the Message Component as the Inner Component for `Modal`.

❹ Notice how we pass the `message` as `Input` to `Modal` even though it doesn't define it's own `Input` Type. That's because its `Input` Type is the same Type as the Inner Component's.

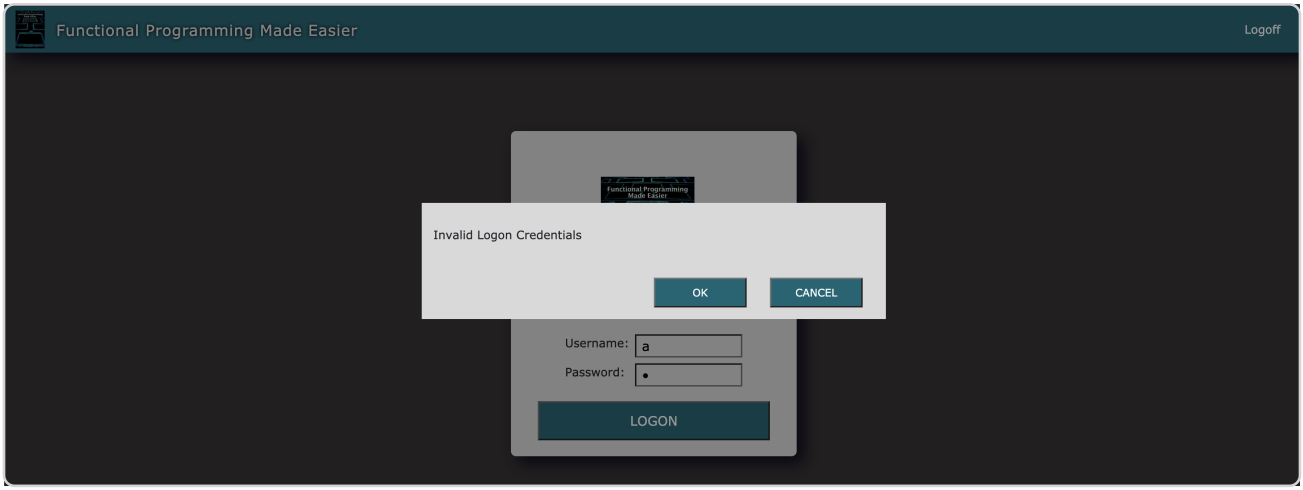
❺ Handle `Modal` output here with the final Parameter to `HH.stot`.

At this point, you can re-build the Application:

```
make build
```

Next, try to logon with invalid credentials.

You should see:



Now it's your turn. Use what we've done in `Logon` to use the Message Modal Component as a replacement for `alertError` on the Users Page, i.e. `Component.Users`.

You can pretty much follow the same steps as were outlined in the `Logon` changes if you've forgotten what we did and why.

Please make sure to try this yourself before reading on.

Here are the changes to the Types in `Component.Users`:

```
import Component.Message as Message ❶
import Component.Modal as Modal
import Type.Proxy (Proxy(..))

data Action
  = Initialize
  | UserSelected User
  | SelectedUserName (Maybe String)
  | Modal (Modal.Output Message.Output) ❷

type Slots =
  ( modal :: H.Slot Message.Query (Modal.Output Message.Output) Unit ) ❸

_modal = Proxy :: Proxy "modal" ❸

type State =
  { authorized :: Boolean
  , selectedUser :: Maybe User
  , users :: Map String User
  , initUserName :: Maybe String
  , errorMessage :: Maybe String ❹
  }
```

❶ Added these imports.

❷ Added this Data Constructor.

❸ Stole these from `Logon`. Also removed the Type Signature.

❹ Added optional `errorMessage` to `State`.

Now for the change to `initialState`:

```
component
  :: ∀ m
  .  MonadAff m
  => MonadAsk Env m
  => Navigate m Route
  => H.Component Query Input Output m
component = H.mkComponent
  { initialState: \initUserName ->
      { authorized: false
      , selectedUser: Nothing
      , users: Map.empty
      , initUserName
      , errorMessage: Nothing ❶
      }
  , render
  , eval: H.mkEval H.defaultEval
      { handleAction = handleAction
      , receive = Just <<< SelectedUserName
      , initialize = Just Initialize
      }
  }
-- BELOW CODE REDACTED
```

❶ Initialize `errorMessage` as `Nothing`.

Then, we'll add the code to handle the `Modal` Action in `handleAction`:

```
  handleAction :: Action -> H.HalogenM State Action Slots Output m Unit
  handleAction = case _ of
-- CODE REDACTED
    Modal output -> case output of ❶
      Modal.Affirmative -> H.modify_ _ { errorMessage = Nothing }
      Modal.Negative -> H.modify_ _ { errorMessage = Nothing }
      Modal.InnerOutput _ -> pure unit
    where
-- BELOW CODE REDACTED
```

❶ Added this case. Stole it from `Logon`.

And now we can change the code that calls `alertError`:

```
  handleAction :: Action -> H.HalogenM State Action Slots Output m Unit
  handleAction = case _ of
    Initialize -> do
      { userRef } <- ask
      loggedOnUser' <- H.liftEffect $ Ref.read userRef
      loggedOnUser' # maybe (pure unit) \{ authToken, admin } ->
        when admin do
          queryResponse <-
            apiCall (QueryUsersRequest { authToken })
          case queryResponse of
            Left err -> H.modify_ _ { errorMessage = Just err } ❶
            Right (QueryUsersResponse
                (QueryUsersResultsFailure { reason })) ->
              H.modify_ _ { errorMessage = Just $ "Query Users: " ❶
                <> case reason of
                  NotAuthorized -> "Not Authorized"
                  NotAuthenticated -> "Not Authenticated" }
            Right (QueryUsersResponse
                (QueryUsersResultsSuccess { users })) -> do
              let mkMap f = foldl (\m r -> Map.insert (f r) r m) Map.empty
                  usersMap = mkMap (_.userName <<< unwrap) users
              { initUserName } <- H.get
              H.modify_ _
                { authorized = true
                , users = usersMap
                , selectedUser = initUserName
                    >>= \userName -> Map.lookup userName usersMap
                }
  -- BELOW CODE REDACTED
```

❶ Set `errorMessage` in `State` in 2 places.

And now we can delete the following lines of code:

```
import Web.HTML.Window (alert)
import Web.HTML (window)

    where
    alertError :: String -> H.HalogenM State Action Slots Output m Unit
    alertError msg = H.liftEffect $ window >>= alert msg
```

And finally, we'll update render to first destructure for `errorMessage`:

```
  render :: State -> H.ComponentHTML Action Slots m
  render { authorized, selectedUser, users, errorMessage } =
-- BELOW CODE REDACTED
```

And second, to add the Modal Component as the LAST Element:

```
-- ABOVE CODE REDACTED
    , (errorMessage # maybe (HH.text "") \message ->
      HH.slot _modal unit
        (Modal.component Message.component) message Modal) ❶
    ]
```

❶ Stolen from `Logon`.

And now the code should compile.

We cannot easily test these error messages so we're just going to wink and nod and say it works for now.

30.14. Customizing the Modal Dialog Component

We'd like some control over the Modal Component. The following are areas where we want some control:

- Button labels
- Which buttons are displayed
- Some way to disable/enable the Affirmative button

There's lots more we could configure, e.g. Styling, but for this learning Application, these are actual requirements, whereas anything else are nice-to-haves.

The first 2 are going to be accomplished by passing in a Configuration Record. We'll want to create a Default Configuration so we can just change the fields we care about using PureScript's Record update syntax.

The third feature will be more complex since our Inner Component must communicate to the Modal, i.e. it must Output Data that the Modal Component will process. Currently, the Inner Component's `Output` is just passed on to the Parent with no interpretation.

This won't be hard, but will be a bit more complex and so we'll do that one together.

The first 2 features are pretty easy. So, start by defining a Configuration Record called `Config` in `Component.Modal` that reflects the aforementioned requirements and then return here.

Here's my initial definition:

```
type Config =
  { affirmativeLabel :: String
  , negativeLabel :: String
  , displayAffirmative :: Boolean
  , displayNegative :: Boolean
  }
```

But looking carefully at what I've designed, I see it's possible to hide both buttons. That would never be a good thing unless there was some way for the Inner Component to communicate to `Modal` to close.

Having these 2 `Booleans` is why we have this problem here. We want to make illegal states impossible to represent, so we should do that here.

We could, for example, create the following Sum Type:

```
data ButtonDisplay
  = DisplayBothButtons
  | DisplayAffirmative
  | DisplayNegative
```

But that doesn't support the case where we want no buttons, which we may want. But if we support no buttons then we're going to need to have a more sophisticated communication process between the Inner Component and `Modal`.

It turns out that our third requirement to allow the Inner Component to enable and disable buttons also requires that we have a more sophisticated communication process between the Inner Component and `Modal`.

Next, we're going to update `ButtonDisplay` so we can hide both buttons, which means we COULD go back to having 2 Booleans, and we wouldn't have the Boolean Blindness we have with Parameters because these Values will be part of a Record, which has named fields.

But, Sum Types are almost always better than Booleans, so we'll stay with that and simply add another option to `ButtonDisplay`:

```
data ButtonDisplay
  = DisplayBothButtons
  | DisplayAffirmative
  | DisplayNegative
  | DisplayNoButtons
```

Add this Type to `Component.Modal`.

Now, we can make our `Config` use `ButtonDisplay`:

```
type Config =
  { affirmativeLabel :: String
  , negativeLabel :: String
  , displayButtons :: ButtonDisplay
  }
```

Make sure your definition matches this one.

Now, we're going to need to create a Type that specifies the Type of `Output` that the Inner Component can make.

Currently, all Output from the Inner Component will pass-through to the Parent, but now some Output will be processed by `Modal`.

We'll start our definition with the pass-through case:

```
data InnerOutput iOutput
  = PassThrough iOutput
```

And now, we'd like a way for the Inner Component to enable and disable buttons as per requirement 3:

```
data InnerOutput iOutput
  = PassThrough iOutput
  | EnableAffirmative
  | DisableAffirmative
  | EnableNegative
  | DisableNegative
```

And if both buttons are hidden, we'll need some way to close the Modal. Since `Modal` sends either `Affirmative` or `Negative` to the Parent, we'll want to maintain that behavior. So, we'll allow the Inner Component to dictate that as part of the close:

```
data InnerOutput iOutput
  = PassThrough iOutput
  | EnableAffirmative
  | DisableAffirmative
  | EnableNegative
  | DisableNegative
  | CloseAffirmative
  | CloseNegative
```

Add this to the `Modal` code.

Next, create a `defaultConfig` for `Config` that maintains our current `Modal` behavior and then return here.

Here's that:

```
defaultConfig :: Config
defaultConfig =
  { affirmativeLabel: "OK"
  , negativeLabel: "CANCEL"
  , displayButtons: DisplayBothButtons
  }
```

Now change `component` to use `Config`. And change `render` to abide by the configuration Parameters.

Make that changes and then return here and continue reading.

Here's the change to `component`:

```
component
  :: ∀ iQuery iInput iOutput m
   . MonadAff m
  => Config ❶
  -> H.Component iQuery iInput iOutput m ❷
  -> H.Component iQuery iInput (Output iOutput) m
component config innerComponent = H.mkComponent ❸
-- BELOW CODE REDACTED
```

❶ Added `Config` to Type Signature.

❷ Changed `=>` to `->`.

❸ Added `config` Parameter.

And now, we'll make the following changes to render. First, destructuring `config`:

```
  render
    :: State iInput
    -> H.ComponentHTML (Action iInput iOutput) (Slots iQuery iOutput) m
  render { iInput } =
    let { affirmativeLabel
        , negativeLabel
        , displayButtons
        } = config in ❶
    HH.div [
-- BELOW CODE REDACTED
```

❶ Destructure `config` to get all of the fields into Variables. We could've destructured in `component` instead of defining the `config` Parameter.

And then we'll decide which buttons to display:

```
  render
    :: State iInput
    -> H.ComponentHTML (Action iInput iOutput) (Slots iQuery iOutput) m
  render { iInput } =
    let { affirmativeLabel
        , negativeLabel
        , displayButtons
        } = config
        displayAffirmative = case displayButtons of ❶
          DisplayAffirmative -> true
          DisplayBothButtons -> true
          _ -> false
        displayNegative = case displayButtons of ❷
          DisplayNegative -> true
          DisplayBothButtons -> true
          _ -> false in
-- BELOW CODE REDACTED
```

❶ Decide whethere to display the Affirmative button or not.

❷ Decide whethere to display the Negative button or not.

Now, change the rest of the rendering code to abide by the configuration before reading further.

Here's that change:

```
-- ABOVE CODE REDACTED
        [ if not displayAffirmative then HH.text "" else ❶
          HH.button [
              buttonStyle
            , HE.onClick $ const AffirmativeClicked
            ] [ HH.text affirmativeLabel ] ❷
        , if not displayNegative then HH.text "" else ❶
          HH.button [
              buttonStyle
            , HE.onClick $ const NegativeClicked
            ] [ HH.text negativeLabel ] ❷
        ]
      ]
    ]
    where
      buttonStyle = HC.style do
-- BELOW CODE REDACTED
```

❶ Insert empty Text Node when NOT displayed.

❷ Use the configuration labels.

Now, rebuild the Application:

```
make build
```

You should get 2 compiler errors. If you're in VSCode and are using the Terminal Pane, you can `Option-click` the File and Line number link in the Terminal Pane, or `Alt-click` on Windows, and the editor will jump to the line with the error.

Fix the errors and once you're program is compiling, return here.

Here are those fixes. First, one in `Component.Logon`:

```
-- ABOVE CODE REDACTED
    , (errorMessage # maybe (HH.text "") \message ->
      HH.slot _modal unit
        (Modal.component Modal.defaultConfig Message.component) ❶
        message Modal)
    ]
    where logonDisabled = trim userName == "" || trim password == ""
```

❶ Added `defaultConfig`.

And the same change is made in `Component.Users`:

```
-- ABOVE CODE REDACTED
    , (errorMessage # maybe (HH.text "") \message ->
      HH.slot _modal unit
        (Modal.component Modal.defaultConfig Message.component) ❶
        message Modal)
    ]
```

❶ Added `defaultConfig`.

Now when you rebuild, it should build but you will get the following warning:

```
[warn] None of your project files import modules from some projects that are
in the direct dependencies of your project.
These dependencies are unused. To fix this warning, remove the following
packages from the list of dependencies in your config:
- web-html
```

Remove this from our `spago.dhall` file before we forget.

Now refresh your Browser, then try to logon with invalid credentials to see if our Modal Component still works.

When I tried, the `Modal` still works. Thanks good. That means that we didn't break it.

Next, let's do a little housekeeping. First, create a Folder under `src/Component` called `Modal`. We're going to put all of our Inner Components for `Modal` here.

Then move `Message.purs` from `src/Component` to `src/Component/Modal`.

Change the module name in `Message.purs` to `Component.Modal.Message`.

Then change the imports of `Message.purs` in both `Component.Logon` and `Component.Users`.

Then just to make sure we got everything:

```
make build
```

Next, change both `Logon` and `Users` to only display the `OK` button. Instead of duplicating the code in both places, make a configuration called `errorConfig` and put it in a File called `Common.purs` under the `src/Component/Modal`.

Then return here when you're ready to move on.

First, here's the code in `Component.Modal.Common`:

```
module Component.Modal.Common where

import Component.Modal (Config, ButtonDisplay(..), defaultConfig)

errorConfig :: Config
errorConfig = defaultConfig { displayButtons = DisplayAffirmative }
```

Here's the change in `Logon`:

```
import Component.Modal.Common as ModalCommon ❶

-- ABOVE CODE REDACTED
    , (errorMessage # maybe (HH.text "") \message ->
      HH.slot _modal unit
        (Modal.component ModalCommon.errorConfig Message.component) ❷
        message Modal)
    ]
    where logonDisabled = trim userName == "" || trim password == ""
```

❶ Added this import.

❷ Use `errorConfig`.

And the change in `Users`:

```
import Component.Modal.Common as ModalCommon ❶

-- ABOVE CODE REDACTED
    , (errorMessage # maybe (HH.text "") \message ->
      HH.slot _modal unit
        (Modal.component ModalCommon.errorConfig Message.component) ❷
        message Modal)
    ]
```

❶ Added this import.

❷ Use `errorConfig`.

Now, test your code by rebuilding, refreshing the browser and trying an invalid logon.

When I do this, only the `OK` button is displayed.

All that's left to do is finish implementing `InnerOutput`. Currently, `Modal` expects the Inner Component's Output to be of Type `iOutput`, i.e. any Type. We'll need to change `Component.Modal` to expect `InnerOutput` instead.

First, we have to change the Child Slot Type:

```
type Slots iQuery iOutput =
  ( inner :: H.Slot iQuery (InnerOutput iOutput) Unit ) ❶
```

❶ Changed `iOutput` to `InnerOutput iOutput`.

And then we'll change `Action`:

```
data Action iInput iOutput
  = Input iInput
  | Output (InnerOutput iOutput) ❶
  | AffirmativeClicked
  | NegativeClicked
```

❶ The Type we get as an `Output` from the Inner Component is now of Type `InnerOutput iOutput`. It used to be just `iOutput`.

Next, we have to change the Type of the Component we pass for the Inner Component to `component` to have `InnerOutput` as its `Output` Type:

```
component
  :: ∀ iQuery iInput iOutput m
   . MonadAff m
  => Config
  -> H.Component iQuery iInput (InnerOutput iOutput) m ❶
  -> H.Component iQuery iInput (Output iOutput) m
```

❶ Changed Output Type.

