Finding Success & Failure

Julie Moronuki

Chris Martin

Finding Success & Failure

Julie Moronuki

Chris Martin

First draft: 2 December 2018

Contents

Preface

We wrote this book because one of the first things people always want to know when they start learning Haskell is what monads are, but it's hard to explain without some background and examples. We also wanted to demonstrate one of the strengths in Haskell: the ability make precise and subtle distinctions between things like `Either` and `Validation`, which give you different behavior with different typeclass instances. By starting with very basic language concepts (`case` and `if-then-else`) and growing a single example very gradually, you can start from very little knowledge of Haskell, and get to `Monad` and `Applicative` by focusing on just one new thing at a time.

The book begins with two chapters on `case` expressions to ensure a solid foundation. From there, we write three functions for checking that inputs are valid passwords according to the rules of our system. The rest of the books iteratively expands on and refactors those functions into a small program that validates usernames and passwords, constructs a `User` (the product of a username and a password) if both are valid inputs, and returns pretty error messages if they are not. Along the way we learn about `Monad` and `Applicative`, how they are similar, how they differ, and how to use types to rethink our solutions to problems.

We assume very little prior knowledge of Haskell. We are focused here on working through examples without understanding theory or how and why things work too deeply.

This book has two intended audiences:

1. You've just started getting into Haskell and are moving from beginner to intermediate level

2. You're not reading this to learn Haskell, but to understand some of the things Haskellers talk about, and to gain some concepts that you can take back to your other programing language of choice.

We encourage you to follow along with the steps that we take in this book, type all of the code yourself, and play with it in the Haskell REPL, GHCi, and do the exercises at the end of each chapter.[1] We will present you with the option of learning to build a Haskell project and compile an executable. The only think you'll need to install is Stack[2] (and Stack will take care of installing the Haskell compiler, GHC, automatically). If you're already comfortable building a project by other means, such as with cabal-install or Nix, then you can still follow along, although we'll assume that you are able to adapt the instructions for your build system of choice.

There are exercises at the end of each chapter. Some are fairly straightforward extensions of what we've just done in the chapter, while others introduce new concepts. In general, they are ordered by difficulty, with the first exercises in the chapter being the most familiar and the last one most likely being the most challenging, probably introducing a new concept or giving you the least amount of help. A few stretch way beyond the current text to introduce entirely new libraries to encourage you to get closer to idiomatic Haskell. You should be able to adequately follow the main body of the text, however, without doing those exercises, so do not feel obligated to complete them all before moving on to the next chapter.

[1] 'REPL' stands for "read-eval-print loop"; you might also know this concept by the name 'interpreter', 'console', or 'shell'.

[2] We use Stack here, but there are other fine build tools you could use if you like.

Chapter 1

Introduction to case expressions

This chapter compares two kinds of expressions in Haskell:

1. Conditional `if-then-else` expressions, which you are probably familiar with from other programming languages
2. `case` expressions, which serve a similar ***branching*** role as conditionals, but with much more generality.

Conditionals and `case` expressions serve a similar purpose; they both allow a function's behavior to vary depending on the value of an expression. However, `case` expressions have some flexibility that `if` expressions do not have, namely allowing behavior to branch on values other than booleans.

1.1 Conditionals

Create a file named `practice.hs`. This file will contain all of the code that we write in this chapter.

Start by typing the following definition for a function, which we have chosen to name `function`.

```
function x y = if (x > y) then (x + 10) else y
```

This word `function` is not a keyword or built-in in Haskell; we're using this generic name because this is a throwaway function (many Haskell tutorials would simply call this `f`, which is the traditional abbreviation for `function`).

This function has two parameters, `x` and `y`. If `x` is greater than `y`, then this function will return `x + 10`; if not, then it will return `y`.

The `if-then-else` pattern in Haskell corresponds to conditional patterns in other languages. It is worth pointing out that the `else` is not optional in Haskell; if you have an `if` you must always define a `then` *and* an `else`. This is because the expression that follows the `if` keyword evaluates to a value of type `Bool`, and we want to make sure we're handling both possible cases: `True` and `False`.[1]

1.2 Reading type signatures

Once you have the function above saved in your `practice.hs` file, you can now open GHCi:

```
$ stack repl
Configuring GHCi with the following packages:
GHCi, version 8.4.3: http://www.haskell.org/ghc/  :? for help
Prelude>
```

And `:load` the file with this function into the REPL:

```
Prelude> :load practice.hs

[1 of 1] Compiling Main    ( practice.hs, interpreted )
Ok, one module loaded.
*Main>
```

The REPL prompt changes from `Prelude>` to `*Main>` to indicate that everything from the module named `Main` is now in scope. We didn't explicitly name our module `Main`, but every module needs a name, and so GHC automatically names your module `Main` if you don't name it yourself.

[1] There is a function `when :: Applicative f => Bool -> f () -> f ()` in the `Control.Monad` module that corresponds to an `if-then` clause in an imperative language, but it will not come up in this book.

Once the module is loaded, you can use the `:type` command to query the type of the function:

```
*Main> :type function
function :: (Ord p, Num p) => p -> p -> p
```

```
      function   ::   (Ord p, Num p)   =>   p -> p -> p
```

The name of the thing whose type we're declaring

Constraints on the type variables

The type

The three `p`s are type variables; this tells us that the function takes two inputs of the same type and returns a value of that same type. We know they have to be the same type since they are all the same letter.[2]

```
      p        ->        p        ->        p
```

Type of the 1st argument

Type of the 2nd argument

Type of the return value

Since the `p`s are variables, the type is not yet determined; it could be any of several types. The type that `p` could be is somewhat constrained, however, by the stuff before the fat arrow, `=>`.

```
( Ord p       ,       Num p )    =>
```

`p` must have an ordering

`p` must be some kind of number

Whatever `p` this is, it must be something that satisfies the constraint `(Ord p, Num p)`, which means it must be a type that has an instance of `Ord` and an instance of `Num`.

`Ord` and `Num` are both ***typeclasses***.

- `Ord` is a typeclass that provides ordering operations, such as `>`, for orderable types, such as numbers and characters.
- `Num` is a numeric typeclass (one of many) that provides operations such as `+` for types of numbers (`Integer`, `Int`, `Float`, and so on).

[2] People sometimes use the phrase ***alpha equivalence*** to refer to this fact. It is unclear to us why GHCi chose `p` here. Often the type variable names you see in the result of the `:type` command are the same names that were used in the source code. In this case, it appears to be somewhat arbitrary.

Since we are using both `>` and `+` with the same arguments, GHC has inferred that any time we use `function`, it must be with a type `p` must be one that has sensible implementations of ordering operations and addition.

The `>` operator takes two parameters of the same ***orderable*** type and returns a `Bool` (either `True` or `False`).

```
*Main> :type (>)
(>) :: Ord a => a -> a -> Bool
```

The expression in the `if` clause of a conditional must always be something that returns a `Bool`. Notice that in our function, the expression in our `if` statement is `(x > y)`. We can see by looking at the type of `>` that this expression does in fact have the type `Bool`.

Because `function` have type parameters, we say that it is ***polymorphic*** – it can take many types. We could assign our function a less polymorphic type signature if we prefer. To the file where you have the `function` definition saved, try adding a type signature:

```
function :: Integer -> Integer -> Integer
function x y = if (x > y) then (x + 10) else y
```

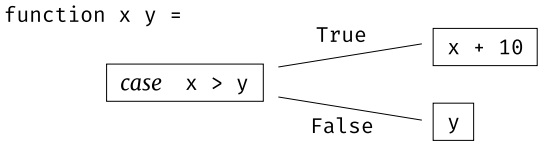
The compiler will usually infer the maximally polymorphic type that it can infer (as above, where it only knew that the type must have instances of `Ord` and `Num` but was otherwise not determined). `Integer` is a type that has instances of both `Ord` and `Num`, though, so there will be no conflict in making our type signature concrete (or ***monomorphic*** – having no type variables) in this case.

`:reload` the file into the REPL and check to make sure it tells you the concrete type now.

```
*Main> :reload
[1 of 1] Compiling Main   ( practice.hs, interpreted )
Ok, one module loaded.
*Main> :type function
function :: Integer -> Integer -> Integer
```

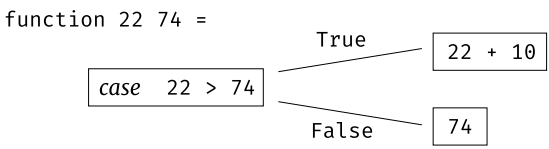
1.3 Branching patterns

Returning to our conditional expression, then, we see that it is a kind of branching pattern. Based on the boolean result in the `if` clause, the function follows one of two paths: when the result is `True`, we take the `then` path; when it is `False`, we take the `else` path.

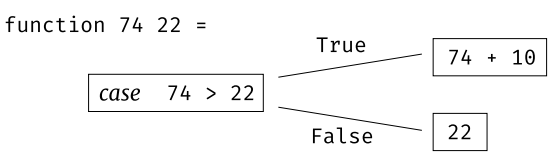


Try it out in your REPL by applying `function` to arguments of various values.

```
*Main> function 22 74
74
```




```
*Main> function 74 22
84
```



Next we'll look at another way of doing the same thing.

1.4 Case expressions

Haskell often has multiple ways of accomplishing a task, and nowhere is this more true than with branching control structures. The one we're going to focus on in this course is called the `case` expression. `case` is a keyword in Haskell and must always be paired with the keyword `of`.

Let's rewrite our `function` example using `case` syntax instead of `if-then-else` syntax. It will start off very similarly, but we'll have to explicitly pattern match on the values of `Bool`.

First, let's make sure we understand the `Bool` type. You can query information about in your REPL using the `:info` command.

```
*Main> :info Bool
data Bool = False | True
```

If you try that in your own REPL, it will give you more information than that. It lists the typeclasses `Bool` has instances of (among these, `Ord` should now be familiar to you, and, yes, that means you can find out if `True` is greater than `False`) and what module each of them is defined in, in case you want to import them or find their documentation in the `base` library.

However, it's the first line that we care about, the line with the datatype declaration for `Bool`. The `Bool` type has only two values (also called its *constructors*), `False` and `True`. Pattern matching on booleans means matching on each of those constructors.

To rewrite `function`, then, we'll match on each of those constructors instead of using `then` and `else` to mark the branching paths. We'll name this function's second incarnation `function2`.

Everything before the equals sign looks the same as it did, but immediately after the equals sign we write `case` and the expression whose result we will branch on, and then `of`. It doesn't matter in this function whether you match on `False` or `True` first.[3]

```
function2 :: Integer -> Integer -> Integer
function2 x y =
    case (x > y) of
        False -> y          -- when False, return y
        True -> x + 10      -- when True, return x + 10
```

[3]There are times when the order of the cases matters; this is not one of those times.

`function` and `function2` do exactly the same thing, but in general, `case` expressions have a flexibility that `if` expressions do not: you can use them to branch on the outcomes of expressions that do not return a `Bool` value. Any expression in an `if` clause ***must*** result in a `Bool` and you can ***only*** branch in one of two ways: `then` when it's `True` and `else` when it's `False`.

Not so with `case` expressions. In the `case` \<***expression***> `of` clause, we have a lot of freedom in the result type of the \<***expression***>. We can do the same sort of pattern matching with the constructors of other types as we did with the values of `Bool` above.

1.5 Sum types

In the next chapter, we'll be practicing with `case` expressions using the `Maybe` type.

```
*Main> :info Maybe
data Maybe a = Nothing | Just a
```

We can see a similarity between `Maybe` and `Bool`: Each has two constructors separated by a pipe. You can read the pipe usually in Haskell as representing some kind of "or".[4] In this case, it represents an ***exclusive or***. A value of type `Bool` is either `False` or `True`, never both. A value of type `Maybe a` is either `Nothing` ***or*** `Just a`, never both. Datatypes like this, having more than one constructor, are called ***sum types***.

There is also an important difference between `Bool` and `Maybe`. `Maybe` takes a type parameter – here (and usually) represented by a variable named `a`. `Maybe` is a parameterized type, also called a ***type constructor***. That means it needs to be applied to a type argument before it's a type.[5] Think of `Maybe a` as a function – but a function over ***types*** instead of a function over ***values***.

We notice that `Nothing` does not contain the `a` of `Maybe a` but `Just a` does. Those `a`s do have to be the same type, so if we applied `Maybe` to `Integer` to make the type `Maybe Integer`, then a value of type `Maybe`

[4] Some related terms you might see for this concept: 'disjunction', 'disjoint union', 'tagged union'.

[5] Sometimes instead of 'type' we say 'concrete type' to emphasize the distinction between types and type constructors.

`Integer` is either `Nothing` or `Just` where could be any value of type `Integer`.

The next chapter gives an extended example of working with `case` expressions and pattern matching on the `Maybe` type.

1.6 Exercises

Exercise 1 – Absolute value The following is an implementation of a function for producing the absolute value of a number, written with an `if-then-else` expression. Rewrite it with `case` expression syntax.

```
absVal :: (Num a, Ord a) => a -> a
absVal x = if (x < 0) then (negate x) else x
```

Note that the parentheses around the expressions in the `if` and `then` clauses are not necessary. We could have written this instead:

```
absVal x = if x < 0 then negate x else x
```

But we sometimes use them anyway. Haskell's keyword and operator precedences are designed to avoid the need for parentheses as much as possible, but there's no harm in adding thoughtful parentheses to make it easier for people to parse your code.

The parentheses around the constraints in the type signature are necessary when there is more than one constraint.

Exercise 2 – (Bool, Bool) The following validation function branches on two conditions, giving four possible results. Rewrite it as a single `case` expression over `(null username, null password)`.

```
validateUsernamePassword :: String -> String -> String
validateUsernamePassword username password =
    if null username
      then (if null password
              then "Empty username and password"
              else "Empty username")
      else (if null password
              then "Empty password"
              else "Okay")
```

The parentheses with the comma in the form `(a, b)` is the syntax for a *tuple*.

`null` is a function from the standard library that has the type `[a] -> Bool` and returns `True` if a list is empty and `False` otherwise.

For example, `(null "", null "hunter")` reduces to `(True, False)`.

