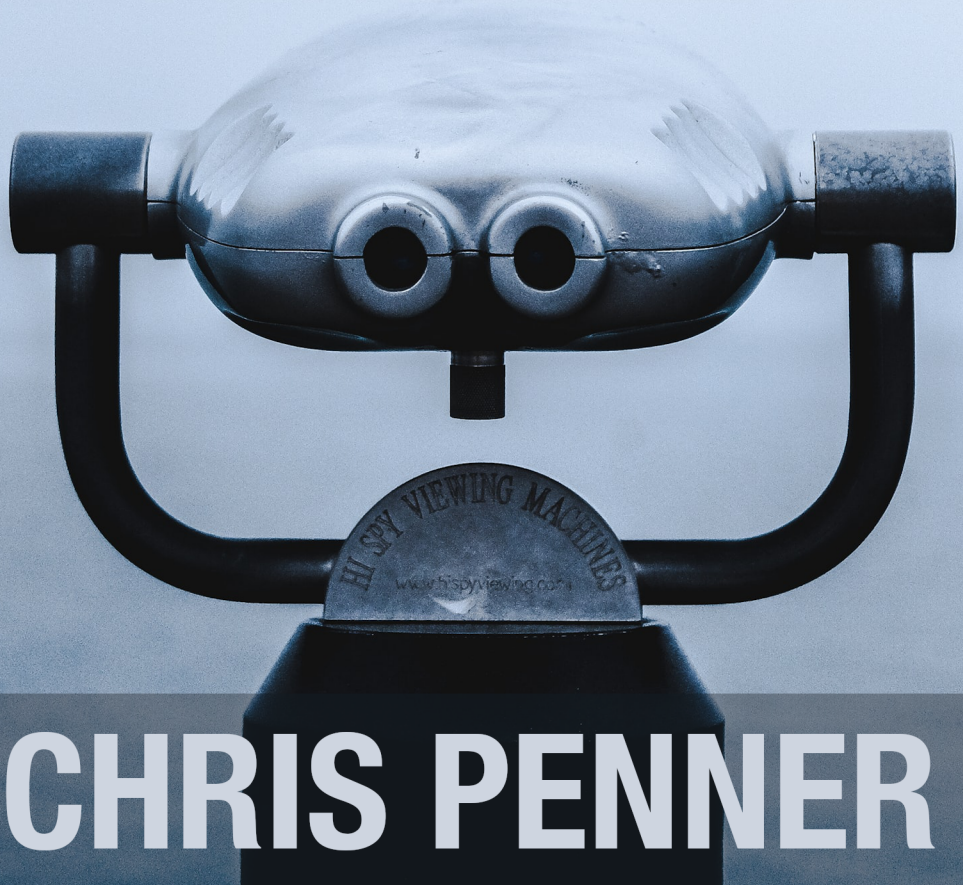
OPTICS
BY EXAMPLE
FUNCTIONAL LENSES IN HASKELL



CHRIS PENNER

Optics By Example

Functional lenses in Haskell

Chris Penner

This book is for sale at http://leanpub.com/optics-by-example

This version was published on 2020-01-01

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

Tweet This Book!

Please help Chris Penner by spreading the word about this book on Twitter!

The suggested tweet for this book is:

Interested in lenses or optics? You seriously need to check out @OpticsByExample by @chrislpenner! #OpticsByExample

The suggested hashtag for this book is #OpticsByExample.

Find out what other people are saying about the book by clicking on this link to search for this hashtag on Twitter:

#OpticsByExample

Contents

1. Obligatory Preamble

1.1 About this book

Welcome to **Optics By Example**! This book is still a **work in progress**, which means that chapters may exist in **unfinished** or **unpolished** states. Please bear with me while I work to make this book the best it can be. That said, if you notice something you think could be improved in any way please let me know[1].

I'll be writing the book iteratively; which means that I'll often loop back and improve previous chapters as I go along. This way I can provide the greatest breadth of content for my readers while getting a good feel for the structure of the book and which areas may need more or less time than others. I'll announce progress and weekly changes on the early-access Patreon page[2].

1.2 Why should I read this book?

It's every author's burden to convince their readers it's worth spending precious time to read their book (which is likely altogether too long). Especially when it has far too few illustrations and they'd much rather be in the park having a nice picnic (counterpoint: consider bringing the book with you on the picnic).

So as not to lose you before we get to the good stuff; I'll do my best to spark your interest in optics!

As I've learned about optics I've continually felt a sense of child-like wonder. It's like learning to program again, except this time the abstractions are more powerful, more concise, and more composable! Nearly every data-related programming problem I've been faced with in the last year has had an elegant optics-based solution which typically takes less than a single line of code!

While learning to use optics for the first time you'll likely feel like a wizard embarking in their first few weeks of spell-casting lessons. At first, things will be confusing and complicated. Your mental model will be entirely incorrect; and most incantations use words you don't know, but over time, you'll master your first few spells and begin to see this "magic" differently. You'll understand the principles behind this magic and things will start to make a bit of sense. That's when you'll realize the powers you've unlocked and will yearn to learn even more arcane arts to further expand your abilities.

I hope you'll allow me to be the eccentric old wizard mentor in your journey, and that you'll be as enamored and enticed with this new power as I once was.

[1] https://forms.gle/fNj1c7Rfgm3Rk5718
[2] https://patreon.com/chrispenner

1.3 How to read this book

The goal of this book is to get you, the reader, up-to-speed on optics quickly, but with a solid understanding of how things work behind the scenes as well. For this reason the book is meant to be read mostly in a linear fashion. It introduces concepts in-order and each topic is usually a pre-requisite for the next. If you're already an optics aficionado then it may be reasonable to skip ahead, but I'd recommend at least skimming any chapters you skip (or perhaps do the exercises).

This book has exercises after introducing each topic. I recommend you do them, and check your answers with those in the back of the book. I find exercises to be the best way to ensure I'm **actually** learning a topic and not just pretending. If you can complete the exercises at the end of a section you're ready to move on to the next one!

1.4 Chosen language and optics encodings

There are many different optics libraries available in many different programming languages. Unfortunately I don't have time to write a book for each of them, so I've picked a specific implementation for consistency (and sanity) sake.

This book's examples and types are written in **Haskell** using the `lens` library[3]. This is the de-facto choice for using optics in Haskell at the time of writing this book.

For those interested in some of the different ways that optics can be implemented (called *encodings*) we'll glance at a few of them as we go along, however it's typically not necessary to know much about the underlying library implementations in order to use optics (even for advanced applications) so it won't be a primary focus in this book.

I'm sorry if this isn't your favourite combination of language, library, and encoding, but I strongly believe that the vast majority of the book will still be very useful to you, whatever your circumstances may be. You'll likely need to mentally "translate" the types, syntax, and possibly a few combinator names to your library of choice, but most optics libraries are similar enough that this shouldn't be any great burden.

If you have no idea what I'm talking about in this section and are just keen to learn some optics, then don't worry about it a bit, just proceed to the next section.

1.5 Your practice environment

As I stated in the previous section, I'll be using the Haskell optics library simply called `lens`[4] for all the examples to follow. Working with optics, and especially learning to use them for the first time,

[3] http://hackage.haskell.org/package/lens
[4] http://hackage.haskell.org/package/lens

requires a lot of experimentation, guesswork, and failure. It's certainly worthwhile to take the time to set up a practice environment where you can get quick feedback!

Here's my recommended setup, however if you're using a different language, or are confident in an alternative setup, then feel free to move on.

First, install the **stack** build tool[5]

Then, create a new project in a directory of your choice:

```
$ stack new optics-exercises && cd optics-exercises
```

Install GHC, This will likely take some time, feel free to go grab a snack:

```
$ stack setup
```

Add `lens` as a dependency to your `package.yaml` file. You may also want to include the following extra dependencies as they'll be used in certain exercises as well:

Your `package.yaml`'s dependencies section should look *roughly* like this:

```
dependencies:
- base >= 4.7 && < 5
- aeson
- containers
- lens
- lens-aeson
- mtl
- text
```

Kick off a new build of the project (then go get another snack):

```
$ stack build
```

Now open up a new repl session:

```
$ stack repl
```

Try to import the `lens` library using the following, if it doesn't report any errors, then you're all set!

[5] https://seanhess.github.io/2015/08/04/practical-haskell-getting-started.html

```
>>> import Control.Lens
```

1.6 Following examples

There'll be a lot of examples coming up; this is **Optics By Example** after all!

You can assume from this point on that all examples have added at least the following imports and pragmas; and if GHC suggests an import or pragma to you then usually you should just follow its suggestions.

```
{-# LANGUAGE TemplateHaskell #-}
{-# LANGUAGE FlexibleInstances #-}
{-# LANGUAGE FlexibleContexts #-}
{-# LANGUAGE RankNTypes #-}
{-# LANGUAGE ScopedTypeVariables #-}
{-# LANGUAGE TypeApplications #-}
{-# LANGUAGE TypeFamilies #-}
{-# LANGUAGE InstanceSigs #-}

import Control.Lens
import Control.Applicative
import Data.Char
import qualified Data.Map as M
import qualified Data.Set as S
import qualified Data.Text as T
```

1.7 About the type signatures

Wherein the author attempts to avoid receiving angry emails

Note dear reader that for the purposes of pedagogy I'll often specify **simplified** type signatures for combinators and optics as they're introduced. It's better to understand the simple concepts first, then upgrade that understanding as familiarity and confidence grows. This may be particular jarring for folks who are already very familiar with optics, I'll do my best to introduce more general type signatures as we build up the background required to understand them.

1.8 About the exercises

I think **doing** is a critical part of learning. Each section provides a few exercises to test your understanding. I strongly recommend at least trying each of the exercises you come across. Answers are provided in the back of the book for reference.

With all that said; I sincerely hope you enjoy the book. I'd love to hear how the book is treating you and about any feedback you may have for me. Please send me feedback![6].

Now, onwards to optics!

[6] https://forms.gle/fNj1c7Rfgm3Rk5718

2. Optics

2.1 What are optics?

Optics in its most general sense is a full field of study! In a slightly more concrete sense, **optics** are a **family** of **tools** which are interoperable with one another. **Lenses**, **Folds**, **Traversals**, **Prisms** and **Isos** are all **types of optics** which we'll explore throughout the book! This isn't a comprehensive list of all optics, in fact new types are still being discovered all the time!

You'll gain an intuition for what the more general concept of an **optic** actually *is* as you learn about each concrete type and begin to understand what they have in common, but to put it in a nutshell: optics are a family of inter-composable combinators for building bidirectional data transformations.

2.2 Strengths

So why do we actually care about these bidirectional transformation things? The short answer is that they solve a *lot* of very common data-manipulation problems in a **composable**, **performant**, and **concise** way, the long answer is what follows from here to the end of the book!

I'll take just a moment to expand on a few strengths:

Composition
: Each optic *focuses* on some subset of data, when composing with other optics they can pick up from where the previous optic left off and diving down even further. This means that each optic you learn becomes a member of your growing **vocabulary** of optics. Just as words in natural language can be strung together to form a sentence which communicates any intent you might want, a sufficiently complete vocabulary of optics can be arranged to effectively manipulate data to achieve your goals

Separation of concerns
: Optics are the abstraction most programmers didn't know they needed. They allow us to cleanly separate concerns in stronger ways than *either* of Object-Oriented or Functional-Programming styles allow on their own. Optics allow us to specify which portions of data we wish to work with **separately** from the operations we wish to perform on them. We could, for example, encode a pre-order traversal of a tree-structure, and combine it with a behaviour which prints the elements. We can swap out either of the **data-selector** or the **action** without affecting the other. Say goodbye to the Visitor Pattern!

Concision
: Although there are *many* other good reasons to love optics, they have the nice property of

being *very* succinct. **Most** tasks can be expressed in a single line of code, and in many cases the resulting code even reads like a simple sentence. For instance: `sumOf (key "transactions" . values . key "cost" . _Number)` will accept a JSON blob, will look into the "transactions" key, will then dive into each of the elements of the array, and in each of those objects will collect the "cost" of the transaction as a **number**, summing them all up into a total. In a typical imperative language this operation would likely take a few lines of code, or would at the very least require some ugly nested brackets. If we wish to instead take the **average** of these numbers we need only swap the `sumOf` action without fussing about with any variables and loops.

Enforcing interface boundaries

Optics can serve as an external interface which remains consistent despite changes to your data layer. They provide an abstraction layer similar to getters & setters which one might write in Java or Python. This allows you to alter your underlying data structures without breaking external consumers. You can even enforce data-consistency invariants when both getting and setting values! This replaces the idea of class-based "getters" and "setters" while also covering significant ground which used to require interfaces.

A principled and mature ecosystem

Optics have been around for long enough now that the ecosystem has ironed out most bugs and performance issues. There are a wide variety of libraries available, and many popular libraries provide optics-based interfaces (e.g. there are optics wrappers around JSON and XML libraries). Optics have a simple universal construction with the surprising benefit that writing an optics combinator **does not** require a dependency on any optics libraries! This allows us to use optics as a primitive building block or interface across libraries without worrying about large transitive or cyclic dependencies.

Hopefully all of this is sounding "too good to be true"! I assure you that optics can deliver on these promises. I can also guarantee that it'll take a little work and more than a few terrible, horrible, no good, very bad type errors to get there, but we're all in this together!

2.3 Weaknesses

Can't always have your cake and eat it too; here are a few areas where optics aren't perfect **yet**:

Type-Errors

Most optics libraries (especially `lens`) can spew out some pretty ugly type errors when something goes wrong. This is one of the **most common** complaints, however it's a problem which is not easily solved. A great deal of type-level complexity is required to keep these libraries polymorphic and performant! We'll approach new types carefully and will address some common mistakes as well as talking about how to read these terrible beasts, so hopefully we can mitigate this one slightly.

Complex Implementation

Most (all?) optics implementations have their fair share of magic (see: complex category theory

and/or dirty hacks) going on behind the scenes. To make matters worse, most libraries are implemented in completely different ways! The Scala implementation is different from the Haskell implementation which is different from the Purescript implementation! They all follow a lot of the same theories and encode the same concepts, but a *perfect* backing implementation hasn't been discovered yet, though the profunctor encoding is looking pretty good so far. Luckily most libraries provide helpers which abstract over the underlying implementation so you won't typically need to worry about it.

Vast collection of combinators

This is one of those "weaknesses" you put on your CV that's actually a strength in disguise. There are a **LOT** of helpers and combinators provided in most optics libraries, it's overwhelming at first, but you'll learn how to search through them and better find the ones you need; and when you can do that effectively it means you'll usually be able to find a helper for performing almost any optics task! Just have a little patience, and finish reading this book of course!

2.4 Practical optics at a glance

I've talked at a high level about how optics help you perform actions over portions of data. Simple **actions** you can perform involve variants of **view**ing, **modify**ing or **traversing** the selected data.

Here are a few examples of varying difficulty and usefulness which represent a few things we'll see:

```
-- View nested fields of some record type
>>> view (address . country) person
"Canada"

-- Update portions of immutable data structures
>>> set _3 False ('a', 'b', 'c')
('a', 'b', False)

-- These selectors compose!
-- We can perform a task over deeply nested subsets of data.
-- Let's sum all numbers which in a 'Left' within the right half of each tuple
>>> sumOf (folded . _2 . _Left)
      [(True, Left 10), (False, Right "pepporoni"), (True, Left 20)]
30

-- Truncate any stories longer than 10 characters, leaving shorter ones alone.
>>> let stories =
      ["This one time at band camp", "Nuff said.", "This is a short story"]
>>> over
      (traversed . filtered ((>10) . length))
      (\story -> take 10 story ++ "...")
```

```
      stories
["This one t...","Nuff said.","This is a ..."]
```

2.5 Impractical optics at a glance

Here are a few of the more archaic and *interesting* things optics can do. It's not important that you understand how these work or what they're doing, they're just here to help demonstrate the sheer **adaptability** of optics. Note how each operation is only one line of code! Hopefully they spark a bit of curiosity!

```
-- Summarize a list of numbers, subtracting the 'Left's, adding the 'Right's!
>>> sumOf (folded . beside negated id) [Left 1, Right 10, Left 2, Right 20]
27

-- Capitalize each word in a sentence
>>> "why is a raven like a writing desk" & worded . _head %~ toUpper
"Why Is A Raven Like A Writing Desk"

-- Multiply every Integer by 100 no matter where they are in the structure:
>>> (Just 3, Left ("hello", [13, 15, 17])) & biplate *~ 100
(Just 300,Left ("hello",[1300,1500,1700]))

-- Reverse the ordering of all even numbers in a sequence.
-- We leave the odd numbers alone!
>>> [1, 2, 3, 4, 5, 6, 7, 8] & partsOf (traversed . filtered even) %~ reverse
[1,8,3,6,5,4,7,2]

-- Sort all the characters in all strings, across word boundaries!
>>> ("one", "two", "three") & partsOf (each . traversed) %~ sort
("eee","hno","orttw")

-- Flip the 2nd bit of each number to a 0
>>> [1, 2, 3, 4] & traversed . bitAt 1 %~ not
[3,0,1,6]

-- Prompt the user with each question in a tuple,
-- then return the tuple with each prompt replaced with the user's input,
>>> let prompts = ( "What is your name?"
                  , "What is your quest?"
                  , "What is your favourite color?"
                  )
>>> prompts & each  %%~ (\prompt -> putStrLn prompt >> getLine)
```

```
What is your name?
> Sir Galahad
What is your quest?
> To seek the holy grail
What is your favourite color?
> Blue I think?
("Sir Galahad","To seek the holy grail","Blue I think?")
```

I hope that was a sufficiently strange list of examples to spark some wonder and creativity. These were meant to show the versatility, expressivity, and concision of optics! These examples are contrived and complex of course, but we'll see some more practical examples as we go on.

3. Lenses

3.1 Introduction to Lenses

We'll start our journey with lenses!

I mentioned in the **optics** section that optics allow us to separate concerns; i.e. split up the **action** we perform on data from the **selection** of data we want to perform it on. To be clear I'll refer to operations which can be performed on data as **actions**, whereas the data selectors are the actual **optics**. Each type of **optic** comes with a set of compatible **actions**.

Each type of optic has a different balance **constraint** vs **flexibility**, moving to and fro on this spectrum results in several different but useful behaviours. Lenses lean closer to the **constrained** side of things, which means you have a lot of **guarantees** about their behaviour, but also means that you need to prove those guarantees to make a lens, so there are fewer **lenses** in the world than there are of the more flexible optics.

Lenses have the following concrete guarantees:

- A Lens **focuses** (i.e. **selects**) a **single** piece of data within a larger **structure**.
- A Lens must **never fail** to **get** or **modify** that focus.

These constraints unlock a few **actions** we can perform on lenses:

- We can use a lens to **view** the **focus** within a structure.
- We can use a lens to **set** the **focus** within a structure.
- We can use a lens to **modify** the **focus** within a structure.

Before we talk too much at a high level, let's see take a look at a concrete usage of a lens and understand the different parts.

Anatomy

Here's a simple snippet which gets the `String` "hello" out from a couple nested tuples:

```
>>> view (_2 . _1) (42, ("hello", False))
"hello"
```

We won't worry about *exactly* what it's doing or how it works yet; for now we'll pick out and name individual pieces of this construction so that I can save myself some typing for the rest of the book.

Let's break it down into its anatomy:

```
      +-> The Action          +-> The Structure
      |                         |
    /-+-\           /----------+----------\
>>> view  (_2 . _1) (42, ("hello", False)))
          \---+---/       \--+--/
              |              |
              |              +-> The Focus
              +-> The Path
```

Note that the names for these things aren't really standardized yet, so you may have poor luck searching for them on Google; but if you start using them confidently around the water cooler I'm sure they'll catch on eventually.

The Action™
: An **action** executes some **operation** over the **focus** of a **path**. E.g. `view` is an action which **gets** the **focus** of a **path** from a **structure**. Actions are often written as an **infix operator**; e.g. `%~`, `^.` or even `<<%@=`!

The Path™
: The **path** indicates which data to **focus** and where to find it within the **structure**. A path can be a single optic, or several optics chained together through *composition*. If you consider dot-notation from most Object-Oriented languages you'll see similarities.

The Structure™
: The **structure** is the hunk of data that we want to work with. The **path** selects data from within the **structure**, and that data will be passed to the **action**.

The Focus™
: The smaller piece of the **structure** indicated by the **path**. The **focus** will be passed to the **action**. E.g. we may want to *get*, *set*, or *modify* the focus.

Exercises – Optic Anatomy

Jump to answers

For each of the following, identify the **action**, **path** and **structure**, don't worry about understanding how they actually work just yet. If you want a real challenge, try to identify the **focus** too! Note that certain optics allow **multiple focuses**, and some actions accept **parameters** other than the **focus**.

```
>>> view (_1 . _2)  ((1, 2), 3)
2

>>> set (_2 . _Left) "new" (False, Left "old")
(False, Left "new")

>>> over (taking 2 worded . traversed) toUpper "testing one two three"
"TESTING ONE two three"

>>> foldOf (both . each) (["super", "cali"],["fragilistic", "expialidocious"])
"supercalifragilisticexpialidocious"
```

3.2 Lens actions

Now that we've got a bit of an optics vocabulary we can start to learn how it all works.

Viewing through lenses

In this section we'll introduce your very first **lens** and **action**!

Here it is, the `_1` lens:

```
_1 :: Lens' (a, b) a
```

Ironic that our first lens is literally the number `_1` isn't it? The `_1` lens is one provided to us by the `lens` library. This lens **focuses** the **first** *slot* of a tuple, allowing us to **get** or **set** that "slot" of the tuple without affecting the other.

This the first time seeing a lens type, so let's break it down a little. The `Lens'` type has two parameters, the **first** indicates the type of the **structure**, the **second** indicates the type of the **focus** within it. In this case we can see that the **structure** is the full tuple and the **focus** is the type in its first slot.

There are also appropriate lenses for other slots of tuples, auspiciously named `_2`, `_3`, `_4`, etc. I've simplified the type here to a 2-tuple so it's easier to understand, but `_1` actually works on any size of tuple within reason.

A lens on its own isn't enough! We need an **action** to perform on it!

`view :: Lens' s a -> s -> a`
: **Get** the **path**'s **focus** from within the **structure**.

You'll notice that unless we have particular types in mind we typically denote the **structure** with an `<s>` and the **focus** with an `<a>`; this is just an idiom that helps us keep things straight.

Now we can put our optics anatomy knowledge to the test and try a few terms to see what happens:

```
>>> view _1 ('a', 'b')
'a'

>>> view _2 ('a', 'b')
'b'
```

When we use the **view** action on the `_1` lens it extracts the first element of the tuple! `(view _1)` and `(view _2)` behave like `fst` and `snd` respectively.

Setting through a lens

Let's learn two more **actions** which work well with **lenses**.

`set :: Lens' s a -> a -> s -> s`
: **Set** a new value at the **path**'s **focus**, leaving the rest of the **structure** unaltered.

`over :: Lens' s a -> (a -> a) -> s -> s`
: Modify the **focus** of a **path** by running a function **over** it, altering it *in-place* and leaving the rest of the **structure** unaltered.

Let's try them out!

```
>>> set _1 'x' ('a', 'b')
('x','b')

>>> over _1 (*100) (1, 2)
(100, 2)
```

Here we can see that using `set` with `_1` replaces the first tuple slot entirely, and `over` runs its multiplication function on the **focus** in-place.

Notice that `view` returns the **focus** type, but `set` and `over` return an altered version of the entire **structure**.

Exercises - Lens Actions

Jump to answers

1. Find the **structure** and the **focus** in the following lens:

```
Lens' (Bool, (Int, String)) Int
```

2. Write the type signature of a Lens with the **structure** `(Char, Int)` and the **focus** `Char`.
3. Name 3 **actions** we can use on a **Lens**.
4. Which lens could I use to **focus** the character 'c' in the following structure:

```
('a', 'b', 'c')
```

5. Write out the (simplified) types of each identifier in the following statement.

```
>>> over _2 (*10) (False, 2)
```

What is the result of running it?

3.3 Lenses and records

This chapter deals a lot with Haskell records; if you're using lenses in a language other than Haskell there's likely some direct parallel (objects, classes, structs, etc.). Try to draw those parallels wherever you can, don't skip this chapter, it introduces many foundational lens concepts.

Records are a fundamental part of doing real work in Haskell; however working with records in Haskell has always had a few warts, mostly concerning the built-in mechanism for updating record values. Did you know that optics themselves were actually discovered to ease the pain of getting and setting fields of deeply nested record fields when doing games programming[1]?

Lenses help to ease this pain as we deal with real-world complicated and nested state.

Lenses subsume the "accessor" pattern

Most Object-Oriented languages like Python or Java have some form of the "Accessor Pattern". The basic idea is to define **getter** and **setter** methods for properties on your object as a form of encapsulation and protection. Lenses allow us to capture the essence of this pattern in a "functional-programming" style.

Field lenses are both the **getter** and **setter** for a field bundled together into a single **value**! Notice how I said **value**? In functional programming **functions are values**, so this means we can freely pass around our "Accessor Pattern" as a value, and can even have different accessors for the same fields if we want!

[1] https://github.com/ekmett/lens/wiki/History-of-Lenses

Building a lens for a record field

We're going to need a record type before going much further; so let's define one! Remember to add all the imports and pragmas specified in the introduction if you're following along.

```
data Ship =
    Ship { _name    :: String
         , _numCrew :: Int
         }
    deriving (Show)
```

This defines a `Ship` record type with two simple fields, one for the ship's name, and the other for the number of crew on board.

Notice how we named our fields using leading underscores? That's no accident! Because lenses provide a more convenient and powerful interface than 'stock' Haskell record fields it's common to give preference to the lensy interface. For this reason, when writing a record definition typically record fields will be named with a leading underscore. For example using `_fieldName` in the record definition leaves `fieldName` available for the corresponding lens.

We're going to build lenses so we can **get**, **set**, and **modify** these fields.

So how do we actually create a lens? As it turns out, the `lens` library provides a lens-builder helper function that makes it pretty easy! It's a function called (at risk of sounding like a broken record): `lens`!

```
lens :: (s -> a) -> (s -> b -> t) -> Lens s t a b
```

Whoah; that's a lot of type variables, that looks complicated! This function can build fully **polymorphic** lenses, we'll explore what that means later on, but for record fields we don't require that level of flexibility (yet), so we'll stick with the simpler optics types.

Here's the same function with the simpler (but more restrictive) type signature, I've also labeled the important parts:

```
           +-> Getter Function
           |                              +-> Structure type
       |---+--|                           |
lens :: (s -> a) -> (s -> a -> s) -> Lens' s a
                    |-----+-----|            |
                          |                  +-> Focus type
                          +-> Setter Function
```

Note the `'` at the end of `Lens'`, it's not a typo! When you see a type in `lens` suffixed with a tick it means it's a **simple** optic, as opposed to a **polymorphic** optic which require more type parameters. We'll need polymorphic optics later on, but for now we'll stick with those of **simple** ilk.

The type signature has only two type variables: `<s>` and `<a>`. These are the idiomatic names of type variables for the **structure** and **focus** types of a lens. Whenever you see `<s>` in a lens type think "**structure**", and whenever you see an `<a>` think "**focus**".

Let's use the helper to build a lens for the `_numCrew` field. The helper needs **two** things to make a lens for us; a **getter** function and a **setter** function.

Let's start with the **getter**, which has the following type:

```
(s -> a)
```

We're getting the number of crew members from inside the `Ship`; so the `Ship` is our **structure** and an `Int` is our **focus**. That means that we can substitute `s` ~ `Ship` and `a` ~ `Int` in the above signature and end up with:

```
(Ship -> Int)
```

That's easy enough to implement:

```
getNumCrew :: Ship -> Int
getNumCrew = _numCrew
```

Well that was easy! We probably don't need to actually define this as a whole new function, but I've gotta fill up this book somehow right?

Even though we're building lenses for our record fields the default Haskell record accessor functions still exist, remember that we named with the leading _ prefix, which is a common idiom when defining lensy field accessors

`getNumCrew` matches the **getter** signature perfectly! Next we need a **setter**!

The setter has the type:

```
(s -> a -> s)
```

Using our optics vocabulary we see that this function takes a **structure** and a **focus** and returns a *new* **structure**. Let's specialize this one to our situation too:

```
(Ship -> Int -> Ship)
```

Okay great! So now we just have to write a function like this; we can use Haskell record-update syntax for that:

```
setNumCrew :: Ship -> Int -> Ship
setNumCrew ship newNumCrew =
    ship{_numCrew = newNumCrew}
```

If you haven't seen this syntax before it can be a bit weird. We specify fields we want to update and their new values inside the curly braces.

Now that we've got all our pieces we can build the lens itself! The `lens` helper function snaps the **getter** and **setter** pieces together like puzzle pieces, resulting in a lens!

```
numCrew :: Lens' Ship Int
numCrew = lens getNumCrew setNumCrew
```

Now that we have one of these cool "lens" things we can start to use it, but first let's try some quick exercise problems.

Exercises - Records Part One

Jump to answers

1. The **structure** and **focus** of a lens are typically represented by which letters in type signatures?
2. Which **two** components are required to create a lens?
3. Implement the following lens:

```
name :: Lens' Ship String
```

Getting and setting with a field lens

Congratulations on writing your first lens! Let's take it for a spin.

Now that we've seen how the sausage gets made we can understand how our lens actions work a little better. To make a lens we bind a getter and setter together into a single value; when we use the lens actions we're essentially just taking one of those pieces back out.

For instance, you can imagine that `view` just extracts the **getter** function from a lens:

```
view :: Lens' s a -> (s -> a)
```

Let's confirm that it works as we expect:

```
purplePearl :: Ship
purplePearl =
    Ship { _name    = "Purple Pearl"
         , _numCrew = 38
         }
```

```
>>> view numCrew purplePearl
38
```

It works the same as using the field accessor:

```
>>> _numCrew purplePearl
38
```

The **set** action simply extracts the **setter** from the lens:

```
set :: Lens' s a -> a -> s -> s
```

Let's try it:

```
>>> purplePearl
Ship
    { _name = "Purple Pearl"
    , _numCrew = 38
    }

>>> set numCrew 41 purplePearl
Ship
    { _name = "Purple Pearl"
    , _numCrew = 41
    }
```

This behaves the same as if we used the `setNumCrew` function we used to build the lens in the first place.

Note that we still have to set `numCrew` to an `Int`; anything else wouldn't match the type of the `_numCrew` field in the `Ship` record type.

Modifying fields with a lens

Now that we can both **get** and **set** fields in records using a `lens`, we can also look at how **modifying** fields works!

There's really not a lot to it, a **modification** is equivalent to **viewing** the existing value, running some function on it, then **setting** the result back into the same field.

This is essentially what the `over` action does, we don't need to provide a `modifier` function when building a lens because it can be implemented using the getter and setter.

```
>>> over numCrew (+3) purplePearl
Ship
    { _name = "Purple Pearl"
    , _numCrew = 41
    }
```

Using `over` instead of using `view` and then `set` allows us to express what we want to do more clearly and concisely, but here's how we could do the same thing without it:

```
>>> set numCrew (view numCrew purplePearl + 3) purplePearl
Ship {_name = "Purple Pearl", _numCrew = 41}
```

Automatically generating field lenses

Writing our field accessors manually taught us about the relationship between **lenses**, **getters**, and **setters**, but writing them by hand is **mechanical**, **boring**, and **error-prone**! Did I hear someone yell boilerplate from the back!? There's only one correct way to get or set a record field, let's let the computer figure it out for us.

We can use Template Haskell to write our lenses for us! It's basically a macro system for generating Haskell code. To use it we'll need to enable the GHC extension by adding the following pragma to the top of our Haskell module:

```
{-# LANGUAGE TemplateHaskell #-}
```

With that enabled, we can add the appropriate Template Haskell expression right **after** our data declaration:

```
data Ship =
    Ship { _name    :: String
         , _numCrew :: Int
         }
    deriving (Show)

makeLenses ''Ship
```

The double-ticks `''` aren't a typo, they're how we pass an identifier name to the `makeLenses` macro!

This will generate the appropriate lens for each field in our `Ship` record type. By default `makeLenses` chooses names for the lenses by stripping the leading underscore _ from the field name. It'll generate the exact same lens we wrote by hand and will even have the same name! You'll need to delete, move, or rename your lens for the `numCrew` field if you still have that sitting around.

Note that it **won't** generate lenses for fields that aren't named with underscores, so don't forget to add it!

In general `makeLenses` does "The Right Thing"™, so I recommend taking advantage of it whenever you can. Make sure you do try writing a few lenses by hand though, it's a very good exercise when you're learning.

Template Haskell runs at compile-time, so the lenses produced by `makeLenses` won't ever show up in your source code. This can be a bit confusing at first, so if you're having trouble tracking down where a lens is coming from, it's very possible it's being generated by Template Haskell somewhere, make sure to take a look for a `makeLenses` call!

`makeLenses`

If you want more control over the names and fields that `makeLenses` generates with, you can provide additional rules using `makeLensesFor` and `makeLensesWith`.

Note that the **position** of Template Haskell expressions within a file **matters**. You can imagine that each Template Haskell expression splits the file into two modules, whatever is ABOVE the expression and whatever is BELOW. This means you may need to re-organize things in your module a bit, but so long as you don't need the generated lenses in the module itself it's usually safe to put all the calls to `makeLenses` together at the bottom of your data declarations, but above any uses of the lenses it generates.

You can check that everything is working properly by importing your module in `ghci` and checking to see if the lenses exist:

```
>>> :t name
name :: Functor f => (String -> f String) -> Ship -> f Ship

>>> :t numCrew
numCrew :: Functor f => (Int -> f Int) -> Ship -> f Ship
```

We see some type signatures printed to screen, and they're nigh incomprehensible so they must be optics right?

If this is your first exposure to these sorts of types this may be a bit of a surprise. Unfortunately, the *actual* types of lenses are pretty messy like this. Don't ignore them or get scared of them though, you can read them with a bit of practice! I'll teach you how to unpack and read these types a bit later on!

Here are a few heuristics to help you guess what's going on in optics types:

- If you see `Functor f` it's *probably* a **lens**.
- The **focus** of the optic appears first (and second) in the signature, the **structure** appears last (and second-last). In the following signature:

```
Functor f => (Int -> f Int) -> Ship -> f Ship
```

`Int` is the **focus**, `Ship` is the **structure**.

We'll learn more about this later.

Now that we've learned all of that, here's a cheat-code: If at any time you want to see everything exported by a module, you can see all its exports using the `:browse` command!

```
>>> :browse MyLenses
data Ship = Ship {_name :: String, _numCrew :: Int}
purplePearl :: Ship
name :: Lens' Ship String
numCrew :: Lens' Ship Int
getNumCrew :: Ship -> Int
setNumCrew :: Ship -> Int -> Ship
```

You'll also notice that this prints out the "nice" names for our lenses! Note the lens types of `name` and `numCrew`

Field lenses generated by `makeLenses` *usually* have the type `Lens' recordType fieldType`. There are exceptions involved when we start dealing with type variables and types with multiple constructors, but this rule should help get you started.

Exercises - Records Part Two

Jump to answers

1. List the lenses which would be generated by the following `makeLenses` call, including their types.

```
data Inventory =
    Inventory { _wand    :: Wand
              , _book    :: Book
              , _potions :: [Potion]
              }

makeLenses ''Inventory
```

2. Using the heuristics in this chapter, rewrite the following type as a `Lens'`:

```
gazork :: Functor f => (Spuzz -> f Spuzz) -> Chumble -> f Chumble

-- Fill in the blanks:
gazork :: Lens' _ _
```

3. The following code won't compile! Can you see what's wrong and fix the problem?

```
data Pet = Pet
    { _petName :: String
    , _petType :: String
    }

getPetName :: Pet -> String
getPetName pet = view petName pet

makeLenses ''Pet
```

The error says:

```
• Variable not in scope: petName :: Lens' Pet String
```

3.4 Limitations

So far we've only built a few simple lenses, but hopefully your mind has started churning out ideas of other cool lenses you could write! Sorry to be a buzz kill, but there are actually a few limitations on what **can** or **can't** be made into a lens!

Lens
: An optic which **always** accesses **exactly one focus**.

There are certain things you might want to do that don't fit into the idea of focusing on exactly **one** thing within a **structure**. In most of these cases there's a different type of optic which can do what you need, but it's helpful to explicitly recognize the distinction.

You won't usually need to worry about breaking this rule by accident since the `lens` helper function steers you in the right direction. However when you're thinking about what you can or can't make into a lens it's handy to know the rules ahead of time. The best way to learn the rules is to practice!

Let's make a game out of it, step on down to play: **Is It A Lens?**

Is it a Lens?

Welcome to "Is it a Lens"! The simple text-based game-show where the only prize is your own knowledge!

The rules are simple, I'll ask you whether it's possible to write a **valid lens** which matches the provided signature. You just answer **yes** or **no** and be ready to back up your answer. If you want to play along at home, try to actually implement the lens and see why it is or isn't possible. Also; partial functions aren't allowed!

I've grouped the answers together **AFTER** all the questions at the end of this chapter so you that you're not tempted to peek. I recommend thinking about each problem for a while, come up with your guess and justifications for it, then check the answer before moving on to the next one. Be careful when checking the answers, they're all grouped up on the same page. Hold your hand over your screen (or your eyes) if you need to...

Let's play!

---

1. Is it possible to build a lens which focuses the second element from a three-tuple?

```
second :: Lens' (a, b, c) b
```

Jump to answer

---

2. What about a lens which gets or sets the value inside a `Maybe`?

```
inMaybe :: Lens' (Maybe a) a
```

Jump to answer

---

3. How about a lens which focuses on the value inside an Either?

```
left :: Lens' (Either a b) a
```

Jump to answer

And now a word from our sponsors!

Or at least there would be, if this weird textbook gameshow could get any sponsors... seriously, if you've got hookups, let me know...

Back to your regularly scheduled *programming*

---

4. How about this lens which focuses the third element of a list. Is `listThird` a valid lens?

```
listThird :: Lens' [a] a
```

Jump to answer

Here's a tricky one!

5. Can you write a lens which focuses the second element of a tuple if the Bool in the first slot is `True` and focuses the third element if it's `False`?

```
conditional :: Lens' (Bool, a, a) a
```

Jump to answer

Last one! Try to give our audience a big finale!

6. Consider the following data type (which would generally be considered bad style):

```
data Err =
    ReallyBadError { _msg :: String }
  | ExitCode       { _code :: Int }
makeLenses ''Err
```

Can you write a valid lens for:

```
msg :: Lens' Err String
```

Jump to answer

That's all for **Is it a Lens**! Thanks for playing! Here are those answers I promised.

Is it a Lens? – Answers

1. Is it possible to build a lens which focuses the second element from a three-tuple:

```
second :: Lens' (a, b, c) b
```

Yes! This is a valid lens because no matter which three-tuple we're given we can always get or set the second element! This is provided by the `lens` library as `_2`.

---

2. What about a lens which gets or sets the value inside a `Maybe`?

```
inMaybe :: Lens' (Maybe a) a
```

Nope! We can focus the value inside a `Just`; but what if the caller passes us a `Nothing` instead! We wouldn't be able to get a value out, what would `view` do!?

Here's what it would look like if we tried to write this lens:

```
inMaybe :: Lens' (Maybe a) a
inMaybe = lens getter setter
  where
    getter (Just a) = a
    getter Nothing = ???
    setter oldStructure newA = fmap (const newA)
```

We can get close, but we simply can't **get** anything out of a `Nothing`!

Keep an eye out for the `_Just` prism which helps us out with doing this sort of thing, we'll revisit it later, but it's not a lens!

---

3. How about a lens which focuses on the value inside an Either?

```
left :: Lens' (Either a b) a
```

This one's a trick question, if you answered with "it depends" (the one true answer to all programming-related questions) then you're right!

The **exact** type signature I specified **cannot** be a proper lens. For example if we specialize it to `Either String Int` we can't always get and set a `String` for every possible value we might be passed; what if we get a `Right`!? So that means it can't be a valid lens.

However, if we slightly alter the type:

```
chosen :: Lens' (Either a a) a
```

In this case **both** `Left` and `Right` contain the same type of value! That means we **can** focus exactly **one** value of type `a` no matter what they pass us, so this is a valid lens! This is provided by the `lens` library as the `chosen`[2] combinator. This particular example provides valuable insight that some lenses can **choose** what they'll focus on depending on the **structure** they're passed in. The only requirement is that they always choose exactly one focus at a time.

Only a lens deals with absolutes;
– Optic-wan Kenobi

---

4. How about this lens which focuses the third element of a list. Is `listThird` a valid lens?

```
listThird :: Lens' [a] a
```

Nope, Can you think of a list value which would prevent you from selecting the third element? Remember that we **must** be able to select exactly **one** element as the focus for our optic to be a lens. If the user provides an empty list there's no third element for us to focus on!

In fact there's **no** valid lens for the type `(Lens' [a] a)` because it's always possible that the input list is empty!

---

5. Can you write a lens which focuses the second element of a tuple if the provided Bool is `True` and the third element if it's `False`?

```
conditional :: Lens' (Bool, a, a) a
```

`conditional` is a valid lens! Even though there is more than one value of type `a` present, we will only ever focus **exactly one** of them at a time, and we can always get or set exactly one thing. Try writing this lens using the `lens` helper, it's good practice.

---

6. Consider the following data type (which would generally be considered bad style):

[2] https://hackage.haskell.org/package/lens/docs/Control-Lens-Lens.html#v:chosen

```
data Err =
    ReallyBadError { _msg :: String }
  | ExitCode       { _code :: Int }
```

Can you write a valid lens for:

```
msg :: Lens' Err String
```

Errm... technically you **can** write a lens using total functions which matches this signature, but it gets a bit weird!

What happens when you're passed an `ExitCode` instead of `ReallyBadError`? Well, the setter could just do nothing, but what about the getter? Since you know it needs a string I suppose you could just return an empty string right? This would **compile** but it still just *seems* **wrong** doesn't it?

The reason this **feels** wrong is because it circumvents our expectations of what a lens is supposed to do! This intuition is backed up by the **lens laws**, which we'll talk about in depth in the next section, so remember this example till then.

3.5 Lens Laws

I've been playing pretty fast and loose with the rules, types, and laws around lenses lately so we could start to build some intuition quickly without drowning in academic verbosity; but it's probably best that I cover a bit of the boring stuff with more academic rigour before we go any further. I promise there's a surprise waiting for you if you make it through the whole chapter.

Why do optics have laws?

A natural thought is to wonder why need laws at all; getters and setters in most other languages don't usually have any laws!

The reason we want laws is that the restrictions they impose let us **reason** about our code more clearly. We certainly wouldn't want to try getting a value from a tuple and accidentally end up lauching nukes or erasing our hard-drive! Having laws tells us how a *well-behaved* lens should act. It also prevents lenses with behaviour that just feels **weird**, like the last question in the previous "Is It A Lens" chapter. Sometimes *weird* (a.k.a. unlawful) lenses can be useful, but it's important to know the rules so we can think carefully before deciding to break them or not.

Let's see the actual laws which can tell us whether a lens is *weird* or not.

The Laws

Most typeclasses in Haskell have laws! These laws provide invariants which the typeclass methods must follow. These laws allow programmers to reason about the behaviour of typeclasses when they're writing polymorphic functions. They help ensure that polymorphic code still does "The Right Thing"™.

Even though lenses are **values** instead of **typeclasses** they still have a few laws. These laws are mostly *guidelines*, but following the laws helps make lenses more predicatable and things will still behave properly when we pass lenses off to helper functions or lens combinators. Typically you should follow these laws when writing field accessors and basic data accessors, but there's a bit more wiggle room if you're using lenses to emulate virtual fields or to do input cleanup/correction. We'll look at some of these examples later on.

Here's the full list of the lens laws from the `lens` library itself, I use == here to mean that both the left-hand side and right-hand side should be equal.

You get back what you set (set-get)

If you `set` using a lens, then `view` through it, you should get back what you just set!

```
view myLens (set myLens newValue structure)
  ==
newValue
```

Setting back what you got doesn't do anything (get-set)

This law helps ensure that our lenses aren't causing any weird side-effects like ticking a counter or mutating something other than their focus. If we set the focus to what that focus currently contains (according to `view`), then of course nothing should change!

```
set myLens (view myLens structure) structure
  ==
structure
```

Setting twice is the same as setting once (set-set)

This law is similar to the previous in that it ensures there's no funny business happening behind the scenes. Setting through a lens is a declarative statement, and should be idempotent. If we set the focus twice, only the last `set` should have any visible effect.

```
set myLens differentValue (set myLens value structure)
  ==
set myLens differentValue structure
```

Here we can see that a `set` should **overwrite** any previous `sets` completely as though the first set had never happened. The lens shouldn't *record* anything, mutate any extra state, or erase your hard-drive when it's used..

Each of these laws is there for a reason. In general they help to make lenses easier to use, so if you find yourself wanting to break one of the laws there's usually a different type of optic which better suits your needs!

Don't worry about checking any lenses generated by `makeLenses`, they're always lawful.

Breaking Laws

Nothing will explode or go catastrophically wrong if you break a law or two; sometimes unlawful lenses are very useful! Combinators in lens typically don't require that the lenses you pass them are lawful; however if you're writing a library it's a good idea to note in the documentation when a lens is unlawful so people know what to expect.

It's like jazz; first you've got to learn the rules, then you can break them.

Case Study: `_1`

That was a lot of words, let's see some code! First we'll see a lens which **follows** all of the laws. Let's use `_1`;

You get back what you set (set-get)

```
>>> let newValue = "Firefly"
>>> view _1 (set _1 newValue ("Star Wars", "Star Trek"))
 "Firefly"
```

We can express the full law as a computation like this:

```
>>> view _1 (set _1 newValue ("Star Wars", "Star Trek")) == newValue
True
```

For simple laws like this we could actually write property tests using a library like QuickCheck[3] if we wanted to. I won't get into property-based testing right now, but the idea is that we could accept a lens, try this check on hundreds of random structures and focuses, and verify that this assertion holds for all of them!

[3] http://hackage.haskell.org/package/QuickCheck

Setting back what you got doesn't do anything (get-set)

```
>>> let structure = ("Simpsons", "Seinfeld")
>>> set _1 (view _1 structure) structure
("Simpsons","Seinfeld")

>>> set _1 (view _1 structure) structure == structure
True
```

Setting twice is the same as setting once (set-set)

```
>>> let value1 = "Coffee"
>>> let value2 = "Red Bull"
>>> let structure = ("Beer", "Tea")
>>> set _1 value2 (set _1 value1 structure)
("Red Bull","Tea")
>>> set _1 value2 structure
("Red Bull","Tea")
>>> set _1 value2 (set _1 value1 structure) == set _1 value2 structure
True
```

Not much to say about this, we punched in the code and it told us we passed! For more complex lenses you'll want to think about edge-cases to test too.

You can assume most lenses included with an optics library will be lawful; if they aren't they'll likely be in a separate `Unsafe` or `Unsound` module or will have documentation detailing how they break the laws.

Case Study: `msg`

In the **Is it a Lens** game we saw the following data type:

```
data Err =
    ReallyBadError { _msg :: String }
  | ExitCode       { _code :: Int }
```

And considered the following lens:

```
msg :: Lens' Err String
```

Let's implement it now and see why it's no good!

```haskell
msg :: Lens' Err String
msg = lens getMsg setMsg
  where
    getMsg (ReallyBadError message) = message
    -- Hrmm, I guess we just return ""?
    getMsg (ExitCode _) = ""
    setMsg (ReallyBadError _) newMessage = ReallyBadError newMessage
    -- Nowhere to set it, I guess we do nothing?
    setMsg (ExitCode n) newMessage = ExitCode n
```

Despite making a few strange choices we've managed to implement a lens with the needed signature; so is it lawful? Let's see!

You get back what you set (set-get)

```
>>> let newMessage = "False alarm!"
>>> view msg (set msg newMessage (ReallyBadError "BAD BAD BAD"))
"False alarm!"
>>> view msg (set msg newMessage (ReallyBadError "BAD BAD BAD")) == newMessage
True
```

So far we're passing! Let's see what happens if it was an `ExitCode` instead!

```
>>> let newMessage = "False alarm!"
>>> view msg (set msg newMessage (ExitCode 1))
""
>>> view msg (set msg newMessage (ExitCode 1)) == newMessage
False
```

Oh no! We fail this test, so the lens isn't lawful. We can finally pin down why this lens seems weird; we don't always get back what we set!

Case Study: `lensProduct`

Let's take a look at a very useful law-breaking lens: `lensProduct`[4].

```haskell
lensProduct :: Lens' s a -> Lens' s b -> Lens' s (a, b)
```

This lens is actually provided by the `lens` library, but it's in `Control.Lens.Unsound` to indicate that you should be careful when using it.

[4] https://hackage.haskell.org/package/lens-4.18/docs/Control-Lens-Unsound.html#v:lensProduct

It allows you to take two lenses which accept the same structure to simultaneously focus two distinct parts of it. That sounds reasonable enough, and indeed this situation comes up relatively often. Can you see why this lens could break laws on certain inputs? It's not immediately obvious, so let's dive into an example. First let's see a practical use for `lensProduct`.

We've got a little setup to do first:

```
type UserName = String
type UserId = String

data Session =
    Session { _userId      :: UserId
            , _userName    :: UserName
            , _createdTime :: String
            , _expiryTime  :: String
            }
            deriving (Show, Eq)
makeLenses ''Session
```

Here's a data type representing a user session (humour me while I attempt to invent helpful examples). You can imagine any situation in which you want to extract or operate on **more than one** thing within a deeply nested chunk of state; I'll keep this one flat for simplicity.

Let's say we want to dive into the session to operate on the `UserId` and `UserName`, but don't care about the created or expiry times. We can use `lensProduct` to take the **product** (i.e. tuple) of the `userId` and `userName` lenses, creating a new lens over a tuple of their focuses!

```
userInfo :: Lens' Session (UserId, UserName)
userInfo = lensProduct userId userName
```

Let's make sure it works:

```
>>> let session =
  Session "USER-1234" "Joey Tribbiani" "2019-07-25" "2019-08-25"
>>> view userInfo session
("USER-1234", "Joey Tribbiani")
```

Nifty! Now we can perform our actions over just those two things without anything getting in the way.

So what can possibly go wrong? I'll save you the time in checking and tell you that the lens we've constructed here actually **IS lawful**; not every lens we can create with `lensProduct` is illegal, the issue is that it is **able** to create illegal lenses if we're not extra careful.

In order to stay lawful, we must ensure that the two focuses are **disjoint**; i.e. they don't have any overlap. If instead we decide we want to pair up the `userId` alongside the **entire** original `Session` type; then we have a problem:

```
alongsideUserId :: Lens' Session (Session, UserId)
alongsideUserId = lensProduct id userId
```

> Here we use `id` as a lens! This seems a bit magical until we understand a bit more about the underlying encodings, which we'll get to later, but for now you can pretend that `id` has the following type:
>
> ```
> id :: Lens' s s
> ```
>
> That is to say, it focuses the full structure it's passed, it's the identity lens!
>
> We use it here to include the full session type in one half of the product.

In this case, our two focuses are **NOT** disjoint; they're... ermm... *joint* I guess? The `UserId` we focus in one half of the tuple is also present in the full `Session`! Now we start to see the problem. What is supposed to happen when we `set` the `UserId` in the session, and also set the `UserId` directly in the other half of the tuple? Let's see what our laws have to say about this one:

You get back what you set (set-get)

```
>>> let session =
  Session "USER-1234" "Joey Tribbiani" "2019-07-25" "2019-08-25"
>>> let newSession = session{_userId="USER-5678"}
>>> view alongsideUserId (set alongsideUserId (newSession, "USER-9999") session)
( Session
    { _userId = "USER-9999"
    , _userName = "Joey Tribbiani"
    , _createdTime = "2019-07-25"
    , _expiryTime = "2019-08-25"
    }
, "USER-9999"
)
>>> view alongsideUserId (set alongsideUserId (newSession, "USER-9999") session)
  == (newSession, "USER-9999")
False
```

We can see that the `"USER-9999"` change has overwritten the "USER-1234" change we applied to the session. If we provide the lenses in the opposite order then the reverse would be true; the value in the session would overwrite the other. Try it out!

The other two laws pass for `lensProduct`, but breaking one law is enough to be a bit unintuitive.

This is a great example of an unlawful lens that's still very useful, so long as you're careful how you use it.

We've seen a few lenses which fail the first law, but what about the other two? The first exercise is to implement one yourself!

Exercises - Laws

Jump to answers

1. Implement a lens which breaks the second and/or third law. That's get-set and set-set respectively.
2. Test the get-set and set-set laws for the `msg` lens we wrote this chapter. Does it pass these laws?
3. There's a different way we could have written the `msg` lens such that it would PASS the set-get law and the set-set law, but fail get-set. Implement this other version.
4. Think up a new lens which is still useful even though it breaks a law or two.
5. **BONUS** (this one is tricky): Live a little; write a lens which violates ALL THREE LAWS
6. **BONUS** (another tricky one): Can you write a lawful lens for the following type:

```
data Builder =
    Builder { _context :: [String]
            , _build   :: [String] -> String
            }
```

Your lens should be of type:

```
Lens' Builder String
```

Hint: There's a bit of a trick to get this one to work; the get-set law will be the trickiest, especially since you won't be able to directly compare `Builders` for equality!

Great job getting through everything; here's your surprise as promised!

A cat

3.6 Virtual Fields

I mentioned earlier how lenses subsume the Accessor Pattern from Object-Oriented programming. We've already seen how lenses take care of the **getters** and **setters** for record fields, this chapter covers how to represent "virtual fields" with lenses.

What's a virtual field

I'm using the term **virtual field** to mean any conceptual piece of data that doesn't exist as an actual field in your record definition. These are sometimes called "computed properties" or "managed attributes" in lanuages like Java, Python, etc. They're often used to present the data from concrete fields in a more convenient or enriched way. Sometimes they combine several concrete fields together, other times they're just used to avoid breaking changes when refactoring the structure of the record.

At the end of the day; they're really just normal lenses! Let's look at a few examples.

Writing a virtual field

For a simple example let's look at the following type:

```
data Temperature =
    Temperature { _location :: String
                , _celsius  :: Float
                }
    deriving (Show)
```

This generates the field lens:

```
celsius :: Lens' Temperature Float
```

Which we can use to get or set the temperature in celsius.

```
>>> let temp = Temperature "Berlin" 7.0
>>> view celsius temp
7.0

>>> set celsius 13.5 temp
Temperature {_location = "Berlin", _celsius = 13.5}

-- Bump the temperature up by 10 degrees Celsius
>>> over celsius (+10) temp
Temperature {_location = "Berlin", _celsius = 17.0}
```

But what about our American colleagues who'd prefer **Fahrenheit**? It'd be easy enough to write a function which convert **Celsius** to **Fahrenheit** and call that on the result, but you'd still need to **set** new temperatures using **Celsius**!

First we'll define our conversion functions back and forth, nothing too interesting there:

```
celsiusToFahrenheit :: Float -> Float
celsiusToFahrenheit c = (c * (9/5)) + 32

fahrenheitToCelsius :: Float -> Float
fahrenheitToCelsius f = (f - 32) * (5/9)
```

Here's how we *could* get and set using Fahrenheit:

```
>>> let temp = Temperature "Berlin" 7.0
>>> celsiusToFahrenheit . view celsius temp
44.6

>>> set celsius (fahrenheitToCelsius 56.3) temp
Temperature {_location = "Berlin", _celsius = 13.5}

-- Bump the temp by 18 degrees Fahrenheit
>>> over celsius (fahrenheitToCelsius . (+18) . celsiusToFahrenheit) temp
Temperature {_location = "Berlin", _celsius = 17.0}
```

The first two aren't **too** bad, but the `over` example is getting a bit clunky and error prone!

If we instead encode the **Fahrenheit** version of the temperature as a virtual field we gain better usability, cleaner code, and avoid a lot of possible mistakes.

Now for the fun part! We can write a `fahrenheit` lens **using** the existing `celsius` lens! We simply convert back and forth when getting and setting.

```
fahrenheit :: Lens' Temperature Float
fahrenheit = lens getter setter
  where
    getter = celsiusToFahrenheit . view celsius
    setter temp f = set celsius (fahrenheitToCelsius f) temp
```

Look how it cleans up the call site:

```
>>> let temp = Temperature "Berlin" 7.0
>>> view fahrenheit temp
44.6

>>> set fahrenheit 56.3 temp
Temperature {_location = "Berlin", _celsius = 13.5}

>>> over fahrenheit (+18) temp
Temperature {_location = "Berlin", _celsius = 17.0}
```

Much cleaner! Even though our `Temperature` record doesn't have a field for **Fahrenheit** we faked it using lenses to create a **virtual field**!

Breakage-free refactoring

Another great benefit of using lenses instead of field accessors for interacting with our data is that we gain more freedom when refactoring. To continue with the Temperature example, let's say as

we've developed our wonderful weather app further we've discovered that Kelvin is a much better canonical representation for temperature data. We'd love to swap our `_celsius` field for a `_kelvin` field instead.

We'll consider two possible universes, in one this book was never written, so we didn't use lenses to access our fields. In the other (the one you're living in) we finished the book and decided to use lenses as our external interface instead.

The universe without lenses

In the sad universe without lenses we had the following code scattered throughout our app:

```
updateTempReading :: Temperature -> IO Temperature
updateTempReading temp = do
  newTempInCelsius <- readTemp
  return temp{_celsius=newTempInCelsius}
```

Then we refactored our `Temperature` object to the following:

```
data Temperature =
    Temperature { _location :: String
                , _kelvin  :: Float
                }
    deriving (Show)
```

And unfortunately every file that used record update syntax now fails to compile, because the `_celsius` field no longer exists. If we had instead used pattern matching, the situation would be even worse:

```
updateTempReading :: Temperature -> IO Temperature
updateTempReading (Temperature location _) = do
  newTempInCelsius <- readTemp
  return (Temperature location newTempInCelsius)
```

In this case the code will still compile, but we've completely switched units, this will behave completely incorrectly!

The glorius utopian lenses universe

Come with me now to the happy universe. In this universe we decided to use lenses as our interface for interacting with `Temperatures`, meaning we didn't expose the field accessors and thus disallowed fragile record-update syntax. We used the `celsius` lens to perform the update instead:

```
updateTempReading :: Temperature -> IO Temperature
updateTempReading temp = do
  newTempInCelsius <- readTemp
  return $ set celsius newTempInCelsius temp
```

Now when we refactor, we can simply export a replacement `celsius` lens in place of the old one:

```
data Temperature =
    Temperature { _location :: String
                , _kelvin  :: Float
                }
    deriving (Show)
makeLenses ''Temperature

celsius :: Lens' Temperature Float
celsius = lens getter setter
  where
    getter = (subtract 273.15) . view kelvin
    setter temp c = set kelvin (c + 273.15) temp
```

By adding the replacement lens we avoid breaking any external users of the type! Even our `fahrenheit` lens was defined in terms of `celsius`, so it will continue to work perfectly.

This is a simple example, but this idea works for more complex refactorings as well. When adopting this style it's important to avoid exporting the data type constructor or field accessors. Export a "smart constructor" function and the lenses for each field instead.

Exercises – Virtual Fields

Jump to answers

Consider this data type for the following exercises:

```
data User =
  User { _firstName :: String
       , _lastName :: String
       , _username :: String
       , _email :: String
       }
makeLenses ''User
```

1. We've decided we're no longer going to have separate usernames and emails; now the email will be used in place of a username. Your task is to delete the `_username` field and write a replacement `username` lens which reads and writes from/to the `_email` field instead. The change should be unnoticed by those importing the module.

2. Write a lens for the user's `fullName`. It should append the first and last names when "getting". When "setting" treat everything till the first space as the first name, and everything following it as the last name.

It should behave something like this:

```
>>> let user = User "John" "Cena" "invisible@example.com"
>>> view fullName user
"John Cena"
>>> set fullName "Doctor of Thuganomics" user
User
    { _firstName = "Doctor"
    , _lastName = "of Thuganomics"
    , _email = "invisible@example.com"
    }
```

3.7 Data correction and maintaining invariants

We just learned about using lenses for computed and virtual fields; there's an extension to this idea where we can use lenses to perform certain types of data correction to ensure our data remains in a valid state. This is easiest explained with an example so we'll jump right in.

Including correction logic in lenses

Imagine we've got a rudimentary data type for storing clock time:

```
data Time =
    Time { _hours :: Int
         , _mins  :: Int
         }
    deriving (Show)
```

We want to allow users to edit the time of the clock, so we'll expose some lenses! However, we want to make sure that no matter what, our `hours` value remains between 0-23, and the minutes remain between 0-59. If the user tries to set the values outside of that range we'll simply clamp the value to fit the range instead. We can do this pretty easily by adding some simple logic to our **setter**s

```
clamp :: Int -> Int -> Int -> Int
clamp minVal maxVal a = min maxVal . max minVal $ a

hours :: Lens' Time Int
hours = lens getter setter
  where
    getter (Time h _) = h
    setter (Time _ m) newHours = Time (clamp 0 23 newHours) m

mins :: Lens' Time Int
mins = lens getter setter
  where
    getter (Time _ m) = m
    setter (Time h _) newMinutes = Time h (clamp 0 59 newMinutes)
```

These custom lenses clamp any new values we're setting to be within the expected range.

```
>>> let time = Time 3 10
>>> time
Time {_hours = 3, _mins = 10}

>>> set hours 40 time
Time {_hours = 23, _mins = 10}

>>> set mins (-10) time
Time {_hours = 3, _mins = 0}
```

This ensures that the values are within the expected ranges when setting! If you're you're a bit paranoid you could also clamp the getters.

Hopefully at some point during this section you thought "wait a minute, is this lawful"? The answer is **no**, **these are not lawful lenses**. If we `set` a bad value, we'll `get` the corrected value instead. This is usually fine, but it's good to think carefully about whether this behaviour is acceptable to you or not.

This isn't the only type of correction we could make in this scenario. If we wanted we could actually have the "minutes" and "hours" fields *roll over* when out of bounds. This makes it possible to do operations like adding 90 minutes to a time and still getting a sensible answer. Let's see how that would look:

```haskell
hours :: Lens' Time Int
hours = lens getter setter
  where
    getter (Time h _) = h
    -- Take the hours 'mod' 24 so we always end up in the right range
    setter (Time _ m) newHours = Time (newHours `mod` 24) m

mins :: Lens' Time Int
mins = lens getter setter
  where
    getter (Time _ m) = m
    -- Minutes overflow into hours
    setter (Time h _) newMinutes
      = Time ((h + (newMinutes `div` 60)) `mod` 24) (newMinutes `mod` 60)
```

In this new configuration we can add or subtract minutes and hours from the clock time and the lens will automatically **normalize** the minutes and hours!

```
>>> let time = Time 3 10
>>> time
Time {_hours = 3, _mins = 10}

>>> over mins (+ 55) time
Time {_hours = 4, _mins = 5}

>>> over mins (subtract 20) time
Time {_hours = 2, _mins = 50}

>>> over mins (+1) (Time 23 59)
Time {_hours = 0, _mins = 0}
```

Nifty! Again; these lenses are **unlawful**, but still useful!

You're probably wondering whether there are ways to provide an error message on invalid input rather than silently correcting it; and indeed there are! We'll just need to learn a few more things before we're ready to take that on.

Exercises – Self-Correcting Lenses

Jump to answers

Consider the following:

```
data ProducePrices =
    ProducePrices { _limePrice  :: Float
                  , _lemonPrice :: Float
                  }
    deriving Show
```

1. We're handling a system for pricing our local grocery store's citrus produce! Our first job is to write lenses for setting the prices of limes and lemons. Write lenses for `limePrice` and `lemonPrice` which prevent **negative** prices by rounding up to `0` (we're okay with given produce out for free, but certainly aren't going to pay others to take it).
2. The owner has informed us that it's VERY important that the prices of limes and lemons must NEVER be further than 50 cents apart or the produce world would descend into total chaos. Update your lenses so that when setting lime-cost the lemon-cost is rounded to within 50 cents; (and vice versa).

It should behave something like this; don't worry if you can't get it exactly right, this one is tricky!

```
>>> let prices = ProducePrices 1.50 1.48
>>> set limePrice 2 prices
ProducePrices
    { _limePrice = 2.0
    , _lemonPrice = 1.5
    }
>>> set limePrice 1.8 prices
ProducePrices
    { _limePrice = 1.8
    , _lemonPrice = 1.48
    }
>>> set limePrice 1.63 prices
ProducePrices
    { _limePrice = 1.63
    , _lemonPrice = 1.48
    }
>>> set limePrice  (-1.00) prices
ProducePrices
    { _limePrice = 0.0
    , _lemonPrice = 0.5
    }
```

4. Polymorphic Optics

4.1 Introduction to polymorphic optics

Simple vs Polymorphic optics

So far we've been working with **Simple** lens types, things like:

- `Lens' Person Address`
- `Lens' (a, b) a`
- `Lens' Pet Name`

These all match the general form:

```
Lens' s a
```

Where `s` is the **structure** and `a` is the **focus**.