Now, we have to fix `handleAction` to deal with all the cases of `InnerOutput`:

```
handleAction
    :: Action iInput iOutput
    -> H.HalogenM (State iInput) (Action iInput iOutput)
                  (Slots iQuery iOutput) (Output iOutput) m Unit
  handleAction = case _ of
    Input input -> H.modify_ _ { iInput = input }
    Output innerOutput -> case innerOutput of ❶
      PassThrough output -> H.raise $ InnerOutput output ❷
      EnableAffirmative -> pure unit ❸
      DisableAffirmative -> pure unit
      EnableNegative -> pure unit
      DisableNegative -> pure unit
      CloseAffirmative -> pure unit
      CloseNegative -> pure unit
    AffirmativeClicked -> H.raise Affirmative
    NegativeClicked -> H.raise Negative
```

❶ Handle the cases of `innerOutput`.

❷ This was our default behavior.

❸ All new behaviors have been stubbed out for now.

Now, we're ready to handle each case. Let's start with the enabling and disabling of buttons.

We're going to need to keep the state of the buttons, which means `State` will need to change.

Make the changes to `State` to add `affirmativeDisabled` and `negativeDisabled` to `State` and then change `initialize` to set them to `false`. And finally, change `render` to abide by these Values.

Return here when you're ready to move forward.

First, here's the change to `State`:

```
type State iInput =
  { iInput :: iInput
  , affirmativeDisabled :: Boolean ❶
  , negativeDisabled :: Boolean ❶
  }
```

❶ Added these 2 lines.

Next, is the change to `initialize`:

```
-- ABOVE CODE REDACTED
component config innerComponent = H.mkComponent
  { initialState: \iInput ->
    { iInput
    , affirmativeDisabled: false ❶
    , negativeDisabled: false ❶
    }
  , render
  , eval: H.mkEval $ H.defaultEval
      { handleAction = handleAction
      , handleQuery = handleQuery
      , receive = Just <<< Input
      }
  }
-- BELOW CODE REDACTED
```

❶ Initialize both to `false`.

Now, that I step back and consider things, it seems like our `Modal.Config` should allow us to initialize the state of the buttons. The changes we just made always set them to NOT disabled, i.e. `false`.

When we're done, we'll support that.

And finally, the code change to `render`:

```
import Halogen.HTML.Properties as HP ❶

  render
    :: State iInput
    -> H.ComponentHTML (Action iInput iOutput) (Slots iQuery iOutput) m
  render { iInput, affirmativeDisabled, negativeDisabled } = ❷
-- BELOW CODE REDACTED

-- ABOVE CODE REDACTED
        [ if not displayAffirmative then HH.text "" else
          HH.button [
              buttonStyle
            , HP.disabled affirmativeDisabled ❸
            , HE.onClick $ const AffirmativeClicked
            ] [ HH.text affirmativeLabel]
        , if not displayNegative then HH.text "" else
          HH.button [
              buttonStyle
            , HP.disabled negativeDisabled ❸
            , HE.onClick $ const NegativeClicked
            ] [ HH.text negativeLabel]
        ]
      ]
    ]
    where
-- BELOW CODE REDACTED
```

❶ Added this import.

❷ Destructure to get `affirmativeDisabled` and `negativeDisabled`.

❸ Added Property `disabled` in each button.

Now, we should allow `Modal.Config` to state the initial state of the buttons.

Make the change to `Config` in `Component.Modal` to add the fields `affirmativeDisabled` and `negativeDisabled`. Also, change `defaultConfig` to have the buttons enabled, i.e. the current behavior.

Return here once you're ready.

Here are the changes to `Config` and `defaultConfig`:

```
type Config =
  { affirmativeLabel :: String
  , negativeLabel :: String
  , displayButtons :: ButtonDisplay
  , affirmativeDisabled :: Boolean ❶
  , negativeDisabled :: Boolean ❶
  }

defaultConfig :: Config
defaultConfig =
  { affirmativeLabel: "OK"
  , negativeLabel: "CANCEL"
  , displayButtons: DisplayBothButtons
  , affirmativeDisabled: false ❷
  , negativeDisabled: false ❷
  }
```

❶ Allow configuration whether the buttons are initially disabled.

❷ Default to `false`.

Next, change `initialState` to use these new `Config` fields before reading any further.

And here's the change to `initialState`:

```
-- ABOVE CODE REDACTED
component config innerComponent = H.mkComponent
  { initialState: \iInput ->
    { iInput
    , affirmativeDisabled: config.affirmativeDisabled ❶
    , negativeDisabled: config.negativeDisabled ❶
    }
  , render
  , eval: H.mkEval $ H.defaultEval
      { handleAction = handleAction
      , handleQuery = handleQuery
      , receive = Just <<< Input
      }
  }
-- BELOW CODE REDACTED
```

❶ Changed from `false` to use the `Config` Values.

Now, we're ready to support the rest of the `InnerOutput` cases. Here's where we left off in `handleAction`:

```
-- ABOVE CODE REDACTED
    Output innerOutput -> case innerOutput of
      PassThrough output -> H.raise $ InnerOutput output
      EnableAffirmative -> pure unit
      DisableAffirmative -> pure unit
      EnableNegative -> pure unit
      DisableNegative -> pure unit
      CloseAffirmative -> pure unit
      CloseNegative -> pure unit
-- BELOW CODE REDACTED
```

Just implement the code for enabling and disabling the buttons and then return back here to continue.

Here's those changes:

```
-- ABOVE CODE REDACTED
    Output innerOutput -> case innerOutput of
      PassThrough output -> H.raise $ InnerOutput output
      EnableAffirmative -> H.modify_ _ { affirmativeDisabled = false } ❶
      DisableAffirmative -> H.modify_ _ { affirmativeDisabled = true } ❶
      EnableNegative -> H.modify_ _ { negativeDisabled = false } ❶
      DisableNegative -> H.modify_ _ { negativeDisabled = true } ❶
      CloseAffirmative -> pure unit
      CloseNegative -> pure unit
-- BELOW CODE REDACTED
```

❶ Modify `State` accordingly for all 4 cases.

And now implement the code for closing with both `Affirmative` and `Negative`. Then return here to continue.

Here's those changes:

```
  handleAction
    :: Action iInput iOutput
    -> H.HalogenM (State iInput) (Action iInput iOutput)
                  (Slots iQuery iOutput) (Output iOutput) m Unit
  handleAction = case _ of
    Input input -> H.modify_ _ { iInput = input }
    Output innerOutput -> case innerOutput of
      PassThrough output -> H.raise $ InnerOutput output
      EnableAffirmative -> H.modify_ _ { affirmativeDisabled = false }
      DisableAffirmative -> H.modify_ _ { affirmativeDisabled = true }
      EnableNegative -> H.modify_ _ { negativeDisabled = false }
      DisableNegative -> H.modify_ _ { negativeDisabled = true }
      CloseAffirmative -> handleAction AffirmativeClicked ❶
      CloseNegative -> handleAction NegativeClicked ❶
    AffirmativeClicked -> H.raise Affirmative
    NegativeClicked -> H.raise Negative
```

❶ Recurse with the appropriate `Action`.

Notice how I recurse instead of duplicating code. This is to help future-proof `handleAction`.

Imagine a future where more things are done in `AffirmativeClicked` or `NegativeClicked`. We wouldn't want to have to duplicate that code in multiple places.

And worse off, it's more likely that a developer would make changes to one and not the other.

By recursing, we're automatically getting all the behavior that we'd get if the user clicked on the buttons. This is what we mean semantically with `CloseAffirmative` and `CloseNegative`. The only difference being that the program "clicked" the specified button.

A canonical case for this is an Inner Component that supports Enter and Esc inputs for Affirmative and Negative. When the user hits the Esc key, the logic in the Inner Component will output `CloseNegative`. This should have the same behavior as if the user had clicked the Negative Button.

Now, the last thing to do is to change the Message Component, i.e. `Component.Modal.Message`, to support `InnerOutput`:

```
import Component.Modal (InnerOutput) ❶

component
  :: ∀ m
  .  MonadAff m
  => H.Component Query Input (InnerOutput Output) m ❷
```

❶ Added this import.

❷ We wrap the ACTUAL `Output` Type of the Message Component in `InnerOutput` to appease the expectations of `Modal` who will unwrap it BEFORE sending its Parent.

`InnerOutput` is a private Type between `Modal` and the Inner Component.

Now, everything should compile. Test that theory with:

```
make build
```

Hopefully, our Modal Component is configurable enough to support an Inner Component that is a Form since that's the next part we'll need to work on.

30.15. Create Users Modal Component

We're going to create another Inner Component for our Modal. This will be far more sophisticated than our Message Component.

Start by creating a File called `CreateUser.purs` in `src/Component/Modal` with the following:

```
module Component.Modal.CreateUser where

import Prelude

import Halogen as H
import Halogen.HTML as HH
import Halogen.HTML.CSS as HC
```

Noticed that our Inner Components are all in `src/Component/Modal`. This is because these Components cannot stand on their own and have very specific interface requirements to work with `Modal`. So, it's good practice to segregate them.

Now, before we code any more, we should think through the design of this Modal:

- Fields
  - User Name
  - Password
  - Admin (Checkbox)
  - First Name
  - Last Name
- Buttons

- Create
- Cancel

Like with `Logon`, we don't want the user to click the Create button until all of the fields have been entered.

Also, we need to handle the case where we fail to create a `User`. We'll use the Message Component with `Modal` for that. Yes, Inner Components can have `Modal` Dialog Boxes of their own with our design.

Another thing to keep in mind is that once we successfully create a new user, we'll want to send that new user to our Parent so that it can be added to the list of users.

Let's start by defining our Types keeping all of this in mind and using our existing code as reference.

Define `Input` and `Output` Types before reading any further.

Here are those Type definitions:

```
import Entity.User (User)

type Input = Unit ❶
type Output = User ❷
```

❶ No `Input`.

❷ We output the `User` so our Parent can add it to its list of users.

Next, define `Query` and `Actions`, then return here to continue.

Here's `Query`:

```
import Data.Const (Const)

type Query :: ∀ k. k -> Type
type Query = Const Void ❶
```

❶ No `Queries`.

Defining `Action` is going to take a bit more thought before we run off coding.

We know that we're going to handle enabling the button once all of the fields have data. We did this in `Logon`, so we'll at least need a similar `Action` like it had.

Let's look at `Action` from `Logon`:

```
data Action
  = Logon
  | Input (State -> State) ❶
  | Modal (Modal.Output Message.Output)
```

❶ This `Action` looks like it handles inputs.

Let's checkout `handleAction` from `Logon` to see how `Input` works:

```
  handleAction
    :: Action
    -> H.HalogenM State Action Slots (InnerOutput Output) m Unit
  handleAction = case _ of
    Input f -> H.modify_ f
-- BELOW CODE REDACTED
```

Well, that doesn't look like it's doing anything special. Where else should we look in `Logon` to see where the button enabling/disabling happens?

The `render` Function has to do that. So look there to see if you can find out how we're handling it and return here when you've found it.

Here's what I find at the very bottom of `render` in `Logon`:

```
    where logonDisabled = trim userName == "" || trim password == ""
```

Okay, so this will work for us in our `CreateUser` Component. But what about `Input`?

How and where is it used in `Logon`? Take a look and see what you find.

It's used in `render` for `userName`:

```
-- ABOVE CODE REDACTED
        [ HH.label [
            HC.style do
              width (rem 6.0)
          ] [ HH.text "Username: " ]
        , HH.input [
            HE.onValueInput $ Input <<< \s -> _ { userName = s } ❶
          , HC.style do
              backgroundColor paperColor
              width (vw 8.3)
              paddingLeft (rem 0.5)
              paddingRight (rem 0.5)
              fontSize (vw 1.0)
          ]
        ]
-- BELOW CODE REDACTED
```

❶ Here's is where it's used. Notice that we're using Function Composition. That's because we want to take the `String` that `onValueInput` gives us and update `username` with it and then pass it to the `Input` Data Constructor.

And here it's used again for `password`:

```
-- ABOVE CODE REDACTED
        [ HH.label [
            HC.style do
              width (rem 6.0)
          ] [ HH.text "Password: " ]
        , HH.input [
            HP.type_ InputPassword
          , HE.onValueInput $ Input <<< \s -> _ { password = s } ❶
          , HC.style do
              backgroundColor paperColor
              width (vw 8.3)
              paddingLeft (rem 0.5)
              paddingRight (rem 0.5)
              fontSize (vw 1.0)
          ]
        ]
-- BELOW CODE REDACTED
```

❶ Again, we're using Function Composition. That's because we want to take the `String` that `onValueInput` gives us and update `password` with it and then pass it to the `Input` Data Constructor.

This approach seems good. There's no need to create a special `Action` for each field in the Form.

Here's `Action` from `Logon` again:

```
data Action
  = Logon
  | Input (State -> State)
  | Modal (Modal.Output Message.Output)
```

We know we want to use `Input`. We also want to steal the Data Constructor `Logon`, which is triggered when the user clicks the Logon Button, but we'll call ours `CreateUser`.

The `Modal` is going to prove useful too since we are going to have a `Modal` when we get an error on the `CreateUsersRequest` from the `server`, so we'll use that too:

```
import Component.Modal as Modal ❶
import Component.Modal.Message as Message

data Action
  = CreateUser
  | Input (State -> State) -- COMPILER ERROR!! ❷
  | Modal (Modal.Output Message.Output)
```

❶ Added these imports.

❷ We'll ignore this for now since we haven't defined `State` yet.

Notice that our Inner Component has its own Modal. We just have to make sure that it layers over the top of us, since we're also a Modal.

This will most likely happen since our `Modal` will create Elements in the DOM after our own and should, therefore, display above us. One never knows with CSS, so we'll have to wait and see.

Add the above code to `Component.Modal.CreateUser`.

Now, we should define the `State` to fix the compiler error. Give this a shot using `Logon` as an example. Then return here to move on.

Here's the `State` definition:

```
import Data.Maybe (Maybe) ❶

type State =
  { userName :: String ❷
  , password :: String
  , admin :: Boolean
  , firstName :: String
  , lastName :: String
  , errorMessage :: Maybe String ❸
  }
```

❶ Added this import.

❷ `userName` through `lastName` are the Form fields.

❸ `errorMessage` is for when our `CreateUserRequest` fails.

One thing to note here is that by handling the API call and error message in this Component, we can present the User with the reason for the error and allow them to augment their input and try again WITHOUT having to reenter everything.

Our final part before digging into the Component implementation is to define the Child Slots. Keep in mind that our only Child is the `Modal`.

Steal what you can from `Logon` or `Users` and then return here when you're ready to move forward.

Here's a copy and paste from `Logon`:

```
import Type.Proxy (Proxy(..)) ❶

type Slots =
  ( modal :: H.Slot Message.Query (Modal.Output Message.Output) Unit )

_modal = Proxy :: Proxy "modal"
```

❶ Added this import.

At this point, all of our standard Halogen Types have been defined and we're now ready to code `component`. Write as much of `component` as you can before reading on.

We're going to cheat and use `Logon` as a reference to produce the following:

```
import Component.Modal (InnerOutput) ❶
import Data.Maybe (Maybe(..)) ❷
import Effect.Aff.Class (class MonadAff) ❶

component
  :: ∀ m
  .  MonadAff m
  => H.Component Query Input (InnerOutput Output) m ❸
component = H.mkComponent
  { initialState: \_ ->
    { userName: ""
    , password: ""
    , admin: false
    , firstName: ""
    , lastName: ""
    , errorMessage: Nothing
    }
  , render -- COMPILER ERROR!!
  , eval: H.mkEval H.defaultEval
      { handleAction = handleAction
      }
  }
```

❶ Add these imports.

❷ Added `(..)` to get the Data Constructors.

❸ Notice the `InnerOutput`. This is an Inner Component.

The compiler error is there to remind us to write `render`. Write a stub for `render` for now except take the time to fully destructure the `State` Parameter.

Return here when that's complete.

Here's the stubbed out `render`:

```
-- ABOVE CODE REDACTED
  where
  render :: State -> H.ComponentHTML Action Slots m
  render _ = HH.text ""
```

Now, the compiler's nagging us to define `handleAction`. So, add `handleAction` with all of the handling logic stubbed out for now, i.e. write the `case` expression and stub out the code under each case, and then return here.

Continue reading once that's complete.

Here's that implementation:

```
-- ABOVE CODE REDACTED
  where
  handleAction
    :: Action
    -> H.HalogenM State Action Slots (InnerOutput Output) m Unit
  handleAction = case _ of
    Input _ -> pure unit
    Modal _ -> pure unit
    CreateUser -> pure unit (1)
```

(1) I've placed `CreateUser` as the last Pattern in the `case` since I expect this to contain the most code and I'd like to not have to scroll to see the other cases.

Now, implement the `Input` case and then return here.

We can just steal the implementation from `Logon`:

```
-- ABOVE CODE REDACTED
  where
  handleAction
    :: Action
    -> H.HalogenM State Action Slots (InnerOutput Output) m Unit
  handleAction = case _ of
    Input f -> H.modify_ f
    Modal _ -> pure unit
    CreateUser -> pure unit
```

Now, implement the `Modal` case using `Logon` or `Users` for reference, then continue reading.

Here's the implementation of `Modal` lifted whole from `Logon` with no modification since it does exactly what we need:

```
-- ABOVE CODE REDACTED
  where
  handleAction
    :: Action
    -> H.HalogenM State Action Slots (InnerOutput Output) m Unit
  handleAction = case _ of
    Input f -> H.modify_ f
    Modal output -> case output of
      Modal.Affirmative -> H.modify_ _ { errorMessage = Nothing }
      Modal.Negative -> H.modify_ _ { errorMessage = Nothing }
      Modal.InnerOutput _ -> pure unit
    CreateUser -> pure unit
```

And just like before, we're going to want to steal what we can from `Logon` to help us write the `CreateUser` handler.

We're going to want to do nearly all the same steps, so refresh your memory on how `Logon` is handled in the Logon Component before attempting to code the `CreateUser` case.