Exercise 3 – A question of types In the standard `Prelude`, the `head` and `tail` list functions – for returning just the first element of a list or everything *but* the first element of a list, respectively – are *unsafe*: they throw exceptions on empty lists. Here we give an implementation of the `tail` function that does not:

```
tail' :: [a] -> [a]
tail' [] = []
tail' (x:xs) = xs
```

Note the syntax in the third line, `(x:xs)`. This deconstructs the list into two pieces, the head `x` and the tail `xs`. Having deconstructed it with this pattern, we can return just the part that we want. Later we'll use this syntax again to check each element of a list, recursively.

Consider the `head` function next. In the `tail` function above, we were able to avoid throwing an exception on an empty list by returning an empty list instead. But the following will not compile.

```
head' :: [a] -> a
head' [] = []
head' (x:xs) = x
```

Why doesn't it work?

Exercise 4 – Maybe for safety Write new, *safe* versions of `head` and `tail` that return `Maybe` values, using `Nothing` for the empty list cases.

```
tail' :: [a] -> Maybe [a]
```

```
head' :: [a] -> Maybe a
```

Chapter 2

Case expressions practice

This chapter presents an extended example of using `case` expressions. In particular, we look carefully at an example that pattern matches on `Maybe` values, since the rest of the book assumes you understand that well. The example that we work through in this chapter checks two `String` inputs to determine whether they are anagrams of one another. Additionally, we write a `main` I/O action that prompts for user inputs and uses the anagram-checking function on those inputs.

What we want to get out of this is a program that prompts us to enter two words in the terminal then compares those words to see if they are anagrams. The error messages should tell us clearly if the words are or are not anagrams and also whether we gave any invalid inputs.

2.1 The anagram checker

We begin by writing our primary function that will take two `String` inputs and return a `Bool` value that indicates whether the two strings are anagrams or not.

Start by adding to your `practice.hs` file a type signature that reflects this:

```
isAnagram :: String -> String -> Bool
```

Now we implement this function. It is idiomatic in Haskell to use the plural of `x` and `y` – `xs` and `ys` – to indicate variables that are lists or strings.

```
isAnagram xs ys = _
```

The underscore after the equals sign is called a ***hole*** or a ***typed hole***. Holes are always represented by an underscore alone or suffixed by a name, as in `_a`, `_b`, and the like. Holes can be useful as a development technique, although we're not really going to use them that way in this book. We're using them as placeholders for code we haven't written yet. We could also use `undefined`; we've chosen not to use `undefined` because we generally don't use `undefined` in our real code. Having `undefined` in a program allows it to compile without warnings or errors, so it can throw a runtime exception if you've forgotten to take it out. Typed holes, on the other hand, always give either a warning or an error unless you explicitly opt out. By default, holes in your code throw errors, which means your module won't compile with holes in it. If you would like to use holes but turn the errors into warnings so you may continue to compile the code that doesn't have holes in it, you can use the command `:set -fdefer-typed-holes` in the REPL and continue loading your files, with holes, and interpreting them. There will be warnings, but they'll disappear once we've filled in the blanks.

When you're writing generic functions that might be reused a lot, it can make sense to use maximally generic parameter names. However, we'll go ahead and call our parameters `word1` and `word2`, because these ***aren't*** just generic strings; we want the names to serve as a reminder that these strings are supposed to be words.

```
isAnagram word1 word2 = _
```

We will determine if two words are equal by sorting them and then seeing if the two sorted strings are equal. We can import a `sort` function from a `base` module called `Data.List` to do the sorting for us. Add the import to the top of the file, like this:

```
import Data.List
```

The type `String` is a synonym for `[Char]` (or list of `Char`, with the list represented by the square brackets) so we can use all the functions in `Data.List` with `Strings`.

The type of `sort` is:

```
sort :: Ord a => [a] -> [a]
```

When specialized to lists of `Char` (which we can do because `Char` has an instance of the `Ord` class which defines the way in which characters are ordered), it has the type

```
sort :: [Char] -> [Char]
```

Which is equivalently written as

```
sort :: String -> String
```

This means that we can use `sort` to sort the characters in a string.

Our `isAnagram` implementation sorts each input and uses the `==` operator to compare the sorted results:

```
isAnagram word1 word2 = (sort word1) == (sort word2)
```

You can now `:reload` the file in GHCi and apply it to two strings to make sure it's working:

```
*Main> isAnagram "julie" "eiluj"
True

*Main> isAnagram "julie" "chris"
False
```

2.2 The word validator

We've arbitrarily decided that our anagram checker will only check words that are composed entirely of alphabetic characters; no cute numeric anagrams here, please! And we certainly want to treat empty strings as invalid inputs. So, the next function we write is `isWord`.

`isWord` will take a single `String` input and return a `Maybe String`, allowing us to use `Nothing` for the invalid input cases. Valid inputs will be wrapped in a `Just` constructor, so the output of this function at the term level will be either `Nothing` or `Just`.

Add the type signature for `isWord` to the `practice.hs` file:

```
isWord :: String -> Maybe String
```

We first check if it's an empty `String`. There are different ways you can handle this, but we'll use a `case` expression with the `null` function.

```
null :: [a] -> Bool
```

The `null` function is in the `Prelude` module, so we don't need to import it. It takes a `String` input and will return `True` when the `String` is *empty*. So, in the event that `null word` is `True`, we want to return `Nothing` to represent the rejection of an invalid input.

```
isWord word =
    case (null word) of
        True -> Nothing
        False -> _
```

If the input is *not empty*, we next want to ensure that all characters in the string are alphabetic. For this we'll look to a function from the `Data.Char` module of `base` that checks if a *character* is alphabetic (returning `True` when it is).

Add an import of `Data.Char` to your file. The import list should now look like this:

```
import Data.Char
import Data.List
```

The order of the imports does not matter, but in accordance with tradition we have listed them alphabetically.

Now that we have a function in scope that can check whether a character is alphabetic, we'll fold[1] that over our `String` using a function called `all`.

```
all :: (a -> Bool) -> [a] -> Bool
```

The `all` function is included in the `Prelude`, so it does not need to be imported. It returns `True` when `isAlpha` returns `True` for *all* the characters in the word; it returns `False` if a single character is not alphabetic. Thus, it's the `False` case that is our invalid input, returned as a `Nothing`. If all the characters are alphabetic, then we return `Just word`.

```
isWord word =
    case (null word) of
        True -> Nothing
        False ->
            case (all isAlpha word) of
                False -> Nothing
                True -> Just word
```

[1] You might know the concept of a "fold" in other languages as 'reduce'.

Note the nesting indentations. Haskell is very whitespace sensitive.

So we've pattern matched on `Bool` values to return `Maybe` values, but we promised that we would do some pattern matching on `Maybe` values, so we'll do that next. Our next function will use the two functions we've already written to

1. ***First*** test each of two input strings to see if they are words;
2. ***then*** check to see if those are anagrams.

Step 2 doesn't always take place, because the inputs may get rejected by step 1.

Since our `isWord` function returns `Maybe String`, we will have to pattern match on `Maybe` to handle its two constructors.

2.3 Validate first, then compare

The next function will be called `checkAnagram`. Like the previous function, it will take two `String` inputs. However, this time, instead of just returning a `Bool`, we will return a `String` that is either:

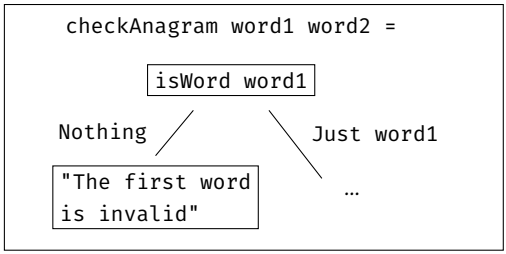
- A message telling us that we have proper anagrams; or
- a message explaining why we do not have proper anagrams.

As always, start by writing the type:

```
checkAnagram :: String -> String -> String
```

We will first test the first input, `word1`, with our `isWord` function. The `isWord` function returns `Maybe` values, so we then pattern match on `Nothing` and `Just word1`.

```
checkAnagram word1 word2 =
    case (isWord word1) of
        Nothing -> "The first word is invalid."
        Just word1 -> _
```



When it returns `Nothing`, then we'll return an error message saying that the first input was invalid. When it returns `Just word1`, then we go into a second `case` expression that applies `isWord` to the other argument, `word2`. A `Nothing` here tells us that the second input is invalid, so we return that message.

```
checkAnagram word1 word2 =
    case (isWord word1) of
        Nothing -> "The first word is invalid."
        Just word1 ->
            case (isWord word2) of
                Nothing -> "The second word is invalid."
                Just word2 -> _
```

If it returns `Just word2`, then we enter yet another `case` expression. This time we're applying `isAnagram` to those two words. Now, `isAnagram` returns a `Bool`, so our final pattern matching is once again on booleans.

```
checkAnagram word1 word2 =
    case (isWord word1) of
        Nothing -> "The first word is invalid."
        Just word1 ->
            case (isWord word2) of
                Nothing -> "The second word is invalid."
                Just word2 ->
                    case (isAnagram word1 word2) of
                        False -> "These words are not anagrams."
                        True -> "These words are anagrams."
```

Try it out in the REPL and then we'll write a nifty `main` executable and do a little input-output.

2.4 Interactive program

In Haskell, the entry point for an executable program must be named `main` and have an `IO` type (nearly always `IO ()`). We do not need to understand this fully right now. In a very general sense, it means that it does some I/O and performs some effects.

We'll use `do` syntax for this, so our definition opens with the `do` keyword which opens what we call a *do-block*. We have some special syntax inside this block, namely the backwards arrow, `<-`, which binds a name to the result of some action. The name goes on the left of the arrow and the action or function resulting in the value that we're naming goes on the right.

```
main :: IO ()
main =
  do
    word1 <- getLine
    word2 <- getLine
    print (checkAnagram word1 word2)
```

We use `getLine`, an `IO String` action, to read user input from the terminal and bind the string they input to a name. The type of `getLine` is

```
getLine :: IO String
```

When we write `word1 <- getLine`, we are in some sense pulling the `String` out of the `IO String` and giving the name `word1` to the `String` that is produced by executing `getLine`.

The final line of our `do`-block says to `print` the result of applying `checkAnagram` to the two inputs.

`:reload` this file into GHCi and type `main` at the prompt. Type a word and hit enter; type a second word and hit enter. This works, but is perhaps a bit awkward; we didn't give it any prompt telling you to type in a word, it just waits for you to figure it out. That isn't very user-friendly, so let's add some prompts telling our "user" what to do. We use `putStrLn` to print some prompt strings to the terminal.

```
main :: IO ()
main =
  do
    putStrLn "Please enter a word."
```

```
    word1 <- getLine
    putStrLn "Please enter a second word."
    word2 <- getLine
    print (checkAnagram word1 word2)
```

Now if you `:reload` the file into GHCi and run `main`, it should look like this:

```
*Main> main
Please enter a word.
```

Try it out!

In the next chapter, we'll get started with the code that we'll be working on for the rest of the course, validating some passwords. There will be a lot of similarities, especially early on, to what we just did here, so hold onto this work!

2.5 Exercises

Exercise 5 – Palindromes Based on the anagrams program we wrote in this chapter, write a short interactive program that checks to see if a word is a palindrome. Palindromes are words that are spelled the same forward and backward.

Since much of the program will be very similar to this one, we will not give much in the way of hints. Instead of using `sort` as we did for the anagram checker, you may want to use the `Prelude` function called `reverse`.

```
*Main> :type reverse
reverse :: [a] -> [a]

*Main> reverse "Julie"
"eiluJ"
```

You may modify the `isWord` function as you think appropriate. For example:

- You may reject the empty string, because although that is sort of a palindrome, it isn't a very interesting one.

- You might want it to simplify the string. Consider that a phrase like "Madam I'm Adam" is usually considered a palindrome. Look at the `Data.Char` module and experiment.

Exercise 6 – Hacker voice In this exercise, we'll write a small program to turn inputs into *leetspeak*. If you don't already know what leetspeak is, it's ok; you can look it up or work only from the guidance we give you here. The idea is that you take a word or phrase and substitute similar-looking characters. It was apparently invented originally to let people write in a way that defeated spam-detection algorithms and the sort of bots that used to kick people out of MSN Chat rooms for cursing. It allows you to type *s3cr3t* messages with your friends and feel like an all-around *k3wl* person.

This program should have at least three things:

1. A function that substitutes similar-looking characters for normal English characters. It can be a mapping from the characters you want to change to the characters they should change into, and you can use a `case` expression.
2. A function that maps that function over an input string.
3. An interactive `main` that lets you type in some input and translate it into *l33tsp3ak*.

For the character substitution function, this should be your type. It's a mapping of characters. So the cases of your `case` expression will be different characters, not Booleans or the like. The *last* case should be an underscore that represents a wildcard or "otherwise" case that returns the character unchanged.

```
substituteChar :: Char -> Char
substituteChar c =
    case c of
        'e' -> '3'
```

For the translation function, let's look at the `Prelude` function called `map`.

```
*Main> :type map
map :: (a -> b) -> [a] -> [b]

*Main> map (+1) [1, 2, 3]
[2,3,4]
```

```
*Main> map isAlpha "juli3"
[True,True,True,True,False]
it :: [Bool]
```

`map` applies a function to each element of a list. The `String` type is a synonym for a list of characters.

```
type String = [Char]
```

So `map` works with `String`, too! Your `substituteChar` function should work on a single character, but you want to translate a whole word or phrase, and `map` can help you do that.

```
translateWord :: String -> String
```

Model your `main` on the interactive `main` programs we have already written.

Chapter 3

Validation functions

Here we begin writing the code that we will stay with through the rest of the book. Over the course of the book, we will be writing validation rules for passwords and usernames, and constructing a user that is a product of the two. We start in this chapter with just passwords.

In this chapter we will write three functions:

1. One checks a `String` for a maximum length.
2. The second verifies that a `String` input is all alphanumeric characters – no special characters allowed in our passwords!
3. The third strips any leading white space from of a `String` input. All three return a `Maybe String`, so invalid inputs are all `Nothing`.