However, there are also things called **polymorphic lenses** which drop the trailing tick `'` and have a few extra type parameters:

```
Lens s t a b
```

You've probably seen these around if you've looked through the documentation of your favourite lens library.

This adds `t` and `b` parameters, which allow us to distinguish the types flowing **in** from those flowing **out**. In other words, `s` and `a` represent the types of the structure and focus **before** they've been acted on, `t` and `b` are the structure and focus **after** being acted on.

- `s`: structure **before** action
- `t`: structure **after** action
- `a`: focus **before** action
- `b`: focus **after** action

The simple lens type: `Lens'` is actually just an alias to polymorphic one:

```
type Lens' s a = Lens s s a a
```

When we run an action (like `over`) on our lens we dive deep into the structure `<s>` to find the focus `<a>`, then pass that to the **action**. The action has the option of modifying the focus to return a `<b>`, then the lens glues everything back together and construct a `<t>`.

We need polymorphic lenses whenever an action might want to change the type of the focus.

4.2 When do we need polymorphic lenses

Let's expand on one of our simple examples:

```
_1 :: Lens' (a, b) a
```

Here's the simplified type of `over` we've been using so far:

```
over :: Lens' s a -> (a -> a) -> s -> s
```

If we partially apply `over` to `_1` we get:

```
over _1 :: (a -> a) -> (a, b) -> (a, b)
```

Let's make it even more concrete by saying we're working with a tuple `(Int, Bool)`, so `a = Int` and `b = Bool`:

```
over _1 :: (Int -> Int) -> (Int, Bool) -> (Int, Bool)
```

The types we've used show that we're restricted to a modification from `(Int -> Int)`. What if instead we want to use `show` to turn that `Int` into a `String`? Our simple lens types **prevent** us from doing so.

Tuples have a separate type variable for each slot, if we're focusing on that slot we should be able to change type! The **output structure** would simply need to reflect that we've changed the type corresponding to the focused slot.

Type-changing focuses

Let's start converting our **simple** lenses to **polymorphic** ones.

Let's take a look at `_1`. Sorry to be the one to tell you this; but I've been *lying* about the type of the `_1` lens this whole time! It's actually a polymorphic lens. Here's a new type signature for it, (though I'm still lying a *little*).

```
_1 :: Lens (a, other) (b, other) a b
```

This signature says that if you provide a tuple `(a, other)` you can run a type-changing action from `a -> b` and you'll get back the result: `(b, other)`.

I was lying about the type of `over` too; here's a more relaxed version which shows how it handles polymorphic lenses:

```
over :: Lens s t a b -> (a -> b) -> s -> t
```

Let's follow the types as we apply `over` to our problem.

We'll start by partially applying `over` to our `_1` lens:

```
over _1 :: (a -> b) -> (a, other) -> (b, other)
```

We can imagine that `show` has the type (`show :: Int -> String`), so if we apply `over _1 show`:

```
over _1 show :: (Int, other) -> (String, other)
```

Our `over` action allowed us to **change the type of the focus**, resulting in a change of type to the full **structure**.

We've seen how `over` is a type-changing action, but what about `set` or `view`?

The `set` action takes a new value and puts it into the focused slot, replacing the old value. This **forgets** the existing focus and provides the new value as the **output** focus. The polymorphic version of `set` looks like this:

```
set :: Lens s t a b -> b -> s -> t
```

To see it more concretely, in the context of the previous example it would be something like this:

```
set :: Lens (anything, other) (String, other) anything String
    -> String
    -> (String, other)
```

Lastly we'll look at `view`. `view` is a bit strange because `view` returns ONLY the focus and doesn't really perform any actions or modifications on it. Since it doesn't perform any modifications, `view` just accepts a simple lens instead of a polymorphic one:

```
view :: Lens' s a -> s -> a
```

As you learn about the other families of optics going forwards, remember that most other optics can be polymorphic just like lenses can. You're officially equipped to handle polymorphic optics in the wild!

Changing type variables with polymorphic lenses

We saw how we can use polymorphic lenses to change the type of specific *slots* of a tuple's type, but this principle generalizes to type variables in other data types as well.

Here's a type which represents a discount for any item type:

```
data Promotion a =
    Promotion { _item :: a
              , _discountPercentage :: Double
              }
    deriving (Show)
```

This time we're going to implement the lens by hand to really get a handle on it. If we want a lens which edits the `_item` field, we notice that the top-level type-variable will need to change too. It's a good idea in this situation to write out the type of the lens you plan to write before starting:

```
item :: Lens (Promotion a) (Promotion b) a b
```

This says that we can go from a `<Promotion a>` to a `<Promotion b>` if you provide an action which goes from `<a>` to `<b>`.

Let's implement it:

```
item :: Lens (Promotion a) (Promotion b) a b
item = lens getter setter
    where
      getter :: Promotion a -> a
      getter = _item
      setter :: Promotion a -> b -> Promotion b
      setter promo newItem = promo{_item = newItem}
```

Great! Now we can change a promo for a really delicious Peach into a promo for a whole list of mint-condition Buffy the Vampire figurines. Don't ask me why; I just follow the laws of supply and demand…

```
>>> let peachPromo = Promotion "A really delicious Peach" 25.0

>>> :t peachPromo
peachPromo :: Promotion String

>>> let buffyFigurines = ["Buffy", "Angel", "Willow", "Giles"]
>>> :t buffyFigurines
buffyFigurines :: [String]

>>> let buffyPromo = set item buffyFigurines peachPromo
>>> buffyPromo
Promotion
    { _item = [ "Buffy" , "Angel" , "Willow" , "Giles" ]
    , _discountPercentage = 25.0
    }

>>> :t buffyPromo
buffyPromo :: Promotion [String]
```

Here's a different data type for representing our favourite and least favourite things:

```
data Preferences a =
    Preferences { _best :: a
                , _worst :: a
                }
    deriving (Show)
```

Think for a second about whether we can write polymorphic lenses for `best` and `worst`, in fact I encourage you to try it out!

Did you give it a try? We actually can't do it! Since a lens focuses exactly **one** thing, and there are **two** things in the structure with the same type variable, there's no way we can use a lens to change them both at once. We'll learn about a different type of polymorphic optic which can do this later on. (Spoilers: it's called a Traversal).

Exercises – Polymorphic Lenses

Jump to answers

1. Write the type signature of the polymorphic lens which would allow changing a `Vorpal x` to a `Vorpal y`.

2. Find one possible way to write a polymorphic lens which changes the type of the `best` and `worst` fields in the `Preferences` type above. You're allowed to change the type of the lenses or alter the type itself!
3. We can change type of more complex types too. What is the type of a lens which could change the type variable here:

```
data Result e =
    Result { _lineNumber :: Int
           , _result     :: Either e String
           }
```

4. It's thinking time! Is it possible to change more than **one** type variable at a time using a polymorphic lens?
5. **BONUS** Come up with some sort of lens to change from a `Predicate a` to a `Predicate b`

```
data Predicate a =
    Predicate (a -> Bool)
```

Hey you! You're doing a **fantastic** job cruising through this book! Did I mention how much I appreciate you investing in it and taking the time to read it through? If you've learned something or enjoyed your read maybe you could tell a friend about it, tweet about it[1], or even send me a short testimonial[2]! If something's wrong, or you have feedback on how it could be better, please tell me about it[3]!

I hope you don't mind taking a quick minute to help me make the book the best it can be, and to help me make even more content like this! The only reason I can spend time on this book (and hopefully more like it!) is because people like you share it around and recommend it to others. Thanks so much for your support!

All done? You shared it with all your friends? Told your favourite grandma to buy a copy? Okay great! Carry on your way...

4.3 Composing Lenses

By now you've got a rough understanding of what makes a lens tick; it's time to see what they're really great at.

[1] https://twitter.com/intent/tweet?text=Interested%20in%20lenses%20or%20optics?%20You%20seriously%20need%20to%20check%20out%20@OpticsByExample%20by%20@chrislpenner!%20%23OpticsByExample

[2] https://forms.gle/rLhnCfN9GPfY4vku8

[3] https://forms.gle/x8yx3N5JeLKpw77U7

How do I update fields in deeply nested records?

You may have noticed in the section on using lenses for record fields that although bundling everything into a single value is cool; we didn't really save a *lot* in terms of code-size or clarity. Updating a single field isn't too tough to do using Haskell's built in record syntax, but when we start to look at deeply **nested** fields it's a whole different story.

Let's define a new record so you can see how lenses can help when things get a bit more complicated.

Let's make some nested types!

```
data Person =
    Person { _name    :: String
           , _address :: Address
           }
    deriving (Show)


data Address =
    Address { _streetAddress :: StreetAddress
            , _city          :: String
            , _country       :: String
            }
    deriving (Show)

data StreetAddress =
    StreetAddress { _streetNumber :: String
                  , _streetName   :: String
                  }
    deriving (Show)

makeLenses ''Person
makeLenses ''Address
makeLenses ''StreetAddress
```

Note how we put the `makeLenses` calls at the end of ALL data declarations.

If you get errors saying that types are "not-in-scope" even though they clearly are, it's likely your call to `makeLenses` needs to move around.

See `makeLenses` for more info.

These records are probably nested deeper than you'd really need to, but I'd wager that 3 levels of nesting isn't all that uncommon in most code-bases.

Let's say we have a person:

```
sherlock :: Person
sherlock =
    Person { _name    = "S. Holmes"
           , _address = Address
                { _streetAddress = StreetAddress
                      { _streetNumber = "221B"
                      , _streetName   = "Baker Street"
                      }
                , _city          = "London"
                , _country       = "England"
                }
           }
```

Now, say that Sherlock likes the view better from the other apartment unit; and he decides he'd like to move to "221A Baker Street" instead;

Take a look at the dumpster fire we end up with when we have to use normal record update syntax:

```
setStreetNumber :: String -> Person -> Person
setStreetNumber newStreetAddress person =
    let existingAddress       = _address person
        existingStreetAddress = _streetAddress existingAddress
    in person { _address = existingAddress
                  { _streetAddress = existingStreetAddress
                        { _streetNumber = newStreetAddress
                        }
                  }
              }
```

I hope you believe me when I swear I'm not even *trying* to make this look bad, this is really what you have to do to update nested records! After skimming over that monstrosity how sure are you that I didn't make any mistakes? Did I accidentally forget to set the field I intended? Did I maybe overwrite more than I intended? I bet your brain probably told you to just skip over that mess entirely.

I **did** perform the update correctly, but the point remains: **this is VERY hard to read**. The complexity **compounds** as the nesting gets deeper.

Imperative programmers are beginning to snicker at us from across the room, this would be a simple `sherlock.address.streetAddress.streetNumber = "221A"` in most of those languages! A primary reason for this ugliness is that record updates in Haskell use special **syntax** for updates rather than using a **function**, and syntax doesn't compose!

"There's gotta be a better way"
– Someone about to invent lenses

Let's at least confirm that all the hard work has paid off and that our update function works:

```
>>> setStreetNumber "221A" sherlock
Person
    { _name = "S. Holmes"
    , _address = Address
        { _streetAddress = StreetAddress
            { _streetNumber = "221A"
            , _streetName = "Baker Street"
            }
        , _city = "London"
        , _country = "England"
        }
    }
```

Note that you'd need to write something similarly awful for **each and every** field you'd ever want to update! I hope you'll agree at this point that this isn't a very ergonomic way to program.

> Does "Record Update Syntax" **spark joy**? If not, thank it for its service and throw it away.
>
> – Marie Kondo
>
> (from alternate universe 3.8C where she's a reputable functional programmer)

Composing update functions

If you were very clever, like, "I can reverse a binary tree on a white-board in 30s flat" sort of clever, you could reduce the boilerplate very slightly by breaking down the update functions into smaller chunks like this:

```
-- Update a Person's Address
updateAddress :: (Address -> Address)
              -> (Person -> Person)
updateAddress modify existingPerson =
    existingPerson
    { _address = modify . _address $ existingPerson
    }

-- Update a Street Address within an Address
updateStreetAddress :: (StreetAddress -> StreetAddress)
                    -> (Address -> Address)
updateStreetAddress modify existingAddress =
    existingAddress
    { _streetAddress = modify . _streetAddress $ existingAddress
    }
```

```
-- Update a Street Number within a Streed Address
updateStreetNumber :: (String -> String)
                   -> (StreetAddress -> StreetAddress)
updateStreetNumber modify existingStreetAddress =
    existingStreetAddress
    { _streetNumber = modify . _streetNumber $ existingStreetAddress
    }
```

That's a lot of boring code, but the important bit is to notice how each of these functions accepts a function for modifying a **single field** and returns a modifying function over the **larger record**.

For the rest of this chapter I'll refer to a "modifier" as any function which accepts and returns the same type (for some concrete type), and an **updater** as a function which accepts a **modifier** and returns a different **modifier**. In general, things that look something like this:

```
modifier :: (a -> a)

updater :: (a -> a) -> (s -> s)
```

There's a nifty trick we can use with updaters like this; if we line them up just right we can **compose** a **chain** of **updaters** so that they all feed into each other, collapsing like dominos!

Let's try it. Here's a reminder of our types:

```
updateStreetAddress
  :: (StreetAddress -> StreetAddress) -> (Address -> Address)

updateStreetNumber
  :: (String -> String) -> (StreetAddress -> StreetAddress)
```

If you look at these two functions you can see that `updateStreetAddress` *needs* a `StreetAddress` **modifier**, and we can see that `updateStreetNumber` *returns* a `StreetAddress` **modifier** if we give it a **modifier** for the `streetNumber` String!

That means we can feed the **result** of `updateStreetNumber` into `updateStreetAddress` using composition!

```
>>> :t (updateStreetAddress . updateStreetNumber)
  :: (String -> String) -> (Address -> Address)
```

See how **composing** two **updater**s results in new **updater**? The new updater accepts the **modifier** over the street number, but returns a modifier over the whole `Address`. By writing smaller separate updaters we've gained the ability to **compose** them in different ways to build new **updaters** that act how we'd like.

We're on a roll, let's see what we get when we go even further by composing `updateAddress` on the front.

```
>>> :t (updateAddress . updateStreetAddress . updateStreetNumber)
  :: (String -> String) -> (Person -> Person)
```

Nifty! So it looks like we can give it a modifier on **only the street number** and it'll build a modifier for the **whole person**! We're now using **function composition** instead of **syntax** to update our records.

Something interesting to notice here is that since each updater accepts an updater for a **smaller** piece of state, if we read the composition from **left to right** it reads like we're diving deeper one field at a time! It's almost like Object Oriented **dot-notation**, but using **updaters** instead! We **compose** our way down to the field we need.

In the predicament of dear Mr. Sherlock we only need to `set` his street number to a new one; not really `update` it. Since we want to **ignore** the current value of `_streetNumber` and just **replace** it with "221A" we can use the `const` function, which always returns a value regardless of its input, as our **modifier**.

```
>>> :t const "221A"
const "221A" :: b -> String
```

The `<b>` is polymorphic, so this type unifies to `(String -> String)` when it needs to.

By applying our chain of composed updaters to a final **modifier** we cause the whole chain to collapse into a simple function: `(Person -> Person)`

```
-- Here's our updater chain:
>>> :t (updateAddress . updateStreetAddress . updateStreetNumber)
  :: (String -> String) -> (Person -> Person)

-- We apply it to our final modifier to build a new modifier
>>>:t (updateAddress . updateStreetAddress . updateStreetNumber) (const "221A")
  :: (Person -> Person)

-- We apply the returned modifier to our person
>>> (updateAddress . updateStreetAddress . updateStreetNumber)
      (const "221A")
      sherlock
Person
    { _name = "S. Holmes"
    , _address = Address
        { _streetAddress = StreetAddress
            { _streetNumber = "221A"
            , _streetName = "Baker Street"
            }
```

```
        , _city = "London"
        , _country = "England"
        }
    }
```

Wow it works! Maybe I should try to sound a bit less surprised about that…

I know this can be a bit confusing to think; the updater returns another function when it's applied to a function! So many functions flying around. If you're having a bit of trouble grasping higher-order functions and composition like this I strongly recommend that you work through the above example yourself and run lots of type-checks along the way.

Composing Lenses

Okay, let's talk about what happened in that last section, we started out with some pretty ugly record updates, then broke each of those down into **small independent updaters**, each of which knew how to apply a modifier on a specific subset of its input. These **updater** things ended up being composable which gave us a much nicer way to update the field we cared about.

That was an awful lot of **non-lensy** stuff we just did there, what's the deal?

What if I told you that the updaters, and the way we composed them together represent the **essence** of optics! We experimented specifically with **updating** fields, but the idea actually generalizes over arbitrary behaviours with just a few tweaks!

Composition is at the *core* of how optics work

Optics deal with the notion of diving **deeper** into a **structure** one step at a time, handing off the metaphorical baton at each step, just like our **updaters** did.

If we were to use field lenses generated by `makeLenses` it would look like this; I'll show it side by side with the updaters:

```
updaters = (updateAddress . updateStreetAddress . updateStreetNumber)
lenses   = (address       . streetAddress       . streetNumber)
```

See the similarities?

Here are the full type signatures for each of these terms:

```
updaters ::              (String ->   String) -> Person ->   Person
lenses   :: Functor f => (String -> f String) -> Person -> f Person

-- And here's the simplified type alias signature of 'lenses'
lenses   :: Lens' Person String
```

Just as composing updaters makes another updater, **composing lenses** makes a new **lens**!

They're almost identical! `lenses` just has an extra `Functor` threaded through its signature. The `Functor` is the magic that lets us run all sorts of interesting **actions** besides just updates. By choosing the right `Functor` we can even do things that don't look like updates, like `view` the focus. We won't dig too deeply into the actual implementation of the library right now, but this should at least give you an intuition for the magic of optic composition, and give you hint about why "lens composition" is actually really just "function composition".

How do Lens Types Compose?

In the previous section we saw how the types of updaters snapped together like legos during composition, let's see how lens types do the same!

Using lenses from the previous example:

```
address       :: Lens' Person Address

streetAddress :: Lens' Address StreetAddress

streetNumber  :: Lens' StreetAddress String
```

It's pretty easy to see how lenses fit together. It's a lot like connecting pipe fittings, or aligning dominoes, or mixing metaphors...

Hopefully the following ASCII art is low resolution enough for you to picture dominoes, pipes, conveyor belts, or whichever roughly rectangular shape you prefer. Each lens converts from one type into the next, you really just have to line up the matching types.

We'll represent each lens as a domino-like shape, here's the `address` lens as a domino:

```
address :: Lens' Person Address

+---------+---------+
| Person  | Address |
+---------+---------+
```

When composing `(address . streetAddress)` The second type parameter of `address` must match the first type parameter of `streetAddress`. That is, we have to line up the matching sides of the dominos!

```
address :: Lens' Person Address
streetAddress :: Lens' Address StreetAddress
```

```
      (address         .         streetAddress)
+---------+---------+ <-> +---------+---------------+
| Person  | Address | <-> | Address | StreetAddress |
+---------+---------+ <-> +---------+---------------+
```

After grafting two dominos together like this we can forget about the parts on the "inside" since they're not accessible anymore. We can treat the composition like one big domino:

```
address . streetAddress :: Lens' Person StreetAddress
```

```
+---------------------+-----------------------------+
| Person              |               StreetAddress |
+---------------------+-----------------------------+
```

Now we've got different attachment points where we could add another lens, or even an action!

An action "caps" off a chain of dominos, by performing a modification on the focus of the domino chain. We can only graft an action onto the **right-hand side**, not the left.

```
streetNumber  :: Lens' StreetAddress String
action :: String -> String
```

```
        streetNumber
+---------------+--------+ >------+
| StreetAddress | String |     action
+---------------+--------+ <------+
```

Now for polymorphic optics. Polymorphic optics add a few more type variables, but really they still work a lot like dominos. Polymorphic dominos have separate types for the "pre-action" data and the "post-action" data, we can represent this in our diagram as a domino where the **top-half** consists of the **pre-action** types, and the **bottom-half** has the **post-action** types.

`_2` is a polymorphic lens, so here's what it looks like wired up with an action:

```
_2 :: Lens (a, b) (a, c) b c
action :: (b -> c)

+--------+---+
| (a, b) | b | >------+
+--------+---+     action
| (a, c) | c | <------+
+--------+---+
```

Now let's see a real example of a composition of polymorphic lenses with a real action.

For our example we're working on a small game where the player can collect items, and can craft them from one type to another. We want to keep track of the quantity of material the player in our video-game is holding alongside some other player state. Here are some skeleton types for us to work with:

```
-- Some dead-simple types which represent our game
data Player  = Player  deriving Show
data Wool    = Wool    deriving Show
data Sweater = Sweater deriving Show

data Item a =
    Item { _material :: a
         , _amount   :: Int
         }
    deriving Show
```

This generates the following lenses:

```
material :: Lens  (Item a) (Item b) a b
amount   :: Lens' (Item a) Int
```

Say that the player is holding some `Wool` and wants to turn all their `Wool` into `Sweaters`. They can "craft" the wool into sweaters using the `weave` action, we need to perform the update on our game state.

Here's what we have at our disposal:

```
weave :: Wool -> Sweater
weave Wool = Sweater
gameState :: (Player, Item Wool)
gameState = (Player, Item Wool 5)
```

To turn the `Wool` into a `Sweater` we'll need to dive deep into our `gameState`, focus the `Wool`, and run the `weave` function on it. First we'll see what it looks like in Haskell code and then we'll break it into a diagram.

```
>>> over (_2 . material) weave gameState
(Player,Item {_material = Sweater, _amount = 5})
```

To diagram the composition we'll look at the type of our path:

```
-- The generalized type of our path:
(_2 . material) :: Lens (other, Item a) (other, Item b) a b

-- The path specialized to our specific types
(_2 . material)
  :: Lens (Player, Item Wool) (Player, Item Sweater) Wool Sweater
```

If we turn each of our lenses into dominos we get this beautifully crafted ASCII diagram version, I should really hire an illustrator:

```
                    _2                    .          material
+-------------------------+--------------+ +--------------+---------+
| (Player, Item Wool)     | Item Wool    | | Item Wool    | Wool    | >---+
+-------------------------+--------------+ +--------------+---------+   weave
| (Player, Item Sweater)  | Item Sweater | | Item Sweater | Sweater | <---+
+-------------------------+--------------+ +--------------+---------+
```

Optics compose easily without much boiler-plate, so we should prefer many small precise optics rather than large bulky ones; we'll end up writing a lot less duplicated code this way. It's more of the "functional way", Simon Peyton Jones would be proud.

Even though we've only learned about lenses so far, all optics compose basically this same way. We can even mix & match the different types of optics together. Every new optic we discover unlocks a combinatoric explosion of possibilities! Once we've got a few different types of optics under our belt, each new discovery feels like we're unlocking a new skill-tree in our functional programming adventure!

Exercises – Lens Composition

Jump to answers

1. Fill in the blank with the appropriate composition of tuple lenses in the following statement:

```
>>> view _ ("Ginerva", (("Galileo", "Waldo"), "Malfoy"))
"Waldo"
```

When experimenting with tuple lenses you may get warnings or errors along the lines of:

- `Found hole: _1 :: b1 -> b0`

This is GHC helpfully trying to use the "typed-holes" feature for an identifier which starts with a _; in this case all you need to do is bring the `_1` lens into scope:

```
import Control.Lens
```

2. Given the following lens types, fill in the missing type of `mysteryDomino`

```
fiveEightDomino :: Lens' Five Eight
mysteryDomino   :: Lens' ???? ????
twoThreeDomino  :: Lens' Two Three

dominoTrain :: Lens' Five Three
dominoTrain = fiveEightDomino . mysteryDomino . twoThreeDomino
```

3. Using what you know about how lenses work under the hood; rewrite the following signature as a polymorphic lens of the form: `Lens s t a b`. Then identify each animal as one of: pre-action structure, post-action structure, pre-action focus, post-action focus

```
Functor f => (Armadillo -> f Hedgehog) -> (Platypus -> f BabySloth)
```

4. Find a way to compose ALL of the following lensees together into one big path using each exactly once. What's the type of the resulting lens?

```
spuzorktrowmble   :: Lens Chumble      Spuzz      Gazork       Trowlg
gazorlglesnatchka :: Lens Gazork       Trowlg     Bandersnatch Yakka
zinkattumblezz    :: Lens Zink         Wattoom    Chumble      Spuzz
gruggazinkoom     :: Lens Grug         Pubbawup   Zink         Wattoom
banderyakoobog    :: Lens Bandersnatch Yakka      Foob         Mog
boowockugwup      :: Lens Boojum       Jabberwock Grug         Pubbawup
snajubjumwock     :: Lens Snark        JubJub     Boojum       Jabberwock
```

5. Operators

5.1 Lens Operators

Programming deals with a **lot** of property access and modification, so it makes sense to ensure that it's as convenient and readable as possible. Ideally the syntax for these operations fades into the background, allowing the actual semantics of the code to be more evident. To this end the `lens` library has a huge vocabulary of **infix operators**. This means that instead of using written names for our **actions** like `view` or `set` or `over`, we can instead use symbols like `^.`, `.~` and `%~`. These are very common *in the wild* and you should get used to using optics with both styles.

What's an **infix operator** you ask? Think of `*`, `+` or `<>`! They're just symbols which represent functions that can be written 'infix', i.e. in **between** their arguments. `x * y` is the same as calling the multiplication function with `x` and `y` like this: `(*) x y`.

When first learning about optics the huge variety of operators can be a bit confusing. Many of the operators are quite similar to one another, so it's tough to tell them apart when your understanding is still a bit fuzzy.

If you're ever faced with an operator you don't recognize, Hoogle[1] can help you find it!

Once you get more comfortable working with the operator style it can **really** cut down on the verbosity of common operations. Don't get carried away, but in general the operator notation is preferred when working with optics (at least in Haskell). Here's a quick reference table with the actions we've learned so far and their infix operator names:

Action	Operator	Type
flip view	`^.`	`s -> Lens' s a -> a`
set	`.~`	`Lens s t a b -> b -> s -> t`
over	`%~`	`Lens s t a b -> (a -> b) -> s -> t`

Make special note that the arguments to `view`'s operator `^.` has its arguments flipped!

These new names behave exactly the same as their English counterparts, but they also have the amazing benefit of being 57% harder to google! Wonderful! Let's see what lenses look like in operator-style!

[1] https://hoogle.haskell.org/

5.2 `view` a.k.a. `^.`

Let's define some simple records to use for these examples:

```
data Payload =
    Payload { _weightKilos :: Int
            , _cargo       :: String
            } deriving (Show)
makeLenses ''Payload

data Ship =
    Ship { _payload :: Payload
         } deriving (Show)
makeLenses ''Ship
```

And here's a value we can use:

```
serenity :: Ship
serenity = Ship (Payload 50000 "Livestock")
```

Here's a comparison of viewing a property using each style:

```
>>> view (payload . cargo) serenity
"Livestock"

>>> serenity ^. payload . cargo
"Livestock"
```

Note how the operator precedence of `^.` allows us to drop a pair of brackets in this case!

Some people prefer to drop the spaces around certain operators to look more like Object Oriented property access:

```
>>> serenity^.payload.cargo
"Livestock"
```

Use whichever version helps you sleep at night.

Do robots dream of lens operators with or without superfluous spacing?

5.3 `set` a.k.a. `.~`

Next up is `set`! This one's a bit more interesting, how can we possibly define a three-argument function as an infix operator you ask? Hold my lens... here's the comparison:

```
>>> set (payload . cargo) "Medicine" serenity
Ship
    { _payload = Payload
        { _weightKilos = 50000
        , _cargo = "Medicine"
        }
    }

>>> serenity & payload . cargo .~ "Medicine"
Ship
    { _payload = Payload
        { _weightKilos = 50000
        , _cargo = "Medicine"
        }
    }
```

This one probably deserves a little more breaking down!

First off, here's the type of `.~`, which you'll notice is **exactly** same the as `set`.

```
(.~) :: Lens s t a b -> b -> s -> t

set  :: Lens s t a b -> b -> s -> t
```

So it takes a lens and a new value to set, then lastly accepts a structure and runs the whole operation to return a new structure as a result. When using infix operators in Haskell, the argument on the **left** will be passed to the operator's **first** slot, and the argument on the **right** is passed to the **second** slot, so if we write only the part directly surrounding `.~` we get:

```
payload . cargo .~ "Medicine"
```

Do you remember your order of operations mnemonic from High School (pick one: <BEDMAS, PEMDAS, BODMAS, BIDMAS>)? Now forget it, Haskell has its own rules on operator precedence which helps it know which order to do things. It's called **fixity**!

fixity

The *binding* precedence of an operator. It determines priority when deciding which operator to bind to its arguments next.

Fixity ranges from `0` to `9`. Low numbers mean that other things happen first; e.g. `$` has a fixity priority of `0` which means all other operators will be applied before it! Dot composition: `.` has a fixity priority of `9` so composition is applied before anything else.

There's also the notion of whether operators are right or left associative, but this isn't really something you need to worry about when working with lens operators so I'll leave you to do your own reading on that one.

The composition operator (`.`) has the highest possible operator priority: 9, whereas (`.~`) is only 4; so Haskell will bind the composition first. That means that the following is equivalent to the former; it might help to think of it this way:

```
(payload . cargo) .~ "Medicine"
```

But most people would leave the brackets off, so I will too.

We're applying (`.~`) to its first two arguments: the **lens** and the new **value**. After this it needs one more argument, currying works no differently for operators than for "normal" functions. We're left with a function from the **input structure** to the **output structure**:

```
>>> :t payload . cargo .~ "Medicine"
payload . cargo .~ "Medicine" :: Ship -> Ship
```

We can't simply add the ship on the END of the previous computation:

```
payload . cargo .~ "Medicine" serenity
```

Because Haskell's precedence rules will always try to apply normal function application first, and it sees this as trying to apply the "function" `"Medicine"` to the argument `serenity`, which obviously isn't going to work out so well.

We can use brackets or even `$` to give Haskell a few hints in the hopes that it'll stop being so foolish, here's what those look like:

```
>>> (payload . cargo .~ "Medicine") serenity

>>> payload . cargo .~ "Medicine" $ serenity
```

But the brackets get to be a bit annoying, and we'll see as we go further that the `$` approach gets a bit clumsy when we want to run multiple lens actions in a row, so instead we'll learn yet another infix operator called `(&)`!

`(&)` is defined as flipped function application:

```
(&) :: a -> (a -> b) -> b
(&) = flip ($)
```

If you ever want to use this in projects without importing `Control.Lens`, it's available a'la carte under `Data.Function`.

As it turns out, it's just the right fixity and type signature to solve all of our problems! I suspect this isn't just a coincidence... We can use it to get the precedence right in our operation and write out our operation so it reads like a proper sentence:

```
serenity & payload . cargo .~ "Medicine"
```

I read `&` as 'and-then', and read `.~` as `set`, so if we add a bit of filler we end up with:

> "Take serenity and then, regarding its payload's cargo, set it to Medicine"

That sentence may not be winning a Pulitzer any time soon, but hey, it gets the point across just fine. Now you can read expressions using lens operators in your head.

5.4 Chaining many operations

Our new friend `(&)` has a few more tricks up its sleeve. The fixity works out so that we can use it to chain many operations together. In other languages this is typically called a "pipeline operator". Conceptually we're pushing a value *forwards* through a **pipeline** of operations.

We can use it to chain multiple `set` actions in a row:

```
>>> serenity
      & payload . cargo .~ "Chocolate"
      & payload . weightKilos .~ 2310
Ship
    { _payload = Payload
        { _weightKilos = 2310
        , _cargo = "Chocolate"
        }
    }
```

If you still prefer the traditional action names we can mix and match the two:

```
>>> serenity
      & set (payload . cargo) "Chocolate"
      & set (payload . weightKilos) 2310
Ship
    { _payload = Payload
        { _weightKilos = 2310
        , _cargo = "Chocolate"
        }
    }
```

5.5 Using %~ a.k.a. over

Lastly we've got over! The operator for over is %~. If you need a mnemonic, % is often the **mod**ulo operator, and over **mod**ifies its focus. Apparently this was actually the motivation for its name, lens is a bit cheeky like that. %~ works just like .~ except it takes a **modification function** as its right-hand argument rather than a new value.

We can intersperse uses of %~ and .~ with the & operator without any trouble, let's see what everything looks like all together:

```
>>> serenity
      & payload . weightKilos %~ subtract 1000
      & payload . cargo .~ "Chocolate"
Ship
    { _payload = Payload
        { _weightKilos = 49000
        , _cargo = "Chocolate"
        }
    }
```

5.6 Learning Hieroglyphics

At first glance the collection of available operators may look like an earthquake hit the mechanical keyboard factory, but there's actually a sensible language to the whole thing which starts to make sense after a bit of practice. If you can learn the basics of this language you can usually *guess* the name of a symbol which does what you need, and oftentimes it exists! It's a bit like trying to order a beer in Portuguese when all you know is a bit of Spanish; you can probably get by!

This is a key principle; most people (myself included) don't **memorize** the list of **all** optics combinators, they know about **patterns** and can use those patterns to discover the tools they need. Hoogle helps a lot too!

Let's learn the basics of operators vocabulary; I'm not sure what the optics equivalent of hailing a cab or booking a hotel are, but you should be able to do that after learning a few of these symbols.

Each **operator** can be broken up into smaller **symbols**. In general each symbol is a single character, but sometimes it's two!

You already know `(^.)`; we can break it down into two parts:

`^` *Usually* denotes that the action **views/gets** something
`.` Typically used to represent the **absence** of any other modifiers

So `^.` means "**Get** the **focus** without doing anything else" which is exactly what it does!

Now lets try `(%~)` We can break this into:

`%` Means "**modify**" using a function
`~` Denotes that this action **updates or sets** something

So roughly this means `%~` is a **setter or modifier** which runs a **function**.

It's not a perfect system, so if it seems a bit awkward, that's because *it is*. The more operators you learn, the more sense it'll all make.

There are many more symbols to learn, the `lens` library defines many "short-hand" operators. All of these can be implemented manually using `%~`; but they're a bit more concise. Most of these symbol choices should be pretty self-explanatory:

`+~`, **`-~`**, **`*~`**
: **Add**, **Subtract**, or **Multiply**, a value with the focus

```
>>> (2, 30) & _2 +~ 5
(2, 35)
```

`//~` **Divide** the focus by a provided denominator

```
>>> (2, 30) & _2 //~ 2
(2, 15.0)
```

`^~`, `^^~`, `**~`
: Raise the `Numeric`, `Fractional`, or `Floating` (respectively) focus to the provided exponent. Note that `^~` means "update by raising to a power", it doesn't have anything to do with `view`.

```
>>> (2, 30) & _1 ^~ 3
(8, 30)
```

`||~`, `&&~`
: Logical OR or AND (respectively) of the focus with the value

```
>>> (False, 30) & _1 ||~ True
(True, 30)
```

`<>~` `mappend` a value onto the focus (from the right).

```
>>> ("abra", 30) & _1 <>~ "cadabra"
("abracadabra", 30)
```

5.7 Modifiers

There are a few more symbols which act a bit like Adverbs; they modify the behaviour of an existing operator in some small way. Here's a super small data type we can consider in examples:

```
data Thermometer =
    Thermometer { _temperature :: Int
                } deriving Show

makeLenses ''Thermometer
```

The following can be used as **prefixes** to almost any **setter** operator:

`<` **Get** the altered focus in **addition** to modifying it.

```
>>> Thermometer 20 & temperature <+~ 15
(35, Thermometer {_temperature = 35})
```

See how it **added** 15 to the temperature, returning the **altered structure**, but *ALSO* returned the result of the addition in the tuple? This can be handy if you want to change a nested value but still need the result for another computation. It tends to be a bit more efficient than doing the equivalent update followed by a view.

`<<` **Get** the **OLD** focus in **addition** to setting a new one.

```
>>> Thermometer 20 & temperature <<+~ 15
(20, Thermometer {_temperature = 35})
```

This one adds 15 to the temp like before, but `<<` it returns the **pre-action** focus value rather than the updated one.

5.8 When to use operators vs named actions?

There really aren't any "rules" for this, but some people find it nicer to use the named versions when **partially applying** lens expressions, and use the operator versions the rest of the time. For example, if we wanted to map a `view` action over a list of tuples we would prefer to use the `view` name rather than `^.`:

```
map (view _2) [("Patrick", "Star"), ("SpongeBob", "SquarePants")]
[ "Star" , "SquarePants" ]
```

You **COULD** do it this way, but I think it looks a bit silly:

```
>>> map (^. _2) [("Patrick", "Star"), ("SpongeBob", "SquarePants")]
[ "Star" , "SquarePants" ]
```

The same goes for `set` and `over`:

```
>>> map (over _2 reverse) [("Patrick", "Star"), ("SpongeBob", "SquarePants")]
[("Patrick", "ratS"), ("SpongeBob", "stnaPerauqS")]


>>> map (_2 %~ reverse) [("Patrick", "Star"), ("SpongeBob", "SquarePants")]
[("Patrick", "ratS"), ("SpongeBob", "stnaPerauqS")]
```

I prefer using the full names in this particular situation, but you can do whichever you prefer.

That's it for the basic lens operators, we'll cover more symbol fragments and operators once we've learned about a few more types of optics!

If you ever need just a quick reminder I've included a full cheatsheet for operators in an appendix for reference.

5.9 Exercises – Operators

Jump to answers

1. Consider the following laundry list of types:

```
data Gate =
    Gate { _open    :: Bool
         , _oilTemp :: Float
         } deriving Show
makeLenses ''Gate

data Army =
    Army { _archers :: Int
         , _knights :: Int
         } deriving Show
makeLenses ''Army

data Kingdom =
    Kingdom { _name :: String
```

```
            , _army :: Army
            , _gate :: Gate
            } deriving Show
makeLenses ''Kingdom
```

Given the following starting state:

```
duloc :: Kingdom
duloc =
    Kingdom { _name = "Duloc"
            , _army = Army { _archers = 22
                           , _knights = 14
                           }
            , _gate = Gate { _open    = True
                           , _oilTemp = 10.0
                           }
            }
```

Write a chain of expressions using infix operators to get from the start state to each of the following goal states:

Hard mode: Try doing it without using `%~` or `.~`!

```
>>> goalA
Kingdom
    { _name = "Duloc: a perfect place"
    , _army = Army
        { _archers = 22
        , _knights = 42
        }
    , _gate = Gate
        { _open = False
        , _oilTemp = 10.0
        }
    }
```

```
>>> goalB
Kingdom
    { _name = "Dulocinstein"
    , _army = Army
        { _archers = 17
        , _knights = 26
        }
    , _gate = Gate
        { _open = True
        , _oilTemp = 100.0
        }
    }
```

Bonus: Good luck with this one! Notice the tuple around the finished Kingdom! Also; there definitely aren't any typos! This is your goal. You've got all the tools you need to figure it out. Let's see what you can do.

```
>>> goalC
( "Duloc: Home"
, Kingdom
    { _name = "Duloc: Home of the talking Donkeys"
    , _army = Army
        { _archers = 22
        , _knights = 14
        }
    , _gate = Gate
        { _open = True
        , _oilTemp = 5.0
        }
    }
)
```

2. Enter the appropriate operator in the `undefined` slot to make each code example consistent:

```
>>> (False, "opossums") `undefined` _1 ||~ True
(True, "opossums")
```

Remember that `id` is a lens which focuses the full **structure**

```
>>> 2 & id `undefined` 3
6
```

```
>>> import Data.Char (toUpper)
>>> ((True, "Dudley"), 55.0)
      & _1 . _2 `undefined` " - the worst"
      & _2 `undefined` 15
      & _2 `undefined` 2
      & _1 . _2 `undefined` map toUpper
      & _1 . _1 `undefined` False
((False,"DUDLEY - THE WORST"),20.0)
```

3. Name a lens operator that takes only two arguments
4. What's the type signature of %~? Try to figure it without checking! Look at the examples above if you have to.

6. Folds

6.1 Introduction to Folds

Guess what! We get to learn a brand spankin' new type of optic now! Lenses are OUT, Folds are the new black!

Folds are like queries. You can select subsets of data from all over your structure and collect only the pieces which you're interested in, filtering out the stuff you don't care about.

Now that you've mastered lenses we can describe folds in terms of something you know. The key differences between lenses and folds are that:

- Lenses must focus **ONE** thing, Folds can focus **MANY** things
- Lenses can **get** and **set**, Folds can **only get**.

Okay, so we're dealing with some trade-offs; we can't **set** using a fold, so we can forget about using all those fancy setting actions we just learned (for now). What we gain in return is that we can now focus whatever suits our liking; or focus nothing at all if we don't want to! In fact, folds don't even have any laws; it's the wild-wild-west, if something forms a useful query, ain't nobody can stop us from building it. All together this allows us to write arbitrary filters in small composable pieces to build up a query language for asking complex questions about our structures.

Where does the name come from? Folds are pretty well known in functional programming, you may be familiar with the usual suspects `foldr`, `foldl`, and even `foldMap` or `fold` from `Data.Foldable`; this is the same sort of idea, but lifted into the magical unicorn realm of optics where we have oh-so-wonderful composition!

I like the metaphor of 'folding in' ingredients to pastry dough. Folds focus an arbitrary number of elements (*ingredients* in this metaphor) and combine them all according to some *recipe*. Once you've folded everything together into the dough for a delicious flaky pie crust you might be able to get a sense of which ingredients went in, but your chances of separating it into separate ingredients again are slim-to-none. This helps us understand why we can't **set** things using a fold; all the pieces are merged together haphazardly and we can't tell where they came from, so we couldn't possibly rebuild the structure we started with.

Wow, that was a tangent… I'm going to go get some pie, but when I come back we'll go through an example or two.

Focusing all elements of a container

The pie was delicious by the way; thanks for asking! Let's get to an example.

After a late night recruiting down at "The Drunken Dodo" we've got a full crew roster for our next pirating endeavour! Unfortunately we had a bit too much rum and don't remember who signed up; luckily our assistant "Responsible Redbeard" jotted every recruit down for us, including which role each member signed up for:

```
data Role
    = Gunner
    | PowderMonkey
    | Navigator
    | Captain
    | FirstMate
    deriving (Show, Eq, Ord)

data CrewMember =
    CrewMember { _name    :: String
               , _role    :: Role
               , _talents :: [String]
               }
    deriving (Show, Eq, Ord)
makeLenses ''CrewMember

roster :: S.Set CrewMember
roster = S.fromList
    --           Name               Role         Talents
    [ CrewMember "Grumpy Roger"     Gunner       ["Juggling", "Arbitrage"]
    , CrewMember "Long-John Bronze" PowderMonkey ["Origami"]
    , CrewMember "Salty Steve"      PowderMonkey ["Charcuterie"]
    , CrewMember "One-eyed Jack"    Navigator    []
    ]
```

First things first we need to find out **which roles we've successfully filled**. At this point we don't care *who* filled *which* role, we just want a **set of the fulfilled crew roles** from our roster.

This task involves querying small pieces of information from many places across our structure, i.e. we need to know the **Role** of **each** of the crew members, so that sounds like a fold! Before we start hacking & slashing our way through the data it's a good exercise to think about what we're actually going to want at the end of the day. If we can get that right it's just a matter of filling in the gaps until we have something that type-checks; and if it type-checks it must be correct... right?

Let's look at the type of a typical fold:

```
myFold :: Fold s a
```

Notice the distinct LACK of a trailing tick (`'`); since folds can't be used for **setting**, there aren't any **polymorphic actions** which run on folds. This means we can get away with only one **structure** type and one **focus** type.

This type says that if you give us a **structure** of type `<s>` we can find *zero or more* focuses of type `<a>`. Note that a fold only specifies how to **find** the focuses (*ingredients*), not how to **combine** them. The decision of **how** to mix them all together is left up to the **action**.

In our case we have a `(roster :: S.Set CrewMember)` and we care about the `Roles` of each crew member, so we probably want to end up with a fold like this once we're done:

```
rosterRoles :: Fold (S.Set CrewMember) Role
```

This says that the `rosterRoles` fold can focus **zero or more** `Roles` if you give it a `(S.Set CrewMember)`.

Take note that we specify the focus as `Role` and **not** `[Role]`. All folds focus **zero or more** focuses so it would be redundant to denote that by using a list.

So how do we actually build a fold like this? As you know, optics are all about **composition** and folds are no different. We can build this final fold type by composing smaller folds.

Here's how I suggest we break down this particular problem:

A. First focus each individual `CrewMember` in the roster
B. Then focus the `Role` within each of those `CrewMembers`
C. Run some action to collect all of these focuses from the roster.

If we write each of these steps as folds we get the following:

A. `crewMembers :: Fold (S.Set CrewMember) CrewMember`
B. `crewRole :: Fold CrewMember Role`
C. We'll need an action something like this:

```
toListSomehow :: Fold (S.Set CrewMember) Role -> S.Set CrewMember -> [Role]
```

If we can somehow define these smaller pieces we can use composition to combine steps `A` and `B`. This gets us the fold we need to focus `Roles` from a set of `CrewMembers`. After that we just need to run the action for collecting the focuses into a list.

Decomposing Problems

Decomposing complicated problems into several smaller ones is a key aspect of learning to be effective with optics. Typically it's as easy as thinking about the different **steps** you take to drill into a data-structure towards the focus you want and finding or building the optic which takes each step.

In certain cases it's a bit more complicated, but **most** steps have some combinator or helper within the `lens` library that will solve your problem in less than a line of code! The examples in this book should help to introduce you to these helpers as you need them, but when in doubt it's worth taking a bit of time to dig through the many options available in the library. Good luck!

Collapsing the Set

Let's think about the first step:

```
crewMembers :: Fold (S.Set CrewMember) CrewMember
```

This operation is pretty simple conceptually, given a *set* of *things*, we want to **focus** on each *thing*. Surely this is a common enough task that there's already a helper for this!

Indeed there is, introducing `folded`:

```
folded :: Foldable f => Fold (f a) a
```

This is a **generalized** version of what we want to do; it takes **ANY** `Foldable` container as a structure and will **focus** each element inside it. We can use this same helper when we want to get all the values from a Set, List, Map, or even Tree! That's a pretty great value proposition for typing only 6 characters!

In our case we can specialize `<f>` to be `Set`, which is Foldable.

```
crewMembers :: Fold (S.Set CrewMember) CrewMember
crewMembers = folded
```

We of course don't even need to write this definition if we don't want to; `folded` works just fine in-line.

Before we move on to the next step we should definitely check that what we've got so far is working as expected. When we tested out lenses we could test them with **view** to see what they focused. Let's learn about the action which does the equivalent for folds.

Collecting focuses as a list

Lenses must focus exactly one value, whereas folds can focus many. Similarly, `view` (a.k.a. `(^.)`) always retrieves exactly **one** thing, but `toListOf` (a.k.a. `(^..)`) retrieves **zero or more** values in a list!

> You **can** actually use `view` on a fold; but it doesn't do quite what you'd expect so I recommend avoiding it for now. If you're too curious to let it pass you by; this is what it looks like if you DO try to use `(^.)` on this particular fold:
>
> ```
> >>> roster ^. crewMembers
>
> <interactive>:1:11: error:
>     • No instance for (Monoid CrewMember)
>         arising from a use of ‘crewMembers’
>     • In the second argument of ‘(^.)’, namely ‘crewMembers’
>       In the expression: roster ^. crewMembers
>       In an equation for ‘it’: it = roster ^. crewMembers
> ```
>
> GHC is complaining that we're missing a Monoid instance for our focus type! Because `view` needs to get a single value out it will use `mempty` if no focuses are found; and if there are many it will use their Semigroup (`<>`) to squish them all into one thing! Generally this behaviour is a bit confusing; so I'd usually recommend you avoid it; you can do this sort of fold more explicitly using `foldOf` if you desire it.

So, we've got the `toListOf` action! It's got an operator alias too: `(^..)`; which is just like `(^.)` but, with like, one more dot... This action return a **list** of focuses rather than a single **focus**.

Here are the types:

```haskell
toListOf :: Fold s a -> s -> [a]
-- a.k.a.
(^..) :: s -> Fold s a -> [a]
```

Now that we've got an **action** and a **fold** let's try using them together to see how `folded` works on the Set:

```
>>> -- a.k.a.  roster ^.. folded
>>> toListOf folded roster
[ CrewMember
    { _name = "Grumpy Roger"
    , _role = Gunner
    , _talents =
        [ "Juggling"
        , "Arbitrage"
        ]
    }
, CrewMember
    { _name = "Long-John Bronze"
    , _role = PowderMonkey
    , _talents = [ "Origami" ]
    }
, CrewMember
    { _name = "One-eyed Jack"
    , _role = Navigator
    , _talents = []
    }
, CrewMember
    { _name = "Salty Steve"
    , _role = PowderMonkey
    , _talents = [ "Charcuterie" ]
    }
]
```

We can see what was **focused** by the fold by seeing which values end up in the list. In this case we can see it focused each element of the Set. We don't really want to end up with a list of `CrewMembers` at the end of all of this; so we'll need to focus in further.

`folded` is a pretty important optic for working with folds, so before moving on let's take a quick aside to try out `folded` and `toListOf` on some other data structures, remember that it works on anything that implements the `Foldable` typeclass.

```
-- `Maybe` is Foldable!
>>> Just "Buried Treasure" ^.. folded
[ "Buried Treasure" ]

-- `folded` might focus zero elements
>>> Nothing ^.. folded
[]

>>> Identity "Cutlass" ^.. folded
[ "Cutlass" ]

-- Remember that the Foldable instance of tuple only focuses the right-hand value
>>> ("Rubies", "Gold") ^.. folded
[ "Gold" ]

-- Folding a Map focuses only the values not the keys
>>> M.fromList [("Jack", "Captain"), ("Will", "First Mate")]
      ^.. folded
[ "Captain"
, "First Mate"
]
```

Hopefully that clears up `folded` and `(^..)`/`toListOf` a little bit!

Using lenses as folds

Where did we leave off again? Ahh right! `folded` lets us focus each `CrewMember` from within our `S.Set CrewMember`, but we still need a fold to focus the `Role` from the `CrewMember`: (Fold CrewMember Role)‘.

Recalling the `CrewMember` data structure:

```
data CrewMember =
    CrewMember { _name    :: String
               , _role    :: Role
               , _talents :: [String]
               }
    deriving Show
makeLenses ''CrewMember
```

We can see that our fold essentially just needs to focus the `_role` field of the record.

We’ve even used `makeLenses ''CrewMember` above, so incidentally we already have `role :: Lens' CrewMember Role` sitting around. Ready for a mind-blower? Lenses can be used directly as folds!

```
crewRole :: Fold CrewMember Role
crewRole = role
```

This will work just fine! We don't actually need to re-define it like this of course, we can just use the `role` lens directly as a fold in our path if we like, something like this:

```
>>> let jerry = CrewMember "Jerry" PowderMonkey ["Ice Cream Making"]
>>> jerry ^.. role
[PowderMonkey]
```

We can see that the `role` lens focuses Jerry's role. How does the lens know how to fold? Remember that every lens has a **getter** inside it which knows how to get exactly **one** focus from the **structure**. **One** element *technically* fits within the range of **zero or more** focuses required to be a fold, so it uses the getter and it all works out! No conversions, adapters, or modifications required! You can drop in a lens anywhere you need a fold.

When we use a lens as a fold we can mentally substitute the types like this:

```
Lens' s a
-- becomes
Fold s a
```

Composing folds

We've got all the pieces we need now; we just need to fit them all together!

Folds compose exactly like lenses do, so let's combine the folds we've discovered and see what they focus!

```
>>> roster ^.. folded . role
[ Gunner
, PowderMonkey
, Navigator
, PowderMonkey
]
```

The *specialized* types of the folds in this expression are:

```
folded :: Fold (S.Set CrewMember) CrewMember
role   :: Fold CrewMember Role
```

Great! We've collapsed our `S.Set CrewMember` into the `[Role]` we wanted by composing folds to dive into our structure to select the pieces we care about!

Foundational fold combinators

Before continuing to exercises I'm going to introduce a couple more ways of folding down common structures. First we'll look at `both`!

```
both :: Bitraversable r => Traversal (r a a) (r b b) a b
```

You'll notice this is actually a `Traversal`. `Traversals` are a bit more complex than folds, and we'll cover them in detail in their own chapter. So for now you just need to know that every `Traversal` is also a valid fold. Assuming that, we can simplify the signature in our heads to this:

```
both :: Bitraversable r => Fold (r a a) a
```

We still have this crazy `Bitraversable` thing here though; what's up with that? A `Bitraversable` structure is like a `Traversable` structure, but it can be Traversed over two different type parameters! The simplest types like this are tuples: `(a, b)`, and eithers: `(Either a b)`. Both of these are traversable structures which have two type parameters. Long story short, `both` allows us to fold over both parameters *when the parameters are the same*. Let's see some examples:

```
-- `both` on tuples focuses both at once
>>> ("Gemini", "Leo") ^.. both
["Gemini","Leo"]

-- `both` on an Either type focuses whichever side is present
>>> Left "Albuquerque" ^.. both
["Albuquerque"]

>>> Right "Yosemite" ^.. both
["Yosemite"]

-- Only the last two type params of a tuple are 'bitraversable'
>>> ("Gemini", "Leo", "Libra") ^.. both
["Leo","Libra"]
```

That last tuple example seems a bit quirky. What if we want the same idea, but over arbitrary sized tuples? There's a combinator for that called `each`!

```
each :: Each s t a b => Traversal s t a b
-- Simplified:
each :: Each s s a a => Fold s a
```

This is another traversal which we'll use as a fold for now. It comes with its own typeclass, but not to worry, the implementors of `lens` have done the bulk of the hard work. Each is a typeclass with a different implementation for many different structure types. Most type with a sensible instance already have one defined, so if you can imagine breaking a type apart into many pieces which all have the same type then there's possibly an `Each` instance for that. Here are some examples to get you started:

```
-- There's an `Each` instance for all reasonable sizes of tuples
>>> (1, 2, 3, 4, 5) ^.. each
[1, 2, 3, 4, 5]

-- Selects each element of a list
>>> [1, 2, 3, 4, 5] ^.. each
[1, 2, 3, 4, 5]

-- Folds over each character in a chunk of text Text
>>> ("Made him an offer he couldn't refuse" :: T.Text) ^.. each
"Made him an offer he couldn't refuse" :: String

-- Folds over each Word8 in a ByteString
>>> ("Do or do not" :: BS.ByteString) ^.. each
[68,111,32,111,114,32,100,111,32,110,111,116]
```

`each` can be a bit confusing to use since it behaves slightly differently for each type, but it almost always does the "Right Thing"™, so it's usually worth a try.

We've covered a lot of ground, let's put our skills to the test!

Exercises – Simple Folds

Jump to answers

1. What's the result of each expression? Make sure to guess before trying it out in the repl!

```
beastSizes :: [(Int, String)]
beastSizes = [(3, "Sirens"), (882, "Kraken"), (92, "Ogopogo")]

>>> beastSizes ^.. folded

>>> beastSizes ^.. folded . folded

>>> beastSizes ^.. folded . folded . folded
```

```
>>> beastSizes ^.. folded . _2

>>> toListOf (folded . folded) [[1, 2, 3], [4, 5, 6]]

>>> toListOf
      (folded . folded)
      (M.fromList [("Jack", "Captain"), ("Will", "First Mate")])

>>> ("Hello", "It's me") ^.. both . folded

>>> ("Why", "So", "Serious?") ^.. each

quotes :: [(T.Text, T.Text, T.Text)]
quotes = [("Why", "So", "Serious?"), ("This", "is", "SPARTA")]

>>> quotes ^.. each . each . each
```

2. Write out the 'specialized' type for each of the requested combinators used in each of the following expressions.

`folded` and `_1`

```
>>> toListOf (folded . _1) [(1, 'a'), (2, 'b'), (3, 'c')]
[1, 2, 3]
```

`folded`, `_2`, and `toListOf`

```
>>> toListOf (_2 . folded) (False, S.fromList ["one", "two", "three"])
["one", "two", "three"]
```

`folded` and `folded` and `toListOf`

```
>>> toListOf
      (folded . folded)
      (M.fromList [("Jack", "Captain"), ("Will", "First Mate")])
"CaptainFirst Mate"
```

3. Fill in the blank with the appropriate fold to get the specified results

```
>>> [1, 2, 3] ^.. _
[1, 2, 3]

>>> ("Light", "Dark") ^.. _
["Light"]

>>> [("Light", "Dark"), ("Happy", "Sad")] ^.. _
["Light","Dark","Happy","Sad"]

>>> [("Light", "Dark"), ("Happy", "Sad")] ^.. _
["Light","Happy"]

>>> [("Light", "Dark"), ("Happy", "Sad")] ^.. _
"DarkSad"

>>> ("Bond", "James", "Bond") ^.. _
["Bond","James","Bond"]
```

6.2 Custom Folds

So far we've learned about `folded` which folds over the contents of any `Foldable` container. Sometimes however, the data we want to access isn't in one of the stock container-types; maybe it's in a record, or maybe it's a part of the data-structure other than the 'Foldable' bit. E.g. what if we want to access all the keys of a map instead of the values?

Luckily for us this is still quite easy to do!

Continuing on the Pirate motif for some reason, I'll define this data structure representing a ship's crew:

```
newtype Name = Name
    { getName :: String
    } deriving Show

data ShipCrew = ShipCrew
    { _shipName :: Name
    , _captain    :: Name
    , _firstMate  :: Name
    , _conscripts :: [Name]
    } deriving (Show)
makeLenses ''ShipCrew
```

Our task is: given a `ShipCrew`, list out every crew member's name. Unfortunately this time our ship's crew is defined in a slightly unorthodox way! Our `ShipCrew` structure is unfortunately **not** **`Foldable`**

since there's no **type parameter**! If we want to list out our whole crew we'll need to find some other way.

To make things even trickier, it seems that the captain and first mate have been given special treatment (typical!) and are being stored separately from the rest of the crew. How can we focus every crew member's name from this weird layout?

When working with lenses we had the `lens` helper which allowed us to define arbitrary lenses; the equivalent helper for folds is called `folding`.

```
folding :: Foldable f => (s -> f a) -> Fold s a
```

This helper takes a single projection function and returns a fold. The projection function allows us to manually specify a way to fold our type into the focuses we want. It's okay if the structure itself isn't `Foldable`, we only need to be able to **project** the pieces we want to focus into something that *is* `Foldable`.

Although the projection function is polymorphic over the type of `Foldable` it returns, typically we just use a list, however you could use a Set or Map or whatever if you really wanted to. If we write a function which collects all our crew members into a list then we can use `folding` to convert it into a right proper fold!

```
collectCrewMembers :: ShipCrew -> [Name]
collectCrewMembers crew =
    [_captain crew, _firstMate crew] ++ _conscripts crew

crewMembers :: Fold ShipCrew Name
crewMembers = folding collectCrewMembers
```

Our `collectCrewMembers` function simply dumps all the crew members names into a flat list, since the `_conscripts` field *already* contains a list we just append it directly at the end. We then pass the function to `folding` to create a fold which focuses these names.

```
myCrew :: ShipCrew
myCrew =
    ShipCrew
    { _shipName   = Name "Purple Pearl"
    , _captain    = Name "Grumpy Roger"
    , _firstMate  = Name "Long-John Bronze"
    , _conscripts = [Name "One-eyed Jack", Name "Filthy Frank"]
    }

>>> myCrew ^.. crewMembers
[ Name { getName = "Grumpy Roger" }
```

```
, Name { getName = "Long-John Bronze" }
, Name { getName = "One-eyed Jack" }
, Name { getName = "Filthy Frank" }
]
```

This idea works in general; if you write a function which somehow gets your data into a `Foldable` structure, you can build it into a custom fold using `folding`.

Our names are still in the `Name` newtype wrapper however, it'd be nice to get at the underlying strings instead! We could of course `fmap` over the list of names inside `collectCrewMembers`, but that breaks us out of our beautifully composable fold. This is an optics book after all, let's see what we can do.

Mapping over folds

So we've got a fold which selects many `Names`, but we'd love to modify it to focus the underlying `String` instead.

When faced with a problem like this it's common for people to want to *edit* the optics they already have, for instance changing `collectCrewMembers` to unpack the `Names` for us when dumping them into the list. This seems like a good idea at the time, but remember the optics design principals:

> Prefer **many small** and **precise** combinators which can be composed in different combinations to solve many different problems.

It's much more **composable** and **reusable** to write small new optics and chain them together to get the behaviour you want. In this case, instead of editing our `crewMembers` fold to return `Strings`; we'll *chain* on a new fold to do the unpacking instead.

The idea in this case is that we want to `map` over the end of our fold somehow; and there's a function exactly for this purpose!

```
to :: (s -> a) -> Fold s a
```

The `to` helper is pretty simple, it converts a function directly into a fold! You can think of it as `mapping` over your fold and converting every element that passes through it. Let's try it out all on its own to see how it works.

Technically `to` is actually a `Getter` rather than a fold, a `Getter` is just a fold which has this 1-to-1 mapping property, it's basically the "getter" half of a lens. A Getter can ALWAYS transform an input into an output. A pure function `s -> a` shouldn't ever fail, so we can make this stronger guarantee. Since we're guaranteed an output from `to` we can use it with `view` or `^.` directly:

```
>>> Name "Two-faced Tony" ^. to getName
"Two-faced Tony"

-- We can chain many `to`s in a row
>>> import Data.Char (toUpper)
>>> Name "Two-faced Tony" ^. to getName . to (fmap toUpper)
"TWO-FACED TONY"

-- Or simply use function composition before passing to `to`
-- However, I find it confusing to switch from reading
-- left-to-right into right-to-left like this:
>>> Name "Two-faced Tony" ^. to (fmap toUpper . getName)
"TWO-FACED TONY"
```

`to` allows us to easily interleave function transformations into a path of composed optics.

Now let's try it on our example:

```
>>> myCrew ^.. crewMembers . to getName
[ "Grumpy Roger"
, "Long-John Bronze"
, "One-eyed Jack"
, "Filthy Frank"
]
```

Great! It unpacks each `Name` from its newtype wrapper.

Combining multiple folds on the same structure

Sometimes we have existing folds on a structure and we want to use many of them at once. To accomplish this we need to write a new fold which combines them together. We'll need to write a new fold which combines each of the individual folds.

We can use each of the smaller folds inside a call to `folding` to build up a bigger fold. Let's see another way we could build a fold over all crew members. Here's a reminder of what our structure looks like:

```
data ShipCrew = ShipCrew
    { _shipName :: Name
    , _captain    :: Name
    , _firstMate  :: Name
    , _conscripts :: [Name]
    } deriving (Show)
makeLenses ''ShipCrew
```

So we've got the following lenses in scope:

```
shipName :: Lens' ShipCrew Name
captain :: Lens' ShipCrew Name
firstMate :: Lens' ShipCrew Name
conscripts :: Lens' ShipCrew [Name]
```

Every lens is a valid fold! Here's how we can combine folds together to make a larger fold:

```
crewNames :: Fold ShipCrew Name
crewNames =
  folding (\s -> s ^.. captain
              <> s ^.. firstMate
              <> s ^.. conscripts . folded)
```

This simply gets the elements of each fold as a list, then concatenates all those lists together where they'll be handled by the call to `folding`.
It's not much different from the first version, but remember that we can combine any folds or lenses this way, even ones defined in libraries or other modules, and this works even if we don't have access to the field accessors of the record.

We end up with what we'd expect:

```
>>> myCrew ^.. crewNames . to getName
[ "Grumpy Roger"
, "Long-John Bronze"
, "One-eyed Jack"
, "Filthy Frank"
]
```

Exercises – Custom Folds

Jump to answers

1. Fill in each blank with either `to`, `folded`, or `folding`.

```
>>> ["Yer", "a", "wizard", "Harry"] ^.. folded . _
"YerawizardHarry"

>>> [[1, 2, 3], [4, 5, 6]] ^.. folded . _ (take 2)
[1, 2, 4, 5]

>>> [[1, 2, 3], [4, 5, 6]] ^.. folded . _ (take 2)
[[1,2], [4,5]]

>>> ["bob", "otto", "hannah"] ^.. folded . _ reverse
["bob", "otto", "hannah"]

>>> ("abc", "def") ^.. _ (\(a, b) -> [a, b]). _ reverse . _
"cbafed"
```

2. Fill in the blank for each of the following expressions with a **path** of folds which results in the specified answer. Avoid partial functions and `fmap`.

```
>>> [1..5] ^.. _
[100,200,300,400,500]

>>> (1, 2) ^.. _
[1, 2]

>>> [(1, "one"), (2, "two")] ^.. _
["one", "two"]

>>> (Just 1, Just 2, Just 3) ^.. _
[1, 2, 3]

>>> [Left 1, Right 2, Left 3] ^.. _
[2]

>>> [([1, 2], [3, 4]), ([5, 6], [7, 8])]
      ^.. _
[1, 2, 3, 4, 5, 6, 7, 8]

>>> [1, 2, 3, 4] ^.. _
[Left 1, Right 2, Left 3, Right 4]

>>> [(1, (2, 3)), (4, (5, 6))] ^.. _
[1, 2, 3, 4, 5, 6]
```

```
>>> [(Just 1, Left "one"), (Nothing, Right 2)]
       ^.. _
[1, 2]

>>> [(1, "one"), (2, "two")] ^.. _
[Left 1, Right "one", Left 2, Right "two"]

>>> S.fromList ["apricots", "apples"] ^.. _
"selppastocirpa"
```

3. **BONUS** – Devise a fold which returns the expected results. Think outside the box a bit.

```
>>> [(12, 45, 66), (91, 123, 87)] ^.. _
"54321"

>>> [(1, "a"), (2, "b"), (3, "c"), (4, "d")] ^.. _
["b", "d"]
```

6.3 Fold Actions

We've already seen `toListOf` which collects all the focused items into a list; but there are a ton of other actions we can perform on folds! These actions represent **queries** we can ask of our structures. We can use a fold to select the parts of the structure we care about, then ask questions about it.