Also, look to `Component.Users` to see how we got the `authToken` from the `LoggedOnUser` when you write the code to make the API call to the server.

Then give it a go and return here when you get stuck or when you finish.

I'm going to follow the same steps that `Logon` follows. First, get the fields from the `State`:

```
-- ABOVE CODE REDACTED
    CreateUser -> do ❶
      { userName, password, admin, firstName, lastName } <- H.get
      ???
      pure unit -- TODO remove ❷
```

❶ Started a `do` block.

❷ Put this here temporarily to make the compiler happy. I have to remember to remove this.

Next step is to make the API call using our handy dandy `apiCall` helper. To do this, we'll also have to build the `CreateUserRequest`:

```
import Data.Api.CreateUser (CreateUserRequest(..), CreateUserResponse(..)
                          , CreateUserResults(..)) ❶ ❷
import Utils (apiCall)

-- ABOVE CODE REDACTED
    CreateUser -> do
      { userName, password, admin, firstName, lastName } <- H.get
      createResponse <- apiCall (CreateUserRequest
        { authToken -- COMPILER ERROR!!
        , user:
          { userName
          , password
          , admin
          , firstName
          , lastName
          , temporaryPassword: true
          }
        })
      pure unit -- TODO remove
```

❶ Added these imports.

❷ There's more imported here than is currently needed in anticipation that we'll need it soon.

The compiler error is because we don't have the `authToken`. We have to get that from our `ReaderT` environment.

If you didn't get this far, catch up you code to here and add the code to get just the `authToken` from the logged on user before reading on.

Here's that change with a stubbed out part:

```
import Control.Monad.Reader.Class (class MonadAsk, ask) ❶
import Data.Maybe (Maybe(..), maybe) ❷
import Effect.Ref as Ref ❶
import Env (Env) ❶

component
  :: ∀ m
  .  MonadAff m
  => MonadAsk Env m ❸
  => H.Component Query Input (InnerOutput Output) m
-- BELOW CODE REDACTED

-- ABOVE CODE REDACTED
    CreateUser -> do
      { userRef } <- ask ❹
      loggedOnUser' <- H.liftEffect $ Ref.read userRef ❺
      loggedOnUser' # maybe (pure unit {- TODO -}) \{ authToken } -> do ❻ ❼
        { userName, password, admin, firstName, lastName } <- H.get ❽
        createResponse <- apiCall (CreateUserRequest ❾
          { authToken
          , user:
            { userName
            , password
            , admin
            , firstName
            , lastName
            , temporaryPassword: true
            }
          }) -- COMPILER ERROR!!
        ???
        pure unit -- TODO remove
```

❶ Added these imports.

❷ Added `maybe` to the import list.

❸ Added Constraint on `m` so we could call `ask`.

❹ Get `userRef` from the `ReaderT` environment, i.e. a Value of Type `Env`.

❺ Read the `userRef` reference to get the logged on user.

❻ `ref` can be `Nothing`, which we're not handling yet, hence, the `TODO` comment.

❼ Destructure `LoggedOnUser` in the Lambda to get the `authToken`.

❽ The lines from this point on got tabbed in to be in the Lambda.

❾ I'm sure glad we factored out the API logic into `apiCall`.

Don't worry about the compiler error. It's because the compiler cannot figure out which `decodeJSON` to call in `apiCall` until we use `createResponse`.

Now, we're ready to add the code for handling the response. Steal the structure for the `case` expression that we use in `Logon`, but stub out the implementation for the `Right` cases for now.

Then return back here.

Here's that change:

```
import Data.Either (Either(..)) ❶

-- ABOVE CODE REDACTED
    CreateUser -> do
      { userRef } <- ask
      loggedOnUser' <- H.liftEffect $ Ref.read userRef
      loggedOnUser' # maybe (pure unit {- TODO -}) \{ authToken } -> do
        { userName, password, admin, firstName, lastName } <- H.get
        createResponse <- apiCall (CreateUserRequest
          { authToken
          , user:
            { userName
            , password
            , admin
            , firstName
            , lastName
            , temporaryPassword: true
            }
          })
        case createResponse of ❷
          Left err -> H.modify_ _ { errorMessage = Just err } ❸
          Right CreateUserResultsSuccess -> pure unit -- TODO
          Right (CreateUserResultsFailure _) -> pure unit -- TODO
```

❶ Added this import.

❷ We don't need the `pure unit` to keep the compiler happy anymore since the `case` expression is the final result of the `do` block.

❸ This will set the `errorMessage` in `State` so `render` will display the Modal.

Now, we need to make some decisions regarding all of the `TODOs`.

The first `TODO` is when we have no logged on user. In that case, we should display a message and then route

to the logon page.

Let's create an optional `Action` in our `State` that will tell us what `Action` to perform when the user clicks the `Modal's` Affirmative Button.

Most of the time this will be `Nothing`, but in this case, we'll have an `Action` called `RouteToLogon` that we'll set here so we'll route to the Logon Page when our error message is done.

Create a new `Action` called `RouteToLogon` before reading on.

Here's that change:

```
data Action
  = CreateUser
  | Input (State -> State)
  | Modal (Modal.Output Message.Output)
  | RouteToLogon ❶
```

❶ Added this Data Constructor.

Not surprisingly, we broke `handleAction`. So let's fix that next. Add code to handle the routing to `Logon`. Use `Component.ChangePassword` as a reference to see how we routed to the Users Page once the button is clicked.

Keep in mind that you'll have to add the `Navigation` Capability to our `m` in `component` to be able to call `navigate`.

Make these changes before continuing any further.

Here are those changes:

```
import Capability.Navigate (class Navigate, navigate) ❶
import Data.Route (Route)
import Data.Route as Route

component
  :: ∀ m
  .  MonadAff m
  => MonadAsk Env m
  => Navigate m Route ❷
  => H.Component Query Input (InnerOutput Output) m
-- BELOW CODE REDACTED


-- ABOVE CODE REDACTED
  where
  handleAction
    :: Action
    -> H.HalogenM State Action Slots (InnerOutput Output) m Unit
  handleAction = case _ of
    Input f -> H.modify_ f
    Modal output -> case output of
      Modal.Affirmative -> H.modify_ _ { errorMessage = Nothing }
      Modal.Negative -> H.modify_ _ { errorMessage = Nothing }
      Modal.InnerOutput _ -> pure unit
    RouteToLogon -> navigate Route.Logon ❸
-- BELOW CODE REDACTED
```

❶ Added these imports.

❷ Constrain the `Monad` that our component runs in to have a `Navigate` Instance.

❸ Navigate to the `Logon` route.

Now, we can change the `State` to have the optional `Action` to perform AFTER the Error Modal is closed. Let's call it `postModalAction` for lack of a better name. Return here once that's done.

Here are those changes:

```
type State =
  { userName :: String
  , password :: String
  , admin :: Boolean
  , firstName :: String
  , lastName :: String
  , errorMessage :: Maybe String
  , postModalAction :: Maybe Action ❶
  }

-- ABOVE CODE REDACTED
component = H.mkComponent
  { initialState: \_ ->
    { userName: ""
    , password: ""
    , admin: false
    , firstName: ""
    , lastName: ""
    , errorMessage: Nothing
    , postModalAction: Nothing ❷
    }
  , render
  , eval: H.mkEval H.defaultEval
      { handleAction = handleAction
      }
  }
-- BELOW CODE REDACTED
```

❶ Added `postModalAction`. Don't forget that it's optional.

❷ Initialized our new field.

Now, write the code to display an error message in a Modal and setup to route to the Logon Page AFTER the Message Modal is close. Put the code in place of the `pure unit` where the `TODO` comment is in the following line of code:

```
      loggedOnUser' # maybe (pure unit {- TODO -}) \{ authToken } -> do
```

Return here once completed.

Here are those changes:

```
import Data.Maybe (Maybe(..)) ❶

-- ABOVE CODE REDACTED
    CreateUser -> do
      { userRef } <- ask
      loggedOnUser' <- H.liftEffect $ Ref.read userRef
      case loggedOnUser' of ❷
        Nothing -> H.modify_ _
          { errorMessage = Just "Session has timed out" ❸
          , postModalAction = Just RouteToLogon ❹
          }
        Just { authToken } -> do ❺
          { userName, password, admin, firstName, lastName } <- H.get
          createResponse <- apiCall (CreateUserRequest
            { authToken
            , user:
              { userName
              , password
              , admin
              , firstName
              , lastName
              , temporaryPassword: true
              }
            })
          case createResponse of
            Left err -> H.modify_ _ { errorMessage = Just err }
            Right CreateUserResultsSuccess -> pure unit -- TODO
            Right (CreateUserResultsFailure _) -> pure unit -- TODO
```

❶ Removed `maybe` from the import list.

❷ Sometimes `maybe` makes our code more readable and sometimes it does not. In those times, using a `case` expression usually improves readability.

❸ We let the user know that they're no longer logged on.

❹ We set up the action to perform AFTER the modal is closed. We still need to support this.

❺ Our Lambda code is now under the `Just` case.

And finally, handle the `postModalAction` when the Modal is closed. Then return here to continue reading.

Here's that change:

```
import Data.Maybe (Maybe(..), maybe) ❶

-- ABOVE CODE REDACTED
    Modal output -> case output of
      Modal.Affirmative -> do ❷
        { postModalAction } <- H.get ❸
        H.modify_ _ { errorMessage = Nothing, postModalAction = Nothing } ❹
        postModalAction # maybe (pure unit) handleAction ❺
      Modal.Negative -> H.modify_ _ { errorMessage = Nothing }
      Modal.InnerOutput _ -> pure unit
-- BELOW CODE REDACTED
```

❶ Added `maybe` BACK into the import list.

❷ Added a `do` block since we have multiple Monadic steps.

❸ Get `postModalAction` from `State`.

❹ Clear the error message AND the Post Modal `Action`.

❺ Call `handleAction` with the contents of `postModalAction` unless it's `Nothing`, then just `pure unit`.

The order here is deliberate. We want to update `State` first, then perform the `postModalAction` with the updated `State`. If we did the opposite order, then our changes to `State` COULD overwrite the changes that `postModalAction` makes.

This probably will never be an issue for us, but it's a good practice to consider order when dealing with `State`. In this case, we have no idea what changes will be made in the future where this will matter. Better safe than sorry.

The next `TODO` is `CreateUserResultsSuccess`, but let's skip that for now and move to `CreateUserResultsFailure`.

So, implement the code for:

```
Right (CreateUserResultsFailure _) -> pure unit -- TODO
```

Remember, all it has to do is display the `Modal`. Once you've completed that return here.

Here's that change:

```
import Data.Api.CreateUser (CreateUserRequest(..) , CreateUserResults(..)
                          , CreateUserFailureReason(..)) ❶

-- ABOVE CODE REDACTED
            Right (CreateUserResultsFailure { reason }) -> case reason of
              NotAuthorized ->
                errorMsg "You're not authorized to create a new user."
              NotAuthenticated -> errorMsg "Your session just timed out."
                *> H.modify_ _ { postModalAction = Just RouteToLogon }
              AlreadyExists ->
                errorMsg "A user with that User Name already exists."
              FileIOError err -> errorMsg
                $ "A fatal error on the Accounts File occurred: "
                <> err
    where
    errorMsg msg = H.modify_ _ { errorMessage = Just msg } ❷
-- BELOW CODE REDACTED
```

❶ Removed `CreateUserResponse` and added `CreateUserFailureReason`.

❷ Created a helper to reduce code complexity.

Now that we have `errorMsg`, we should also use elsewhere:

```
-- ABOVE CODE REDACTED
      case loggedOnUser' of
        Nothing -> H.modify_ _ { postModalAction = Just RouteToLogon }
           *> errorMsg "Session has timed out" ❶
-- BELOW CODE REDACTED
```

❶ Note that this has the unfortunate effect of causing our `render` Function to be called twice. Once for each change to the `State`.

The inefficiency of the above code won't be a problem for our case, but here's how to fix that problem:

```
-- ABOVE CODE REDACTED
      case loggedOnUser' of
        Nothing -> H.modify_ $ errorMsg' "Session has timed out"
          <<< _ { postModalAction = Just RouteToLogon } ❶
-- BELOW CODE REDACTED

-- ABOVE CODE REDACTED
    where
    errorMsg' msg = _ { errorMessage = Just msg } ❷

    errorMsg = H.modify_ <<< errorMsg' ❸
-- BELOW CODE REDACTED
```

❶ Compose the 2 `State` modification Functions before calling `H.modify_`. Now, we'll re-render our Component only once.

❷ This is a version of `errorMsg` that returns a Function that will modify the `State` Record to add the error message.

❸ This Function didn't change its behavior, but now leverages `errorMsg'` to avoid duplicate code.

Go ahead and update your version to this more efficient one just so you have it as a reference. Also, stop here and make sure you understand how this more efficient version works. Someday you may need to make your `State` updates as efficient as possible because of a costly page rendering.

You can also make the same change here if you like even though it's probably overkill:

```
-- ABOVE CODE REDACTED
            Right (CreateUserResultsFailure { reason }) -> case reason of
              NotAuthorized ->
                errorMsg "You're not authorized to create a new user."
              NotAuthenticated ->
                H.modify_ $ errorMsg' "Your session just timed out."
                  <<< _ { postModalAction = Just RouteToLogon } ❶
              AlreadyExists ->
                errorMsg "A user with that User Name already exists."
              FileIOError err -> errorMsg
                $ "A fatal error on the Accounts File occurred: "
                <> err
    where
-- BELOW CODE REDACTED
```

❶ Reduced 2 `State` updates to only 1 using Functional Composition and `errorMsg'`.

And the final `TODO` is the one we put off, which is the case where we have succesfully created a `User`. We

want to tell our Parent about the new `User`. Make that change before reading any further.

Here's that change:

```
import Component.Modal (InnerOutput(..)) ❶
import Entity.User (User(..)) ❶

-- ABOVE CODE REDACTED
            Right CreateUserResultsSuccess -> H.raise $ PassThrough ❷
              $ User ❸
                { userName
                , admin
                , firstName
                , lastName
                , temporaryPassword: true
                }
-- BELOW CODE REDACTED
```

❶ Added `(..)` to get Data Constructors.

❷ Send `Output` to Parent. Don't forget `PassThrough`.

❸ Remember our `Output` Type is `User`.

`Modal` will see a `PassThrough` and unwrap its payload and `H.raise` that to its Parent.

Now, we'll need a button for adding users on the Users Page. We do NOT have to check to make sure that the user has Admin rights since they wouldn't be able see any part of the Page if they don't.

Switch to `Component.Users`. Here are the changes to `render`:

```
import AppTheme (paperColor, selectedColor, themeColor, themeFont) ❶
import CSS.Common (center) ❷
import CSS.Font (FontWeight(..), color, fontWeight, fontSize) ❸
import CSS.Geometry ( minWidth, maxHeight, padding, paddingBottom
                    , paddingRight, paddingLeft, width, height) ❹
import CSS.Overflow (overflowY, overflowAuto) ❷
import CSS.Property (value) ❷
import CSS.Size (rem, pct, px) ❺

-- ABOVE CODE REDACTED
      [ HH.ul ❻
        [
          HC.style do
            maxHeight (pct 80.0)
```

```
            overflowY overflowAuto
        ]
          (Map.values users # fromFoldable
            <#> \(user@(User { userName })) ->
              let isSelected = Just user == selectedUser in
              HH.li [
                HP.class_ $ ClassName "list-group-item"
              , HC.style do
                  backgroundColor
                    if isSelected then selectedColor
                    else paperColor
                  color
                    if isSelected then white
                    else black
                  cursor pointer
              , HE.onClick $ const $ UserSelected user
              ]
              [ HH.text userName ])
      , HH.div  ❼
        [
          HC.style do
            display flex
            flexDirection row
            justifyContent center
            paddingLeft (px 40.0)
        ]
        [ HH.button
          [
            HC.style do
              backgroundColor themeColor
              themeFont
              fontWeight $ FontWeight $ value "500"
              fontSize $ rem 0.8
              width (rem 8.0)
              height $ rem 2.5
              color white
              cursor pointer
          -- COMPILER ERROR!!
          , HE.onClick $ const CreateUser ❽
          ]
          [ HH.text "Create User" ]
        ]
      ]
-- BELOW CODE REDACTED
```

❶ Added `themeColor` and `themeFont` to import list.

❷ Added these imports.

❸ Added `FontWeight`, `fontWeight` and `fontSize` to import list.

❹ Added `maxHeight`, `paddingLeft`, `width` and `height` to import list.

❺ Added `px` to import list.

❻ Replaced `HH.ul_` with `HH.ul` since it takes Properties. Also, added the styling for `ul`.

❼ Added this whole `div` as a sibling to the `ul`. It has a button as its child .

❽ This `Action` needs to be implemented, hence the compiler error.

And now implement the `CreateUser` Action and stub out its handling code. Then return here.

Here's those changes:

```
data Action
  = Initialize
  | UserSelected User
  | SelectedUserName (Maybe String)
  | Modal (Modal.Output Message.Output)
  | CreateUser ❶

  handleAction :: Action -> H.HalogenM State Action Slots Output m Unit
  handleAction = case _ of
-- CODE REDACTED
    CreateUser -> pure unit ❷
```

❶ Added this Data Constructor.

❷ Stubbed out this case.

Now rebuild the Application and refresh the Browser and see how it works so far. Don't expect our new button to work just yet since its code in `handleAction` is just a stub.

We'll need to add the `CreateUser` Modal to the Users Page, i.e. `Component.Users`. Use existing code that uses `Modal` to help you add the `CreateUser` Modal to the page.

When you're finished, return here and continue on.