3.1 Project setup

We start a project using Stack:[1]

```
$ stack new validation-book simple
Downloading template "simple" to create project
"validation-book" in validation-book/ ...
```

[1] We use Stack here, but there are other fine build tools you could use if you like.

`validation-book` is the name of the project that we're creating, and `simple` specifies which project template we want to use. "Simple" is, as you might guess, the most minimal of the templates that Stack provides.

When we run this, Stack creates a directory called `validation-book`.

```
$ tree validation-book
validation-book
├── LICENSE
├── README.md
├── Setup.hs
├── src
│   └── Main.hs
├── stack.yaml
└── validation-book.cabal
```

The template includes a BSD3 license,[2] an empty `README.md` file, and a few other build files which we don't need to worry about for the moment. `validation-book.cabal` is the main build file that contains metadata including a description of the package, a list of its modules, and a list of its dependencies. We can leave it as is for now, though we'll come back to it to add a dependency in chapter 8.

What we are immediately concerned with is the source directory (`src`) and the `Main.hs` Haskell source file it contains. We will mostly be editing `Main.hs` throughout the rest of the book.

```
module Main where

main :: IO ()
main = do
  putStrLn "hello world"
```

The module name is `Main` because an executable (in contrast with a module in a library) must always be named `Main`.[3] Likewise, the entry point to the program must be named `main`, and it should have the type `IO`

[2] The `LICENSE` file contains a placeholder where you should insert your name if you are going to use this license.

[3] As we mentioned earlier, `Main` is the module name assigned by default to any source file that doesn't declare any module name, but we suppose the template designer wanted to be explicit about it.

`()`. Here the simple template has given us an implementation of `main` that prints the traditional programmer greeting, "hello world".

Enter the `validation-book` directory that Stack created:

```
$ cd validation-book
```

Then start GHCi:

```
$ stack repl
Configuring GHCi with the following packages:
validation-book
Using main module: 1. Package `validation-book' component
exe:validation-book with main-is file: src/Main.hs
GHCi, version 8.4.3: http://www.haskell.org/ghc/
:? for help
[1 of 1] Compiling Main   ( src/Main.hs, interpreted )
Ok, one module loaded.
*Main>
```

This is the same Haskell REPL that you will get if you run `ghci`, but `stack repl` gives us a "project-aware" REPL that has our project's dependencies (though we have none other than `base`, yet) and an easy mechanism to load our project's code into it. Keep this REPL open all the time while and use it to recompile and experiment the code as you make changes to `Main.hs`.

Run the `main` action in GHCi to verify that it prints "hello world" as it should.

```
*Main> main
hello world
```

3.2 checkPasswordLength

Now that we have a project started, let's write some password-validating code. Our first function will reject a password if it's over a maximum length.

```
checkPasswordLength :: String -> Maybe String
```

The argument to this function is a `String` which will contain a password. The type of the result is `Maybe String`. Recall that `Maybe` has two

constructors:

```
data Maybe a = Nothing | Just a
```

Our `checkPasswordLength` validation function will return `Nothing` when it rejects an invalid password for being too long, and `Just` the password otherwise.

```
checkPasswordLength :: String -> Maybe String
checkPasswordLength password =
    case (length password > 20) of
        True -> Nothing
        False -> Just password
```

Add the definition of `checkPasswordLength` to your `Main.hs` file, and update the `main` action to look like this:

```
main :: IO ()
main =
  do
    putStrLn "Please enter a password"
    password <- getLine
    print (checkPasswordLength password)
```

The new `main`:

1. First prints a prompt ("Please enter a password");
2. Uses `getLine` to get an input, binding the output of `getLine` to a variable called `password`;
3. Applies our `checkPasswordLength` function and prints the result.

Reload the code into GHCi and run `main` again to try it out.

```
*Main> :reload
Ok, one module loaded.

*Main> main
Please enter a password.
julie loves books
Just "julie loves books"

*Main> main
Please enter a password.
julie loves books and hiking
Nothing
```

3.3 `requireAlphaNum`

Now let's write another typical password rule to outlaw whitespace and special characters. This will be a little like the anagrams in the previous chapter; we saw a function called `isAlpha` to check for alphabetic characters. This time we'll expand that to allow numeric characters as well, using `isAlphaNum`, which also comes from the `Data.Char` module.

`requireAlphaNum` has the same type as `checkPasswordLength`, and we'll write another `case` expression that's very similar. We apply `all isAlphaNum` to the string. If it is `False`, then not all of the characters are alphanumeric, so our validation rejects the string and produces `Nothing`. If it is `True`, then we return `Just` the input.

```
requireAlphaNum :: String -> Maybe String
requireAlphaNum password =
    case (all isAlphaNum password) of
        False -> Nothing
        True -> Just password
```

Notice that although we wrote this function for the purpose of validating passwords, it could be used just as well with strings other than passwords (and indeed we will reuse it for usernames in a later chapter). So rather than call the variable `password`, let's rename it to something more generic instead.

As we mentioned earlier, it is a common convention in Haskell to name a local variable `x`, or to name it `xs` (read this as the plural of `x`) when the value is a list. Recall that a string is defined as a list of characters

```
type String = [Char]
```

and so we chose the name `xs` here to remind us of its plural nature.

```
requireAlphaNum :: String -> Maybe String
requireAlphaNum xs =
    case (all isAlphaNum xs) of
        False -> Nothing
        True -> Just xs
```

Change `main` to use `requireAlphaNum` now instead of `checkPasswordLength`:

```
main :: IO ()
main =
  do
    putStrLn "Please enter a password"
    password <- getLine
    print (requireAlphaNum password)
```

Then reload, and try it out again:

```
*Main> :reload
Ok, one module loaded.

*Main> main
Please enter a password.
julie loves books
Nothing

*Main> main
Please enter a password.
julielovesbooks
Just "julielovesbooks"
```

The first test gives us `Nothing` because we entered a password that contains spaces; `all isAlphaNum "julie loves books"` is `False`.

3.4 `cleanWhitespace`

Next we'll write something a bit more complex, and this one will involve recursion. This aim of this function is to enforce a requirement that a valid password must begin with a non-whitespace character. So it does two things:

1. Strip any leading whitespace out of the password.
2. If the resulting string is empty, reject it as invalid.

The type is same as the other two.

```
cleanWhitespace :: String -> Maybe String
cleanWhitespace "" = Nothing
cleanWhitespace (x : xs) =
    case (isSpace x) of
```

```
        True -> cleanWhitespace xs
        False -> Just (x : xs)
```

This time, we have a separate case for the empty string `""`. This is critical because it is a base case for our recursion.

We have to do something different than we did with the previous two functions; we have to deconstruct the list of characters as `x : xs`. The `:` operator (sometimes pronounced "cons") is what builds a list, and it's also what we use to deconstruct a list into its head `x` and tail `xs`. With `checkPasswordLength` and `requireAlphaNum`, we were passing the entire string into another function, and so we did not have to deconstruct the list into its constituent parts. But in this case, we need to check for whitespace ***character by character*** to eliminate leading white spaces, so we must pass the string in one element at a time.

We test the first character of the string, `x`, with `isSpace` function, another function from the `Data.Char` module in `base`. If it ***is*** a whitespace character, then it is a character that we want to omit in the output. So when we issue a recursive call to `cleanWhitespace`, it is over just the tail of the list `xs`, discarding that first character.

For example, when `checkWhitespace` is applied to the string `" ab"`:

1. ```
   checkWhitespace " ab"
   ```
2. ```
   checkWhitespace (' ' : "ab")
   ```
3. ```
   case (isSpace ' ') of
       True -> cleanWhitespace "ab";
       False -> _
   ```
4. ```
   cleanWhitespace "ab"`
   ```
5. ```
   checkWhitespace ('a' : "b")
   ```
6. ```
   case (isSpace 'a') of
       True -> _
       False -> Just ('a' : "b")
   ```
7. ```
   Just ('a' : "b")
   ```
8. ```
   Just "ab"
   ```

However, that function call will again deconstruct the remaining list into its head and tail and check just the head to see if it is whitespace. It will

keep checking whether the first character is a space, and each time it's true it will call itself again.

Always think about the termination condition when you write a recursive function! Recursion requires some *basis*, the point at which it stops. Each recursive call must be making some progress toward a base case, or else you may be writing an infinite loop. There are two ways the evaluation of this function can terminate:

1. If it ever hits the empty case, it will return `Nothing`, because there is no non-whitespace character. This is the basis for our recursion and also has the effect of rejecting empty strings as invalid passwords.
2. If it reaches a point where `isSpace x` is `False`, then it will return `Just (x : xs)` – the `x` cons'ed back together with the `xs`, giving us back the input string that we now know does not begin with whitespace.

Later we'll combine all of these functions, but for now just change `main` again so that we can test the `cleanWhitespace` function.

```
main :: IO ()
main =
  do
    putStrLn "Please enter a password"
    password <- getLine
    print (cleanWhitespace password)
```

```
Please enter a password.
        julie
Just "julie"

Please enter a password.

Nothing
```

3.5 Exercises

Exercise 7 – Minimum length We have a maximum length for the passwords, but what about a *minimum* length? Are we going to let people use "123" as their password? Nope!

Add a minimum length to the `checkPasswordLength` function. You do not have to write a whole new function; you can add it to the `case` expression that is already there. You want to reject strings that are shorter than 10 characters and strings that are longer than 20 characters. In English, your expression will read something like, "case x is less than 10 or greater than 20 of" – check out the `Prelude` Boolean function `(||)` to help you.

Exercise 8 – Combine them all So now we have three functions of type `String -> Maybe String`:

1. `checkPasswordLength`
2. `requireAlphaNum`
3. `cleanWhitespace`

The goal next will be to combine these into a function which, given one string input, will apply all three validation functions to see whether it is a valid password according to ***all*** of the rules.

Notice that `cleanWhitespace` is special among these three, because it is the only one that might transform its input. The other two only validate it and either reject it or return what they were given; they don't change the input in any way.

When we combine these into one function, we'll want `cleanWhitespace` to happen ***first***. Then the ***result*** of `cleanWhitespace` will get passed through `requireAlphaNum` and `checkPasswordLength`, and at the end we return either `Just` that string or a `Nothing`. This is important because, for example, take the input `"  julie"`:

- If you apply `requireAlphaNum` first, it will reject the input because it contains spaces.
- If you apply `cleanWhitespace` first, it will produce the result `Just "julie"`. Then when you feed that result `"julie"` into `requireAlphaNum`, the validation will succeed.

In the anagrams chapter, we showed how to combine functions like these using nested `case` expressions. Try to do that again here: Write a function that uses nested `case` expressions to apply `cleanWhitespace`, then apply `requireAlphaNum` to the result, and then apply `checkPasswordLength`.

We'll discuss the solution in the next chapter.

Exercise 9 – Vacuous truth What does `requireAlphaNum ""` return? Why?

Chapter 4

The Maybe Monad

At the end of the previous chapter, we left you with the task of writing one function that combines all three of our smaller password validation functions. Here we'll look at one possible solution using nested `case` expressions. Then we'll introduce an operator called "bind", `>>=`, that can be used to reduce the laborious `case` pattern matching and make a more concise expression. Since `>>=` is the principle operator of the `Monad` typeclass, we will discuss what monads are.

Our solution to the exercise will have one drawback: It won't tell you *which* of the validation rules failed. We'll fix that in the next chapter.

4.1 Combining the validation functions

As we discussed earlier, we want to start with `cleanWhitespace` so that we can then feed its possibly-transformed output into the other functions.

```
validatePassword :: String -> Maybe String
validatePassword password =
    cleanWhitespace password
```

It doesn't matter whether you handle `requireAlphaNum` or `checkPasswordLength` next. We'll pick `requireAlphaNum`.

You might have tried writing something like this:

```
validatePassword :: String -> Maybe String
validatePassword password =
    requireAlphaNum (cleanWhitespace password)
```

But the types don't line up:

```
error:
  • Couldn't match type ‘Maybe String’ with ‘[Char]’
    Expected type: String
      Actual type: Maybe String
```

Assuming that `password` argument is a `String`, then it can be the input for `cleanWhitespace`, but `cleanWhitespace` returns a `Maybe String` and `requireAlphaNum` cannot accept a `Maybe String` as input. The types do not match up.

Instead we need to handle the `Maybe String` value the same way we did with the output of `isWord` in the anagram checker in chapter 2. In the `Just` branch, we'll have an unwrapped `String` value that we can pass to `requireAlphaNum`.

```
validatePassword :: String -> Maybe String
validatePassword password =
    case (cleanWhitespace password) of
        Nothing -> Nothing
        Just password2 ->
            requireAlphaNum password2
```

We're not done yet, but let's unpack what we've done so far. `cleanWhitespace` returns a `Maybe String` so the values we are pattern matching on are `Maybe String` values. We introduced a new variable, `password2`, because after the application of `cleanWhitespace`, the result we get is a *different string*.

For the last check, we do the same thing again:

```
validatePassword :: String -> Maybe String
validatePassword password =
    case (cleanWhitespace password) of
        Nothing -> Nothing
        Just password2 ->
            case (requireAlphaNum password2) of   -- (1)
                Nothing -> Nothing                -- (2)
```

```
                Just password3 ->                -- (3)
                    checkPasswordLength password3
```

1. Apply the validation function to the password (which is `password2` this time)
2. When the result is `Nothing`, then `requireAlphaNum` rejected the password, so our new function `validatePassword` likewise returns `Nothing` to reject the password also.
3. When the result is `Just` a password, then we'll keep going again onto the third validation rule.

If you were really into the spirit of `case` expressions, you might have kept going and written a `case` for the result of `checkPasswordLength`:

```
validatePassword :: String -> Maybe String
validatePassword password =
   case (cleanWhitespace password) of
      Nothing -> Nothing
      Just password2 ->
         case (requireAlphaNum password2) of
            Nothing -> Nothing
            Just password3 ->
               case (checkPasswordLength password3) of  -- (1)
                  Nothing -> Nothing                    -- (2)
                  Just password4 -> Just password4      -- (3)
```

1. Apply the validation function.
2. If the result is `Nothing`, reject the password.
3. If the result is `Just`, then validation has completed successfully. So `validatePassword` returns `Just password4`, signifying that `password4` is a valid password that has been through the entire gauntlet of checks.