Some questions we can answer with fold actions include:

- Which focuses match this **predicate**?
- What's the **largest** element in my structure
- What's the result of running this **side-effect** on every focus?
- What's the **sum** of these numeric focuses?
- Does this fold focus **any** elements?
- Does this **specific value** exist in my structure?

So many options, this really just scratches the surface.

Writing queries with folds

A rule of thumb when looking for which action to use on a fold: think of the function you'd use on normal ol' Haskell list for the same purpose, then just add the suffix `-Of`! Need the total of all elements in your fold? With a list you'd use `sum`; so for the fold you'll use `sumOf`! Want to find the smallest element? On a list you'd use `minimum` so for a fold you'd use `minimumOf`!

Most of these behave as though you've collected your fold into a list and then run the respective operation; but they tend to optimize performance a little better. That's why if you look at the *actual* types of these operators you'll notice some strange things. For instance look at the actual type of `sumOf`:

```
sumOf :: Num a => Getting (Endo (Endo a)) s a -> s -> a
```

It's really really not important to actually know what a `Getting (Endo (Endo a))` is; I don't personally know the *actual* types of most of these actions. What I **DO** know though is that when I see a `Getting (Some Crazy Type) s a` I know I can substitute nearly any optic into that slot, including a `(Fold s a)` or a `(Lens' s a)`. The "behind the scenes" types can unify themselves with the `(Some Crazy Type)` portion of a `Getter`; so usually I mentally just substitute any `Getting _ s a` with a `Fold s a`.

That's a lot of words, let's see a lot of examples instead. There are a LOT of these actions, and they're mostly named in a straightforward way, so we're going to go through them pretty quickly! Each section will just have a simplified type signature and some examples. We'll make up for the simplicity here with a more in-depth example later on.

Note, I'll be demoing most of these folds by folding over a simple list since it makes it the easiest to understand, but you can pass any arbitrarily complex fold to these actions.

Does my fold contain a given element?

```
elemOf :: Eq a => Fold s a -> a -> s -> Bool
```

```
>>> elemOf folded 3 [1, 2, 3, 4]
True
>>> elemOf folded 99 [1, 2, 3, 4]
False
```

Do ANY focuses match a predicate?

```
anyOf :: Fold s a -> (a -> Bool) -> s -> Bool
```

```
>>> anyOf folded even [1, 2, 3, 4]
True
>>> anyOf folded (>10) [1, 2, 3, 4]
False
```

Do ALL focuses match a predicate?

```
allOf :: Fold s a -> (a -> Bool) -> s -> Bool
```

```
>>> allOf folded even [1, 2, 3, 4]
False
>>> allOf folded (<10) [1, 2, 3, 4]
True
```

Find the first element matching a predicate

```
findOf :: Fold s a -> (a -> Bool) -> s -> Maybe a
```

```
>>> findOf folded even [1, 2, 3, 4]
Just 2

>>> findOf folded (>10) [1, 2, 3, 4]
Nothing
```

Does my fold have any elements or not?

```
has    :: Fold s a -> s -> Bool
hasn't :: Fold s a -> s -> Bool
```

```
>>> has folded []
False
>>> has folded [1, 2]
True

>>> hasn't folded []
True
>>> hasn't folded [1, 2]
False
```

How many focuses are there?

```
lengthOf :: Fold s a -> s -> Int
```

```
>>> lengthOf folded [1, 2, 3, 4]
4
```

What's the sum or product of my focuses?

```
sumOf     :: Num n => Fold s n -> s -> n
productOf :: Num n => Fold s n -> s -> n

>>> sumOf folded [1, 2, 3, 4]
10

>>> productOf folded [1, 2, 3, 4]
24
```

What's the first or last focus?

`firstOf`, `preview`, and `^?` are all effectively equivalent; use whichever you like.

```
firstOf :: Fold s a -> s -> Maybe a
preview :: Fold s a -> s -> Maybe a
(^?)    :: s -> Fold s a ->  Maybe a

lastOf :: Fold s a -> s -> Maybe a

>>> firstOf folded []
Nothing
>>> firstOf folded [1, 2, 3, 4]
Just 1
>>> preview folded [1, 2, 3, 4]
Just 1
>>> [1, 2, 3, 4] ^? folded
Just 1

>>> lastOf folded [1, 2, 3, 4]
Just 4
```

Find the minimum or maximum focus

```
minimumOf :: Ord a => Fold s a -> s -> Maybe a
maximumOf :: Ord a => Fold s a -> s -> Maybe a

>>> minimumOf folded [2, 1, 4, 3]
Just 1
>>> maximumOf folded [2, 1, 4, 3]
Just 4

>>> minimumOf folded []
Nothing
>>> maximumOf folded []
Nothing
```

Queries case study

That's a lot of information without much context; let's try these actions out with a more concrete example.

As an aspiring TV Show critic I've amassed an incredibly comprehensive collection of facts about a vast assortment (2) of TV shows:

```
data Actor =
    Actor { _name      :: String
          , _birthYear :: Int
          } deriving (Show, Eq)
makeLenses ''Actor

data TVShow =
    TVShow { _title        :: String
           , _numEpisodes  :: Int
           , _numSeasons   :: Int
           , _criticScore :: Double
           , _actors       :: [Actor]
           } deriving (Show, Eq)

makeLenses ''TVShow

howIMetYourMother :: TVShow
howIMetYourMother = TVShow
    { _title = "How I Met Your Mother"
    , _numEpisodes = 208
    , _numSeasons = 9
    , _criticScore = 83
```

```
    , _actors =
        [ Actor "Josh Radnor" 1974
        , Actor "Cobie Smulders" 1982
        , Actor "Neil Patrick Harris" 1973
        , Actor "Alyson Hannigan" 1974
        , Actor "Jason Segel" 1980
        ]
    }

buffy :: TVShow
buffy = TVShow
    { _title = "Buffy the Vampire Slayer"
    , _numEpisodes = 144
    , _numSeasons = 7
    , _criticScore = 81
    , _actors =
        [ Actor "Sarah Michelle Gellar" 1977
        , Actor "Alyson Hannigan" 1974
        , Actor "Nicholas Brendon" 1971
        , Actor "David Boreanaz" 1969
        , Actor "Anthony Head" 1954
        ]
    }

tvShows :: [TVShow]
tvShows = [ howIMetYourMother
          , buffy
          ]
```

Impressive, no?

Let's practice our folding prowess by running some queries over the collection!

We'll start off easy and find the total number of episodes in my collection.

```
>>> sumOf (folded . numEpisodes) tvShows
352
```

Easy! We focus each of the shows using `folded`, then focus on the number of episodes in each using the `numEpisodes` lens, then use `sumOf` to aggregate all those focuses into the total.

Next let's see what the best critic score in the collection is:

```
>>> maximumOf (folded . criticScore) tvShows
Just 83.0
```

83% on Rotten Tomatoes; not too bad but I've seen better.

What if we want to know which show actually has this score? There's another action called `maximumByOf` which allows us to score parts of our fold by some comparison function, just like `maximumBy`.

```
maximumBy   :: Foldable t => (a -> a -> Ordering) -> t a -> a
maximumByOf :: Fold s a   -> (a -> a -> Ordering) -> s   -> Maybe a
```

The type signature highlights the fact that this combinator works effectively the same as the one in `base`; but allows you to provide your folding behaviour a'la carte as a fold.

Notice that the `lens` version of this combinator is actually safer than the one in `base`, it returns a `Nothing` if no elements are found whereas `maximumBy` will throw an error.

Let's get the `TVShow` which has the best critic score:

```
>>> import Data.Ord (comparing)
>>> _title <$> maximumByOf folded (comparing _criticScore) tvShows
Just "How I Met Your Mother"
```

We find the maximum show by comparing them on their critic scores, then `fmap` into the `Maybe` result to get just the title since that's all we care about.

We can use the same trick to find the oldest actor from any of our shows!

```
>>> minimumByOf (folded . actors . folded) (comparing _birthYear) tvShows
Just
    ( Actor
        { _name = "Anthony Head"
        , _birthYear = 1954
        }
    )
```

This more effectively shows the usefulness of folds for this sort of task. It's still a bit annoying to specify the comparator as a function rather than another optic, but it's easy enough to fix; we can write a new combinator which solves this for us! Let's try it just for fun:

```
comparingOf :: Ord a => Lens' s a -> s -> s -> Ordering
comparingOf l = comparing (view l)
```

`comparingOf` accepts a lens and will return the sort of comparator function that actions like `minimumByOf` expect.

I think this is the first time we've passed an optic as an argument to a function we've written ourselves. Optics are just values, we can pass them around to functions if we like. Let's see an Object Oriented language do THAT with dot-notation.

This allows us to write the previous expression like this instead:

```
>>> minimumByOf (folded . actors . folded) (comparingOf birthYear) tvShows
Just
    ( Actor
        { _name = "Anthony Head"
        , _birthYear = 1954
        }
    )
```

Which is a little nicer I think! Note that this accepts a Lens rather than a fold for the comparison, what would it mean to compare many results? What if we compare with something which has no focuses for the comparator? A lens is simpler to deal with in this case. Try rewriting it to use a fold instead and see how it behaves!

Folding with effects

Next we want to print out each actor and their age; printing requires `IO` effects of course! If you're familiar with `Data.Foldable` you'll probably know of `for_`, `traverse_`, and `mapM_` for handling effects. Each of these functions does the same thing but with different argument orderings. `mapM_` has a `Monad` constraint instead of `Applicative`, an unfortunate result of the fact that `Applicative` was discovered after `Monad`, so I recommend sticking with `for_` and `traverse_` when you can.

```
traverse_ :: (Foldable t, Applicative f) => (a -> f b) -> t a -> f ()

for_      :: (Foldable t, Applicative f) => t a -> (a -> f b) -> f ()
```

Let's see what the optics versions of these functions look like:

```
traverseOf_ :: Functor f => Fold s a -> (a -> f r) -> s -> f ()

forOf_ :: Functor f => Fold s a -> s -> (a -> f r) -> f ()
```

> It may seem strange to see the `Functor f` constraint rather than an `Applicative f` constraint like you might expect; this is a result of the way the `lens` library was engineered; it allows you to benefit from relaxed constraints if your optic has stronger guarantees. For instance, we don't need the `Applicative` constraint if we pass these functions a `Lens' s a` because we *know* that we'll focus exactly one argument. If you pass a `Fold s a` instead, then as constraints propagate you'll eventually find yourself needing an `Applicative f` constraint. This can be a bit strange or even annoying, but it's a trade-off between having a flexible interface and having clear types. It may not be the clearest option, but it's what we have to work with, just know that there are legitimate reasons for the way it was designed.

In this case we just want to print everything out, which means we only care about the **effects** and not the **result**. This is good because as you can see, these functions clobber any results! In order to edit things in place we need a *traversal* instead of a *fold* and we'll learn about those soon.

Okay, enough jibber-jabber, let's start printing everyone's age. Judging from what everyone tells me about writing a book it'll probably take another 10 years to actually get this book published, so I'm going to plan ahead and pretend it's the year 2030. We'll need a way to calculate everyone's age from their birth year.

```
calcAge :: Actor -> Int
calcAge actor = 2030 - _birthYear actor

showActor :: Actor -> String
showActor actor =  _name actor <> ": " <> show (calcAge actor)
```

With these in place we can show and print every actor, sequencing all the effects together with `traverseOf_`:

```
>>> traverseOf_ (folded
                 . actors
                 . folded
                 . to showActor)
                 putStrLn
                 tvShows
Josh Radnor: 56
Cobie Smulders: 48
Neil Patrick Harris: 57
Alyson Hannigan: 56
Jason Segel: 50
Sarah Michelle Gellar: 53
Alyson Hannigan: 56
```

```
Nicholas Brendon: 59
David Boreanaz: 61
Anthony Head: 76
```

We can do this for other types of effects too! If you wanted to be extremely eccentric and deliberately obtuse we could count the number of shows in our collection like this:

```
>>> import Control.Monad.State
>>> execState (traverseOf_ folded (modify . const (+1)) tvShows) 0
2
```

This is a silly example of using the State effect, but oftentimes it's handy to build up a stateful computation from the focuses of a fold.

Combining fold results

When I think about folds my mind also tends to drift towards `Monoids`. If you haven't gotten around to learning about `Monoids` yet that's okay, but they're really very handy! If it's a concept that's foreign to you I'd recommend you take an hour or two to read up about Monoids and Semigroups[1] and experiment with them a bit. I promise it will serve you well throughout the rest of this book, but also throughout the rest of your time with functional programming in general.

To summarize, folds are all about collecting pieces of things and Monoids are all about combining things together. This allows them to work together. We can find many focuses within a structure, then combine the pieces together using a Monoid.

It was hidden behind the scenes up until now, but almost ALL of the fold actions we've been using so far (e.g. `maximumOf`, `lengthOf`, `anyOf`, `findOf`, etc...) are all implemented using Monoids! The `lens` library also exposes this power to use as we please. `Data.Foldable` has the `fold` and `foldMap` functions which collapse `Foldable` structures using Monoids, so it's only natural that there are also `foldOf` and `foldMapOf` functions.

Here are two new actions for our folding toolbox:

```
foldOf    :: Monoid a => Fold s a -> s -> a
foldMapOf :: Monoid r => Fold s a -> (a -> r) -> s -> r

-- Their *real* signatures:
foldOf    :: Getting a s a -> s -> a
foldMapOf :: Getting r s a -> (a -> r) -> s -> r
```

[1] https://medium.com/@sjsyrek/five-minutes-to-monoid-fe6f364d0bba

Most of the basic folding tasks have dedicated actions (e.g. `sumOf`, `lengthOf`), so you'll likely only need these actions when working with your own Monoids.

An example of a task which *doesn't* already have an action provided to us is taking the *average* (i.e. **mean**) of a set of focused numbers. Let's compute the **average age** of all actors in our collection. To calculate that we'll need the sum of all the ages as well as the total **number** of actors! We could use `sumOf` and `lengthOf` to do this for us, but that would take two total passes over the structure, and well, that's clearly unacceptable for this arbitrary example which I'm just making up. We can get it down to a single pass by writing a custom Monoid.

First we need to understand the Monoid which we'll use for this. As I just mentioned we'll need two pieces of info: the total number of actors and the sum of their ages. To collect the total number of actors we can use the `Sum` Monoid, by mapping each element into a `Sum 1` we get the total number of actors after we **combine** all the elements together. We can do the exact same thing with the total of all ages; by wrapping them up in `Sum` the Monoidal append will add them into a total.

So now we have two separate Monoids, but we want to collect everything in a single pass, so we need to combine them into *one* Monoid. As it turns out, a tuple of two Monoids is also a Monoid! Combining tuples of Monoids combines the matching pieces together separately:

```
>>> (Sum 1, Sum 32) <> (Sum 1, Sum 20)
( Sum { getSum = 2 }
, Sum { getSum = 52 }
)
```

Okay, so let's try it out. First we'll write a helper function to transform an actor into the our Monoid:

```
import Data.Monoid

ageSummary :: Actor -> (Sum Int, Sum Int)
ageSummary actor = (Sum 1, Sum (calcAge actor))
```

Now we can use `(to)` to build a `Fold Actor (Sum Int, Sum Int)` and chain that onto a fold over all actors, then use `foldOf` to combine all the focuses together now that they're a Monoid!

```
>>> foldOf (folded . actors . folded . to ageSummary) tvShows
( Sum { getSum = 10 }
, Sum { getSum = 572 }
)
```

We can see that we have 10 actors in our fold and that the sum of their ages is 572. Now we can just divide to get the average:

```
computeAverage :: (Sum Int, Sum Int) -> Double
computeAverage (Sum count, Sum total) = fromIntegral total / fromIntegral count
```

Putting it all together:

```
>>> computeAverage $ foldOf (folded . actors . folded . to ageSummary) tvShows
57.2
```

Looks like the average age of the actors in our favourite shows as of 2030 is `57.2`!

`foldMapOf` exists specifically for this sort of case where we need to transform our data into a specific Monoid before we fold it; we can use it to separate the fold logic from the Monoid transformation like this:

```
>>> computeAverage $ foldMapOf (folded . actors . folded) ageSummary tvShows
57.2
```

We get the same answer of course.

Using 'view' on folds

A very common mistake when people get started with folds is to accidentally use `(^.)` or `view` on a **fold** instead of a **lens**. This is especially confusing because it **actually works** in **some** cases but not others; let's see what the big deal is. Here's an example of this strangeness:

```
-- This works just fine
>>> Just "do it" ^. folded
"do it"

-- This one crashes and burns!
>>> Just (42 :: Int) ^. folded
error:
    • No instance for (Monoid Int) arising from a use of ‘folded’
    • In the second argument of ‘(^.)’, namely ‘folded’
      In the expression: Just (42 :: Int) ^. folded
      In an equation for ‘it’: it = Just (42 :: Int) ^. folded
```

So what gives? Shouldn't these work the same? Why does the error mention a `Monoid`?

This situation is actually caused by a bit of `lens`'s implementation "leaking" out. The implementation of `view` accepts any type of optic, but adds **constraints** which **usually** only work on a lens. HOWEVER, if the **focus** you're viewing *happens* to be a `Monoid` it can "view" through the fold by using the Monoid typeclass.

Normally `view` retrieves only a single element, but a **fold** can focus **zero** or **more** things. If the focus is a Monoid, then `view` can return `mempty` (the identity element) if no focuses could be found, and can use `(<>)` to combine all the focuses into a **single** focus if there are many!

Here's how this works:

```
-- When there's a single focus, we just return it
>>> Just "do it" ^. folded
"do it"

-- When there aren't any focuses, return 'mempty'
>>> Nothing ^. folded :: String
""

-- When there are multiple focuses, combine them with (<>).
>>> ("one", "two", "three") ^. each
"onetwothree"
```

In general you should avoid this "weird" behaviour and just use `foldOf` explicitly when you want this behaviour. It's much more semantically clear, and makes it obvious that you **meant** to fold all the focuses together. However, I wanted to bring this up because everyone runs into it from time to time and it's very confusing if you don't know what's going on!

Customizing monoidal folds

Our average is actually a bit off though; did you notice that we have an actor who's in more than one show? Let's see if we can figure out which actors are in multiple shows. We'll write a function to count the number of shows each actor appears in.

This is sort of query is also expressible as a Monoid! Essentially, we can just track the number of times an actor's name occurs amongst all our focuses. We can keep the summary of the occurrences as a `Map String Int`; let's start by collecting all the actors names into a `Map` with a count of `1`.

```
>>> foldMapOf (folded . actors . folded . name) (\n -> M.singleton n 1) tvShows
fromList
  [ ("Alyson Hannigan",1)
  , ("Anthony Head",1)
  , ("Cobie Smulders",1)
  , ("David Boreanaz",1)
  , ("Jason Segel",1)
  , ("Josh Radnor",1)
  , ("Neil Patrick Harris",1)
  , ("Nicholas Brendon",1)
  , ("Sarah Michelle Gellar",1)
  ]
```

This is good, but we've discovered a dark secret of `Map`'s Monoid instance; when we **combine** two maps with the same keys it simply ignores duplicate values of the same key rather than accumulating them like we want it to! Here's an example which demonstrates it more clearly:

```
>>> M.singleton 'a' "first" <> M.singleton 'a' "second"
fromList [('a',"first")]
```

What we want it to do instead is to handle conflicting keys by ADDING the count at each. We can pull this together pretty easily using `unionWith` provided by `Data.Map`:

```
>>> :t M.unionWith
M.unionWith
  :: Ord k => (a -> a -> a) -> M.Map k a -> M.Map k a -> M.Map k a

>>> M.unionWith (+) (M.singleton "an actor" 1) (M.singleton "an actor" 1)
fromList [("an actor",2)]
```

That looks better! When we partially apply `unionWith` with the `(+)` operator we get a binary operation just like `(<>)`, but which properly handles conflicting keys! Since addition is associative, we can use this as a valid substitute for `(<>)` when working with `Maps` as a Monoid. But how do we use this custom function when folding over all of our focuses? Luckily the `lens` library provides a few actions we can use to enact our custom way of combining maps!

```
foldByOf    :: Fold s a -> (a -> a -> a) -> a -> s -> a
foldMapByOf :: Fold s a -> (r -> r -> r) -> r -> (a -> r) -> s -> r
foldrOf     :: Fold s a -> (a -> r -> r) -> r -> s -> r
foldlOf     :: Fold s a -> (r -> a -> r) -> r -> s -> r
```

That's quite a laundry list of functions. The latter two behave effectively the same as the corresponding `Data.List` versions, but work over the focuses of a fold instead. The former two are similar to `foldOf` and `foldMapOf`, but allow us to manually specify substitutes for the Monoidal identity and combining action.

We could effectively use any of these to accomplish our goal; but I'll demo `foldMapOf` and leave the others as an exercise for the reader. The addition of `By` into our action names implies that we're going to specify the Monoid "by which" the folding will occur. We'll just sub-in our custom combination function while still using `mempty` as our identity element.

```
>>> foldMapByOf
      (folded . actors . folded . name)  -- Focus each actor's name
      (M.unionWith (+))  -- Combine duplicate keys with addition
      mempty -- start with the empty Map
      (\n -> M.singleton n 1)  -- inject names into Maps with a count of 1
      tvShows
fromList
  [ ("Alyson Hannigan",2)
  , ("Anthony Head",1)
  , ("Cobie Smulders",1)
  , ("David Boreanaz",1)
  , ("Jason Segel",1)
  , ("Josh Radnor",1)
  , ("Neil Patrick Harris",1)
  , ("Nicholas Brendon",1)
  , ("Sarah Michelle Gellar",1)
  ]
```

That's a lot of arguments for a single function, but each of them dictate a relevant part of how we combine everything together. You may wish to name them separately to enhance readability.

We can now correctly report that "Alyson Hannigan" is in both shows!

Exercises – Fold Actions

Jump to answers

Consider the following list of actions for the exercises below:

```
elemOf    :: Eq a => Fold s a -> a -> s -> Bool
has       :: Fold s a -> s -> Bool
lengthOf  :: Fold s a -> s -> Int
sumOf     :: Num n => Fold s n -> s -> n
productOf :: Num n => Fold s n -> s -> n
foldOf    :: Monoid a => Fold s a -> s -> a
preview   :: Fold s a -> s -> Maybe a
lastOf    :: Fold s a -> s -> Maybe a
minimumOf :: Ord a => Fold s a -> s -> Maybe a
maximumOf :: Ord a => Fold s a -> s -> Maybe a
anyOf     :: Fold s a -> (a -> Bool) -> s -> Bool
allOf     :: Fold s a -> (a -> Bool) -> s -> Bool
findOf    :: Fold s a -> (a -> Bool) -> s -> Maybe a
foldrOf   :: Fold s a -> (a -> r -> r) -> r -> s -> r
foldMapOf :: Monoid r => Fold s a -> (a -> r) -> s -> r
```

1. Pick the matching action from the list for each example:

```
>>> _ folded []
False

>>> _ both ("Yo", "Adrian!")
"YoAdrian!"

>>> _ each "phone" ("E.T.", "phone", "home")
True

>>> _ folded [5, 7, 2, 3, 13, 17, 11]
Just 2

>>> _ folded [5, 7, 2, 3, 13, 17, 11]
Just 11

>>> _ folded ((> 9) . length) ["Bulbasaur", "Charmander", "Squirtle"]
True

>>> _ folded even [11, 22, 3, 5, 6]
Just 22
```

2. Use an action from the list along with any fold you can devise to retrieve the output from the input in each of the following challenges. There may be more than one correct answer.

Find the first word in the input list which is a palindrome; I.e. a word that's the same backwards as forwards.

```
input = ["umbrella", "olives", "racecar", "hammer"]
output = Just "racecar"
```

Determine whether all elements of the following tuple are EVEN

```
input = (2, 4, 6)
output = True
```

Find the pair with the largest integer

```
input = [(2, "I'll"), (3, "Be"), (1, "Back")]
output = Just (3,"Be")
```

Find the sum of both elements of a tuple. This one may require additional alterations AFTER running a fold.

```
input = (1, 2)
output = 3
```

3. **BONUS** – These are a bit trickier

Find which word in a string has the most vowels.

```
input = "Do or do not, there is no try."
output = Just "there"
```

Combine the elements into the expected String

```
input = ["a", "b", "c"]
output = "cba"
```

Good luck with the following ones! Get it to work however you can!

```
input = [(12, 45, 66), (91, 123, 87)]
output = "54321"

input = [(1, "a"), (2, "b"), (3, "c"), (4, "d")]
output = ["b", "d"]
```

6.4 Higher Order Folds

Now we're getting to the fun stuff; there's a whole set of optics combinators which **alter other optics**. I'll call these sorts of combinators **higher-order**, because usually they accept an optic as an argument and return a new one as a result. They can add a lot of power and flexibility to the folds we already know.

Taking, Dropping

First we'll look at `taking` and `dropping`. These are technically **traversals**, but as we'll learn soon, all traversals are also valid folds there's no harm in learning these before we understand traversals. They have the following pretty scary type signatures:

```
taking
  :: (Conjoined p, Applicative f)
  => Int
  -> Traversing p f s t a a
  -> Over p f s t a a

dropping
  :: (Conjoined p, Applicative f)
  => Int
  -> Over p (Control.Lens.Internal.Indexed.Indexing f) s t a a
  -> Over p f s t a a
```

These types are complex, there's really no reason to worry about understanding them fully unless you plan to write your own optics implementation. It's far more important to know and understand the "practical" signature of these optics, i.e. how can they be used? To find that I tend to go check the documentation which usually lists a few possible options for each combinator. If you look at the documentation we'll see a whole list of valid specialized signatures:

```
taking :: Int -> Traversal' s a          -> Traversal' s a
taking :: Int -> Lens' s a               -> Traversal' s a
taking :: Int -> Iso' s a                -> Traversal' s a
taking :: Int -> Prism' s a              -> Traversal' s a
taking :: Int -> Getter s a              -> Fold s a
taking :: Int -> Fold s a                -> Fold s a
taking :: Int -> IndexedTraversal' i s a -> IndexedTraversal' i s a
taking :: Int -> IndexedLens' i s a      -> IndexedTraversal' i s a
taking :: Int -> IndexedGetter i s a     -> IndexedFold i s a
taking :: Int -> IndexedFold i s a       -> IndexedFold i s a
```

From here we can pick out the one we care about at the moment (and grab the same for `dropping`):

```
taking   :: Int -> Fold s a -> Fold s a
dropping :: Int -> Fold s a -> Fold s a
```

I know these signatures are still very intimidating and tricky, but they'll get easier in time. With a bit of practice you'll be able to break up these signatures into sensible pieces and start to notice patterns too! For now, just double check the documentation for any combinator that's giving you trouble and try to find one of the optional signatures which meets your needs.

So how do these two work? They're essentially the same as `take` and `drop` from the Prelude, but which operate on the fold you pass to them instead of a list. For example `(take 3)` accepts a list and returns the first `3` elements as a list, so `(taking 3)` accepts a **fold** which focuses the first three focuses of **myFold**.

Here are a few deceptively simple examples:

```
>>> [1, 2, 3, 4] ^.. taking 2 folded
[1,2]
>>> [1, 2, 3, 4] ^.. dropping 2 folded
[3,4]
```

Here `taking` is passed `folded` as an argument, `folded` is modified so to only focus its first two elements, it's then passed to `(^..)` which collects them as a list, effectively rebuilding `take` using folds!

I say *deceptively* simple because there's an additional tricky factor we sometimes run into when working with nested folds which doesn't ever occur when using `take` and `drop` on lists. The confusion occurs because each additional fold we compose onto our path **branches out** like a tree. You can think of a series of folds as a branching **tree** which simultaneously crawls deeper into many elements at once. When we add additional folds to the path they're *grafted* onto the branches of the tree and thus can only operate over their own subtree of the computation.

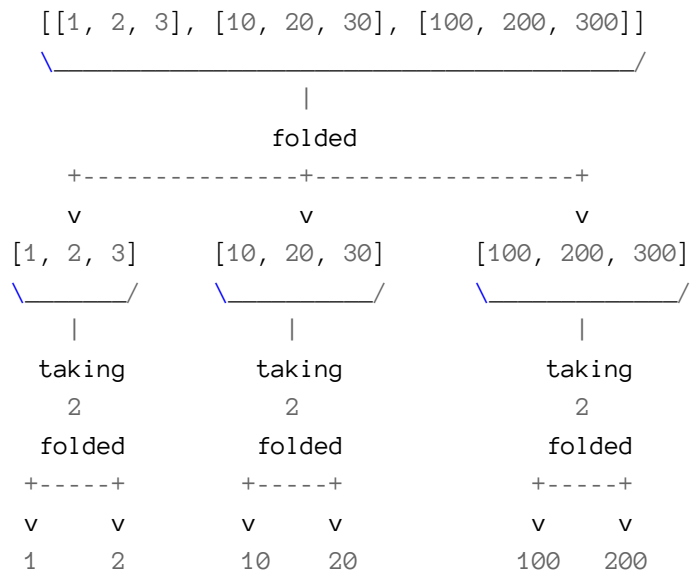
The implications of this branching idea is that when we run something like `taking 2`, it adds a **separate** `taking 2` onto *each* branch of the path to its left. Confused? Me too, this is better explained with some examples!

Consider the following fold:

```
>>> [[1, 2, 3], [10, 20, 30], [100, 200, 300]] ^.. folded . taking 2 folded
[1, 2, 10, 20, 100, 200]
```

This may seem counter-intuitive at first! It's definitely yielding more than 2 elements; wasn't it supposed to only take 2?

If we break up the folds into a tree structure it gets a bit easier to see:



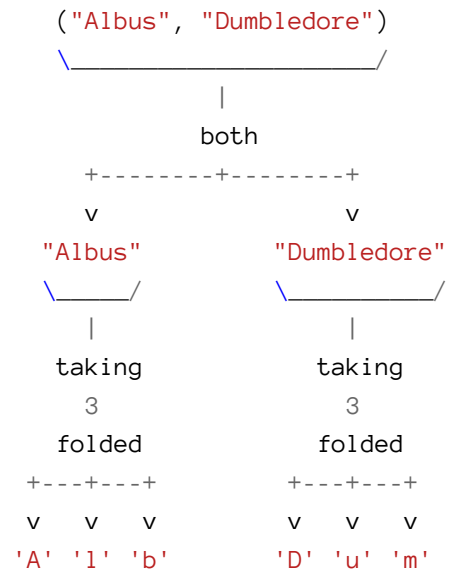
So we can see how `taking 2 folded` constructs a new fold which focuses two elements from each list it runs on, but it **runs once per element in the previous fold**!

If the "tree" metaphor isn't doing it for you, you can also imagine it like a pipeline or stream. Each fold sends values down the stream, `taking 2 folded` runs on each element of the stream and sends two more distinct elements down-stream to the next fold (the one to its right).

Here's another example:

```
>>> ("Albus", "Dumbledore") ^.. both . taking 3 folded
"AlbDum"
```

Here's the breakdown:



This second example is similar. We first branch into each half of the tuple, then we take 3 Characters from each String independently, yielding them side-by-side to get the 6 character String `"AlbDum"`.

Okay, so from the previous diagrams we can see that higher order folds like `taking` can really only affect the focuses being passed to them. What if we really wanted the **whole** fold to select **only** two elements?

We can do this by moving our `taking` modifier to the left (i.e. **up** the tree) to operate on the root of the whole tree! If we run it on the whole path it will limit the elements from **all** branches down-stream. It accepts any optic as an argument, and it's fine if that optic is itself a composition of other optics. Let's see the difference:

```
>>> [[1, 2, 3], [10, 20, 30], [100, 200, 300]]
      ^.. taking 2 (folded . folded)
[1, 2]

>>> ("Albus", "Dumbledore") ^.. taking 3 (both . folded)
"Alb"
```

Notice how we need brackets around the optic we pass to `taking`; if we forget the brackets it'll behave differently!

```
-- No brackets; we're taking '3' from the results of 'both', then folding them
>>> ("Albus", "Dumbledore") ^.. taking 3 both . folded
"AlbusDumbledore"

-- With brackets; we're taking '3' from the results of '(both . folded)'
>>> ("Albus", "Dumbledore") ^.. taking 3 (both . folded)
"Alb"
```

We can keep composing more folds after a call to `taking` or `dropping`:

```
-- Brackets around `taking 2 folded` aren't necessary *in this case*
-- but help with clarity
>>> [[1, 2, 3], [10, 20, 30], [100, 200, 300]] ^.. (taking 2 folded) . folded
[1,2,3,10,20,30]

>>> (["Albus", "Dumbledore"], ["Severus", "Snape"])
      ^.. taking 3 (both . folded)
["Albus","Dumbledore","Severus"]

>>> (["Albus", "Dumbledore"], ["Severus", "Snape"])
      ^.. taking 3 (both . folded) . folded
"AlbusDumbledoreSeverus"
```

I didn't mean to forget about `dropping`, but I didn't want things to get too confusing. It works as you'd expect, dropping the specified amount of leading elements from the fold you pass it:

```
>>> [[1, 2, 3], [10, 20, 30], [100, 200, 300]]
      ^.. dropping 2 (folded . folded)
[3,10,20,30,100,200,300]

>>> [[1, 2, 3], [10, 20, 30], [100, 200, 300]]
      ^.. folded . dropping 2 folded
[3,30,300]

>>> [[1, 2, 3], [10, 20, 30], [100, 200, 300]]
      ^.. dropping 2 folded . folded
[100,200,300]

>>> ("Albus", "Dumbledore") ^.. both . dropping 2 folded
"busmbledore"

>>> ("Albus", "Dumbledore") ^.. dropping 2 (both . folded)
"busDumbledore"
```

Backwards

Here's another **higher-order** fold which is fun to use: `backwards`

```
backwards :: Fold s a -> Fold s a

-- *really*
backwards :: (Profunctor p, Profunctor q)
          => Optical p q (Backwards f) s t a b
          -> Optical p q f s t a b
```

This **higher-order** fold **modifier** takes a fold and reverses the order in which it yields elements.

Observe:

```
>>> [1, 2, 3] ^.. backwards folded
[3,2,1]

>>> ("one", "two") ^.. backwards both
["two","one"]

>>> [(1, 2), (3, 4)] ^.. backwards (folded . both)
[4,3,2,1]

>>> [(1, 2), (3, 4)] ^.. backwards folded . both
```

```
[3,4,1,2]

>>> [(1, 2), (3, 4)] ^.. folded . backwards both
[2,1,4,3]
```

Don't try reversing an infinite fold though; you'll be waiting a while.

Note how `backwards` the behaviour we get really depends on *where* `backwards` is applied. Visualize either the tree or stream metaphor when thinking about where to place it.

TakingWhile, DroppingWhile

Here are two more higher-order folds:

```
takingWhile   :: (a -> Bool) -> Fold s a -> Fold s a
droppingWhile :: (a -> Bool) -> Fold s a -> Fold s a
```

`takingWhile` takes a predicate over the focuses of a fold and will modify the given fold so it will yield elements only until the predicate becomes False, then will stop yielding elements entirely.

```
>>> [1..100] ^.. takingWhile (<10) folded
[1, 2, 3, 4, 5, 6, 7, 8, 9]

-- Once the predicate is false no more elements will be yielded,
-- even if later elements pass the predicate.
>>> [1, 5, 15, 5, 1] ^.. takingWhile (<10) folded
[1, 5]
```

`droppingWhile` is similar, but of course *drops* fold elements until the predicate is false, then yields ALL remaining elements.

```
>>> [1..100] ^.. droppingWhile (<90) folded
[90,91,92,93,94,95,96,97,98,99,100]

-- Once the predicate is false ALL remaining elements are yielded
-- even if later elements pass the predicate.
>>> [1, 5, 15, 5, 1] ^.. droppingWhile (<10) folded
[15,5,1]
```

Exercises – Higher Order Folds

Jump to answers

1. Fill in the blank. You'll need to remember some tricks from previous sections!

```
>>> "Here's looking at you, kid" ^.. _ 7 folded
"looking at you, kid"

>>> ["My Precious", "Hakuna Matata", "No problemo"] ^.. folded . taking 1 _
["My","Hakuna","No"]

>>> ["My Precious", "Hakuna Matata", "No problemo"] ^.. _
["My"]

>>> ["My Precious", "Hakuna Matata", "No problemo"] ^.. folded . _
"MyHakunaNo"

>>> import Data.Char (isAlpha)
>>> ["My Precious", "Hakuna Matata", "No problemo"]
      ^.. folded . _
"MyHakunaNo"

>>> _ (10, 50, 100)
60

>>> ("stressed", "guns", "evil") ^.. _ each
["evil","guns","stressed"]

>>> ("stressed", "guns", "evil") ^.. backwards each . to _
["live","snug","desserts"]

>>> import Data.Char (isAlpha)
>>> "blink182 k9 blazeit420" ^.. _
"1829420"
```

2. Solve the following problems using higher-order folds:

We're doing a temperature study in a given region, but we need to run tests on several subsets of the data.

Here's the series of daily temperature measurements we've been given for the region:

```
sample :: [Int]
sample = [-10, -5, 4, 3, 8, 6, -2, 3, -5, -7]
```

First we're interested in how many days it took until the first thaw. Write an expression which calculates the number of measurements until a temp is above zero.

```
>>> _
2
```

For the next study we need to know the warmest it got in the first 4 days of the sample. Write an expression to calculate it.

```
>>> _
Just 4
```

Write an expression to calculate the temperature on the day AFTER we hit that temperature. Use `preview` or `^?` somewhere in that expression if you can.

```
>>> sample ^? _
Just 3
```

How many days of below-freezing weather did we have consecutively at the END of the sample?

```
>>> _
2
```

Now we're interested in running statistics on temperature data specifically from the first thaw until the next freeze. Write an expression which lists out all temperature samples from the first time we sample above 0, until the next time we're below zero.

```
-- We can combine multiple modifiers to get this behaviour,
-- We drop all leading negative temperatures and then
-- take temperatures until another freeze!
>>> sample ^.. _
[4, 3, 8, 6]
```

BONUS: List out all the temperature samples between the FIRST thaw and the FINAL freeze.

```
>>> sample ^.. _
[4, 3, 8, 6, -2, 3]
```

Generalize this behaviour into a function: `trimmingWhile`. It should drop elements from the start AND end of a fold while a predicate is True.

```
trimmingWhile :: (a -> Bool) -> Fold s a -> Fold s a
```

6.5 Filtering folds

I've mentioned earlier that folds can be thought of as **queries** over a structure. If we think of our folds in terms of a query language like `SQL` we've learned how to `SELECT` and how to `LIMIT`, but we haven't learned how to **filter** like we would with a `WHERE` clause! For example, `folded` just focuses ALL elements of a container, that's not very expressive! We'll see if we can do better.

Filtered

I'm excited to introduce an absolute powerhouse of functionality when it comes to working with optics. `filtered` is extremely flexible and can be used on most types of optics!

It allows you to **filter** down the focused elements to a subset which match some provided predicate:

```
filtered :: (s -> Bool) -> Fold s s

-- *real*
filtered :: (Choice p, Applicative f) => (a -> Bool) -> Optic' p f a a
```

The signature tells us that this fold doesn't change the type of the elements as they pass through, it simply drops elements from the fold if they don't match the predicate.

Let's look at some examples:

```
-- Drop elements unless they're even
>>> [1, 2, 3, 4] ^.. folded . filtered even
[2,4]

-- Drop fruits with names shorter than 6 characters
>>> ["apple", "passionfruit", "orange", "pomegranate"]
      ^.. folded
        . filtered ((>6) . length)
["passionfruit","pomegranate"]
```

That's the basic idea, but the power comes from using it in the midst of other folds. We can filter the elements of the fold on a given property and then dive deeper into a different part of the structure. It's example time.

I've been doing some spring cleaning and I dug up some of my favourite legally ambiguous trading cards! There's a lot in the box though, and I really hate reading, so in order to organize my collection I'll need some folds to help me out. Careful not to 'fold' any of my cards though, that'd ruin the resale value.

Here's what my collection looks like:

```haskell
-- A data structure to represent a single card
data Card =
    Card { _name    :: String
         , _aura    :: Aura
         , _holo    :: Bool
         , _moves   :: [Move]
         } deriving (Show, Eq)

-- Each card has an aura-type
data Aura
    = Wet
    | Hot
    | Spark
    | Leafy
    deriving (Show, Eq)

-- Cards have attack moves
data Move =
    Move { _moveName  :: String
         , _movePower :: Int
         } deriving (Show, Eq)

makeLenses ''Card
makeLenses ''Move
```

That'll help me model my collection. Here are the cards I collected over the years:

```haskell
deck :: [Card]
deck = [ Card "Skwortul"    Wet False   [Move "Squirt" 20]
       , Card "Scorchander" Hot False   [Move "Scorch" 20]
       , Card "Seedasaur"   Leafy False [Move "Allergize" 20]
       , Card "Kapichu"     Spark False [Move "Poke" 10 , Move "Zap" 30]
       , Card "Elecdude"    Spark False [Move "Asplode" 50]
       , Card "Garydose"    Wet True    [Move "Gary's move" 40]
       , Card "Moisteon"    Wet False   [Move "Soggy" 3]
       , Card "Grasseon"    Leafy False [Move "Leaf Cut" 30]
       , Card "Spicyeon"    Hot False   [Move "Capsaicisize" 40]
       , Card "Sparkeon"    Spark True  [Move "Shock" 40 , Move "Battery" 50]
       ]
```

Now we can start to ask some clever questions and queries about the collection. We can use any action we've already learned as a query, but we can also more intelligently filter the data before it reaches the action using `filtered`.

```
-- How many Spark Cards do I have?
>>> lengthOf
      ( folded  -- Fold over each card
      . aura    -- Select the 'aura'
      . filtered (== Spark) )  -- Filter for only Spark auras
      deck
3

-- How many moves have an attack power above 30?
>>> lengthOf
      ( folded -- Fold over each card
      . moves  -- Select the list of moves
      . folded -- Fold over each move
      . movePower -- Select the move Power
      . filtered (>30) ) -- Filter for only movePower greater than 30
      deck
5
```

We can even use other fold queries INSIDE our filter! This lets us filter elements based on a query over the currently focused piece of state.

```
--- List all cards which have ANY move with an attack power greater than 40
>>> deck
      ^.. folded -- Fold over each card
        -- filter on whether the current card has any strong moves
        . filtered (anyOf (moves . folded . movePower) (>40))
        -- Select the card's name
        . name
[ "Elecdude"
, "Sparkeon"
]
```

We can filter based on certain properties and then continue diving deeper into a *different* portion of the data:

```
-- How many moves do my Spark cards have in total?
>>> lengthOf
      ( folded -- Fold over each card
      . filtered ((== Spark) . _aura)  -- Filter for only Spark aura cards
      . moves -- Select moves
      . folded )  -- Fold over all moves
      deck
5

-- List all my Spark Moves with a power greater than 30
>>> deck
      ^.. folded -- Fold over all cards
        . filtered ((== Spark) . _aura) -- Filter for only Spark aura careds
        . moves -- Select moves
        . folded -- Fold over all moves
        . filtered ((>30) . _movePower) -- Filter for moves with >30 attack
        . moveName -- Select move names
[ "Asplode"
, "Shock"
, "Battery"
]
```

This is a really powerful tool, though it seems a bit clunky when we need to switch to using the underscore field accessor to build a predicate for our filter. Luckily there's a tool for that! It's pretty new, so you'll want to double check you have at least `lens >= 4.18.0`.

> If you're using `stack` for your project you'll likely need to be on a nightly snapshot to have access to `lens-4.18.0`. The following resolver (or newer) should work: `resolver: nightly-2019-10-02`

Here's `filteredBy`:

```
-- Simple Signature
filteredBy :: Fold s a -> Fold s s

-- Slightly more accurate signature:
filteredBy :: Fold s a -> IndexedTraversal' a s s

-- Laughably complex *real* type:
filteredBy
  :: (Indexable i p, Applicative f)
  => Getting (First i) a i -> p a (f a) -> a -> f a
```

The simplest signature still looks a bit strange (though not as bad as the real signature), we can see that it accepts a fold from `<s>` to `<a>`, but doesn't actually use the `<a>` for anything. This is because `filteredBy` runs the given fold on all elements it receives and checks whether that yields any values. It filters out any elements for which the given fold yields no values and lets the others through. That is, if `(has theFold s)` is `True` then the value is passed on, otherwise it's filtered out.

We can see from the slightly more complex signature that the value matched by the fold IS actually used as the index of the resulting `IndexedTraversal`, but we'll cover how to use that later on.

Using `filteredBy` we can pass a fold instead of a predicate! Here's that last example again using `filteredBy`:

```
-- List all my Spark Moves with a power greater than 30
>>> deck
      ^.. folded
        . filteredBy (aura . only Spark)
        . moves
        . folded
        . filteredBy (movePower . filtered (>30))
        . moveName
[ "Asplode"
, "Shock"
, "Battery"
]
```

Though it isn't any shorter it can sometimes be more natural to express. It has the benefit that we can continue to **think in folds** and keep reading left-to-right. I use a nifty fold helper here called `only`. Let me explain it quickly before moving on:

```
only :: Eq a => a -> Fold a ()

-- real signature
only :: Eq a => a -> Prism' a ()
```

This is a *utility* fold. We're unlikely to use it on its own, but it's handy for use with things like `filteredBy`. It accepts a reference value and will return a fold which yields a `()` result if and only if the input value is equal to the reference value.

```
>>> 1 ^? only 1
Just ()
>>> 2 ^? only 1
Nothing

>>> has (only "needle") "needle"
True
>>> has (only "needle") "haystack"
False
```

We can write most filters using either of `filtered` or `filteredBy`, so pick whichever method you prefer.

Here's one last complex example which uses queries when specifying the `comparator` to `maximumByOf`. Oftentimes committing to using optics means we can use expressions that read almost like English.

```
>>> import Data.Ord (comparing)
-- Get the holographic card which has the largest number of moves
>>> maximumByOf
      (folded . filtered _holo)  -- filter for holo cards
      (comparing (lengthOf moves)) -- compare them on number of moves
      deck
Just
    ( Card
        { _name = "Sparkeon"
        , _aura = Spark
        , _holo = True
        , _moves =
            [ Move
                { _moveName = "Shock"
                , _movePower = 40
                }
            , Move
                { _moveName = "Battery"
                , _movePower = 50
                }
            ]
        }
    )
```

Exercises – Filtering

Jump to answers

Use a fold to answer each of the questions about my card collection:

- List all the cards whose name starts with `'S'`.
- What's the lowest attack power of all moves?
- What's the name of the first card which has more than one move?
- Are there any `Hot` cards with a move with more than 30 attack power?
- List the names of all holographic cards with a `Wet` aura.
- What's the sum of all attack power for all moves belonging to non-Leafy cards?

6.6 Fold Laws

I figured I should probably have this section for consistency, but the fact is: folds don't have any laws! Unlike lenses we can't write 'set-get', `get-set`, or 'set-set' laws since we can't 'set' at all!

This one's a freebie, there's no wrong way to query a data structure, so go hog wild and try out whatever you like.

7. Traversals

7.1 Introduction to Traversals

Onwards and upwards! Traversals are my personal favourite; they're among the most adaptable and useful of all the optics. Traversals expand on the capabilities offered by folds, allowing all the same querying, but with the added bonus that we can **update** or **set** the values we focus **in-place**. We can chain traversals to isolate an update to only a very specific nested subset of our data.

Traversals get their name from the `Traversable` typeclass and corresponding `traverse` function, so we're going to see a lot of similarities there. Traversals can perform pure updates of course, but can also run effectful update operations just like `traverse` can!

How do Traversals fit into the hierarchy?

Traversals are essentially what you get if you **combine** all the powers of folds and lenses **together**.

As you know, lenses can **get** and **set** only a **single** value, whereas folds can **get multiple** values, but can't **set** or **update**. Traversals are the fusion of the two and allow us to **get or set zero or more values**; for this reason early versions of traversals were initially called *multilenses*. The term *traversal* was coined with the advent of the `lens` library.

Here's a breakdown of our optics so far:

	Lens	**Fold**	**Traversal**
Get	Single	Many	Many
Set/Modify	Single	Nope	Many
Traverse	Single	Nope	Many

This chart correctly implies that all lenses are valid traversals, but the reverse isn't true. Likewise, both traversals and lenses are valid folds, but a fold is neither a valid lens nor traversal.

Traversals can focus many things and lenses can only focus one; so does that mean traversals are somehow **better** than lenses? Nope, with the added generality we lose some guarantees! When working with traversals rather than lenses we can no longer guarantee that our optic **must** have a focus, meaning we need to use the more conservative viewing actions `(^?)` or `(^..)` rather than `(^.)`. In general we need to be more careful using a traversal because it may fail.

Here's a quick outline of which of our optics are valid substitutions for each other. The **row labels** represent what you HAVE, and the **column headers** represent how you'd like to USE it.

	Lens	Fold	Traversal
Lens	Yes	Yes	Yes
Fold	No	Yes	No
Traversal	No	Yes	Yes

From this chart, we can see that a traversal is a valid fold by looking up the **row** with traversal in the first column and referencing its fold column. We can also see that the inverse is NOT true; a fold is **not** a valid traversal.

If you're interested in this table, you can find a *complete* version in the compatibility table appendix.

A bit of Nostalgia

As I alluded to earlier, a few of the folds we've learned already have secretly been traversals! We were able to use them seamlessly as folds without tapping into their true power. Here's a list of combinators we've already learned which are valid traversals!

- `both`, `each`
- `filtered`, `filteredBy`
- `taking`, `dropping`
- `takingWhile`, `droppingWhile`
- All lenses

It's not so tough to understand traversals after you understand both lenses and folds; you'll just have to fuse your understanding of the two concepts together.

From fold to traversal

Let's revisit `both` now that we can use its full power. `both` is a **traversal** which **focuses** *both* sides of a tuple (if it has the same type in each side).

```
-- Specialized to tuples:
both :: Traversal (a, a) (b, b) a b

-- Generally
both :: Bitraversable r => Traversal (r a a) (r b b) a b
```

The `Traversal` type synonym is just like the `Lens` type; it takes the familiar form (`Traversal s t a b`) where `<s>` and `<t>` represent the pre and post action structures, and `<a>` and `<b>` represent the pre/post action focuses. If you need a bit of a reminder on what all the parameters mean, feel free to skim through polymorphic optics again.

As we already learned in the folds chapters, we can use the traversal as a fold:

```
>>> ("Bubbles", "Buttercup") ^.. both
["Bubbles", "Buttercup"]
```

But now that we've unlocked the power of traversals, we can also **modify** the focuses using the **over** action: `(%~)`.

```
>>> ("Bubbles", "Buttercup") & both %~ (++ "!")
("Bubbles!", "Buttercup!")
```

We can see it ran the modification (i.e. added an exclamation point) to *both* elements independently, and returns a new tuple with the modified elements. This behaviour should remind you of modifying using lenses, except that we've focused more than one element.

What happens when we try using **`set`** `(.~)` on a traversal?

```
>>> ("Bubbles", "Buttercup") & both .~ "Blossom"
("Blossom", "Blossom")
```

It **set** *both* elements to the value we specified without regard for what their previous value was. Looks like it's working!

We learned about polymorphic lenses a long time ago; those are optics which allow you to change the *type* of the structure when updating or setting. There are polymorphic traversals too! We saw from the type that `both` is one of these polymorphic traversals:

```
both :: Traversal (a, a) (b, b) a b
```

This signature says we can go from a tuple `(a, a)` to a tuple `(b, b)` if you provide a way to go from an `<a>` to a `<b>`.

Let's change from a `(String, String)` into a `(Int, Int)` by getting the length of each string.

```
>>> ("Bubbles", "Buttercup") & both %~ length
(7, 9)
```

Nice!

`each` is also a Traversal. It allows modifying *each* element in a structure:

```
>>> (1, 2, 3) & each %~ (*10)
(10, 20, 30)

>>> [1, 2, 3] & each %~ (*10)
[10, 20, 30]

>>> import Data.Char (toUpper)
>>> ("Here's Johnny" :: T.Text) & each %~ toUpper
"HERE'S JOHNNY"
```

A polymorphic traversal has to change ALL occurrences of a type variable within the structure in order for the type to change. For example, if you have a list of Strings `[String]` and want to turn it into a list of Integers `[Int]`, you need to change EVERY String in the list to an Integer. If the traversal doesn't focus every element then you'll be unable to change the type. You're not allowed to have a list that's part `Int` and part `String`!

Regarding `each` specifically, whether it's a polymorphic traversal depends on **which structure** you're using it on. Certain structures like lists and tuples can freely change types, but others like `Text` or `ByteString` are *monomorphic* structures. A `Text` block *must* be made up of `Chars`, we can't just change them to `Int`s and expect it to work. If we try to do something like that, we'll get a type error:

```
>>> ("Houston we have a problem" :: T.Text) & each .~ (22 :: Int)

error:
    • Couldn't match type ‘Int’ with ‘Char’
        arising from a use of ‘each’
    • In the first argument of ‘(.~)’, namely ‘each’
      In the second argument of ‘(&)’, namely ‘each .~ (22 :: Int)’
      In the expression:
        ("Houston we have a problem" :: T.Text) & each .~ (22 :: Int)
```

The type error tells us it expected a `Char` where we passed an `Int`.

Now for `taking` and `dropping`; how do those work when we set or update through them?

```
>>> [1, 2, 3, 4, 5] & taking 3 traversed *~ 10
[10, 20, 30, 4, 5]

>>> [1, 2, 3, 4, 5] & dropping 3 traversed *~ 10
[1, 2, 3, 40, 50]

>>> "once upon a time - optics became mainstream"
      -- focus characters until a '-'
      & takingWhile (/= '-') traversed
      %~ toUpper
"ONCE UPON A TIME - optics became mainstream"
```

`filtered` can be used to filter focuses from a traversal!

```
-- Multiply all even numbers by 10
>>> [1, 2, 3, 4, 5]
      & traversed . filtered even
      *~ 10
[1, 20, 3, 40, 5]

-- Reverse only the long strings
>>> ("short", "really long")
      & both . filtered ((> 5) . length)
      %~ reverse
("short","gnol yllaer")
```

`filtered` is an **extremely** powerful tool, it alters the exact same elements which would be focused when you use it in a fold.

7.2 Traversal Combinators

Traversing each element of a container

In the chapters on folds we learned about `folded` which focuses each element of `Foldable` containers like lists, Maps, Strings, etc. We got a lot of use out of it, but as you might have guessed, `folded` is only for folding; if we try to **set** or **update** using `folded` GHC will yell at us (it's not mad, just disappointed):

```
>>> [1, 2, 3] & folded %~ (*10)

error:
    • No instance for (Contravariant Identity)
        arising from a use of ‘folded’
    • In the first argument of ‘(%~)’, namely ‘folded’
      In the second argument of ‘(&)’, namely ‘folded %~ (* 10)’
      In the expression: [1, 2, 3] & folded %~ (* 10)
```

It says it wants an instance for `Contravariant Identity`, when you have conflicting constraints like this it's either because you're composing incompatible optics together, or because you're using an action which is incompatible with your optic. In this case the `%~` action requires a `Setter` and wants to use the `Identity` type, but `folded` is a Fold which requires a `Contravariant` instance, so the two can't get along.

We'll need a more powerful version of `folded` for traversing:

```
-- Slightly simplified
traversed :: Traversable f => Traversal (f a) (f b) a b

-- Real type
traversed :: Traversable f => IndexedTraversal Int (f a) (f b) a b
```

`traversed` can focus on the values inside any `Traversable` data structure. Requiring `Traversable` as opposed to only `Foldable` gives us a bit more power, but also reduces the number of containers we can operate on. For instance, Sets are `Foldable` but *not* `Traversable`. This means `folded` can be used on **more container types** (like `Set`), but `traversed` has strictly **more power** (it can **set** and **update**).

Let's use `traversed` to update values in a few `Traversable` containers.

```
>>> [1, 2, 3] & traversed *~ 10
[10,20,30]

-- Tuples are traversable over their last slot
>>> ("Batman", "Superman") & traversed %~ take 3
("Batman","Sup")

-- We can traverse all values in a Map
>>> import qualified Data.Map as M
>>> let powerLevels = M.fromList
      [ ("Gohan", 710)
      , ("Goku", 9001)
      , ("Krillin", 5000)
```

```
      , ("Piccolo", 408)
      ]
>>> powerLevels
      & traversed %~ \n ->
                       if n > 9000
                          then "Over 9000"
                          else show n
fromList
    [ ("Gohan", "710")
    , ("Goku", "Over 9000")
    , ("Krillin", "5000")
    , ("Piccolo", "408")
    ]
```

We can even traverse over more complex structures like Trees:

```
>>> import Data.Tree
>>> let opticsTree = Node "Lens" [Node "Fold" [], Node "Traversal" []]
>>> opticsTree & traversed %~ reverse
Node "sneL" [Node "dloF" [], Node "lasrevarT" []]
```

The ability to edit elements within their current context comes in handy pretty often.

So far we've basically just been reimplementing `fmap`, but we'll learn a few more actions soon.

More Combinators

Let's tack on a few more fun traversals before the exercises!

If you haven't figured it out yet, traversals and folds can be pretty clever things. They're not restricted to just accessing fields of records, lists or tuples. Here are two traversals which actually focus *sub-sections* of Strings:

```
worded :: Traversal' String String
lined :: Traversal' String String

-- *real types*
worded :: Applicative f => IndexedLensLike' Int f String String
lined  :: Applicative f => IndexedLensLike' Int f String String
```

As you might have guessed, these will focus individual **words** or **lines** of your string.

As folds they work pretty much like `words` and `lines` from the Prelude:

```
>>> "I'll be back!" ^.. worded
["I'll","be","back!"]

>>> "Run\nForrest\nRun" ^.. lined
["Run","Forrest","Run"]
```

But we can also update words and the traversal will rebuild the original string around them!

```
-- Surround each word with '*'s
>>> "blue suede shoes" & worded %~ \s -> "*" ++ s ++ "*"
"*blue* *suede* *shoes*"

>>> import Data.Char (toUpper)
-- Capitalize each word
>>> "blue suede shoes" & worded %~ \(x:xs) -> toUpper x : xs
"Blue Suede Shoes"

-- Add a "#" to the start of each line:
>>> "blue\nsuede\nshoes" & lined %~ ('#':)
"#blue\n#suede\n#shoes"
```

There's a caveat to these traversals though; they rebuild to the result string by effectively calling 'unwords' or 'unlines' on the modified results. This means that they're lossy, and forget a few things about the original string. We'll learn later how they're technically unlawful traversals. For example `words` will replace newlines and multiple sequential spaces with just a single space:

```
-- Mapping the identity function still has the
-- white-space collapsing side-effects of `unwords`
>>> "blue \n suede \n \n shoes" & worded %~ id
"blue suede shoes"
```

Be wary that this is how `worded` and `lined` from `lens` behave; even though they're not lawful they're still useful.

Traversing multiple paths at once

Time to introduce a few new tools for diving through data. Here's another favourite of mine; `beside`.

Like most combinators, there are many possible signatures, and the actual signature is an absolute nightmare. Here are a few specialized ones which are a bit more approachable:

```
beside :: Traversal s t a b
       -> Traversal s' t' a b
       -> Traversal (s,s') (t,t') a b

beside :: Lens s t a b -> Lens s' t' a b -> Traversal (s,s') (t,t') a b
beside :: Fold s a     -> Fold s' a      -> Fold (s,s') a
```

The first thing we notice is that this is a higher-order optic. It takes two optics as arguments and returns a new optic as a result. These signatures are pretty confusing at first, but if you stare at them for long enough you'll figure it out! I'd recommend you take some time to study them and make a guess about what it does before moving on. Learning to read these signatures for yourself is a pretty useful skill.

Did you come up with anything? We can see that `beside` takes a tuple as the input structure. The optics it accepts as arguments provide a way to focus an `<a>` from the pieces on either side of the tuple. We can intuit from this that `beside` uses the provided optics to focus all of the `<a>`s from both sides of the tuple.

Let's see it:

```
>>> let dinos = ("T-Rex", (42, "Stegosaurus"))
>>> :t dinos
dinos :: (String, (Int, String))

>>> dinos ^.. beside id _2
["T-Rex", "Stegosaurus"]

>>> let numbers = ([(1, 2), (3, 4)], [5, 6, 7])
>>> :t numbers
numbers :: ([(Int, Int)], [Int])

-- We can provide a different path to focus Ints for each half of the tuple
>>> numbers ^.. beside (traversed . both) traversed
[1, 2, 3, 4, 5, 6, 7]

>>> ("T-Rex", ("Ankylosaurus", "Stegosaurus")) ^.. beside id both
["T-Rex", "Ankylosaurus", "Stegosaurus"]
```

Can you see how `beside` is more powerful than `both`? `beside` allows us to specify a separate path to find the focuses for each half of the tuple, meaning we can use it on tuples which have a different type in each half so long as the focuses of each half are the same. `both` is a subset of this behaviour, in fact we can write `both` in terms of `beside` by taking each half of the tuple 'as-is':

```
both = beside id id
```

Remember that `id` is a valid optic which always just focuses its input.

`beside` creates a traversal when each of its arguments is a traversal, so we can edit values deep inside each half of the tuple too.

```
>>> import Data.Char (toUpper)
-- We can modify all characters inside both halves of the tuple
>>> ("Cowabunga", ["let's", "order", "pizza"])
      -- Each half of the tuple has a different path to focus the characters
      & beside traversed (traversed . traversed)
      %~ toUpper
("COWABUNGA", ["LET'S", "ORDER", "PIZZA"])
```

In this latest example we `traversed` on the left half to focus each character in "Cowabunga". On the right half of the tuple we use `traversed` twice, once to focus each element of the list, then again to focus the characters of each string in that list.

The types shown above are specialized to tuples, but actually `beside` works on any `Bitraversable` type. This means you can even use `beside` on your own types by implementing `Bitraversable` for them. `Either` is one other `Bitraversable` type you're familiar with, we can tell `beside` how to focus elements from both sides of the `Either` type and it will craft a traversal for us.

For example, we can specialize the following traversal like this:

```
beside both traversed :: Traversal' (Either (Int, Int) [Int]) Int
```

And use it like so:

```
>>> Left (1, 2) & beside both traversed %~ negate
Left (-1, -2)

>>> Right [3, 4] & beside both traversed %~ negate :: Either (Int, Int) [Int]
Right [-3 , -4]
```

Focusing a specific traversal element

We already know about `taking` and `dropping` which can focus a specific number of elements from the beginning or end of a traversal, but what if we want to traverse just a single specific **element**? Here's an auspiciously named combinator for that:

```
element  :: Traversable f =>  Int -> Traversal' (f a) a
```

This is a simple traversal which will focuses only the nth element of a Traversable container, leaving the others alone. Because it doesn't focus EVERY element of the container it's a **monomorphic** traversal. It can't change the type of the container's elements.

```
>>> [0, 1, 2, 3, 4] ^? element 2
Just 2

>>> [0, 1, 2, 3, 4] & element 2 *~ 100
[0, 1, 200, 3, 4]
```

This isn't useful for much other than selecting a particular list element, and we'll learn an even better way to do this when we talk about **indexed optics**, but it gives us intuition for the next traversal. `elementOf` is a higher-order traversal which allows us to select a specific element from an arbitrary traversal:

```
elementOf :: Traversal' s a -> Int -> Traversal' s a
elementOf :: Fold s a       -> Int -> Fold s a


-- `element` is basically `elementOf traversed`
>>> [0, 1, 2, 3, 4] ^? elementOf traversed 2
Just 2

-- We can get a specific element from a composition of traversals
>>> [[0, 1, 2], [3, 4], [5, 6, 7, 8]] ^? elementOf (traversed . traversed) 6
Just 6

-- Modify the 6th integer from within nested lists:
>>> [[0, 1, 2], [3, 4], [5, 6, 7, 8]]
      & elementOf (traversed . traversed) 6 *~ 100
[[0, 1, 2], [3, 4], [5, 600, 7, 8]]
```

Again, not really sure there are a ton of practical uses to this one so we won't spend much time on it, but it's handy to know it exists, maybe you'll find a use for it!