Here are the code changes in `Component.Users`:

```
import Component.Modal.CreateUser as CreateUser ❶

type State =
  { authorized :: Boolean
```

```
  , selectedUser :: Maybe User
  , users :: Map String User
  , initUserName :: Maybe String
  , errorMessage :: Maybe String
  , creatingUser :: Boolean ❷
  }

data Action
  = Initialize
  | UserSelected User
  | SelectedUserName (Maybe String)
  | ErrorModal (Modal.Output Message.Output) ❸
  | CreateUserModal (Modal.Output CreateUser.Output) ❹
  | CreateUser

type Slots = ❺
  ( errorModal :: H.Slot Message.Query (Modal.Output Message.Output) Unit
  , createUserModal
      :: H.Slot CreateUser.Query (Modal.Output CreateUser.Output) Unit
  )

_errorModal = Proxy :: Proxy "errorModal" ❻
_createUserModal = Proxy :: Proxy "createUserModal"

-- ABOVE CODE REDACTED
component = H.mkComponent
  { initialState: \initUserName ->
      { authorized: false
      , selectedUser: Nothing
      , users: Map.empty
      , initUserName
      , errorMessage: Nothing
      , creatingUser: false ❼
      }
  , render
  , eval: H.mkEval H.defaultEval
      { handleAction = handleAction
      , receive = Just <<< SelectedUserName
      , initialize = Just Initialize
      }
  }
  where
  handleAction :: Action -> H.HalogenM State Action Slots Output m Unit
  handleAction = case _ of
```

```
-- CODE REDACTED
    ErrorModal output -> case output of ❽
      Modal.Affirmative -> H.modify_ _ { errorMessage = Nothing }
      Modal.Negative -> H.modify_ _ { errorMessage = Nothing }
      Modal.InnerOutput _ -> pure unit
-- BELOW CODE REDACTED

  render :: State -> H.ComponentHTML Action Slots m
  render { authorized, selectedUser, users, errorMessage, creatingUser } = ❾
-- CODE REDACTED
    , (errorMessage # maybe (HH.text "") \message ->
      HH.slot _errorModal unit ❿
        (Modal.component ModalCommon.errorConfig Message.component)
        message ErrorModal) ⓫
    , if not creatingUser then HH.text "" else ⓬
      HH.slot _createUserModal unit
        (Modal.component createUserConfig CreateUser.component)
        unit CreateUserModal
    ]
    where
      createUserConfig = Modal.defaultConfig ⓭
        { affirmativeLabel = "CREATE"
        , affirmativeDisabled = true
        }
```

❶ Added this import.

❷ Added flag to show `CreateUser` Modal.

❸ Renamed `Modal` to `ErrorModal`.

❹ Added `CreateUserModal`.

❺ Had to rename `modal` to `errorModal` and add `createUserModal`.

❻ Had to rename `_modal` to `_errorModal` and add `_createUserModal`.

❼ Initialized flag to NOT show `CreateUser` Modal.

❽ Changed `Modal` to `ErrorModal`.

❾ Added `creatingUser` to be destructured.

❿ Use renamed label, `_errorModal`.

⓫ Change `Action` from `Modal` to `ErrorModal`.

⓬ Added `CreateUser` Modal when `creatingUser` is `true`.

⓭ Configuration to override the Affirmative Button and initialize it disabled. This is used when we create the `CreateUser` Modal.

To eliminate the compiler error, stub out `CreateUserModal`:

```
  handleAction :: Action -> H.HalogenM State Action Slots Output m Unit
  handleAction = case _ of
-- CODE REDACTED
    CreateUser -> pure unit
    CreateUserModal _ -> pure unit ❶
```

❶ Stubbed out for now.

And now, implement the code for `CreateUser` in `handleAction`. Return here when you're ready to move on.

Here's that code:

```
  handleAction :: Action -> H.HalogenM State Action Slots Output m Unit
  handleAction = case _ of
-- CODE REDACTED
    CreateUser -> H.modify_ _ { creatingUser = true } ❶
```

❶ Set `creatingUser` in `State`.

Now, flesh out `CreateUserModal` in `handleAction`. Handle just the `Modal.Affirmative` and `Modal.Negative` cases and stub out `Modal.InnerOutput` for now. Then return back here and continue reading.

Here's that change:

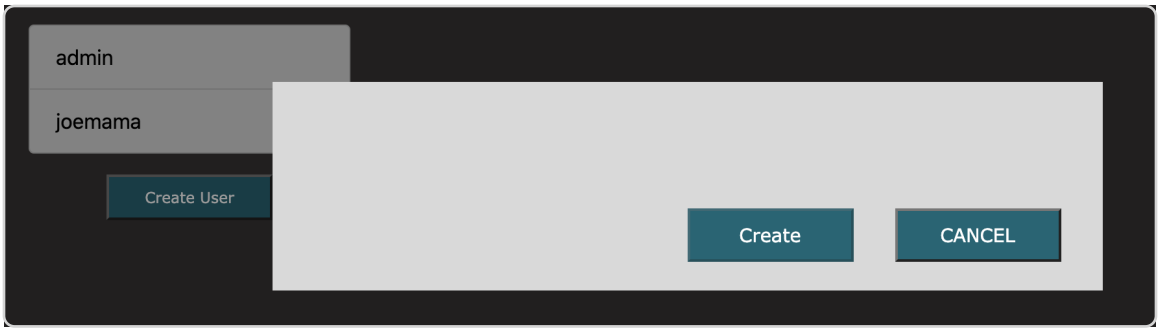
```
  handleAction :: Action -> H.HalogenM State Action Slots Output m Unit
  handleAction = case _ of
-- CODE REDACTED
    CreateUserModal output -> case output of
      Modal.Affirmative -> H.modify_ _ { creatingUser = false } ❶
      Modal.Negative -> H.modify_ _ { creatingUser = false } ❶
      Modal.InnerOutput _ -> pure unit ❷
-- BELOW CODE REDACTED
```

❶ Stop displaying the Modal.

❷ Do nothing for now.

Go ahead and test our `ChangeUser` Modal even though our `render` Function is stubbed out. To test, rebuild the Application, refresh the Browser, logon, click the `Users` button and then click the `Create User` button.

When I do I get:



Clicking the CANCEL button closes the modal, but clicking on the CREATE button doesn't do anything. That's because the UI doesn't reflect that it's disabled. We should make it work like the `Logon` does for the `LOGON` Button.

Look at how we handled styled a disabled button in `Logon` and do the same thing for disabled buttons in `Component.Modal`. When you're ready to move on, return here.

Here's the change:

```
import CSS.Color (rgba, white, gray) ❶
import CSS.Cursor (cursor, notAllowed) ❷

  render
    :: State iInput
    -> H.ComponentHTML (Action iInput iOutput) (Slots iQuery iOutput) m
  render { iInput, affirmativeDisabled, negativeDisabled } =
-- CODE REDACTED
        [ if not displayAffirmative then HH.text "" else
          HH.button
            [
              buttonStyle affirmativeDisabled ❸
            , HP.disabled affirmativeDisabled
            , HE.onClick $ const AffirmativeClicked
            ]
            [ HH.text affirmativeLabel]
        , if not displayNegative then HH.text "" else
          HH.button
            [
              buttonStyle negativeDisabled ❸
            , HP.disabled negativeDisabled
            , HE.onClick $ const NegativeClicked
            ]
            [ HH.text negativeLabel]
        ]
      ]
    ]
    where
      buttonStyle isDisabled = HC.style do ❹
        themeFont
        fontWeight $ FontWeight $ value "500"
        color if isDisabled then gray else white ❺
        when isDisabled $ cursor notAllowed ❻
        padding (rem 0.5) (rem 0.5) (rem 0.5) (rem 0.5)
        backgroundColor themeColor
        width (rem 8.0)
        marginLeft (rem 2.0)
        fontSize (rem 0.9)
```

❶ Added `gray` to the import list.

❷ Added this import.

❸ Pass whether the button is disabled in these 2 places to the `buttonStyle` helper.

❹ Added `isDisabled` Parameter.

❺ Make text `gray` if button is disabled and `white` if not.

❻ Only change the `cursor` to `notAllowed` if the button is disabled. Otherwise, this will leave the `cursor` unspecified and use the default button cursor, which MAY not be `pointer`.

The way we handle the cursor on an enabled button is better here than how we did it in `Component.Logon`. Let's change `Component.Logon` as follows:

```
import CSS.Cursor (cursor, notAllowed) ❶

-- ABOVE CODE REDACTED
      [ HH.button
        [
          HC.style do
            backgroundColor themeColor
            themeFont
            fontWeight $ FontWeight $ value "500"
            fontSize $ vw 1.0
            width (rem 20.0)
            height $ vw 3.0
            color if logonDisabled then gray else white
            when logonDisabled $ cursor notAllowed ❷
        , HE.onClick $ const Logon
        , HP.disabled logonDisabled
        ]
        [ HH.text "LOGON" ]
      ]
-- BELOW CODE REDACTED
```

❶ Removed `pointer` from import list.

❷ Made this code match what we did in `Modal`.

Now test again to see how the `CreateUser` works better, i.e. displays to the user that it's disabled.

When I run it, the button shows that it's disabled nicely now.

The next thing to add is the `render` Function in `Component.Modal.CreateUser`. Since it's mostly HTML and CSS logic, I'm just going to give you the code completed. Please take the time to look it over before typing it in since there's more than just render code here:

```
import Component.Modal.Common as ModalCommon ❶
import CSS.Common (center)
import CSS.Display (display, flex)
```

```
import CSS.Flexbox (alignSelf, flexBasis, flexEnd, flexGrow
                  , flexShrink, justifyContent)
import CSS.Geometry (marginBottom, marginRight, maxHeight, padding)
import CSS.Size (pct, rem)
import DOM.HTML.Indexed.InputType (InputType(..))
import Halogen.HTML.Events as HE
import Halogen.HTML.Properties as HP

data Action
  = CreateUser
  | Input (State -> State)
  | Check (State -> State) ❷
  | Modal (Modal.Output Message.Output)
  | RouteToLogon

  handleAction
    :: Action
    -> H.HalogenM State Action Slots (InnerOutput Output) m Unit
  handleAction = case _ of
    Input f -> H.modify_ f
    Check f -> H.modify_ f ❸
-- BELOW CODE REDACTED

  render :: State -> H.ComponentHTML Action Slots m
  render  { errorMessage } =
    HH.div_ [
      HH.ul_ [ ❹ ❺
        item "User Name" \s -> _ { userName = s }
      , itemPassword "Password" \s -> _ { password = s }
      , itemCheckbox "Administrator" \b -> _ { admin = b }
      , item "First Name" \s -> _ { firstName = s }
      , item "Last Name" \s -> _ { lastName = s }
      ]
    , (errorMessage # maybe (HH.text "") \message -> ❻
      HH.slot _modal unit
        (Modal.component ModalCommon.errorConfig Message.component)
        message Modal)
    ]
    where
      item_ labelStyle inputStyle label type_ valueInput = ❼
        HH.li [
          HC.style do
            display flex
            justifyContent flexEnd
```

```
            padding (rem 0.5) (rem 0.5) (rem 0.5) (rem 0.5)
        ]
        [ HH.label [
            HC.style $ (do
              alignSelf center
              flexGrow 1.0
              flexShrink 1.0
              flexBasis (pct 0.0)
              marginBottom (rem 0.0)
            ) *> labelStyle ❽
          ] [ HH.text $ label <> ":" ]
        , HH.input ([
            valueInput ❾
          , HC.style $ (do
              flexGrow 2.0
              flexShrink 2.0
              flexBasis (pct 0.0)
              maxHeight (rem 2.0)
              padding (rem 0.5) (rem 0.5) (rem 0.5) (rem 0.5)
            ) *> inputStyle ❿
          ] <> (type_ # maybe [] \t -> [ HP.type_ t ])) ⓫
        ]

      defItem = item_ (pure unit) (pure unit) ⓬

      item label = defItem label Nothing <<< textValueInput ⓭

      itemPassword label = defItem label (Just InputPassword)
        <<< textValueInput ⓮

      itemCheckbox label = item_ ⓯
        (do
          flexGrow 0.0
          flexBasis (pct 33.0))
        (do
          alignSelf center
          marginRight (rem 20.5))
        label
        (Just InputCheckbox)
        <<< checkInput

      textValueInput :: (String -> State -> State) -> _ ⓰ ⓱
      textValueInput inputFunc =
        HE.onValueInput $ Input <<< inputFunc ⓲
```

```
      checkInput :: (Boolean -> State -> State) -> _ ⓳ ⓴
      checkInput inputFunc =
        HE.onChecked $ Check <<< inputFunc ⓲
```

❶ Added these imports.

❷ Added `Action` to handle clicking on a Checkbox.

❸ Handle `Check`.

❹ Formatting and layout is controlled with an Unordered List.

❺ The children of `ul` are `li` items that use helpers in the `where` section below.

❻ `Modal` is a Child Component.

❼ General item helper, which is why it has so many Parameters.

❽ Additional label styles applied last to allow overrides.

❾ This is where `HE.onValueInput` or `HE.onClick` go.

❿ Additional input styles applied last to allow overrides.

⓫ Optional Property for Input `type`. Used for password and checkbox.

⓬ Creates a default item with no addition styling.

⓭ Creates a standard input item.

⓮ Creates a password input item.

⓯ Creates a checkbox input item.

⓰ Standard event handler for stardard input item.

⓱ `_` is Type `IProp ( onInput :: Event, value :: String | r ) Action`, i.e. a Property for `onInput`.

⓲ `inputFunc` in these 2 places is expecting a `String` first and then it'll return a Function that modifies the `State`, i.e. `State -> State`, which is what both Data Constructors, `Input` and `Check`, expect.

⓳ Special event handler for checkbox.

⓴ `_` is Type `IProp ( onChange :: Event, checked :: Boolean | r ) Action`, i.e. a Property for `onChange`.

The hardest part of this code to understand is the composition of `checkInput` and `testValueInput`, for example:

```
      defItem = item_ (pure unit) (pure unit)

      item label = defItem label Nothing <<< textValueInput ❶

      itemCheckbox label = item_
        (do
          flexGrow 0.0
          flexBasis (pct 33.0))
        (do
          alignSelf center
          marginRight (rem 20.5))
        label
        (Just InputCheckbox)
        <<< checkInput ❷
```

❶ `textValueInput` is composed with `defItem label Nothing`.

❷ `checkInput` is composed with the Partially Applied `item_`.

They are composed to allow the Functions that are passed to `item`, `itemPassword` and `itemCheckbox` to be passed to `checkInput` or `testValueInput` when used as:

```
-- ABOVE CODE REDACTED
      HH.ul_ [
        item "User Name" \s -> _ { userName = s }
      , itemPassword "Password" \s -> _ { password = s }
      , itemCheckbox "Administrator" \b -> _ { admin = b }
      , item "First Name" \s -> _ { firstName = s }
      , item "Last Name" \s -> _ { lastName = s }
      ]
-- BELOW CODE REDACTED
```

The next issue is that our Checkbox's background is `white`. It look terrible on our `paperColor` background. To make the checkbox match `paperColor`, add the following to the `<head>` section of `index.html`:

```
  <style>
    input[type='checkbox'] { filter: invert(15%) } ❶
  </style>
```

❶ Turns out that Browsers don't give us an easy way to change the background color of the Checkbox so we're forced to do this hack.

Next, we need to add code to enable the Create Button once there is data in all of the fields. We cannot apply

the same logic we used in `Logon` since the Component owns the buttons. We have to inform `Modal` that we want the button enabled.

We'll have to check on every `Input` to determine if we should enable or disable the Create Button. Do this in `Component.Modal.CreateUser` before reading any further.

Here's that change:

```
import Data.Array (filter, null) ❶
import Data.String.Common (trim)

  handleAction
    :: Action
    -> H.HalogenM State Action Slots (InnerOutput Output) m Unit
  handleAction = case _ of
    Input f -> do
      state <- H.modify $ f ❷
      H.raise if canCreate state then EnableAffirmative
              else DisableAffirmative ❸
-- CODE REDACTED
    where
    canCreate s = [ s.userName, s.password, s.firstName, s.lastName ] ❹
      <#> trim
      # filter (_ == "")
      # null

    errorMsg' msg = _ { errorMessage = Just msg }

    errorMsg = H.modify_ <<< errorMsg'
-- BELOW CODE REDACTED
```

❶ Added these imports.

❷ Modify `State` and return the updated `State`, which is why we're using `H.modify` without the `_`.

❸ If we can create a `User`, then enable the button, otherwise disable it.

❹ Take an `Array` of the field values that must be non-empty before we can create a `User`. Then trim them and filter only empty fields and then check to see if the `Array` is empty, which means none of the fields are blank.

Now, rebuild the Application, refresh the browser and create a new user and see what happens. Then return here.

When I do this, I cannot tell if anything got created. I can certainly see that that the new User isn't added to the list on the Users Page.

To check to see if it's there, refresh the browser and relogon and navigate to the Users Page to see if it was created. Return here when you have an answer.

Turns out that the new user isn't there. That meaning that we never created it.

Go to you `server` and check the last things in the logs to see if your `CreateUserRequest` was received. Then keep reading once you've figured that out.