That last pattern match, over `checkPasswordLength password3`, is okay, but it isn't really necessary. You can tell because, in both of its cases, the pattern (the part the left of the `->`) is exactly the same as the expression (the part to the right of the `->`). `Nothing` maps to `Nothing`, and `Just password4` maps to `Just password4`; in other words, nothing changes. So we can simplify the code by removing that last `case`.

4.2 De-nesting with infix operators

As we've seen, nesting `case` expressions gives us a way to perform sequential evaluation of function applications.

1. Clean whitespace.
2. Require alphanumeric characters.
3. Check length.

This may not be obvious, but since Haskell is a lambda calculus, sequencing function applications can only really be done through nesting like this.

- Clean whitespace; then
  - Require alphanumeric characters; then
    * Check length.

If we are nesting a lot of expressions, this can get tedious and repetitive. This is why Haskellers often prefer infix operators whose effect is similar to nesting but without the need for the human to think through the positions of parentheses and the numbers of layers.

For example, when we're concatenating a series of values:

```
f :: String -> String -> String
f x y = mappend x (mappend ": " y)
```

```
*Main> f "Hello" "world"
"Hello: world"
```

We don't want to think of that as a nested expression, particularly not here since these operations are associative and the sequence of events doesn't matter as long as the operands all stay in the same order. So we'd much rather write this using the infix operator `(<>)`.

```
f :: String -> String -> String
f x y = x <> ": " <> y
```

This pattern of de-nesting usually involves an infix operator, especially when there is an associative operation involved. Sometimes a downside of using infix operators is that the reader of your code has to know the associativity of the operator. But when the operation is associative, this worry fades away because it doesn't affect the meaning of the code. You might easily forget whether `a <> b <> c` signifies `(a <> b) <> c` or `a <> (b`

`<> c`), but that's okay because the associative law promises that each of those expressions produces the same result.[1]

For another example, if we have some function like this:

```
f :: String -> String
f x = drop 2 (map toUpper (reverse x))
```

```
*Main> f "eilujab"
"JULIE"
```

Then we may prefer to rewrite it using the infix function composition operator `(.)`:

```
f :: String -> String
f x = (drop 2 . map toUpper . reverse) x
```

The `(.)` operator doesn't help us in our validation code because it requires each link in the chain to take as input the type that was output by the next function.

```
*Main> :type (.)
(.) :: (b -> c) -> (a -> b) -> a -> c
```

The output of `reverse` is a `String`, and the input of `map toUpper` is a `String`, so in the above example the types match up.

But in our validation code, the output of `cleanWhitespace` is a `Maybe String`, and the input of `requireAlphaNum` is a `String`. The `b` returned by the `(a -> b)` function has to be the same `b` as the `b` of the `(b -> c)` function; it can't be a `String` in one place and a `Maybe String` in another. So the types don't match up – at least, not in this way.

4.3 Enter the monad

Behold! the bind operator:

```
*Main> :type (>>=)
(>>=) :: Monad m => m a -> (a -> m b) -> m b
```

We're going to rewrite the `validatePassword` function using `>>=`.

[1] The only time it ***does*** become important how associative expressions parenthesize is when you are thinking about performance.

```
validatePassword :: String -> Maybe String
validatePassword password =
    cleanWhitespace password
        >>= requireAlphaNum
        >>= checkPasswordLength
```

Now we have all three functions in sequence, and the result of `cleanWhitespace password` gets passed to the next two functions. It feels like magic, but it's a monad.

If we ignore the `Monad m` constraint for the moment, we see that a value `m a` can get passed to a function that needs an `a` input. With normal function composition, those types wouldn't match – `a` and `m a` are not the same type. But `(>>=)` gives us a context in which this composition works; how it works depends on the monad.

The `Monad` constraint is important: it tells us that the types `m` have to be monads. That is, they have to be type constructors like `Maybe` that have an instance, or implementation, of this function `>>=`. That implementation defines how to apply a function `(a -> m b)` to a value `m a` and come out with an `m b` at the end. The piece of code defining how a generic function like `(>>=)` works for a specific type (in this case, for a specific monad, that is `Maybe`) is called an `instance` declaration.

4.4 TypeApplications

There is a language extension that we keep turned on in our REPL most of the time that is extremely useful for learning about functions like `>>=`. It's called `TypeApplications`. You can turn it on directly in your REPL like this:

```
*Main> :set -XTypeApplications
```

You can also turn it on by adding it to the top of a file (above the `module` name) that you are compiling with GHC or loading into GHCi using a `LANGUAGE` pragma like this:

```
{-# LANGUAGE TypeApplications #-}
```

`TypeApplications` allows you to specify what type you will apply a polymorphic function to and see the resulting type. So, for example, in the REPL, we can do this:

```
*Main> :type (>>=) @Maybe
(>>=) @Maybe :: Maybe a -> (a -> Maybe b) -> Maybe b
```

The `@Maybe` is the type application, so what we're doing is querying the type signature of `>>=` when the first type argument, the `m`, is `Maybe`. And then we can see that using this operator we can pass a `Maybe a` value to a function that needs an `a` rather than a `Maybe a` as input and returns a `Maybe b`.

Now, we know already that our `a` and `b` values are both `String`. We can further specialize this in the REPL if we like, still using `TypeApplications`.

```
*Main> :type (>>=) @Maybe @String
(>>=) @Maybe @String
  :: Maybe String -> (String -> Maybe b) -> Maybe b

*Main> :type (>>=) @Maybe @String @String
(>>=) @Maybe @String @String
  :: Maybe String -> (String -> Maybe String) -> Maybe String
```

Now the signatures are getting quite long, but we can see that this will let us pass `Maybe String` arguments to `String -> Maybe String` functions and return a `Maybe String`, which is what we were looking for.

4.5 Cases and binds

So, we can use `>>=` to do for us what those nested `case` expressions were doing. It "knows" how to pass a `Maybe a` to an `(a -> Maybe b)` function because of the piece of code called an `instance` that defines this function for the `Maybe` type. A type can have at most one instance of a typeclass, so instance resolution is implicit and straightforward; if the compiler knows what type `m` is, then it knows what class instance to use. Furthermore, because it "knows" about the `Maybe` type, we don't have to explicitly handle the failure cases.

We saw how `case` expressions are a way of branching conditionally on the output of a function, like `if-then-else` expressions and other branching control structures. Using `>>=` is quite a bit like that under many circumstances, but they are easier to chain together into sequences of function

applications. By "easier" we mean that they involve less repetitive reading and writing, allowing the reader or writer to focus on the *functions* that are being chained together rather than the many possible outcomes. This is "easier" when the possible outcomes and the desired behavior at each branch is uniform.

- In the `checkAnagram` function in Chapter 2 , we handled `Nothing` differently each time, because each `Nothing` corresponded to a different error message.
- In the validation code, we do the same thing every time: If it's a "falsey" value, stop evaluating; if it's a "truthy" value, apply the function and pass the result of that function application on to the next function in the chain.

Using `>>=` with `Maybe` here means we lost the ability to tell which `Nothing` we returned and, thus, what our error is. Sometimes short-circuiting evaluation and returning any `Nothing` is all you want, and the `Maybe` monad suffices. However, we might hope that we can harness the benefits of using the monadic interface while still being able to return helpful error messages. As we continue to refactor this code, we'll look at types that can help.

4.6 Exercises

Exercise 10 – Doing I/O `Monad` is a typeclass. A typeclass represents a set of generic functions that work with a set of types; `Monad` defines the types of some functions like `(>>=)` and we can write implementations of that function for a variety of types. `Maybe` is one such monadic type; we will also work with the `Either` monad in this book. Thinking about types as instances of a certain typeclass allows you to focus on the similarities between the types.

```
*Main> :type (>>=)
(>>=) :: Monad m => m a -> (a -> m b) -> m b
```

It's time to address a fact about `IO` in Haskell: it's a monad.

```
*Main> :set -XTypeApplications

*Main> :type (>>=) @IO
(>>=) @IO :: IO a -> (a -> IO b) -> IO b
```

Binding an `IO a` value to a function that takes an `a` and produces another value we call `IO b` allows us to sequence side-effecting `IO` actions. It is very common to write that with `do` blocks, as we have been doing.

Haskell's `do` syntax is intended to allow something like imperative-style programming that ignores the monadic underpinnings. Let's look at how to "desugar" a `do` block into monadic binds.

```
main :: IO ()
main =
  do
    password <- getLine
    print (cleanWhitespace password)
```

You may notice that since `getLine` gives us an `IO String` and `cleanWhitespace` needs a `String` input, the type of this `password` variable we introduced must be `String`. `print` then takes a `String` (or some other value, as long as it is a type with a `Show` instance, `Show` being the typeclass for representing things as printable strings) and returns an `IO ()` – which means it performs an effect, namely printing.

The above could be rewritten with `(>>=)`.

```
main :: IO ()
main = getLine >>= \password -> print (cleanWhitespace password)
```

Or even more cleanly without explicit argument passing, using function composition.

```
main :: IO ()
main = getLine >>= (print . cleanWhitespace)
```

In practice, there are times when Haskellers write it this way and times when `do` syntax is much nicer. This is comparable to noting that sometimes we use `case` expressions and sometimes `(>>=)` to accomplish the same task. However, `case` expressions aren't really available for the `IO` type as you can't pattern match on its constructors the way you can with `Maybe`.

Your exercise is to translate the following into `do` syntax.

```
reverseLine :: IO ()
reverseLine = getLine >>= (print . reverse)
```

Exercise 11 – The bind function for Maybe Implement this function:

```
bindMaybe :: Maybe a -> (a -> Maybe b) -> Maybe b
```

A clever reader might note that this is the type of `(>>=)` specialized to `Maybe`, and so you could just write it like this:

```
bindMaybe = (>>=)
```

But pretend you don't have `(>>=)` available.

Note: If you get a type error like this:

```
• Occurs check: cannot construct the infinite type:
        b ~ Maybe b
```

And you're not sure what it means, check the solutions at the back of the book.

Exercise 12 – The bind function for "StringOrValue" Here's a data type that we've invented for the sake of this exercise:

```
data StringOrValue a = Str String | Val a deriving Show
```

Implement this function:

```
bindStringOrValue
    :: StringOrValue a
    -> (a -> StringOrValue b)
    -> StringOrValue b
```

Chapter 5

Refactoring with Either

We've come a long way already in our code, but some things have been dissatisfying. We have no way to distinguish between different errors. They are all `Nothing`; if the final result is `Nothing`, we don't know which validation function originally produced that `Nothing`.

5.1 Adding error messages

Let's try a straightforward way to rewrite `validatePassword`. If a validation rule fails, then we'll return a `String` with an error message, and if all the validation functions pass, then we'll return the password.

We always have some string to return now; sometimes it's an error message, and sometimes it's a password, but either way it's a string. So the return type is now `String` instead of `Maybe String`, because we've gotten rid of that pesky case in which the function returns `Nothing` which gives us insufficient feedback.

```
validatePassword :: String -> String
validatePassword password =
   case (cleanWhitespace password) of
      Nothing -> "Your password cannot be empty."
      Just password2 ->
```

```
        case (requireAlphaNum password2) of
           Nothing -> "Your password cannot contain \
                      \white space or special characters."
           Just password3 ->
              case (checkPasswordLength password3) of
                 Nothing -> "Your password cannot be \
                            \longer than 20 characters."
                 Just password4 -> password4
```

This time instead of returning the uninformative `Nothing` value, we return messages that are explicit about each function's failure mode, or, in the case of a successful password validation, we return the password string itself.

Reload the project in GHCi and try it out.

```
*Main> validatePassword "   "
"Your password cannot be empty."

*Main> validatePassword "Hello!"
"Your password cannot contain white space or special characters."

*Main> validatePassword "joy123"
"joy123"
```

Now when a password in invalid, we can see which place it failed and why.

So this is an improvement in some ways, but it is worse in others. In particular, this `String -> String` type signature is worrisome. Aside from the general problems that strings can have,[1], we want our error messages and our good data to be distinguishable. We'd like to have a clean and safe separation of failures from successes.

That inability to programmatically distinguish between success and failure values means we had to go back to using nested case expressions; we can't write this with `>>=`. So there's a lot of repetition in explicitly matching on every case through a series of function applications, and we might hope that there's a way to get rid of some of that repetition.

[1] If you're frustrated that all the types are `String`s and you can't tell what they signify at the type level, hang in there – we'll address that in the next chapter.

5.2 Introducing `Either`

Let's take a look at the `Either` type.

```
*Main> :info Either
data Either a b = Left a | Right b
```

This is the definition of the `Either` type in `base`. It is a *sum type* like `Maybe` (and `Bool`), with the pipe `|` representing "or." But it differs from `Maybe` in that it has *two* parameters, `a` and `b`:

- The `Right b` constructor is similar to `Just a` from the `Maybe a` type.
- The `Left a` value is where this really differs from `Maybe` because instead of merely `Nothing`, this `Left` constructor can now carry more information around with it because it can wrap a value of type `a`.

5.3 The `Either` Monad

It is worth pointing out up front that the `>>=` for `Either` works similarly to the `>>=` for `Maybe`, more so than you might initially expect. This is because a monad (in Haskell at least) must be a unary type constructor – that is, a type constructor with a single parameter.