7.3 Traversal Composition

Traversals compose! We can express complex transformations by composing multiple traversals. Each traversal selects focuses, then rebuilds the structure around the transformed results.

```
>>> import Data.Char (toUpper)
-- Capitalize the first char of every word
>>> "blue suede shoes" & worded . taking 1 traversed %~ toUpper
"Blue Suede Shoes"

-- Find all strings longer than 5 chars
-- then surround each word in that string with '*'
>>> ["short", "really long"]
      & traversed
      . filtered ((> 5) . length)
      . worded
      %~ \s -> "*" ++ s ++ "*"
["short", "*really* *long*"]

-- Add "Rich " to the names of people with more than $1000
>>> (("Ritchie", 100000), ("Archie", 32), ("Reggie", 4350))
      & each
      . filtered ((> 1000) . snd)
      . _1
      %~ ("Rich " ++)

(("Rich Ritchie", 100000), ("Archie", 32), ("Rich Reggie", 4350))
```

Exercises – Simple Traversals

Jump to answers

1. Short answer questions:

- What type of optic do you get when you compose a traversal with a fold?
- Which of the optics we've learned can act as a traversal?
- Which of the optics we've learned can act as a fold?

2. Fill in the blank to complete each expression:

```
>>> ("Jurassic", "Park") & _ .~ "N/A"
("N/A", "N/A")

>>> ("Jurassic", "Park") & both . _ .~ 'x'
("xxxxxxxx", "xxxx")

>>> ("Malcolm", ["Kaylee", "Inara", "Jayne"])
      & _ id traversed %~ take 3
("Mal", ["Kay", "Ina", "Jay"])

>>> ("Malcolm", ["Kaylee", "Inara", "Jayne"])
      & _2 . _ 1 .~ "River"
("Malcolm", ["Kaylee", "River", "Jayne"])

-- This one's tricky!
>>> ["Die Another Day", "Live and Let Die", "You Only Live Twice"]
      & traversed . _ _ 1 . traversed .~ 'x'
[ "Die xxxxxxx Day"
, "Live xxx Let Die"
, "You xxxx Live Twice"
]
```

A bit tougher now!

```
>>> ((1, 2), (3, 4)) & _ +~ 1
((2, 3), (4, 5))

>>> (1, (2, [3, 4])) & _ +~ 1
(2, (3, [4, 5]))

>>> import Data.Char (toUpper)
>>> ((True, "Strawberries"), (False, "Blueberries"), (True, "Blackberries"))
      & _
      %~ toUpper
((True, "STRAWberries"), (False, "Blueberries"), (True, "BLACKberries"))

>>> ((True, "Strawberries"), (False, "Blueberries"), (True, "Blackberries"))
      & _
("Strawberries", "Blueberries", "Blackberries")
```

7.4 Traversal Actions

So far I've been demoing traversals in terms of actions you already know, but traversals have a new action or two of their own. These new actions are based on the `traverse` function from `Traversable`.

A Primer on Traversable

If you're not very familiar with `Traversable` I recommend taking a few hours to try it out and get comfortable. I'll cover it very briefly here; but I really recommend looking at some other resources, and even re-implementing the `Traversable` typeclass for a few types before moving on. It's a bit tricky to understand the first time you come across it and we'll be using it a lot.

The `Traversable` typeclass has two main methods; `sequenceA` and `traverse`. We'll start with `sequenceA`!

```
sequenceA :: (Traversable t, Applicative f)
          => t (f a) -> f (t a)
```

This is a very general function so it can be tough to see what's going on at first. We have two important types, an Applicative `<f>` and a Traversable `<t>`. `sequenceA`'s job is to flip-flop these types around each other in an intelligent way. Here are some concrete versions of this flip-flopping idea:

```
[Maybe a] -> Maybe [a]
```

```
Maybe [a] -> [Maybe a]
```

```
Either e (IO a) -> IO (Either e a)
```

```
[IO a] -> IO [a]
```

```
Map k (State s a) -> State s (Map k a)
```

Note that even though it looks like `sequenceA` is reversible because we can flip-flop both of `([Maybe a] -> Maybe [a])` and `(Maybe [a] -> [Maybe a])`, it's actually not reversible **in general**, it just so happens that each of `Maybe` and `[]` are BOTH Traversable and Applicative, so we can flip-flop back and forth if they're nested within each other. Types like `IO` and `(State s)` aren't traversable, and types like `(Map k)` aren't `Applicative`. This means we can go from `(Map k (State s a) -> State s (Map k a))`, but can't go back!

Let's see some actual examples of what this operation looks like in practice:

```
>>> sequenceA [Just 1, Just 2, Just 3]
Just [1,2,3]

>>> sequenceA [Just 1, Nothing, Just 3]
Nothing

>>> sequenceA $ Just (Left "Whoops")
Left "Whoops"

>>> sequenceA ([pure 1, pure 2, pure 3] :: [IO Int]) >>= print
[1,2,3]
```

Effects like `IO` and `State` are a bit tougher to visualize unfortunately.

`sequenceA` uses the `Applicative` instance of the effect to rebuild the structure, so different `Applicatives` will behave differently. Here's how each type behaves when we're "pulling" it to the outside of a data structure:

`Maybe`
: If any element of the structure is 'Nothing', the result will be 'Nothing'.

`IO`
: Run the `IO` action for each element at once, yielding a structure without any `IO` inside.

List
: Create a list of ALL possible versions of the structure given the permutations available for each slot in the structure. This is sometimes called a non-determinism effect since it can represent *many* possible values at once.

Either
: The Either Applicative short circuits if it ever encounters a `Left` value, returning only that value. If no `Left` values are encountered, it will return the structure inside a `Right`.

`traverse` has the exact same flip-flopping behaviour as `sequenceA`, but with a small tweak, `traverse` takes a function as an argument which maps the contents of the `Traversable` container into an effect, then sequences those effects all at once!

```
traverse :: (Traversable t, Applicative f)
         => (a -> f b) -> t a -> f (t b)
```

You can actually implement `traverse` in terms of `sequenceA` like this:

```
traverse f t = sequenceA (fmap f t)
```

Again, the signature is pretty general and confusing, so here are a few concrete signatures that `traverse` could take on:

```
-- The Traversable is '[ ]', and the effect is 'Maybe'
(String -> Maybe Int) -> [String] -> Maybe [Int]


-- The Traversable is '[ ]', and the effect is 'IO'
(FilePath -> IO String) -> [FilePath] -> IO [String]


-- The Traversable is '((,) String)', and the effect is '[ ]'
(Int -> [Int]) -> (String, Int) -> [(String, Int)]
```

We can see that it takes an effectful **handler** function, runs that on the elements, then sequences the effects to the outside.

Let's see some examples of running traverse too.

For our effectful function we'll use `readMaybe`, it's a parsing function which may fail and return Nothing. We'll use it to parse an `Int` from each `String` in a list. If any parse in the list fails, the whole action will fail with `Nothing`, otherwise we return `Just` the list of results.

In this example the effect is `Maybe` and the `Traversable` is a list.

```
>>> import Text.Read (readMaybe)
>>> :t readMaybe
readMaybe :: Read a => String -> Maybe a
-- 'readMaybe' is polymorphic so we need to specify a concrete result type
>>> traverse readMaybe ["1", "2", "3"] :: Maybe [Int]
Just [1,2,3]
>>> traverse readMaybe ["1", "snark", "3"] :: Maybe [Int]
Nothing
```

For another example we'll see how we can convert a list of `FilePaths` into the corresponding list of file contents. We'll need the `IO` effect to access the file system, but we can `traverse` the `readFile` handler to thread the `IO` to the outside and return a list of results.

```
>>> :t readFile
readFile :: FilePath -> IO String
>>> let results = traverse readFile ["file1.txt", "file2.txt"]
>>> :t results
results :: IO [String]
>>> allContents <- results
>>> allContents
["file 1 contents", "file 2 contents"]
```

Lastly we'll use a list as a 'non-determinism' style of effect and traverse a tuple. Conceptually, we're specifying that there are several possible results from our handler. Running the traversal will return a list of all possible structures.

```
>>> traverse (\n -> [n * 10, n * 100]) ("a", 10)
[("a",100),("a",1000)]
```

I've only shown a few possible effects here, but you can use any effect with an Applicative instance, e.g. `State`, `Reader`, `Writer`, `Concurrently`, etc. The possibilities really are endless.

Traverse on Traversals

So how do optical traversals relate to the `traverse` function? Traversals give us a way to sequence effects from ANY possible focus of a structure, not just those values focused by the Traversable instance!

`traverseOf` is an action which runs traverse over the **focuses** of a **traversal**:

```
-- Specialized signature
traverseOf :: Traversal s t a b -> (a -> f b) -> s -> f t

-- Real signature
traverseOf :: LensLike f s t a b -> (a -> f b) -> s -> f t
```

Since the `traversed` traversal focuses every element of a traversable container, if we run `traverseOf` with `traversed` we end up with identical behaviour to the `traverse` function!

```
>>> :t traverse
traverse
  :: (Traversable t, Applicative f)
  => (a -> f b) -> t a -> f (t b)

>>> :t traverseOf traversed
traverseOf traversed
  :: (Traversable t, Applicative f) =>
     (a -> f b) -> t a -> f (t b)
```

By substituting in different traversals, we can run effectful handlers on arbitrary focuses!

In the tuple example above we could only affect the last slot in a tuple using `traverse`, but if we use our `both` traversal we can run a handler on both:

```
>>> import Text.Read (readMaybe)
-- 'readMaybe' is polymorphic so we need to specify a concrete type
>>> traverseOf both readMaybe ("1", "2") :: Maybe (Int, Int)
Just (1, 2)

>>> traverseOf both readMaybe ("not a number", "2") :: Maybe (Int, Int)
Nothing
```

`traverseOf` works just like `traverse`, so we can use it with any effect which has an Applicative instance. Let's try it with the list effect, a.k.a. non-determinism.

```
>>> import Data.Char (toUpper, toLower)
>>> traverseOf both (\c -> [toLower c, toUpper c]) ('a', 'b')
[('a','b'),('a','B'),('A','b'),('A','B')]
```

By passing a handler which indicates that each character in the tuple could be either uppercase or lowercase we can run the traversal and get a list of all lowercase-uppercase pairings. Pretty cool!

Since we can compose Traversals just like other optics we can get pretty wild with the idea. Composing traversals builds a sort of traversal "tree", just like composing folds does. The Applicative for the list effect will find ALL possible combinations of each branch of the traversal tree, so we end up with something like this:

```
>>> traverseOf
      (both . traversed)
      (\c -> [toLower c, toUpper c])
      ("ab", "cd")
[ ("ab","cd"),("ab","cD"),("ab","Cd"),("ab","CD"),("aB","cd"),("aB","cD"),
, ("aB","Cd"),("aB","CD"),("Ab","cd"),("Ab","cD"),("Ab","Cd"),("Ab","CD"),
, ("AB","cd"),("AB","cD"),("AB","Cd"),("AB","CD")]
```

The result is a list of all possible copies of the structure given that every character may be capitalized or not. There's one structure for every possible combination of different capitalizations.

Let's look at a different type of effect. We can use `Either` for doing simple validation.

```
validateEmail :: String -> Either String String
validateEmail email | elem '@' email = Right email
                    | otherwise      = Left ("missing '@': " <> email)
```

```
>>> traverseOf (traversed . _2) validateEmail
      [ ("Mike", "mike@tmnt.io")
      , ("Raph", "raph@tmnt.io")
      , ("Don", "don@tmnt.io")
      , ("Leo", "leo@tmnt.io")
      ]
Right [("Mike","mike@tmnt.io")
      ,("Raph","raph@tmnt.io")
      ,("Don","don@tmnt.io")
      ,("Leo","leo@tmnt.io")
      ]

>>> traverseOf (traversed . _2) validateEmail
      [ ("Mike", "mike@tmnt.io")
      , ("Raph", "raph.io")
      , ("Don", "don@tmnt.io")
      , ("Leo", "leo@tmnt.io")
      ]
Left "missing '@': raph.io"
```

We can see that the `Either` Applicative short-circuits on the first failure and returns the error. If we want to collect *all* the errors we can use the `Validation` type from `Data.Either.Validation` in the `either` package from hackage:

```
import Data.Either.Validation
validateEmail :: String -> Validation [String] String
validateEmail email | elem '@' email = Success email
                    | otherwise      = Failure ["missing '@': " <> email]

>>> traverseOf (both . traversed) validateEmail'
      (["mike@tmnt.io", "raph@tmnt.io"], ["don@tmnt.io",   "leo@tmnt.io"])
Success (["mike@tmnt.io","raph@tmnt.io"],["don@tmnt.io","leo@tmnt.io"])

>>> traverseOf (both . traversed) validateEmail'
      (["mike@tmnt.io", "raph.io"], ["don@tmnt.io",   "leo.io"])
Failure ["missing '@': raph.io", "missing '@': leo.io"]
```

Whether we're doing IO, non-determinism, validation, or any other type of effect, it's very helpful to be able to sequence effects from deep within our structure all the way back to the outside.

If you prefer having the arguments flipped there's also `forOf`, which otherwise behaves the same as `traverseOf`:

```
forOf :: Traversal s t a b -> s -> (a -> f b) -> f t
```

There's also a `sequenceAOf` of course!

```
sequenceAOf :: Traversal s t (f a) a -> s -> f t
```

We can use it to pull effects deep inside our structures to the outside. This is handy for things like running `IO` or extracting a `Maybe` so we can pattern match on it from the outside of our structure:

```
>>> sequenceAOf _1 (Just "Garfield", "Lasagna")
Just ("Garfield","Lasagna")

>>> sequenceAOf _1 (Nothing, "Lasagna")
Nothing

>>> sequenceAOf (both . traversed) ([Just "apples"], [Just "oranges"])
Just (["apples"],["oranges"])

>>> sequenceAOf (both . traversed) ([Just "apples"], [Nothing])
Nothing
```

Infix traverseOf

`traverseOf` has an infix version: `%%~`

```
traverseOf :: Traversal s t a b -> (a -> f b) -> s -> f t

(%%~)      :: Traversal s t a b -> (a -> f b) -> s -> f t
```

Here's the same parser example from earlier using the infix operator:

```
>>> (("1", "2") & both %%~ readMaybe) :: Maybe (Int, Int)
Just (1, 2)

>>> (("not a number", "2") & both %%~ readMaybe) :: Maybe (Int, Int)
Nothing
```

The operator is close to `%~`, but with an extra `%`.

Using Traversals directly

As it turns out, the `lens` library's Van Laarhoven encoding means that most Traversal's can be used as-is as a custom `traverse` function. Instead of using `traverseOf` or `%%~`, we can often just use the traversal itself!

Here we use `both` directly as though it were `traverse`:

```
>>> both readMaybe ("1", "2") :: Maybe (Int, Int)
Just (1, 2)
```

This is possible because of the definition of the `Traversal` type in the `lens` library:

```
type Traversal s t a b = forall f . Applicative f => (a -> f b) -> s -> f t
```

You'll notice this type matches the type of `traverse` exactly, but uses `s` and `t` instead of `t a` and `t b`.

I'd recommend avoiding this style; it's unnecessarily confusing, unidiomatic, and doesn't translate well to other optics libraries, but it's sometimes useful to know that it exists. It can help you to understand type-errors if you know the underlying representation.

As a fun bit of trivia, the fact that `Traversals` match the `traverse` function on their own means that the `lens` library implementation of `traverseOf` and `%%~` looks like this:

```
traverseOf :: LensLike f s t a b -> (a -> f b) -> s -> f t
traverseOf = id

(%%~) :: LensLike f s t a b -> (a -> f b) -> s -> f t
(%%~) = id
```

By using `id` it simply applies the traversal you give it directly to the other arguments. Pretty crazy! The combinators help a lot with readability though, so I'd recommend you use them consistently.

Exercises – Traversal Actions

Jump to answers

1. Fill in the blanks, you know the drill

```
>>> _ _1 (Nothing, "Rosebud")
Nothing

>>> sequenceAOf (traversed . _1) _
[ [('a', 1) ,('c', 2)]
, [('a', 1) ,('d', 2)]
, [('b', 1) ,('c', 2)]
, [('b', 1) ,('d', 2)]]

-- The ZipList effect groups elements by position in the list
>>> sequenceAOf _ [ZipList [1, 2], ZipList [3, 4]]
ZipList {getZipList = [[1,3],[2,4]]}

>>> sequenceAOf (traversed . _2) [('a', ZipList _), ('b', ZipList _)]
ZipList {getZipList = [[('a',1),('b',3)],[('a',2),('b',4)]]}

>>> import Control.Monad.State
>>> let result = traverseOf _
                           (\n -> modify (+n) >> get)
                           ([1, 1, 1], (1, 1))
>>> runState result 0
(([1,2,3],(4,5)), 5)
```

2. Rewrite the following using the infix-operator for `traverseOf`

```
>>> import Data.Char (toUpper, toLower)
>>> traverseOf
      (_1 . traversed)
      (\c -> [toLower c, toUpper c])
      ("ab", True)
[ ("ab",True)
, ("aB",True)
, ("Ab",True)
, ("AB",True)
]

>>> traverseOf
      (traversed . _1)
      (\c -> [toLower c, toUpper c])
      [('a', True), ('b', False)]
[ [('a',True), ('b',False)]
, [('a',True), ('B',False)]
, [('A',True), ('b',False)]
, [('A',True), ('B',False)]
]
```

3. Given the following data definitions, write a validation function which uses `traverseOf` or `%%~` to validates that the given user has an age value above zero and below 150. Return an appropriate error message if it fails validation.

```
data User =
    User { _name :: String
         , _age  :: Int
         } deriving Show
makeLenses ''User

data Account =
    Account { _id   :: String
            , _user :: User
            } deriving Show
makeLenses ''Account

validateAge :: Account -> Either String Account
validateAge = _
```

7.5 Custom traversals

> This chapter assumes you're working with a Van Laarhoven encoded optics library. If you're using some other library look for a traversal helper, or a Van Laarhoven adapter. Most libraries will have at least one of these. Here are some common adapters:
>
> - `purescript-profunctor-lenses` uses: `wander`
> - `optics` uses: `traversalVL`

Optics look like `traverse`

I bet you're thinking; "Ahhh, now's the part of the book when we learn about the wonderful `traversal` helper; which the benevolent library authors have providentially included for us that we may use it to simply and easily build a traversal that does exactly what we want! How wonderful!", and I must inform you, dear reader, that no such helper exists!

There's a good reason a traversal helper isn't necessary in `lens`; Most of the `lens` library is encoded using something called "Van Laarhoven" style. We'll talk more about the different encoding styles later on, but the gist of it is that Van Laarhoven optics all look something like this:

```
type LensLike f s t a b = (a -> f b) -> (s -> f t)
```

All of the optics we've learned so far can be written as some kind of `LensLike`!

As we just learned, the shape of a LensLike optic matches the type of the regular old `traverse` function from the `Traversable` typeclass:

```
traverse :: (Traversable g, Applicative f)
         => (a -> f b) -> (g a -> f (g b))
```

By generalizing (`s = g a`) and (`t = g b`) then we get the type of a traversal:

```
myTraversal :: (Applicative f)
            => (a -> f b) -> (s -> f t)
```

Different optics have differing constraints on the effect type `<f>`. If we strip off the Applicative constraint we end up with the signature of an arbitrary lens-like:

```
myLensLike :: (a -> f b) -> (s -> f t)
```

Here are a few of the optics we've already learned in their `LensLike` form:

```
type Lens s t a b
  = forall f. Functor f => LensLike f s t a b

type Traversal s t a b
  = forall f. Applicative f => LensLike f s t a b

type Fold s a
  = forall f. (Contravariant f, Applicative f) => LensLike f s t a b
```

And here they are in their fully expanded Van Laarhoven glory:

```
type Lens s t a b
  = forall f. Functor f => (a -> f b) -> (s -> f t)

type Traversal s t a b
  = forall f. Applicative f => (a -> f b) -> (s -> f t)

type Fold s a
  = forall f. (Contravariant f, Applicative f) => (a -> f a) -> (s -> f s)
```

> Note, you'll need `RankNTypes` enabled to write quantified type synonyms like this.

The symmetry we see in these shapes is why we can compose optics of differing types together. Every optic is actually **the exact same type** plus or minus constraints on `<f>`! As optics are composed, the constraints on `<f>` are gathered up.

From these expansions we can see that all the optics we've learned so far are just slightly altered versions of the `traverse` function.

> Most optics are really just `traverse` wearing different pants.
>
> – Chris Penner on optics[1]

As always, this will become much clearer once we've seen an example or two.

Our first custom traversal

Now that we understand that Van Laarhoven optics are just a function which matches a 'traverse-like' signature we can write our own traversals by hand.

Before we move on to writing completely custom traversals, let's get the hang of it by implementing a simplified version of `traversed` which works only on lists, we'll call it `values`:

[1] https://twitter.com/chrislpenner/status/1165401774976466944

```
values :: Traversal [a] [b] a b
```

When we're working with custom traversals I recommend **fully expanding** the type signature, it makes it much clearer how to implement the required code:

```
values :: Applicative f => (a -> f b) -> [a] -> f [b]
```

If you've written the instance of `Traversable` for lists before, this next bit is going to look pretty familiar. That makes sense since this traversal is **exactly** `traverse` specialized to lists. We can see that our `values` traversal accepts a **handler** as an argument which should be run on each value in the list, using this we need to take a list of `a`'s and return a list of `b`'s.

Notice that our function has to work for **any possible Applicative**, that means we can *only* use the methods on the `Applicative` typeclass to interact with the `<f>` context. We're not allowed to "assume" the Applicative will be Maybe, or IO, or any other specific effect.

Now that we've decided on the type signature we want we'll break down our implementation one pattern match at a time. Let's handle the empty list first:

```
values :: Applicative f => (a -> f b) -> [a] -> f [b]
values _ [] = pure []
```

This first case is simple, a list of zero values doesn't have anything for us to focus, so we can just use `pure` to embed an empty list into the Applicative context `<f>`. We don't focus any values here, so the handler is ignored completely.

Next we'll add the more interesting case where we have at least one value in the list:

```
values :: Applicative f => (a -> f b) -> [a] -> f [b]
values _ [] = pure []
values handler (a : as) = liftA2 (:) (handler a) (values handler as)
```

We can break down the remaining work by handling one element at a time and recursing on the remainder. We peel off one `<a>` from the list, we need to run our handler on every element, so we apply it to the current head of the list: `(handler a)`. Now we have an `<f b>`. We can process the rest of the list by calling `values` recursively on the remainder of the list, this gives us `<f [b]>`, so we've got an `<f b>` and a `<f [b]>`, both values are inside the Applicative context `<f>`. No problem though, we can use `liftA2` to lift the list cons operator `(:)` so that it operates within an Applicative context. This will prepend the altered value onto the rest of the transformed list, resulting in the single value `<f [b]>` that our type signature says we need.

If this example is a bit tricky to understand then you may want to practice writing `Traversable` instances for some basic types. There are plenty of resources for this across the web.

Now of course we should verify that our implementation behaves how we expect:

```
>>> ["one", "two", "three"] ^.. values
["one", "two", "three"]

>>> ["one", "two", "three"] & values %~ reverse
["eno","owt","eerht"]

-- We should be able to do type-changing transformations too:
>>> ["one", "two", "three"] & values %~ length
[3,3,5]
```

Great, it works just like `traversed`! Now we can work on a more complex example.

Traversals with custom logic

For this example we're writing bank software! Betcha didn't think you'd ever find yourself writing bank software for fun on the weekend; yet here we are!

We need to model a system for processing transactions on bank accounts. We'll start by modelling our `Transaction` type:

```
data Transaction =
      Withdrawal {_amount :: Int}
    | Deposit    {_amount :: Int}
  deriving Show
makeLenses ''Transaction
```

> Note that it's normally bad-style™ to have field names on types with multiple constructors, but it's okay so long as all constructors have the EXACT same field names and types like they do here.

`makeLenses` is pretty smart in this case; it will generate an `amount` lens which will focus the `Int` inside **either** of Withdrawal or Deposit. It'll focus the amount no matter which constructor is provided.

```
amount :: Lens' Transaction Int
```

We're going to naïvely represent a bank account as a list of transactions:

```
newtype BankAccount =
  BankAccount
    { _transactions :: [Transaction]
    } deriving Show
makeLenses ''BankAccount
```

This `makeLenses` call will generate the lens:

```
transactions :: Lens' BankAccount [Transaction]
```

```
    -- In reality makeLenses actually generates the following iso for us,
    -- But we haven't learned about those yet
    transactions :: Iso' BankAccount [Transaction]
```

So now we have the pieces we need to represent a simple bank account! Let's play around with that a little:

```
>>> let aliceAccount = BankAccount [Deposit 100, Withdrawal 20, Withdrawal 10]

-- Get all transactions
>>> aliceAccount ^.. transactions . traversed
[Deposit {_amount = 100},Withdrawal {_amount = 20},Withdrawal {_amount = 10}]

-- Get the amounts for all transactions
>>> aliceAccount ^.. transactions . traversed . amount
[100,20,10]
```

Case Study: Transaction Traversal

Our manager has given us a task: They need a traversal which focuses on only the dollar amounts of **deposits** within a given account.

Here's the type we want:

```
deposits :: Traversal' [Transaction] Int
```

Remember to expand the type synonym to help us understand what we're actually doing:

```
deposits :: Traversal' [Transaction] Int

-- First expand the simple traversal:
deposits :: Traversal [Transaction] [Transaction] Int Int

-- Finally expand into the 'traverse'-like function:
deposits :: Applicative f
            => (Int -> f Int) -> [Transaction] -> f [Transaction]
```

This traversal will look a lot like the `values` traversal we wrote, except we need to **avoid** running our handler on any `Deposit` transactions. This means we'll need to embed a little logic into our custom traversal. Let's write it one pattern-match at a time just like we did before. First the empty case:

```
deposits :: Applicative f
            => (Int -> f Int) -> [Transaction] -> f [Transaction]
deposits _ [] = pure []
```

The empty list match is **exactly** the same as in `values`! We don't have any transactions to inspect, so we can just return an empty list in the Applicative context.

Now for the case where we handle a `Withdrawal` transaction:

```
deposits handler (Withdrawal amt : rest) =
  liftA2 (:) (pure (Withdrawal amt)) (deposits handler rest)
```

We don't want to **focus** on `Withdrawal` transactions, so we don't run the **handler** on them, but we still need to weave them into the result list somehow. If we didn't, then our traversal would simply **erase** all the `Withdrawal` transactions *entirely* and that's not how traversals should work. Since we want to avoid running the handler, we can lift the `Withdrawal` into the Applicative using `pure` instead and cons it onto the recursively transformed remainder of the list. `pure` will lift the value into the Applicative without running the handler, so it won't run any transformations on it, and so the Withdrawal won't be considered a 'focus' in the traversal we're building.

Lastly we can focus the `Deposit`s.

```
deposits handler (Deposit amt : rest) =
  liftA2 (:) (Deposit <$> (handler amt)) (deposits handler rest)
```

The `handler` accepts an `Int` rather than a transaction so we need to unpack the `amt` from the Deposit, run the handler on it, then rebuild the Transaction object by mapping `Deposit` back onto the transformed Int. After we've rebuilt our transaction we can cons it onto the recursively transformed remains of the list.

The **handler** focuses elements, **pure** ignores them.

Remember you should always test custom traversals after you write them:

```
-- Get all the Deposit transaction amounts:
>>> [Deposit 10, Withdrawal 20, Deposit 30] ^.. deposits
[10,30]

-- Multiply the amounts of all Deposits by 10
>>> [Deposit 10, Withdrawal 20, Deposit 30] & deposits *~ 10
[Deposit {_amount = 100},Withdrawal {_amount = 20},Deposit {_amount = 300}]
```

Looks good! We can edit our **deposits** without affecting the **withdrawals**!

Warning, when focusing a subset of a list like this our first thought is often to look at using a helper like `filter` to implement the traversal; but you need to be careful! `filter` is a destructive operation, it **throws away** any parts of the list which don't match. To understand why this is bad, let's write a **bad** version of `deposits` which uses `filter` to observe the difference.

```
isDeposit :: Transaction -> Bool
isDeposit (Deposit _) = True
isDeposit _           = False

badDeposits :: Traversal' [Transaction] Int
badDeposits handler ts = traverse go (filter isDeposit ts)
  where
    go (Deposit amt) = Deposit <$> handler amt
    go (Withdrawal _) = error "This shouldn't happen"
```

This implementation filters the list to contain only deposit objects, then traverses that list applying the handler to every value.

At first glance everything works great:

```
>>> [Deposit 10, Withdrawal 20, Deposit 30] ^.. badDeposits
[10,30]
```

BUT, remember to also check the behaviour when **setting**:

```
>>> [Deposit 10, Withdrawal 20, Deposit 30] & badDeposits *~ 10
[Deposit {_amount = 100},Deposit {_amount = 300}]
```

Oh no! We only wanted to edit the **deposits**, but in the process our traversal has completely deleted the **withdrawals**! No good. This is why we usually end up writing out something that looks like a traverse instance, it allows us to apply edits to parts of the structure while still being able to put things back together without loss.

We haven't learned them yet; but this version breaks the traversal laws.

The `deposits` traversal is relatively simple, so we could have written it lawfully by relying on existing traversals, but that would have been less educational (and less fun). Now that we've done it by hand, here's a valid implementation of `deposits` which relies on `filtered` instead:

```
deposits :: Traversal' [Transaction] Int
deposits = traversed . filtered isDeposit . amount
```

This version is identical to the one we wrote manually. Typically if you can write a traversal as only the composition of other optics you should. Sometimes however you'll need to write one by hand to add the logic you need.

Exercises – Custom Traversals

Jump to answers

1. Rewrite the `amount` transaction lens manually as the following traversal:

```
amountT :: Traversal' Transaction Int
```

2. Reimplement the `both` traversal over tuples:

```
both :: Traversal (a, a) (b, b) a b
```

3. Write the following custom traversal:

```
transactionDelta :: Traversal' Transaction Int
```

It should focus the amount of the transaction, but should reflect the **change** that the transaction causes to the balance of the account. That is, `Deposits` should be a positive number, but `Withdrawals` should be negative. The traversal should not change the underlying representation of the data.

Here's how it should behave:

```
>>> Deposit 10 ^? transactionDelta
Just 10

-- Withdrawal's delta is negative
>>> Withdrawal 10 ^? transactionDelta
Just (-10)

>>> Deposit 10 & transactionDelta .~ 15
Deposit {_amount = 15}

>>> Withdrawal 10 & transactionDelta .~ (-15)
Withdrawal {_amount = 15}

>>> Deposit 10 & transactionDelta +~ 5
Deposit {_amount = 15}

>>> Withdrawal 10 & transactionDelta +~ 5
Withdrawal {_amount = 5}
```

4. Implement `left`:

```
left :: Traversal (Either a b) (Either a' b) a a'
```

5. **BONUS**: Reimplement the `beside` traversal:

```
beside :: Traversal s t a b
       -> Traversal s' t' a b
       -> Traversal (s,s') (t,t') a b
```

Hint: You can use `traverseOf` or `%%~` to help simplify your implementation!

7.6 Traversal Laws

We got away with avoiding laws for folds, but traversals have a bit more structure so we've got some laws to contend with.

Law One: Respect Purity

The first law of a traversal is that running the traversal with an 'empty' handler shouldn't change anything. We can express it with the following axiom:

```
traverseOf myTraversal pure x == pure x
```

This says that running the `pure` handler (which has no effects) using our traversal should be exactly the same as running `pure` on the original structure without using the traversal at all. `pure` embeds an element into an `Applicative` without causing any effects, so this test asserts that the traversal itself isn't 'adding' any effects or altering any data on its own.

Let's double check a few traversals from the `lens` library to make sure they're not pulling a sneaky on us:

```
-- We need to specify the effect type in the signature:
>>> traverseOf both pure ("don't", "touch") :: [(String, String)]
[("don't","touch")]

>>> pure ("don't", "touch") :: [(String, String)]
[("don't","touch")]
```

It's a good idea to test it out with a few different structures to ensure it's working properly. We can test it with different Applicatives too if we like, but since the implementation is completely polymorphic over the Applicative we know that if it passes the law for one Applicative it **must** pass for **all** of them.

Here's an example using the Maybe effect just for fun:

```
>>> traverseOf both pure ("don't", "touch") :: Maybe (String, String)
Just ("don't","touch")

>>> pure ("don't", "touch") :: Maybe (String, String)
Just ("don't","touch")
```

Great! No changes, just as expected!

This law helps us understand what's going to happen when we run a traversal, we expect it to run our handler on the focused elements, then thread the effects through to the outside, but do nothing else!

Here's a traversal that boldly BREAKS this law, how inconsiderate!

```
badTupleSnd :: Traversal (Int, a) (Int, b) a b
badTupleSnd handler (n, a) =
    liftA2 (,) (pure (n + 1)) (handler a)

>>> traverseOf badTupleSnd pure (10, "Yo")
(11,"Yo")

>>> pure (10, "Yo")
(10,"Yo")
```

Uh-Oh! Our traversal has an unexpected side-effect of changing the number in the first slot.

This is a pretty silly example, but the idea is that traversals shouldn't be messing around where they don't belong. Let the handlers do the updating and don't mess with state yourself!

Law Two: Consistent Focuses

This law looks a little crazy the way it's written in the official docs; bonus points if you can work out what this is supposed to mean:

```
import Data.Functor.Compose

fmap (traverseOf myTrav f) . traverseOf myTrav g $ x
==
getCompose . traverseOf myTrav (Compose . fmap f . g) $ x
```

I won't make you stare at it for too long; the meaning of this law is that running a handler over a traversal should **never change which elements are focused** in the traversal. We can express this law through an equality check by stating that running a traversal twice in a row with different handlers should be equivalent to running it **once** with the composition of those handlers. The law above includes the effects of the handler in its law, but for a simpler version we can just consider the updates to the focused value itself by swapping in `%~` instead of `traverseOf`. This is what the above law would simplify down to if we chose `Identity` as the effect.

The simpler subset of this law looks like this:

```
x & myTraversal %~ f
  & myTraversal %~ g
==
x & myTraversal %~ (g . f)
```

The results of the first expression **must** equal the second expression for all possible handlers `<f>` and `<g>`, as well as all possible structures `<x>` of the appropriate type. If you can find any counter-example then the law doesn't hold.

In essence this law states that the traversal should never **change which elements it focuses** due to alterations on those elements.

`both` is a law-abiding traversal, so we expect to see the equality hold for any structure or handlers we can dream up:

```
>>> (0, 0) & both %~ (+10)
           & both %~ (*10)
(100, 100)

==

>>> (0, 0) & both %~ (*10) . (+10)
(100, 100)
```

It's tough to **prove** a law by testing examples, there may always be some law-breaking example you forgot to try, but if you're able to find even a single counter-example you **know** it's an unlawful traversal. There are various property-based testing libraries for **QuickCheck** and **Hedgehog** which can help you test the laws of your custom optics if you like!

Did you know one of the traversals we've already learned actually breaks this law? It turns out that `filtered` is a law-breaking traversal!

```
>>> 2 & filtered even %~ (+1)
      & filtered even %~ (*10)
3

==

>>> 2 & filtered even %~ (*10) . (+1)
30
```

Uh oh! `3` isn't the same as `30`! `filtered` breaks this law because updates can change which elements are in focus! This means that it might actually matter whether we run the traversal twice, instead of once with the composition of our handlers.

Good Traversal Bad Traversal

So what's the deal? Why is there a traversal included in the `lens` library that blatantly breaks one of the laws? As it turns out, the traversal laws are more guidelines than anything. They help you to reason about behaviour, but nothing is going to go horribly wrong if you break them. In fact, none of the combinators in the `lens` library actually rely on coherence to these laws. Some traversals like `filtered` are **very useful** but **can't** be encoded in a legal way in Haskell, at least not without some

advanced dependent types. I use law-breaking traversals all the time and have written several of my own without ever running into difficulties. It's a lot like jazz; you need to learn the laws before you can break them, but sometimes breaking the laws can lead to exciting new possibilities.

Exercises – Traversal Laws

Jump to answers

1. `worded` is a law-breaking traversal! Determine which law it breaks and give an example which shows that it doesn't pass the law.
2. Write a custom traversal which breaks the first law. Be as creative as you like!
3. Write a custom traversal which breaks the second law. Be as creative as you like!
4. For each of the following traversals, decide whether you think it's lawful or not. If they're unlawful come up with a counter-example for one of the laws.

- `taking`
- `beside`
- `each`
- `lined`
- `traversed`

7.7 Advanced manipulation

partsOf

There's one particular traversal combinator that's powerful enough to deserve a whole section for itself: `partsOf`!

`partsOf` is a higher-order combinator which converts a traversal into a lens over a list of the traversal's focuses. Here's the type:

```
partsOf :: Traversal' s a -> Lens' s [a]
```

The lens generated by `partsOf` takes all the focuses of the provided traversal and packs them into a list for you to manipulate however you like. Then, it takes the modified list and maps each element back into the original structure. This definitely seems a bit like magic when you first see it, but it's just the amazing power of optics!

To get a feel for it, let's build a lens with `partsOf` and use it as a **getter**:

```
>>> [('a', 1), ('b', 2), ('c', 3)] ^. partsOf (traversed . _1)
"abc"

>>> [('a', 1), ('b', 2), ('c', 3)] ^. partsOf (traversed . _2)
[1,2,3]
```

You can see this looks a lot like a `toListOf` fold, because we're doing essentially the same thing: viewing the focuses of a traversal as a list! The difference from `toListOf` is that this is a lens, not a fold! That means we can edit the list and write results back! Let's see what happens if we use it as a setter:

```
-- We can "set" the lens to a list to replace the corresponding elements
>>> [('a', 1), ('b', 2), ('c', 3)]
      & partsOf (traversed . _1) .~ ['c', 'a', 't']
[('c',1),('a',2),('t',3)]

-- Any 'extra' list elements are simply ignored
>>> [('a', 1), ('b', 2), ('c', 3)]
      & partsOf (traversed . _1) .~ ['l', 'e', 'o', 'p', 'a', 'r', 'd']
[('l',1),('e',2),('o',3)]

-- Providing too few elements will keep the originals
>>> [('a', 1), ('b', 2), ('c', 3)]
      & partsOf (traversed . _1) .~ ['x']
[('x',1),('b',2),('c',3)]
```

We're welcome to use ANY list manipulation functions we want! Just keep in mind the rules regarding replacement:

- If the list has more elements than the traversal, the extras will be ignored
- If the list has fewer elements than the traversal, unmatched portions of the traversal will be unaffected

It's because of these behaviours that `partsOf` isn't a polymorphic lens, we might need to re-use some of the original elements so we can't change their type!

Here are some other fun functions we can try:

```
>>> [('a', 1), ('b', 2), ('c', 3)]
      & partsOf (traversed . _1) %~ reverse
[('c',1),('b',2),('a',3)]

>>> [('o', 1), ('o', 2), ('f', 3)]
      & partsOf (traversed . _1) %~ sort
[('f',1),('o',2),('o',3)]

-- See if you can follow along with how this one works:
>>> [('o', 1), ('o', 2), ('f', 3)]
      & partsOf (traversed . _1) %~ tail
[('o',1),('f',2),('f',3)]
```

We can get some seriously trippy results with this technique. Check this out:

```
>>> ("how is a raven ", "like a ", "writing desk")
      & partsOf (each . traversed) %~ unwords . sort . words
("a a desk how is"," like r","aven writing")
```

We can collect the characters into a list (a.k.a. String), then split them all into words, sort them, then write back each individual character to a slot!

Let's break this one down a little to fully understand what's going on.

```
("how is a raven ", "like a ", "writing desk")

-- 'each' collects each string
"how is a raven " "like a " "writing desk"

-- 'traversed' focuses each individual character
'h' 'o' 'w' ' ' 'i' 's' ...

-- The 'partsOf' around (each . traversed) collects the focuses into a list:
"how is a raven like a writing desk"

-- 'words' splits the list into words
["how","is","a","raven","like","a","writing","desk"]

-- 'sort' sorts the words
["a","a","desk","how","is","like","raven","writing"]

-- unwords flattens back to a String (a.k.a. list of characters)
"a a desk how is like raven writing"
```

```
-- 'partsOf' then maps each character back to a position
  a_a_desk_how_is    _like_r    aven_writing"
  vvvvvvvvvvvvvvv    vvvvvvv    vvvvvvvvvvvv
("how is a raven ", "like a ", "writing desk")

-- Result:
("a a desk how is"," like r","aven writing")
```

Note that **placement matters**. Just like other higher-order optics the position of `partsOf` within the expression determines what is grouped into a list and how it's handled. Note the difference between these expressions:

```
-- Collect 'each' tuple element into a list, then traverse that list
>>> ("abc", "def") ^.. partsOf each . traversed
["abc","def"]

-- Collect each tuple element, then traverse those strings
-- collecting each character into a list.
>>> ("abc", "def") ^.. partsOf (each . traversed)
["abcdef"]
```

You can use `partsOf` to edit elements using their neighbours as context.

Here's a transformation which replaces each number with its percentage of the sum of ALL numbers in the structure:

```
>>> [('a', 1), ('b', 2), ('c', 3)]
      & partsOf (traversed . _2)
      %~ \xs -> (/ sum xs) <$> xs
[('a',0.16666), ('b',0.33333), ('c',0.5)]
```

Here's a breakdown of this one:

```
[('a', 1), ('b', 2), ('c', 3)]

-- Focus each number with 'partsOf (traversed . _2)'
[1, 2, 3]

-- Substitute into the function:
(/ sum [1, 2, 3]) <$> [1, 2, 3]

-- Evaluate
[0.16666,0.33333,0.5]

-- partsOf maps back each element into the original structure
    0.16666   0.33333     0.5
       v         v         v
[('a', 1), ('b', 2), ('c', 3)]

-- Result:
[('a',0.16666), ('b',0.33333), ('c',0.5)]
```

Polymorphic partsOf

We've seen how powerful partsOf can be, but it can be a bit annoying that we can't change the type of the focus. As we learned earlier, this is because we may need to use some elements from the original structure in the result if the list we're `setting` doesn't contain enough elements to replace every value in the traversal. There is, of course, a way around this! Here's `unsafePartsOf`:

```
unsafePartsOf :: Traversal s t a b -> Lens s t [a] [b]
```

It's called **unsafe** for a reason, if we set or modify with the wrong number of list elements it'll **crash** catastrophically!

```
>>> [('a', 1), ('b', 2), ('c', 3)]
      & unsafePartsOf (traversed . _1) .~ [True, False]
[ (True,1)
, (False,2)
, (*** Exception: unsafePartsOf': not enough elements were supplied
```

Whoops! I'd recommend not using this with `set`, as it's pretty easy to get it wrong by accident. You can use it safely with `over` if you ensure your modifications maintain the length of the list.

If we use the correct number of elements (or more) everything works out:

```
>>> [('a', 1), ('b', 2), ('c', 3)]
      & unsafePartsOf (traversed . _1) .~ [True, False, True]
[(True,1),(False,2),(True,3)]
```

This example pairs each element with its successor (if it exists).

```
>>> [('a', 1), ('b', 2), ('c', 3)]
      & unsafePartsOf (traversed . _1)
      %~ \xs -> zipWith (,) xs ((Just <$> tail xs) ++ [Nothing])
[(('a',Just 'b'),1),(('b',Just 'c'),2),(('c',Nothing),3)]
```

partsOf and other data structures

We've only scratched the surface of the "real" applications for `partsOf`. It can take some practice to understand when and where to use it, but it's very satisfying to recognize a use-case when one comes up!

I've only shown relatively simple example so far, but it works on arbitrarily complex traversals, so go ham wild!

I discovered one very practical use-case for `partsOf` when helping a friend with an issue they were having at work. They had a tree structure filled with user IDs and they wanted to replace each ID with a full `User` object from their database. They had planned to traverse a lookup function over the tree, but that meant running a separate database query for every single user in the tree! They had a more efficient batch lookup function, but it accepted a LIST of user IDs, not a tree!

Here's the problem laid out with types:

```
userIds :: Tree UserId
lookupUsers :: [UserId] -> IO [User]

treeLookup :: Tree UserId -> IO (Tree UserO
treeLookup = ???
```

The problem seemed intractable at first, but then I remembered `partsOf`! Using `partsOf` with our traversal actions makes the solution trivial:

```
treeLookup :: Tree UserId -> IO (Tree User)
treeLookup = traverseOf (unsafePartsOf traversed) lookupUsers
```

We know that our `lookupUsers` function should always return the same number of users as the number of user IDs it's passed, so this invariant makes the code safe (though it wouldn't hurt to add a test or two).

Exercises – partsOf

Jump to answers

What fits in the blanks?

```
-- Viewing
>>> [1, 2, 3, 4] ^. partsOf (traversed . filtered even)
_

>>> ["Aardvark", "Bandicoot", "Capybara"]
      ^. traversed . partsOf (taking 3 traversed)
_

>>> ([1, 2], M.fromList [('a', 3), ('b', 4)])
      ^. partsOf _
[1,2,3,4]

-- Setting
>>> [1, 2, 3, 4]
      & partsOf (traversed . _) .~ [20, 40]
[1,20,3,40]

>>> ["Aardvark", "Bandicoot", "Capybara"]
      & partsOf _ .~ "Kangaroo"
["Kangaroo","Bandicoot","Capybara"]

>>> ["Aardvark", "Bandicoot", "Capybara"]
      & partsOf (traversed . traversed) .~ "Ant"
["Antdvark","Bandicoot","Capybara"]

-- Modifying
-- Tip: Map values are traversed in order by KEY
>>> M.fromList [('a', 'a'), ('b', 'b'), ('c', 'c')]
      & partsOf traversed %~ \(x:xs) -> xs ++ [x]
M.fromList [('a','b'),('b','c'),('c','a')]

>>> ('a', 'b', 'c') & partsOf each %~ reverse
('c', 'b', 'a')

-- Bonus
>>> [1, 2, 3, 4, 5, 6]
      & partsOf (taking 3 traversed) %~ reverse
```

```
[3,2,1,4,5,6]
```

```
>>> ('a', 'b', 'c')
      & unsafePartsOf each %~ \xs -> fmap ((,) xs) xs
(("abc",'a'),("abc",'b'),("abc",'c'))
```

8. Indexable Structures

8.1 What's an "indexable" structure?

`lens` is a "batteries included" library; this means it includes not only the basic optic types but a lot of useful helpers for common tasks. One of those tasks is working with indexable data structures!

What do we mean when we talk about an **indexable** data structure? Indexable structures store values at **named locations** which can be identified by some **index**. That is, an **index** represents a **specific location** within a data structure where a value **might** be stored.

Lists name their slots using a numeric offset, Maps use a key. Even certain tree structures can label their values, for instance a Rose Tree (a tree with lists of children at each node) can reference a specific node in the tree using a list of integers.

Each of these data structures has the same notion of getting or setting the value at a specific **index**; but there's no unified interface for doing these things in the Prelude. Each structure has provided its own separate methods for getting or setting values. This is not only *annoying*, it also means that we can't easily generalize functions over these sorts of data structures and sometimes have to re-implement the same functionality more than once.

Here are some examples of some common data structures and the sorts of **index based operations** they support:

Lists allow getting values by an `Int` index:

```
>>> let xs = ["Henry I", "Henry II", "Henry III"]
-- Getting values at an index uses !!
>>> xs !! 0
"Henry I"
>>> xs !! 1
"Henry II"
>>> xs !! 3
"*** Exception: Prelude.!!: index too large
```

Lists don't have a built-in way to update values at an index; but we'll sort that out ASAP.

Maps allow getting, adding, updating, and deleting values at keys.

```
>>> let turtles = M.fromList [("Leo", "Katanas"), ("Raph", "Sai")]

-- Maps use `lookup` to 'get' values at an index
>>> M.lookup "Leo" turtles
Just "Katanas"

-- Maps use `adjust` to update the value at an index
>>> M.adjust ("Two " <>) "Leo" turtles
fromList [("Leo","Two Katanas"),("Raph","Sai")]
```

You'll notice that not only are the interfaces for maps and lists different, lists are missing a way to update values at specific indexes. The one method they do have for retrieving elements is **partial**, it might crash our program. This is quite frankly embarrassing, optics to the rescue!

8.2 Accessing and updating values with 'Ixed'

The `Ixed` Class

The `lens` library has a way of unifying the interface to all these data structures. It provides a few typeclasses and type families which structures can implement to take advantage of the ecosystem, the most important is the `Ixed` typeclass:

```
class Ixed m where
  ix :: Index m -> Traversal' m (IxValue m)
```

The `Ixed` class contains the `ix` method; it takes an index for the structure in question and builds a traversal over the value at that index so we can get or set it as we please. `ix` builds a **traversal** rather than something like a **lens** because there may not be a value at the specified location. Perhaps we've provided an index that's out of the range of a list, or maybe we tried to access a key which is missing from a Map. The `ix` traversal won't focus anything if there isn't a value at the provided index.

You'll notice the type signature also mentions `Index m` and `IxValue m`; where are these coming from? `Index` and `IxValue` are both **type families**, which tell us which index type or value type to use for a given structure.

If you haven't seen type families before, they're really just functions that operate on types! Here's what the `Index` and `IxValue` type family implementations look like for Lists and Maps:

```
-- The index for lists is `Int`
type instance Index [a] = Int
-- The index for Maps is the key type
type instance Index (Map k a) = k

type instance IxValue [a] = a
type instance IxValue (Map k a) = a
```

`type instance Index` indicates we're providing an implementation of the `Index` type function for a specific input type. We then specify that the index type of a list is always an `Int`, and the index type of a `Map` is the same as the **key** type of that map.

The `IxValue` type of a list is unsurprisingly just the type of the contents of the list, and the `IxValue` of a Map is the value type of that map.

Pretty straight-forward really, using type families like this rather than relying on higher-kinded types allows us to use the same interface for monomorphic types like bytestrings and text as well.

```
type instance Index ByteString = Int
type instance Index Text = Int

type instance IxValue Text = Char
type instance IxValue ByteString = Word8
```

You'll rarely, if ever, need to write implementations of these yourselves as all the main types you'll regularly use are provided for you already. However if you create your own indexed structure you can easily implement these type classes and type families yourself.

Accessing and setting values with `ix`

Okay, so how does it look to actually use `ix`? Let's see! Remember, we provide it with an **index** and it builds a **traversal** over the **value** at that location.

```
>>> let humanoids = ["Borg", "Cardassian", "Talaxian"]

-- Get the value at index 1:
>>> humanoids ^? ix 1
Just "Cardassian"

-- There's no value at index 10 so the traversal doesn't focus anything
>>> humanoids ^? ix 10
Nothing
```

```
-- It's a traversal, so we can 'set' new values at that index
>>> humanoids & ix 1 .~ "Vulcan"
["Borg","Vulcan","Talaxian"]

-- A 'set' will do nothing if the given index doesn't have a value
>>> humanoids & ix 10 .~ "Romulan"
["Borg","Cardassian","Talaxian"]
```

Traversals handle "missing" elements by simply not focusing anything, so if the value at an index doesn't exist then `^?` will return `Nothing`. Surprisingly we can't "set" missing elements to insert them into a list or Map using `ix`; in order to be consistent with the traversal laws the `ix` traversal can't add or remove focuses. Besides, if we were to try to set the tenth element of the `humanoids` list, what would fill in all the newly empty slots in between?

Here's how we can use `ix` to interact with a Map:

```
>>> let benders =
      M.fromList [("Katara", "Water"), ("Toph", "Earth"), ("Zuko", "Fire")]

-- Get the value at key "Zuko"
>>> benders ^? ix "Zuko"
Just "Fire"

-- If there's no value at a key, the traversal returns zero elements
>>> benders ^? ix "Sokka"
Nothing

-- We can set the value at a key, but only if that key already exists
>>> benders & ix "Toph" .~ "Metal"
fromList [("Katara","Water"),("Toph","Metal"),("Zuko","Fire")]

-- Setting a non-existent element of a Map does NOT insert it.
>>> benders & ix "Iroh" .~ "Lightning"
fromList [("Katara","Water"),("Toph","Earth"),("Zuko","Fire")]
```

Again we see we can access and edit values at indices which exist, but operations have no effect if a value doesn't already exist at an index in the structure. Even setting a value at an index does not allow it to be inserted. `ix` can neither create nor delete slots in an indexed structure, doing so would actually break the second Traversal law! Don't worry though, we'll learn a perfectly law abiding way to insert into Maps soon.

Indexed Structures

Many data structures have a reasonable `Ixed` instance provided for you already. Here's an incomplete list of structures and their respective index + value types:

Structure	Index	Value
`[a]`	`Int`	`a`
`NonEmpty a`	`Int`	`a`
`Seq a`	`Int`	`a`
`Vector a`	`Int`	`a`
`Set a`	`a`	`()`
`Map k a`	`k`	`a`
`Identity a`	`()`	`a`
`Maybe a`	`()`	`a`
`Tree a`	`[Int]`	`a`
`Text`	`Int`	`Char`
`ByteString`	`Int`	`Word8`
`(e -> a)`	`e`	`a`

Indexing monomorphic types

Indexing works with monomorphic types like `Text` and `ByteString` just as you'd expect:

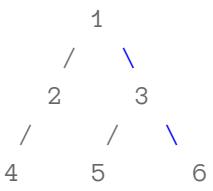
```
>>> ("hello" :: T.Text) ^? ix 0
Just 'h'

>>> ("hello" :: T.Text) ^? ix 10
Nothing

>>> ("hello" :: T.Text) & ix 0 .~ 'j'
"jello"

-- Word8's are shown as integers
>>> ("hello" :: BS.ByteString) ^? ix 0
Just 104

>>> ("hello" :: BS.ByteString) ^? ix 10
Nothing

-- We can edit a Word8 within a ByteString as though it's an integer.
>>> ("hello" :: BS.ByteString) & ix 0 +~ 2
"jello"
```

```
-- Remember, we can always 'traverse' a traversal using effectful handlers!
>>> ("hello" :: T.Text) & ix 1 %%~ \_ -> ("aeiou" :: [Char])
[ "hallo"
, "hello"
, "hillo"
, "hollo"
, "hullo"
]
```

Indexing stranger structures

Let's round out our exploration with a quick look at a few of the structures whose index types may not be entirely clear.

Considering the following tree:



We can encode this as a Rose Tree using `Data.Tree` like so:

```
import Data.Tree

tree :: Tree Int
tree = Node 1 [ Node 2 [Node 4 []]
              , Node 3 [Node 5 [], Node 6 []]]
```

If you look at the chart you'll see that the index for a `Tree` is a list of integers. Starting at the root of the tree we pick the child node which corresponds to the first integer in the list, then descend into that child and treat it as the root, then continue indexing downward using the tail of the list. Eventually we'll have an empty list of indexes which indicates that we wish to focus the value stored in the current node. If the provided child number doesn't exist at any point, the traversal will fail and will return zero elements.

```
-- The empty list represents the value at the root:
>>> tree ^? ix []
Just 1

-- 0 represents the first child of the root
>>> tree ^? ix [0]
Just 2

-- 0, 0 descends into the first child twice then focuses the value at that node
>>> tree ^? ix [0, 0]
Just 4

>>> tree ^? ix [1, 0]
Just 5

>>> tree ^? ix [1, 1]
Just 6

-- Invalid paths simply return an empty traversal
>>> tree ^? ix [5, 6]
Nothing
```

We can also update existing tree elements using `ix`, just as we can with lists and Maps.

Maybe you saw on the bottom of the chart that we can actually use `ix` to index into functions! This one's more of a fun trick than something you should use in a production codebase, but maybe you can show it off at your next dinner party. The index to a function `(e -> a)` is the argument `(e)`. We can access the value stored at a specific `(e)` value by running the function on it; and can edit the value of a function by wrapping it with a function which intercepts or edits values from the original. This logic is provided for us in the `Ixed` instance.

```
-- Indexing into a function runs the function with the index as input
>>> reverse ^? ix "Stella!"
Just "!alletS"

-- We can "set" or traverse individual results of a function!
-- Here we overwrite the function's output at the input value "password"
-- so it instead returns a new value.
>>> let specialReverse = reverse & ix "password" .~ "You found the secret!"
>>> specialReverse "password"
```

```
"You found the secret!"
-- The function is unaffected at all other inputs
>>> specialReverse "Stella!"
"!alletS"
```

8.3 Inserting & Deleting with 'At'

Map-like structures

Now we can **access** or **alter** elements inside structures, but **inserting** and **deleting** elements are also very common operations.

```
>>> let turtles = M.fromList [("Leo", "Katanas"), ("Raph", "Sai")]
>>> M.insert "Mikey" "Nunchaku" turtles
fromList [("Leo","Katanas"),("Mikey","Nunchaku"),("Raph","Sai")]

>>> M.delete "Leo" turtles
fromList [("Raph","Sai")]
```

As mentioned earlier, allowing deletion or insertion within a traversal like `ix` would break the second traversal law by changing the number of 'slots' the traversal accesses. What's another way we could encode this information?

The `At` typeclass allows focusing values within map-like structures which allow arbitrary insertion or deletion.

```
class At where
  at :: Index m -> Lens' m (Maybe (IxValue m))
```

For comparison, here's `ix` and `at` side-by-side:

```
ix :: Index m -> Traversal' m (IxValue m)
at :: Index m -> Lens'      m (Maybe (IxValue m))
```

They're very similar, but `ix` is a **traversal** over the value itself; whereas `at` provides a **lens** over a `Maybe` wrapped value. This subtle distinction provides some new options to us.

To **insert** or **replace** an element we can **set** a value wrapped in `Just`; to **delete** we can set the focus to `Nothing`. We can update a value arbitrarily using `over` and providing a function from `Maybe a -> Maybe a`. This provides us with flexibility that `ix` does not. We've encoded the existence of a value into the focus of the lens itself. This means we can always focus a value with our **lens**, and the value may simply be `Nothing` if missing.

Unfortunately we can't write an `At` instance for lists since we can't insert elements arbitrarily. You can't insert a 10th element unless you have a 9th! `At` is used for Map-like structures where elements can be added or deleted at will.

Here's how we can use `At` with Maps:

```
>>> let benders =
      M.fromList [("Katara", "Water"), ("Toph", "Earth"), ("Zuko", "Fire")]

-- Since 'at' creates a lens, we use `^.` instead of `^?`
>>> benders ^. at "Zuko"
Just "Fire"

>>> benders & at "Zuko" .~ Nothing
fromList [("Katara", "Water"), ("Toph", "Earth")]

>>> benders & at "Iroh" .~ Just "Lightning"
fromList [ ("Iroh", "Lightning")
         , ("Katara", "Water")
         , ("Toph", "Earth")
         , ("Zuko", "Fire")
         ]
```

Because inserting new values using:

```
at i .~ Just newValue
```

is a very common pattern, the `lens` library provides a helper called `?~` which wraps values in `Just` before setting the value:

```
(?~) :: Traversal s t a (Maybe b) -> b -> s -> t
```

Using this operator we can avoid writing out the `Just` when inserting elements.

```
>>> benders & at "Iroh" ?~ "Lightning"
fromList [ ("Iroh", "Lightning")
         , ("Katara", "Water")
         , ("Toph", "Earth")
         , ("Zuko", "Fire")
         ]

-- We can use this with other optics if we want,
-- but it's usually used with `at`
>>> (1, 2) & both ?~ "twins!"
(Just "twins!", Just "twins!")
```

Deletion is a common operation too! The `lens` library provides `sans` as a helper to easily delete elements:

```
sans :: At m => Index m -> m -> m
sans k = at k .~ Nothing
```

`sans` is just short-hand for setting the value `at` an index to `Nothing`:

```
>>> benders
fromList [ ("Katara", "Water")
         , ("Toph", "Earth")
         , ("Zuko", "Fire")
         ]

>>> sans "Katara" benders
fromList
    [ ("Toph", "Earth")
    , ("Zuko", "Fire")
    ]
```

Manipulating Sets

The `At` typeclass works with **Sets** as well as Maps. One way to imagine a Set is as a map where the set elements are keys: `(Map v ())`. Using unit: `()` as the value means the only real information stored in the Map is whether a value exists or not, so the `IxValue` for a Set is just the unit type: `()`. A `Just ()` represents that the value exists at the provided index, and a `Nothing` represents that the value is missing from the Set.

```
import qualified Data.Set as S

primes :: S.Set Int
primes = S.fromList [2, 3, 5, 7, 11, 13]

>>> primes ^? ix 5
Just ()

>>> primes ^? ix 4
Nothing

>>> primes & at 17 ?~ ()
fromList [2, 3, 5, 7, 11, 13, 17]

>>> sans 5 primes
fromList [2, 3, 7, 11, 13]

-- BTW we can use & to chain uses of 'sans'
>>> primes & sans 5
           & sans 7
           & sans 11
fromList [2, 3, 13]
```

Exercises – Indexable Structures

Jump to answers

1. Fill in the blanks

```
>>> ["Larry", "Curly", "Moe"]
      & _ 1 .~ "Wiggly"
["Larry","Wiggly","Moe"]

>>> let heroesAndVillains =
          M.fromList [("Superman", "Lex"), ("Batman", "Joker")]
>>> heroesAndVillains & _ "Spiderman" .~ Just "Goblin"
M.fromList [("Batman","Joker"),("Spiderman","Goblin"),("Superman","Lex")]

>>> _ "Superman" heroesAndVillains
M.fromList [("Batman","Joker")]

>>> S.fromList ['a', 'e', 'i', 'o', 'u']
      & at 'y' _ ()
```

```
      & at 'i' .~ _
S.fromList "aeouy"
```

2. Use `ix` and `at` to go from the input to the output:

```
input = M.fromList [("candy bars", 13), ("soda", 34), ("gum", 7)]
output = M.fromList [("candy bars",13),("ice cream",5),("soda",37)]
```

8.4 Custom Indexed Data Structures

I mentioned earlier that we can implement the indexing typeclasses ourselves if we want to use `ix` or `at` for one of our own types. Let's see how to implement these by writing a new list-type which implements an alternative version of indexing.

Custom Ixed: Cyclical indexing

Let's build a newtype over lists which handles indexing as though the list were infinitely cycled. For example, if we had the list:

```
>>> xs = Cycled ['a', 'b', 'c']
```

When we look for index `3` we reach the end of the list, if we 'cycle' by continuing to count at the beginning of the list, then we'd end at `'a'`.

Here's how we want it to behave:

```
>>> Cycled ['a', 'b', 'c'] ^? ix 1
Just 'b'

>>> Cycled ['a', 'b', 'c'] ^? ix 3
Just 'a'

>>> Cycled ['a', 'b', 'c'] ^? ix 10
Just 'b'

-- Why not wrap around on negative numbers too?
>>> Cycled ['a', 'b', 'c'] ^? ix (-1)
Just 'c'

-- We expect setting to work as well
>>> Cycled ['a', 'b', 'c'] & ix 0 .~ '!'
```

```
Cycled "!bc"

>>> Cycled ['a', 'b', 'c'] & ix 10 .~ '!'
Cycled "a!c"

>>> Cycled ['a', 'b', 'c'] & ix (-1) .~ '!'
Cycled "ab!"
```

Let's see if we can build it to spec!

`ix` is already defined for lists, so we need a `newtype` wrapper so we can define our own instance.

```
newtype Cycled a = Cycled [a]
  deriving Show
```

Next we need to define what the `Index` and `IxValue` for our type are going to be. In order to define these we'll need to enable the `TypeFamilies` language extension:

```
{-# LANGUAGE TypeFamilies #-}

type instance Index (Cycled a) = Int
type instance IxValue (Cycled a) = a
```

I've chosen the boring but expected types; `Int` for indexing and of course the value we want to get out is simply `a`.

Next we can implement the `Ixed` typeclass!

```
instance Ixed (Cycled a) where
  ix :: Int -> Traversal' (Cycled a) a
  ix i handler (Cycled xs) =
    Cycled <$> (traverseOf (ix (i `mod` length xs)) handler xs)
```

Remember from the chapter on custom traversals that the `Traversal'` type synonym can be expanded like this:

```
ix :: Int -> Traversal' (Cycled a) a
-- expands into:
ix :: Applicative f => Int -> (a -> f a) -> Cycled a -> f (Cycled a)
```

Does the implementation make a little more sense if we show that type signature instead?

```
instance Ixed (Cycled a) where
  ix :: Applicative f => Int -> (a -> f a) -> Cycled a -> f (Cycled a)
  ix i handler (Cycled xs) =
    Cycled <$> (traverseOf (ix (i `mod` length xs)) handler xs)
```

To get the 'cycling' effect I simply take the modulus of the requested index by the length of the list, this causes the index to 'wrap around' when it overflows and guarantees the number I end up with resides within the list.

This implementation unwraps the incoming `Cycled` newtype so that we can get at the length of the underlying list (which I need to compute the modulus), then we simply rely on the `ix` instance for normal lists to run the rest of the traversal for us, packing the result back into a `Cycled` at the end.