When I check my `server`, the last thing I see is the `QueryUsersRequest`:

```
(Sun, 30 May 2021 16:07:42 GMT) REQUEST: { body:
"{\"contents\":{\"authToken\":\"ca23ed77-2778-4354-ba71-
a34e20452f2d\"},\"tag\":\"QueryUsersRequest\"}", headers: accept: */*
accept-encoding: gzip, deflate, br
accept-language: en-US,en;q=0.9
connection: keep-alive
content-length: 91
content-type: text/plain;charset=UTF-8
cookie: backupvs-session=d61cd30c-58b3-49b8-a4f2-56f76e6f5565
host: localhost:3000
origin: http://localhost:3000
referer: http://localhost:3000/
sec-ch-ua: " Not A;Brand";v="99", "Chromium";v="90", "Google Chrome";v="90"
sec-ch-ua-mobile: ?0
sec-fetch-dest: empty
sec-fetch-mode: cors
sec-fetch-site: same-origin
user-agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_15_7) AppleWebKit/537.36
(KHTML, like Gecko) Chrome/90.0.4430.212 Safari/537.36

, httpVersion: HTTP/1.1, method: Post, path: [], query: (fromFoldable []),
url: "/" } ((UUID 40851f10-c58a-46fb-a524-ea0ccd183b90))
(Sun, 30 May 2021 16:07:42 GMT) RESPONSE: { headers: Content-Length: 342

, status: 200 } [1.0 ms] ((UUID 40851f10-c58a-46fb-a524-ea0ccd183b90))
```

We're not calling our `server`, which means we're not handling the Create Button click properly.

Looking at the code, we see there's an `Action` called `CreateUser`:

```
data Action
  = CreateUser ❶
  | Input (State -> State)
  | Check (State -> State)
  | Modal (Modal.Output Message.Output)
  | RouteToLogon
```

❶ Here's `CreateUser`.

Now, look at the code to see what triggers `CreateUser` and return here once you've found the answer.

When I look, I realize that we don't own the Create Button. It's owned by `Modal`. That means that we need to have `Modal` tell the Inner Component when the buttons have been clicked.

We can do that by creating an `H.tell` query from `Modal` to the Inner Component. But, currently when the Parent wants to query the Inner Component, it queries with the Inner Component's Query Type, what we called `iQuery`.

But now, we're going to dictate a `Query` Type to our Inner Components in `Component.Modal` as follows:

```
-- COMPILER ERROR!!
data InnerQuery iQuery a ❶
  = AffirmativeClicked a ❷
  | NegativeClicked a ❸
  | PassThroughQuery iQuery a ❹
```

❶ Note the 2 Type Parameters. First, `iQuery` is the Inner Component's unique Query Type. And the second is the standard "rest of the computation" Type that all Query Algebras support.

❷ This will be sent from the `Modal` to Inner Component when the Affirmative Button has been clicked.

❸ This will be sent from the `Modal` to Inner Component when the Negative Button has been clicked.

❹ This is how the `Modal's` Parent can send Inner Component specific queries.

Parents of the `Modal` now will have to send Queries that are unique to the Inner Component using `PassThroughQuery`.

Add this Type to `Component.Modal` realizing that there will be compiler errors. Then return here to continue.

The reason for the compiler error is because we have used `AffirmativeClick` and `NegativeClicked` again. They were also used in our Action. Let's rename the `Action` ones:

```
data Action iInput iOutput
  = Input iInput
  | Output (InnerOutput iOutput)
  | AffirmativeClickedInternal ❶
  | NegativeClickedInternal ❶
```

❶ Added the suffix `Internal`.

Now, we'll need to change `handleAction`:

```
  handleAction
    :: Action iInput iOutput
    -> H.HalogenM (State iInput) (Action iInput iOutput)
                  (Slots iQuery iOutput) (Output iOutput) m Unit
  handleAction = case _ of
    Input input -> H.modify_ _ { iInput = input }
    Output innerOutput -> case innerOutput of
      PassThrough output -> H.raise $ InnerOutput output
      EnableAffirmative -> H.modify_ _ { affirmativeDisabled = false }
      DisableAffirmative -> H.modify_ _ { affirmativeDisabled = true }
      EnableNegative -> H.modify_ _ { negativeDisabled = false }
      DisableNegative -> H.modify_ _ { negativeDisabled = true }
      CloseAffirmative -> handleAction AffirmativeClickedInternal ❶
      CloseNegative -> handleAction NegativeClickedInternal ❶
    AffirmativeClickedInternal -> H.raise Affirmative ❶
    NegativeClickedInternal -> H.raise Negative ❶
```

❶ Added the suffix `Internal`.

And now, we'll need to change `render`:

```
  render
    :: State iInput
    -> H.ComponentHTML (Action iInput iOutput) (Slots iQuery iOutput) m
  render { iInput, affirmativeDisabled, negativeDisabled } =
-- CODE REDACTED
        [ if not displayAffirmative then HH.text "" else
          HH.button
            [
              buttonStyle affirmativeDisabled
            , HP.disabled affirmativeDisabled
            , HE.onClick $ const AffirmativeClickedInternal ❶
            ]
            [ HH.text affirmativeLabel]
        , if not displayNegative then HH.text "" else
          HH.button
            [
              buttonStyle negativeDisabled
            , HP.disabled negativeDisabled
            , HE.onClick $ const NegativeClickedInternal ❶
            ]
            [ HH.text negativeLabel]
        ]
      ]
    ]
    where
-- BELOW CODE REDACTED
```

❶ Added the suffix `Internal`.

Notice how we named one of `InnerQuery's` Data Constructors as `PassThroughQuery`. That's because we already had a `PassThrough` Data Constructor for `Output`:

```
data InnerOutput iOutput
  = PassThrough iOutput ❶
  | EnableAffirmative
  | DisableAffirmative
  | EnableNegative
  | DisableNegative
  | CloseAffirmative
  | CloseNegative
```

❶ Here's the Data Constructor in question.

Rename it to `PassThroughOutput` so there's no ambiguity and then fix the compiler errors that this change caused before returning here.

Here are the changes:

```
data InnerOutput iOutput
  = PassThroughOutput iOutput ❶
  | EnableAffirmative
  | DisableAffirmative
  | EnableNegative
  | DisableNegative
  | CloseAffirmative
  | CloseNegative

  handleAction
    :: Action iInput iOutput
    -> H.HalogenM (State iInput) (Action iInput iOutput)
                  (Slots iQuery iOutput) (Output iOutput) m Unit
  handleAction = case _ of
    Input input -> H.modify_ _ { iInput = input }
    Output innerOutput -> case innerOutput of
      PassThroughOutput output -> H.raise $ InnerOutput output ❶
-- BELOW CODE REDACTED
```

❶ Added `Output` suffix.

We're going to have to change `iQuery` to `InnerQuery iQuery` in `component` just like we did with `InnerOutput`:

```
component
  :: ∀ iQuery iInput iOutput m
   . MonadAff m
  => Config
  -> H.Component (InnerQuery iQuery) iInput (InnerOutput iOutput) m ❶
  -> H.Component (InnerQuery iQuery) iInput (Output iOutput) m ❷
```

❶ This change is for the same reason we did with `InnerOutput`.

❷ Added `InnerQuery` here also.

You may be wondering why the return Type for `component` uses `Output iOutput` Instead of `InnerOutput iOutput`. This is because `Modal` has its own `Output` but not its own `Query`, which is why the return Type uses `InnerQuery`.

We are also expecting the Inner Component's `Query` to be `InnerQuery`, so we'll need to change our `Slot`

defintion:

```
type Slots iQuery iOutput =
  ( inner :: H.Slot (InnerQuery iQuery) (InnerOutput iOutput) Unit ) ❶
```

❶ Changed `iQuery` to `InnerQuery iQuery`.

And we'll need to handle this `InnerQuery`, so `handleQuery` will change:

```
  handleQuery
    :: ∀ a
    .  (InnerQuery iQuery) a ❶
    -> H.HalogenM (State iInput) (Action iInput iOutput)
                  (Slots iQuery iOutput) (Output iOutput) m (Maybe a)
  handleQuery = H.query _inner unit
```

❶ We get an `InnerQuery` now, whereas, we used to get `iQuery`.

Now we need to Query the Inner Commponent to let it know that the buttons have been clicked:

```
  handleAction
    :: Action iInput iOutput
    -> H.HalogenM (State iInput) (Action iInput iOutput)
                  (Slots iQuery iOutput) (Output iOutput) m Unit
  handleAction = case _ of
-- CODE REDACTED
    AffirmativeClickedInternal -> do
      void $ H.tell _inner unit AffirmativeClicked ❶
      H.raise Affirmative
    NegativeClickedInternal -> do
      void $ H.tell _inner unit NegativeClicked ❶
      H.raise Negative
```

❶ Let Inner Component know about button clicks just like we let the Parent know on the next line.

When the Inner Component programatically clicks the buttons using `CloseAffirmative` or `CloseNegative`, the `Modal` will inform it that the button has been clicked, which, in this case, is redundant. In this scenario, the Inner Component can just ignore these `Queries`.

Now, we're going to need to change both of our Inner Components to support this new Query Type, i.e. `InnerQuery`.

We're going to start with `Component.Modal.Message`. Here's the current Type:

```
type Query = Const Void
```

We need to change it to `InnerQuery` from `Modal`:

```
import Component.Modal (InnerOutput, InnerQuery) ❶

type Query = Void ❷ ❸
```

❶ Added `InnerQuery` to import list.

❷ `Query` is no longer `Type -> Type`. It's now just `Type` since `InnerQuery` has Kind `Type -> Type`, which Halogen expects.

❸ We also do not need the Type Signature.

We can now delete:

```
import Data.Const (Const)
```

Now, fix the compiler error before reading on.

Here's the fix:

```
component
  :: ∀ m
  .  MonadAff m
  => H.Component (InnerQuery Query) Input (InnerOutput Output) m ❶
```

❶ Just like we did with `InnerOutput`, we added `InnerQuery`.

Now, we can turn our attention to `Component.Modal.CreateUser`. Go to that module and change `PassThrough` to `PassThroughOutput`:

```
-- ABOVE CODE REDACTED
          case createResponse of
            Left err -> H.modify_ _ { errorMessage = Just err }
            Right CreateUserResultsSuccess -> H.raise $ PassThroughOutput ❶
              $ User
                { userName
                , admin
                , firstName
                , lastName
                , temporaryPassword: true
                }
-- BELOW CODE REDACTED
```

❶ Changed `PassThrough` to `PassThroughOutput`.

Now, you'll notice that we have a compiler error regarding `Message.Query`. That's because we changed it to be `InnerQuery`. Fix that before reading on.

Here's that fix:

```
import Component.Modal (InnerOutput(..), InnerQuery) ❶

type Slots =
  ( modal :: H.Slot (InnerQuery Message.Query) ❷
                    (Modal.Output Message.Output) Unit )
```

❶ Added `InnerQuery` to import list.

❷ Wrapped `Message.Query` in `InnerQuery`.

This change will have to be done all throughout our code base. Search all occurences of `H.Slot Message.Query` in all Project Files. If you're using VSCode use `Shift-Cmd-F` or `Shift-Ctrl-F` for Windows.

Then fix them in the exact same way before reading any more.

Here are those fixes:

```
-- in Component.Logon
type Slots =
  ( modal :: H.Slot (Modal.InnerQuery Message.Query) ❶
                    (Modal.Output Message.Output) Unit )

-- in Component.Users
type Slots =
  ( errorModal :: H.Slot (Modal.InnerQuery Message.Query) ❶
                         (Modal.Output Message.Output) Unit
  , createUserModal :: H.Slot (Modal.InnerQuery CreateUser.Query) ❶
                              (Modal.Output CreateUser.Output) Unit
  )
```

❶ Applied `Modal.InnerQuery` in these 3 places.

There's a compiler error in `Component.Users` since we haven't finished changing `Component.Modal.CreateUser`. Return to that file and change `Query` like we did in `Component.Modal.Message`.

Return here to continue on.

Here's that change:

```
type Query = Void ❶ ❷
```

❶ Changed from `Const Void` to `Void`.

❷ Removed Type Signature.

We also deleted:

```
import Data.Const (Const)
```

And now, we have a compiler error. We've fixed this before in `Component.Modal.Message`. Fix it here before reading on.

Here's that fix:

```
component
  :: ∀ m
  .  MonadAff m
  => MonadAsk Env m
  => Navigate m Route
  => H.Component (InnerQuery Query) Input (InnerOutput Output) m ❶
```

❶ Wrapped `Query` in `InnerQuery`.

Now, return to `Component.Users` and save the file. It should compile.

Return to `Component.Modal.CreateUser`.

Next, we need to handle `Queries` from `Modal`, i.e. `AffirmativeClicked`. Add `handleQuery` and then copy all of the code from `handleAction` under the `CreateUser` case to the `AffirmativeClicked` case.

While you're at it, delete `CreateUser` from the `Action` Type since we're never going to use it.

Take a look at what we did in `handleQuery` for `Component.Router` to help refresh your memory on how to handle `Queries`.

Return here once you're ready to move forward.

Here's that code:

```
import Component.Modal (InnerOutput(..), InnerQuery(..)) ❶

data Action ❷
  = Input (State -> State)
  | Check (State -> State)
  | Modal (Modal.Output Message.Output)
  | RouteToLogon

  handleAction
    :: Action
    -> H.HalogenM State Action Slots (InnerOutput Output) m Unit
  handleAction = case _ of
-- BELOW CODE REDACTED ❸

-- ABOVE CODE REDACTED
  errorMsg' :: String -> State -> State ❹
  errorMsg' msg = _ { errorMessage = Just msg }

  errorMsg ❹
    :: String
```

```
    -> H.HalogenM State Action Slots (InnerOutput Output) m Unit
  errorMsg = H.modify_ <<< errorMsg'

  handleQuery ❺
    :: ∀ a
    .  InnerQuery Query a
    -> H.HalogenM State Action Slots (InnerOutput Output) m (Maybe a)
  handleQuery = case _ of
    AffirmativeClicked a -> do
      { userRef } <- ask
      loggedOnUser' <- H.liftEffect $ Ref.read userRef
      case loggedOnUser' of
        Nothing -> H.modify_ $ errorMsg' "Session has timed out"
          <<< _ { postModalAction = Just RouteToLogon }
        Just { authToken } -> do
          { userName, password, admin, firstName, lastName } <- H.get
          createResponse <- apiCall (CreateUserRequest
            { authToken
            , user:
              { userName
              , password
              , admin
              , firstName
              , lastName
              , temporaryPassword: true
              }
            })
          case createResponse of
            Left err -> H.modify_ _ { errorMessage = Just err }
            Right CreateUserResultsSuccess -> H.raise $ PassThroughOutput
              $ User
                { userName
                , admin
                , firstName
                , lastName
                , temporaryPassword: true
                }
            Right (CreateUserResultsFailure { reason }) -> case reason of
              NotAuthorized ->
                errorMsg "You're not authorized to create a new user."
              NotAuthenticated ->
                H.modify_ $ errorMsg' "Your session just timed out."
                  <<< _ { postModalAction = Just RouteToLogon }
              AlreadyExists ->
```

```
                errorMsg "A user with that User Name already exists."
              FileIOError err -> errorMsg
                $ "A fatal error on the Accounts File occurred: "
                <> err
      pure $ Just a ❻
    _ -> pure Nothing ❼
```

❶ Added `(..)` to `InnerQuery` to import Data Constructors.

❷ Deleted `CreateUser` Data Constructor.

❸ Cut code for `CreateUser` and pasted it to make `handleQuery`.

❹ Moved these 2 helper Functions out from the `where` section under `handleAction` to the same level as `handleAction` so that `handleQuery` could also use them. Also, added Type Signatures.

❺ Created from code cut from `handleAction`.

❻ Don't forget to return `Just a` as the last step of the `AffirmativeClicked` case so that our Query processing will be honored.

❼ Don't care about any other `Queries` since we don't need to do anything for `NegativeClicked` and `PassThroughQuery` will never happen since our `Query` is `Void`.

And we don't want to forget to evaluate `handleQuery`:

```
component = H.mkComponent
  { initialState: \_ ->
    { userName: ""
    , password: ""
    , admin: false
    , firstName: ""
    , lastName: ""
    , errorMessage: Nothing
    , postModalAction: Nothing
    }
  , render
  , eval: H.mkEval H.defaultEval
      { handleAction = handleAction
      , handleQuery = handleQuery ❶
      }
  }
```

❶ Added this.

Now, rebuild the Application and create a user called `test` with a first name of `Test` and a last of `Test`, then go to the `server` and check the `accounts.csv`. Then return back here.

When I do this, I get a 3rd Account for `test`.

You may have noticed that our Users list did NOT get updated. The `CreateUser` Inner Component is supposed to `Output` the `User` that got added.

The Users Page needs to listen for that and update its `State` accordingly.

First, make sure that `Component.Modal.CreateUser` is outputing the `User` on a successful create. Look at its `handleQuery` to confirm. Then return here once you've confirmed that it is.

Here's what I found:

```
          case createResponse of
            Left err -> H.modify_ _ { errorMessage = Just err }
            Right CreateUserResultsSuccess -> H.raise $ PassThroughOutput ❶
              $ User
                { userName
                , admin
                , firstName
                , lastName
                , temporaryPassword: true
                }
-- BELOW CODE REDACTED
```

❶ Calling `H.raise` to output to `Modal` and since it's a `PassThroughOutput`, `Modal` will forward `User` to its Parent.

Now that we've confirmed it's being sent, all we have to do is handle the `Output` from the `Modal` in `Component.Users`. Here's what we're doing now:

```
  render :: State -> H.ComponentHTML Action Slots m
  render { authorized, selectedUser, users, errorMessage, creatingUser } =
-- CODE REDACTED
    , if not creatingUser then HH.text "" else
      HH.slot _createUserModal unit
        (Modal.component createUserConfig CreateUser.component)
        unit CreateUserModal ❶
-- BELOW CODE REDACTED
```

❶ `Output` from this Child will be turned into the `Action` called `CreateUserModal`.

Go look at `CreateUserModal` in `handleAction` to see what we're doing with it before reading any further.

Here's what I found:

```
handleAction :: Action -> H.HalogenM State Action Slots Output m Unit
  handleAction = case _ of
-- CODE REDACTED
    CreateUserModal output -> case output of
      Modal.Affirmative -> H.modify_ _ { creatingUser = false }
      Modal.Negative -> H.modify_ _ { creatingUser = false }
      Modal.InnerOutput _ -> pure unit ❶
-- BELOW CODE REDACTED
```

❶ Here is the `Output` from the Inner Component.