`Maybe a` only has one type parameter, but `Either a b` has two:

```
*Main> :set -XTypeApplications

*Main> :type (>>=) @Maybe
(>>=) @Maybe :: Maybe a -> (a -> Maybe b) -> Maybe b

*Main> :type (>>=) @Either
error:
    • Expecting one more argument to ‘Either’
      Expected kind ‘* -> *’, but ‘Either’ has kind
          ‘* -> * -> *’
```

So to make the instance work, we have to partially apply `Either` to its leftmost parameter.

```
*Main> :type (>>=) @(Either String)
(>>=) @(Either String) :: Either String a ->
```

```
    (a -> Either String b) -> Either String b
```

However, we don't need to care about what type the first parameter of `Either` is applied to, since our monadic functions won't touch it. Thus we can also query the type of `Either`'s `(>>=)` by applying it to a wildcard, represented by an underscore. We can use underscores in this way, or as the final "catchall" match of a `case` or pattern match, when we don't care what the value is. This says, "tell me the type of `(>>=)` applied to `Either` that has been applied to some type that we don't care about".

```
*Main> :type (>>=) @(Either _)
(>>=) @(Either _) :: Either w a ->
    (a -> Either w b) -> Either w b
```

Let's take a quick look at what the `Monad` instance for `Either` looks like in `base`. The instance for `Maybe` is also repeated here for comparison.

```
instance Monad (Either e)
  where
    Left  l >>= _    = Left l
    Right r >>= f    = f r
```

```
instance Monad Maybe
  where
    Nothing  >>= _   = Nothing
    (Just x) >>= f   = f x
```

This should remind you of the `StringOrValue` type we made in the exercises of the previous chapter and the `bindStringOrValue` function you wrote. That was the same as an `Either` that has already been concretely applied to the `String` type on the left.

Although it's called `e` here, it's the same, leftmost type parameter that we call `a` in `Either a b`. The first line of the `instance` declaration tells us that the `Monad` instance is for `Either e` – not for `Either`! One way to think of it is that the leftmost type parameter is fixed; whatever type it is, whatever value we have there, it cannot be transformed. Its type is fixed as `e`, whatever `e` turns out to be.

The `e` in the context of this instance definition is a *rigid type variable*. The concrete type that it takes is determined by the function caller; the class instance has no knowledge or control over what `e` may be, and so we know that the implementation of `>>=` must work for all types `e` and cannot *do*

anything with the `e` values. It is an untouchable value as far as our bind operation is concerned.

When we speak of `Either a b` being monadic, we are saying that `Either a` has a `Monad` instance; the `b` is the type variable that the monad is free to play with, and the `a` just tags along.

Let's review the type of `>>=`:

```
(>>=) :: Monad m => m a -> (a -> m b) -> m b
```

The first argument is a value of type `m a` and the second argument is a function that can be applied to an `a`. If the `m a` value is a `Left l` value, then it isn't available for use as the input value of a function because its type is fixed as part of `m`. Hence, we can only return the `Left l` value.

However, if the `m a` is a `Right r` value and we bind it to a function `f`, then we apply the `f` function to the `r` value. So, when we said that it acts similarly to the `Maybe` instance, we mean that `>>=` is contractually obligated to treat a `Left a` value the same as it treats `Nothing` and applies a function to the `Right b` value the same as it does for a `Just a` value.

Because `Maybe` and `Either` are so similar, both in their structure and in how their typeclass instances (such as their `Monad` instances) work, refactoring from `Maybe` to `Either` is often straightforward. That's what we'll be working on here.

5.4 Using `Either`

So, now we will be using a datatype that allows us to keep track of two bits of data, one on the `Left` and one on the `Right`. While we will continue using `String` for both our error messages and our password strings, that will at least give us a way to separate them and be able to return error messages instead of the cold void of `Nothing`.

Begin refactoring by changing the names of the types and the constructors.

1. The `Maybe String` in type signatures becomes `Either String String`;
2. `Nothing` becomes `Left` *<string>*; and

3. `Just` <*string*> becomes `Right` <*string*>.

```
checkPasswordLength :: String -> Either String String
checkPasswordLength password =
    case (length password > 20) of
        True -> Left "Your password cannot be longer \
                     \than 20 characters."
        False -> Right password
```

We can already see that using `Left` is going to allow us to pass much more information along and be able to tell our users (and remind ourselves) exactly what problem we encountered.

We'll need to rewrite the `requireAlphaNum` and `cleanWhitespace` functions as well to use `Either`. For now, we give the types; one of your exercises for this chapter will be to rewrite them on your own.

```
requireAlphaNum :: String -> Either String String
```

```
cleanWhitespace :: String -> Either String String
```

Now we need to make the type of `validatePassword` match the types of the functions that we have bound together here.

```
validatePassword :: String -> Either String String
validatePassword password =
    cleanWhitespace password
        >>= requireAlphaNum
        >>= checkPasswordLength
```

Interestingly, we don't have to change the implementation of `validatePassword` at all. This is because the only function we used was the polymorphic `>>=` operator, which we can use in exactly the same way regardless of whether we're using `Maybe`, `Either`, or anything else with a `Monad` instance.

We also do not have to change `main`. Even though the return type of `validatePassword` has changed, it's fine because all we were doing was applying the `print` function to it – and `print` is also polymorphic. It works with anything that has a `Show` instance, so it can print an `Either String String` just as well as a `Maybe String`.

5.5 Exercises

Exercise 13 – Converting to Either Earlier we said that we'd need to rewrite `requireAlphaNum` and `cleanWhitespace` to use `Either`. That way, the types of all these functions match and we can easily bind them together with `(>>=)`. Your first task is to rewrite them now, on the model of how we rewrote the `checkPasswordLength` function.

```
requireAlphaNum :: String -> Either String String
```

```
cleanWhitespace :: String -> Either String String
```

Exercise 14 – REPL play Load what you have so far into your REPL and try it out on various input strings.

1. Come up with an input that fails multiple validation rules. Which error message do you get?
2. Change the order of `requireAlphaNum` and `checkPasswordLength` within `validatePassword` and see which messages we get back for which failures.

Remember that you have to use the `:reload` (or abbreviated `:r`) command in GHCi to load the new code when you change the `Main.hs` file.

Noticing where the computation stops and why is an important part of understanding monads and will contrast clearly with how applicative functors work in a later lesson.

Exercise 15 – Testing Playtime is over, now let's get serious. When you write a real program, you're probably going to want to write some tests.

Testing doesn't always have to be complicated or use a framework. We just want to automate what you've already been doing in the REPL, so you can test a lot of examples at once instead of typing them over and over. So we'll write an `IO ()` action that includes some examples of using our validation functions and prints something to let us know if any of the results don't match up with what we expect.

We can use `Either` to represent test results, just like we used it to represent the result of validating user input strings! In this case, the type we'll use is `Either String ()`.

- The `Left` constructor representing failure contains a `String` with an error message, just like before.
- For successful tests, there's no information we need to convey, so the type for the `Right` constructor is `()`. This is similar to how we use `IO ()` for I/O actions like `print` that don't produce a result.

We'll give you a tiny "test framework" to work with, consisting of the following two functions `printTestResult` and `eq`.

The `printTestResult` function takes a test result and prints it nicely to the terminal.

```
printTestResult :: Either String () -> IO ()
printTestResult r =
    case r of
        Left err -> putStrLn err
        Right () -> putStrLn "All tests passed."
```

The `eq` function represents an assertion that two values ought to be the same. It has three parameters:

1. `n` – An ID number that will be included in the error message if the assertion fails, so that when you're reading the test results you can know ***where*** the error occurred (assuming you've given each test a unique number).
2. `actual` – This will the the result of some function application, e.g. `checkPasswordLength "julielovesbooks"`.
3. `expected` – This will be what you know the result of that function application ***should*** be, e.g. `Right "julielovesbooks"`.

```
eq :: (Eq a, Show a) => Int -> a -> a -> Either String ()
eq n actual expected =
    case (actual == expected) of
        True -> Right ()
        False -> Left (unlines
            [ "Test " ++ show n
            , "  Expected:  " ++ show expected
            , "  But got:   " ++ show actual
            ])
```

Here's an example of a small suite of tests.

```
test :: IO ()
test = printTestResult $
```

```
  do
    eq 1 (checkPasswordLength "") (Right "")
    eq 2 (checkPasswordLength "julielovesbooks")
        (Right "julielovesbooks")
```

```
*Main> test
All tests passed.
```

Expand upon it to write tests for the other validation functions, including success and error cases for each.

Exercise 16 – Kinds We've noted that all monads must be unary type constructors – that is, type constructors with a single type parameter. In Haskell, we use a notation called ***kinds*** to talk about the arity[2] of a type constructor. Kinds are also sometimes defined as "the types of types" but in practice we are usually only working directly with one type of type, and it is represented in GHC as a `*`.[3]

We can query the kind of a type or type constructor in GHCi.

```
*Main> :kind Bool
Bool :: *
```

```
*Main> :kind Maybe
Maybe :: * -> *
```

This tells us that the `Bool` type is already a concrete type, while `Maybe` is a function that takes one type as an argument and returns a type.

```
*Main> :kind Maybe String
Maybe String :: *
```

All monads have kind `* -> *`. `Either` has the wrong ***kindedness*** so we partially apply it to make a `Monad` instance.

```
*Main> :kind Either
Either :: * -> * -> *
```

[2]The term ***arity*** refers to how many arguments a function takes. Functions that take two arguments have 2-arity and are called ***binary***; functions with one parameter are called ***unary***; functions with an arity of zero are called ***nullary***, although we usually don't call those 'functions' in Haskell. ***Ternary*** means accepting three arguments.

[3]GHC has plans to eventually replace `*` with the word `Type`. We use `*` in this book because, as of this writing `*` is still in use in most places. If you want to investigate this further, look into the `StarIsType` language flag.

```
*Main> :kind Either String
Either String :: * -> *
```

Your task right now is to query the kinds of the following types and decide whether or not they *could* be monads. That is, do they have the same arity as `Maybe` or `Either String`.

1. `String`
2. `[]`
3. `(,)`
4. `(,) Int`
5. `data Pair a = Pair a a`

You can type that last one directly into your REPL or put it in a file and then load it.

Chapter 6

Working with newtypes

In this chapter we introduce some new types. We use `newtype`s that will help us distinguish at the type level between strings that represent inputs (password, username) and strings that represent errors.

6.1 Introducing newtypes

One of the things we have noted as we've been refactoring this code is the general undesirability of all these `String`s. If one reason we use Haskell is for type safety and for getting certain guarantees about our programs, then this isn't giving us much; there's nothing in our types that says errors and passwords are different things. And if we'd also like to start validating usernames, those would also be `String`s and how would we keep those separate from the other `String`s?

There are different ways to solve this problem, as there are for most problems, but a convenient and useful way to do so in Haskell is by introducing `newtype`s.

A `newtype` declaration in Haskell is different from a type synonym (written with the `type` keyword) or a data declaration (written with the `data` keyword) in a few ways. A type synonym[1] allows you to refer to a type

[1]We use the phrases "type synonym" and "type alias" interchangeably.

by a new name; we've been working with the `String` type, and we noted that it is a type alias for a list of `Char`.

```
type String = [Char]
```

Type synonyms like this are only a new name, not a new type. Their values are values of whatever the underlying type is, so, while you may use `String` as a shorthand in type signatures, any `String` value is indistinguishable from a `[Char]`.

```
x :: String
x = "hello"

f :: [Char] -> [Char]
f xs = reverse xs
```

```
*Main> f x
olleh
```

We could use type synonyms for our desired `Password` and `Username` types:

```
type Password = String
type Username = String
```

But this doesn't offer ***safety***; it just gives you another way to write the same type that provides you some reminder of what it means when you the code. In the example below, we accidentally provide a password to a function that expects a username.

```
x :: Password
x = "julielovesbooks"

greet :: Username -> IO ()
greet username = print ("Hello, " ++ username)
```

```
*Main> greet x
"Hello, julielovesbooks"
```

A `newtype` wrapper is similar to a type alias in that it allows us to give some underlying type a new name, but is not ***merely*** a new name; it is also a new type. It is semantically distinct. It has a data constructor that is named differently from the underlying type's constructors, which allows us to construct values of the new type.

```
*Main> newtype Password = Password String deriving Show
newtype Password = Password String

*Main> :type Password
Password :: String -> Password
```

By using a `newtype` wrapper, we have a data constructor that *constructs* a value of type `Password`. This allows us much greater ability to keep a `Password` separate from, say, a `Username` or an `Error` or any other variety of `String`.

Since it is a new type (despite having a machine representation that is identical to the first type), it also allows us to write new typeclass instances that are different from those of the underlying type; this is something we cannot do with type synonyms.

There are a few important differences between a `newtype` and a `data` declaration, most of which won't be terribly important in this book:

- `newtype` may only have a single unary constructor.
- The guarantee that there will only ever be one unary constructor allows for certain optimizations because the `newtype` has the exact same representation as the underlying type.
- `newtype` offers safe and cheap coercion between the `newtype` and its underlying type (we'll see this later in the book), which `data` does not.
- You can use a language extension called `GeneralizedNewtypeDeriving` to derive typeclass instances based on the underlying type.
- `newtype` and `data` have some different behavior with regard to nonstrictness.

6.2 Declaring new types

We will add three `newtype` declarations to our code. For the code we have written so far, we only need a `Password` and an `Error` type in order to distinguish between our two kinds of `String`. However, in this chapter we'll also begin extending our code so that we can validate usernames (which we'll later user to combine a valid username and a valid password into a `User`).

Each of the three newtype declarations declaration begins with the

`newtype` keyword, a name for the new type, a data constructor with the same name as the type constructor[2], and the underlying type that we're wrapping.[3]

We derive a `Show` instance for each type as well, which we need to be able to print values of the type to the terminal.

```
newtype Password = Password String
    deriving Show
```

```
newtype Error = Error String
    deriving Show
```

```
newtype Username = Username String
    deriving Show
```

One thing that can be handy in larger programs or in programs that involve a lot of imports is you can use the `:type` and `:info` commands in GHCi to get information about any types in scope. So, for example, try reloading the file with these newly-defined types into GHCi and querying the `:info` of `Error` and the `:type` of `Password`. You'll be able to see where they're defined and what typeclass instances they have in scope, same as for the built-in types in `base` and the like.

6.3 Using our new types

We're going to first construct similar but different functions to check `Username` length and `Password` length.

First we'll refactor our `checkPasswordLength` function to use the `Error` and `Password` types.

```
checkPasswordLength :: String -> Either Error Password
checkPasswordLength password =
  case (length password > 20) of
    True -> Left (Error "Your password cannot be longer \
```

[2]The data constructor does not have to have the same name as the type constructor. That is a common naming strategy but not an obligatory one.