We can verify it works as we planned:

```
>>> Cycled ['a', 'b', 'c'] ^? ix 1
Just 'b'

>>> Cycled ['a', 'b', 'c'] ^? ix 10
Just 'b'

>>> Cycled ['a', 'b', 'c'] ^? ix (-1)
Just 'c'

>>> Cycled ['a', 'b', 'c'] & ix 0 .~ '!'
Cycled "!bc"

>>> Cycled ['a', 'b', 'c'] & ix 10 .~ '!'
Cycled "a!c"

>>> Cycled ['a', 'b', 'c'] & ix (-1) .~ '!'
Cycled "ab!"
```

Custom At: Address indexing

In this section I'll show you how we can actually index any arbitrary type; it doesn't need to be a traditional "container type". So long as we can implement the typeclass it'll all work!

Consider a data type which represents an `Address`. Addresses are complicated, and consist of many pieces. A given address may be missing any individual piece of information[1]. Let's model this in Haskell:

[1] https://www.mjt.me.uk/posts/falsehoods-programmers-believe-about-addresses/

```
data Address =
    Address { _buildingNumber  :: Maybe String
            , _streetName      :: Maybe String
            , _apartmentNumber :: Maybe String
            , _postalCode      :: Maybe String
            } deriving Show
makeLenses ''Address
```

Any individual piece of our address may be missing; or perhaps it just hasn't been specified yet. Now let's define our type family implementations for `Index` and `IxValue`:

```
type instance Index Address   = ??
type instance IxValue Address = String
```

The `IxValue` is clear, each piece is a String, but what can we use as a type to uniquely identify each field? We could use a String which specifies what you want to access; e.g. `"apartment number"` but that's error prone, easy to mis-type, and what would happen if someone "set" a piece of the address we don't store? Let's define a custom data-type instead, we can enumerate all the pieces an address can have:

```
data AddressPiece
    = BuildingNumber
    | StreetName
    | ApartmentNumber
    | PostalCode
    deriving Show
```

We can use this as our `Index` type:

```
type instance Index Address   = AddressPiece
type instance IxValue Address = String
```

Now we can think about implementing `At`. `Ixed` is a superclass of `At` (everything that implements `At` MUST implement `Ixed`), so we'll need an implementation of that as well, but luckily for us there's a **default implementation** of `Ixed` in terms of `At`; so if we define `At` we get `Ixed` for free!

```
-- We need the instance declaration for Ixed,
-- but can leave the implementation blank.
-- It will be implemented automatically in terms of `At`
instance Ixed Address

instance At Address where
  at :: AddressPiece -> Lens' Address (Maybe String)
  at BuildingNumber  = buildingNumber
  at StreetName      = streetName
  at ApartmentNumber = apartmentNumber
  at PostalCode      = postalCode
```

We've already generated lenses of the correct types with TemplateHaskell, so our implementation of `at` just picks the right lens for each possible index.

In reality we should likely just use the generated lenses to interact with the fields rather than going through `Ixed` or `At`, but hopefully you've learned *how* to implement these classes for your own types for when you need it.

```
>>> let addr = Address Nothing Nothing Nothing Nothing
>>> addr
Address
    { _buildingNumber  = Nothing
    , _streetName      = Nothing
    , _apartmentNumber = Nothing
    , _postalCode      = Nothing
    }

>>> let sherlockAddr = addr & at StreetName      ?~ "Baker St."
                            & at ApartmentNumber ?~ "221B"
>>> sherlockAddr
Address
    { _buildingNumber  = Nothing
    , _streetName      = Just "Baker St."
    , _apartmentNumber = Just "221B"
    , _postalCode      = Nothing
    }

>>> sherlockAddr & ix ApartmentNumber .~ "221A"
Address
    { _buildingNumber  = Nothing
    , _streetName      = Just "Baker St."
    , _apartmentNumber = Just "221A"
```

```
    , _postalCode      = Nothing
    }

>>> sherlockAddr & sans StreetName
Address
    { _buildingNumber  = Nothing
    , _streetName      = Nothing
    , _apartmentNumber = Just "221B"
    , _postalCode      = Nothing
    }
```

Exercises – Custom Indexed Structures

Jump to answers

1. Implement both `Ixed` and `At` for a newtype wrapper around a `Map` which makes indexing case insensitive, you can specialize to `String` or `Text` keys. Write the `ix` instance manually even though it has a default implementation. It's okay to assume that user will only interact with the map via `Ixed` and `At`.

8.5 Handling missing values

The use of folds, traversals, and prisms (which we'll learn about soon) includes with it an inherent possibility of **failure**, where **failure** in this case simply means that no elements were selected. Depending on the situation we'll want to handle these failures differently.

It's easy to determine failure cases when collecting information using `preview` or `toListOf` by simply checking if the result is **Nothing** or **the empty list** respectively, but how do we determine whether an update has succeeded?

The `failover` **action** helps us handle this.

Checking whether updates succeed

```
-- Actual signature
failover :: Alternative m => LensLike ((,) Any) s t a b -> (a -> b) -> s -> m t

-- Specialized
failover :: Traversal s t a b -> (a -> b) -> s -> Maybe t
```

There's not a lot to this really, it's an **action** which basically applies the idea of "failure" from `preview` and uses it for updates as well. It takes an update function and will apply it to all focuses, if it doesn't

find any focuses at all then it will return the value of `empty` for whichever `Alternative` type is inferred as the return value, but we usually use `Nothing` from `Maybe`.

Here's a really simple example where we try to update an element which missing from a list:

```
-- There's no element at index 6, so the update fails
>>> "abcd" & failover (ix 6) toUpper :: Maybe String
Nothing

-- There's an element at index 2, so the update succeeds!
>>> "abcd" & failover (ix 2) toUpper :: Maybe String
Just "abCd"
```

Notice how I manually specify the return type as `Maybe String`. `failover` can return its result inside **any** wrapper type which implements Alternative, for instance List, Maybe, or even IO!

Although I don't recommend it, here's what happens if we change our return type to be inside IO:

```
>>> "abcd" & failover (ix 6) toUpper :: IO String
*** Exception: user error (mzero)
```

The `Alternative` instance of IO defines `empty` to be the same as its `mzero` method, which throws an Exception!

`failover` works with any traversal or prism (the chapter on prisms is coming up soon). You can also use it with lenses if you really want to, but it will be **guaranteed** to succeed so there's really no reason to do so.

Here are a few more examples.

```
>>> [] & failover _head (*10) :: Maybe [Int]
Nothing

-- We can nest traversals, the whole chain fails if it focuses no elements
>>> (M.fromList[('a', [1, 2])]
      & failover (ix 'a' . ix 3) (*10) :: Maybe (M.Map Char [Int]))
Nothing

-- It even works with filters
>>> [1, 3, 5] & failover (traversed . filtered even) (*10) :: Maybe [Int]
Nothing

>>> [1, 4, 5] & failover (traversed . filtered even) (*10) :: Maybe [Int]
Just [1,40,5]
```

Since `Maybe` has an `Alternative` instance we can use `(<|>)` to chain update attempts, the result will be the first update to succeed!

```
>>> let s = "abcdefg"
>>> failover (ix 8) toUpper s
      <|> failover (ix 6) toUpper s
      <|> failover (ix 4) toUpper s
"abcdefG"
```

This is a bit clunky though, there's a better way to do this sort of thing; let's learn about `failing`.

Fallbacks with 'failing'

`failover` allowed us to determine failure of update actions when we run them, but sometimes we want to make choices based on success or failure **within** an optics path itself!

Here's a new toy, `failing`:

```
failing :: (Conjoined p, Applicative f)
        => Traversing p f s t a b
        -> Over p f s t a b
        -> Over p f s t a b

-- Specialized signatures
failing :: Fold s t a b -> Fold s t a b -> Fold s t a b
failing :: Traversal s t a b -> Traversal s t a b -> Traversal s t a b
```

We can see from the specialized types that `failing` is a higher order combinator, it accepts two optics as arguments and returns a new optic which is **the same type as its arguments**.

It combines two optics by **trying the first, and falling back on the second**. It will run the first optic you pass it, if that focuses zero elements it will use the second optic instead. You can think of `failing` as though it **keeps** the first optic which **succeeds**. As always, an example clears this up better than my awkward words can:

```
-- Try to get something from index 10,
-- failing that, get something from index 2
>>> M.fromList [('a', 1), ('b', 2)] ^? (ix 'z' `failing` ix 'b')
Just 2

-- It works with updates as well:
>>> M.fromList [('a', 1), ('b', 2)]
      & (ix 'z' `failing` ix 'b') *~ 10
fromList [('a',1),('b',20)]
```

Typically `failing` is used infix with backticks like this. This allows you to chain as many options in a row as you need:

```
-- Get the first album available in the map in order of preference
>>> M.fromList [("Bieber" :: String, "Believe"), ("Beyoncé", "Lemonade")]
      ^? (ix "Swift" `failing` ix "Bieber" `failing` ix "Beyoncé")
Just "Believe"
```

The optics we pass to failing can be arbitrarily complex so long as the **type** of the focus is the same for each argument.

```
>>> M.fromList [('a', (1, [2, 3, 4])), ('b', (5, [6, 7, 8]))]
      ^.. (ix 'z' . _1
          `failing` ix 'a' . _2 . ix 10
          `failing` ix 'b' . _2 . traversed
          )
[6,7,8]
```

Default elements

Sometimes when a value is missing at some particular key you'd prefer to use a default value for it instead. For example, if we were counting the number of occurrences of unique words in a text it would be quite cumbersome and inefficient to first check whether we've encountered a word already, and only then add it to the map if its missing or increment it if it already exists. We would much prefer to access-and-increment all in one step. In the case of counting unique words it's semantically correct to use '0' as a default value, making this assumption simplifies and speeds up the task. Let's learn how we can do this elegantly using `non`.

```
non :: Eq a => a -> Iso' (Maybe a) a

-- For all intents and purposes we'll pretend it has this signature:
non :: Eq a => a -> Traversal' (Maybe a) a
```

`non` is technically an `Iso`, which we haven't covered yet, but you don't really need to understand `Iso`s in order to use it. For all intents and purposes you can treat it like a traversal.

We configure `non` by passing it a **default value**, it uses the default value to build a traversal which focuses the value within a `Just`, or in the case of a `Nothing` focuses the default value instead.

The simplest use is to unpack a provided `Maybe`:

```
>>> Nothing ^. non 0
0

>>> Nothing ^. non "default"
"default"

>>> Just "value" ^. non "default"
"value"
```

This mimics the behaviour of `fromMaybe` using optics.

`non` becomes much more useful when we combine it with the '`at`' combinator. Remember that '`at`' focuses a value wrapped in `Maybe`, and `Nothing` is used if the value is missing. This behaviour dovetails with `non` to substitute a default value for missing values:

```
>>> let favouriteFoods =
          M.fromList [("Garfield", "Lasagna"), ("Leslie", "Waffles")]

-- Using `non` unwraps the Maybe so we can view values directly
-- "Leslie" has a favourite food, so we can look it up:
>>> favouriteFoods ^. at "Leslie" . non "Pizza"
"Waffles"

-- If we don't know someone's favourite food, the default is "Pizza"
>>> favouriteFoods ^. at "Leo" . non "Pizza"
"Pizza"
```

This works great for viewing values, but we can use it when setting as well. When setting through `non` it allows us to set values directly without worrying about wrapping them in `Just`.

```
>>> favouriteFoods & at "Popeye" . non "Pizza" .~ "Spinach"
fromList [ ("Garfield", "Lasagna")
         , ("Leslie"  , "Waffles")
         , ("Popeye"  , "Spinach")
         ]
```

`non` has interesting behaviour when setting a key to the default value, it maps the default value back to `Nothing`, which '`at`' will be exclude from the map. The reason for this behaviour is so that `non` can pass the isomorphism laws (which we'll learn later), but in the case of inserting values into Maps it still mostly make sense. If something is the default value it's redundant to store it. If we try to view that key later we'll still receive the correct value by using the default. We can use this to build sparse maps and save some performance if one particular value in the map is very common.

Here's what happens if we set a key to the default value. To save us some typing we can make an alias for our accessor optic first.

```
-- Define an alias for our optic with the default value included
>>> let fav name = at name . non "Pizza"
>>> let favouriteFoods =
          M.fromList [("Garfield", "Lasagna"), ("Leslie", "Waffles")]

>>> let newFavourites = favouriteFoods & fav "Garfield" .~ "Pizza"

-- "Garfield" isn't stored when his favourite matches the default
>>> newFavourites
fromList [("Leslie","Waffles")]

-- We still get the correct value when retrieving Garfield's favourite
>>> newFavourites ^. fav "Garfield"
"Pizza"
```

We can modify values through `non` too, it behaves as you'd hope!

Let's keep tally of the number of hours each employee has worked in a Map. If we add hours for an employee missing from the map they should be added as though they had logged `0` hours.

```
-- Erin will be added to the map since she's missing.
>>> M.fromList [("Jim", 32), ("Dwight", 39)]
      & at "Erin" . non 0 +~ 10
M.fromList [("Dwight",39),("Erin",10),("Jim",32)]

-- Since Jim already has hours logged we simply increment them.
>>> M.fromList [("Jim", 32), ("Dwight", 39)]
      & at "Jim" . non 0 +~ 8
fromList [("Dwight",39),("Jim",40)]

-- When we pay-out an employee's hours, we set their hours to `0`
-- 'non' removes any keys with the default value from the list entirely.
>>> M.fromList [("Jim", 32), ("Dwight", 39)]
      & at "Dwight" . non 0 .~ 0
fromList [("Jim",32)]
```

Checking fold success/failure

We've seen how `non` can be used with lenses which focus values wrapped in `Maybe` like `at`, but what if we want to retrieve default values when focusing list indices or using an arbitrary fold?

In general you can use `preview` a.k.a. `(^?)` and `fromMaybe` to achieve this effect:

```
>>> import Data.Maybe (fromMaybe)
>>> fromMaybe 'z' ("abc" ^? ix 10)
'z'
```

But on the odd occasion we may want to run this sort of check in-line. We can use `non` with a new fold combinator: `pre`

```
-- Real signature
pre :: Getting (First a) s a -> IndexPreservingGetter s (Maybe a)

-- Specialized
pre :: Fold s a -> Getter s (Maybe a)
```

`pre` is a version of `preview` as a higher-order optic. You pass it a fold and it will try to get a value from it, returning `Nothing` if it can't find one. Note that it returns a Getter, so we can't **set** or **update** using `pre`.

Here's the last example side-by-side with a version which uses `pre`:

```
>>> fromMaybe 'z' ("abc" ^? ix 10)
'z'

>>> "abc" ^. pre (ix 10) . non 'z'
'z'
```

Personally I find the `fromMaybe` version easier to read, but there are cases where `pre` is necessary.

Here are a few more examples:

```
-- We use ^. rather than ^? since `pre` turns the fold into a Getter.
>>> [1, 2, 3, 4] ^. pre (traversed . filtered even)
Just 2

>>> [1, 3] ^. pre (traversed . filtered even)
Nothing
```

We can combine this with `non` and `ix` to get default values when accessing list elements:

```
>>> "abc" ^. pre (ix 20) . non 'z'
'z'
```

`pre` isn't very commonly used, however I figured I'd include it here just in case you have a use for it.

Exercises – Missing Values

Jump to answers

1. Write an optic which focuses the value at key “first” or, failing that, the value at key “second”.

```
>>> let optic = ???
>>> M.fromList [("first", False), ("second", False)]
      & optic .~ True
M.fromList [("first",True),("second",False)]

>>> M.fromList [("second", False)]
      & optic .~ True
fromList [("second",True)]
```

2. Write an optic which focuses the first element of a tuple iff it is even, and the second tuple element otherwise. Assume each slot contains an integer.

```
>>> let optic = ???
>>> (1, 1) & optic *~ 10
(1,10)

>>> (2, 2) & optic *~ 10
(20,2)
```

3. Write an optic which focuses all even numbers in a list, if none of the members are odd then focus ALL numbers in the list.

```
>>> let optic = ???
>>> [1, 2, 3, 4] ^.. optic
[2,4]

>>> [1, 3, 5] ^.. optic
[1,3,5]
```

4. Fill in the blanks

```
>>> Nothing ^. non "default"
_

>>> Nothing & _ +~ 7
Just 107

>>> M.fromList [("Perogies", True), ("Pizza", True), ("Pilsners", True)]
      ^. at "Broccoli" . _
False

>>> M.fromList [("Breath of the wild", 22000000), ("Odyssey", 9070000)]
      & _ +~ 999
M.fromList
  [ ("Breath of the wild",22000000)
  , ("Odyssey",9070000)
  , ("Wario's Woods",999)
  ]

>>> ["Math", "Science", "Geography"]
      ^. _ . non "Unscheduled"
"Unscheduled"
```

BONUS

```
-- Use 'pre' and 'non'
>>> [1, 2, 3, 4] ^.. _
[-1, 2, -1, 4]
```

9. Prisms

9.1 Introduction to Prisms

How do Prisms fit into the hierarchy?

Prisms are an interesting category of optics. In most respects they behave similarly to traversals. However, prisms can only focus **at most one** value. Since **at most one** technically fits into **zero or more**, all prisms are valid traversals. Prisms have a secret weapon which traversals don't, Prisms can be run **backwards**, taking a **focus** and embedding it into a **structure**! We'll see throughout this chapter how prisms are a natural candidate for specifying pattern-matching semantics and help to work with sum-types as well.

Here are the abilities of all the optics we've explored so far:

	Lens	**Fold**	**Traversal**	**Prism**
Get	Single	Many	Many	Zero or One
Set/Modify	Single	Nope	Many	Zero or One
Traverse	Single	Nope	Many	Zero or One
Embed	Nope	Nope	Nope	One

One could describe traversals as **finding many focuses within a structure**, conversely, prisms are more concerned with determining whether a **single value** matches some set of properties. Effectively they represent pattern-matching or predicate-checking in the world of optics.

Simple Pattern-Matching Prisms

Each prism represents a **possible pattern** that a structure could match. Sometimes this **pattern** is simple, like matching on the value's data constructor. Other times it's a bit more abstract.

Let's see the simple case first with the constructors of the `Either` type.

In case you forgot, it's defined like this:

```
data Either l r =
    Left l
  | Right r
```

By convention prisms which match on the **constructor** of a value are named by adding an underscore to the name of the constructor they match on. For the `Either` type we have `_Left` and `_Right` prisms, which are exported by the `lens` library:

```
_Left  :: Prism (Either l r) (Either l' r) l l'
_Right :: Prism (Either l r) (Either l r') r r'
```

As indicated by the types, each of these prisms will pattern match on the `Left` or `Right` side of the either type, focusing on the value inside. We can use `preview` a.k.a. `(^?)` to "run" the pattern match, returning **Nothing** if it's not a match:

```
>>> Left "message" ^? _Left
Just "message"
>>> Left "message" ^? _Right
Nothing

>>> Right 42 ^? _Right
Just 42
>>> Right 42 ^? _Left
Nothing
```

Since prisms are valid traversals we can set, update, or traverse the focused value through them:

```
>>> Left 10 & _Left +~ 5
Left 15

>>> Right "howdy" & _Right %~ reverse
Right "ydwoh"
```

We have prisms for pattern-matching on the `Maybe` type too! Here's the definition of the `Maybe` type alongside the prisms which are provided by `lens`:

```
data Maybe a =
    Nothing
  | Just a

_Nothing :: Prism' (Maybe a) ()
_Just    :: Prism  (Maybe a) (Maybe b) a b
```

`_Nothing` matches on the `Nothing` constructor, this constructor doesn't have any fields, but the Prism has to focus *something* when it succeeds so it focuses the unit value: `()`.

`_Just` matches and unpacks the `Just` constructor, focusing the value inside:

```
>>> Nothing ^? _Nothing
Just ()
>>> Nothing ^? _Just
Nothing

>>> Just "gold" ^? _Just
Just "gold"
>>> Just "gold" ^? _Nothing
Nothing

>>> Just 20 & _Just %~ (+10)
Just 30
```

Checking pattern matches with prisms

If you want a quick way to simply check whether a pattern matches, we can use the following combinators:

```
has   :: Fold s a      -> s -> Bool
isn't :: Prism s t a b -> s -> Bool
```

There's also an `is` to match `isn't`, but that's hidden away in the `Control.Lens.Extras` module to avoid naming conflicts, so usually it's more convenient to use `has`.

`has` checks whether the given **fold** yields *any* elements when run on the provided structure; while `isn't` returns whether a given prism **doesn't** match the input. Don't get tripped up by any double negatives!

```
>>> has _Right (Left "message")
False
>>> has _Right (Right 37)
True

>>> isn't _Nothing (Just 12)
True

>>> isn't _Left (Left ())
False
```

Generating prisms with makePrisms

Here's a laughably simplified representation of HTTP web requests as a Sum type:

```haskell
-- A path is a list of URL segments
type Path = [String]
type Body = String

data Request =
      Post Path Body
    | Get Path
    | Delete Path
  deriving Show

makePrisms ''Request
```

We can generate prisms for arbitrary types using TemplateHaskell just like we did with lenses. Unsurprisingly, instead of `makeLenses` we use `makePrisms`.

If we `:browse` the module we can see that some prisms were generated for us:

```haskell
>>> :browse MyPrisms
type Path = [String]
type Body = String
data Request = Post Path Body | Get Path | Delete Path
_Post :: Prism' Request (Path, Body)
_Get :: Prism' Request Path
_Delete :: Prism' Request Path
```

Each prism pattern matches on a single constructor and focuses all the fields that constructor contains. The `makePrisms` combinator is smart enough to pack up multiple fields into a tuple for us. This is why `_Post` prism focuses a tuple of `(Path, Body)`.

This is all we need to use these prisms for getting or setting as if they were traversals:

```haskell
>>> Get ["users"] ^? _Get
Just ["users"]

>>> Get ["users"] ^? _Post
Nothing

>>> Get ["users"] & _Get .~ ["posts"]
Get ["posts"]

>>> Post ["users"] "name: John" ^? _Post
Just (["users"],"name: John")

>>> Post ["users"] "name: John" & _Post . _1 <>~ ["12345"]
Post ["users","12345"] "name: John"
```

Embedding values with prisms

So far we've only been getting and setting values and everything we've demonstrated could be done just as well with a traversal, but the secret sauce lies in the fact that every prism represents a pattern-match which can be **reversed**. Looking through a prism one way matches a structure on the pattern and focuses the relevant data that's left over after the match, but we can also turn the prism around! By feeding a prism a focus we can **embed** that focus into **a structure** via the context implied by the pattern.

Even though **view**ing through a prism may fail if the pattern doesn't match, **embed**ding a value into a pattern using a prism **always** succeeds! To run a prism backwards we use the `review` action; which you can think of as short for "reverse view". It embeds the focus into the prism's pattern.

```
review :: Prism s t a b -> b -> t

-- Infix alias:
(#) :: Prism s t a b -> b -> t
```

For example, prisms which match on a constructor of some type can be reversed to **embed** fields into the constructor. The reverse of **unpacking** a specific constructor is to **pack** those fields **into** that constructor.

```
>>> review _Get ["posts"]
Get ["posts"]

-- We can use the infix alias to accomplish the same:
>>> _Get # ["posts"]
Get ["posts"]

>>> _Delete # ["posts"]
Delete ["posts"]

-- Constructors with multiple fields accept a tuple of the fields
>>> _Post # (["posts"], "My blog post")
Post ["posts"] "My blog post"
```

This works with `Left` and `Right` as well of course:

```
-- Construct a `Left` from a string
>>> review _Left "an error"
Left "an error"

>>> review _Right 42
Right 42
```

Since composing prisms results in a new prism, we can compose prisms before passing them to `review` to build a nested constructor function:

```
>>> _Just . _Left # 1337
Just (Left 1337)
```

You'll notice that the composition of constructors still reads left-to-right like all optics do.

It may seem a bit silly to use `review` when you could just use the actual constructors of the type, and in general I suggest that you use the normal constructor rather than `review` when possible, however the ability to reverse a prism can be used to implement some of the prism combinators we'll look at later on. That is, even if we never use `review` directly, it's part of the contract that allows the prism combinators to exist.

Other types of patterns

So far all the prisms we've learned have matched on specific constructors. Although **most** of the prisms you'll encounter will be used for matching on data type constructors, prisms can also encode more complex and abstract patterns.

`_Cons` is a more abstract prism which pattern matches on the first element of a list-like type. It uses a typeclass to do so, but I've also included the specialized signatures of a few structures which implement that class.

```
-- Actual type:
class Cons s t a b | s -> a, t -> b, s b -> t, t a -> s where
  _Cons :: Prism s t (a, s) (b, t)

-- Some implementations:
_Cons :: Prism  [a]        [b]        (a, [a])           (b, [b])
_Cons :: Prism  (Seq a)    (Seq b)    (a, Seq a)         (b, Seq b)
_Cons :: Prism  (Vector a) (Vector b) (a, Vector a)      (b, Vector b)
_Cons :: Prism' String                (Char, String)
_Cons :: Prism' Text                  (Char, Text)
_Cons :: Prism' ByteString            (Word8, ByteString)
```

This prism handles the common task of peeling an element off the top of a list-like type, but generalizes it so we can use the same combinator for all list-like types. Unlike regular pattern matching if a prism fails to match it won't **crash**, instead the prism simply won't focus anything.

```
>>> [1, 2, 3] ^? _Cons
Just (1, [2, 3])

>>> "Freedom!" ^? _Cons
Just ('F', "reedom!")

>>> "" ^? _Cons
Nothing

>>> ("Freedom!" :: T.Text) ^? _Cons
Just ('F', "reedom!")

>>> ("Freedom!" :: BS.ByteString) ^? _Cons
Just (70, "reedom!")

>>> "Freedom!" & _Cons . _2 %~ reverse
"F!modeer"
```

`_Cons` is a prism, so we can run it backwards using `review` (a.k.a. `#`), to **cons** an element onto the front of a list-like type. This operation will never fail!

```
>>> _Cons # ('F', "reedom")
"Freedom"

>>> _Cons # (65, "BC" :: BS.ByteString)
"ABC"
```

If all we want to do is access the head or tail safely, the `Control.Lens.Cons` module provides these traversals as `_head` and `_tail` for convenience.

```
>>> "Freedom!" & _tail %~ reverse
"F!modeer"

>>> "Hello" & _head .~ 'J'
"Jello"
```

Note that `_head` and `_tail` are only traversals and not prisms because there's no way for us to run them in reverse like we can with `_Cons`. Take some time to convince yourself why this isn't possible.

There's also an `_Empty` prism which allows us to match on the empty value of many different types. It's also implemented using a typeclass so that you can extend it with your own types:

```
class AsEmpty a where
  _Empty :: Prism' a ()
```

Since there's no interesting information in an empty container, it simply matches the `()` type; so it's usually only useful to use this with predicates like `has` or `isn't`:

```
>>> isn't _Empty []
False

>>> isn't _Empty [1, 2, 3]
True

-- The phrasing for 'has' isn't quite as clear
-- Feel free to simply define 'is = has' if that helps.
>>> has _Empty M.empty
True

>>> has _Empty (S.fromList [1, 2, 3])
False
```

So `_Cons` represents a pattern match other than constructor type we can match on using prisms.

We'll look at one last abstract prism: `_Show`

```
_Show :: (Read a, Show a) => Prism' String a
```

As implied by the constraints, this prism can `Read` or `Show` values to and from their String representations. The "pattern" we're matching on is whether the value can successfully be parsed into the result type (which will be determined by type inference). If the string fails to parse into the output type properly the prism will not match. To run it in reverse it calls "show" on the provided value to turn it back into a string.

I'd strongly recommend using an actual parsing library for any production use-cases; but it can be handy for some quick'n'dirty parsing.

```
>>> "12" ^? _Show :: Maybe Int
Just 12

-- The type we assert is important
-- If we pick a different type it changes the behaviour
>>> "12" ^? _Show :: Maybe Bool
Nothing

>>> "apple pie" ^? _Show :: Maybe Int
Nothing
```

If we're clever we can use this in the midst of a path of other optics to great effect:

```
-- Get a list of all Integers in a sentence
>>> "It's True that I ate 3 apples and 5 oranges"
      ^.. worded . _Show :: [Int]
[3 , 5]

-- Make sure you specify the correct output type.
-- '_Show'  uses the output type to decide what to try to read
-- Changing the expected type can even change the result!
>>> "It's True that I ate 3 apples and 5 oranges"
      ^.. worded . _Show :: [Bool]
[ True ]
```

We're really not using the full power of prisms in most of these examples; much of what we've done can be accomplished using traversals alone. We'll learn some more prism specific combinators soon.

Exercises – Prisms

Jump to answers

1. Which prisms will be generated from the following data declaration? Give their names and types.

```
data ContactInfo =
    Email String
  | Telephone Int
  | Address String String String

makePrisms ''ContactInfo
```

2. Fill in the blanks

```
>>> Right 35 & _ +~ 5
Right 40

>>> [Just "Mind", Just "Power", Nothing, Just "Soul", Nothing, Just "Time"]
      ^.. folded . _
["Mind","Power","Soul","Time"]

>>> [Just "Mind", Just "Power", Nothing, Just "Soul", Nothing, Just "Time"]
      & _ <>~ " Stone"
[ Just "Mind Stone"
, Just "Power Stone"
, Nothing
, Just "Soul Stone"
, Nothing
, Just "Time Stone"
]

>>> Left (Right True, "Eureka!")
      & _ %~ not
Left (Right False, "Eureka!")

>>> _Cons _ ("Do",["Re", "Mi"])
["Do", "Re", "Mi"]

>>> isn't (_Show :: _) "not an int"
True
```

3. Write an expression to get the output from the provided input.

```
input = (Just 1, Nothing, Just 3)
output = [1, 3]

---

input = ('x', "yz")
output = "xzy"

---

input = "do the hokey pokey"
output = Left (Just (Right "do the hokey pokey"))
```

9.2 Writing Custom Prisms

So far we've seen how we can generate prisms for matching constructors of data-types using `makePrisms`, but sometimes we want to match on more **abstract** patterns. We can build a prism for any **pattern** which we can **reverse**. We can imagine a prism as "refracting" your data, splitting it up and focusing only the pieces which match the pattern, dissipating the rest. Luckily the `lens` library provides us with some helpers to make constructing prisms easy and fool-proof:

```
prism  :: (b -> t) -> (s -> Either t a) -> Prism s t a b
prism' :: (b -> s) -> (s -> Maybe    a) -> Prism s s a b
```

There are two helpers. `prism` is the most general, it can generate fully polymorphic prisms! We'll often use this one when focusing data which contains type-parameters to allow them to change type. When working with simpler types the `prism'` helper is slightly easier to use.

Both functions take two arguments: an **embed**ding function and a **match**ing function.

`embed :: b -> t`
: The embedding function is where you specify how to **reverse** your pattern match by **embedding** the prism's focus *back* into the pattern. This is the function which will be called when **review**ing and it must never fail.

`match :: (s -> Either t a)` OR `match :: (s -> Maybe a)`
: The matching function determines whether the prism matches or fails. For polymorphic prisms where `<s>` is not equal to `<t>` the match function must return a member of the type which matches the *new* type `<t>`. This allows type changing traversals to work even when the prism fails to match. We'll see an example soon. For simpler prisms it's sufficient to provide `Just` the focus, or `Nothing` if the match fails.

Rebuilding `_Just` and `_Nothing`

To better understand how this all works we'll rebuild the `Maybe` prisms: `_Just` and `_Nothing` from scratch using `prism`. We'll start with `_Just`! Let's look at the type signature:

```
_Just' :: Prism (Maybe a) (Maybe b) a b
```

We can see that it allows **focusing**, and therefore **changing**, the type inside the `Maybe`. Since our input structure: `Maybe a` is not equal to our output structure: `Maybe b` we know that we need to use the polymorphic helper `prism`.

```
_Just' :: Prism (Maybe a) (Maybe b) a b
_Just' = prism embed match
  where
    match :: Maybe a -> Either (Maybe b) a
    match (Just a) = Right a
    match Nothing = Left Nothing
    embed :: b -> Maybe b
    embed b = Just b
```

The implementation is quite simple! We'll start with the `match` function.

Our prism is meant to focus on the values inside the `Just` constructor, so our `match` function should **succeed** with a `Right` if we have a `Just`, and should return a `Left` of the **new type** if we don't match. In this case it's as easy as matching on the constructor and wrapping the appropriate value into the Either type.

Take special note that the `Nothing`'s on each side of the equation have different types! I'll make it a bit clearer:

```
match (Nothing :: Maybe a) = Left (Nothing :: Maybe b)
```

Since `Nothing` doesn't have any `a`'s in it we can change the type variable freely in the `Nothing` case. This is what allows us to have a fully polymorphic prism; **even when we fail to match we can still change types**!

The **embed** function is even easier. To **embed** a `<b>` into a `<Maybe b>` we simply wrap a value in a `Just`; effectively **reversing** the `Just` pattern match.

Next we'll implement the simpler `_Nothing` prism. We've got the hang of it now so I'll drop the whole thing at once:

```
_Nothing' :: Prism  (Maybe a) (Maybe a) () ()
-- OR
_Nothing' :: Prism' (Maybe a) ()
_Nothing' = prism' embed match
  where
    match :: Maybe a -> Maybe ()
    match Nothing = Just ()
    match (Just _) = Nothing
    embed :: () -> Maybe a
    embed () = Nothing
```

Since the `_Nothing` prism doesn't focus the type variable we know it can't be a polymorphic prism. This means we can use the simpler `prism'` helper in this case.

`match` is easy, if we have a `Nothing` we return `Just` the boring value: `()`. If we fail to match, we simply indicate the failure by returning `Nothing`.

`embed` is boring too! It's trivial to 'embed' into a `Nothing`, we just return `Nothing` no matter what.

Hopefully this helps you understand how these helpers work, because we're going to build something a bit more interesting now!

Matching String Prefixes

Let's write a quick prism for matching different string prefixes! We'd like to **match** strings which start with a fixed prefix, and **ignore** ones that **don't**. First step is to determine whether we need to polymorphic or simple `prism` helper. The prism I'm thinking of has the type:

```
_Prefix :: String -> Prism' String String
```

It accepts a string which it uses to build the prism, but the resulting prism is completely monomorphic and boring, so we can use the simpler `prism'` helper. Here's how we can implement it:

```
import Control.Lens
import Data.List (stripPrefix)

_Prefix :: String -> Prism' String String
_Prefix prefix = prism' embed match
  where
    match :: String -> Maybe String
    match s = stripPrefix prefix s
    embed :: String -> String
    embed s = prefix <> s
```

I'm using `stripPrefix` from `Data.List` here:

```
stripPrefix :: Eq a => [a] -> [a] -> Maybe [a]
```

It's perfect for the job! It returns the string with the prefix removed in the case of a match, or Nothing otherwise. When we run a prism we **strip away** the part of the data that we matched; the idea is that we should be able to **match** and then **embed** and end up back where we started. If we simply returned the original string without stripping the prefix first we'd end up with a **duplicated** prefix after making the round trip! Stripping the prefix as we match also allows us to chain multiple prefix prisms in a row to match multiple prefixes one after the other.

It's not just a good idea, it's the law! We'll learn more about why this law exists in the chapter on prism laws.

Let's try out the prism and make sure it works!

```
>>> "http://phishingscam.com" ^? _Prefix "https://"
Nothing

>>> "https://totallylegit.com" ^? _Prefix "https://"
Just "totallylegit.com"

-- We can even define new prisms using our existing one.
>>> let _Secure = _Prefix "https://"
-- Only add our account number if the connection is secure!
>>> "https://mybank.com" & _Secure <>~ "?accountNumber=12345"
"https://mybank.com?accountNumber=12345"

>>> "http://fakebank.com" & _Secure <>~ "?accountNumber=12345"
"http://fakebank.com"
```

Cracking the coding interview: Prisms style!

We'll look at one more slightly abstract use-case for prisms! In an attempt to get revenge against the interviewer for asking us pointless white-board coding questions we're going to implement our solution of FizzBuzz using only prisms!

If you, dear reader, are part of the 1% of folks who haven't heard of FizzBuzz before, here's the gist of it: Our task is to iterate through the numbers 1 to 100, and for each number we must print "Fizz" if the number is divisible by 3, "Buzz" if it's divisible by 5, and "FizzBuzz" if it's divisible by 3 AND 5. If it's not divisible by 3 or 5 we'll just print the number itself.

From the looks of the problem we've only got one "pattern" we need to handle: whether a number is "divisible" by some other number. That is; whether one number is a **factor** of the other. We can imagine this as a prism which "matches" on a number!

```
_Factor :: Int -> Prism' Int Int
```

Since this is a *simple* prism, we can use the simpler `prism'` helper to build it!

Our `match` function checks whether the number is a proper factor of the other, and if so, divides it out! Our `embed` function reverses the match by running **multiplying** by the number instead, this **makes** one number a factor of the other.

```haskell
_Factor :: Int -> Prism' Int Int
_Factor n = prism' embed match
  where
    embed :: Int -> Int
    embed i = i * n
    match :: Int -> Maybe Int
    match i
      | i `mod` n == 0 = Just (i `div` n)
      | otherwise = Nothing
```

Like always we'll test it out!

```haskell
-- Is 3 a factor of 9?
>>> has (_Factor 3) 9
True

-- Is 7 NOT a factor of 9?
>>> isn't (_Factor 7) 9
True

-- We can factor 5 out of 15 to get 3;
-- 3 * 5 == 15
>>> 15 ^? _Factor 5
Just 3

-- 15 is not evenly divisible by 7
>>> 15 ^? _Factor 7
Nothing
```

Notice how we 'divide out' the number when we match. This allows us to properly chain our matches and discover more factors!

```haskell
>>> 15 ^? _Factor 3 . _Factor 5
Just 1
```

We can now implement fizzbuzz using prisms for our pattern matching! We use `has` to get a boolean from our pattern.

```
prismFizzBuzz :: Int -> String
prismFizzBuzz n
  | has (_Factor 3 . _Factor 5) n = "FizzBuzz"
  | has (_Factor 3) n = "Fizz"
  | has (_Factor 5) n = "Buzz"
  | otherwise = show n

runFizzBuzz :: IO ()
runFizzBuzz = for_ [1..20] $ \n -> putStrLn (prismFizzBuzz n)
```

```
>>> runFizzBuzz
1
2
Fizz
4
Buzz
Fizz
7
8
Fizz
Buzz
11
Fizz
13
14
FizzBuzz
16
17
Fizz
19
Buzz
```

Great work! We're a shoe-in to pass the interview so long as we haven't annoyed our interviewer too much.

Exercises – Custom Prisms

Jump to answers

1. Try to write a custom prism for matching on the tail of a list:

```
_Tail :: Prism' [a] [a]
```

Is this possible? Why or why not?

2. Implement `_Cons` for lists using `prism`:

```
_ListCons :: Prism [a] [b] (a, [a]) (b, [b])
```

BONUS

3. Implement `_Cycles` which detects whether a list consists of exactly 'n' repetitions of a pattern. It should behave as follows:

```
-- Find the subsequence which has been repeated exactly times.
>>> "dogdogdog" ^? _Cycles 3
Just "dog"

-- The input isn't exactly 3 cycles of some input (it's 4), so we don't match
>>> "dogdogdogdog" ^? _Cycles 3
Nothing

-- The input is exactly 3 cycles of "a"
>>> "aaa" ^? _Cycles 3
Just "a"

-- The input isn't a cycle at all.
>>> "xyz" ^? _Cycles 3
Nothing

-- We can review to create cycles.
>>> _Cycles 3 # "dog"
"dogdogdog"

-- We can mutate the sequence that's cycled
>>> "dogdogdog" & _Cycles 3 .~ "cats"
"catscatscats"
```

- Can you implement a version of `_Cycles` which doesn't depend on a specific iteration count? Why or why not?

9.3 Laws

Prisms have several laws, several of which we've intuited along the way. They mostly revolve around the idea that a prism is a **reversible pattern match**.

There are three laws.

Law One: Review-Preview

This law entails the notion that `review` **embeds** values into the same pattern the prism matches.

If we `review` a value through a prism, then preview it through the same prism we should end up back where we started (with an extra `Just`).

In code this law is expressed as:

```
preview p (review p value) == Just value
```

We can observe this with the prisms we've seen:

```
>>> preview _Left (review _Left "Jabberwocky") == Just "Jabberwocky"
True

>>> preview _Just (review _Just "Jabberwocky") == Just "Jabberwocky"
True
```

This is very intuitive for prisms which correspond to constructors of Sum types, but can be a little less clear for more abstract prisms.

Law Two: Prism Complement

The second law is that for a prism, the structure is completely described by the combination of the focus of the prism and the prism itself. This ends up shaking out as effectively the inverse of the previous law. This is a bit ambiguous; so here's code equivalent:

```
-- If we preview with a matching prism to get the focus 'a'
let Just a = preview myPrism s

-- Then we 'review' that focus 'a'
let s' = review myPrism a

-- We must end up back where we started
s == s'
```

We only test the law in cases where the preview succeeds.

```
>>> let s = Right 32
>>> let Just a = preview _Right s
>>> let s' = review _Right a
>>> s == s'
True
```

We've used a prism which is unlawful with respect to this law; the `_Show` prism **normalizes** the string representation when you do the round-trip. This means the result will still represent the same value with respect to `read`; but it may be formatted slightly differently. Here's a demonstration:

```
>>> let s = "[1, 2, 3]"
>>> let Just a = preview (_Show :: Prism' String [Int]) s
>>> let s' = review _Show a
>>> s == s'
False

-- Note the different spacing:
>>> s
"[1, 2, 3]"
-- _Show has normalized the value to have no spaces between values
>>> s'
"[1,2,3]"
```

`_Show` technically fails this law! It normalizes the formatting of the string according to its `Show` instance, which isn't allowed according to the law. In practice this isn't usually a big deal, it's *close enough*. The two values are equal with respect to their **semantic meaning**, so we'll give it a pass, but if you find yourself failing any of the laws for your custom prism you'll want to think carefully about whether your prism makes sense.

Law Three: Pass-through Reversion

In order to understand this law we'll first need to develop our understanding of polymorphic prisms a little better. Note that simple prisms built using `prism'` should always pass this law, since they don't perform any conversions on unmatched portions of the structure.

When working with polymorphic prisms like `_Right`:

```
_Right :: Prism (Either c a) (Either c b) a b
```

We can see that we're able to modify the type of the second type-parameter as the focus. However, prisms **also** allow us to change the type parameter in the case when we **don't** match. We can clearly witness this using the `matching` combinator:

```
matching :: Prism s t a b -> s -> Either t a
```

This combinator witnesses the idea that either we can match on the prism and therefore have access to the focus, **or** the prism doesn't match, and therefore we can change the type of our structure freely into <t> and return that.

This is clearly demonstrated using `_Just`:

```
>>> :t matching _Just
matching _Just :: Maybe a -> Either (Maybe b) a
```

This type shows that we can either access the <a> inside the Maybe (if it was a `Just`), or we can freely typecast the `(Maybe a)` value into `(Maybe b)` because the value MUST be a `Nothing`.

The same can be done with `_Right`:

```
>>> :t matching _Right
matching _Right :: Either c a -> Either (Either c b) a
```

This type states that either have a `Right` value and can retrieve the <a>; or we have a `Left` and can therefore freely cast the second type variable freely into whatever we like.

The actual law states that if the prism fails to match and we type cast the structure into a new type; that we can use the same prism to type cast it back into its original type.

In code we can express it like this:

```
let Left t = matching l s
let Left s' = matching l t
s == s'
```

Let's test out `_Just`, we'll need to annotate which types we want to type-cast into.

```
>>> let s = Nothing :: Maybe Int
>>> let Left (t :: Maybe String) = matching _Just s
>>> let Left (s' :: Maybe Int)   = matching _Just t
>>> s == s'
True

-- In this case the values are trivially equal
-- However it's still relevant that the type casting back and forth succeeded.
>>> s
Nothing
>>> s'
Nothing
```

Here's a slightly more complex example using `_Right`.

```
>>> let s = Left "Yo" :: Either String Int
>>> let Left (t :: Either String Bool) = matching _Right s
>>> let Left (s' :: Either String Int) = matching _Right t
>>> s == s'
True

>>> s
Left "Yo"
>>> s'
Left "Yo"
```

Summary

The heart of the prism laws is that a prism should behave like a reversible pattern match. Usually it's sufficient to ensure that `review myPrism` is the inverse of `preview myPrism` (ignoring the extra `Just`).

Exercises – Prism Laws

Jump to answers

1. Implement the following prism and determine whether it's lawful:

```
_Contains :: forall a. Ord a => a -> Prism' (S.Set a) (S.Set a)
```

It should match on sets which contain the provided element! Reviewing adds the element to the set. It should behave something like this:

```
>>> S.fromList [1, 2, 3] ^? _Contains 2
Just (fromList [1,3])

>>> S.fromList [1, 2, 3] ^? _Contains 10
Nothing

>>> _Contains 10 # S.fromList [1, 2, 3]
fromList [1,2,3,10]

>>> _Contains 2 # S.fromList [1, 2, 3]
fromList [1,2,3]
```

Is it lawful? Why or why not?

2. Is the following prism lawful? Write out the checks to confirm your suspicions.

```
_Singleton :: forall a. Prism' [a] a
_Singleton = prism' embed match
  where
    match :: [a] -> Maybe a
    match [a] = Just a
    match _ = Nothing
    embed :: a -> [a]
    embed a = [a]
```

3. Write your own prism which fails the first law!

9.4 Case Study: Simple Server

I'll be demoing a few more prism combinators using a simple web-server setup. It'll be a bit unconventional as far as web servers go, but should hopefully demonstrate a few prism capabilities. We'll define a Sum type to represent an HTTP request.

```
type Path = [String]
type Body = String

data Request =
      Post Path Body
    | Get Path
    | Delete Path
  deriving Show

makePrisms ''Request
```

This is by no means representative of any way in which a real web server works, but should give us an avenue to explore some concepts which you can then extrapolate into real world use-cases on your own.

All requests contain a `Path` which represents the path segments in the URL. E.g. The request `GET /posts/12345` would be passed into our server as `Get ["posts", 12345]`

`POST` requests contain an additional body parameter that `GET` and `DELETE` requests do not.

Since every possible request has a path; I'll build a quick lens for getting and setting the path no matter which type of request we're dealing with. We're going to need it soon.

```
path :: Lens' Request Path
path = lens getter setter
  where
    getter (Post p body) = p
    getter (Get p) = p
    getter (Delete p) = p
    setter (Post _ body) p = Post p body
    setter (Get _) p = Get p
    setter (Delete _) p = Delete p
```

```
>>> Get ["posts", "12345"] ^. path
["posts","12345"]

>>> Post ["posts", "12345"] "My new post" ^. path
["posts","12345"]

>>> Get ["posts", "12345"] & path .~ ["hello"]
Get ["hello"]
```

We'll say that our super-simple "server" is just a function which accepts a request and returns a response as a String (this is clearly not a book about web servers). Here's our 'simplest-thing-that-could-possibly-work' implementation:

```
-- Default server handler:
serveRequest :: Request -> String
serveRequest _ = "404 Not Found"

>>> serveRequest (Get ["hello"])
"404 Not Found"

>>> serveRequest (Delete ["user"])
"404 Not Found"

>>> serveRequest (Post ["hello"] "Is anyone there?")
"404 Not Found"
```

Not a terribly useful web-server, but you get the idea! Think we're ready for angel investors yet? No? Okay, let's start to add some behaviour!

Path prefix matching

We're going to have to respond differently depending on which path the request is heading to. We'll want a different handler for `GET /posts/12345` vs `GET /users/6789` for instance. Let's say we want to handle request routing based on **path prefixes**. All requests which start with "users" should run one handler, and "posts" should run a different one.

A **path prefix** is definitely something we can pattern match on! We can also *reverse* a **path prefix** pattern match by **prepending** a path with the prefix. These two aspects give us everything we need to build a prism!

In our particular case we can **match** on the path prefix by simply stripping off the prefix if it's there, and failing to match otherwise. We can **embed** a Request into the pattern by prefixing its path with the segment we're matching on.

Here's how it looks; we'll make use of our `path` lens here to do some of the dirty work for us:

```
_PathPrefix :: String -> Prism' Request Request
_PathPrefix prefix = prism' embed match
  where
    -- Add the prefix to the path
    embed :: Request -> Request
    embed req = req & path %~ (prefix :)
    -- Check if the prefix matches the path
    match :: Request -> Maybe Request
    match req
      | has (path . _head . only prefix) req = Just (req & path %~ drop 1)
    match _ = Nothing
```

We should of course test it out before proceeding. When viewing through the prism we expect it to strip the provided path segment from the request if there's a match, otherwise it should fail.

```
>>> (Get ["users", "all"]) ^? _PathPrefix "users"
Just (Get ["all"])

>>> (Delete ["posts", "12345"]) ^? _PathPrefix "posts"
Just (Delete ["12345"])

-- Fails to match when prefixes differ
>>> (Get ["other"]) ^? _PathPrefix "users"
Nothing
```

We should also be able to run the prism backwards with `review`. Reviewing should **embed** the request into the pattern by prefixing the path with the provided segment.

```
-- Can we run it backwards?
>>> _PathPrefix "users" # Get ["all"]
Get ["users","all"]

>>> _PathPrefix "posts" # Delete ["12345"]
Delete ["posts","12345"]
```

Great, looks like it's working properly both forwards and back! So we've defined a pattern for path prefixes, how can we actually use this pattern to customize our web-server?

> We could have implemented `_PathPrefix` much easier using the existing `prefix` prism from the `lens` library, but I wanted to show how it works behind the scenes. In future you can just use `prefixed` from `Data.List.Lens`:
>
> ```
> prefixed :: Eq a => [a] -> Prism' [a] [a]
> ```

Altering sub-sets of functions

We can now pattern match on specific path prefixes, but how do we use this pattern to specify a **behaviour** for our server? We need to somehow pattern match on the **argument** to our `serveRequest` function and change the behaviour of the handler as a result. We're going to do this using an eccentric old prism combinator called `outside`:

```
outside :: Prism s t a b -> Lens (t -> r) (s -> r) (b -> r) (a -> r)
```

I'll admit I've never seen this combinator used in a real codebase, and it may not be the most **practical or readable** way to accomplish this task, but it demonstrates the abilities of a prism well!

This higher order combinator *lifts* a prism so that it matches on the **argument** of a function and returns a special **lens**. This lens focuses on a **portion** of the provided function, specifically the part of the function which runs when the argument matches the prism.

No worries if you're feeling a bit lost, it's weird to think of a lens which focuses a *function* within a *function*! It's especially weird to think of partitioning functions based on properties of their arguments!

I'll say it again in a slightly different way: when applied to a **prism**, the `outside` combinator returns a **lens** which **focuses** the *subset* of a function which runs when the argument matches the **prism**, leaving the rest of the function untouched.

Here's a small demo where we use `outside` to fix up the unsafe version of `tail` from the Prelude.

The original tail behaves like this:

```
>>> tail [1, 2, 3]
[2,3]

>>> tail []
*** Exception: Prelude.tail: empty list
```

Ack! It completely crashes on the empty list! We'll patch it with a quick fix so that instead of crashing it returns an empty list instead. This means we want to **modify** the existing `tail` function, but we only want to change a small portion of its behaviour. Specifically we want to change the behaviour of `tail` when the argument is the **empty list**! We want to **focus** on the part of the function which runs when the input is the empty list.

We already know the `_Empty` prism which matches on any sort of empty container, we can use `outside` to convert it into a **lens** on **functions**:

```
safeTail :: [a] -> [a]
safeTail = tail & outside _Empty .~ const []
```

We pass in the original `tail` function as the value we're modifying, then we focus the portion of the function which deals with the **empty list** by using `outside _Empty`. We replace that portion of the function with one that always returns `[]`.

And just like that we've fixed our bad `tail` function:

```
>>> safeTail [1, 2, 3]
[2,3]
>>> safeTail []
[]
```

In this particular context the types of `_Empty` and `(outside _Empty)` take on the forms:

```
_Empty :: Prism' [a] ()

outside _Empty :: Lens' ([a] -> [a]) (() -> [a])
```

Notice how `outside` runs the `_Empty` prism over the argument position, focusing `[a]` into `()` (which is the focus of `_Empty`) in the resulting function? This isn't terribly helpful for the `_Empty` prism, but it will be more useful as we move back towards our web-server example.

Let's get back to our server! Our current task is to set a custom handler for the `/users` portion of the app. Let's write the handler we want to run on requests with a "/users" prefix:

```
userHandler :: Request -> String
userHandler req =
  "User handler! Remaining path: " <> intercalate "/" (req ^. path)
```

For easy debugging we'll return "User Handler!" along with the remaining path segments.

To set this as the handler of the users section of our site we can take our base `serveRequest` handler, focus the **subset** of that handler which has a path-prefix of "users", and set it to our new `usersHandler` instead:

```
server :: Request -> String
server = serveRequest
           & outside (_PathPrefix "users") .~ userHandler
```

Here's how it behaves:

```
>>> server (Get ["users", "12345"])
"User handler! Remaining path: 12345"

>>> server (Delete ["users", "12345", "profile"])
"User handler! Remaining path: 12345/profile"

-- It should run the default handler for non user paths.
>>> server (Post ["admin"] "DROP TABLE users")
"404 Not Found"
```

See how the `_PathPrefix` prism not only **matches** on the argument to determine whether to run the `userHandler`, but it also **strips** the prefix before passing the request to the `userHandler`. This is because `outside` runs the prism on the argument before running the new function. The `userHandler` already knows it'll be run under the `users` prefix, so it's handy that the prism strips away this extra information.

We've successfully grafted our user handler into our server without affecting other parts of the app! We can easily add other handlers for different subsections in a similar fashion. Let's add a posts handler too!

```
server :: Request -> String
server = serveRequest
           & outside (_PathPrefix "users") .~ userHandler
           & outside (_PathPrefix "posts") .~ const "Posts Handler!"


>>> server (Post ["posts"] "My new post")
"Posts Handler!"
```

Since the prism strips off the prefix it matches on and focuses the altered Request, we can compose multiple `_PathPrefix` prisms to match on multiple sequential path segments:

```
server :: Handler
server = serveRequest
           & outside (_PathPrefix "users") .~ userHandler
           & outside (_PathPrefix "posts") .~ const (return "Posts Handler!")
           & outside (_PathPrefix "posts" . _PathPrefix "index")
              .~ const (return "Post Index")
```

It's important to think about your ordering. Since each line with `&` is **overwriting** a subsection of the handler function the more specific handlers should come AFTER the more general ones or you'll overwrite handlers by accident.

```
>>> server (Get ["posts"])
"Posts Handler!"

>>> server (Get ["posts", "index"])
"Post Index"
```

We can even split out everything under the "posts" prefix into its own definition and set it as a sub-handler for the "posts" prefix:

```
server :: Request -> String
server = serveRequest
           & outside (_PathPrefix "users") .~ userHandler
           & outside (_PathPrefix "posts") .~ postsHandler

postsHandler :: Request -> String
postsHandler =
    const "Posts Handler!"
      & outside (_PathPrefix "index") .~ const "Post Index"
```

This shows how Prisms allow us to do composable pattern matching in a relatively clean way! You probably won't write your next web service this way, but this principle can be leveraged inside libraries and applications to provide nice prism-based DSLs or combinators for specifying matching logic.

Matching on HTTP Verb

The next logical step is to also allow different handlers based on which HTTP Verb. We'll undoubtedly want to handle `POST` requests differently than `GET` or `DELETE`! For this we can turn to our automatically derived prisms for the `Request` type. The `makePrisms` call we added underneath our original data declaration provides the following helpful prisms:

```
_Post :: Prism' Request (Path, Body)
_Get :: Prism' Request Path
_Delete :: Prism' Request Path
```

Each prism focuses the contents of its data constructor. We can actually compose these prisms directly alongside the `_PathPrefix` prisms, but since these focus something other than a `Request` they'll need to be the LAST prism in the chain.

Here's the type of some of our resulting lenses:

```haskell
-- No verb specified, must handle any request type!
outside (_PathPrefix "posts")
  :: Lens (Request -> String) (Request -> String)

-- Match on only Posts, so we pass the POST Body
outside (_PathPrefix "posts" . _Post)
  :: Lens (Request -> String) ((Path, Body) -> String)

-- Match on only Gets, so we don't pass a Body
outside (_PathPrefix "posts" . _Get)
  :: Lens (Request -> String) (Path -> String)
```

Notice how matching on `_Post` changes the type of the required handler to pass it the `Body` whereas the other verbs deal with only the `Path`? This adds additional type-safety to our handlers, pretty cool!

We can now be even more specific in our handlers, and can access specific fields of our request type (like the body of a `POST`) in a perfectly type-safe way! We'll edit our post handler to react to different HTTP Verbs:

```haskell
postsHandler :: Request -> String
postsHandler = const "404 Not Found"
    & outside _Post
        .~ (\(path', body) -> "Created post with body: " <> body)
    & outside _Get
        .~ (\path' -> "Fetching post at path: " <> intercalate "/" path')
    & outside _Delete
        .~ (\path' -> "Deleting post at path: " <> intercalate "/" path')


>>> server (Get ["posts", "12345"])
"Fetching post at path: 12345"

>>> server (Post ["posts", "12345"] "My new post")
"Created post with body: My new post"

>>> server (Delete ["posts", "12345"])
"Deleting post at path: 12345"
```

This handler we've written is equivalent to the following code using classic Haskell pattern matching:

```haskell
postsHandler' :: Request -> String
postsHandler' (Post path' body) =
  "Created post with body: " <> body
postsHandler' (Get path') =
  "Fetching post at path: " <> intercalate "/" path'
postsHandler' (Delete path') =
  "Deleting post at path: " <> intercalate "/" path'
```

This standard code is much more readable, easier to understand, doesn't need a "default case", and will automatically be checked for completeness if you've told GHC to check for incomplete pattern matches. So when should you use prisms?

1. Prisms are composable: doing multiple pattern matches in standard Haskell requires nesting function calls or case expressions
2. Prisms interoperate with the rest of optics, providing a lot of flexibility and expressiveness. You can perform pattern matching inside of an optics chain!

It's up to you as the programmer to investigate your use-case and decide when each use-case is appropriate. In general, prisms are nice when you're not concerned with whether you've covered every possible case.

10. Isos

10.1 Introduction to Isos

We're nearing the end now! This is the last major category of optics we'll be looking at in the book, congratulations for making it this far!

Before we go further I'll explain the name, **iso** is short for **isomorphism**, which is a term borrowed from category theory. Without getting too deep into category theory I'll just say that an **isomorphism is a completely reversible transformation between two types or formats**. The important part of that sentence is "**completely reversible**". By **completely reversible** I mean that there is *zero* information lost by converting between the two forms of data. This means that at their essence isomorphisms allow you to view the same data in different ways.

How do Isos fit into the hierarchy?

Because isos are **completely reversible** they're the **strongest** of all the optics we've seen. By **strongest** I actually mean **most constrained**, every iso **MUST** succeed for all inputs, and **MUST** be completely reversible, these strong constraints let us do a lot with them!

Because of their strength, isos are a valid substitution for any other type of optic we've learned. Here's a chart to show what they can do:

	Lens	**Fold**	**Traversal**	**Prism**	**Iso**
Get	Single	Many	Many	Zero or One	Single
Set/Modify	Single	Nope	Many	Zero or One	Single
Traverse	Single	Nope	Many	Zero or One	Single
Embed	Nope	Nope	Nope	Yup	Yup

There and back again

Before we dive into understanding exactly how isos work, we'll look at how isomorphisms exist in Haskell outside of optics. Generally an isomorphism is just a pair of functions where each function is the inverse of the other! Let's see some examples.

One isomorphism that you probably use often as a Haskell programmer is the transformation between `String` and `Text`. We can convert **back and forth** between these two formats without **losing any data**! Our two functions, which we'll name `to` and `from`, are inverses of one another:

```
to :: String -> T.Text
to = T.pack

from :: T.Text -> String
from = T.unpack
```

When working with these functions we have the expectation that converting from `String` to `Text` and then back to a `String` should return to us the same string we started with. We'd be a bit miffed if converting back and forth changed the formatting, or rearranged our text! The same expectation applies if we go the other direction: from `Text` to `String` and back to `Text`. We really want these transformations to be **completely reversible** and we can express this expectation like so:

```
T.pack . T.unpack = id
T.unpack . T.pack = id
```

Or more generally for all isomorphisms:

```
to . from = id
from . to = id
```

10.2 Building Isos

Now that we roughly understand what an isomorphism is, we can see how to translate the concept into optics. An iso is an optic which views data after running its transformation on it. We can view, modify or traverse the data in it's altered form! If we modify the data in its altered form, the iso will convert the result **back** through the iso into the original form. E.g. we can use an iso to edit a `String` as though it were a `Text`, and end up with a `String` again after the modification.

We'll understand this better after we build one by hand. Building a custom iso is pretty trivial in comparison to building the other optics. An iso is fully described by its `to` and `from` functions, so that's all we need to provide to create one:

```
iso :: (s -> a) -> (b -> t) -> Iso s t a b
```

We'll start off with some simple isos, where `a ~ b` and `s ~ t` (~ means the types are *equal*). I.e. isos of the form `Iso' s a`.

First we'll build an iso for converting between `String` and `Text`.

```
packed :: Iso' String T.Text
packed = iso to' from'
  where
    to' :: String -> T.Text
    to' = T.pack
    from' :: T.Text -> String
    from' = T.unpack
```

Writing the `where` clauses is perhaps overly verbose, but I find it helps to clarify the behaviour of the iso, distinguishing which case is `to` and which is `from`. I add ticks to `to'` and `from'` because there are already combinators named `to` and `from` exported by `Control.Lens`.

Now we've got our iso between `String` and `Text`! What can we do with it? For starters we can use it like a lens!

```
-- I've added several type annotations for clarity
>>> ("Ay, caramba!" :: String) ^. packed
"Ay, caramba!" :: T.Text
```

Using `packed` views Strings as though they were `Text`. I already mentioned we can use isos as prisms, so that means we can review through them too!

```
>>> packed # ("Sufferin' Succotash" :: T.Text)
"Sufferin' Succotash" :: String
```

`review`'ing an iso runs its inverse (a.k.a. the `from'` function we provided), so we go from `Text` to `String` when `review`ing `packed`.

We typically don't `review` isos though, they have a different way of running things backwards! Let's learn about `from`.

10.3 Flipping isos with `from`

Isos add a new higher-order combinator called `from`. Isos are completely reversible, so `from` exploits this fact to turn an iso backwards:

```
from :: Iso s t a b -> Iso b a t s

-- Simpler form:
from :: Iso' s a -> Iso' a s
```

Unlike `review` which returns a **function** (and thus can't be composed in an optics path), `from` returns a whole new `Iso'` which has just as much power as the original.

We can use `from` to flip existing isos!

```
>>> ("Good grief" :: String) ^. packed
"Good grief" :: T.Text

>>> ("Good grief" :: T.Text) ^. from packed
"Good grief" :: String
```

We can even use `from` and `review` together, but that'd just reverse the iso **twice**, ending up back where we started.

If we want to improve clarity we can name these new isos.

```
unpacked :: Iso' T.Text String
unpacked = from packed
```

> P.S. Both `packed` and `unpacked` are available in `Data.Text.Lens` from the `lens` library.

10.4 Modification under isomorphism

We've seen that isos are valid lenses/traversals/prisms, but what does it mean to 'modify' something through an iso? In simple terms we convert the data through the iso, run the modification, then convert it back.

This ends up coming in handy pretty often. For instance, what if we're working with strings, but we really want to use the awesome `replace` function from `Text`?

We can **focus** the `Text` representation of a `String`, run the replacement, then convert back. This does incur the performance penalty of converting back and forth, but it's probably worth the performance hit in this case to avoid implementing `replace` over `Strings` ourselves.

```
>>> let str = "Idol on a pedestal" :: String
>>> str & packed %~ T.replace "Idol" "Sand"
"Sand on a pedestal" :: String

-- Using `over` in place of `%~` also reads nicely
>>> over packed (T.replace "Idol" "Sand") str
"Sand on a pedestal" :: String
```

We can of course compose isos with other optics! For instance if we have a `Text` and want to traverse every character, we can convert to a `String`, then `traverse` over each `Char`!

```
>>> import Data.Char (toUpper)
>>> let txt = "Lorem ipsum" :: T.Text
-- We could just use `T.toUpper` instead, but this demonstrates the point:
>>> txt & from packed . traversed %~ toUpper
"LOREM IPSUM"
```

I used `from packed` to go from `Text` to `String`, but `unpacked` is equivalent if you prefer. It turns out that the implementation of the `each` traversal for `Text` is exactly `unpacked . traversed`!

10.5 Varieties of isomorphisms

Many isomorphisms are 'trivial' but still useful, things like converting between encodings, text formats, strict <-> lazy representations, or even conversions between data structures with different performance characteristics (e.g. Linked List <-> Vector).