We can see that we aren't doing anything with the `Output` from `Component.Modal.CreateUser`, which is a `User`.

Write the code in `Component.Users` to handle this. It should add the `User`, which is the first Parameter of `InnerOutput`, to our `State's` User Map, i.e. `users`.

Return here when that code is complete.

Here's that change:

```
handleAction :: Action -> H.HalogenM State Action Slots Output m Unit
  handleAction = case _ of
-- CODE REDACTED
    CreateUserModal output -> case output of
      Modal.Affirmative -> H.modify_ _ { creatingUser = false }
      Modal.Negative -> H.modify_ _ { creatingUser = false }
      Modal.InnerOutput user@(User { userName }) -> ❶
        H.modify_ \s ->
          s { users = Map.insert userName user s.users } ❷
-- BELOW CODE REDACTED
```

❶ Destructure `User` to get to the Record we're calling `user`.

❷ Inserting `User user` into the map with the key being `user.userName`. Notice we're Constructing `User` from the Record, `user`.

Now, rebuild the app and create a new user to see if it shows up in our list. This time call the user `test2` with first and last names as `Test`. Then return here.

When I do this, it doesn't work, i.e. our new user isn't added to the list. It may work for you, but let's move forward assuming it doesn't.

Now's a good time to step back for a moment and think of what's happening when we click the Create Button.

Let's look at the code starting at the `Modal` since it's the button owner. The button click turns into an `Action`. So look to `handleAction` to see what we're currently doing when we get `AffirmativeClickInternal` then read on.

This is what I find:

```
-- ABOVE CODE REDACTED
    AffirmativeClickedInternal -> do
      void $ H.tell _inner unit AffirmativeClicked ❶
      H.raise Affirmative ❷
-- BELOW CODE REDACTED
```

❶ First, we query/tell our Inner Component that the button was clicked.

❷ Second, we let our Parent know the button was clicked.

Now, ask yourself, what happens in the Inner Component when it gets `AffirmativeClicked`. Go look at `Component.Modal.CreateUser`, which is our Inner Component, and see what it does.

When I look, I see that it calls the API on the `server`.

Next, ask yourself, what happens in the Parent when it gets `Affirmative`. Go look at the Users Page to see, i.e. `Component.Users`.

When I look, I see that it clears the `creatingUser` flag:

```
-- ABOVE CODE REDACTED
    CreateUserModal output -> case output of
      Modal.Affirmative -> H.modify_ _ { creatingUser = false } ❶
      Modal.Negative -> H.modify_ _ { creatingUser = false }
      Modal.InnerOutput user@(User { userName }) -> do
        H.modify_ \s ->
          s { users = Map.insert userName user s.users }
-- BELOW CODE REDACTED
```

❶ Clear `creatingUser` on `Affirmative`.

This will trigger a re-render, which will NOT render the `CreateUser` Modal anymore. This destroys the Modal BEFORE we have a chance to handle the return call from the `server`.

I suspect we have a classic race condition and `CreateUser` is sometimes losing the race. This is why it may be working for you. But it may not ALWAYS work. If the communication between our Application and the server is slow, then there will be no Component to handle the response since the Parent has long since destroyed it by not rendering it.

Let's prove that's the case by going to `Component.Modal` and commenting out the line that notifies the

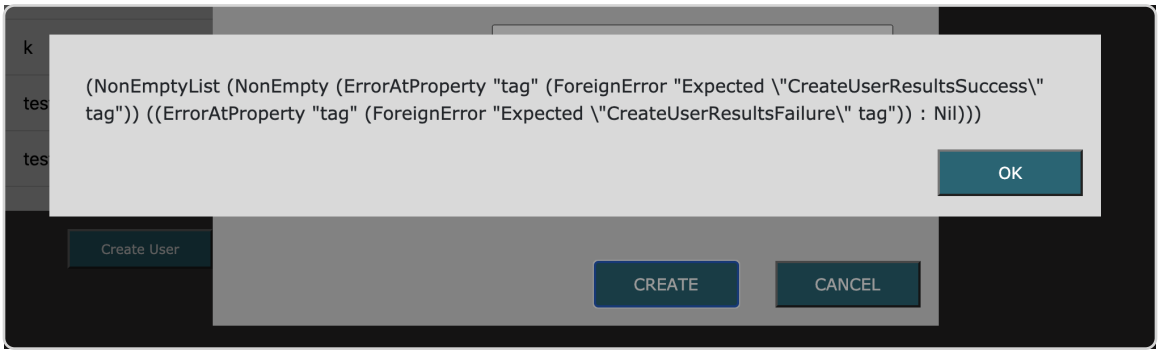
Parent:

```
-- ABOVE CODE REDACTED
    AffirmativeClickedInternal -> do
      void $ H.tell _inner unit AffirmativeClicked
      -- H.raise Affirmative ❶
-- BELOW CODE REDACTED
```

❶ Commented out to test race condition theory.

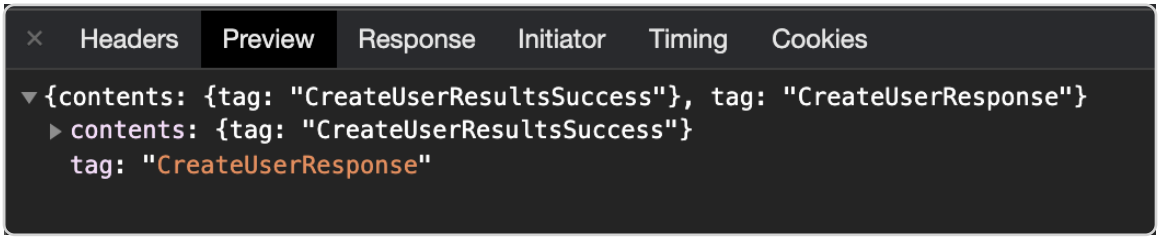
Now, build the Application and test again using a user called `test3` and see what you get. Then return here.

Here's what I get:



So, it looks like we have 2 problems. A race condition and a decoding problem.

When I run this test with the Developer Tools open in Chrome and then go to the Network Tab and check what the `server` sent us, I see this:



This shows us that the `server` sent the correct response. That must mean that we're telling the compiler to use the wrong decoder, i.e. somehow we've told it the wrong Type.

But checking the code in `Component.Modal.CreateUsers` that handles the create response shows us that we aren't giving the `apiCall` line an explicit Type for `createResponse`:

```
-- ABOVE CODE REDACTED
          createResponse <- apiCall (CreateUserRequest
            { authToken
            , user:
              { userName
              , password
              , admin
              , firstName
              , lastName
              , temporaryPassword: true
              }
            }) ❶
-- BELOW CODE REDACTED
```

❶ No explicit Type.

So, the compiler has to infer the Type from how we're using it. Here's how we're using it:

```
-- ABOVE CODE REDACTED
          case createResponse of
            Left err -> H.modify_ _ { errorMessage = Just err }
            Right CreateUserResultsSuccess -> do ❶
              H.raise $ PassThroughOutput
                $ User
                  { userName
                  , admin
                  , firstName
                  , lastName
                  , temporaryPassword: true
                  }
            Right (CreateUserResultsFailure { reason }) -> case reason of ❷
              NotAuthorized ->
                errorMsg "You're not authorized to create a new user."
              NotAuthenticated ->
                H.modify_ $ errorMsg' "Your session just timed out."
                  <<< _ { postModalAction = Just RouteToLogon }
              AlreadyExists ->
                errorMsg "A user with that User Name already exists."
              FileIOError err -> errorMsg
                $ "A fatal error on the Accounts File occurred: "
                <> err
-- BELOW CODE REDACTED
```

❶ We're expecting a `CreateUserResultsSuccess`.

❷ We're expecting a `CreateUserResultsFailure`.

It should be clear that this code is bad. We're missing the fact that `CreateUserResultsSuccess` and `CreateUserResultsFailure` are wrapped in `CreateUserResponse`.

Fix my mistake in `Component.Modal.CreateUser` and then keep reading.

Here's that fix:

```
import Data.Api.CreateUser (CreateUserRequest(..) , CreateUserResults(..)
                          , CreateUserFailureReason(..)
                          , CreateUserResponse(..)) ❶

-- ABOVE CODE REDACTED
          case createResponse of
            Left err -> H.modify_ _ { errorMessage = Just err }
            Right (CreateUserResponse CreateUserResultsSuccess) -> do ❷
              H.raise $ PassThroughOutput
                $ User
                  { userName
                  , admin
                  , firstName
                  , lastName
                  , temporaryPassword: true
                  }
            Right (CreateUserResponse ❷
                (CreateUserResultsFailure { reason })) -> case reason of
              NotAuthorized ->
                errorMsg "You're not authorized to create a new user."
              NotAuthenticated ->
                H.modify_ $ errorMsg' "Your session just timed out."
                  <<< _ { postModalAction = Just RouteToLogon }
              AlreadyExists ->
                errorMsg "A user with that User Name already exists."
              FileIOError err -> errorMsg
                $ "A fatal error on the Accounts File occurred: "
                <> err
-- BELOW CODE REDACTED
```

❶ Added `CreateUserResponse` to import list.

❷ Wrapped in `CreateUserResponse` in both places.

Now rebuild and retest calling the user `test4` and see what happens.

When I do this, it decodes properly now and adds the new user to the list. But, of course, we still need to tell the Parent because the Modal never goes away.

We must also fix our race condition. What we need to do is create a way for the Inner Component to let `Modal` know when it's safe to tell the Parent about the button click. This will give the Inner Component time to do whatever it needs to do before the Modal is destroyed. In our case, we want to create the user and then handle the response accordingly.

We already have a path from the Inner Component to its Parent, i.e. to `Modal`. This mechanism is called `InnerOutput`:

```
data InnerOutput iOutput
  = PassThroughOutput iOutput
  | EnableAffirmative
  | DisableAffirmative
  | EnableNegative
  | DisableNegative
  | CloseAffirmative
  | CloseNegative
```

You may think that `CloseAffirmative` or `CloseNegative` might solve our race condition, but once you see their implementations, you won't think so:

```
handleAction
    :: Action iInput iOutput
    -> H.HalogenM (State iInput) (Action iInput iOutput)
                  (Slots iQuery iOutput) (Output iOutput) m Unit
  handleAction = case _ of
    Input input -> H.modify_ _ { iInput = input }
    Output innerOutput -> case innerOutput of
      PassThroughOutput output -> H.raise $ InnerOutput output
      EnableAffirmative -> H.modify_ _ { affirmativeDisabled = false }
      DisableAffirmative -> H.modify_ _ { affirmativeDisabled = true }
      EnableNegative -> H.modify_ _ { negativeDisabled = false }
      DisableNegative -> H.modify_ _ { negativeDisabled = true }
      CloseAffirmative -> handleAction AffirmativeClickedInternal ❶
      CloseNegative -> handleAction NegativeClickedInternal ❶
-- BELOW CODE REDACTED
```

❶ These recurse and act as if the button was clicked by the user.

Using this mechanism would result in an infinite loop since they just call the same handler for the button click. This is how this whole process got started.

We'll need to have a different message for letting the Parent know. For our `Message` Inner Component, we'll output this Value immediately, but for the `CreateUser` Inner Commponent, we'll want to wait until we've successfully created a user.

In the case of the `NegativeClicked` Query from `Modal`, we'll immediately let `Modal` know that it's safe to let the Parent know of the button click.

This sounds more complex than it is. Let's start by adding the new `InnerOutputs` in `Component.Modal`:

```
data InnerOutput iOutput
  = PassThroughOutput iOutput
  | EnableAffirmative
  | DisableAffirmative
  | EnableNegative
  | DisableNegative
  | CloseAffirmative
  | CloseNegative
  | ParentAffirmative ❶
  | ParentNegative ❶
```

❶ New outputs.

Now, change `handleAction` to handle these new outputs..

Return here when you've completed that step.

Here's that change:

```
handleAction
  :: Action iInput iOutput
  -> H.HalogenM (State iInput) (Action iInput iOutput)
                (Slots iQuery iOutput) (Output iOutput) m Unit
handleAction = case _ of
  Input input -> H.modify_ _ { iInput = input }
  Output innerOutput -> case innerOutput of
    PassThroughOutput output -> H.raise $ InnerOutput output
    EnableAffirmative -> H.modify_ _ { affirmativeDisabled = false }
    DisableAffirmative -> H.modify_ _ { affirmativeDisabled = true }
    EnableNegative -> H.modify_ _ { negativeDisabled = false }
    DisableNegative -> H.modify_ _ { negativeDisabled = true }
    CloseAffirmative -> handleAction AffirmativeClickedInternal
    CloseNegative -> handleAction NegativeClickedInternal
    ParentAffirmative -> H.raise Affirmative ❶
    ParentNegative -> H.raise Negative ❶
  AffirmativeClickedInternal -> ❷
    void $ H.tell _inner unit AffirmativeClicked ❸
  NegativeClickedInternal -> ❷
    void $ H.tell _inner unit NegativeClicked ❸
```

❶ Let Parent know ONLY when the Inner Component tells us it's safe to do so. These were cut and pasted from `AffirmativeClickInternal` and `NegativeClickedInternal`.

❷ Removed the superfluous `do`.

❸ Removed the code to tell the Parent. One of those should have been commented out from when we did our test.

Now, change `Component.Modal.Message` to tell `Modal` that it's safe to let the Parent know about the button click as soon as it happens.

Return here when you're done or if you get stuck.

If you got stuck, look at how `Component.Modal.CreateUser` finds out about button clicks. You'll need to implement a similar mechanism in `Component.Modal.Message`.

Return here when you're ready to move forward.

Here's that change:

```
import Component.Modal (InnerOutput(..), InnerQuery(..)) ❶
import Data.Maybe (Maybe(..)) ❷

  handleQuery
    :: ∀ a
    .  InnerQuery Query a
    -> H.HalogenM State Action Slots (InnerOutput Output) m (Maybe a)
  handleQuery = case _ of
    AffirmativeClicked a -> do
      H.raise ParentAffirmative ❸
      pure $ Just a ❹
    NegativeClicked a -> do ❺
      H.raise ParentNegative
      pure $ Just a
    _ -> pure Nothing ❻
```

❶ Added `(..)` on both Types to get their Data Constructors.

❷ Added this import.

❸ Let `Modal` know that it can tell the Parent of the button click.

❹ Remember to return `Just a` so our Query will be honored.

❺ Here is the same logic for the `Negative` button.

❻ We simply do nothing for all other Queries.

We cannot forget to add `handleQuery` to the `EvalSpec`:

```
component
  :: ∀ m
  .  MonadAff m
  => H.Component (InnerQuery Query) Input (InnerOutput Output) m
component = H.mkComponent
  { initialState: \message -> { message }
  , render
  , eval: H.mkEval H.defaultEval
    { handleQuery = handleQuery ❶
    }
  }
```

❶ Support `Queries`.

Now, let `Modal` know it's safe to let its Parent know about the `Affirmation` Button click after a user has been successfully created in `Component.Modal.CreateUser`.

When the `Negative` button is clicked then let `Modal` know it can tell its Parent immediately. Then return back here to continue.

Here are those changes:

```
  handleQuery
    :: ∀ a
    .  InnerQuery Query a
    -> H.HalogenM State Action Slots (InnerOutput Output) m (Maybe a)
  handleQuery = case _ of
    AffirmativeClicked a -> do
-- CODE REDACTED
          case createResponse of
            Left err -> H.modify_ _ { errorMessage = Just err }
            Right (CreateUserResponse CreateUserResultsSuccess) -> do ❶
              H.raise $ PassThroughOutput
                $ User
                  { userName
                  , admin
                  , firstName
                  , lastName
                  , temporaryPassword: true
                  }
              H.raise ParentAffirmative ❷
            Right (CreateUserResponse
                (CreateUserResultsFailure { reason })) -> case reason of
              NotAuthorized ->
                errorMsg "You're not authorized to create a new user."
              NotAuthenticated ->
                H.modify_ $ errorMsg' "Your session just timed out."
                  <<< _ { postModalAction = Just RouteToLogon }
              AlreadyExists ->
                errorMsg "A user with that User Name already exists."
              FileIOError err -> errorMsg
                $ "A fatal error on the Accounts File occurred: "
                <> err
      pure $ Just a
    NegativeClicked a -> do ❸
      H.raise ParentAffirmative
      pure $ Just a
    _ -> pure Nothing
```

❶ Added the `do`.

❷ On a SUCCESSFUL call to `CreateUser`, we let `Modal` know that it can tell the Parent about the button click. Notice how we do this AFTER we send the newly created `User` to the `Modal's` Parent. This way we ensure that it gets through first, just in case. We wouldn't want another race condition.

❸ Immediately let `Modal` know that it can tell the Parent of the `Negative` Button click.

Now, our new mechanism is complete and we're ready to test. Rebuild the Application and retest and see what happens now. Call the new user `test5`.