[3]The constructor in a `newtype` is very often a function in a record field (`newtype Password = Password { unPassword :: String }`), but we are not going to use that idiom in this book.

```
                        \than 20 characters.")
    False -> Right (Password password)
```

Notice that we have to apply the `Error` and `Password` data constructors to the strings in order to return values of those types.

Next we want an analogous function to check the length of usernames for us. This time it will return an `Either Error Username`.

```
checkUsernameLength :: String -> Either Error Username
checkUsernameLength name =
  case (length name > 15) of
    True -> Left (Error "Username cannot be longer \
                        \than 15 characters.")
    False -> Right (Username name)
```

We will use `requireAlphaNum` and `cleanWhitespace` for both `Username` and `Password` so those should both return `Either Error String` and have a generic parameter name, reflecting the fact that we will use them for strings that are passwords and also strings that are usernames.

```
requireAlphaNum :: String -> Either Error String
requireAlphaNum xs =
  case (all isAlphaNum xs) of
    False -> Left (Error "Cannot contain white space \
                         \or special characters.")
    True -> Right xs
```

```
cleanWhitespace :: String -> Either Error String
cleanWhitespace "" = Left (Error "Cannot be empty.")
cleanWhitespace (x:xs) =
  case (isSpace x) of
    True -> cleanWhitespace xs
    False -> Right (x:xs)
```

Next we need to change `validatePassword` to use our new types. We can now take a `Password` value as an input instead of a generic `String`, validate that, then return it as a `Password` value, ensuring that it can't get confused somewhere along the way with any other sort of `String`.

```
validatePassword :: Password -> Either Error Password
validatePassword (Password password) =
    cleanWhitespace password  -- String -> Either Error String
    >>= checkPasswordLength   -- String -> Either Error Password
    >>= requireAlphaNum       -- String -> Either Error String
```

Notice that in order to pass it as a `String` to the first function `cleanWhitespace`, we have to "unwrap" it from its `Password` data constructor via pattern matching at the point where we apply `validatePassword` to a `Password` argument.[4]

Now if you try to load this into the REPL, it won't typecheck! It will tell you where the errors are, more or less, but it might take a moment to understand it.

We need to return either an `Error` or a `Password` – not a `String`. But `requireAlphaNum` is the last function in our series of functions, and what type does `requireAlphaNum` return? What type does `checkPasswordLength` return? Does it return the type we need? Yes! So we can reorder `checkPasswordLength` and `requireAlphaNum` and fix (most of) those type errors.

```
validatePassword :: Password -> Either Error Password
validatePassword (Password password) =
    cleanWhitespace password  -- String -> Either Error String
    >>= requireAlphaNum       -- String -> Either Error String
    >>= checkPasswordLength   -- String -> Either Error Password
```

In case you're worried about what you might do if you didn't want to reorder them, we'll explore an alternative design in chapter **??**.

6.4 Revising `main`

The remaining type error lies in `main`. It should be telling you that it can't match the type `[Char]` with the type `Password` in the first argument of `validatePassword`. Remember, we changed `validatePassword` to require a `Password` input – but since the type of `getLine` is `IO String`, then the type of the `password` value we bind its result to is `String`. That mismatch is the source of this error.

```
main :: IO ()
main =
```

[4] While this technique is technically called "pattern matching" and is related to the pattern matching we do when writing functions that, for example, branch on the different constructors of `Maybe`, it may feel strange to call it that – what is the *pattern* exactly? Pattern matching can be done on nearly any constructor or value, and it allows us to dispatch behavior based on having found the value or constructor that matches.

```
  do
    putStrLn "Please enter a password."
    password <- getLine
    print (validatePassword password)
```

You might consider trying this to fix it (it doesn't work):

```
main :: IO ()
main =
  do
    putStrLn "Please enter a password."
    password <- Password getLine
    print (validatePassword password)
```

The problem is that `getLine` is an `IO String`, and we want to apply the `Password` constructor to the `String` – not to the `IO`! In a binding of the form `a <- b`, `a` and `b` are different things and they have different types. When we write `password <- getLine`,

- `getLine` has the type `IO String`;
- `password` has the type `String`.

To apply a function to the `String`, then, we need to do so *after* the line where the `password` variable is bound. Since `password` has the type `String`, we'll introduce a new variable `password'` to be the one that has the type `Password`. To write a definition in a `do` block that *doesn't* do any action, rather merely introduces a new variable, we use the `let` keyword.

```
main :: IO ()
main =
  do
    putStrLn "Please enter a password."
    password <- getLine
    let password' = Password password
    print (validatePassword password')
```

It's a bit cumbersome that we now need two lines of code. We can condense this with a special kind of function application called `fmap`. That will apply the `Password` constructor *inside* the `IO`, changing our `IO String` into an `IO Password` which allows the `Password` to be passed to `validatePassword` (and allows us to go on ignoring `IO`).

Here, we use the infix operator for `fmap`: `<$>`.[5]

[5] `fmap` and `<$>` are defined in `Data.Functor` and re-exported by `Prelude`.

```
main :: IO ()
main =
  do
    putStrLn "Please enter a password."
    password <- Password <$> getLine
    print (validatePassword password)
```

Now all of this should typecheck, and you should be able to do some experimentation with it in the REPL.

6.5 Exercises

Exercise 17 – Usernames Write a username validation function:

```
validateUsername :: Username -> Either Error Username
```

It should clean up whitespace with `cleanWhitespace`, require only alphanumeric characters with `requireAlphaNum`, and limit the length of usernames using `checkUsernameLength`.

Exercise 18 – Generalizing with a parameter We've so far been using (and for the purposes of the main text, will continue to use) separate functions for checking the length of the username and password with hard-coded maximum lengths. We reproduce them here for convenience.

```
checkPasswordLength :: String -> Either Error Password
checkPasswordLength password =
  case (length password > 20) of
    True -> Left (Error "Your password cannot be longer \
                        \than 20 characters.")
    False -> Right (Password password)

checkUsernameLength :: String -> Either Error Username
checkUsernameLength name =
  case (length name > 15) of
    True -> Left (Error "Username cannot be longer \
                        \than 15 characters.")
    False -> Right (Username name)
```

While there isn't anything *wrong* with this, it's somewhat redundant. We could write one generic function that is parameterized on an `Int` – there

is no need here for the infinitude of `Integer` – and could work for both passwords and usernames, even if we want them to have different maximum lengths. Your job is to write that function.

```
checkLength :: Int -> String -> Either Error String
```

A couple of things to think about:

- Currently the two functions return the new type indicating whether it's a `Password` or `Username`, but this one won't be able to.
- You'll need to do *something* about the `validatePassword` and `validateUsername` functions to make them return the correct type. Remember that those data constructors, `Password` and `Username`, are like functions and can be applied to values.

Exercise 19 – Non-binding I/O Rewrite the previous `main` using `>>=` instead of `do`. It currently looks like this:

```
main :: IO ()
main =
  do
    putStrLn "Please enter a password."
    password <- Password <$> getLine
    print (validatePassword password)
```

You will need to use `>>` instead of `>>=` after the `putStrLn` because `putStrLn` doesn't return a *value* that can be bound as an argument; it only performs an effect of printing to the screen.

```
(>>=) @IO :: IO a -> (a -> IO b) -> IO b
```

```
(>>)  @IO :: IO a ->       IO b  -> IO b
```

Exercise 20 – Doing Either One of the things we like about Haskell is that it is generally systematic rather than arbitrary; although so far we have only used `do` syntax with the `IO` monad, it also works with *any other monad*: `Maybe`, `Either`, etc. Try rewriting `validatePassword` with `do` rather than `>>=`.

Here is the original version of `validatePassword` for reference:

```
validatePassword :: Password -> Either Error Password
validatePassword (Password password) =
    cleanWhitespace password
```

```
    >>= requireAlphaNum
    >>= checkPasswordLength
```

6.6 Notes on monadic style

Now that you've refactored a couple functions to change them back and forth between `do` and `>>=`, you may be wondering you should use one style or the other. It's often a matter of preference, but we can offer a few guidelines:

Too many lambdas If it seems like you're writing way too many lambda expressions,

```
a1 >>= \x1 ->
a2 >>= \x2 ->
a3 >>= \x3 ->
f x1 x2 x3
```

consider changing from `>>=` to `do`.

```
do
  x1 <- a1
  x2 <- a2
  x3 <- a3
  f x1 x2 x3
```

Too many variables If it seems like you have too many variables that get introduced one one line only to get used on the next,

```
do
  x <- a
  x1 <- f1 x
  x2 <- f2 x1
  x3 <- f3 x2
  f4 x3
```

consider changing from `do` to `>>=`.

```
a >>= f1 >>= f2 >>= f3 >>= f4
```

Why not both? Don't forget that you can use a combination of the two approaches.

```
do
  x1 <- a1 >>= f1 >>= f2
  x2 <- a2 >>= f3 >>= f4
  f5 x1 x2
```

Chapter 7

Introducing Applicative

This chapter picks up where the previous one ended and adds a `validateUsername` function. Then, since we'd like to keep a username and a password together as a single value, we write a product type called `User` and a `makeUser` function that constructs a `User` from the conjunction of a valid `Username` and a valid `Password`. We will introduce the `Applicative` typeclass to help us write that function.

7.1 Validating usernames

`validateUsername` is the same as `validatePassword` but this time for our `Username` strings. We will still want to pattern match the `Username`-typed input when we pass it to our function so that what we're left with is a `String` that can be passed to `cleanWhitespace`:

```
validateUsername :: Username -> Either Error Username
validateUsername (Username username) =
    cleanWhitespace username
    >>= requireAlphaNum
    >>= checkUsernameLength
```

As we saw previously with `validatePassword`, the length-validating function takes a `String` as input but returns a `Username` (in the `Right`, or

successful, case), so it will be convenient for us to make that the last in this chain of functions so we don't need to do any extra wrapping or unwrapping of newtype constructors to end up with the return type we're looking for.

7.2 Adding to `main`

Since we now have the ability to validate usernames, we can go ahead and add this to our `main` program so that we can take username inputs as well as password inputs.

Our `main` currently looks like this:

```
main :: IO ()
main =
  do
    putStrLn "Please enter a password."
    password <- Password <$> getLine
    print (validatePassword password)
```

Once again, the things we're adding will mimic the things we've already done. We will go ahead and add a prompt to request that the user enter a username *above* the prompt requesting the password; that ordering isn't strictly necessary since we could ask for them in any order and still pack them into our `User` type later in the order we want them in. But we'll ask for the username first and the password second, as is the conventional user interface, and uncoincidentally also the order of the fields in the `User` constructor.

```
main :: IO ()
main =
  do
    putStrLn "Please enter a username."
    username <- Username <$> getLine
    putStrLn "Please enter a password."
    password <- Password <$> getLine
    print (validateUsername username)
    print (validatePassword password)
```

You may recall from the last lesson that `<$>` is the infix version of `fmap` and that we are using it to turn the `IO String` from `getLine` into an `IO`

`Username` or `IO Password` since our validating functions take `Username` and `Password` inputs (not `String`s).

Since we do not yet have a concept of a single value representing a username and password *together*, we've added a separate line to print the username as we did for the password. Before moving on, experiment a bit with `main` in your REPL.

7.3 Constructing a `User`

Next we'll write a type that conjoins a username and a password into a single entity.

```
data User = User Username Password
    deriving Show
```

So far we've only really talked about sum types and newtypes; `User` is a *product type*. A product type is a conjunction ("and") of two or more types, not an "or" as we saw with sum types. So, a `User` is a value constructed from the conjunction of a `Username` *and* a `Password`, wrapped up together in the `User` constructor.[1]

We want to put together a `makeUser` function that will take a `Username` input and a `Password`, verify that they are both valid, and if so construct a `User`. We'll be using our `validate` functions that we've already constructed, which gives us the possibility of having an `Error` for either the `Username` or the `Password`, which would leave us unable to construct a `User`; thus our return type is `Either Error User`.

```
makeUser :: Username -> Password -> Either Error User
```

It may be worth observing that the type of `makeUser` quite resembles the type of the `User` constructor; the difference is that `makeUser` introduces a possibility of failure where `User` does not.

```
User     :: Username -> Password ->              User
```

When we wrote the `validateUsername` and `validatePassword` functions, we noted the importance of using the monadic `>>=` operator when

[1] For when you don't want to name your product, there are the canonical product types (sometimes called *anonymous products* or *tuples*). So for example, the type `(Username, Password)` is the anonymous product of `Username` and `Password`.

the input of a function ***must depend*** on the output of the previous function. We wanted the inputs to our character and length checks to ***depend on*** the output of `cleanWhitespace` because it might have transformed the data as it flowed through our pipeline of validators.

However, in this case, we have a different situation. We want to validate the name and password inputs ***independently*** – the validity of the password does not depend on the validity of the username, nor vice versa – and then bring them together in the `User` type only if both operations are successful.

For that, then, we will use the primary operator of a different typeclass: `Applicative`. We often call that operator "tie-fighter" or sometimes "apply" or "ap".

```
(<*>) :: Applicative f => f (a -> b) -> f a -> f b
```

`Applicative` occupies a space between `Functor` (from which we get `<$>`) and `Monad`, and `<*>` is doing something very similar to `<$>` and `>>=`, which is allowing for function application in the presence of some outer type structure.

In our case, the "outer type structure" is the `Either a` functor. As we've seen before with `fmap` and `>>=`, `<*>` must effectively ignore the `a` parameter of `Either`, which is the `Error` in our case, and only apply the function to the `Right b` values. It still returns a `Left error` value if either side evaluates to the error case, but unlike `>>=`, there's nothing inherent in the type of `<*>` that would force us to "short-circuit" on an error value. We don't see evidence of this in the `Either` applicative, which behaves coherently with its monad, but we will see the difference once we're using the `Validation` type.