We can however imagine more interesting isomorphisms, all the previous examples have moved between two VERY similar types, but what if we have an isomorphism from a type to itself? We're allowed to re-arrange data in a structure however we want so long as we know how to 'undo' it with an inverse function. There are several interesting isomorphisms that don't even change the type of the data!

As a first example consider **reversing** a list. Each list contains the same data, and we know how to undo the transformation.

```
reverse :: [a] -> [a]
```

Reverse is its own inverse, so a 'reversing' isomorphism looks like this:

```
reversed :: Iso' [a] [a]
reversed = iso reverse reverse
```

There's even a helper for building isomorphisms where a function is its own inverse. It's got a fun and fancy name: `involuted`.

```
involuted :: (a -> a) -> Iso' a a
involuted f = iso f f
```

Here's an equivalent definition using involuted (which incidentally is almost the same length).

```
reversed :: Iso' [a] [a]
reversed = involuted reverse
```

We can quickly check to see that the round-trip property holds:

```
>>> reverse . reverse $ [1, 2, 3]
[1,2,3]
```

We can use it to operate on the reversed version of the list:

```
>>> [1, 2, 3] & reversed %~ drop 1
[1,2]

>>> [1, 2, 3] & reversed %~ take 1
[3]
```

We gain a lot of power by combining isos with all the other combinators we've learned.

```
>>> [1, 2, 3, 4] ^.. reversed . takingWhile (> 2) traversed
[4,3]

-- We can reverse more than once! But you probably shouldn't...
>>> "Blue suede shoes" & reversed . taking 1 worded . reversed .~ "gloves"
"Blue suede gloves"
```

There are many isos which simply re-arrange things for convenience; for instance here's one which 'swaps' the values in a tuple:

```
swapped :: Iso (s, s') (t, t') (a, a') (b, b')

-- Simpler, but more restrictive:
swapped :: Iso' (a, b) (b, a)
```

`swapped` lets us view a tuple as though the slots were swapped around.

```
>>> ("Fall","Pride") ^. swapped
("Pride","Fall")
```

The real type of `swapped` is actually even *more* general; it's backed by a `Swapped` typeclass and works on a lot of different Bifunctors:

```
swapped :: (Bifunctor p, Swapped p) => Iso (p a b) (p c d) (p b a) (p d c)
```

So one other useful way to use this is to swap the tags in an `Either`!

```
>>> Right "Field" ^. swapped
Left "Field"
```

There's not much to say about that; sometimes you just need to rearrange things a bit!

We'll go rapid fire through a few other simple isomorphisms, they're all relatively easy to understand.

We can flip function arguments, flip is its own inverse:

```
flipped :: Iso' (a -> b -> c) (b -> a -> c)
```

```
>>> let (++?) = (++) ^. flipped
>>> "A" ++? "B"
"BA"
```

We can curry then uncurry function arguments:

```
curried   :: Iso' ((a, b) -> c)  (a -> b -> c)
uncurried :: Iso' (a -> b -> c)  ((a, b) -> c)
```

```
>>> let addTuple = (+) ^. uncurried
>>> addTuple (1, 2)
3
```

We can perform all manners of reversible numeric operations:

```
>>> import Numeric.Lens
>>> 10 ^. negated
-10

>>> over negated (+10) 30
20

>>> 100 ^. adding 50
150

-- Be careful of division by 0 for 'dividing' and 'multiplying'
>>> 100.0 ^. dividing 10
10.0
```

Composing isos

Isos compose just like any other optic! They compose both the 'forwards' AND 'reversed' transformations.

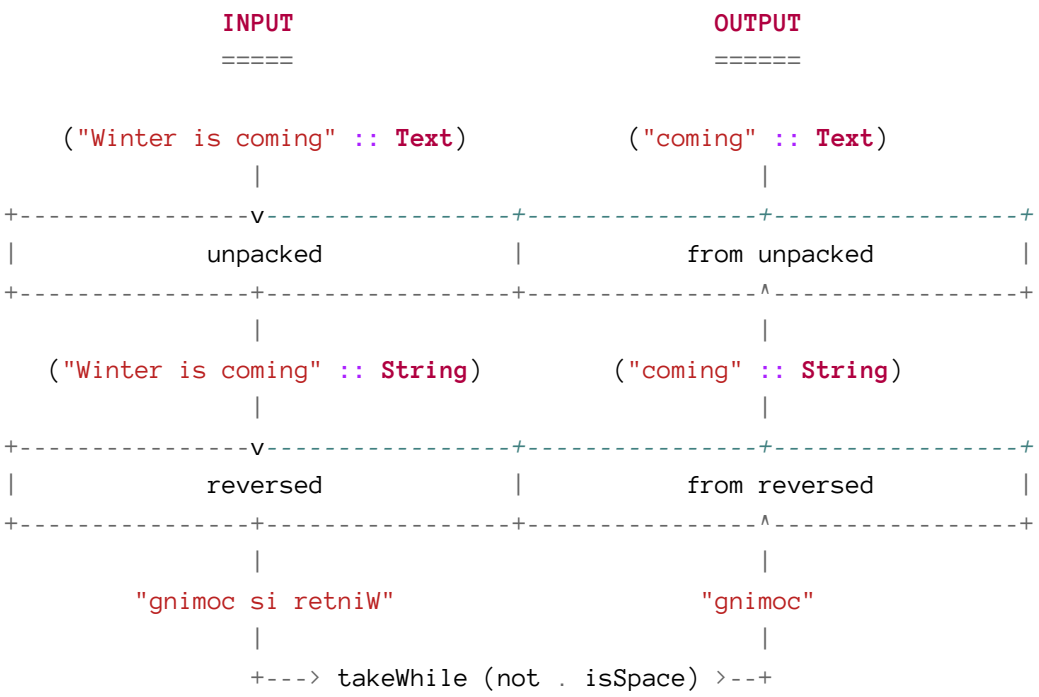
For example we can transform `Text -> String` and then reverse the String!

```
>>> let txt = "Winter is coming" :: T.Text
>>> txt ^. unpacked . reversed
"gnimoc si retniW"
```

When we modify values through an iso it gets a bit more confusing:

```
>>> import Data.Char (isSpace)
>>> let txt = "Winter is coming" :: T.Text
>>> txt & unpacked . reversed %~ takeWhile (not . isSpace)
"coming" :: T.Text
```

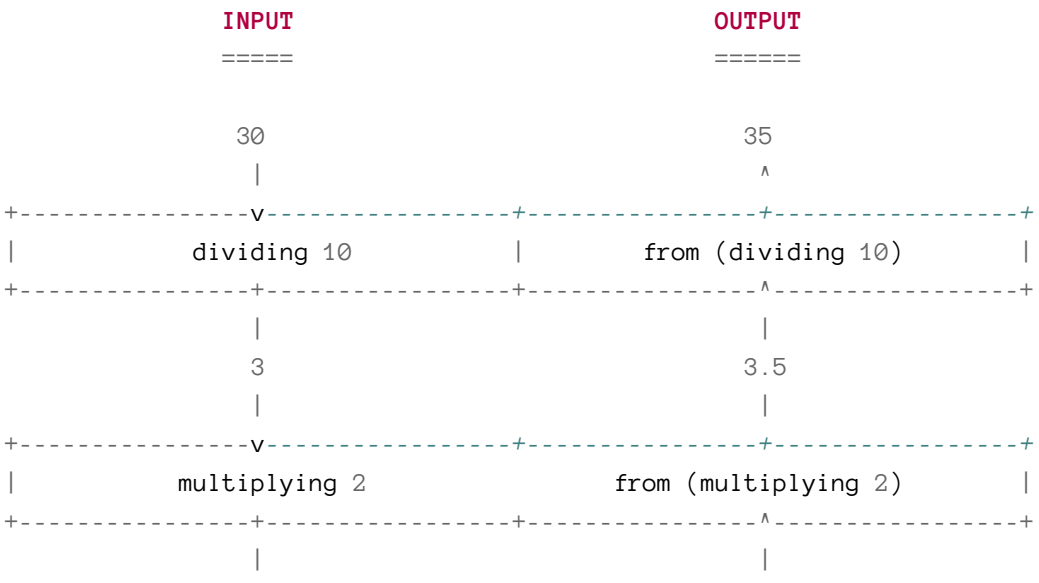
Let's walk through this step by step. We run the transformations forwards on the left; then modify the focus at the bottom, then reverse the transformations as we come back up the right hand side.



Hopefully that helps to clear up what's happening. Let's break down one more example:

```
>>> import Numeric.Lens
>>> 30 & dividing 10 . multiplying 2 +~ 1
35
```

The inverse of division is multiplication, and the inverse of multiplication is division.



Exercises – Intro to Isos

Jump to answers

1. For each of the following tasks, choose whether it's best suited to a **Lens**, **Traversal**, **Prism**, or **Iso**:

- Focus a Celsius temperature in Fahrenheit
- Focus the last element of a list
- View a JSON object as its corresponding Haskell Record
- Rotate the elements of a three-tuple one to the right
- Focus on the 'bits' of an Int as Bools.
- Focusing an `IntSet` from a `Set Int`

2. Fill in the blank

```
>>> ("Beauty", "Age") ^. _
("Age", "Beauty")

>>> 50 ^. _ (adding 10)
40

>>> 0 & multiplying _ +~ 12
3.0

>>> 0 & adding 10 . multiplying 2 .~ _
2

>>> [1, 2, 3] & reversed %~ _
[1, 2]

>>> (view _ (++)) [1, 2] [3, 4]
[3,4,1,2]

>>> [1, 2, 3] _ reversed
[3, 2, 1]

-- BONUS: Hard ones ahead!
```

```
>>> import Data.List (transpose)
-- Note: transpose flips the rows and columns of a nested list:
>>> transpose [[1, 2, 3], [10, 20, 30]]
[[1,10],[2,20],[3,30]]

>>> [[1, 2, 3], [10, 20, 30]] & involuted transpose %~ _
[[2,3],[20,30]]

-- Extra hard: use `switchCase` somehow to make this statement work:
>>> import Data.Char (isUpper, toUpper, toLower)
>>> let switchCase c = if isUpper c then toLower c else toUpper c
>>> (32, "Hi") & _2 . _  .~ ("hELLO" :: String)
(32,"Hello")
```

3. You can convert from Celsius to Fahrenheit using the following formula:

```
celsiusToF :: Double -> Double
celsiusToF c = (c * (9/5)) + 32
```

Implement the following iso

```
fahrenheit :: Iso' Double Double
```

10.6 Projecting Isos

The strong guarantees which isos provide allow us to easily lift them into other structures. Since they're reversible and guaranteed to succeed, we can sneakily lift an iso over the contents of any Functor by exploiting `fmap`!

```
mapping' :: Functor f => Iso' s a -> Iso' (f s) (f a)
mapping' i = iso (fmap (view i)) (fmap (review i))
```

A more general version of this is provided in `lens`:

```
mapping :: (Functor f, Functor g)
        => Iso s t a b -> Iso (f s) (g t) (f a) (g b)
```

This let's us easily transform nested portions of structures in our optics path as we go along!

Let's say we have the following function which converts a list of strings into a YAML list:

```
toYamlList :: [String] -> String
toYamlList xs = "- " <> intercalate "\n- " xs

>>> putStrLn $ toYamlList ["Milk", "Eggs","Flour"]
- Milk
- Eggs
- Flour
```

But in our app we've got a `[Text]` instead of a `[String]`! We already have the `unpacked` iso from `Data.Text.Lens` to convert between `Text` and `String`, and we can use `mapping` to lift it into an iso from `[Text]` to `[String]`!

```
>>> let shoppingList = ["Milk", "Eggs","Flour"] :: [T.Text]
>>> let strShoppingList = shoppingList ^. mapping unpacked :: [String]
>>> putStrLn $ toYamlList strShoppingList
- Milk
- Eggs
- Flour

-- Or we can do it all in one go!
>>> putStrLn $ shoppingList ^. mapping unpacked . to toYamlList
- Milk
- Eggs
- Flour

-- We can even use `traverseOf_`!
>>> traverseOf_  (mapping unpacked . to toYamlList)  putStrLn $ shoppingList
- Milk
- Eggs
- Flour
```

This helps for one-off cases, but what if we to convert the function itself from `[String] -> String` into `[Text] -> Text`? Remember when I said we can lift isos through functors? That applies to **contravariant functors**, **bifunctors** and even **profunctors** too!

```
contramapping :: Contravariant f
              => Iso s t a b -> Iso (f a) (f b) (f s) (f t)

bimapping :: (Bifunctor f, Bifunctor g)
          => Iso s t a b -> Iso s' t' a' b'
          -> Iso (f s s') (g t t') (f a a') (g b b')

dimapping :: (Profunctor p, Profunctor q)
          => Iso s t a b -> Iso s' t' a' b'
          -> Iso (p a s') (q b t') (p s a') (q t b')
```

Here are some simpler (but more restrictive) signatures for these:

```
contramapping :: (Contravariant f)
              => Iso' s a -> Iso (f a) (f s)

bimapping :: (Bifunctor f)
          => Iso' s a -> Iso' s' a'
          -> Iso' (f s s') (f a a')

dimapping :: (Profunctor p)
          => Iso' s a -> Iso' s' a'
          -> Iso' (p a s') (p s a')
```

A function is an instance of profunctor, meaning we can contramap over its input and map over its output all in one go using `dimapping`!

```
textToYamlList :: [T.Text] -> T.Text
textToYamlList = toYamlList ^. dimapping (mapping unpacked) packed
```

That's a lot of stuff packed into a single line! We **view** the `toYamlList` function through an iso; this iso is built up from parts! The first chunk: `(mapping unpacked)` is an `(Iso' [T.Text] [String])` which will be **contramapped** onto the function's argument, `dimapping` runs the iso backwards when contramapping, which is why it might seem like it's the wrong way round at first.

The second argument: `(packed)` is much simpler, signifying we want to `pack` the output of the function before returning it. The arguments to `dimapping` read best when you imagine the information flowing through each iso from left to right; we take our argument, **map** and **unpack**, then run our function. Lastly we **pack** the functions result.

Now I'm definitely not saying that the given code is any better or more readable than the simple version:

```
textToYamlList' :: [T.Text] -> T.Text
textToYamlList' = T.pack . toYamlList . fmap T.unpack
```

Quite the opposite in fact; I'd strongly prefer the latter version in any codebase I'm working with, but isos are powerful and compact building blocks we can combine them using these tools and export them as combinators for others to use.

Exercises – Projected Isos

Jump to answers

1. Fill in the blank

```
>>> ("Beauty", "Age") ^. mapping reversed . _
("egA","Beauty")

>>> [True, False, True] ^. mapping (_ not)
[False,True,False]

>>> [True, False, True] & _ %~ filter id
[False]

>>> (show ^. _) 1234
"4321"
```

2. Using the `enum` iso provided by `lens` (type below):

```
enum :: Enum a => Iso' Int a
```

Implement the following `intNot` function, use `dimapping` in your implementation.

```
intNot :: Int -> Int

>>> intNot 0
1
>>> intNot 1
0
>>> intNot 2
*** Exception: Prelude.Enum.Bool.toEnum: bad argument
```

Can you simplify your implementation of this function somehow?

10.7 Isos and newtypes

Coercing with isos

In Haskell we often need to add `newtype` wrappers around existing types, either for added type-safety or maybe to define some custom typeclass instances. The interesting thing about **newtypes** is that we **know** the newtype is exactly equivalent to the underlying representation, that's a requirement of defining a newtype! This means we have a trivial isomorphism between any type and any newtypes over that type.

In fact, `GHC` even implements the conversion function for us:

```
>>> import Data.Coerce (coerce)
>>> :t coerce
coerce :: Coercible a b => a -> b
```

`Coercible` is a special constraint, it's implemented for us by GHC for any newtype, we can use it even though it won't show up in instance lists.

Let's say we want to add some additional type-safety in our app to make sure we don't mix all of our Strings up. We'll create newtype wrappers over `String` so we can keep our **email addresses** separate from **user ids**.

```
newtype Email = Email String
    deriving (Show)

newtype UserID = UserID String
    deriving (Show)

-- Coercible isn't in the list of instances, but it exists!
>>> :info Email
newtype Email = Email String
instance Show Email
```

Each of these types are **representationally** equal to `Strings`; the only difference is the type!

We can use `coerce` to convert between them.

```
-- We need to specify which type to coerce into
>>> coerce ("joe@example.com" :: String) :: Email
Email "joe@example.com"

-- If two types are representationally equal
-- we can even skip the middle-man and go directly between two newtypes:
>>> coerce (Email "joe@example.com") :: UserID
UserID "joe@example.com"
```

This is pretty cool! It's easy to see how `coerce` is its own inverse. `lens` provides a handy iso for converting between newtypes called `coerced`:

```
coerced :: (Coercible s a, Coercible t b) => Iso s t a b


>>> over coerced
      (reverse :: String -> String)
      (Email "joe@example.com") :: Email
Email "moc.elpmaxe@eoj"
```

Unfortunately, because `coerced` can go between **many** different types, type inference really struggles here, and we'll almost always need to add a lot of extra annotations.

If we don't have enough annotations we'll run into strange errors. Here's what it might look like:

```
>>> import Data.Char (toUpper)
>>> Email "joe@example.com" & coerced . traversed %~ toUpper

<interactive>:1:1: error:
    • Couldn't match representation of type ‘String’
                               with that of ‘f0 Char’
    • In the ambiguity check for the inferred type for ‘it’
      To defer the ambiguity check to use sites, enable AllowAmbiguousTypes
      When checking the inferred type
        it :: forall (f :: * -> *) b.
              (Coercible String (f Char), Traversable f, Coercible b (f Char))
              => b
```

We can of course add extra type annotations, but it's not always clear where to put them! Sometimes (like this example) it can get pretty awkward:

```
>>> Email "joe@example.com"
      & (coerced :: Iso' Email String) . traversed %~ toUpper
Email "JOE@EXAMPLE.COM"
```

We'd prefer to avoid that if possible! If we forget the annotation, or even put it in the wrong spot, then our goose is pretty much cooked!

This is pretty clearly not ideal, there are a few ways we can do a little better, one simple way is to use `coerced` to define your own helper:

```
email :: Iso' Email String
email = coerced

>>> Email "joe@example.com" & email . traversed %~ toUpper
Email "JOE@EXAMPLE.COM"
```

By adding an explicit type signature when defining `email` we no longer need to specify it at the call site. This cleans it up pretty nicely.

As it turns out, `makeLenses` is actually smart enough to derive exactly this iso if we define our newtype using Record syntax:

```
newtype Email = Email {_email :: String}
    deriving (Show)
makeLenses ''Email
```

Now `makeLenses` will generate an `email` iso which behaves just like the one we manually defined!

```
>>> :browse Isos.Newtypes
newtype Email = Email {_email :: String}
email :: Iso' Email String
newtype UserID = UserID String
```

Newtype wrapper isos

I personally recommend using the iso derived by `makeLenses`; it's easy to work with, and is pretty easy to read too, but there are a few other options provided by `lens` that we should still look into. They can be especially handy when working with newtypes provided by `base`.

`lens` provides the following isos which are a more restricted form of `coerced`:

```
_Wrapped'   :: Wrapped s => Iso' s (Unwrapped s)
_Unwrapped' :: Wrapped s => Iso' (Unwrapped s) s
```

These isos are essentially restricted forms of `coerced` which **only** map between newtypes and their unwrapped form, they won't transitively map directly between different wrappers. This means they **won't** map between `Email` and `UserID` for instance. They also don't allow type-changing transformations. These restrictions mean they tend to result in much better type-inference.

These are based on the `Wrapped` typeclass, which unfortunately isn't derived magically for us like `Coercible` is. Luckily it's already implemented for most newtypes we'll work with, but we'll need instances for the newtypes we just defined.

If we try to use them without defining an instance we'll get a very confusing error message:

```
<interactive>:1:27: error:
    • Couldn't match type ‘Unwrapped Email’ with ‘[a0]’
      Expected type: ASetter Email Email [a0] [a0]
        Actual type: (Unwrapped Email -> Identity (Unwrapped Email))
                     -> Email -> Identity Email
      The type variable ‘a0’ is ambiguous
    • In the first argument of ‘(%~)’, namely ‘_Wrapped'’
      In the second argument of ‘(&)’, namely ‘_Wrapped' %~ reverse’
      In the expression: Email "joe@example.com" & _Wrapped' %~ reverse
```

It's easy to add though; `lens` provides a `makeWrapped` TemplateHaskell helper to generate the needed instance:

```
makeWrapped ''Email
```

With that; we can use `_Wrapped'` and `_Unwrapped'`. I find it a bit tough to remember which one to use when, remember to use the iso that describes the state of the **structure** which you're passing in. So if we're passing in an `Email`, we use the `_Wrapped'` iso to get at the `String` inside.

To me, this naming scheme seems a bit backwards; but c'est la vie, that's just the way it is!

```
>>> Email "joe@example.com" & _Wrapped' %~ reverse
Email {_email = "moc.elpmaxe@eoj"}
```

This is still not great; when working with large optics paths it's very unclear what's happening here; what are we unwrapping? We can use `_Wrapping'` instead to make our path more self-documenting without affecting the behaviour:

```
_Wrapping' :: Wrapped s => (Unwrapped s -> s) -> Iso' s (Unwrapped s)
```

This is an alternate version of `_Wrapped'` which let's you pass the name of the Newtype constructor, it will use the type of the constructor to infer the type of the iso which can also improve inference, it works the same:

```
>>> Email "joe@example.com" & _Wrapping' Email %~ reverse
Email {_email = "moc.elpmaxe@eoj"}
```

Again; I'd typically prefer to use an explicit iso such as one generated by `makeLenses`, but for types in `base` this can be convenient. E.g. `_Wrapping' Sum`.

10.8 Laws

The iso laws are extremely simple, and we've already touched on them when we were writing custom isos so I'll keep this pretty short.

The one and only law: Reversibility

Isos are all about transformations which **maintain information**. The one and only law of an iso is that we can **completely reverse** the transformation.

We can express this as code like so:

```
myIso . from myIso == id

-- And
from myIso . myIso == id
```

As we've learned earlier on, `id` is a valid optic which always focuses its whole argument, it's a valid iso as well! In this case we're using `id` as the iso: `Iso' s s`. In simple terms, if we view through an iso, then view *from* that iso (e.g. view in reverse) then we should end up exactly where we started!

This property should be true for *any* of the ways we can use an iso; but it's sufficient for us to check the behaviour with the 'view' action.

We can try this out on some of our trivial isos:

```
>>> view (reversed . from reversed) ("Testing one two three")
"Testing one two three"

>>> view (from reversed . reversed) ("Testing one two three")
"Testing one two three"

>>> import Numeric.Lens
>>> view (negated . from negated) 23
23

>>> view (from negated . negated) 23
23
```

Since the composition of isos is also an iso, we should expect the law to hold there too!

```
myIso :: Iso' Double Double
myIso = negated . adding 10.0 . multiplying 372.0

>>> view (myIso . from myIso) 23.0
23.0

>>> view (from myIso . myIso) 23.0
23.000000000000227
```

Okay, so it's not **perfect**, but close enough so long as we're not working with money or trying to land on the moon.

Isos are one of the cases where you really should make sure the laws hold. If you can't make the laws work then it's likely your transformation is better suited to a prism or traversal instead of an iso.

Exercises – Iso Laws

Jump to answers

1. The following iso is unlawful; provide a counter example which shows that it breaks the law.

```
mapList :: Ord k => Iso' (M.Map k v) [(k, v)]
mapList = iso M.toList M.fromList
```

2. Is there a lawful implementation of the following iso? If so, implement it, if not, why not?

```haskell
import Data.List.NonEmpty

nonEmptyList :: Iso [a] [b] (Maybe (NonEmpty a)) (Maybe (NonEmpty b))
```

3. Is there a lawful implementation of an iso which 'sorts' a list of elements? If so, implement it, if not, why not?

```haskell
sorted :: Ord a => Iso' [a] [a]
```

4. What about the following iso which pairs each element with an Int which remembers its original position in the list. Is this a lawful iso? Why or why not? If not, try to find a counter-example.

```haskell
sorted :: (Ord a) => Iso' [a] [(Int, a)]
sorted = iso to' from'
  where
    to' xs = L.sortOn snd $ zip [0..] xs
    from' xs = fmap snd $ L.sortOn fst xs
```

11. Indexed Optics

11.1 What are indexed optics?

Indexed optics are an adaptable swiss army knife, they help to address a lot of practical problems you'll come across when using optics in everyday life. Since optics are *already* a swiss army knife all on their own, indexed optics are like a swiss army knife WITHIN a swiss army knife! How can you possibly go wrong?

The basic idea of indexed optics is that they allow you to **accumulate information** about your **current focus** as you dive deeper into an optics path. This information could be anything that makes sense in the context of the optic, but it's commonly used with indexed structures to indicate the location you're currently focusing on. For instance traversing a list will provide the list-index, maps provide the 'key' of the focused value as an index. A tree traversal would provide the location of the current element in the tree, you get the idea.

The good news is that almost all optics and combinators you've already learned work seamlessly with indices! Indexed optics compose just fine with indexed and non-indexed optics alike, there are a few gotchas and tricks with that; but I'll introduce them as the come up.

To start getting an intuition for indices and how they work let's learn a new combinator!

```
itraversed :: TraversableWithIndex i t => IndexedTraversal i (t a) (t b) a b
```

`itraversed` comes from a new typeclass: `TraversableWithIndex`. Luckily most of the instances we could ever want are already defined for us (lists, maps, vectors, trees, etc.). When implementing the typeclass you need to specify the types of the indexes and values for the container, this should remind you of implementing the `Ixed` and `At` typeclasses. We'll cover that later on.

The `itraversed` traversal behaves identically to `traversed`, BUT it keeps track of the current index as it does so! We can use it in place of `traversed` without noticing any difference:

```
>>> toListOf itraversed ["Summer", "Fall",  "Winter", "Spring"]
["Summer","Fall","Winter","Spring"]
```

But the magic happens when we start using **indexed actions**! There are indexed versions of almost all actions, take any of your favourite actions and try adding an `i` to the front. Let's try `itoListOf`:

```
>>> itoListOf itraversed ["Summer", "Fall",  "Winter", "Spring"]
[(0,"Summer"),(1,"Fall"),(2,"Winter"),(3,"Spring")]
```

It includes the list index of each element in the result!

```
itoListOf :: IndexedFold i s a -> s -> [(i, a)]

-- It has an operator:
(^@..) :: s -> IndexedFold i s a -> [(i, a)]
```

It's **very important** to note that it's the **action** which adds the index to the result; the index isn't part of the **focus** of any optics in the path; we can compose optics together without worrying about passing the index manually.

Actions with indexed variants often also have an infix operator version. Try adding `@` in the middle of your favourite operators to get the indexed action:

action	operator	indexed action	indexed operator
`toListOf`	`(^..)`	`itoListOf`	`(^@..)`
`over`	`(%~)`	`iover`	`(%@~)`
`traverseOf`	`(%%~)`	`itraverseOf`	`(%%@~)`
`set`	`(.~)`	`iset`	`(.@~)`
`view`	`(^.)`	`iview`	`(^@.)`

Every **indexed action** accepts an **indexed optic**, e.g. `itoListOf` uses an `IndexedFold`. There are indexed versions of `Lens`, `Traversal`, `Fold`, `Getter`, `Setter`, etc. The lens library doesn't provide indexed versions of Prisms or Isos (they're relatively uncommon), but you could write type aliases for them if you really wanted to.

Typically you don't need an index when accessing only a single element with a lens, prism or iso, so folds and traversals are the most common form of indexed optics.

Let's see how `itraversed` behaves for some different container types which implement `TraversableWithIndex`:

```
>>> let agenda = M.fromList [("Monday", "Shopping"), ("Tuesday", "Swimming")]

-- The index type of maps is the key,
-- so we can get a list of all elements and their key:
>>> agenda ^@.. itraversed
[("Monday","Shopping"),("Tuesday","Swimming")]

-- The index type of tuples is the first half of the tuple:
>>> (True, "value") ^@.. itraversed
[(True,"value")]
```

```
-- The index type of trees is a list of int's
-- which indicates their location in the tree
-- (See the section on indexed data structures)
>>> import Data.Tree
>>> let t = Node "top" [Node "left" [], Node "right" []]
>>> t ^@.. itraversed
[([],"top"),([0],"left"),([1],"right")]
```

In general the index of a structure will be the same type as whatever you'd pass to `ix` or `at` to access it.

11.2 Index Composition

Indexes can compose alongside the optics, but unfortunately it can be a bit less intuitive.

By default, the index of a path will be the index of the **last** optic in the path.

```
>>> let agenda =
  M.fromList [ ("Monday"   , ["Shopping", "Yoga"])
             , ("Saturday", ["Brunch", "Food coma"])
             ]

>>> agenda ^@.. itraversed . itraversed
[ (0,"Shopping")
, (1,"Yoga")
, (0,"Brunch")
, (1,"Food coma")
]
```

The resulting list only includes the list-index of each item, but we've lost track of which day of the week the events occur on!

If we end the path with a non-indexed optic (like `traverse`) we'll get an error:

```
>>> agenda ^@.. itraversed . traverse
error:
    • Couldn't match type ‘Indexed i a (Const (Endo [(i, a)]) a)’
                          with ‘[Char] -> Const (Endo [(i, a)]) [Char]’
      Expected type: IndexedGetting i (Endo [(i, a)]) (Map [Char] [[Char]]) a
        Actual type: ([Char] -> Const (Endo [(i, a)]) [Char])
                     -> Map [Char] [[Char]]
                     -> Const (Endo [(i, a)]) (Map [Char] [[Char]])
    • In the second argument of ‘(^@..)’, namely
        ‘itraversed . traverse’
      In the expression: agenda ^@.. itraversed . traverse
      In an equation for ‘it’: it = agenda ^@.. itraversed . traverse
    • Relevant bindings include
        it :: [(i, a)] (bound at <interactive>:1:1)
```

Ewww. Even after formatting it nicely it's still impossible to read! When using with indexed actions make sure that your path has the type you expect and ends with an indexed optic!

Okay, so when composing optics the index of the whole path is the index of the LAST optic, and only if it's an **indexed optic**. So what if we want to look at an index provided earlier in the path; for example what if we want our list of activities paired with their day of the week instead? For that we need to rethink how we're **composing** our optics! The `lens` library provides several different index-aware composition operators:

- `(<.)`: Use the index of the optic to the left
- `(.>)`: Use the index of the optic to the right (This is how `.` already behaves)
- `(<.>)`: Combine the indices of both sides as a tuple

Let's use `(<.)` to say we only care about the index of the `itraversed` which iterates through the Map, we'll keep the Map key as the index rather than the list offset:

```
>>> let agenda =
  M.fromList [ ("Monday"  , ["Shopping", "Yoga"])
             , ("Saturday", ["Brunch", "Food coma"])
             ]

>>> agenda ^@.. itraversed <. itraversed
[ ("Monday","Shopping")
, ("Monday","Yoga")
, ("Saturday","Brunch")
, ("Saturday","Food coma")
]
```

And if we care about both indices we can keep **both** by using `(<.>)`:

```
>>> agenda ^@.. itraversed <.> itraversed
[ (("Monday",   0), "Shopping")
, (("Monday",   1), "Yoga")
, (("Saturday", 0), "Brunch")
, (("Saturday", 1), "Food coma")
]
```

If we compose more than two indexed optics this way the tuples start to nest. Unlike normal `(.)`, `(<.>)` is **not** associative, re-associating will change the way the tuples nest. It gets a bit silly if you go too far:

```
-- This is a ridiculous thing to do, but it'll show how the indexes nest,
-- we'll use an indexed traversal over the characters of each activity name:
>>> take 8 $ agenda ^@.. itraversed <.> itraversed <.> itraversed
[ (("Monday", (0, 0)), 'S')
, (("Monday", (0, 1)), 'h')
, (("Monday", (0, 2)), 'o')
, (("Monday", (0, 3)), 'p')
, (("Monday", (0, 4)), 'p')
, (("Monday", (0, 5)), 'i')
, (("Monday", (0, 6)), 'n')
, (("Monday", (0, 7)), 'g')
]
```

Custom index composition

If we want to be a bit smarter about how indices are combined we can use `icompose`. The type signature is a bit messy, the gist is that it `icompose` composes any two indexed optics into a new indexed optic by combining the indexes together with a user-defined function.

```
-- Simplified signature; `IndexedOptic` doesn't exist;
-- it's a stand-in for any of the indexed optic types.
icompose :: (i -> j -> k)
         -> IndexedOptic i s t a b
         -> IndexedOptic j a b c d
         -> IndexedOptic k s t c d

-- Real signature
icompose :: Indexable p c
         => (i -> j -> p)
         -> (Indexed i s t -> r)
         -> (Indexed j a b -> s -> t)
```

```
        -> c a b
        -> r
```

As an example we could use `icompose` to append the list index to the day of the week as a string:

```
showDayAndNumber :: String -> Int -> String
showDayAndNumber = (\a b -> a <> ": " <> show b)

>>> agenda ^@.. icompose showDayAndNumber itraversed  itraversed
>>> results
[ ("Monday: 0", "Shopping")
, ("Monday: 1", "Yoga")
, ("Saturday: 0", "Brunch")
, ("Saturday: 1", "Food coma")
]
```

This can be a bit clunky, so if it's something you'll be using often it's typically nice to define a new operator.

Let's define a custom optics composition operator that will automatically append `String` indices with a comma separator.

```
(.++) :: (Indexed String s t -> r)
      -> (Indexed String a b -> s -> t)
      -> Indexed String a b -> r
(.++) = icompose (\a b -> a ++ ", " ++ b)
```

We can use our new composition operator like this:

```
populationMap :: M.Map String (M.Map String Int)
populationMap =
    M.fromList
      [ ("Canada",  M.fromList [("Ottawa", 994837),  ("Toronto", 2930000)])
      , ("Germany", M.fromList [("Berlin", 3748000), ("Munich", 1456000)])
      ]

>>> populationMap ^@.. itraversed .++ itraversed
[ ("Canada, Ottawa"  , 994837)
, ("Canada, Toronto" , 2930000)
, ("Germany, Berlin" , 3748000)
, ("Germany, Munich" , 1456000)
]
```

The hardest part about writing these custom operators is figuring out their type! Optics have complex types which means inference can sometimes be ambiguous.

When defining a new composition operator like this you can follow this template:

```
(<symbols>) :: (Indexed <indexTypeA>  s t -> r)
            -> (Indexed <indexTypeB>  a b -> s -> t)
            -> Indexed <combinedType> a b -> r
(<symbols>) = icompose <combinationFunction>
```

Or, if that fails you, you can leave the type signature off and simply use the operator in an expression, then ask GHCi for the type!

Exercises – Indexed Optics

Jump to answers

1. Fill in the blanks

```
>>> M.fromList [("streamResponse", False), ("useSSL", True)]
      _ itraversed
[("streamResponse",False),("useSSL",True)]

>>> (M.fromList [('a', 1), ('b', 2)], M.fromList [('c', 3), ('d', 4)])
      ^@.. _
[('a',1),('b',2),('c',3),('d',4)]

>>> M.fromList [('a', (True, 1)), ('b', (False, 2))]
      ^@.. itraversed _ _1
[('a', True), ('b', False)]

>>> [ M.fromList [("Tulips", 5), ("Roses", 3)]
    , M.fromList [("Goldfish", 11), ("Frogs", 8)]
    ] ^@.. _
[ ((0,"Roses"), 3)
, ((0,"Tulips"), 5)
, ((1,"Frogs"), 8)
, ((1,"Goldfish"), 11)
]

>>> [10, 20, 30] & itraversed _ (+)
[10,21,32]
```

```
>>> _
      itraversed
      (\i s -> putStrLn (replicate i ' ' <> s))
      ["one", "two", "three"]
one
 two
  three

>>> itraverseOf_
      itraversed
      (\n s -> putStrLn _)
      ["Go shopping", "Eat lunch", "Take a nap"]
0: Go shopping
1: Eat lunch
2: Take a nap
```

11.3 Filtering by index

Similar to how we can use `filter` to narrow down the focuses of a traversal or fold using a predicate, we can use `indices` to narrow down the focuses using a predicate on the index of our fold or traversal!

Here's the type:

```
-- Actual signature:
indices :: (Indexable i p, Applicative f)
        => (i -> Bool) -> Optical' p (Indexed i) f a a
```

It's a bit tough to simplify this type down at all since it doesn't really fit any of the categories of optics we've learned about already. This optic **depends on** the index of the path you compose it with, and we don't really have a good name for that sort of thing yet. Luckily `indices` is quite easy to use. You simply give it a predicate to run on the index and it'll ignore any focuses that don't match.

```
-- Get list elements with an 'even' list-index:
>>> ['a'..'z'] ^.. itraversed . indices even
"acegikmoqsuwy"

>>> let ratings = M.fromList
      [ ("Dark Knight", 94)
      , ("Dark Knight Rises", 87)
      , ("Death of Superman", 92)]

>>> import Data.List.Lens (prefixed)
>>> ratings ^.. itraversed . indices (has (prefixed "Dark"))
[ 94
, 87
]
```

If we want to be more specific we can target an **exact** index using the `index` filter:

```
index :: (Indexable i p, Eq i, Applicative f)
      => i -> Optical' p (Indexed i) f a a
```

`index` works similarly to `indices`, but ignores any focus that doesn't have the exact index you specify. Note that depending on the structure it's entirely possible that there may be more than element with the same index in a traversal.

```
>>> ['a'..'z'] ^? itraversed . index 10
Just 'k'

>>> ratings ^? itraversed . index "Death of Superman"
Just 92
```

Usually you should prefer the `ix` traversal when working with data structures, but when combining many indexed optics sometimes it's necessary to use `indices` or `index`.

Exercises – Index Filters

Jump to answers

1. Given the following exercise schedule:

```
exercises :: M.Map String (M.Map String Int)
exercises = M.fromList
  [ ("Monday"   , M.fromList [("pushups", 10), ("crunches", 20)])
  , ("Wednesday", M.fromList [("pushups", 15), ("handstands", 3)])
  , ("Friday"   , M.fromList [("crunches", 25), ("handstands", 5)])
  ]
```

- Compute the total number of "crunches" you should do this week.
- Compute the number of reps you need to do across all exercise types on Wednesday.
- List out the number of pushups you need to do each day, you can use `ix` to help this time if you wish.

2. Given the following board:

```
let board = ["XOO"
            ,".XO"
            ,"X.."]
```

- Generate a list of positions alongside their `(row, column)` coordinates.
- Set the empty square at `(1, 0)` to an `'X'`. HINT: When using the custom composition operators you'll often need to introduce parenthesis to get the right precedence.
- Get the 2nd **column** as a list (e.g. `"OX."`). Try to do it using `index` instead of `indices`!
- Get the 3rd **row** as a list (e.g. `"X.."`). Try to do it using `index` instead of `indices`! HINT: The precedence for this one can be tricky too.

11.4 Custom indexed optics

Writing your own indexed optics can sometimes be useful, let's work through an example to see what they can be used for and how to build them manually.

Let's say we're modelling a game of Tic-Tac-Toe! We want to be precise about it, so we define a grid that's **exactly** `3x3`.

```
data Board a =
    Board
      a a a
      a a a
      a a a
  deriving (Show, Foldable)
```

We want to allow indexing into the grid, but want to avoid using `Int`s because they might be out-of-bounds and we don't want the game to crash. Instead we'll define our own position type:

```
data Position = I | II | III
  deriving (Show, Eq, Ord)
```

Now we can talk about individual squares in our grid as an X-Y coordinate of `Positions`. Let's define a value of our `Board` type for testing:

```
testBoard :: Board Char
testBoard =
    Board
      'X' 'O' 'X'
      '.' 'X' 'O'
      '.' 'O' 'X'
```

Looks like `'X'` won this one!

Custom IndexedFolds

Now we'll want some way to access the positions in our grid, let's start by building an `IndexedFold` which accesses each board 'slot' with the approprate `Positions` as an index. Normally we'd build a custom fold with `folding`; so to build an `IndexedFold` we can use `ifolding`:

```
-- Simple signature:
ifolding :: Foldable f => (s -> f (i, a)) -> IndexedFold i s a

-- Actual signature
(Foldable f, Indexable i p, Contravariant g, Applicative g) => (s -> f (i, a)) -> Ov\
er p g s t a b
```

The simple signature tells us that we simply have to project our focuses from the structure into a foldable container (like a list). The difference from `folding` is that we *also* provide an index of type `<i>` alongside each value. Let's try it out:

```
slotsFold :: IndexedFold (Position, Position) (Board a) a
slotsFold =
    ifolding $ \board ->
        -- Use a list comprehension to get the list of all coordinate pairs
        -- in the correct order, then zip them with all the slots in our board
        zip [(x, y) | y <- [I, II, III], x <- [I, II, III]]
            (toList board)
```

We're piggy-backing on the `Foldable` instance for our `Board`, but we're adding some extra information in the form of coordinates.

Now we can use our indexed fold to see all the moves and their coordinates:

```
>>> testBoard ^@.. slotsFold
[ ((I,I)    , 'X')
, ((II,I)   , 'O')
, ((III,I)  , 'X')
, ((I,II)   , '.')
, ((II,II)  , 'X')
, ((III,II) , 'O')
, ((I,III)  , '.')
, ((II,III) , 'O')
, ((III,III), 'X')
]
```

We can use `indices` to filter our fold by a predicate **on the index**. Let's get only the second row of the board:

```
-- Filter indices where the Y coord is 'II'
>>> testBoard ^@.. slotsFold . indices ((== II) . snd)
[ ((I  , II), '.')
, ((II , II), 'X')
, ((III, II), 'O')
]
```

Cool stuff, but what if we want to edit the slots in our board? I'd recommend implementing the `Ixed` typeclass we learned about earlier so that we can use `ix` to access particular slots, but I'll also show you how to do it by building an `IndexedTraversal`!

Custom IndexedTraversals

This particular indexed traversal will have a bit of boiler-plate, but they're not usually this bad.

Unfortunately, there isn't a helper method for building custom traversals, just like regular traversals we'll have to do this by hand.

```
-- I define a polymorphic indexed traversal
-- with a tuple of positions as the index:
slotsTraversal
  :: IndexedTraversal (Position, Position) (Board a) (Board b) a b
slotsTraversal p (Board
                    a1 b1 c1
                    a2 b2 c2
                    a3 b3 c3)
  = Board <$> indexed p (I  , I)   a1
          <*> indexed p (II , I)   b1
```

```
          <*> indexed p (III, I)   c1
          <*> indexed p (I  , II)  a2
          <*> indexed p (II , II)  b2
          <*> indexed p (III, II)  c2
          <*> indexed p (I  , III) a3
          <*> indexed p (II , III) b3
          <*> indexed p (III, III) c3
```

Okay, it looks like a lot just because we have to do everything *nine* times, but it's not as bad as it looks. We define an `IndexedTraversal` just like a regular `Traversal`, the only difference is that we use the `indexed` function (from `lens`) to pass an index to the handler each time we use it. Typically you can write a normal traversal, then just sprinkle `indexed` around wherever you call your handler, pass in the index alongside each value, and everything works out.

Let's try it out first, then we'll see why we build it this way. Note that our `slotsTraversal` makes the `slotsFold` redundant, since every traversal is also a fold.

```
>>> testBoard ^@.. slotsTraversal
[ ((I,I)    , 'X')
, ((II,I)   , 'O')
, ((III,I)  , 'X')
, ((I,II)   , '.')
, ((II,II)  , 'X')
, ((III,II) , 'O')
, ((I,III)  , '.')
, ((II,III) , 'O')
, ((III,III), 'X')
]
```

We can filter it using `indices` just like last time, but now we can set or modify the slots!

Let's get a quick win by setting the whole second row to `'O'`.

```
>>> testBoard & slotsTraversal  . indices ((== II) . snd) .~ 'O'
Board
  'X' 'O' 'X'
  'O' 'O' 'O'
  '.' 'O' 'X'
```

We can even write a function to print our board, if we pass the coordinates to our printing function it can easily add newlines in the right spots!

```
printBoard :: Board Char -> IO ()
printBoard = itraverseOf_ slotsTraversal printSlot
  where
    printSlot (III, _) c = putStrLn [c]
    printSlot (_, _) c = putStr [c]

>>> printBoard testBoard
XOX
.XO
.OX
```

—

Let's take a deeper look at why our implementation uses this weird `indexed` function. To see why we need to do this let's take a look behind the curtain and see the expanded type of an `IndexedTraversal`:

```
type IndexedTraversal i s t a b
  = forall p f. (Indexable i p, Applicative f) => p a (f b) -> s -> f t
```

We see here that instead of a **function** handler, we're passed a `<p>`, which is constrained to be `Indexable`. The `Indexable` constraint basically just means that it's a `Profunctor` which can be used like a function handler that takes an additional argument `<i>` which is the index.

Don't worry about understanding the underlying machinery here, it gets messy quickly, this is where the `lens` library starts to get a bit a less elegant. The important part is that any `Indexable` profunctor can be reduced into a regular function using `indexed`:

```
indexed :: Indexable i p => p a b -> i -> a -> b
```

And that's why we call `indexed` on the handler in our `slotsTraversal`.

That's a crash course on writing custom folds and traversals the hard way!

You can also write an indexed lens using the `ilens` helper; unlike the others it's pretty straightforward:

```
ilens :: (s -> (i, a)) -> (s -> b -> t) -> IndexedLens i s t a b
```

You simply provide the index type alongside the focus in your getter and `ilens` will wire it up correctly!

Index helpers

There are many other ways to add indexes to optics. The simplest is `indexing`

```
indexing :: Traversal s t a b -> IndexedTraversal Int s t a b
indexing :: Lens s t a b      -> IndexedLens Int  s t a b
indexing :: Fold s a          -> IndexedFold Int s a
indexing :: Getter s a        -> IndexedGetter Int s a
```

The `indexing` helper takes a normal optic and simply adds a numeric index alongside its elements. For example:

```
>>> ("hello" :: Text) ^@.. indexing each
[(0,'h'),(1,'e'),(2,'l'),(3,'l'),(4,'o')]
```

It's an easy way to keep track of your position within a simple optic.

We can re-map or edit the indexes of an optic using `reindexed`:

```
reindexed :: Indexable j p => (i -> j) -> (Indexed i a b -> r) -> p a b -> r
```

Yet another difficult and confusing type signature! This one uses the provided function to map over the indexes of an optic.

```
>>> ['a'..'c'] ^@.. itraversed
[(0,'a'),(1,'b'),(2,'c')]

>>> ['a'..'c'] ^@.. reindexed (*10) itraversed
[(0,'a'),(10,'b'),(20,'c')]

-- We can even change index types!
>>> ['a'..'c'] ^@.. reindexed show itraversed
[("0",'a'),("1",'b'),("2",'c')]
```

—

Another helper is `selfIndex`:

```
selfIndex :: Indexable a p => p a fb -> a -> fb
```

The `selfIndex` helper is another strange optic that doesn't really fit into other categories. It can be injected into a path to set the index of the path to the current value. This is particularly useful when you need context from "higher up" in the structure to do some edits "lower down". It's tough to imagine what this is good for until you come across it in the wild. It turns out to be pretty handy working with JSON objects, so we'll see more examples when we get there.

```
-- `selfIndex` copies a snapshot of the current focus into the index
>>> [("Betty", 37), ("Veronica", 12)]
      ^@.. itraversed . selfIndex <. _2
[(("Betty",37),37), (("Veronica",12),12)]
```

Exercises – Custom Indexed Optics

Jump to answers

1. Write a sensible implementation for the following indexed fold:

```
pair :: IndexedFold Bool (a, a)  a
```

Such that:

```
>>> ('a', 'b') ^@.. pair
[(False,'a'),(True,'b')]
```

Once you've done that; try writing it as an indexed traversal:

```
pair' :: IndexedTraversal Bool (a, a) (b, b)  a b
```

2. Use `reindexed` to provide an indexed list traversal which starts at '1' instead of '0'.

```
oneIndexed :: IndexedTraversal Int [a] [b] a b
```

It should behave like so:

```
>>> ['a'..'d'] ^@.. oneIndexed
[(1,'a'),(2,'b'),(3,'c'),(4,'d')]
```

If you want a challenge, try using `selfIndex`, `<.>` and `reindexed` to write a traversal indexed by the distance to the **end** of the list. E.g.

```
invertedIndex :: IndexedTraversal Int [a] [b] a b


>>> ['a'..'d'] ^@.. invertedIndex
[(3,'a'),(2,'b'),(1,'c'),(0,'d')]
```

3. Build the following combinators using only compositions of other optics:

```
chars :: IndexedTraversal Int T.Text T.Text Char Char

>>> ("banana" :: T.Text) ^@.. chars
[(0,'b'),(1,'a'),(2,'n'),(3,'a'),(4,'n'),(5,'a')]
```

This one's a thinker; index each character (except newlines) by their line and column numbers:

```
charCoords :: IndexedTraversal (Int, Int) String String Char Char

>>> "line\nby\nline" ^@.. charCoords
[ ((0,0),'l'),((0,1),'i'),((0,2),'n'),((0,3),'e')
, ((1,0),'b'),((1,1),'y')
, ((2,0),'l') ,((2,1),'i'),((2,2),'n'),((2,3),'e')]
```

11.5 Index-preserving optics

As you've probably already realized, using `(<.)`, `(.>)` and `(<.>)` all the time can be a bit annoying. By default, indexes are "forgotten" with every composition of optics, if you want an index to persist to the end you need to carry it there manually using the composition operators.

The `lens` library provides a way around that in the way of **index-preserving optics**!

Index preserving optics are just regular optics which **pass-through** any existing index in the path. As always, this is easiest explained with an example:

```
-- This one won't compile!
-- We require an indexed optic since we're using `^@..`,
-- but `_1` 'forgets' the index from `itraversed`
-- without adding any index of its own.
>>> [('a', True), ('b', False), ('c', True)]
      ^@.. itraversed . _1
INSERT VERBOSE ANGRY ERROR HERE

-- However, we can turn `_1` into an index preserving lens!
>>> let _1' = cloneIndexPreservingLens _1
-- Now the index 'passes-through' `_1'` to the end.
>>> [('a', True), ('b', False), ('c', True)]
      ^@.. itraversed . _1'
[(0,'a'),(1,'b'),(2,'c')]

-- The previous is equivalent to use explicit index passing with `<.`
>>> [('a', True), ('b', False), ('c', True)]
      ^@.. itraversed <. _1
```

Obviously in this case it's quicker, easier, and clearer to just use `(<.)`, but if you're exporting lenses for a library it can be helpful to know about this trick, it can often improve usability and user-experience if composing your optics just does "The Right Thing™".

The easiest way to make an index preserving optic is to write a normal optic and use the appropriate `cloneIndexPreserving*` method:

```
cloneIndexPreservingLens :: Lens s t a b
                         -> IndexPreservingLens s t a b

cloneIndexPreservingTraversal :: Traversal s t a b
                              -> IndexPreservingTraversal s t a b

cloneIndexPreservingSetter :: Setter s t a b
                           -> IndexPreservingSetter s t a b
```

Strangely, `cloneIndexPreservingFold` seems to be missing from the library.

There's also an `iplens` combinator for constructing index-preserving lenses from scratch. Lenses are the most likely to be `IndexPreserving` since folds and traversals are more likely to have their **own** indexes.

`iplens` is identical to `lens` except the resulting lens will be **index-preserving**.

```
iplens :: (s -> a) -> (s -> b -> t) -> IndexPreservingLens s t a b
```

You won't see index-preserving optics come up very often, but sometimes they're just what the doctor ordered. Keep them in mind if you're ever designing an optics-based API.

12. Dealing with Type Errors

It's inevitable that working with optics will eventually lead you to some nasty type errors. There's a lot of wonderful functionality hidden behind those lens combinators, but with great power comes great complexity and sometimes the abstraction leaks a bit. It's impossible to avoid these errors altogether, as programmers a huge part of our jobs is making mistakes and we're bound to compose things in the wrong order or make incorrect assumptions, so we'll have to learn how to correct our mistakes when they occur.

The most important thing is to **not get discouraged**. When facing complex error messages the secret is to **read the message**, **break it down into pieces**, and use some **heuristics** to help understand the problem. Also: **PRACTICE**! Every time you come across an error, spend some time trying to solve it before asking for help, it will improve your error-reading skills as you go along.

12.1 Interpreting expanded optics types

We've already learned a *little* about how **Van Laarhoven** optics are represented as part of the custom traversals section, learning to recognize these patterns can help a lot when trying to determine *which* optic is expected to go *where*.

Let's start with a little case study and see how this technique can help us troubleshoot problems, let's say we're trying to multiply every element of a list by `10`:

```
>>> [(1, True), (2, False), (3, True)]
      & folded . _1 *~ 10

error:
    • Could not deduce (Contravariant Identity)
        arising from a use of ‘folded’
      from the context: Num a
        bound by the inferred type of it :: Num a => [(a, Bool)]
        at <interactive>:1:1-54
    • In the first argument of ‘(.)’, namely ‘folded’
      In the first argument of ‘(*~)’, namely ‘folded . _1’
      In the second argument of ‘(&)’, namely ‘folded . _1 *~ 10’
```

Uh-oh, something went wrong! The important part of the error is right at the top: "`Could not deduce (Contravariant Identity)`". The first step to understanding this error is to separate the relevant **data type** from the **constraints**. In this case we have:

Constraints
`Contravariant`
Data types
`Identity`

We know that we have a mismatch because `Identity` doesn't implement `Contravariant`. We can look in the optic ingredients table to see that we expect to see `Identity` when running modifiers like `set` or `over`, so it makes sense we'd see it when running `(*~)` which modifies the value via multiplication. That means our problem lies with the `Contravariant` constraint. We have two optics it could be coming from, either `folded` or `_1`. We can look up `Contravariant` in the table to learn that it's typically required by **folds**, and `folded` is a **fold**!

That discovery means that we'll need to change `folded` to something without a `Contravariant` requirement, looking through the table we see that `Traversals` don't have a `Contravariant` constraint, they have `Applicative` instead, and `Identity` implements `Applicative`! If we swap `folded` for `traversed` it should work out better:

```
>>> [(1, True), (2, False), (3, True)] & traversed . _1 *~ 10
[(10,True),(20,False),(30,True)]
```

This process isn't always so straight-forward, but it's often a good approach to take to at least understand which constraints are at play in your expression. The more you practice the more you'll start to remember which optics have which constraints and the whole thing gets a bit easier.

12.2 Type Error Arena

To help you build error solving skills we'll practice them in a *safe* environment. Welcome to the "**Type-Error Arena**" where you and your wits will face off against the most devious of lensy type-errors armed with nothing but GHCi and its (sometimes) archaic error messages. I'll present a **puzzle** in the form of an expression which fails to compile. Your task is to determine the problem by looking at the error message, and fix the code so it compiles! I'll provide hints following each question, but please try to solve it on your own before looking at the answer.

Please note that type-errors can change depending on your GHC version, so it's possible the errors from your GHC are *slightly* different from the ones I include here, but you should still be able to follow along.

With that; you're ready to face your first foe in the **Type-Errors Arena**!

See if you can determine what's wrong with this first expression by **carefully reading the errors**.

First Foe: Level 1 Lenslion

```
>>> :load
Ok, modules loaded: none.
>>> view _1 ('a', 2)
error:
    Variable not in scope: view :: t0 -> (Char, Integer) -> t

error:
    • Found hole: _1 :: t0
      Where: ‘t0’ is an ambiguous type variable
      Or perhaps ‘_1’ is mis-spelled, or not in scope
    • In the first argument of ‘view’, namely ‘_1’
      In the expression: view _1 ('a', 2)
      In an equation for ‘it’: it = view _1 ('a', 2)
    • Relevant bindings include it :: t (bound at <interactive>:1:1)
      Valid hole fits include: ...
```

Hints:

- Hrmm, we're getting two errors here, do they seem separate, or is it possible that they're related?
- What could it imply when it says that the `view` variable is not in scope?
- I know I usually assume all the right things have been imported when presenting examples, is it possible I forgot to import something this time?

Go to the answer

Level 2 Tuplicant

```
>>> view ("abc", 123) _1

error:
    • Couldn't match type ‘([Char], Integer)’
                      with ‘(t -> Const t t)
                            -> ((a0 -> f0 b0) -> s0 -> f0 t0)
                            -> Const t ((a0 -> f0 b0) -> s0 -> f0 t0)’
      Expected type: Getting t ((a0 -> f0 b0) -> s0 -> f0 t0) t
        Actual type: ([Char], Integer)
    • In the first argument of ‘view’, namely ‘("abc", 123)’
      In the expression: view ("abc", 123) _1
      In an equation for ‘it’: it = view ("abc", 123) _1
    • Relevant bindings include it :: t (bound at <interactive>:1:1)
```

See if you can solve it!

Hints:

- Lenses and optics tend to have types roughly matching: `(a -> f b) -> s -> f t`. Do you see anything in the error message that vaguely resembles this? Maybe that's a lens!
- Which two types is GHC not able to unify? Is it possible we're trying to pass in the wrong thing?

Go to the answer

Level 3 Settersiren

```
>>> ("old", False) $ _1 .~ "new"

<interactive>:1:1: error:
    • Couldn't match expected type ‘(s0 -> t0) -> t’
                  with actual type ‘([Char], Bool)’
    • The first argument of ($) takes one argument,
      but its type ‘([Char], Bool)’ has none
      In the expression: ("old", False) $ _1 .~ "new"
      In an equation for ‘it’: it = ("old", False) $ _1 .~ "new"
    • Relevant bindings include it :: t (bound at <interactive>:1:1)
```

Hints:

- Focus on:

```
• The first argument of ($) takes one argument,
  but its type ‘([Char], Bool)’ has none
```

- Try rebuilding this expression without using `$` at all

Go to the answer

Level 4 Composicore

```
>>> view (_3 . _1) (('a', 'b', 'c'), 'd')

<interactive>:1:1: error:
    • No instance for
    (Field3 ((Char, Char, Char), Char) ((Char, Char, Char), Char) a0 b0)
```

Hints:

- Double-check the *ordering* of everything

Go to the answer

Level 5 Foldasaurus

```
>>> ("blue", Just (2 :: Int)) ^. _2 . _Just

error:
    • No instance for (Monoid Int) arising from a use of ‘_Just’
    • In the second argument of ‘(.)’, namely ‘_Just’
      In the second argument of ‘(^.)’, namely ‘_2 . _Just’
      In the expression: ("blue", Just (2 :: Int)) ^. _2 . _Just
```

Hints:

- The `Monoid` requirement is a bit of a red-herring which is a consequence of how `(^.)` is implemented.
- The value inside a `Maybe` might be missing, but `(^.)` always expects to get a value. How can we relax the requirement that we NEED a focus to exist?
- Is there a combinator similar to `(^.)` which might help us handle failure?

Go to the answer

Level 6 Higher Order Beast

```
>>> ['a'..'z'] ^.. taking 5 . folded

error:
    • Couldn't match type ‘[]’ with ‘p a’
      Expected type: Getting
                       (Endo
                          [BazaarT p f a a a])
                       [Char]
                       (BazaarT p f a a a)
        Actual type: (BazaarT p f a a a
                      -> s0 -> BazaarT p f a a a)
                     -> Over p f s0 (BazaarT p f a a a) a a
    • Probable cause: ‘(.)’ is applied to too few arguments
      In the second argument of ‘(^..)’, namely ‘taking 5 . folded’
      In the expression: ['a' .. 'z'] ^.. taking 5 . folded
      In an equation for ‘it’: it = ['a' .. 'z'] ^.. taking 5 . folded
    • Relevant bindings include
        it :: [BazaarT p f a a a]
```

Hints:

- This one's tricky, the mistake made here is more substantial than in other problems.
- Look up the type signature of `taking`, is it being used correctly here?
- Remember that `taking` is a **higher-order** combinator.

Go to the answer

Level 7 Traversacula

```
>>> over both putStrLn ("one", "two")

error:
    • No instance for (Show (IO ())) arising from a use of ‘print’
    • In a stmt of an interactive GHCi command: print it
```

Hints:

- Looks like it's running, but not getting te effect we want! Check what the type of this expression is.
- We're calling an **effectful** function on each element, but **over** is for **pure** functions, is there an effectful version of **over**?

Go to the answer

13. Optics and Monads

Monads are an integral part of writing software, so it's only natural that there be some intersection between lenses and monads! The `lens` library exports several combinators which exist solely as quality of life improvements for when you're working with optics in a specific monadic contexts.

13.1 Reader Monad and View

Most Haskell apps have monad stacks and most monad stacks have some a `Reader` in them somewhere. It's a handy way to pass settings and configuration around to everywhere that need them. If you have a large or nested environment object which you've defined lenses for, there's a quick and easy way to use them to access fields in a reader monad.

Here's a simple example environment:

```
type UserName = String
type Password = String
data Env =
    Env { _currentUser :: UserName
        , _users :: M.Map UserName Password
        }
    deriving Show

makeLenses ''Env
```

Given this environment, if we wanted to write a monadic action which prints out the current user, we'd normally do it like this:

```
printUser :: ReaderT Env IO ()
printUser = do
    user <- asks _currentUser
    liftIO . putStrLn $ "Current user: " <> user

main :: IO ()
main = runReaderT printUser (Env "jenkins" (M.singleton "jenkins" "hunter2"))
```

```
>>> main
Current user: jenkins
```

The equivalent lensy version uses `view` like this:

```
printUser :: ReaderT Env IO ()
printUser = do
    user <- view currentUser
    liftIO . putStrLn $ "Current user: " <> user
```

This is actually the exact same **view** function we're used to using with lenses! Here's the real type of `view`:

```
view :: MonadReader s m => Getting a s a -> m a
```

`view` takes an optic and returns an action in some `MonadReader` which returns the focus, this works in the standard non-monadic case because functions have a `MonadReader` instance! That is, the function `s -> a` is a valid `MonadReader s m => m a` where `m ~ (->) s`. If you haven't seen this before it can be a bit surprising, but try it out to see for yourself.

Fun fact, `preview` works the same way! This means we can run folds over the environment and get back a result wrapped in `Maybe`:

```
getUserPassword :: ReaderT Env IO ()
getUserPassword = do
    userName <- view currentUser
    maybePassword <- preview (users . ix userName)
    liftIO $ print maybePassword

main :: IO ()
main = runReaderT getUserPassword (Env "jenkins" (M.singleton "jenkins" "hunter2"))
```

```
>>> main
Just "hunter2"
```

Using `view` is the idiomatic way of accessing your environment if you've got lenses defined for your environment.

13.2 State Monad Combinators

`State` is another common Haskell monad, since both lenses and the State monad revolve around the ideas of accessing and mutating state they make a lovely pair.

The lenses library provides combinators for both accessing and mutating state within the State monad.

Let's explore this with a small example; we'll write a State monad action which runs a till calculator for recording the sale of a couple beers.

We'll start with some setup:

```
import Control.Lens
import Control.Monad.State
import Text.Printf

data Till =
    Till { _total   :: Double
         , _sales   :: [Double]
         , _taxRate :: Double
         } deriving Show

makeLenses ''Till
```

Our simple till can hold a running total of the drinks in an order, it also keeps a list of ALL sales made throughout the day, and holds the current local tax rate so we can use it in our calculations.

Let's write up an action which represents the sale a few beers:

```
saleCalculation :: StateT Till IO ()
saleCalculation = do
    total .= 0
```

We start by clearing the current total (just in case). `(total .= 0)` specifies to use the `total` lens over our state, and set the focus to `0`. The `(.=)` operator should remind you of `.~`, which is the operator we use for "pure" setters. Almost ALL setter combinators have `State` equivalents which simply replace the ~ with an =! Here's the type in case you're wondering:

```
(.=) :: MonadState s m => Lens s s a b -> b -> m ()
```

We can see a few things from the type. Firstly, note that these combinators work in any monad which implements `MonadState`, including `State` and `StateT` of course! We notice the lens isn't 'fully' polymorphic, that's because the type of `<s>` is fixed by the State Monad, so we're not allowed to change it.

Let's add a few beer sales to the total:

```
saleCalculation :: StateT Till IO ()
saleCalculation = do
    total .= 0
    total += 8.55 -- Delicious Hazy IPA
    total += 7.36 -- Red Grapefruit Sour
```

Now we're using `(+=)`! This works just like the pure version: `(+~)`. It **adds** the provided number to the total, mutating the State!

```
saleCalculation :: StateT Till IO ()
saleCalculation = do
    total .= 0
    total += 8.55 -- Delicious Hazy IPA
    total += 7.36 -- Red Grapefruit Sour
    totalSale <- use total
    liftIO $ printf "Total sale: $%.2f\n" totalSale
```

Now that we've added the patron's beers to the order, we want to print out the total. We can access the portion of our State focused by a lens with the `use` combinator:

```
use :: MonadState s m => Lens' s a -> m a

-- It actually works on any Getter!
use :: MonadState s m => Getting a s a -> m a
```

`use` is like `view`, but for `MonadState` rather than `MonadReader`!