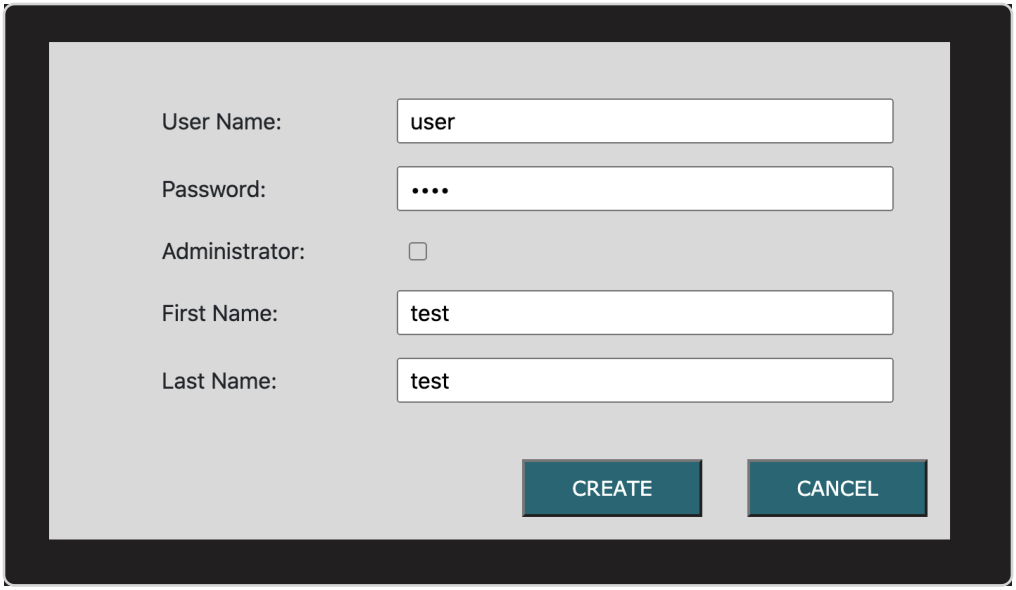
When I do this, it works!!! And notice how our list of users is updated.

Also, notice how it's sorted. We didn't make it sorted, but somehow it got sorted. Here's most likely why:

```
    [ HH.div [
      HC.style do
        display flex
        flexDirection column
        minWidth (rem 20.0)
        paddingRight (rem 2.0)
      ]
      [ HH.ul [
          HC.style do
            maxHeight (pct 80.0)
            overflowY overflowAuto
        ]
          (Map.values users # fromFoldable ❶
            <#> \(user@(User { userName })) ->
-- BELOW CODE REDACTED
```

❶ Looks like `Map.values` gets the Values in the Key order.

There is one other problem that I see when I run the following test:



Please run this same test. If you already have a user with that `userName`, then just use `user2`. But make sure you have the first and last names EXACTLY how I've typed it here.

Once the user is created, go to your `server` project and restart the server. What happens?

Here's what I get:

```
Cannot load accounts: (ParseError "Character 't' did not satisfy predicate"
(Position { line: 3, column: 146 }))
```

This is a problem on line 3, which for me is as follows:

```
user,3b7b539fcfa135efbc1c71d2a4edc996042424638884d8b6d9053cfa71914a7628d8fe3dd
00e131537c9d82799bd58d5da86aff0ae905a26c3054a3decd70405,true,false,test,test
                                                                 ^
                                                                 |
This is column 146 ----------------------------------------------+
```

Why do you think this is?

If you remember correctly, our Parser doesn't allow lowercase letters to start a First Name. It also doesn't allow uppercase letters except for the first character.

We are doing ZERO validation at the API.

There are many solutions:

1. Validate in the UI.
2. Allow lowercase characters, i.e. change the parser.

3. Do a round trip check encoding and decoding of the user in the server to make sure it can be read back. In fact, checking that our decoded user is the same as the one we started would be even better.

We should really do #1.

#2 is just giving up.

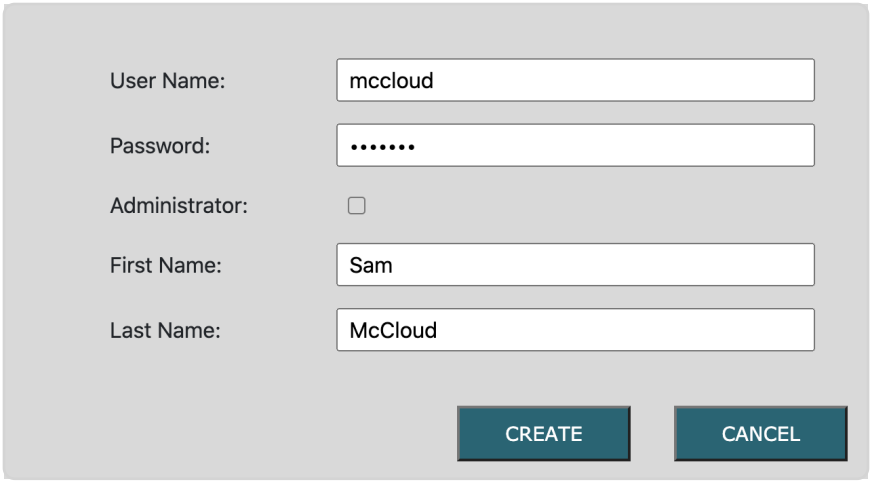
#3 should also be done but if the client is our ONLY client, doing #1 fixes this. But this is still a dangerous excuse. This too should really be done.

We, however, are going to do none of these. We've done plenty and I trust that if you've gotten this far that you can easily implement #1 and #3.

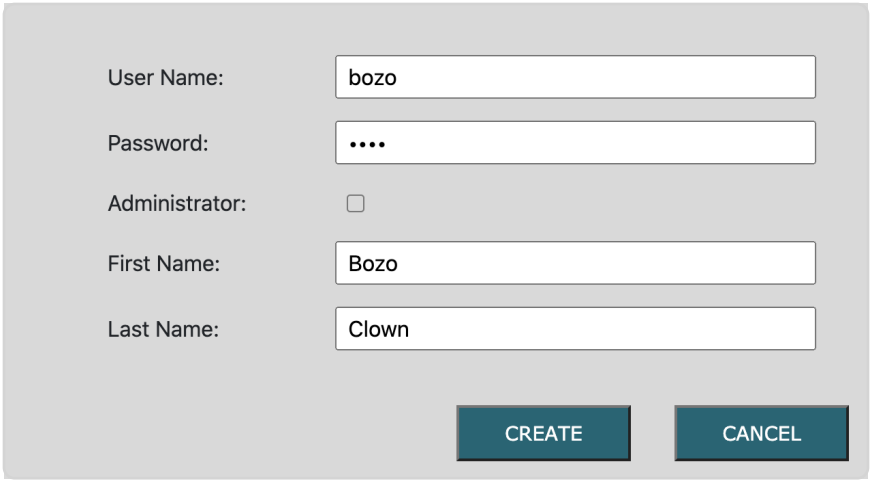
Just keep in mind that you may have to either edit or delete your `accounts.csv` if you forget to enter uppercase characters in the First and Last Name fields.

And since we did just that, edit the `accounts.csv` and delete the offending line. Then start up your server.

Once your server is up and running again, test it with another type of bad name by creating a user with the last name of `McCloud`:



After creating `mccloud`, create another user and make sure to make that user unique enough so you'll remember them:



Now, restart your server. Now, re-logon to the Application and navigate to the Users Page.

Notice how your unique user is MISSING?

This is because of a problem in our parser. To understand why, let's revisit our parser.

First, there an important thing to notice about `accountParser`:

```
accountParser :: AccountParserT Account
accountParser = do
  userName' <- userName # comma
  passwordHash' <- passwordHash # comma
  temporaryPassword' <- temporaryPassword # comma
  admin' <- admin # comma
  firstName' <- firstName # comma
  lastName' <- lastName
  pure $ Account
    { userName: userName'
    , passwordHash: passwordHash'
    , temporaryPassword: temporaryPassword'
    , admin: admin'
    , firstName: firstName'
    , lastName: lastName' ❶
    }
  where
    comma :: ∀ a. AccountParserT a -> AccountParserT a
    comma p = p <* char ','
```

❶ Notice here that `lastName` has nothing following it.

Since, nothing follows the `lastName`, the account line parsers just fine. Had we put `McCloud` for the `firstName`, we would've gotten a parser error since a name of `McCloud` is invalid and a comma and `lastname` follows.

Here's the implementation for names as a reminder:

```
properName :: AccountParserT String
properName = do
  first <- satisfy (isUpper <<< codePointFromChar)
  rest <- many $ satisfy (isLower <<< codePointFromChar) ❶
  pure $ fromCharArray (first : rest)

firstName :: AccountParserT String
firstName = properName

lastName :: AccountParserT String
lastName = properName
```

❶ Only accept zero or more lowercase characters.

`lastName` will parse our input `String` right up until it hits the `C` in `McCloud`. That's because it's not a lowercase character.

Then the next thing that parses starting at the offending `C` is the `sepBy` parser:

```
accountsParser :: AccountParserT (Array Account)
accountsParser =
  (Just <$> accountParser <|> pure Nothing)
    `sepBy` char '\n' <#> catMaybes <<< fromFoldable ❶
```

❶ `sepBy's` parser is `char`, which is used here to look for a `\n`.

`char '\n'` will fail to parse when it encounters the `C`. This will terminate the parsing of the file data WITH NO ERROR.

But why no error?

That's because we wrote `accountsParser` to parse the file one line at a time and keep parsing as long as every line ends in `\n`.

But in the case of `McCloud` as the `lastName`, the line parses only up to `Mc`.

Then `sepBy` doesn't find a `\n` next and it finishes successfully.

We never check to see if there's anything left to parse in the file, i.e. we never check to see if we stopped prematurely.

We need to have a parser that will check for ANY character and then fail on a successful match. This parser will succeed if it finds NO characters. This way we can use it to see if there are any characters left unparsed.

In our example, there are lots of unparsed characters, viz. `Cloud\n` and a whole other line.

Here's the code to fix our parser bug in our `server` project in file `Parser.Account`:

```
import Text.Parsing.Parser.Combinators (sepBy, notFollowedBy) ❶
import Text.Parsing.Parser.String (satisfy, char, string, anyChar) ❷

accountsParser :: AccountParserT (Array Account)
accountsParser =
  ((Just <$> accountParser <|> pure Nothing)
    `sepBy` char '\n' <#> catMaybes <<< fromFoldable) ❸
    <* notFollowedBy anyChar ❹ ❺
```

❶ Added `notFollowedBy` to import list.

❷ Added `anyChar` to import list.

❸ Had to put Parentheses around this block. This is a common requirement when many Binary Operators are involved.

❹ We are using `applyFirst`, i.e. `<*`, so we get the result of just our parser as we had in our old code. But we still want to execute the additional parsing.

❺ `notFollowedBy` will try the supplied parser and FAIL if that parser succeeds.

Change your code and rebuild and run your server to see the error message.

When I run this, I get:

```
Cannot load accounts: (ParseError "Negated parser succeeded" (Position { line:
10, column: 156 }))
```

This generic error message isn't very helpful. So we'll change the code to add a custom message by using `alt` and `fail`:

```
accountsParser :: AccountParserT (Array Account)
accountsParser =
  ((Just <$> accountParser <|> pure Nothing)
    `sepBy` char '\n' <#> catMaybes <<< fromFoldable)
    <* (notFollowedBy anyChar <|> fail "Failed to parse complete file") ❶ ❷
```

❶ Had to add Parentheses to this block.

❷ If `notFollowedBy` fails, then we'll try to use the other parser, which always fails, but with a message that tells us so much more.

Update your code to make this improvement and rebuild and run your server to see the new error message.

When I run this, I get:

```
Cannot load accounts: (ParseError "Failed to parse complete file" (Position {
line: 10, column: 155 }))
```

This message is so much better than the old generic one.

Now, edit your `accounts.csv` to remove the `McCloud` account and restart your server. Then return to your `client` project.

Now that we've fixed our parser, there's one last thing I'd like to do before we leave the Users Page. I'd like the newly added User to be the `selectedUser`. This way it's selected in the list and the information will show on the right.

Make that change to `Component.Users` and then return here.

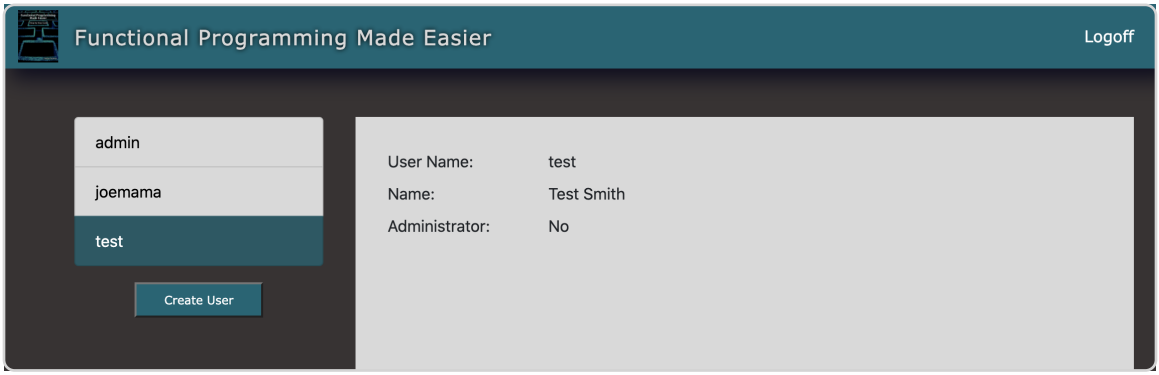
Here's that change:

```
-- ABOVE CODE REDACTED
    CreateUserModal output -> case output of
      Modal.Affirmative -> H.modify_ _ { creatingUser = false }
      Modal.Negative -> H.modify_ _ { creatingUser = false }
      Modal.InnerOutput user@(User { userName }) ->
        H.modify_ \s ->
          s { users = Map.insert userName user s.users
            , selectedUser = Just user } ❶
-- BELOW CODE REDACTED
```

❶ Make the newly created user the selected user.

Now, rebuild, refresh the Browswer and create a new user.

Here's how it should looks right after hitting the Create Button:



30.16. Logoff Page

The Logoff Page will be a fairly simple. When we navigate to that Page, it will call our backend to log the user off and display a message letting the user know that they have successfully logged off. This message will be on the page, not a Modal. The page will also have a button for them to log back on. Clicking that button will take them back to the Logon Page.

The Logoff Page will also display a Modal if there are errors in the API call.

We have done all of this before and so we can steal layouts from ChangePassword Page and Modals from the Logon or Users Page.

Before you begin coding, create a File called `Logoff.purs` under `src/Component`.

Since this is that last exercise for this Application, try to code the whole page without reading to see my solution. Also, change the Page Component's `Logoff` button to navigate to the Logoff Page.

Make sure you understand the specifications before forging ahead.

Then return here to see how you did or to get over any small roadblocks.

The first thing I did was to copy and paste all of the code from `Component.ChangePassword` to `Component.Logoff`.

This is usually bad practice since we're creating a lot of duplicate code and we should really try to see where we can generalize things, but the code is so small and this is just a learning application after all, so we're just going to go with it.

After pasting the code, these are the changes that need to be made:

```
module Component.Logoff where ❶

-- ABOVE CODE REDACTED
  handleAction :: Action -> H.HalogenM State Action Slots Output m Unit
  handleAction = case _ of
    Click -> navigate Route.Logon ❷

  render :: State -> H.ComponentHTML Action Slots m
  render _ =
    HH.div [
      HC.style do
        display flex
        flexDirection row
        alignItems center
        justifyContent center
        paddingTop $ vw 0.65
    ]
    [ HH.text "You have been successfully logged off" ❸
    , HH.button [
        HC.style do
          backgroundColor themeColor
          themeFont
          fontWeight $ FontWeight $ value "500"
          fontSize $ vw 1.0
          width (rem 20.0)
          height $ vw 3.0
          color white
          cursor pointer
      , HE.onClick $ const Click
      ] [ HH.text "Log Back On" ] ❹
    ]
```

❶ Renamed the module to match the File name.

❷ Make the button click route to `Logon`.

❸ Added text to the page to let the user know that they've been logged off. I know that this will have to be conditional at some point.

❹ Changed the button text. This too will need to be conditional at some point.

Next, we'll copy what we did with the `Component.Modal.Message` in `Logon`:

```
import Component.Modal as Modal ❶
import Component.Modal.Common as ModalCommon ❶
```

```
import Component.Modal.Message as Message ❶
import Data.Maybe (Maybe(..), maybe) ❷
import Type.Proxy (Proxy(..)) ❶

type State =
  { errorMessage :: Maybe String ❸
  }

data Action
  = Click
  | Modal (Modal.Output Message.Output) ❹

type Slots =
  ( modal :: H.Slot (Modal.InnerQuery Message.Query)
                    (Modal.Output Message.Output) Unit ) ❺

_modal = Proxy :: Proxy "modal" ❻

-- ABOVE CODE REDACTED
component = H.mkComponent
  { initialState: \_ -> { errorMessage: Nothing } ❼
  , render
  , eval: H.mkEval H.defaultEval
      { handleAction = handleAction
      }
  }
  where
  handleAction :: Action -> H.HalogenM State Action Slots Output m Unit
  handleAction = case _ of
    Click -> navigate Route.Logon
    Modal output -> case output of ❽
      Modal.Affirmative -> H.modify_ _ { errorMessage = Nothing }
      Modal.Negative -> H.modify_ _ { errorMessage = Nothing }
      Modal.InnerOutput _ -> pure unit

  render :: State -> H.ComponentHTML Action Slots m
  render { errorMessage } = ❾
    HH.div [
      HC.style do
        display flex
        flexDirection row
        alignItems center
        justifyContent center
        paddingTop $ vw 0.65
    ]
```

```
    [ HH.text "You have been successfully logged off"
    , HH.button [
        HC.style do
          backgroundColor themeColor
          themeFont
          fontWeight $ FontWeight $ value "500"
          fontSize $ vw 1.0
          width (rem 20.0)
          height $ vw 3.0
          color white
          cursor pointer
      , HE.onClick $ const Click
      ] [ HH.text "Log Back On" ]
    , (errorMessage # maybe (HH.text "") \message -> ⓾
      HH.slot _modal unit
        (Modal.component ModalCommon.errorConfig Message.component)
        message Modal)
    ]
```

❶ Added these imports.

❷ Added `maybe` to import list.

❸ Added the optional error message.

❹ Added `Modal` Action.

❺ Added Row Type Label for Modal Child Component.

❻ Added Symbol Proxy for Modal Child.

❼ Initialized `State`.

❽ Handle Modal Actions.