We've now seen the type of `<*>` and we know we're using the `Either a Applicative`; so what is our `f (a -> b)` function?

```
(<*>) :: Applicative f => f (a -> b) -> f a -> f b
                          ---------
                             ?
```

7.4 Constructors are functions

We used `<$>` to map our data constructors (`Username` and `Password`) over `getLine`. Let's look at the types of what we were doing there:

```
*Main> :type (<$>)
(<$>) :: Functor f => (a -> b) -> f a -> f b

*Main> :type Password
Password :: String -> Password

*Main> :type getLine
getLine :: IO String

*Main> :type (<$>) @IO
(<$>) @IO :: (a -> b) -> IO a -> IO b

*Main> :type (Password <$> getLine)
(Password <$> getLine) :: IO Password
```

In this usage, then, our `(a -> b)` function is the data constructor `Password`. It has the type `String -> Password`, so when we `fmap` that over a value of type `IO String`, we end up with a value of type `IO Password`.

For our `makeUser` function, we're constructing a value of a type that has *two* parameters.

```
*Main> :type User
User :: Username -> Password -> User
```

We're going to `fmap` that over a value of type `Either Error Username`.

```
*Main> :type (validateUsername (Username "name"))
(validateUsername (Username "name")) :: Either Error Username
```

When we do that, then our `User` constructor function will be partially applied:

```
*Main> :type (User <$> (validateUsername (Username "name")))
(User <$> (validateUsername (Username "name")))
  :: Either Error (Password -> User)
```

That type, `Either Error (Password -> User)`, shows us where our `f (a -> b)` is — the `f` is the `Either Error` and the `(a -> b)` is the `(Password -> User)` function. We're constructing a function *inside* an applicative type (`Either`) and that's exactly the context the tie-fighter is for!

7.5 Using `Applicative`

So, our `makeUser` function must look like this:

```
makeUser :: Username -> Password -> Either Error User
makeUser name password =
    User <$> validateUsername name
         <*> validatePassword password
```

Experiment with passing it different kinds of invalid inputs in the REPL and confirm for yourself that it is returning the ***leftmost*** error value and that value will be the first error case that the monadic validate function hits. In the next lesson, we'll rely on a different applicative functor to get a different behavior out of this function.

This time around, we only need to change the last lines of `main` to use our brand-new `makeUser` function.

```
main :: IO ()
main =
  do
    putStrLn "Please enter a username."
    username <- Username <$> getLine
    putStrLn "Please enter a password."
    password <- Password <$> getLine
    print (makeUser username password)
```

`:reload` this into the REPL to make sure everything type checks, and do some experimentation. There are some things we want to fix. For one thing, on success cases, we're displaying a lot of constructors to our user, revealing much more to them than we should. We'll work on that in chapter 9.

Another thing we notice is that we can only return one error at a time, even though we know our inputs have multiple errors. If you've ever used a website that can only show you one error message at a time, you're aware how cruel it is to force your users to go through this unnecessary interaction:

1. Username: `mynameisjuliemoronuki`
   Password: `qahw*me#zdrh$sr`

   Error: Username cannot be longer than 15 characters.

2. Username: `julie`
   Password: `qahw*me#zd j%frh$sr)fta!nroi4hjf`

   Error: Cannot contain white space or special characters.

3. Username: `julie`
   Password: `qahwXmeXzdXjXfrhXsrXftaXnroi4hjf`

   Error: Your password cannot be longer than 20 characters.

We'll use the `Validation` type in the next chapter to fix this situation.

7.6 Exercises

Exercise 21 – Fitting in Perhaps an administrator might need to create a new user account on behalf of someone else, with a temporary password that the user is expected to change later. We'll assume that choice of password is fixed in the code and doesn't need to go through the validation checks.

Here's an attempt at writing a function that creates a user with a default password:

```
makeUserTmpPassword :: Username -> Validation Error User
makeUserTmpPassword name =
    User <$> validateUsername name
         <*> Password "temporaryPassword"
```

Why doesn't this work? What do we have to do to fix it? (Hint: read "doesn't need to go through the validation checks" as "is always *successful*.")

Exercise 22 – Purely success There's a little more to `Applicative` that we've let on so far. We've talked about the interesting part, `(<*>)`, but there's also a function called `pure`.

```
class Functor f => Applicative (f :: * -> *) where
    pure :: a -> f a
    (<*>) :: f (a -> b) -> f a -> f b
```

The reason `Applicative` has the `pure` function is that when we're using `<*>`, we very often need to do exactly what we did in the previous ex-

ercise: lift some value into the context of `f` without doing anything else to it.

It's usually possible to guess how `pure` is defined for a given type constructor because `pure` typically does very little and there aren't many possible choices for how to write it. Implement each of the following functions:

1. `pureMaybe :: a -> Maybe a`
2. `pureEither :: a -> Either l a`
3. `pureValidation :: a -> Validation err a`

Exercise 23 – Applicative I/O Every type with a `Monad` instance also necessary has an `Applicative` instance. We discussed before that `IO` is a monad – and indeed literature often emphasizes its monadic aspect by referring to it as "the IO monad" – but it's easy to forget that it's also an applicative functor.

Occasionally it's convenient to write in an "applicative style" with IO instead of using `do` or `(>>=)`. The anagram program we wrote in chapter 2 is an example of a program that mostly fits the pattern. It is repeated for reference here:

```
main :: IO ()
main =
  do
    putStrLn "Please enter a word."
    word1 <- getLine
    putStrLn "Please enter a second word."
    word2 <- getLine
    print (checkAnagram word1 word2)
```

We'll break each of the input prompts into its own separate action:

```
promptWord1 :: IO String
promptWord1 =
  do
    putStrLn "Please enter a word."
    getLine
```

```
promptWord2 :: IO String
promptWord2 =
  do
    putStrLn "Please enter a second word."
    getLine
```

What type of expression goes into the blank below? (Hint: GHC can answer this for you.)

```
main :: IO ()
main =
  do
    result <- _
    print result
```

Fill in the hole to complete `main`, using `(<$>)` and `(<*>)`.

Chapter 8

Refactoring with Validation

In this chapter we do a thorough refactoring to switch from `Either` to `Validation`. Despite the fact that those two types are essentially the same,[1] their `Applicative` instances are quite different and switching to `Validation` allows us to accumulate errors on the left. In order to do this, we'll need to learn about a typeclass called *Semigroup* to handle the accumulation of `Error` values.

8.1 Introducing `validation`

We begin by looking at the package that type comes from.[2]

Although the `Validation` type is isomorphic to `Either`, because they are different types they can have different instances of the `Applicative` class. Since `instance` declarations define how functions work, this means overloaded operators from the `Applicative` typeclass can work differently for `Either` and `Validation`.

[1] More precisely, these two types are *isomorphic*, by which we mean that you can convert values back and forth between `Either` and `Validation` without discarding any information in the conversion.

[2] The `validation` package on Hackage: https://hackage.haskell.org/package/validation

We used the `Applicative` for `Either` in the last chapter and we noted we used `Applicative` instead of `Monad` when we didn't need the input of one function to depend on the output of the other. We also noted that although we weren't technically getting the "short-circuiting" behavior of `Monad`, we could still only return one error string. The "accumulating `Applicative`" of `Validation` will allow us to return more than one.

The way the `Applicative` for `Validation` works is that it appends values on the left/error side using a `Semigroup`. We will talk more about semigroups later, but for now we can say that our code will be relying on the semigroup for lists, which is concatenation.

The `validation` package has one module, `Data.Validation`, and that is where we find the documentation of the type and its associated instances and functions. We'll get to adding a proper dependency to our project in a minute. First, try launching GHCi like this:

```
$ stack repl --package validation
```

This starts a GHCi session in which the `validation` package is available. Since the `validation` package has some dependencies of its own, namely on the `lens` library, this may take a while the first time.

If you type `import Data.Validation` and then `:info Validation`, you can see the type definition

```
data Validation err a = Failure err | Success a
```

The type has two parameters, one called `err` and the other called `a`, and two constructors, `Failure err` and `Success a`. The output of `:info Validation` also includes a list of instances.

`Validation` is not a `Monad` The instance list does *not* include `Monad`. Because of the accumulation on the left, the `Validation` type is not a monad. If it were a monad, it would have to "short circuit" and lose the accumulation of values on the left side. Remember, monadic binds, since they are a sort of shorthand for nested `case` expressions, must evaluate sequentially, following a conditional, branching pattern. When the branch that it's evaluating reaches an end, it must stop. So, it would never have the opportunity to evaluate further and find out if there are more errors. However, since functions chained together with applicative operators instead of monadic ones can be evaluated independently, we can accumulate the errors from

several function applications, concatenate them using the underlying semigroup, and return as many errors as there are.

`err` needs a `Semigroup` Notice that `Applicative` instance has a `Semigroup` constraint on the left type parameter.

```
instance Semigroup err => Applicative (Validation err)
```

That's telling us that the `err` parameter that appears in `Failure err` must be a semigroup, or else we don't have an `Applicative` for `Validation err`. You can read the `=>` symbol like implication:

```
instance
  Semigroup err                -- If 'err' is semigroupal
  =>                           -- then
  Applicative (Validation err) -- 'Validation err' is
                               --     applicative
```

Our return types all have our `Error` type as the first argument to `Either`, so as we convert this to use `Validation`, the `err` parameter of `Validation` will be `Error`. The `Semigroup` constraint on the `Applicative` instance tells us that our `Error` type must be a semigroup. Currently, `Error` is a `newtype` for `String` and `String` is a semigroup. However, as we discussed when we wrote the type, you don't inherit the instances of the underlying type by default. This is because, as we'll see shortly, that's not always the instance that you want.

8.2 Adding a dependency

`stack repl --package validation` gave you an ephemeral environment containing the `validation` package. To affix this dependency to your package, you need to edit its Cabal file. Open `validation-book.cabal`.

```
name:     validation-book
version:  0.1.0.0
[ ... more fields ... ]

executable validation
  hs-source-dirs:      src
  main-is:             Main.hs
```

```
  default-language:    Haskell2010
  build-depends:       base >= 4.7 && < 5
```

The bottom chunk consisting of `executable validation` and indented lines under it is called a ***stanza***. This one is an ***executable*** stanza, which means it defines how to produce an executable program named `validation` when we run `stack build`. (If we wanted our package to produce multiple programs when we build it, we could add more `executable` stanzas.)

Currently, the `base` package is our only dependency, but we need to add `validation` under `build-depends`. Be sure your `v` is lower case![3] Make sure to separate it from the `base` dependency with a comma.

```
executable validation
  hs-source-dirs:      src
  main-is:             Main.hs
  default-language:    Haskell2010
  build-depends:       base >= 4.7 && < 5
                     , validation
```

If you have GHCi open, you'll need to close it and run `stack repl` again to start a new REPL for the change to take effect.

Next we need to import that `Data.Validation` module into our `Main` module, as we imported `Data.Char` previously.

```
module Main where

import Data.Char
import Data.Validation
```

Now the type, constructors, instances, and so forth from `Data.Validation` will all be in scope for our `Main` module.

[3] Package names are case sensitive, and there can (and do) exist packages whose names vary only in capitalization. We are using the package named `validation` here. There also exists a deprecated package named `Validation`, which we are ***not*** using.

8.3 Nominal refactoring

We begin the `Either`-to-`Validation` refactor by altering the `Error` type slightly. Since we now want to allow the possibility of accumulating multiple error messages, we change our `Error` type to be a list of strings rather than a single string.

```
newtype Error = Error [String]
    deriving Show
```

It's a small change, but it means we need to make sure all the `Error` values are now lists of strings as part of our refactoring.

Since `Validation` is so similar to `Either`, most of the refactoring work only involves changing the names of the constructors.

- At the type level:
  - `Either` becomes `Validation`.
- At the term level:
  - `Left` becomes `Failure`.
  - `Right` becomes `Success`.

Our first pass at refactoring can thus be done with "find and replace" in our text editor; watch out because there are a couple of error messages where we used the word "left" which is not the same as `Left` and should not be replaced with `Failure`.

Then wrap the `Error` strings in square brackets to make them lists, so `Error "..."` becomes `Error ["..."]`.

For example, `requireAlphaNum` used to look like this:

```
requireAlphaNum :: String -> Either Error String
requireAlphaNum xs =
  case (all isAlphaNum xs) of
    False -> Left (Error "Cannot contain white space \
                         \or special characters.")
    True -> Right xs
```

And now should look something like this:

```
requireAlphaNum :: String -> Validation Error String
requireAlphaNum xs =
  case (all isAlphaNum xs) of
```

```
    False -> Failure (Error ["Cannot contain white space \
                             \or special characters."])
    True -> Success xs
```

8.4 Interpreting the errors

Reload the project in GHCi to see what else we need to do to complete our refactor. If you have changed all the names successfully and put the list brackets around every `Error` string (at the term level), then GHCi should return two error messages. These were both things that we had anticipated from reading the `validation` documentation.

No instance for `Monad (Validation Error)` arising from a use of `>>=` It also says the line where the error occurs. Since there is more than one use of `>>=`, that is slightly misleading; it will need to be fixed on more than one line. This is a type error we anticipated, as we know that `Validation` is not a monad. We will have to change the functions that use `>>=` to use something from the `Applicative` class instead.

No instance for `(Semigroup Error)` arising from a use of `<*>` If you recall, when we looked at the `Applicative` instance for `Validation`, we noted that there is a `Semigroup` constraint on the `err` parameter, which, for us, is `Error`. Our use of `<*>` now relies on that `Validation` `Applicative` instance and so we must have a `Semigroup` instance for `Error`, otherwise it doesn't know how to do the error accumulation on the left.

`Error` is *representationally equivalent* to `[String]` but not *nominally equivalent*. If we had not made this `newtype`, we could have relied on the list semigroup that is already in `base`, but one of the points of making `newtype` wrappers is to preserve one type of equivalence[4] while introducing nominal differences that allow us to have different instances and behaviors based on the name.

[4] If we had made `Error` a `data` declaration instead of a `newtype` declaration, then `Error` and `[String]` would ***not*** be representationally equivalent.

8.5 An `Error` semigroup

We tackle the `Semigroup` issue first. The error message nearly tells us what to do: The problem is that there is no `Semigroup Error` instance, so the fix is to write an `instance` of the `Semigroup` typeclass for our `Error` type.