Let's run what we've got so far, I'll use `execStateT` so we can see the final state!

```
>>> execStateT saleCalculation (Till 0 [] 1.11)
Total sale: $15.91
Till {_total = 15.91, _sales = [], _taxRate = 1.11}
```

We see that the total has been properly updated, and we've printed it as part of our calculation.

Next we'll record the total of the sale for our records:

```
saleCalculation :: StateT Till IO ()
saleCalculation = do
    total .= 0
    total += 8.55 -- Delicious Hazy IPA
    total += 7.36 -- Red Grapefruit Sour
    totalSale <- use total
    liftIO $ printf "Total sale: $%.2f\n" totalSale
    sales <>= [totalSale]
```

This uses the stateful version of `(<>~)` which uses a `Monoid` to append the sale to the focus of our `sales` lens.

Lastly, let's add in the tax, update the total, and print the final total:

```
saleCalculation :: StateT Till IO ()
saleCalculation = do
    total .= 0
    total += 8.55 -- Delicious Hazy IPA
    total += 7.36 -- Red Grapefruit Sour
    totalSale <- use total
    liftIO $ printf "Total sale: $%.2f\n" totalSale
    sales <>= [totalSale]
    total <~ uses taxRate (totalSale *)
    taxIncluded <- use total
    liftIO $ printf "Tax included: $%.2f\n" taxIncluded

>>> execStateT saleCalculation (Till 0 [] 1.11)
Total sale: $15.91
Tax included: $17.66
Till {_total = 17.6601, _sales = [15.91], _taxRate = 1.11}
```

To finish off we multiply the sale by the tax rate, then print out the new total. We use two new combinators here; let's start with `uses`:

```
uses :: MonadState s m => Lens' s a -> (a -> r) -> m r
```

`uses` gets a value from the state, and runs an accessor function over the focus before returning it. It's a simple convenience function to make `(f <$> use l)` a little easier to read.

If you look closely, you'll see we don't actually bind the result! We use the `(<~)` combinator, that's not a typo! This combinator is a short-hand for running some monadic action and setting its result into some portion of your state. In this case we run a monadic action to calculate the tax rate, then store it into our total using the `total` lens. When we bind `taxIncluded` on the next line we see that it has the tax included.

```
(<~) :: MonadState s m => Lens s s a b -> m b -> m ()
```

I've personally found that `(<~)` doesn't come up very often, but it can save a little boilerplate when it does.

All of these `MonadState` combinators have alternate versions which return the **existing** or **altered** versions of the focus, see `(<+=)`, `(<<+=)`, `(<<~)`, etc.

That about wraps up the basic monadic combinators you'll need. There are also combinators for dealing with `MonadWriter`, but these come up so rarely that I'll leave that as homework for you to dig into yourself.

13.3 Magnify & Zoom

Reader and State monads are wonderful abstractions, but sometimes it's annoying that we need to have the same environment or state throughout the *entire* computation. It prevents us from specifying more granular dependencies, and typically means that every action we want to use in that monad stack has a global dependency on the environment or state type.

The lens library provides combinators which allow us to 're-scope' a Reader or State monad to a portion of the type focused by a lens.

We'll start with `magnify`!

```
magnify :: Lens' s a -> ReaderT a m r -> ReaderT s m r

-- or the more permissive:
magnify :: Getter s a -> ReaderT a m r -> ReaderT s m r
```

`magnify` uses the `Magnified` type family, so you'll need to implement that to use it for your custom types, but I'll show how to use it for `ReaderT` here.

We'll write an environment type:

```
data Weather =
    Weather { _temperature :: Float
            , _pressure    :: Float
            }
  deriving Show

makeLenses ''Weather
```

And I'll write a ReaderT action which prints its environment with a given statistic name:

```
printData :: String -> ReaderT Float IO ()
printData statName = do
    num <- ask
    liftIO . putStrLn $ statName <> ": " <> show num
```

If we have a `ReaderT Weather IO ()` we'll need to 'focus' individual stats within the `Weather` so that we can run our `printData` helper. We can't embed `printData` directly in a `ReaderT Weather IO ()` because the environment types don't match!

Here's how it looks to use magnify

```
weatherStats :: ReaderT Weather IO ()
weatherStats = do
    magnify temperature (printData "temp")
    magnify pressure (printData "pressure")

>>> runReaderT weatherStats (Weather 15 7.2)
temp: 15.0
pressure: 7.2
```

You can see that by magnifying the Reader environment through a lens which focuses a `Float` we can run `printData` against that particular stat!

We can do the exact same thing for the `StateT` monad using `zoom`:

```
zoom :: Monad m => Lens' s a -> StateT a m r -> StateT s m r
```

Let's write a State action which runs against our `Weather` object and converts the temperature stored there from celsius to fahrenheit:

```
convertCelsiusToFahrenheit :: StateT Float IO ()
convertCelsiusToFahrenheit = do
    modify (\celsius -> (celsius * (9/5)) + 32)
```

This should probably just be a pure function, but bear with me. This action assumes the temperature we're converting is the state of the monad. In order to run it in a State monad over the `Weather` type we'll need to zoom-in on the temperature when we run it:

```
weatherStats :: StateT Weather IO ()
weatherStats = do
    zoom temperature convertCelsiusToFahrenheit
```

By running this action we can see it modifies the temperature properly:

```
>>> execStateT weatherStats (Weather 32 12)
Weather {_temperature = 89.6, _pressure = 12.0}
```

14. Classy Lenses

14.1 What are classy lenses and when do I need them?

Just in case you've never heard of classy lenses before we'll go over roughly what they are and what they're for. Classy lenses aren't really a new type of optic so much as a whole **design pattern**. Typically one reaches for classy lenses to solve one of the following design needs:

- Polymorphism over specific **record fields**
- Separating layers of logic without cross-dependencies
- Isolating the 'knowledge' of a given module of code

It's tough to describe these design concerns as bullet points, so let's see an example of some problems you can run into in large apps and how classy optics help solve them.

No duplicate record fields

A consistent annoyance in Haskell is that you **can't have two records with the same field names**. The record accessors conflict with each other, so you need to either qualify them by their module or add prefixes to disambiguate them. Classy lenses provide an (imperfect) solution for this. Let's take a look.

If we have several records which all have a `name` field, we need to disambiguate the fields. The idiom is to use record name as a field prefix:

```
data Person =
    Person { _personName :: String
           } deriving Show

data Pet =
    Pet { _petName :: String
        }
    deriving Show
```

Not only is this annoyingly verbose, but there's a greater problem in Haskell as a whole: we can't write code which is polymorphic over record fields! If we want to print a record's **name**, we need a separate method for each record type. Even though the name field has the same type, there's no easy way to unify these (almost) identical functions:

```
greetPerson :: Person -> IO ()
greetPerson p = putStrLn $ "Hello " <> _personName p <> "!"

greetPet :: Pet -> IO ()
greetPet p = putStrLn $ "Hello " <> _petName p <> "!"
```

We see that each of them work as expected, but we need to use the corresponding function for each record type:

```
>>> greetPerson (Person "Calvin")
Hello Calvin!

>>> greetPet (Pet "Hobbes")
Hello Hobbes!
```

This is where `makeFields` comes to help! If we've defined our records in this idiomatic way (i.e. with their record name as a field-prefix for all fields) then we can use `makeFields` to generate record-polymorphic lenses for all the fields. Let's see what that actually means:

```
-- We need a few extra pragmas:
{-# LANGUAGE TemplateHaskell #-}
{-# LANGUAGE MultiParamTypeClasses #-}
{-# LANGUAGE RankNTypes #-}
{-# LANGUAGE FunctionalDependencies #-}

data Person =
    Person { _personName :: String
           } deriving Show

data Pet =
    Pet { _petName :: String
        }
    deriving Show

makeFields ''Person
makeFields ''Pet
```

We can `:browse` to see what was generated:

```
data Person = Person {_personName :: String}
data Pet = Pet {_petName :: String}
class HasName s a | s -> a where
  name :: Lens' s a
  {-# MINIMAL name #-}
```

You can see that instead of making separate lenses for each record type, it has detected and stripped the record-name prefixes and created a unified `HasName` class! If we check `:info HasName` we see it's generated instances for each of the records too:

```
>>> :info HasName
class HasName s a | s -> a where
  name :: Lens' s a
  {-# MINIMAL name #-}
instance HasName Person String
instance HasName Pet String
```

`HasName` is a **Field Class**, a typeclass which provides a lens we can use to **get** or **set** the value of that field for any type which implements the typeclass.

It wrote an instance for each of our records. You'll see it uses `FunctionalDependencies` in the class definition, this helps us maintain type inference when using the field lenses, but you can basically ignore that if you like, it's more of an implementation detail.

Here's how we can unify our two greeting functions using our awesome new typeclass:

```
greetByName :: HasName r String => r -> IO ()
greetByName r = putStrLn $ "Hello " <> r ^. name <> "!"
```

This definition says: given that some type `r` has a `name` field of type `String`, I can take a value of that type and run some IO.

With that, we can greet anything that has a name! We can even use this function on records defined in completely different modules or even different libraries, so long as they implement the `HasName` typeclass.

```
>>> greetByName (Person "Calvin")
Hello Calvin!

>>> greetByName (Pet "Hobbes")
Hello Hobbes!
```

Although `makeFields` may seem a bit like magic, it works by relying on typeclass instances. `makeFields` will look for the appropriate `Has*` class in scope, if it exists already it will just implement an instance. If it can't find an existing instance it will both define the class AND implement an instance for it. This means that if you have two isolated modules which each define `HasName` classes but neither of them imports each-other, that they'll inadvertently define two separate and incompatible versions of the `HasName` typeclass. This is something you'll generally want to avoid.

Do your best to have a canonical instance for each `Has*` class. If needed you can define a separate module just for that field which calls `makeFields` on a dummy record, then exports ONLY the field typeclass for other modules to import, we'll explore that idea later in this chapter.

This covers the basics of what classy lenses are and one of their possible uses, but they're also useful for limiting scope and improving code robustness. Let's see an example of that.

Separating logic and minimizing global knowledge

Monad transformer stacks are extremely common in production Haskell code, these stacks **almost always** include a `Reader` monad which provides some configuration or environment variables to the running code. It might include logging settings, hostnames, ports, database connection info, just about anything used by an application. A traditional Reader Monad must have exactly **one** environment for its entire computation, so typically the whole monad stack is stuck with a single environment type. Here's a cliff-notes version of what this could look like in a very simple app. I'll use MTL-style monad constraints for this example, feel free to google around a bit if you haven't seen that style before. Kindly pretend it does more than just print its config to the console.

```
data Env =
    Env { _portNumber :: Int
        , _hostName   :: String
        , _databaseUrl      :: String
        } deriving Show

makeLenses ''Env

connectDB :: (MonadIO m, MonadReader Env m) => m ()
connectDB = do
    url <- view databaseUrl
    liftIO $ putStrLn ("connecting to db at: " <> url)

initialize :: (MonadIO m, MonadReader Env m) => m ()
initialize = do
    port <- view portNumber
```

```
    host <- view hostName
    liftIO $ putStrLn ("initializing server at: " <> host <> ":" <> show port)

main :: IO ()
main = do
    flip runReaderT (Env 8000 "example.com" "db.example.com") $ do
        initialize
        connectDB
```

When we run it:

```
>>> main
initializing server at: example.com:8000
connecting to db at: db.example.com
```

Great!

So, in this setup we can notice a few things. The code is functional, and doesn't have any **problems** per se, but both `initialize` and `connectDB` depend directly on `Env`, and presumably so does **EVERY** action in our entire code-base that pulls config from the environment. This means that we've set a very **rigid dependency** on our `Env` type, if that type changes significantly we'll have to fix code across our entire app.

Another issue is that each of our actions depends on the entirety of our environment all the time. We don't have any isolation guarantees, and none of the type signatures for our actions clearly communicate their dependencies.

These 'global' dependencies can either be really convenient or terribly dangerous depending on your organization, size of app, and privacy/security requirements. It's up to you to decide whether this is capital B *"Bad"* or not.

Another annoyance appears when we eventually have multiple different contexts in which we'd like to re-use the action. What if we use the same shared data-base initialization action across multiple services in our organization? With this setup it's simply not possible unless ALL services use the same global `Env` state. This is rarely the case.

We can alter the environment before running certain actions to get finer granularity if we want, but honestly it gets pretty messy. For instance we could specialize the `connectDB` action to be a reader over only the `DatabaseUrl` if we like, then manually pass that part of the environment:

```
type DatabaseUrl = String
connectDB :: (MonadIO m, MonadReader DatabaseUrl m) => m ()
connectDB = do
    url <- ask
    liftIO $ putStrLn ("connecting to db at: " <> url)

main :: IO ()
main = do
    flip runReaderT (Env 8000 "example.com" "db.example.com") $ do
        initialize
        ask >>= \e -> runReaderT connectDB (_databaseUrl e)
```

Here we've changed the environment of `connectDB` to only the piece it needs (the DatabaseUrl), then we manually run the reader layer with the correct piece in `main`. This is messy, and also requires you to know your full monad-stack wherever you do this trick, which is a definite anti-pattern in MTL style. This clearly isn't a viable option.

The `magnify` combinator we learned about earlier can help a little with this situation:

```
main :: IO ()
main = do
    flip runReaderT (Env 8000 "example.com" "db.example.com") $ do
        initialize
        magnify databaseUrl connectDB
```

This is nice, but it's still a lot of noise in our code, and it gets very difficult when we need to depend on more than one piece of state, for instance our `initialize` action needs both the host *and* port! We still need to do better.

Granular dependencies with `makeFields`

Using `makeFields` cleans this up. We can define each field's class in some relevant module, then just implement those field typeclasses on our `Env` object, thus negating any need for a dependency between the two modules.

In the module which defines our Database types and behaviours we can add a dummy `DBFields` record which serves only to specify the fields which actions in the module need:

```haskell
{-# LANGUAGE MultiParamTypeClasses #-}
{-# LANGUAGE FunctionalDependencies #-}
{-# LANGUAGE TemplateHaskell #-}
{-# LANGUAGE RankNTypes #-}
module DB (HasDatabaseUrl(..), connectDB) where

import Control.Lens
import Control.Monad.Reader

type DatabaseUrl = String
data DbFields = DbFields {_dbFieldsDatabaseUrl :: DatabaseUrl}

makeFields ''DbFields

type DatabaseUrl = String
connectDB :: (MonadIO m, HasDatabaseUrl e DatabaseUrl, MonadReader e m) => m ()
connectDB = do
    url <- view databaseUrl
    liftIO $ putStrLn ("connecting to db at: " <> url)
```

Note carefully how we don't need to export `DbFields` or it's record accessors at all, it exists solely to define the fields needed by the module. I also want to draw attention to the fact that the field prefix `dbFields` must be spelled **exactly** like this or `makeFields` will ignore it. The prefix must be the exact name of the record, but with the first letter lowercased. This can be a bit annoying sometimes, so we can use `makeFieldsNoPrefix` to get around it, especially if we're not worried about record-accessor conflicts. We don't have to worry about conflicts here since we're not exporting the record itself anyways, so we can rewrite the data definition like this and get the same effect:

```haskell
data DbFields = DbFields {_databaseUrl :: String}

makeFieldsNoPrefix ''DbFields
```

Now, when we define our `Env` type we import each of the modules which have fields we want to define:

```haskell
{-# LANGUAGE MultiParamTypeClasses #-}
{-# LANGUAGE RankNTypes #-}
{-# LANGUAGE TemplateHaskell #-}
{-# LANGUAGE FunctionalDependencies #-}
module Main where

import Control.Lens
import DB

data Env =
    Env { _envPortNumber  :: Int
        , _envHostName    :: String
        , _envDatabaseUrl :: String
        } deriving Show

makeFields ''Env
```

We still have cross-dependencies of course, but they've been inverted! The new dependency structure is much more modular, extensible, and allows us to fix the issues we discovered earlier. For example; now our `DB` module can be shared amongst any number of services which simply have to add the appropriate `makeFields` calls on their `Env` types in order to use it.

We DO however have to import the `DB` module just to get the instances in scope, otherwise `makeFields` will define *new* classes for each of its fields. This is a gotcha we need to be wary of, but take heart that our service would eventually fail to compile and alert us about the problem when we try to run our `connectDB` action inside a `MonadReader` whose `Env` was missing the required instances.

Field requirements compose

A benefit of field typeclasses is that they're specified with **constraints**, and constraints compose! This means that if we want to accept a record with multiple specific fields we can do so easily by just adding multiple `Has*` constraints. Here's what it looks like if we give our `initialize` action the `fields` treatment:

```
{-# LANGUAGE MultiParamTypeClasses #-}
{-# LANGUAGE TemplateHaskell #-}
{-# LANGUAGE FunctionalDependencies #-}
module Init (HasHostName(..), HasPortNumber(..)) where

import Control.Lens
import Control.Monad.Reader

data InitFields =
    InitFields { _hostName   :: String
               , _portNumber :: Int
               }

makeFieldsNoPrefix ''InitFields


initialize :: ( MonadIO m
              , HasHostName e String
              , HasPortNumber e Int
              , MonadReader e m
              )
           => m ()
initialize = do
    port <- view portNumber
    host <- view hostName
    liftIO $ putStrLn ("initializing server at: " <> host <> ":" <> show port)
```

Our `initialize` action simply specifies it requires BOTH fields on the environment and everything works out.

14.2 `makeFields` vs `makeClassy`

There are a half-dozen different ways to generate lenses using TemplateHaskell so you'd certainly be forgiven for getting them confused. This chapter covers the distinction between `makeFields` and `makeClassy`.

Let's jump right in and define a record to see what each version generates for us:

```
module People where
-- ...
data Person = Person{ _name :: String
                    , _favouriteFood :: String
                    } deriving Show
```

If we add `makeFieldsNoPrefix ''Person` we get:

```
>>> :browse People
data Person = Person {_name :: String, _favouriteFood :: String}

class HasName s a | s -> a where
  name :: Lens' s a
  {-# MINIMAL name #-}

class HasFavouriteFood s a | s -> a where
  favouriteFood :: Lens' s a
  {-# MINIMAL favouriteFood #-}
```

So `makeFieldsNoPrefix` has generated a class for each field as we've seen before.

If we just ADD `makeClassy` we get conflicting typeclass method names, so we need to delete `makeFields` before adding `makeClassy`. Here's another `:browse People` to see what changed:

```
>>> :browse People
data Person = Person {_name :: String, _favouriteFood :: String}

class HasPerson c where
  person :: Lens' c Person
  favouriteFood :: Lens' c String
  name :: Lens' c String
  {-# MINIMAL person #-}
```

Interesting! So `makeClassy` has generated only a **single** class for the whole record, whereas `makeFields` makes classes for **each** field.

The key difference between these styles is that `makeClassy` is a bit less granular, it effectively allows you to specify that a type has a `Person` nested within it somewhere. When (manually) implementing this typeclass for larger types we need only specify a single lens to access the `Person` object and we can then use each of that object's field lenses directly.

Let's re-factor our Database Config example from earlier to use `makeClassy` instead so we can see how it differs.

```
module DBClassy (DbConfig(..), HasDbConfig(..), connectDB) where

import Control.Lens
import Control.Monad.Reader

data DbConfig =
    DbConfig { _databaseUrl    :: String
             , _maxConnections :: Int
             } deriving Show

makeClassy ''DbConfig

connectDB :: (MonadIO m, HasDbConfig e, MonadReader e m) => m ()
connectDB = do
    url <- view databaseUrl
    numConnections <- view maxConnections
    liftIO
      $ putStrLn ("connecting to db at: "
                  <> url
                  <> " with max connections: "
                  <> show numConnections)
```

I added an extra field `_maxConnections` just to make it a bit more interesting this time. We see a few differences right away, the action now only needs to specify `HasDbConfig` making no mention of specific fields, but we can still use the individual field lenses within the action itself. The constraints are definitely a bit more concise as well!

Another difference is that in this version we **export** the `DBConfig` alongside the typeclass since anyone implementing the typeclass will presumably need to have a `DBConfig` nested in their object somewhere. Here's how it looks when we integrate this into our `Env`:

```
{-# LANGUAGE MultiParamTypeClasses #-}
{-# LANGUAGE RankNTypes #-}
{-# LANGUAGE TemplateHaskell #-}
{-# LANGUAGE FunctionalDependencies #-}
module EnvClassy where

import Control.Lens
import Control.Monad.Reader
import DBClassy

data Env =
    Env { _envDbConfig :: DbConfig
        } deriving Show
```

Using the classy approach we embed the entire `DbConfig` type into our record rather than each of the individual fields. If we try to add the standard `makeFields` call we actually get an error!

```
    • Expected kind ‘* -> Constraint’,
        but ‘HasDbConfig Env’ has kind ‘Constraint’
    • In the instance declaration for ‘HasDbConfig Env DbConfig’
   |
16 | makeFields ''Env
```

This is a little weird, but we can track it down. Basically, it's trying to implement an instance `HasDbConfig` as a **field** typeclass, but it's a **classy** typeclass instead! These have different constraints and are generally incompatible, so typically you don't mix and match `makeFields` with `makeClassy`, they dont always play well together (although it's possible if you're careful).

Instead of using `makeFields` or `makeClassy`, we can go back to our roots with `makeLenses`:

```
data Env =
    Env { _envDbConfig :: DbConfig
        } deriving Show

makeLenses ''Env


>>> :browse EnvClassy
data Env = Env {_envDbConfig :: DbConfig}
envDbConfig :: Iso' Env DbConfig
```

Okay, we have a normal ol' lens for `envDbConfig`, now for the secret sauce! We can write an instance for `HasDbConfig` which specifies where in our record to find the `DbConfig`:

```
instance HasDbConfig Env where
  dbConfig = envDbConfig
```

Here we're just "delegating" to the lens we generated with `makeLenses`. By implementing this single lens we can now use each of the `HasDbConfig` field lenses directly on an `Env`! Observe:

```
>>> let env = Env (DbConfig "db.example.com" 100)

-- The lenses know how to dig into the DbConfig
>>> env ^. databaseUrl
"db.example.com"

>>> env ^. maxConnections
100

-- We can also get the whole config if we like:
>>> env ^. dbConfig
DbConfig {_databaseUrl = "db.example.com", _maxConnections = 100}
```

As you can see, each approach lends itself to different styles. I'd say `makeFields` is more helpful for reducing the annoyances of records with shared names and for implementing field-polymorphic functions, whereas `makeClassy` tends to scale a little better for large projects. `makeClassy` is less granular, but also requires significantly fewer constraints. The best advice I can give is to try both on a small project and see which suits your needs better.

15. JSON

Working with JSON values is an extremely common task in modern web applications. In most cases it's recommended to deserialize your JSON values into proper Haskell records and to use optics to work with those instead of unstructured JSON since it provides better type safety and tends to be easier to read. However, the web is basically the wild wild west and JSON data is often very loosely structured. Perhaps certain objects are missing keys, or maybe you're more interested in performing loosely structured queries over the JSON you've got on hand. As it turns out, optics are **fantastically** well-suited to this type of task!

This chapter uses the popular `aeson`[1] library for working with JSON values. If you're completely unfamiliar with it, it wouldn't hurt to skim a tutorial[2] on it before continuing.

There are many helpers for working with JSON values within the `aeson` library itself, however for interop with optics we'll also need the `lens-aeson` library (note that this is distinct from the `aeson-lens` library which isn't recommended). The following examples assume you've included the `aeson`, `lens-aeson`, and `raw-strings-qq` packages in your project.

```
{-# LANGUAGE TemplateHaskell #-}
{-# LANGUAGE QuasiQuotes #-}
{-# LANGUAGE OverloadedStrings #-}
module JSON where

import Control.Lens
import Data.Aeson
import Data.Aeson.Lens
import Text.RawString.QQ (r)
```

15.1 Introspecting JSON

JSON values are effectively untyped, or more accurately they're all part of one BIG type; `aeson` calls this JSON catch-all type `Value`. When we have a `Value` we know it's *valid* JSON, but we don't know what sort of JSON it is. It could be a number, string, array, object, boolean, or even null! Even if we're pretty confident that a particular `Value` is an array we should still handle the possibility that the value was actually something else. If we want to explore our unknown JSON value we need to "pattern match" on things as we dive deeper. If you've been following along with the book so far hopefully you're thinking Prisms!

[1] http://hackage.haskell.org/package/aeson
[2] https://artyom.me/aeson

`lens-aeson` provides prisms for all the major JSON subtypes. These prisms are abstracted over their input type in such a way that you can use them directly on any sort of value which *might* contain JSON (e.g. a `Value`, `ByteString`, `String`, or `Text`). This can be really convenient as it allows us to avoid a separate parsing step, we can just dive straight in! Failure to parse a `ByteString` or `Text` simply means the Prism won't match anything. If you want clearer errors you may want to parse it explicitly anyways.

Here's a list of prisms provided to us, we'll represent a "json-like" type as `t`:

- `_String :: Prism' t Text`
- `_Bool :: Prism' t Bool`
- `_Null :: Prism' t ()`
- `_Object :: Prism' t (HashMap Text Value)`
- `_Array :: Prism' t (Vector Value)`

There are also several numeric prisms which automatically convert the generic JSON number type into whichever you'd like to work with:

- `_Number :: Prism' t Scientific`
- `_Double :: Prism' t Double`
- `_Integer :: Prism' t Integer`
- `_Integral :: Integral n => Prism' t n`

Remember that we can view through a prism using `^?` or `preview` and we'll get a result out wrapped in a `Maybe` to encode failures. Let's try it out on a few different JSON values:

```
-- The prisms will automatically parse values from strings into JSON
>>> "42" ^? _Double
Just 42.0 :: Maybe Double

>>> "42" ^? _String
Nothing :: Maybe Text

-- Failure to parse a value fails the prism traversal
>>> "{invalid JSON}" ^? _String
Nothing :: Maybe Text

>>> "\"Hello, World!\"" ^? _String
Just "Hello, World!" :: Maybe Text

-- If we mismatch the expected type it will fail
-- even if the JSON is a well formed value of a different type
>>> "\"Hello, World!\"" ^? _Double
Nothing :: Maybe Double
```

Each prism will short-circuit the entire path if it fails, this makes composition really nice and easy! Unfortunately it can also make it a bit tricky to find where our assumptions about the JSON's shape were incorrect.

Let's try digging into some of the more structured JSON types. I'll use the `r` Quasi-Quoter to avoid the need to escape every set of quotes we write.

```
jsonObject :: String
jsonObject = [r|
{
  "name": "Jack Sparrow",
  "rank": "Captain"
}
|]

jsonArray :: String
jsonArray = [r|
[
  "North",
  "East",
  "South",
  "West"
]
|]
```

```
>>> jsonObject ^? _Object
Just (fromList [("name",String "Jack Sparrow"),("rank",String "Captain")])
  :: Maybe (HashMap Text Value)

>>> jsonArray ^? _Array
Just [String "North",String "East",String "South",String "West"]
  :: Maybe (Vector Value)
```

We can notice a few things from these last two; firstly that `_Object` and `_Array` are parsing objects and arrays as expected, secondly that they return some types which may seem a bit unorthodox at first. `_Object` returns a `HashMap` with `Text` keys and values which are the Aeson `Value` type; `_Array` returns a `Vector` of `Value` objects. This can be a bit annoying if you're used to working with more common containers like `Data.Map` or lists respectively. The reason it returns these specific types is that these are the types which are used under the hood inside the Aeson library and exposing different container types would impose a performance penalty. Don't worry though, by introducing a few more helpers we'll realize that we rarely need to deal with the `HashMaps` or `Vectors` directly.

15.2 Diving deeper into JSON structures

Let's dive into a more complex JSON structure:

```
blackPearl :: String
blackPearl = [r|
{
  "name": "Black Pearl",
  "crew": [
    {
      "name": "Jack Sparrow",
      "rank": "Captain"
    },
    {
      "name": "Will Turner",
      "rank": "First Mate"
    }
  ]
}
|]
```

Let's say we want to get the first crew member of the ship from its array of crew members. We *could* get there by unpacking the top-level object into a HashMap, find the "crew" field using `ix`, unpack *that* into a Vector, then grab the first element of that then keep digging:

```
>>> blackPearl
      ^? _Object
      . ix "crew"
      . _Array
      . ix 0
      . _Object
      . ix "name"
      . _String
Just "Jack Sparrow"
```

This *works*, but it's long, hard to read, and it's **easy to introduce mistakes**.

`lens-aeson` provides helpers which combine these operations for us:

```
key :: Text -> Traversal' t Value
nth :: Int  -> Traversal' t Value
```

These helpers are defined like this:

```
key :: Text -> Traversal' t Value
key k = _Object . ix k

nth :: Int  -> Traversal' t Value
nth i = _Array . ix i
```

Using these new tricks we can clean up our query into something much more readable.

```
>>> blackPearl ^? key "crew" . nth 0 . key "name" . _String
Just "Jack Sparrow"
```

Note that `key` and `nth` are traversals which use `ix`; so they **can't insert new values** unless there's already a value at the given key. To insert values into a JSON Object you can match on the Object then use `at` like you would with a Map.

```
>>> myJSON & _Object . at "newKey" ?~ newValue
```

Tracking down mistakes

Since we're mostly working with values of the ambiguous type `Value` it's quite common to make mistakes in our lens path. When writing traversals like this I recommend loading in an example value into ghci and building up the path one section at a time, checking intermediate results as you go to verify that your assumptions continue to hold.

15.3 Traversing into multiple JSON substructures

Sometimes we need to collect or modify values from many spots within within a JSON structure, maybe we have a list of users and want to collect all of their email addresses, perhaps we've got a large kubernetes configuration and we want to find the specs for all pods with a given tag. Using a traversal we could even make an API call to replace an ID inside our JSON with the object that ID represents! The possibilities are nearly endless.

Traversing Arrays

One of the most common branching queries is when we need to collect a piece of information from every element of an array, there's a traversal for that!

`values` allows us to traverse all the values in a `Value` if it's an Array:

```
values :: AsValue t => IndexedTraversal' Int t Value
values = _Array . traversed
```

Here's a very simple example where we collect the numeric contents of a list:

```
>>> "[1, 2, 3]" ^.. values . _Integer
[1, 2, 3]
```

Remember that if any elements of the traversal don't match our assumptions they'll be silently excluded. Unlike Haskell, JSON allows arrays with differing value types.

```
>>> "[1, null, 2, \"Hi mom!\", 3]" ^.. values . _Integer
[1, 2, 3]
```

Note that it's actually the `_Integer` prism which is filtering out possibilities here, `values` focuses every array element, but `_Integer` pattern matches on only the numbers.

`values` is an `IndexedTraversal' Int` so we have access to the array index even after we've descended into each element. Here we return the index alongside each collected element using the indexed "itoListOf" operator (`^@..`).

```
>>> "[\"a\", \"b\", \"c\"]" ^@.. values . _String
[(0,"a"),(1,"b"),(2,"c")]

-- It includes the correct array index even if we've filtered out values
>>> "[\"a\", 1, \"b\", null, \"c\", []]" ^@.. values . _String
[(0,"a"),(2,"b"),(4,"c")]
```

Remember that as long as we're chaining together `Traversals` and `Prisms` we can modify our focus:

```
>>> import Data.Text (toUpper)
>>> "[\"a\", \"b\", \"c\"]" & values . _String %~ toUpper
"[\"A\",\"B\",\"C\"]"
```

The `values` traversal will even return the result to its original encoding; in this case a `String`!

Let's try out these techniques with a more complex example.

Here's a sufficiently complex JSON object for us to play with:

```
fleet :: String
fleet = [r|
[
  {
    "name": "Black Pearl",
    "crew": [
      {
        "name": "Jack Sparrow",
        "rank": "Captain"
      },
      {
        "name": "Will Turner",
        "rank": "First Mate"
      }
    ]
  },
  {
    "name": "Flying Dutchman",
    "crew": [
      {
        "name": "Davy Jones",
        "rank": "Captain"
      },
      {
        "name": "Bootstrap Bill",
        "rank": "First Mate"
      }
    ]
  }
]
|]
```

Let's say we want to collect the names of every ship in our fleet. This is a branching query where we want to collect a specific piece of info from each ship in the top-level array.

A helpful technique when trying to determine how to 'lensify' a JSON query task like this is to think of what the traditional JSON path would look like in Javascript notation:

```
fleet[i].name
```

It's not so complicated, now we need to convert it to optics!

The first step in our path has us indexing into an array; our example chooses index `i`, but we really want to go into **ALL** indices of the array. As we've learned, this is exactly what `values` is for! We'll use `(^..)` to get the list of all values we've focused.

```
>>> fleet ^.. values
<all the ships>
```

Okay, so next in our path is `.name`, the dot implies object access, and `name` is the key of that object, so we can use the `key` helper we looked at earlier. We also know that the name should be a String, and probably just want to ignore it if it's some other type or is missing entirely, so we can tack on a `_String` to make that assertion. Leaving us with the following query:

```
>>> fleet ^.. values . key "name" . _String
["Black Pearl","Flying Dutchman"]
```

Let's go even deeper and get the name of every crew member in our entire fleet!

```
>>> fleet ^.. values . key "crew" . values . key "name" . _String
[ "Jack Sparrow"
, "Will Turner"
, "Davy Jones"
, "Bootstrap Bill"
]
```

If we remember some clever tricks from the section on indexed lenses we can even include the ship each member is from alongside their name, admittedly it begins to look a bit less elegant, but think about the work that would be involved to do this in long-form:

```
>>> fleet
     ^@.. values
     . reindexed (view (key "name" . _String)) selfIndex
     <. (key "crew" . values . key "name" . _String)
[ ("Black Pearl"    , "Jack Sparrow")
, ("Black Pearl"    , "Will Turner")
, ("Flying Dutchman", "Davy Jones")
, ("Flying Dutchman", "Bootstrap Bill")
]
```

Traversing Objects

We've learned how to simultaneously focus all values within a JSON array, but what about objects? Sometimes JSON objects are structured in such a way that we want to access all values at once; for instance if we had a map from user-id to user and wanted to focus every user at once. For this we have `members`!

```
members :: AsValue t => IndexedTraversal' Text t Value
```

Just like `values` focuses each array element as a `Value`, `members` focuses each `Value` in the Object. `members` is an `IndexedTraversal`, so it keeps track of each Object's key in the index as `Text`. Enough chatter, let's try it out!

Our enterprising pirates could never quite understand the merits of spreadsheets, so they've foolishly decided to use JSON as their primary inventory system. They're keeping track of their cargo as a JSON object mapping between item-type and quantity:

```
cargo :: String
cargo = [r|
{
  "emeralds": 327,
  "rubies": 480,
  "sapphires": 621,
  "opals": 92,
  "dubloons": 34
}
|]
```

Let's start off simple and see if we can list all the quantities, and maybe even take the count of all the items on board.

```
>>> cargo ^.. members . _Integer
[34,327,92,621,480]

>>> sumOf (members . _Integer) cargo
1554
```

It seems to be working!

Knowing all the quantities isn't so helpful unless we know **what** it is that we're counting, so how can we get the **key** alongside the **quantity**? Sounds like a job for indexed optics!

Let's start off by getting the item name alongside the quantity in our query:

```
>>> cargo ^@.. members . _Integer
[ ("dubloons" , 34)
, ("emeralds" , 327)
, ("opals"    , 92)
, ("sapphires", 621)
, ("rubies"   , 480)
]
```

This properly parses the input into an Object, validates and parses all the values as integers, then includes the keys as `Text` alongside each value. Luckily `_Integer` is an index-preserving optic so we don't need to muck about with `(<.)` to carry our indexes to the right spot.

Let's say we want to know which item we have the most of:

```
>>> import Data.Ord (comparing)
>>> maximumByOf (members . _Integer . withIndex) (comparing snd) cargo
Just ("sapphires" , 621)
```

Unfortunately there's no `imaximumOf` so we'll need to explicitly include the index into the fold using `withIndex`, but it's not too much extra work. We then compare only on the second element of the tuple.

15.4 Filtering JSON Queries

We're already collecting info from across our huge JSON blob like a pro; but oftentimes we need to ask more complex questions. What if we only want the IDs of users whose accounts have expired? Let's try to find only the crew members in our fleet who are Captains.

The trick with this sort of query is that we may be filtering on a different aspect than we return from the query. In this case we only want to focus the name of the crew member, but we're filtering on the crew member's rank. This means we need to run our filter in the middle of our path before we've lost access to the information we need.

We've already learned about how `filtered` allows us to run predicates in the midst of a fold to thin out our results. We can put this to great effect when working with JSON!

Let's try a few more complex queries over our fleet.

```
fleet :: String
fleet = [r|
[
  {
    "name": "Black Pearl",
    "crew": [
      {
        "name": "Jack Sparrow",
        "rank": "Captain"
      },
      {
        "name": "Will Turner",
        "rank": "First Mate"
      }
    ]
  },
  {
    "name": "Flying Dutchman",
    "crew": [
      {
        "name": "Davy Jones",
        "rank": "Captain"
      },
      {
        "name": "Bootstrap Bill",
        "rank": "First Mate"
      }
    ]
  }
]
|]
```

Let's track down all crew members who are captains. We'll use the nifty `filteredBy` combinator we learned in the traversal chapters. Remember that you'll need an up-to-date version of `lens` to use `filteredBy`, it's quite new.

```
>>> fleet
      ^.. values
      . key "crew"
      . values
      . filteredBy (key "rank" . only "Captain")
      . key "name"
      . _String
[ "Jack Sparrow"
, "Davy Jones"
]
```

If you don't have `filteredBy` available to you, you can get by using `has` with `filtered` instead:

```
filtered (has (key "rank" . only "Captain"))
```

You probably didn't even notice, but we're actually relying on the `FromString` instance of `Value` here. `OverloadedStrings` allows our `"Captain"` string to be changed into a JSON string `Value` which is compared to the rank's `Value`. If that's too much magic for you we can be a bit more explicit:

```
filtered (has (key "rank" . _String . only "Captain"))
```

We can use `filtered` and `filteredBy` at any location within a path to trim down the values we're focusing. This allows us to select all the values we want from within a large JSON object with laser-like precision!

15.5 Serializing & Deserializing within an optics path

So far we've seen how we can use the prisms provided by `lens-aeson` to deserialize the standard JSON types like objects, arrays, numbers and strings, but what about more complex types? In Haskell we would typically prefer to work with strongly typed records rather than nebulously typed `Value` objects. Let's see how we can reflect values from their `Value` representations into their structured Haskell forms and **back** within a traversal.

In addition to the standard imports you'll also want the following extensions:

```
{-# LANGUAGE DeriveAnyClass #-}
{-# LANGUAGE DerivingStrategies #-}
{-# LANGUAGE DeriveGeneric #-}
```

As well as this import:

```
import GHC.Generics
```

With these in scope we can define a record type and nicely derive our ToJSON and FromJSON instances. In a production application you'll want to define these manually, but this is fine for a quick example.

```
data Creature =
    Creature
        { creatureName :: String
        , creatureSize :: Double
        } deriving stock (Generic, Show)
          deriving anyclass (ToJSON, FromJSON)
```

We've received the following JSON missive of creature sightings via carrier dolphin. It carefully details the sightings of massive creatures across the seven seas. The data specifies the location of the sighting, the name of the creature, and the estimated length of the creature (in number of peglegs placed end-to-end).

```
creatureSightings :: String
creatureSightings = [r|
{
  "Arctic": [
    {
      "creatureName": "Giant Squid",
      "creatureSize": 45.2
    }
  ],
  "Pacific": [
    {
      "creatureName": "Kraken",
      "creatureSize": 124.4
    },
    {
      "creatureName": "Ogopogo",
      "creatureSize": 34.6
    }
  ]
}
|]
```

Blackbeard is writing an old pirate shanty and wants a list of all the creatures from around the world, here's how we might retrieve such a list:

```
>>> creatureSightings  ^.. members . values . _JSON :: [Creature]
[ Creature
    { creatureName = "Kraken"
    , creatureSize = 124.4
    }
, Creature
    { creatureName = "Ogopogo"
    , creatureSize = 34.6
    }
, Creature
    { creatureName = "Giant Squid"
    , creatureSize = 45.2
    }
]
```

Notice that we've returned properly structured `Creature` objects rather than JSON objects!

This introduces the `_JSON` prism. It's a prism which can convert between any JSON-like `t` and any type `a` which can be serialized to or from a JSON value:

```
_JSON :: (FromJSON a, ToJSON a) => Prism' t a
```

Note that in future releases of `lens-aeson` the type of `_JSON` will be generalized further, the simple version we use here will be renamed to `_JSON'`. Try `_JSON` first, if it doesn't work in some examples try switching to `_JSON'`

Since the result type of `_JSON` is polymorphic over anything that implements `ToJSON` and `FromJSON` we'll usually need to use type annotations to tell it which type we expect it to deserialize to. In the previous code snippet I simply added an annotation to the result of the operation; but as we'll see in the following examples you may wish to enable the `TypeApplications` language pragma so that we can provide annotations inline.

While we're at it, let's see how `_JSON` allows us to 'write back' changes we make to the record representation into the original JSON blob. When defining our `Creature` type we didn't prefix our field names with the standard underscores because it would mess with our derived JSON instances, but we'll still want some lenses for interacting with our `Creature` type, so let's derive some anyways using `makeLensesFor`. This allows us to specify the names of our field lenses manually. Add the following underneath your definition of `Creature`

```
--                field name        lens name
makeLensesFor [ ("creatureName", "creatureNameL")
              , ("creatureSize", "creatureSizeL")
              ] ''Creature
```

We've specified to create lenses for each field by making a mapping of field name to the lens name we want. Now we have lenses for each field suffixed with an `L`.

Let's see how our traversals allow us to make edits to structured data and have that reflect back into the JSON representation.

This year's standard peg-leg standard sizes have changed from last year; to correct our information we'll need to multiply the creature size by `1.2` wooden peg legs (I guess pirates are getting a bit shorter these days). Let's give it a go:

```
>>> putStrLn $ creatureSightings
                & members
                . values
                . _JSON @_ @Creature
                . creatureSizeL
                *~ 1.2
{
  "Pacific": [
    {
      "creatureName": "Kraken",
      "creatureSize": 149.28
    },
    {
      "creatureName": "Ogopogo",
      "creatureSize": 41.52
    }
  ],
  "Arctic": [
    {
      "creatureName": "Giant Squid",
      "creatureSize": 54.24
    }
  ]
}
```

It's a bit tough to tell, but if you reference back to the previous figures you'll see we have indeed increased the creature size as required! `_JSON` allowed us to edit the creatures as records, but converted the result back into a `Value` to embed back into a JSON string when all was said and done!

You'll see that this time we specified an annotation for `_JSON` inline using `TypeApplications`; the `@_` says that we don't really care what we're converting **from** (although in this case it's simply a `Value`), the second annotation states that we want it to deserialize into a `Creature`. In this particular case GHC could infer the `Creature` type due to the use of the `creatureL` lens, but it never hurts to add a bit more clarity, this will typically improve type errors if something goes wrong.

Now it's time to put all our new JSON skills to the test!

15.6 Exercises: Kubernetes API

Jump to answers

For the exercises in this section I'll give you some realistic JSON output from the kubernetes pod description API. Even though I've simplified it significantly it's still a lot to navigate! Although it's probably easier to "eyeball" most of the answers, the exercise is to write an expression which would work on an arbitrarily complex JSON payload.

Although some of these expressions seem ridiculously long, consider the work it would be to do the same thing in the "long" way, unpacking and re-packing the entire JSON object as you go.

Ready for a cornucopia of excessively nested JSON? Here we go!

You can find this data as a gist here[3] if you want an easy way to copy-paste it into your project.

```
pods :: BS.ByteString
pods = [r|
{
    "kind": "List",
    "apiVersion": "v1",
    "items": [
        {
            "kind": "Pod",
            "apiVersion": "v1",
            "metadata": {
                "name": "redis-h315w",
                "creationTimestamp": "2019-03-23T19:42:21Z",
                "labels": {
                    "name": "redis",
                    "region": "usa"
                }
            },
            "spec": {
                "containers": [
```

[3] https://gist.github.com/ChrisPenner/53e4b505ff0673b39c60e6c926b2715d

```
                {
                    "name": "redis",
                    "image": "redis",
                    "ports": [
                        {
                            "name": "redis",
                            "hostPort": 27017,
                            "containerPort": 27017,
                            "protocol": "TCP"
                        }
                    ],
                    "resources": {
                        "requests": {
                            "cpu": "100m"
                        }
                    }
                }
            ]
        }
    },
    {
        "kind": "Pod",
        "apiVersion": "v1",
        "metadata": {
            "name": "web-4c5bj",
            "creationTimestamp": "2019-02-24T20:23:56Z",
            "labels": {
                "name": "web",
                "region": "usa"
            }
        },
        "spec": {
            "containers": [
                {
                    "name": "web",
                    "image": "server",
                    "ports": [
                        {
                            "name": "http-server",
                            "containerPort": 3000,
                            "protocol": "TCP"
                        }
                    ],
```

```
                        "resources": {
                            "requests": {
                                "cpu": "100m"
                            }
                        }
                    }
                ]
            }
        }
    ]
}
|]
```

1. Your first task should be mostly straightforward: get the api version which was used to make the call.

```
>>> pods ^? key "apiVersion" . _String
Just "v1"
```

2. Next, count the **number** of all **containers** across all **pods**. You can assume that every element of "items" is a pod.

```
>>> lengthOf (key "items" . values . key "spec" . key "containers" . values) pods
2
```

3. Return the "name" (as `Text`) of all **containers** which have the same value for their "image" and "name" fields.

```
>>> toListOf (key "items" . values . key "spec" . key "containers" . values . filter\
ed (\v -> v ^? key "name" == v ^? key "image") . key "name" . _String) pods
[ "redis" ]
```

4. Collect a list of all "containerPort"s alongside their Pod's "metadata > name".

```
>>> toListOf ((key "items" . values . reindexed (view (key "metadata" . key "name" .\
 _String)) selfIndex <. (key "spec" . key "containers" . values . key "ports" . valu\
es . key "containerPort" . _Integer)) . withIndex) pods
[
    ( "redis-h315w"
    , 27017
    )
,
    ( "web-4c5bj"
    , 3000
    )
]
```

5. Uppercase the label values inside each pod's metadata

```
>>> import qualified Data.Text as T
>>> pods & key "items" . values . key "metadata" . key "labels" . members . _String \
%~ T.toUpper
<this is way too big to print out>
```

6. Set a resource request of `memory: "256M"` for every container.

```
>>> pods & key "items" . values . key "spec" . key "containers" . values . key "reso\
urces" . key "requests" . _Object . at "memory" ?~ "256M"
<this is way too big to print out>
```

BONUS Questions

These problems require a bit more thought, and will probably require some unorthodox uses of indexes and filtering. Good luck! Don't get too frustrated, they're deliberately very hard!

7. Get a **Set** of all metadata label **keys** used in the response.

```
>>> foldMapOf (key "items" . values . key "metadata" . key "labels" . members . asIn\
dex) S.singleton pods
fromList
    [ "name"
    , "region"
    ]
```

8. **Set** the `hostPort` to `8080` on any "port" descriptions where it is **unset**.

```
>>> pods & key "items" . values . key "spec" . key "containers" . values . key "port\
s" . values . _Object . at "hostPort" . filteredBy _Nothing ?~ _Number # 8080
```

9. **Prepend** the **region** to the **name** of each container.

```
>>> pods & key "items" . values . reindexed (view (key "metadata" . key "labels" . k\
ey "region" . _String)) selfIndex <. (key "spec" . key "containers" . values . key "\
name"
```

Jump to answers

16. Appendices

16.1 Optic Composition Table

This table is adapted from the documentation of the Scala optics library **Monocle**[1]. I've simply altered it to match the `lens` library.

The value of each cell denotes the most general type you can achieve by composing the column header with the row header.

The type of an optic is determined by collecting all the constraints of all composed optics in a path. Since constraints collection acts as a set union (which is commutative) the order of composition has no effect on the resulting optic type. Therefore the following table is symmetric across its diagonal.

"–" signifies that the optics are incompatible and do not compose.

	Fold	**Getter**	**Setter**	**Traversal**	**Prism**	**Lens**	**Iso**
Fold	Fold	Fold	–	Fold	Fold	Fold	Fold
Getter	Fold	Getter	–	Fold	Fold	Getter	Getter
Setter	–	–	Setter	Setter	Setter	Setter	Setter
Traversal	Fold	Fold	Setter	Traversal	Traversal	Traversal	Traversal
Prism	Fold	Fold	Setter	Traversal	Prism	Traversal	Prism
Lens	Fold	Getter	Setter	Traversal	Traversal	Lens	Lens
Iso	Fold	Getter	Setter	Traversal	Prism	Lens	Iso

For example, to determine which type we get by composing `traverse` with `_Just` we first check each of their types to discover that `traverse` is a `Traversal` and `_Just` is a `Prism`. We then look up the column (or row) with the header `Traversal`, then find the cell with the corresponding header `Prism` on the other axis. Performing this look up we see that the composition `traversed . _Just` results in a `Traversal`.

16.2 Optic Compatibility Chart

The following chart details which optics are valid substitutions for one another.

As an example, let's say we were curious if all **Prism**s are a valid **Traversal**; we first find the **row** with **Prism** in the first column; then find the corresponding **Traversal** column and find a `Yes`; meaning that a Prism **is** a valid substitution for a Traversal.

[1] https://julien-truffaut.github.io/Monocle/optics.html

	Fold	Getter	Setter	Traversal	Lens	Review	Prism	Iso
Fold	Yes	No	No	No	No	No	No	No
Getter	Yes	Yes	No	No	No	No	No	No
Setter	No	No	Yes	No	No	No	No	No
Traversal	Yes	Yes	Yes	Yes	No	No	No	No
Lens	Yes	Yes	Yes	Yes	Yes	No	No	No
Review	No	No	No	No	No	Yes	No	No
Prism	Yes	No	Yes	Yes	No	Yes	Yes	No
Iso	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes

16.3 Operator Cheat Sheet

Operators may look like an earthquake hit the mechanical keyboard factory, but there's actually a bit of a language to the whole thing which starts to make sense after a bit of practice.

Once you get used to the ideas you can usually *guess* the name of a symbol which does what you need, and it'll usually exist!

Legend for Getters

Symbol	Description
`^`	Denotes a **Getter**
`@`	Include the **index** with the result
`.`	Get a **single** value
`..`	Get a **List of** values
`?`	**Maybe** get the first value
`!`	**Force** a result or throw an exception if missing

Examples

`(^@..) :: s -> IndexedFold i s a -> [(i, a)]`

A **getter** (`^`) which **includes the index** (`@`) in a **list of all focuses** (`..`).

```
"Yarrr" ^@.. folded
[(0,'Y'),(1,'a'),(2,'r'),(3,'r'),(4,'r')]
```

`(^?!) :: s -> Traversal' s a -> a`

A **getter** (`^`) which **forcibly gets** (`!`) a **possibly missing** (`?`) value.

```
>>> Just "Nemo" ^?! _Just
"Nemo"
>>> Nothing ^?! _Just
*** Exception: (^?!): empty Fold
CallStack (from HasCallStack):
  error, called at src/Control/Lens/Fold.hs:1285:28 in lens
E:Control.Lens.Fold
  ^?!, called at <interactive>:1:1 in interactive:Ghci4
```

Legend for Setters/Modifiers

Symbol	Description
.	**Set** the focus
%	**Mod**ify the focus
~	Denotes a **Setter**/**Modifier**
=	Denotes a **Setter**/**Modifier** over a `MonadState` context
<	Include the **altered** focus with the result
<<	Include the **unaltered** focus with the result
%%	Perform a **traversal** over the focus
<>	`mappend` over the focus
?	Wrap in `Just` before setting
+	**Add** to the focus
-	**Subtract** from the focus
*	**Multiply** the focus
//	**Divide** the focus
||	Logically **or** the focus
&&	Logically **and** the focus
@	Pass the **index** to the modification function

Examples

`(<>~) :: Monoid a => Traversal' s a -> a -> s -> s`

A **setter** (~) which `mappends` (<>) a new value to the focus.

```
>>> ["Polly want a", "Salty as a soup"] & traverse <>~ " cracker!"
["Polly want a cracker!","Salty as a soup cracker!"]
```

`(<<%@=) :: MonadState s m => Traversal s s a b -> (i -> a -> b) -> m a`

Modify (%) the focus from within a `MonadState` (=), passing the **index** (@) to the function as well. Also return **unaltered** (<<) original value.

This one's a bit tricky:

```
>>> runState (itraversed <<%@= \k v -> "item: " <> k <> ", quantity: " <> v) (M.sing\
leton "bananas" "32")
("32",fromList [("bananas","item: bananas, quantity: 32")])
```

(%%~) :: Traversal s t a b -> (a -> f b) -> s -> f t

A **setter** (~) which **traverses** (%%) a function over the focus.

```
>>> (1, 2) & both %%~ (\n -> if even n then Just n else Nothing)
Nothing

>>> (2, 4) & both %%~ (\n -> if even n then Just n else Nothing)
Just (2,4)
```

16.4 Optic Ingredients

When reading type-errors or strange optics signatures you can usually guess the type of optic an operation needs by matching the constraints against the following chart.

Remember that most optics take the form:

```
(a -> f b) -> (s -> f t)
```

However `Isos`, `Prisms` and `Reviews` generalize over the Profunctor type and thus can also have constraints on `p`:

```
p a (f b) -> p s (f t)
```

Keep this in mind when reading the chart.

Optic	Constraints
Lens	Functor f
Fold	Contravariant f, Applicative f
Traversal	Applicative f
Setter	Settable f
Getter	Contravariant f, Functor f
Iso	Functor f, Profunctor p
Prism	Applicative f, Choice p
Review	Settable f, Profunctor p, Bifunctor p

Composing optics adds **constraints** together, then running an optic with an **action** matches a data type which fulfills those constraints. Here's a table of lens actions and the data-type they use to

"run" the optics you pass to them:

Action	Data type
view (^.)	Const
set (.~)	Identity
over (%~)	Identity
fold queries: (toListOf, sumOf, lengthOf, etc.)	Const
review (#)	Tagged
traverseOf (%%~)	None
matching	Market

17. Answers to Exercises

17.1 Optic Anatomy

Back to questions

For each of the following, identify the **action**, **path** and **structure**, don't worry about understanding how they actually work just yet. If you want a real challenge, try to identify the **focus** too! Note that certain optics allow **multiple focuses**, and some actions accept **parameters** other than the **focus**.

```
>>> view (_1 . _2)  ((1, 2), 3)
2
```

action
    `view`
path
    `(_1 . _2)`
structure
    `((1, 2), 3)`
focus
    `2`

```
>>> set (_2 . _Left) "new" (False, Left "old")
(False,Left "new")
```

action
    `set`
path
    `(_2 . _Left)`
structure
    `(False, Left "old")`
focus
    `"old"`

```
>>> over (taking 2 worded . traversed) toUpper "testing one two three"
"TESTING ONE two three"
```

action
    `over`
path
    `(taking 2 worded . traversed)`
structure
    `"testing one two three"`
focus
    `"TESTING ONE"`

```
>>> foldOf (both . each) (["super", "cali"],["fragilistic", "expialidocious"])
"supercalifragilisticexpialidocious"
```

action
    `foldOf`
path
    `(both . each)`
structure
    `(["super", "cali"],["fragilistic", "expialidocious"])`
focus
    `"super","cali","fragilistic","expialidocious"`

17.2 Lens Actions

Back to questions

1. Find the **structure** and the **focus** in the following lens:

```
Lens' (Bool, (Int, String)) Int
```

Structure
    `(Bool, (Int, String))`
Focus
    Int

2. Write the type signature of a Lens with the **structure** `(Char, Int)` and the **focus** `Char`.

```
Lens' (Char, Int) Char
```

3. Name 3 **actions** we can use on a **Lens**.

`view`, `set`, `over`

4. Which lens could I use to **focus** the character 'c' in the following structure

```
('a', 'b', 'c')
```

You can use: `_3`

5. Write out the (simplified) types of each identifier in the following statement:

```
>>> over _2 (*10) (False, 2)
```

over
    general: `Lens' s a -> (a -> a) -> s -> s`
    specialized: `Lens' (Bool, Int) Int -> (Int -> Int) -> (Bool, Int) -> (Bool, Int)`
_2; **either are correct**:
    general: `Lens' (a, b) b`
    specialized: `Lens' (Bool, Int) Int`
(*10)
    `Int -> Int`
(False, 2)
    `(Bool, Int)`

What is the result of running it?

```
(False, 20)
```

17.3 Records Part One

Back to questions

1. The **structure** and **focus** of a lens are typically represented by which letters in type signatures?

structure
    `s`
focus
    `a`

2. Which **two** components are required to create a lens?

A **getter** and a **setter**

3. Implement the following lens:

```
name :: Lens' Ship String
name = lens getName setName
  where
    getName = _name
    setName ship newName = ship{_name=newName}
```

17.4 Records Part Two

Back to questions

1. What are the types of the lenses the following `makeLenses` call would generate?

- `wand :: Lens' Inventory Wand`
- `book :: Lens' Inventory Book`
- `potions :: Lens' Inventory [Potion]`

2. Using the heuristics in this chapter, what would you guess is the simple type of the following lens?

```
gazork :: Lens' Chumble Spuzz
```

3. The following code won't compile! What's wrong?

```
data Pet = Pet
    { _petName :: String
    , _petType :: String
    }

getPetName :: Pet -> String
getPetName pet = view petName pet

makeLenses ''Pet
```

The error says:

```
• Variable not in scope: petName :: Lens' Pet String
```

This one catches people all the time! `makeLenses` introduces a Template Haskell **splice**! It effectively splits compilation of the file into two chunks; the part **before** the splice and the part **after**. Basically we're not defining the `petName` lens until **after** we use it! Usually Haskell doesn't care about ordering within a file, so this can be a bit frustrating; you can usually avoid this problem by keeping your data declarations at the top of the file and calling `makeLenses` immediately after each data declaration.

Laws

Return to questions

1. Implement a lens which breaks the second and/or third law. That's get-set and set-set respectively.

Here's one that passes the set-get law, but breaks both of the others! Verify it yourself and see if you can prove it fails!

```
recorder :: Lens' ([a], a) a
recorder = lens getter setter
  where
    getter (_, a)  = a
    setter (history,  a) newVal = (a : history, newVal)
```

2. Test the get-set and set-set laws for the `msg` lens we wrote this chapter. Does it pass these laws?

Try it out yourself, you should see that it does pass these laws!

3. There's a different way we could have written the `msg` lens such that it would PASS the set-get law and the set-set law, but fail get-set. Implement this other version.

In this other version we REPLACE an `ExitCode` with a `ReallyBadError` when setting.

```
msg :: Lens' Err String
msg = lens getMsg setMsg
  where
    getMsg (ReallyBadError message) = message
    -- Hrmm, I guess we just return ""?
    getMsg (ExitCode _) = ""
    setMsg (ReallyBadError _) newMessage = ReallyBadError newMessage
    -- Nowhere to set it, I guess we do nothing?
    setMsg (ExitCode _) newMessage = ReallyBadError newMessage
```

4. Think up a new lens which is still useful even though it breaks a law or two.

You're on your own for this one, try going for a walk and pondering the wonders of the universe. It's all about the journey.

5. **BONUS** (this one is tricky): Live a little; write a lens which violates ALL THREE LAWS

Here's an example of a confusing one; every time you **set** it changes the focus! What sort of sadistic monster could invent such a thing?

```
flipper :: Lens' (Bool, a, a) a
flipper = lens getter setter
  where
    getter (True, a, _)  = a
    getter (False, _, a) = a
    setter (True,  a, b) newVal = (False, newVal, b)
    setter (False, a, b) newVal = (True, a, newVal)
```

6. **BONUS** (another tricky one): Can you write a lawful lens for the following type:

```
data Builder =
    Builder { _context :: [String]
            , _build   :: [String] -> String
            }
```

Your lens should be of type:

```
Lens' Builder String
```

For this one we need to get a bit creative!

An initial attempt might look like this:

```
builder :: Lens' Builder String
builder = lens getter setter
  where
    getter (Builder c f) = f c
    setter (Builder c f) newVal =
      Builder c (const newVal)
```

This gets really close! But it fails one of the laws? Can you see which one?

It doesn't pass `get-set`! If we get something from this type and set it back we've changed the function inside it entirely! This is clearly demonstrated if we give the builder a simple function like `concat`:

```
>>> let stringBuilder = Builder ["a", "b", "c"] concat
>>> let stringBuilder = Builder ["a", "b", "c"] Prelude.concat
>>> view builder stringBuilder
"abc"
>>> let newBuilder = set builder "abc" stringBuilder
>>> view builder newBuilder
"abc"
```

Everything **looks** fine; but what if we try changing the context?

```
>>> view builder newBuilder{_context=["d", "e", "f"]}
"abc"
```

This has clearly changed from the original behaviour; meaning we fail the law that setting what we get leaves us where we started.

Here's a dodgy patch we can do instead to make this work (though it's not exactly beautiful)

```
builder :: Lens' Builder String
builder = lens getter setter
  where
    getter (Builder c f) = f c
    setter (Builder c f) newVal =
      Builder c $ \c' ->
          if c' == c
             then newVal
             else f c'
```

We now check the input value every time we run the function, and only set the return value when it's equal to the original context!

Please don't do tricky things like this with your lenses, but it's fun to challenge ourselves to see how it can be done!

17.5 Virtual Fields

Return to questions

Consider this data type for the following exercises:

```
data User =
  User { _firstName :: String
       , _lastName :: String
       , _username :: String
       , _email :: String
       }
makeLenses ''User
```

1. We've decided we're no longer going to have separate usernames and emails; now the email will be used in place of a username. Your task is to delete the `_username` field and write a replacement `username` lens which reads and writes from/to the `_email` field instead. The change should be unnoticed by those importing the module.

```
data User =
    User { _firstName :: String
         , _lastName  :: String
         , _email     :: String
         }
    deriving Show

-- Rather than writing it out the long way (which you're free to do)
-- we can actually just set them to be equal!
username :: Lens' User String
username = email
```

2. Write a lens for the user's `fullName`. It should append the first and last names when "getting". When "setting" treat everything till the first space as the first name, and everything following it as the last name.

It should behave something like this:

```
>>> let user = User "John" "Cena" "invisible@example.com"
>>> view fullName user
"John Cena"
>>> set fullName "Doctor of Thuganomics" user
User
    { _firstName = "Doctor"
    , _lastName = "of Thuganomics"
    , _email = "invisible@example.com"
    }
```

```
fullName :: Lens' User String
fullName = lens getter setter
  where
    getter user = view firstName user <> " " <> view lastName user
    setter user new =
        let withFirstName = set firstName (head (words new)) user
         in set lastName (unwords . tail . words $ new) withFirstName
```

17.6 Self-Correcting Lenses

Return to questions

Consider the following:

```
data ProducePrices =
    ProducePrices { _limePrice  :: Float
                  , _lemonPrice :: Float
                  }
    deriving Show
```

1. We're handling a system for pricing our local grocery store's citrus produce! Our first job is to write lenses for setting the prices of limes and lemons. Write lenses for `limeCost` and `lemonCost` which prevent **negative** prices by rounding up to `0` (we're okay with given produce out for free, but certainly aren't going to pay others to take it).

```
limePrice :: Lens' ProducePrices Float
limePrice = lens getter setter
  where
    getter = _limePrice
    setter prices new = prices{_limePrice = max 0 new}

lemonPrice :: Lens' ProducePrices Float
lemonPrice = lens getter setter
  where
    getter = _lemonPrice
    setter prices new = prices{_lemonPrice = max 0 new}
```

2. The owner has informed us that it's VERY important that the prices of limes and lemons must NEVER be further than 50 cents apart or the produce world would descend into total chaos. Update your lenses so that when setting lime-cost the lemon-cost is rounded to within 50 cents; (and vice versa).