❾ Destructure `State` to get `errorMessage`.

❿ Added Modal Child Component rendering.

The next thing to do is to optionally display the message and the button to relogon. First, we'll need something in the `State` that lets us know that we've logged off:

```
type State =
  { loggedOff :: Boolean ❶
  , errorMessage :: Maybe String
  }

-- ABOVE CODE REDACTED
component = H.mkComponent
  { initialState: \_ ->
    { loggedOff: false ❷
    , errorMessage: Nothing
    }
  , render
  , eval: H.mkEval H.defaultEval
      { handleAction = handleAction
      }
  }
```

❶ This will be initialized to `false` and only set to `true` when we get a successful `Logoff` API response.

❷ Initialize `loggedOff` to `false` since we must be logged on for the Router Component to display this page.

And now we can conditionally display the message and button:

```
render :: State -> H.ComponentHTML Action Slots m
render { loggedOff, errorMessage } = ❶
  HH.div
    [
      HC.style do
        display flex
        flexDirection row
        alignItems center
        justifyContent center
        paddingTop $ vw 0.65
    ]
    if loggedOff then ❷
      [ HH.text "You have been successfully logged off"
      , HH.button
        [
          HC.style do
            backgroundColor themeColor
            themeFont
            fontWeight $ FontWeight $ value "500"
            fontSize $ vw 1.0
            width (rem 20.0)
            height $ vw 3.0
            color white
            cursor pointer
        , HE.onClick $ const Click
        ]
        [ HH.text "Log Back On" ]
      ]
    else [] <> ❸
      [ (errorMessage # maybe (HH.text "") \message ->
        HH.slot _modal unit
          (Modal.component ModalCommon.errorConfig Message.component)
          message Modal)
      ]
```

❶ Destructure to get `loggedOff`.

❷ If `loggedOff` then prepend this `Array` of HTML Elements to the Modal Child Element.

❸ Otherwise, prepend an empty `Array`.

And finally, we need to call the Logoff API when we land on the Page. That will warrant an `Intialize` Action:

```
data Action
  = Initialize ❶
  | Click
  | Modal (Modal.Output Message.Output)
```

❶ Added `Initialize`.

And now, we'll handle the `Initialize` Action by making the API call to log the user off by looking at what we did on the Users Page:

```
import Control.Monad.Reader.Class (class MonadAsk, ask) ❶
import Data.Api.Logoff (LogoffRequest(..), LogoffResponse(..)
                       , LogoffResults(..))
import Data.Either (Either(..))
import Effect.Ref as Ref
import Env (Env)
import Utils (apiCall)

component
  :: ∀ m
  .  MonadAff m
  => MonadAsk Env m ❷
  => Navigate m Route
  => H.Component Query Input Output m
component = H.mkComponent
  { initialState: \_ ->
    { loggedOff: false
    , errorMessage: Nothing
    }
  , render
  , eval: H.mkEval H.defaultEval
      { initialize = Just Initialize ❸
      , handleAction = handleAction
      }
  }
  where
  handleAction :: Action -> H.HalogenM State Action Slots Output m Unit
  handleAction = case _ of
    Initialize -> do ❹
      { userRef } <- ask
      loggedOnUser' <- H.liftEffect $ Ref.read userRef
      loggedOnUser' # maybe (pure unit) \{ authToken } -> do ❺
        logoffResponse <- apiCall (LogoffRequest { authToken })
        case logoffResponse of
          Left err -> H.modify_ _ { errorMessage = Just err }
          Right (LogoffResponse LogoffResultsFailure) ->
            H.modify_ _ { errorMessage = Just $ "Unable to logoff" } ❻
          Right (LogoffResponse LogoffResultsSuccess) -> do
            H.liftEffect $ Ref.write Nothing userRef ❼
            H.modify_ _ { loggedOff = true } ❽
-- BELOW CODE REDACTED
```

❶ Added these imports.

❷ Added `MonadAsk` Constraint so we can call `ask`.

❸ Handle `initialize`.

❹ Handle `Initialze` case.

❺ Get `authToken` from `LoggedOnUser` to make API call.

❻ Display Modal if API call fails.

❼ Clear out the `userRef` since the user's no longer logged in.

❽ Set `loggedOff` to `true` so `render` will display message and button.

And the final step before we test is to change the Router Component to display the Logoff Component:

```
import Component.Logoff as Logoff ❶

  render :: State -> H.ComponentHTML Action Slots m
  render { route } = case route of
    Logon -> HH.slot_ _logon unit (defaultPage Logon.component) unit
    Logoff ->
      HH.slot_ _logoff unit (defaultPage Logoff.component) unit ❷
-- BELOW CODE REDACTED
```

❶ Added import.

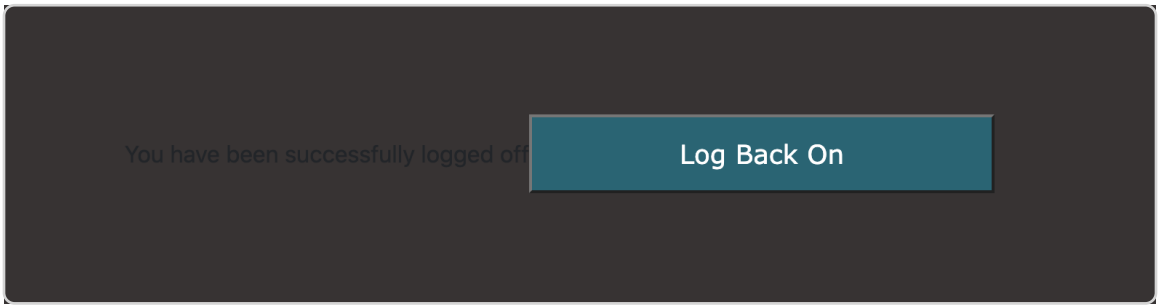
❷ Render `Logoff` Child Component.

We can now delete these imports:

```
import CSS (color, white)
import Halogen.HTML.CSS as HC
```

Now we can test. Build the code, refresh the Browser, log on and then click the Logoff link in the Page Header in the upper right-hand corner.

When I do this I get:



The layout is a bit messed up. Let's fix that.

The first thing to change is the `flexDirection` from `row` to `column`. We'll also need to add some padding between the message and the button. And the message text should be in `white`:

```
import CSS.Flexbox (flexDirection, column, justifyContent, alignItems) ❶
import CSS.Geometry (width, height, paddingTop, paddingBottom) ❷

  render :: State -> H.ComponentHTML Action Slots m
  render { loggedOff, errorMessage } =
    HH.div
      [
        HC.style do
          display flex
          flexDirection column ❸
          alignItems center
          justifyContent center
          paddingTop $ vw 0.65
      ]
      if loggedOff then
        [ HH.span ❹
          [
            HC.style do
              color white ❺
              paddingBottom (rem 1.0) ❻
          ]
          [ HH.text "You have been successfully logged off" ] ❼
        , HH.button
          [
            HC.style do
              backgroundColor themeColor
              themeFont
              fontWeight $ FontWeight $ value "500"
              fontSize $ vw 1.0
              width (rem 20.0)
              height $ vw 3.0
              color white
              cursor pointer
          , HE.onClick $ const Click
          ]
          [ HH.text "Log Back On" ]
        ]
      else [] <>
        [ (errorMessage # maybe (HH.text "") \message ->
          HH.slot _modal unit
            (Modal.component ModalCommon.errorConfig Message.component)
            message Modal)
        ]
```

❶ Changed `row` to `column`.

❷ Added `paddingBottom` to import list.

❸ Changed `row` to `column` here too.

❹ Put message in a `span` so we can style it.

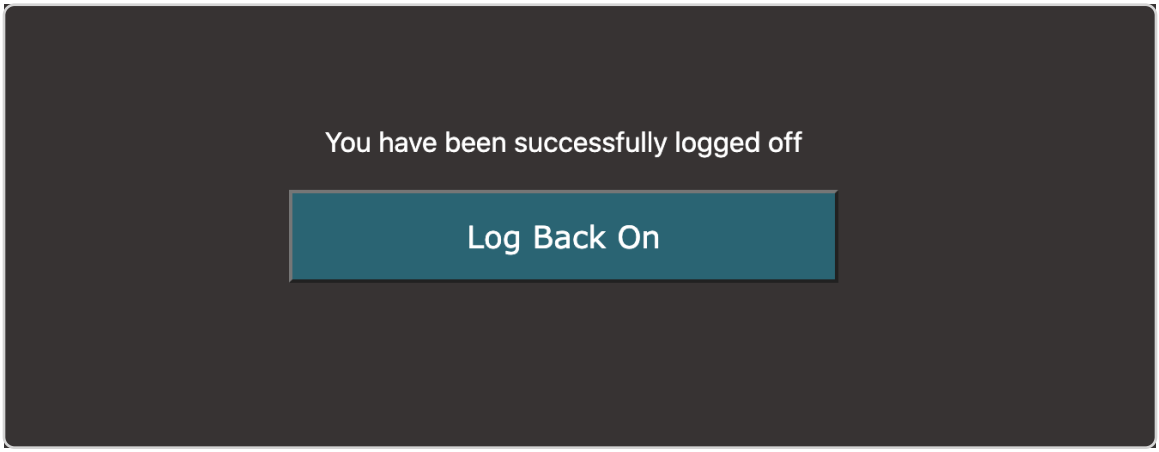
❺ Make text readable on our dark background.

❻ Add some padding between it and the button.

❼ Don't miss the added `]` here.

Now, build the Application and try to log off again.

When I do this I get:



Now, click on the `Log Back On` button. What happens?

Nothing happens. Why?

Something isn't working in our routing logic. The URL in the Address Bar changed to `http://localhost:3000/#/logon` but the page isn't getting rerendered.

Let's look at that logic in `Component.Router`:

```
-- ABOVE CODE REDACTED
  handleQuery = case _ of
    Navigate route a -> do
      { userRef } <- ask
      ref <- H.liftEffect $ Ref.read userRef
      if isNothing ref then navigate Logon ❶
      else H.modify_ _ { route = route }
      pure (Just a)
```

❶ If there's no logged on user then we navigate to `Logon`.

Can you see what went wrong?

I cannot see what's wrong at this point. So, let's think through the issues.

If there's no logged on user, we navigate to `Logon`. That's the case that should execute when we click on the `Log Back On` button.

Let's add some debugging code to `Component.Router` to see if we can figure out what's wrong:

```
import Data.Route as Route ❶
import Effect.Class.Console (log)
import Routing.Duplex as RouteDuplex

  handleQuery = case _ of
    Navigate route a -> do
      { userRef } <- ask
      ref <- H.liftEffect $ Ref.read userRef
      log $ "ref: " <> show ref ❷
      log $ "route: " <> RouteDuplex.print Route.routeCodec route ❸
      if isNothing ref then navigate Logon
      else H.modify_ _ { route = route }
      pure (Just a)
```

❶ Added these imports TEMPORARILY.

❷ Log the userRef to the console.

❸ Log the route to the console.

Add the debugging code and rebuild the Application, refresh the Browser and logon and then logoff. When you click on the `Log Back On` button, the following should be logged to the console:

```
ref: Nothing
route: /logon
```

That means that the hash in the URL is `/logon`. Since `ref` is `Nothing`, we'll call `navigate` to navigate to `Logon`, which writes `/logon` to hash. This will cause `matchesWith` to fire calling this `Navigate` handler again.

At which point, the `ref` will still be `Nothing` causing the whole process to repeat infinitely. That's not good. But wait. We're not seeing our debugging messages printed over and over ad infinitum.

Instead, it only prints once. That must mean that writing to the hash with the SAME value won't trigger `matchesWith`. Let's test that.

Assuming you haven't navigated away from our last location, run this command in the Browser's console to

see what happens:

```
window.location.hash = "/test"
```

When I do this, our debugging does not print out, which means that `matchesWith` doesn't trigger. But why?

Try this:

```
window.location.hash = "/logoff"
```

When I do this, the following prints:

```
ref: Nothing
route: /logoff
ref: Nothing
route: /logon
```

So, that tells us that `matchesWith` will not call us if our custom route parser fails. That makes sense. `/test` is an invalid `Route` for this Application, whereas `logoff` is a valid one.

At this point, our hash should be `/logon`. So, let's test our theory by writing the SAME hash value and see what happens:

```
window.location.hash = "/logon"
```

When I do this, none of our debug logging is printed. That's why it's not infinitely looping. Writing the SAME hash will NOT trigger `matchesWith`.

Okay, so now that we understand how this stuff really works, we can turn our attention back to the code at hand:

```
  handleQuery = case _ of
    Navigate route a -> do
      { userRef } <- ask
      ref <- H.liftEffect $ Ref.read userRef
      log $ "ref: " <> show ref
      log $ "route: " <> RouteDuplex.print Route.routeCodec route
      if isNothing ref then navigate Logon ❶
      else H.modify_ _ { route = route }
      pure (Just a)
```

❶ The logic in question.

When we don't have a logged on user AND we try to route to `Logon`, this logic will ignore that request and try to route us to `Logon` AGAIN. That second route will NOT trigger `matchesWith` when we're already on `Logon`.

This is our problem.

When no one is logged on, we want to ignore where we JUST navigated to and force a navigate to `Logon` EXCEPT when we just navigated to `Logon`:

```
  handleQuery = case _ of
    Navigate route a -> do
      { userRef } <- ask
      ref <- H.liftEffect $ Ref.read userRef
      if isNothing ref && route /= Logon then navigate Logon ❶
      else H.modify_ _ { route = route }
      pure (Just a)
```

❶ We will ONLY navigate to `Logon` if there's no logged on user AND we didn't just navigate to `Logon`.

I also removed the debugging code and deleted:

```
import Data.Route as Route
import Effect.Class.Console (log)
import Routing.Duplex as RouteDuplex
```

Now, rebuild the Application, refresh the Browser and retest logging on and then off and then on again using the `Log Back On` button.

When I do this, the Logon Page properly displays.

What we discovered about how hash changes work means that we don't need to do the following test anymore:

```
main :: Effect Unit
main = do
  userRef <- Ref.new Nothing
  HA.runHalogenAff do
    body <- HA.awaitBody
    let router = H.hoist (runAppM { userRef }) Router.component
    io <- runUI router unit body
    liftEffect $ matchesWith (RouteDuplex.parse routeCodec)
      \old' new -> when (old' /= Just new) $ ❶
        launchAff_ $ io.query $ H.mkTell $ Router.Navigate new
```

❶ These will ALWAYS be NOT equal.

Let's remove that:

```
main :: Effect Unit
main = do
  userRef <- Ref.new Nothing
  HA.runHalogenAff do
    body <- HA.awaitBody
    let router = H.hoist (runAppM { userRef }) Router.component
    io <- runUI router unit body
    liftEffect $ matchesWith (RouteDuplex.parse routeCodec) ❶
      \_ new -> launchAff_ $ io.query $ H.mkTell $ Router.Navigate new
```

❶ Removed `when (old' /= Just new) $`.

Congratulations. You now have completed the front-end portion of your Application meaning that you can now write a Full-stack Application in a Pure Functional Language, viz. PureScript.

Appendix A: Epilogue

You have now built a complete working Web Application in a Purely Functional Language, i.e. PureScript.

I hope that this proves to you that you can write and maintain everyday software that runs in production in a Purely Functional Language. Not that others can or it's possible in theory, but that you can do this right now. In fact, I don't have to prove it to you, since you've already proven it to yourself.

It's always good to look back at what we've accomplished. While you're going through this book, you may have felt that you were constantly not getting it or that every thing was a bit of a struggle. Well, you couldn't have finished the `fullstack` project without knowing a lot more than when you started.

Here's a list of SOME of the things you had to learn so that you could write this Application:

- Pure Functions
- Immutability
- Higher-order Functions
- Currying
- Partial Application
- Recursion
- Tail Recursion
- Pattern Matching
- Types
- Typeclasses
- Multi-parametric Typeclasses
- Overlapping Instances
- Orphaned Instances
- Instance Dependencies
- Functional Dependencies
- Abstract Algebra
    - Magma
    - Semigroup
    - Monoid
    - Group
    - Abelian Group (aka. Commutative Group)
    - Semiring

    - Ring and Commutative Ring
    - Euclidean Ring
- Folds
- Algebraic Data Type (ADT)
- Functors (Covariant, Contravariant and Invariant)
- Functors with Values and Functions
- Bifunctors
- Profunctors
- Applicative Functor
- Traversables
- Foldables
- Applicative Parsers
- Monads
- Monadic Parsers
- Monad Stacks
- Category Theory (superficially)
    - Definition
    - Hask Category
    - Functors
    - Applicative
    - Kleisli Category

And that's just the theory part. The following is a list of SOME of the practical things you've learned along the way:

- Interpreting Compiler Errors
- Type Holes
- Effects (Synchronous and Asynchronous)
- AVars and Refs
- Data Bus
- Ajax
- JSON decoding
- Foreign Function Interface (FFI)
- Hash Routing

- Static File Servers
- CORS

And you wrote 1000s of lines of code to get here.

Keep in mind that you will not retain everything here. That's to be expected and is normal. The code you've written should be the first place you look to refresh your memory and the book should be the second. There's nothing wrong with rereading sections of the book to help solidify concepts that were only superficially understood the first time.

So don't hesitate to revisit this book from time to time to see if there's some deeper understanding that you might get from it on a second or third pass.

It was a long journey of hard work and discipline that, hopefully, hasn't been too arduous. I really hope this book has helped reduce the effort that it used to take to get to this level of understanding.

I want to encourage you to continue your Functional Language learning and take a good hard look at Haskell. All of the general concepts we've learned here will apply.

You should feel pretty accomplished for having put in the time and effort to learn everything in this book. Your programming life will certainly be better for it. I'm glad that I could help you embark on this journey and wish you the best of luck in your future endeavors.

Take care.