It should behave something like this; don't worry if you can't get it exactly right, this one is tricky!

```
>>> let prices = ProducePrices 1.50 1.48
>>> set limePrice 2 prices
ProducePrices
    { _limePrice = 2.0
    , _lemonPrice = 1.5
    }
>>> set limePrice 1.8 prices
ProducePrices
    { _limePrice = 1.8
    , _lemonPrice = 1.48
    }
>>> set limePrice 1.63 prices
ProducePrices
    { _limePrice = 1.63
    , _lemonPrice = 1.48
    }
>>> set limePrice  (-1.00) prices
ProducePrices
    { _limePrice = 0.0
    , _lemonPrice = 0.5
    }
```

Here's the implementation I came up with:

```
limePrice :: Lens' ProducePrices Float
limePrice = lens getter setter
  where
    getter = _limePrice
    setter prices newLimePrice =
        let correctedLimePrice = max 0 newLimePrice
            correctedLemonPrice
              =   min (correctedLimePrice + 0.50)
                . max (correctedLimePrice - 0.50)
                $ (_lemonPrice prices)
         in prices{_limePrice = correctedLimePrice, _lemonPrice = correctedLemonPric\
e}

lemonPrice :: Lens' ProducePrices Float
lemonPrice = lens getter setter
  where
    getter = _lemonPrice
    setter prices newLemonPrice =
        let correctedLemonPrice = max 0 newLemonPrice
```

```
            correctedLimePrice
              =   min (correctedLemonPrice + 0.50)
                . max (correctedLemonPrice - 0.50)
                $ (_limePrice prices)
        in prices{_lemonPrice = correctedLemonPrice, _limePrice = correctedLimePric\
e}
```

17.7 Polymorphic Lenses

Back to questions

1. Write the type signature of the polymorphic lens which would allow changing a `Vorpal x` to a `Vorpal y`.

```
Lens (Vorpal x) (Vorpal y) x y
```

2. Find one possible way to write a polymorphic lens which changes the type of the `best` and `worst` fields in the `Preferences` type above. You're allowed to change the type of the lenses or alter the type itself!

```
data Preferences a =
    Preferences { _best :: a
                , _worst :: a
                }
    deriving (Show)
```

Here's one possible way:

```
favourites :: Lens (Preferences a) (Preferences b) (a, a) (b, b)
```

By pairing up both values in the focus we can pass **both** values packed up into a tuple. We're still only selecting **one** focus with the lens, but we're being sneaky and filling that focus with **two** values.

This isn't terribly helpful in this case, but there are times when you may need to use this sort of roundabout method.

Another option would be to rewrite the data type to have two type variables instead:

```
data Preferences a b =
    Preferences { _best :: a
                , _worst :: b
                }
    deriving (Show)
```

Now it's easy to write polymorphic lenses with the following signatures:

```
best  :: (Preferences a b) (Preferences a' b) a a'
worst :: (Preferences a b) (Preferences a b') b b'
```

3. We can change type of more complex types too. Write out the type of a valid lens which could be used to change from a `(Result e)` to a `(Result f)`

```
data Result e =
    Result { _lineNumber :: Int
           , _result     :: Either e String
           }
```

Because the `e` is inside an `Either` type we can't focus it with a lens directly, we're not always guaranteed to have an `e` available. We'll have to focus the entire `Either` and let the user decide what to do if it's missing:

```
result :: Lens (Result e) (Result f) (Either e String) (Either f String)
```

4. It's thinking time! Is it possible to change more than **one** type variable at a time using a polymorphic lens?

It sure is! We might need to get clever about it though; Here's an example

```
data ParseResult e a =
    Error e
  | Result a
  deriving Show
```

We can write a lens which allows the user to edit either of the two cases:

```
result :: Lens (ParseResult e a) (ParseResult f b) (Either e a) (Either f b)
result = lens getter setter
  where
    getter (Error e) = Left e
    getter (Result a) = Right a
    setter _ (Left e) = Error e
    setter _ (Right a) = Result a
```

This example is a bit silly, but it works in more complex situations too!

5. **BONUS** Come up with some sort of lens to change from a `Predicate a` to a `Predicate b`

This is another one of those cases where we can't focus the type variable directly, we'll need to pass the full structure to the user and let them handle it:

```
data Predicate a =
    Predicate (a -> Bool)


Lens (Predicate a) (Predicate b) (a -> Bool) (b -> Bool)
```

17.8 Lens Composition

Return to questions

1. Fill in the blank with the appropriate composition of tuple lenses in the following statement:

```
>>> view (_2 . _1 . _2) ("Ginerva", (("Galileo", "Waldo"), "Malfoy"))
"Waldo"
```

2. Given the following lens types, fill in the missing type of `mysteryDomino`

```
fiveEightDomino :: Lens' Five Eight
mysteryDomino   :: Lens' ???? ????
twoThreeDomino  :: Lens' Two Three

dominoTrain :: Lens' Five Three
dominoTrain = fiveEightDomino . mysteryDomino . twoThreeDomino


mysteryDomino :: Lens' Eight Two
```

3. Using what you know about how lenses work under the hood; rewrite the following signature as a polymorphic lens of the form: `Lens s t a b`

```
Functor f => (Armadillo -> f Hedgehog) -> (Platypus -> f BabySloth)
```

```
Lens Platypus BabySloth Armadillo Hedgehog
```

Platypus
: `s` – The **pre-action structure**

BabySloth
: `t` – The **post-action structure**

Armadillo
: `a` – The **pre-action focus**

Hedgehog
: `b` – The **post-action focus**

4. Find a way to compose ALL of the following lensees together into one big path using each exactly once. What's the type of the resulting lens?

```
spuzorktrowmble   :: Lens Chumble      Spuzz      Gazork       Trowlg
gazorlglesnatchka :: Lens Gazork       Trowlg     Bandersnatch Yakka
zinkattumblezz    :: Lens Zink         Wattoom    Chumble      Spuzz
gruggazinkoom     :: Lens Grug         Pubbawup   Zink         Wattoom
banderyakoobog    :: Lens Bandersnatch Yakka      Foob         Mog
boowockugwup      :: Lens Boojum       Jabberwock Grug         Pubbawup
snajubjumwock     :: Lens Snark        JubJub     Boojum       Jabberwock
```

Here's the full behemoth!

```
snajubfoogog :: Lens Snark JubJub Foob Mog
snajubfoogog =
    snajubjumwock     -- Lens Snark JubJub Boojum Jabberwock
  . boowockugwup      -- Lens Boojum Jabberwock Grug Pubbawup
  . gruggazinkoom     -- Lens Grug Pubbawup Zink Wattoom
  . zinkattumblezz    -- Lens Zink Wattoom Chumble Spuzz
  . spuzorktrowmble   -- Lens Chumble Spuzz Gazork Trowlg
  . gazorlglesnatchka -- Lens Gazork Trowlg Bandersnatch Yakka
  . banderyakoobog    -- Lens Bandersnatch Yakka Foob Mog
```

17.9 Operators

Return to questions

1. Consider the following type:

```
data Gate =
    Gate { _open    :: Bool
         , _oilTemp :: Float
         } deriving Show
makeLenses ''Gate

data Army =
    Army { _archers :: Int
         , _knights :: Int
         } deriving Show
makeLenses ''Army

data Kingdom =
    Kingdom { _name :: String
            , _army :: Army
            , _gate :: Gate
            } deriving Show
makeLenses ''Kingdom
```

Given the following starting state:

```
duloc :: Kingdom
duloc =
    Kingdom { _name = "Duloc"
            , _army = Army { _archers = 22
                           , _knights = 14
                           }
            , _gate = Gate { _open    = True
                           , _oilTemp = 10.0
                           }
            }
```

Write a chain of operations using infix operators to get from the start state to each of the following goal states:

Hard mode: Try doing it without using `%~` or `.~`!

```
>>> goalA
Kingdom
    { _name = "Duloc: a perfect place"
    , _army = Army
        { _archers = 22
        , _knights = 42
        }
    , _gate = Gate
        { _open = False
        , _oilTemp = 10.0
        }
    }
```

I'll do all answers in Hard Mode:
Answer:

```
goalA :: Kingdom
goalA = duloc
      & gate . open &&~ False
      & army . knights *~ 3
      & name <>~ ": a perfect place"
```

```
>>> goalB
Kingdom
    { _name = "Dulocinstein"
    , _army = Army
        { _archers = 17
        , _knights = 26
        }
    , _gate = Gate
        { _open = True
        , _oilTemp = 100.0
        }
    }
```

Answer:

```
goalB :: Kingdom
goalB = duloc
      & gate . oilTemp ^~ 2
      & army . archers -~ 5
      & army . knights +~ 12
      & name <>~ "instein"
```

Bonus: Good luck with this one! Notice the tuple around the finished Kingdom! Also; there definitely aren't any typos! This is your goal. You've got all the tools you need to figure it out. Let's see what you can do.

```
>>> goalC
( "Duloc: Home"
, Kingdom
    { _name = "Duloc: Home of the talking Donkeys"
    , _army = Army
        { _archers = 22
        , _knights = 14
        }
    , _gate = Gate
        { _open = True
        , _oilTemp = 5.0
        }
    }
)
```

Answer (there's a simpler way to do this one too, see if you can find it!):

```
goalC :: (String, Kingdom)
goalC = duloc
    & name <<>~ ": Home"
    & _2 . name <>~ " of the talking Donkeys"
    & _2 . gate . oilTemp //~ 2
```

2. Enter the appropriate operator in the `undefined` slot to make each code example consistent:

a.

```
>>> (False, "opossums") & _1 ||~ True
(True, "opossums")
```

Remember that `id` is a lens which focuses the full **structure**

```
>>> 2 & id *~ 3
6


>>> ((True, "Dudley"), 55.0) & _1 . _2 <>~ " - the worst" & _2 -~ 15 & _2 //~ 2 & _1\
 . _2 %~ map toUpper & _1 . _1 &&~ False
((False,"DUDLEY - THE WORST"),20.0)
```

3. Name a lens operator that takes only two arguments

```
(^.) :: s -> Lens' s a -> a`
```

4. What's the type signature of %~? Try to figure it without checking! Look at the examples above if you have to!

```
(%~) :: Lens s t a b -> (a -> b) -> s -> t
```

17.10 Simple Folds

Return to questions

1. What's the result of each expression?

```
beastSizes :: [(Int, String)]
beastSizes = [(3, "Sirens"), (882, "Kraken"), (92, "Ogopogo")]

>>> beastSizes ^.. folded
[(3,"Sirens"),(882,"Kraken"),(92,"Ogopogo")]

>>> beastSizes ^.. folded . folded
["Sirens","Kraken","Ogopogo"]

>>> beastSizes ^.. folded . folded . folded
"SirensKrakenOgopogo"

>>> beastSizes ^.. folded . _2
["Sirens","Kraken","Ogopogo"]

>>> toListOf (folded . folded) [[1, 2, 3], [4, 5, 6]]
[1, 2, 3, 4, 5, 6]

>>> toListOf
```

```
      (folded . folded)
      (M.fromList [("Jack", "Captain"), ("Will", "First Mate")])
"CaptainFirst Mate"

>>> ("Hello", "It's me") ^.. both . folded
"HelloIt's me"

>>> ("Why", "So", "Serious?") ^.. each
["Why", "So", "Serious?"]

quotes :: [(T.Text, T.Text, T.Text)]
quotes = [("Why", "So", "Serious?"), ("This", "is", "SPARTA")]

>>> quotes ^.. each . each . each
"WhySoSerious?ThisisSPARTA"
```

2. Write out the 'specialized' type for each of the requested combinators used in each of the following expressions:

`folded` and `_1`

```
>>> toListOf (folded . _1) [(1, 'a'), (2, 'b'), (3, 'c')]
[1, 2, 3]


folded :: Fold [(Int, Char)] (Int, Char)

_1 :: Fold (Int, Char) Int
```

`folded`, `_2`, and `toListOf`

```
>>> toListOf (_2 . folded) (False, S.fromList ["one", "two", "three"])
["one", "two", "three"]


_2 :: Fold (Bool, S.Set String) (S.Set String)

folded :: Fold (S.Set String) String
```

`folded` and `folded` and `toListOf`

```
>>> toListOf
      (folded . folded)
      (M.fromList [("Jack", "Captain"), ("Will", "First Mate")])
"CaptainFirst Mate"
```

```haskell
-- First folded
folded :: Fold (M.Map String String) String
-- Second folded
folded :: Fold String Char

toListOf :: Fold (M.Map String String) Char
         -> (M.Map String String)
         -> String
```

3. Fill in the blank with the appropriate fold to get the specified results

```
>>> [1, 2, 3] ^.. folded
[1, 2, 3]

>>> ("Light", "Dark") ^.. _1
["Light"]

>>> [("Light", "Dark"), ("Happy", "Sad")] ^.. folded . both
["Light","Dark","Happy","Sad"]

>>> [("Light", "Dark"), ("Happy", "Sad")] ^.. folded . _1
["Light","Happy"]

>>> [("Light", "Dark"), ("Happy", "Sad")] ^.. folded . _2 . folded
"DarkSad"

>>> ("Bond", "James", "Bond") ^.. each
["Bond","James","Bond"]
```

17.11 Writing Custom Folds

Return to questions

1. Fill in each blank with either `to`, `folded`, or `folding`.

```
>>> ["Yer", "a", "wizard", "Harry"] ^.. folded . folded
"YerawizardHarry"

>>> [[1, 2, 3], [4, 5, 6]] ^.. folded . folding (take 2)
[1, 2, 4, 5]

>>> [[1, 2, 3], [4, 5, 6]] ^.. folded . to (take 2)
[[1,2],[4,5]]

>>> ["bob", "otto", "hannah"] ^.. folded . to reverse
["bob", "otto", "hannah"]

>>> ("abc", "def") ^.. folding (\(a, b) -> [a, b]). to reverse . folded
"cbafed"
```

2. Fill in the blank for each of the following expressions with a **path** of Folds which results in the specified answer. Avoid partial functions and `fmap`.

```
>>> [1..5] ^.. folded . to (*100)
[100,200,300,400,500]

>>> (1, 2) ^.. folding (\(a, b) -> [a, b])
[1, 2]

>>> [(1, "one"), (2, "two")] ^.. folded . to snd
["one", "two"]

>>> (Just 1, Just 2, Just 3) ^.. folding (\(a, b, c) -> [a, b, c]) . folded
[1, 2, 3]

>>> [Left 1, Right 2, Left 3] ^.. folded . folded
[2]

>>> [([1, 2], [3, 4]), ([5, 6], [7, 8])]
      ^.. folded . folding (\(a, b) -> a <> b)
[1, 2, 3, 4, 5, 6, 7, 8]

>>> [1, 2, 3, 4] ^.. folded . to (\n -> if odd n then Left n else Right n)
[Left 1, Right 2, Left 3, Right 4]

>>> [(1, (2, 3)), (4, (5, 6))] ^.. folded . folding (\(a, (b, c)) -> [a, b, c])
[1, 2, 3, 4, 5, 6]
```

```
>>> [(Just 1, Left "one"), (Nothing, Right 2)]
      ^.. folded . folding (\(a, b) -> a ^.. folded <> b ^.. folded)
[1, 2]

>>> [(1, "one"), (2, "two")] ^.. folded . folding (\(a, b) -> [Left a, Right b])
[Left 1, Right "one", Left 2, Right "two"]

>>> S.fromList ["apricots", "apples"] ^.. folded . folding reverse
"selppastocirpa"
```

3. **BONUS** – Devise a fold which returns the expected results. Think outside the box a bit.

```
>>> [(12, 45, 66), (91, 123, 87)]
     ^.. folded
       . _2
       . to show
       . to reverse
       . folded
"54321"

>>> [(1, "a"), (2, "b"), (3, "c"), (4, "d")]
  ^.. folded
    . folding (\(a, b) -> if (even a)
                          then return b
                          else [])
["b", "d"]
```

17.12 Querying Using Folds

Return to questions

Consider the following list of actions for the exercises below:

```
elemOf    :: Eq a => Fold s a -> a -> s -> Bool
has       :: Fold s a -> s -> Bool
lengthOf  :: Fold s a -> s -> Int
sumOf     :: Num n => Fold s n -> s -> n
productOf :: Num n => Fold s n -> s -> n
foldOf    :: Monoid a => Fold s a -> s -> a
preview   :: Fold s a -> s -> Maybe a
lastOf    :: Fold s a -> s -> Maybe a
minimumOf :: Ord a => Fold s a -> s -> Maybe a
```

```
maximumOf :: Ord a => Fold s a -> s -> Maybe a
anyOf     :: Fold s a -> (a -> Bool) -> s -> Bool
allOf     :: Fold s a -> (a -> Bool) -> s -> Bool
findOf    :: Fold s a -> (a -> Bool) -> s -> Maybe a
foldrOf   :: Fold s a -> (a -> r -> r) -> r -> s -> r
foldMapOf :: Monoid r => Fold s a -> (a -> r) -> s -> r
```

1. Pick the matching action from the list for each example:

```
>>> has folded []
False

>>> foldOf both ("Yo", "Adrian!")
"YoAdrian!"

>>> elemOf each "phone" ("E.T.", "phone", "home")
True

>>> minimumOf folded [5, 7, 2, 3, 13, 17, 11]
Just 2

>>> lastOf folded [5, 7, 2, 3, 13, 17, 11]
Just 11

>>> anyOf folded ((> 9) . length) ["Bulbasaur", "Charmander", "Squirtle"]
True

>>> findOf folded even [11, 22, 3, 5, 6]
Just 22
```

2. Use an action from the list along with any Fold you can devise to retrieve the output from the input in each of the following challenges. There may be more than one correct answer.

Find the first word in the input list which is a palindrome; I.e. a word that's the same backwards as forwards.

```
>>> findOf folded (\s -> s == reverse s) ["umbrella", "olives", "racecar", "hammer"]
Just "racecar"
```

Determine whether all elements of the following tuple are EVEN

```
>>> allOf each even (2, 4, 6)
output = True
```

Find the pair with the largest integer

```
>>> maximumByOf
      folded
      (compare `on` fst)
      [(2, "I'll"), (3, "Be"), (1, "Back")]
Just (3,"Be")
```

Find the sum of both elements of a tuple. This one may require additional alterations AFTER running a Fold.

```
>>> getSum $ foldMapOf both Sum (1, 2)
3
```

3. **BONUS** – These are a bit trickier

Find which word in a string has the most vowels.

```
>>> maximumByOf (folding words)
                (compare `on` (length . filter (`elem` "aeiouy")))
                "Do or do not, there is no try."
Just "there"
```

Combine the elements into the expected String

```
-- You can replace foldByOf with foldrOf or foldlOf
>>> foldByOf folded (flip (++)) "" ["a", "b", "c"]
"cba"
-- The following also works
>>> import Data.Monoid (Dual(..))
>>> getDual $ foldMapOf folded Dual ["a", "b", "c"]
"cba"
```

Good luck with the following ones! Get it to work however you can!

```
>>> [(12, 45, 66), (91, 123, 87)]
      ^.. folded
        . _2
        . to show
        . to reverse
        . folded
"54321"

>>> [(1, "a"), (2, "b"), (3, "c"), (4, "d")]
  ^.. folded
    . folding (\(a, b) -> if (even a)
                          then return b
                          else [])
["b", "d"]
```

17.13 Higher Order Folds

Return to questions

1. Fill in the blank. You'll need to remember some tricks from previous sections!

```
>>> "Here's looking at you, kid" ^.. dropping 7 folded
"looking at you, kid"

>>> ["My Precious", "Hakuna Matata", "No problemo"]
      ^.. folded . taking 1 worded
["My","Hakuna","No"]

>>> ["My Precious", "Hakuna Matata", "No problemo"]
      ^.. taking 1 (folded . worded)
["My"]

>>> ["My Precious", "Hakuna Matata", "No problemo"]
      ^.. folded . taking 1 worded . folded
"MyHakunaNo"

>>> import Data.Char (isAlpha)
>>> ["My Precious", "Hakuna Matata", "No problemo"]
      ^.. folded . takingWhile isAlpha folded
"MyHakunaNo"
-- OR
>>> ["My Precious", "Hakuna Matata", "No problemo"]
```

```
      ^.. folded . taking 1 (folding words) . folded
"MyHakunaNo"

>>>  sumOf (taking 2 each) (10, 50, 100)
60

>>> ("stressed", "guns", "evil" :: String) ^.. backwards each
["evil","guns","stressed"]

>>> ("stressed", "guns", "evil" :: String) ^.. backwards each . to reverse
["live","snug","desserts"]

>>> import Data.Char (isAlpha)
>>> "blink182 k9 blazeit420" ^.. worded . droppingWhile isAlpha folded
"1829420"
```

2. Solve the following problems using higher-order folds:

We're doing a temperature study in a given region, but we need to tests on several subsets of the data.

Here's the series of daily temperature measurements we've been given for the region:

```
sample :: [Int]
sample = [-10, -5, 4, 3, 8, 6, -2, 3, -5, -7]
```

First we're interested in how many days it took until the first thaw. Write an expression which calculates the number of measurements until a temp is above zero.

```
>>> lengthOf (takingWhile (<= 0) folded) sample
2
```

For the next study we need to know the warmest it got in the first 4 days of the sample. Write an expression to calculate it.

```
>>> maximumOf (taking 4 folded) sample
Just 4
```

Write an expression to calculate the temperature on the day AFTER we hit that temperature. Use `preview` or `^?` somewhere in that expression if you can.

```
>>> sample ^? dropping 1 (droppingWhile (/= 4) folded)
Just 3
```

How many days of below-freezing weather did we have consecutively at the END of the sample?

```
>>> lengthOf (takingWhile (<0) (backwards folded)) sample
2
```

Now we're interested in running statistics on temperature data specifically from the first thaw until the next freeze. Write an expression which lists out all temperature samples from the first time we sample above 0, until the next time we're below zero.

```
-- We can combine multiple modifiers to get this behaviour,
-- We drop all leading negative temperatures and then
-- take temperatures until another freeze!
>>> sample ^.. takingWhile (> 0) (droppingWhile (< 0) folded)
[4,3,8,6]
```

BONUS: List out all the temperature samples between the FIRST thaw and the FINAL freeze.

```
>>> sample ^.. backwards (droppingWhile (< 0) (backwards (droppingWhile (< 0) folded\
)))
[4,3,8,6,-2,3]
```

Generalize this behaviour into a function: `trimmingWhile`. It should drop elements from the start AND end of a fold while a predicate is True.

```
trimmingWhile :: (a -> Bool) -> Fold s a -> Fold s a


trimmingWhile :: (a -> Bool) -> Fold s a -> Fold s a
trimmingWhile predicate theFold =
-- Note that we're composing **modifiers** here, NOT composing Folds!
-- It'll help to read the composition chain from the bottom up!

  -- We reverse the fold ordering again to get back to the initial order
    backwards
  -- Due to the reversal, the suffix is now the prefix,
  -- so we drop matching elements again:
  . droppingWhile predicate
  -- We reverse the fold over all the remaining elements:
  . backwards
  -- First we drop the leading elements which match:
  . droppingWhile predicate
  $ theFold
```

17.14 Filtering

Return to questions

1. Write a Fold to answer each of the questions about my card collection:

   - List all the cards whose name starts with `'S'`

```
>>> deck ^.. folded . name . filtered ((== 'S') . head)
[ "Skwortul"
, "Scorchander"
, "Seedasaur"
, "Spicyeon"
, "Sparkeon"
]
```

   - What's the lowest attack power of all moves?

```
>>> minimumOf (folded . moves . folded . movePower) deck
Just 3
```

   - What's the name of the first card which has more than one move?

```
>>> deck ^? folded . filtered ((>1) . length . _moves) . name
Just "Kapichu"
```

   - Are there any `Hot` cards with a move with more than 30 attack power?

```
>>> anyOf
      ( folded
      . filteredBy (aura . only Hot)
      . moves
      . folded
      . movePower )
      (>30)
      deck
True
```

   - List the names of all holographic cards with a `Wet` aura

```
>>> deck ^.. folded . filtered _holo . filtered ((==Wet) . _aura) . name
[ "Garydose" ]
```

- What's the sum of all attack power for all moves belonging to non-Leafy cards?

```
>>> sumOf
      ( folded
      . filtered ((/= Leafy) . _aura)
      . moves
      . folded
      . movePower )
      deck
303
```

17.15 Simple Traversals

Return to questions

1. Short answer questions:

- What type of optic do you get when you compose a traversal with a fold?

A fold

- Which of the optics we've learned can act as a Traversal?

Either a lens or traversal

- Which of the optics we've learned can act as a Fold?

Lens, traversal, or fold

2. Fill in the blank to complete each expression:

```
>>> ("Jurassic", "Park") & both .~ "N/A"
("N/A", "N/A")

>>> ("Jurassic", "Park") & both . each .~ 'x'
-- OR
>>> ("Jurassic", "Park") & both . traversed .~ 'x'
("xxxxxxxx", "xxxx")

>>> ("Malcolm", ["Kaylee", "Inara", "Jayne"])
      & beside id traversed %~ take 3
("Mal", ["Kay", "Ina", "Jay"])

>>> ("Malcolm", ["Kaylee", "Inara", "Jayne"])
      & _2 . element 1 .~ "River"
("Malcolm", ["Kaylee", "River", "Jayne"])

>>> ["Die Another Day", "Live and Let Die", "You Only Live Twice"]
      & traversed . elementOf worded 1 . traversed .~ 'x'
[ "Die xxxxxxx Day"
, "Live xxx Let Die"
, "You xxxx Live Twice"
]

>>> ((1, 2), (3, 4)) & both . both +~ 1
((2, 3), (4, 5))

>>> (1, (2, [3, 4])) & beside id (beside id traversed) +~ 1
(2, (3, [4, 5]))

>>> import Data.Char (toUpper)
>>> ((True, "Strawberries"), (False, "Blueberries"), (True, "Blackberries"))
      & each . filtered fst . _2 . taking 5 traversed
      %~ toUpper
((True, "STRAWberries"), (False, "Blueberries"), (True, "BLACKberries"))

>>> ((True, "Strawberries"), (False, "Blueberries"), (True, "Blackberries"))
      & each %~ snd
("Strawberries", "Blueberries", "Blackberries")
```

17.16 Traversal Actions

Return to questions

1. Fill in the blanks, you know the drill

```
>>> sequenceAOf _1 (Nothing, "Rosebud")
Nothing

>>> sequenceAOf (traversed . _1) [("ab", 1), ("cd", 2)]
[ [('a', 1) ,('c', 2)]
, [('a', 1) ,('d', 2)]
, [('b', 1) ,('c', 2)]
, [('b', 1) ,('d', 2)]]

-- The ZipList effect groups elements by position in the list
>>> sequenceAOf traversed [ZipList [1, 2], ZipList [3, 4]]
ZipList {getZipList = [[1,3],[2,4]]}

>>> sequenceAOf (traversed . _2) [('a', ZipList [1, 2]), ('b', ZipList [3, 4])]
ZipList {getZipList = [[('a',1),('b',3)],[('a',2),('b',4)]]}

>>> import Control.Monad.State
>>> let result = traverseOf (beside traversed both)
                            (\n -> modify (+n) >> get)
                            ([1, 1, 1], (1, 1))
>>> runState result 0
(([1,2,3],(4,5)), 5)
```

2. Rewrite the following using the infix-operator for `traverseOf`

```
>>> import Data.Char (toUpper, toLower)
>>> traverseOf
      (_1 . traversed)
      (\c -> [toLower c, toUpper c])
      ("ab", True)
[ ("ab",True)
, ("aB",True)
, ("Ab",True)
, ("AB",True)
]

-- operator syntax:

>>> ("ab", True) & _1 . traversed %%~ (\c -> [c, toUpper c])
[("ab",True),("aB",True),("Ab",True),("AB",True)]
```

```
>>> traverseOf
      (traversed . _1)
      (\c -> [toLower c, toUpper c])
      [('a', True), ('b', False)]
[ [('a',True), ('b',False)]
, [('a',True), ('B',False)]
, [('A',True), ('b',False)]
, [('A',True), ('B',False)]
]

-- operator syntax:

>>> [('a', True), ('b', False)]
      & traversed . _1  %~ (\c -> [toLower c, toUpper c])
[ [('a',True), ('b',False)]
, [('a',True), ('B',False)]
, [('A',True), ('b',False)]
, [('A',True), ('B',False)]
]
```

3. Given the following data definitions, write a validation function which uses `traverseOf` or `%%~` to validates that the given user has an age value above zero and below 150. Return an appropriate error message if it fails validation.

```
data User =
    User { _name :: String
         , _age  :: Int
         } deriving Show
makeLenses ''User

data Account =
    Account { _id   :: String
            , _user :: User
            } deriving Show
makeLenses ''Account

validateAge :: Account -> Either String Account
validateAge = traverseOf (user . age) check
  where
    check n | n < 0 = Left "A bit young to have an account here aren't ya?"
            | n > 150 = Left "A bit old to have an account here aren't ya?"
            | otherwise = Right n
```

17.17 Custom Traversals

Return to questions

1. Rewrite the `amount` lens manually as a traversal:

```
amountT :: Traversal' Transaction Int
amountT handler (Deposit n) = Deposit <$> handler n
amountT handler (Withdrawal n) = Withdrawal <$> handler n
```

2. Reimplement the `both` traversal over tuples:

```
both :: Traversal (a, a) (b, b) a b
both handler (a, a') =
    liftA2 (,)
      (handler a)
      (handler a')
```

3. Write the following custom traversal:

```
transactionDelta :: Traversal' Transaction Int
```

It should focus the amount of the transaction, but should reflect the **change** that the transaction causes to the balance of the account. That is, `Deposits` should be a positive number, but `Withdrawals` should be negative. The traversal should not change the underlying representation of the data.

Here's how it should behave:

```
>>> Deposit 10 ^? transactionDelta
Just 10

-- Withdrawal's delta is negative
>>> Withdrawal 10 ^? transactionDelta
Just (-10)

>>> Deposit 10 & transactionDelta .~ 15
Deposit {_amount = 15}

>>> Withdrawal 10 & transactionDelta .~ (-15)
Withdrawal {_amount = 15}

>>> Deposit 10 & transactionDelta +~ 5
```

```
Deposit {_amount = 15}

>>> Withdrawal 10 & transactionDelta +~ 5
Withdrawal {_amount = 5}
```

Here's the implementation:

```
transactionDelta :: Traversal' Transaction Int
transactionDelta handler (Deposit n) = Deposit <$> handler n
transactionDelta handler (Withdrawal n) = Withdrawal . negate <$> handler (negate n)
```

4. Implement `left`:

```
left :: Traversal (Either a b) (Either a' b) a a'
left f (Left a) = Left <$> f a
left _ (Right b) = pure (Right b)
```

5. **BONUS**: Reimplement the `beside` traversal:

```
beside :: Traversal s t a b
       -> Traversal s' t' a b
       -> Traversal (s,s') (t,t') a b
beside left' right' handler (s, s') =
    liftA2 (,)
      (s & left' %%~ handler)
      (s' & right' %%~ handler)
```

Hint: You can use `traverseOf` or `%%~` to help simplify your implementation!

17.18 Traversal Laws

Return to questions

1. `worded` is a law-breaking traversal! Determine which law it breaks and give an example which shows that it doesn't pass the law.

It breaks the second law; here's a counter-example for the test:

```
replaceUnderscores :: Char -> Char
replaceUnderscores '_' = ' '
replaceUnderscores c = c

>>> "one two" & worded %~ (<> " missisipi") & worded %~ reverse
"eno ipisissim owt ipisissim"

-- Should be equal to:
>>> "one two" & worded %~ reverse . (<> " missisipi")
"ipisissim eno ipisissim owt"
```

It fails because adding a space to the string during an update creates new focuses for the `worded` traversal.

2. Write a custom traversal which breaks the first law. Be as creative as you like!

You're on your own for this one captain! Just verify that you have a counter example for the first law.

3. Write a custom traversal which breaks the second law. Be as creative as you like!

You're on your own for this one captain! Just verify that you have a counter example for the second law.

3. For each of the following traversals, decide whether you think they're lawful or unlawful, and if they're unlawful come up with a counter-example for one of the laws.

- `taking` is lawful (so long as the traversal you pass as an argument is lawful)
- `beside` is lawful
- `each` is lawful
- `lined` is unlawful! What happens if the update function introduces additional newlines?
- `traversed` is lawful

17.19 partsOf

Return to questions

Fill in the blanks

```
-- Viewing
>>> [1, 2, 3, 4] ^. partsOf (traversed . filtered even)
[2, 4]

>>> ([1, 2], M.fromList [('a', 3), ('b', 4)])
      ^. partsOf (beside traversed traversed)
[1,2,3,4]

>>> ["Aardvark", "Bandicoot", "Capybara"]
      ^. traversed . partsOf (taking 3 traversed)
"AarBanCap"

-- Setting
>>> [1, 2, 3, 4]
      & partsOf (traversed . filtered even) .~ [20, 40]
[1,20,3,40]

>>> [1, 2, 3, 4]
      & partsOf (traversed . filtered even) .~ [20, 40]
[1,20,3,40]

>>> ["Aardvark", "Bandicoot", "Capybara"]
      & partsOf (traversed . traversed) .~ "Kangaroo"
["Kangaroo","Bandicoot","Capybara"]

>>> ["Aardvark", "Bandicoot", "Capybara"]
      & partsOf (traversed . traversed) .~ "Ant"
["Antdvark","Bandicoot","Capybara"]

-- Modifying
-- Tip: Map values are traversed in order by KEY
>>> M.fromList [('a', 'a'), ('b', 'b'), ('c', 'c')]
      & partsOf traversed %~ \(x:xs) -> xs ++ [x]
M.fromList [('a','b'),('b','c'),('c','a')]

>>> ('a', 'b', 'c') & partsOf each %~ reverse
('c', 'b', 'a')

-- Bonus
>>> [1, 2, 3, 4, 5, 6]
      & partsOf (taking 3 traversed) %~ reverse
[3,2,1,4,5,6]
```

```
>>> ('a', 'b', 'c')
      & unsafePartsOf each %~ \xs -> fmap ((,) xs) xs
(("abc",'a'),("abc",'b'),("abc",'c'))
```

17.20 Indexable Structures

Return to questions

1. Fill in the blanks

```
>>> ["Larry", "Curly", "Moe"]
      & ix 1 .~ "Wiggly"
["Larry","Wiggly","Moe"]

>>> let heroesAndVillains =
          M.fromList [("Superman", "Lex"), ("Batman", "Joker")]
>>> heroesAndVillains & at "Spiderman" .~ Just "Goblin"
M.fromList [("Batman","Joker"),("Spiderman","Goblin"),("Superman","Lex")]

>>> sans "Superman" heroesAndVillains
M.fromList [("Batman","Joker")]

>>> S.fromList ['a', 'e', 'i', 'o', 'u']
      & at 'y' ?~ ()
      & at 'i' .~ Nothing
S.fromList "aeouy"
```

2. Use `ix` and `at` to go from the input to the output:

```
input = M.fromList [("candy bars", 13), ("soda", 34), ("gum", 7)]
output = M.fromList [("candy bars",13),("ice cream",5),("soda",37)]

>>> input & sans "gum"
          & ix "soda" +~ 3
          & at "ice cream" ?~ 5
```

17.21 Custom Indexed Structures

Return to questions

1. Implement both `Ixed` and `At` for a newtype wrapper around a `Map` which makes indexing case insensitive, you can specialize to `String` or `Text` keys. Write the `ix` instance manually even though it has a default implementation. It's okay to assume that user will only interact with the map via `Ixed` and `At`.

```haskell
module Indexed.Custom where

import Control.Lens
import qualified Data.Map as M
import Data.Char

newtype CaseInsensitive v
  = CaseInsensitive {_unCaseSensitive :: M.Map String v }
  deriving Show

-- I generate a lens for unwrapping the contents
-- We can use this to make building ix and at easier.
makeLenses ''CaseInsensitive

type instance Index (CaseInsensitive v) = String
type instance IxValue (CaseInsensitive v) = v

instance Ixed (CaseInsensitive v) where
  -- After lower-casing the key,
  -- we can rely on the `ix` traversal for the underlying Map.
  ix :: String -> Traversal' (CaseInsensitive v) v
  ix k = unCaseSensitive . ix (map toLower k)

instance At (CaseInsensitive v) where
  at :: String -> Lens' (CaseInsensitive v) (Maybe v)
  at k = unCaseSensitive . at (map toLower k )
```

17.22 Missing Values

Return to questions

1. Write an optic which focuses the value at key “first” or, failing that, the value at key “second”.

```
>>> let optic = ???
>>> M.fromList [("first", False), ("second", False)]
      & optic .~ True
M.fromList [("first",True),("second",False)]

>>> M.fromList [("second", False)]
      & optic .~ True
fromList [("second",True)]
```

2. Write an optic which focuses the first element of a tuple iff it is even, and the second tuple element otherwise. Assume each slot contains an integer.

```
>>> let optic = ???
>>> (1, 1) & optic *~ 10
(1,10)

>>> (2, 2) & optic *~ 10
(20,2)
```

3. Write an optic which focuses all even numbers in a list, if none of the members are odd then focus ALL numbers in the list.

```
>>> let optic = (traversed . filtered even `failing` traversed)
>>> [1, 2, 3, 4] ^.. optic
[2,4]

>>> [1, 3, 5] ^.. optic
[1,3,5]
```

4. Fill in the blanks

```
>>> Nothing ^. non "default"
"default"

>>> Nothing & non 100 +~ 7
Just 107

>>> M.fromList [("Perogies", True), ("Pizza", True), ("Pilsners", True)]
      ^. at "Broccoli" . non False
False

>>> M.fromList [("Breath of the wild", 22000000), ("Odyssey", 9070000)]
      & at "Wario's Woods" . non 0 +~ 999
M.fromList
  [ ("Breath of the wild",22000000)
  , ("Odyssey",9070000)
  , ("Wario's Woods",999)
  ]

>>> ["Math", "Science", "Geography"]
      ^. pre (ix 7) . non "Unscheduled"
"Unscheduled"
```

BONUS

```
>>> [1, 2, 3, 4] ^.. traversed . pre (filtered even) . non (-1)
[-1,2,-1,4]
```

17.23 Prisms

Return to questions

1. Which prisms will be generated from the following data declaration? Give their names and types.

```
data ContactInfo =
    Email String
  | Telephone Int
  | Address String String String

makePrisms ''ContactInfo


_Email :: Prism' ContactInfo String
_Telephone :: Prism' ContactInfo Int
_Address :: Prism' ContactInfo (String, String, String)
```

2. Fill in the blanks

```
>>> Right 35 & _Right +~ 5
Right 40

>>> [Just "Mind", Just "Power", Nothing, Just "Soul", Nothing, Just "Time"]
      ^.. folded . _Just
["Mind","Power","Soul","Time"]

>>> [Just "Mind", Just "Power", Nothing, Just "Soul", Nothing, Just "Time"]
      & traversed . _Just <>~ " Stone"
[ Just "Mind Stone"
, Just "Power Stone"
, Nothing
, Just "Soul Stone"
, Nothing
, Just "Time Stone"
]
```

```
>>> Left (Right True, "Eureka!")
      & _Left . _1 . _Right %~ not
Left (Right False, "Eureka!")

>>> _Cons # ("Do",["Re", "Mi"])
["Do", "Re", "Mi"]

>>> isn't (_Show :: Prism' String Int) "not an int"
True
```

3. Write an expression to get the output from the provided input.

```
input = (Just 1, Nothing, Just 3)
output = [1, 3]

>>> input ^.. each . _Just
[1, 3]

---

input = ('x', "yz")
output = "xzy"

>>> ('x', "yz") & review _Cons & _tail %~ reverse
"xzy"
-- or
>>> ('x', "yz") & _2 %~ reverse & review _Cons
"xzy"

---

input = "do the hokey pokey"
output = Left (Just (Right "do the hokey pokey"))

>>> _Left . _Just . _Right # "do the hokey pokey"
(Just (Right "do the hokey pokey"))
```

17.24 Custom Prisms

Return to questions

1. Try to write a custom prism for matching on the tail of a list:

```
_Tail :: Prism' [a] [a]
```

Is this possible? Why or why not?

No, this isn't possible! We don't have enough information to run this prism in reverse. Try to implement `embed` and you'll see why!

2. Implement `_Cons` for lists using `prism`:

```
_ListCons :: Prism [a] [b] (a, [a]) (b, [b])
_ListCons = prism embed match
  where
    match :: [a] -> Either [b] (a, [a])
    match [] = Left []
    match (a : as) = Right (a, as)
    embed :: (b, [b]) -> [b]
    embed (b, bs) = b : bs
```

BONUS

3. Implement `_Cycles` which detects whether a list consists of exactly 'n' repetitions of a pattern. It should behave as follows:

```
-- Find the subsequence which has been repeated exactly times.
>>> "dogdogdog" ^? _Cycles 3
Just "dog"

-- The input isn't exactly 3 cycles of some input (it's 4), so we don't match
>>> "dogdogdogdog" ^? _Cycles 3
Nothing

-- The input is exactly 3 cycles of "a"
>>> "aaa" ^? _Cycles 3
Just "a"

-- The input isn't a cycle at all.
>>> "xyz" ^? _Cycles 3
Nothing

-- We can review to create cycles.
>>> _Cycles 3 # "dog"
"dogdogdog"

-- We can mutate the sequence that's cycled
```

```
>>> "dogdogdog" & _Cycles 3 .~ "cats"
"catscatscats"
```

```
_Cycles :: (Eq a) => Int -> Prism' [a] [a]
_Cycles n = prism' embed match
  where
    match xs
       = let iteration = take (length xs `div` n) xs
          in if (concat $ replicate n iteration) == xs
                then Just iteration
                else Nothing
    embed xs = concat $ replicate n xs
```

- Can you implement a version of `_Cycles` which doesn't depend on a specific iteration count? Why or why not?

No you can't. When reviewing we'd need to know how many times to repeat the focus. If we chose an "infinite" cycle then how could we ever "match"? You could write the function which runs the check, but it would never complete! The only valid implementation of this prism will loop infinitely on a match, and that's not very helpful.

17.25 Prism Laws

Return to questions

1. Implement the following prism and determine whether it's lawful:

```
_Contains :: forall a. Ord a => a -> Prism' (S.Set a) (S.Set a)
```

It should match on sets which contain the provided element! Reviewing adds the element to the set. It should behave something like this:

```
>>> S.fromList [1, 2, 3] ^? _Contains 2
Just (fromList [1,3])

>>> S.fromList [1, 2, 3] ^? _Contains 10
Nothing

>>> _Contains 10 # S.fromList [1, 2, 3]
fromList [1,2,3,10]

>>> _Contains 2 # S.fromList [1, 2, 3]
fromList [1,2,3]
```

Is it lawful? Why or why not?

No, unfortunately it's not lawful. It fails the Review-Preview requirement of the first law because Set insertion is idempotent!

Here's proof:

```
-- We should get what we put in, i.e. [1, 2, 3]
>>> preview (_Contains 3) (review (_Contains 3) (S.fromList [1, 2, 3]))
Just (fromList [1,2])

>>> preview (_Contains 3) (review (_Contains 3) (S.fromList [1, 2, 3])) == Just (S.f\
romList [1,2,3])
False
```

2. Is the following prism lawful? Write out the checks to confirm your suspicions.

```
_Singleton :: forall a. Prism' [a] a
_Singleton = prism' embed match
  where
    match :: [a] -> Maybe a
    match [a] = Just a
    match _ = Nothing
    embed :: a -> [a]
    embed a = [a]
```

Yes! It's lawful. A singleton list is fully described by the `_Singleton` prism and a single element. Write out the checks to convince yourself.

3. Write your own prism which fails the first law!

You're on your own here captain.

17.26 Intro to Isos

Return to questions

1. For each of the following tasks, choose whether it's best suited to a **Lens**, **Traversal**, **Prism**, or **Iso**:

- Focus a Celsius temperature in Fahrenheit
  - Iso: the conversion is reversible
- Focus the last element of a list
  - Traversal: we're selecting a portion of a larger structure which might be missing.
- View a JSON object as its corresponding Haskell Record
  - Prism: We can *always* build valid JSON from a record, but may fail to construct a record from the json.
- Rotate the elements of a three-tuple one to the right
  - Iso: We can reverse it by rotating back to the left or rotating twice more to the right.
- Rotate the elements of a three-tuple one to the right
  - Iso: We can reverse it by rotating back to the left or rotating twice more to the right.
- Focus on the 'bits' of an Int as Bools.
  - Traversal or Prism: Either of `Traversal' Int Bool` or `Prism [Bool] Int` would be reasonable
- Focus on the 'bits' of an Int as Bools.
  - Traversal or Prism: Either of `Traversal' Int Bool` or `Prism [Bool] Int` would be reasonable
- Focusing an `IntSet` from a `Set Int`
  - Iso: The conversion is trivially reversible and lossless.

2. Fill in the blank

```
>>> ("Beauty", "Age") ^. swapped
("Age", "Beauty")

>>> 50 ^. from (adding 10)
40

>>> 0 & multiplying 4 +~ 12
3.0

>>> 0 & adding 10 . multiplying 2 .~ 24
2

>>> [1, 2, 3] & reversed %~ tail
[1, 2]
```

```
>>> (view flipped (++)) [1, 2] [3, 4]
[3,4,1,2]

>>> [1, 2, 3] ^. reversed
[3, 2, 1]

-- BONUS: Hard ones ahead!
>>> import Data.List (transpose)
-- Note: transpose flips the rows and columns of a nested list:
>>> transpose [[1, 2, 3], [10, 20, 30]]
[[1,10],[2,20],[3,30]]

>>> [[1, 2, 3], [10, 20, 30]] & involuted transpose %~ tail
[[2,3],[20,30]]

-- Extra hard: use `switchCase` somehow to make this statement work:
>>> import Data.Char (isUpper, toUpper, toLower)
>>> let switchCase c = if isUpper c then toLower c else toUpper c
>>> (32, "Hi") & _2 . involuted (fmap switchCase) .~ ("hELLO" :: String)
(32,"Hello")
```

3. You can convert from Celsius to Fahrenheit using the following formula:

```
celsiusToF :: Double -> Double
celsiusToF c = (c * (9/5)) + 32
```

Implement the following iso

```
fahrenheit :: Iso' Double Double
fahrenheit = iso celsiusToF fahrenheitToC
  where
    fahrenheitToC f = (f - 32) * (5/9)
    celsiusToF c = (c * (9/5)) + 32
```

17.27 Projected Isos

Return to questions

1. Fill in the blank

```
>>> ("Beauty", "Age") ^. mapping reversed . swapped
("egA","Beauty")

>>> [True, False, True] ^. mapping (involuted not)
[False,True,False]

>>> [True, False, True] & mapping (involuted not) %~ filter id
[False]

>>> (show ^. mapping reversed) 1234
"4321"
```

2. Using the `enum` iso provided by `lens` (type below):

```
enum :: Enum a => Iso' Int a
```

Implement the following `intNot` function, use `dimapping` in your implementation.

```
intNot :: Int -> Int
intNot = not ^. dimapping enum (from enum)
```

Can you simplify your implementation of this function somehow?

```
intNot' :: Int -> Int
intNot' = enum %~ not
```

17.28 Iso Laws

Return to questions

1. The following iso is unlawful; provide a counter example which shows that it breaks the law.

```
mapList :: Ord k => Iso' (M.Map k v) [(k, v)]
mapList = iso M.toList M.fromList
```

This iso doesn't handle duplicate keys in the list properly, here's a small counter-example:

```
>>> view (from mapList . mapList) [('a', 1), ('a', 2)]
[('a',2)]
```

2. Is there a lawful implementation of the following iso? If so, implement it, if not, why not?

Yes, it's lawful:

```
nonEmptyList :: Iso [a] [b] (Maybe (NonEmpty a)) (Maybe (NonEmpty b))
nonEmptyList = iso nonEmpty (maybe [] toList)
```

3. Is there a lawful implementation of an iso which 'sorts' a list of elements? Why or why not?

```
sorted :: Ord a => Iso' [a] [a]
```

No, there isn't. Once you've sorted a list you lose all information about the original ordering and can't return it to the unsorted state!

4. What about the following iso which pairs each element with an Int which remembers its original position in the list. Is this a lawful iso? Why or why not? If not, try to find a counter-example.

```
sorted :: (Ord a) => Iso' [a] [(Int, a)]
sorted = iso to' from'
  where
    to' xs = L.sortOn snd $ zip [0..] xs
    from' xs = fmap snd $ L.sortOn fst xs
```

It's close, but no cigar!

It passes the first half of the law:

```
>>> view (sorted . from sorted) "cab"
"cab"
```

But if you choose your input carefully it will fail the second half!

```
>>> [(9, 'a'), (8, 'b'), (7, 'c')] ^. from sorted . sorted
[(2,'a'),(1,'b'),(0,'c')]
```

It 'normalizes' the list positions, This might be 'close enough' for some purposes, but it's not technically lawful! We've **lost** information about the original list!

3. Design a lawful iso which represents a `Map` as some sort of list.

17.29 Indexed Optics

Return to questions

1. Fill in the blanks

```
>>> M.fromList [("streamResponse", False), ("useSSL", True)]
      ^@.. itraversed
[("streamResponse",False),("useSSL",True)]

>>> (M.fromList [('a', 1), ('b', 2)], M.fromList [('c', 3), ('d', 4)])
      ^@.. both . itraversed
[('a',1),('b',2),('c',3),('d',4)]

>>> M.fromList [('a', (True, 1)), ('b', (False, 2))]
      ^@.. itraversed <. _1
[('a', True), ('b', False)]

>>> [ M.fromList [("Tulips", 5), ("Roses", 3)]
    , M.fromList [("Goldfish", 11), ("Frogs", 8)]
    ] ^@.. itraversed <.> itraversed
[ ((0,"Roses"), 3)
, ((0,"Tulips"), 5)
, ((1,"Frogs"), 8)
, ((1,"Goldfish"), 11)
]

>>> [10, 20, 30] & itraversed %@~ (+)
[10,21,32]

>>> itraverseOf_
      itraversed
      (\i s -> putStrLn (replicate i ' ' <> s))
      ["one", "two", "three"]
one
 two
  three

>>> ["Go shopping", "Eat lunch", "Take a nap"]
      & itraversed %@~ \n s -> show n <> ": " <> s
["0: Go shopping","1: Eat lunch","2: Take a nap"]

>>> itraverseOf_
      itraversed
      (\n s -> putStrLn (show n <> ": " <> s))
      ["Go shopping", "Eat lunch", "Take a nap"]
0: Go shopping
1: Eat lunch
2: Take a nap
```

17.30 Index Filters

Return to questions

1. Given the following exercise schedule:

```
exercises :: M.Map String (M.Map String Int)
exercises = M.fromList
  [ ("Monday"   , M.fromList [("pushups", 10), ("crunches", 20)])
  , ("Wednesday", M.fromList [("pushups", 15), ("handstands", 3)])
  , ("Friday"   , M.fromList [("crunches", 25), ("handstands", 5)])
  ]
```

- Compute the total number of "crunches" you should do this week.

```
>>> sumOf (traversed . itraversed . index "crunches") exercises
45
```

- Compute the number of reps you need to do across all exercise types on Wednesday.

```
>>> sumOf (itraversed . index "Wednesday" . traversed) exercises
18
```

List out the number of pushups you need to do each day, you can use `ix` to help this time if you wish.

```
>>> exercises ^@.. itraversed <. ix "pushups"
[("Monday",10),("Wednesday",15)]
```

2. Given the following board:

```
let board = ["XOO"
            ,".XO"
            ,"X.."]
```

- Generate a list of positions alongside their `(row, column)` coordinates.

```
>>> board ^@.. itraversed <.> itraversed
[ ((0,0),'X') ,((0,1),'O') ,((0,2),'O')
, ((1,0),'.') ,((1,1),'X') ,((1,2),'O')
, ((2,0),'X') ,((2,1),'.') ,((2,2),'.')
]
```

- Set the empty square at `(1, 0)` to an `'X'`. HINT: When using the custom composition operators you'll often need to introduce parenthesis to get the right precedence.

```
>>> board & (itraversed <.> itraversed) . index (1, 0) .~ 'X'
[ "XOO"
, "XXO"
, "X.."]
```

- Get the 2nd **column** as a list (e.g. `"OX."`). Try to do it using `index` instead of `indices`!

```
>>> board ^.. traversed . itraversed . index 1
"OX."
```

- Get the 3rd **row** as a list (e.g. `"X.."`). Try to do it using `index` instead of `indices`! HINT: The precedence for this one can be tricky too.

```
>>> board ^.. (itraversed <. traverse) . index 2
"X.."
```

17.31 Custom Indexed Optics

Return to questions

1. Write a sensible implementation for the following indexed fold:

```
pair :: IndexedFold Bool (a, a)  a
pair = ifolding (\(a, b) -> [(False, a), (True, b)])
```

Such that:

```
>>> ('a', 'b') ^@.. pair
[(False,'a'),(True,'b')]
```

Once you've done that; try writing it as an indexed traversal:

```
pair' :: IndexedTraversal Bool (a, a) (b, b)  a b
pair' handler (a, a') =
    (,) <$> indexed handler False a <*> indexed handler True a'
```

2. Use `reindexed` to provide an indexed list traversal which starts at '1' instead of '0'.

```
oneIndexed :: IndexedTraversal Int [a] [b] a b
oneIndexed = reindexed (+1) traversed
```

It should behave like so:

```
>>> ['a'..'d'] ^@.. oneIndexed
[(1,'a'),(2,'b'),(3,'c'),(4,'d')]
```

If you want a challenge, try using `selfIndex`, `<.>` and `reindexed` to write a traversal indexed by the distance to the **end** of the list. E.g.

```
invertedIndex :: IndexedTraversal Int [a] [b] a b
invertedIndex =
  reindexed
    (\(xs, i) -> (length xs - 1) - i)
    (selfIndex <.> traversed)


>>> ['a'..'d'] ^@.. invertedIndex
[(3,'a'),(2,'b'),(1,'c'),(0,'d')]
```

3. Build the following combinators using only compositions of other optics:

```
chars :: IndexedTraversal Int T.Text T.Text Char Char
chars = indexing each

>>> ("banana" :: T.Text) ^@.. chars
[(0,'b'),(1,'a'),(2,'n'),(3,'a'),(4,'n'),(5,'a')]
```

This one's a thinker; index each character (except newlines) by their line and column numbers:

```
charCoords :: IndexedTraversal (Int, Int) String String Char Char
charCoords = indexing lined <.> indexing each
```

```
>>> "line\nby\nline" ^@.. charCoords
[ ((0,0),'l'),((0,1),'i'),((0,2),'n'),((0,3),'e')
, ((1,0),'b'),((1,1),'y')
, ((2,0),'l') ,((2,1),'i'),((2,2),'n'),((2,3),'e')]
```

17.32 Type Errors

First Foe: Level 1 Lenslion

```
>>> :load
Ok, modules loaded: none.
>>> view _1 ('a', 2)
error:
    Variable not in scope: view :: t0 -> (Char, Integer) -> t

error:
    • Found hole: _1 :: t0
      Where: ‘t0’ is an ambiguous type variable
      Or perhaps ‘_1’ is mis-spelled, or not in scope
    • In the first argument of ‘view’, namely ‘_1’
      In the expression: view _1 ('a', 2)
      In an equation for ‘it’: it = view _1 ('a', 2)
    • Relevant bindings include it :: t (bound at <interactive>:1:1)
      Valid hole fits include: ...
```

Solution:

```
>>> import Control.Lens
>>> view _1 ('a', 2)
'a'
```

This sort of error comes up from time to time when we forget to import something. It's particularly annoying with combinators that start with _ because GHC will assume we meant it as a "type-hole" and will suggest replacements rather than complaining that it's out of scope. Keep an eye out for that one!

Return to question

Level 2 Tuplicant

```
>>> view ("abc", 123) _1

error:
    • Couldn't match type ‘([Char], Integer)’
                     with ‘(t -> Const t t)
                           -> ((a0 -> f0 b0) -> s0 -> f0 t0)
                           -> Const t ((a0 -> f0 b0) -> s0 -> f0 t0)’
      Expected type: Getting t ((a0 -> f0 b0) -> s0 -> f0 t0) t
        Actual type: ([Char], Integer)
    • In the first argument of ‘view’, namely ‘("abc", 123)’
      In the expression: view ("abc", 123) _1
      In an equation for ‘it’: it = view ("abc", 123) _1
    • Relevant bindings include it :: t (bound at <interactive>:1:1)
```

The issue here is simply that we've passed our arguments in the wrong order. The biggest clue that this is the case comes from the lines:

```
      Expected type: Getting t ((a0 -> f0 b0) -> s0 -> f0 t0) t
        Actual type: ([Char], Integer)
```

Which says it was expecting a `Getting`, which is a type of `Getter`, but it got a normal value: `([Char], Integer)`. Why would it be expecting an optic where we've passed it a value? Ahhh, we must have mixed up our argument order!

Solution:

```
>>> view _1 ("abc", 123)
"abc"
```

Return to question

Level 3 Settersiren

```
>>> ("old", False) $ _1 .~ "new"

<interactive>:1:1: error:
    • Couldn't match expected type ‘(s0 -> t0) -> t’
                  with actual type ‘([Char], Bool)’
    • The first argument of ($) takes one argument,
      but its type ‘([Char], Bool)’ has none
      In the expression: ("old", False) $ _1 .~ "new"
      In an equation for ‘it’: it = ("old", False) $ _1 .~ "new"
    • Relevant bindings include it :: t (bound at <interactive>:1:1)
```

Solution:

We've mistakenly used `$` instead of `&`, we just have to swap it in:

```
>>> ("old", False) & _1 .~ "new"
("new",False)
```

Return to question

Level 4 Composicore

```
>>> view (_3 . _1) (('a', 'b', 'c'), 'd')

<interactive>:1:1: error:
    • No instance for
    (Field3 ((Char, Char, Char), Char) ((Char, Char, Char), Char) a0 b0)
```

We've composed our lenses in reverse order! Remember that lenses compose left-to-right, so we need to swap them around:

```
>>> view (_1 . _3) (('a', 'b', 'c'), 'd')
'c'
```

Return to question

Level 5 Foldasaurus

```
>>> ("blue", Just (2 :: Int)) ^. _2 . _Just

<interactive>:1:35: error:
    • No instance for (Monoid Int) arising from a use of ‘_Just’
    • In the second argument of ‘(.)’, namely ‘_Just’
      In the second argument of ‘(^.)’, namely ‘_2 . _Just’
      In the expression: ("blue", Just (2 :: Int)) ^. _2 . _Just
```

Solution:

```
>>> ("blue", Just (2 :: Int)) ^? _2 . _Just
Just 2
```

By switching to `preview` a.k.a. `(^?)` we can handle failure by returning `Nothing` instead. This removes the `Monoid` requirement and will behave correctly in cases where there's a `Nothing` instead of a `Just`.

Return to question

Level 6 Foldasaurus

```
>>> ['a'..'z'] ^.. taking 5 . folded

error:
    • Couldn't match type ‘[]’ with ‘p a’
      Expected type: Getting
                       (Endo
                          [BazaarT p f a a a])
                       [Char]
                       (BazaarT p f a a a)
        Actual type: (BazaarT p f a a a
                      -> s0 -> BazaarT p f a a a)
                     -> Over p f s0 (BazaarT p f a a a) a a
    • Probable cause: ‘(.)’ is applied to too few arguments
      In the second argument of ‘(^..)’, namely ‘taking 5 . folded’
      In the expression: ['a' .. 'z'] ^.. taking 5 . folded
      In an equation for ‘it’: it = ['a' .. 'z'] ^.. taking 5 . folded
    • Relevant bindings include
        it :: [BazaarT p f a a a]
```

Solution:

`taking` is a **higher-order** traversal, that means we need to pass it another optic as an **argument**! Here we're accidentally **composing** the other optic rather than passing it as an argument. Here's the fix:

```
>>> ['a'..'z'] ^.. taking 5 folded
"abcde"
```

Return to question

Level 7 Traversacula

```
>>> over both print ("one", "two")

error:
    • No instance for (Show (IO ())) arising from a use of ‘print’
    • In a stmt of an interactive GHCi command: print it
```

Solution:

If we want to thread effects through a traversal or lens we need to use an action which knows about effects. In this case the right choice is `traverseOf_`:

```
>>> traverseOf_ both putStrLn ("one", "two")
one
two
```

Return to question

17.33 Kubernetes API

Return to questions

1. Your first task should be mostly straightforward: get the api version which was used to make the call.

```
>>> pods ^? key "apiVersion" . _String
Just "v1"
```

2. Next, count the **number** of all **containers** across all **pods**. You can assume that every element of "items" is a pod.

```
>>> pods
    & lengthOf
    ( key "items"
    . values
    . key "spec"
    . key "containers"
    . values
    )
2
```

3. Return the "name" (as `Text`) of all **containers** which have the same value for their "image" and "name" fields.

```
>>> pods
    & toListOf
    (key "items"
    . values
    . key "spec"
    . key "containers"
    . values
    . filtered (\v -> v ^? key "name" == v ^? key "image")
    . key "name"
    . _String
    )
[ "redis" ]
```

4. Set a resource request of `memory: "256M"` for every container.

```
>>> pods
    & key "items"
    . values
    . key "spec"
    . key "containers"
    . values
    . key "resources"
    . key "requests"
    . _Object
    . at "memory"
    ?~ "256M"
```

BONUS Questions

These problems require a bit more thought, and will probably require some unorthodox uses of indexes and filtering. Good luck! Don't get too frustrated, they're deliberately very hard!

5. Get a **Set** of all metadata label **keys** used in the response.

```
>>> foldMapOf
      ( key "items"
      . values
      . key "metadata"
      . key "labels"
      . members
      . asIndex
      )
      S.singleton
      pods
S.fromList
    [ "name"
    , "region"
    ]
```

6. **Set** the `hostPort` to `8080` on any "port" descriptions where it is **unset**.

```
>>> pods
    & key "items"
    . values
    . key "spec"
    . key "containers"
    . values
    . key "ports"
    . values
    . _Object
    . at "hostPort"
    . filteredBy _Nothing
    ?~ _Number # 8080

-- Alternatively, using `below`:
>>> pods
    & key "items"
    . values
    . key "spec"
    . key "containers"
```

```
    . values
    . key "ports"
    . values
    . _Object
    . at "hostPort"
    . filteredBy _Nothing
    . below _Integer
    ?~ 8080
```

7. **Prepend** the **region** to the **name** of each container.

```
>>> pods
    & key "items"
    . values
    . reindexed
          (view (key "metadata" . key "labels" .  key "region" . _String))
          selfIndex
    <. ( key "spec"
       . key "containers"
       . values
       . key "name"
       . _String
       ) %@~ (\region name -> region <> "-" <> name)
```

8. Collect a list of all “containerPort”s alongside their Pod’s “metadata > name”.

```
>>> pods
    & toListOf
    ( ( key "items"
      . values
      . reindexed (view (key "metadata" . key "name" . _String)) selfIndex
      <. ( key "spec"
         . key "containers"
         . values
         . key "ports"
         . values
         . key "containerPort"
         . _Integer
         )
      ) . withIndex)
[
    ( "redis-h315w"
```

```
    , 27017
    )
,
    ( "web-4c5bj"
    , 3000
    )
]
```

9. Uppercase the label values inside each pod's metadata

```
>>> import qualified Data.Text as T
>>> pods
    & key "items"
    . values
    . key "metadata"
    . key "labels"
    . members
    . _String
    %~ T.toUpper
```

Jump to answers

18. Upcoming Chapters

The existing book is a comprehensive guide to optics in Haskell, but there are many additional bits and bobs that it would be nice to write about. The following chapters are still in the works, but will be added to the book as they're finished.

- Classy prisms and their applications towards exception handling
- Uniplate/Biplate combinators
- `failover`
- Generic optics (`generic-lens`)

19. Thanks

Thanks to the many folks who've helped make this book the best it can be!

Special thanks to those who have read and offered helpful feedback throughout the writing process:

- Fintan Halpenny
- Sandy Maguire
- Solomon Bothwell
- TheMatten

19.1 Patreon Supporters

And of course, huge thanks to all of those who helped support me on Patreon leading up to the book's release, I couldn't do it without you! You've played a vital role in making this book happen!

- Adam Bissonnette
- Adam Krauze
- Alex B
- Alexander Nakos
- Allan Lukwago
- Amilcar Blake
- Anatolii Prylutskyi
- Andrew
- Angel Vanegas
- Artem Chernyak
- Arya Irani
- Barry Moore
- Carl Hedgren
- Chessai
- Daniel Tebbutt
- David Burkett
- Diz
- Fabian
- Federico Sordillo
- Fintan Halpenny
- Gabriele Lana
- GhiOm

- Giovanni Ornaghi
- Heneli Kailahi
- Hong Minhee
- James
- Jan Hrček
- Jason Davidson
- Javier Gonzalez
- John A Lotoski
- Jonas De Vuyst
- Jonathan Lorimer
- Jose Iborra Lopez
- Josh Miller
- Juan Manuel Gimeno
- Ken Aguilar
- Kenny Parnell
- Lally Singh
- Lyle Kopnicky
- Marco Faustinelli
- Martin Allard
- Matteo Ricci
- Matthew Mongeau
- Mohan Radhakrishnan
- Myron Wu
- Nathan Pointer
- Nic C.
- Panagiotis Papadakos
- Patryk Zieliński
- Paul Harrington
- Paweł Nosal
- Paweł Szulc
- Peter Murphy
- Philippe Derome
- Proto
- Ramakrishna
- Riccardo
- Rik van der Kleij
- Rohit Ramesh
- Rui Morais
- Sam Raker
- Sandy Maguire
- Serg Nesterov
- Shane Sveller

- Simon
- Solomon Bothwell
- Subra
- Thomas
- Tina Wuest
- Tobias Pflug
- Tyler Augustine
- Tyler Weir
- Vincent Orr
- Vladimir Ciobanu
- Xavier V
- voidfraction

19.2 Book Cover

Thanks to Drew Graham on Unsplash[1] for the beautiful cover photo.

[1] https://unsplash.com/@dizzyd718