Looking at the documentation for `Data.Semigroup`, we notice the `class` declaration defines the following operator, usually called `mappend`.

```
class Semigroup a where
    (<>) :: a -> a -> a
```

There are other functions defined in the class, but a definition of just `<>` suffices as a *minimal complete definition* for an `Semigroup` instance, and the other functions have default implementations based on `<>`.

```
import Data.Semigroup                          -- 1

instance Semigroup Error where                 -- 2
    Error xs <> Error ys = Error (xs ++ ys)    -- 3
```

1. We need to add an import of `Data.Semigroup` along with our other imports.
2. Then we open an instance declaration with the `instance` keyword followed by the typeclass name and name of the type that the instance is for, followed by `where`.
3. The second line of the instance defines the behavior of the `<>` operator for the `Error` type. By the type signature given in the `Semigroup` `class` declaration above, we know our implementation needs to take two `Error` values and return one `Error` value. Each of those `Error` values contains a `[String]`, which we represent with `xs` and `ys`.

For concatenating the two lists, we used the `++` function, which is defined in `Data.List` and re-exported by `Prelude`.

```
(++) :: [a] -> [a] -> [a]
```

We can compare this directly with the type signature of `<>` and see that they're the basically same, except that `<>` is more general and `++` is more concrete.

```
(++) :: [a] -> [a] -> [a]
(<>) ::  m  ->  m  ->  m
```

In other words, the type of `<>` can *specialize to* to the type of `++`, by replacing `m` by `[a]`. Note that *both* functions are polymorphic; even though `++` is *more* constrained than `<>`, it does still have a type variable.

The list `Semigroup` Curiously, if you look at the source code for the `Data.Semigroup` module, you can see that the `Semigroup` instance for lists defines the `<>` operator as `++`.

```
instance Semigroup [a] where
    (<>) = (++)
```

Therefore any time you use the `++` operature, you could replace it with the more general `<>` operator.

```
*Main> [1,2] ++ [3,4,5]
[1,2,3,4,5]

*Main> [1,2] <> [3,4,5]
[1,2,3,4,5]
```

So we could have written the `Semigroup Error` instance like this:

```
instance Semigroup Error where
    Error xs <> Error ys = Error (xs <> ys)
```

Writing it this way sort of highlights the fact that our `Semigroup` instead doesn't really *do* much, because it works exactly the same as the instance of its underlying type.

Newtype deriving In cases like this where this does happen to be the instance you want, you can take advantage of this by *deriving* an instance instead of writing it yourself. You would need a language extension called `GeneralizedNewtypeDeriving` turned on in order to do that, but it's a safe and mostly straightforward feature.

```
{-# LANGUAGE GeneralizedNewtypeDeriving #-}

newtype Error = Error [String]
    deriving (Semigroup, Show)
```

You can read this definition of `Error` as "an error is a list of strings, and the semigroup for errors works the same as the semigroup for lists of

strings." Deriving means you get the standard instance for the underlying type that the newtype is isomorphic to.

Haskellers often use features like generalized newtype deriving (among others) to avoid having to write typeclass instances for themselves. This is not always the case, though, as you will see when you write another `Semigroup` instance in a later exercise.

8.6 Using `Applicative`

That takes care of the `Semigroup` error, and now we are left with the `Monad` error. Both of these errors involved missing typeclass instances, but their solutions are different:

- In the previous section, we fixed an error by providing the missing instance.
- But `Validation` cannot have a `Monad` instance, so we need to fix this problem by refactoring the `validatePassword` and `validateUsername` functions to use some `Applicative` functions instead of monadic binds.

We do still want `cleanWhitespace` to be applied to, and possibly transform, the argument *first* and for the inputs of our later functions to depend on that. There are various ways we could handle this,[5] but since we have already used `case` expressions quite a bit for that, we do so again. You don't always need to look for a function that gives you the most concise way to write something; a `case` expression is a great default tool to reach for.

```
validatePassword :: Password -> Validation Error Password
validatePassword (Password password) =
    case (cleanWhitespace password) of
        Failure err -> Failure err
        Success password2 -> _
```

Next we need to replace that `_`. We need to apply `requireAlphaNum` and then a length checking function to the `Success` value. However, we do not need the output of `requireAlphaNum` – we only need the error if it returns

[5] Another way we could have fixed this is by replacing the first `>>=` with `bindValidation`, a function helpfully provided by the `Data.Validation` module to deal with just this situation.

one. We do need the output of the length-checking functions, because they return the appropriate type.

The `Applicative` typeclass has a pair of operators that we like to call left- and right-facing bird, but some people call them left and right shark. Either way, the point is they eat one of your values.

```
(*>) :: Applicative f => f a -> f b -> f b
```

```
(<*) :: Applicative f => f a -> f b -> f a
```

These effectively let you sequence function applications, discarding either the first return value or the second one. The thing that's pertinent for us now is they do not eat any ***effects*** that are part of the `f`. Remember, when we talk about the `Applicative` instance for `Validation`, it's really the `Applicative` instance for `Validation err` because `Validation` must be applied to its first argument, so our `f` is `Validation Error`, and that instance lets us accumulate `Error` values via a `Semigroup` instance (concatenation).

```
*Main> Failure ["x"] *> Failure ["y"]
Failure ["x","y"]
```

Any `Error` values that are part of that `f` will get passed along, but `*>` leaves the `a` value behind and just returns the `f b` at the end – either a `Failure Error` or a `Success Password`.

```
*Main> Success 1 *> Failure ["y"]
Failure ["y"]
```

```
*Main> Failure ["x"] *> Success 1
Failure ["x"]
```

```
*Main> Failure ["x"] *> Success 1 *> Failure ["y"]
Failure ["x","y"]
```

Whether we use the left- or right- bird only affects which result we get when everything is successful.

```
*Main> Success 1 *> Success 2 *> Success 3
Success 3
```

```
*Main> Success 1 <* Success 2 <* Success 3
Success 1
```

With that in mind, let's write the `Success` case for our `validatePassword` function:

```
validatePassword :: Password -> Validation Error Password
validatePassword (Password password) =
    case (cleanWhitespace password) of
        Failure err -> Failure err
        Success password2 -> requireAlphaNum password2 *>
                             checkPasswordLength password2
```

You notice we have to apply each of the functions – `requireAlphaNum` and the length-checking functions – to their arguments, instead of being able to sort of thread the argument through as we might do with `>>=`. Look back at the type of `*>` to understand why.

```
*Main> :type (*>)
(*>) :: Applicative f => f a -> f b -> f b

*Main> :type requireAlphaNum
requireAlphaNum :: String -> Validation Error String

*Main> :type requireAlphaNum "julie"
requireAlphaNum "julie" :: Validation Error String
```

It's the second one – `requireAlphaNum` applied to a string literal – that has the type `f a`, not `requireAlphaNum` by itself.

We have by now accomplished most of the goals we set for ourselves at the outset of this book. We are able to return a list of helpful error messages instead of only a `Nothing` or at most one error, and our functions are mostly small and readable. But we can keep improving! Next we will add some helper functions to help us present the errors in a more aesthetically pleasing fashion, and that will also lead us to another very useful tool for working with `newtypes`.

8.7 Exercises

Exercise 24 – Applicative do In standard GHC Haskell, `do` syntax can only be used in monadic contexts. It's used most commonly in `IO` but can

be used with other monads, as we've seen. It is convenient enough and popular enough that some people want to also use it in applicative contexts, and so GHC now (as of version 8.0) has a language extension called `ApplicativeDo` that lets you use `do` syntax even when you don't have a monad. For this exercise, we want to enable that extension.

Add this to the very top of your module (above the module name):

```
{-# LANGUAGE ApplicativeDo #-}
```

And then rewrite `makeUser` using `do` syntax

```
makeUser :: Username -> Password -> Validation Error User
makeUser name password = _
```

Chapter 9

Better Error Messages

We have come a very long way in this book and we might be tempted to rest after all we've done. However, with a just a little more effort we can pass much nicer, easier to read error (and success!) messages along to our users. To facilitate this, we write an `errorCoerce` function so that we can use list functions with our `Error newtype`.

9.1 The problem

The first thing we would like to do is distinguish between errors caused by the username input and errors caused by the password input. Currently, we only have a way to do that when the problem is caused by one (or both) of the inputs being too long, because we have separate length-checking functions for different inputs, so the error can say whether the username or password failed.

```
*Main> main
Please enter a username.
abcdefghijklmnopqrstuvwxyz
Please enter a password.
abcdefghijklmnopqrstuvwxyz
Failure (Error [
```

```
    "Username cannot be longer than 15 characters.",
    "Your password cannot be longer than 20 characters."])
```

But since we used `requireAlphaNum` and `cleanWhitespace` for ***both*** sorts of inputs, the error messages are not specific to which input caused the failure.

```
*Main> main
Please enter a username.

Please enter a password.

Failure (Error ["Cannot be empty.","Cannot be empty."])
```

Instead we might prefer to see something like this:

```
*Main> main
Please enter a username.

Please enter a password.

Invalid username:
Cannot be empty.
Invalid password:
Cannot be empty.
```

We might at first think we could handle this in the `validateUsername` and `validatePassword` functions, since we have a clear distinction there between the inputs. However, the way they're structured right now, the `Failure` case is the failure of the application of `cleanWhitespace` to the input, which means it only returns ***one*** failure; the other failures would accumulate after `cleanWhitespace` succeeds. There are ways you could restructure this, but since that function does what it's meant to do already, a simpler solution might be to write new functions. Generally small functions that do one thing well are preferable to large functions that try to do a lot of things.

The new functions will branch on the results of the `validate` functions. When the `validate` function has succeeded, it will return the input as a `Success` value. When the `validate` function returns errors, the `Failure` case will prepend a label to the list of errors and format them.

9.2 The error functions

We'll call the first of this pair of functions `passwordErrors` to indicate its role in our program. The type is the same as the type of `validatePassword`. We can think of it as a complementary function to `validatePassword`; where `validatePassword` does nothing interesting in its `Failure` branch, this one will do nothing interesting in its `Success` branch.

We open a `case` expression that branches on the output of `validatePassword password`. In the `Failure` case, that returns a value we've called `err`; we know from the type that that value is a list of strings wrapped in the `Error` constructor. What we want to do is have another string, something like `"Invalid password: "`, and concatenate that with the rest of the strings.

The cool thing is we already have a way to concatenate `Error` values directly, a method that takes two `Error` lists of strings and concatenates the two lists of strings to return a single `Error` value: it's the `Semigroup` instance we wrote!

We can use `<>` to concatenate two `Error` lists, so we can make our error message label an `Error` value and then concatenate them:

```
passwordErrors :: Password -> Validation Error Password
passwordErrors password =
   case validatePassword password of
      Failure err -> Failure (Error ["Invalid password:"]
                              <> err)
      Success password2 -> Success password2
```

As mentioned above, this function doesn't need to do anything to the `Success` value; it only exists to help us identify the source of errors.

The `usernameErrors` function, then, will look quite similar to the function we just wrote. This time the error label will specify that it's the username that's invalid and concatenate that with any errors that result on the username side of the inputs.

```
usernameErrors :: Username -> Validation Error Username
usernameErrors username =
   case validateUsername username of
      Failure err -> Failure (Error ["Invalid username:"]
```

```
                                <> err)
        Success username2 -> Success username2
```

Now we have two functions that accumulate all the error messages from the validation process and concatenate them into two lists, with labels telling us which input failed.

9.3 Gathering up the errors

To combine both of those lists of errors into a single list, we change `makeUser` to use the `Errors` functions instead of the `validate` functions.

```
makeUser :: Username -> Password -> Validation Error User
makeUser name password =
    User
        <$> usernameErrors name
        <*> passwordErrors password
```

Again, because it's using the `Validation Applicative`, it will accumulate all the errors from `usernameErrors` and `passwordErrors` and since those functions include the labels as part of the error messages they return, the labels will persist into our final, single list result.

```
*Main> main
Please enter a username.
>
Please enter a password.
>
Failure (Error ["Invalid username:","Cannot be empty.",
                "Invalid password:","Cannot be empty."])
```

9.4 Lists upon lists

Let's finish our program by cleaning up the way we display the messages on the screen to the user. We mentioned earlier that we want to also clean up the way the `Success` messages print to the terminal, as well as the `Failures`. So we'll write a new function for this.

This function will rely on `makeUser` but format the errors and success values for the screen. Since this function is going to include the instructions for printing to the screen, it will return the `IO ()` type (meaning it performs effects rather than returning a value).

```
displayErrors :: Username -> Password -> IO ()
displayErrors name password =
    case makeUser name password of
        Failure err -> _
```

As we've said, that `err` value is a list of strings wrapped in an `Error` constructor. We want a function that takes a list of strings and returns a string that has newline separators so it will print nicely to the screen. The function we want is called `unlines`.

```
*Main> :type unlines
unlines :: [String] -> String
```

```
*Main> unlines ["Julie", "loves", "Haskell"]
"Julie\nloves\nHaskell\n"
it :: String
```

When we combine the output of `unlines` with `putStrLn` we'll get a nice formatting for the screen.

```
*Main> putStrLn (unlines ["Julie", "loves", "Haskell"])
Julie
loves
Haskell
```

That will look nicer once each string is an error message sentence rather than a single word.

9.5 Coercion

That combination of `putStrLn` and `unlines` will work – except our list of strings is wrapped in the `Error` constructor, so we need to first get it out of there.

Since `Error` and `[String]` are representationally equivalent, you might think there would be a built-in easy way to apply functions that need a `[String]` input directly to an `Error` value and make that work. You're not

far off, but since they are ***nominally*** different types, we have to ***coerce*** the `Error` value into its underlying type.[1]

As a reminder of what we said about `newtype` in chapter 6: Even though it gives us this bit of extra coercion obligation, it's good that `Error` and `[String]` are different nominal types, because:

- The distinction allows us to have different typeclass instances for them.
- Explicit type conversions make it clear in our code that we have points where we begin to think about data in a different light, for example that we now want to shift from an `Error` to a `[String]` mindset.

There isn't much magic here. We'll take an `Error` value and return the `[String]` that it wraps. We unpack that `[String]` value by pattern matching on the constructor in the input, and then we return that value.

```
errorCoerce :: Error -> [String]
errorCoerce (Error err) = err
```

Now if we use `errorCoerce` on our `Error err` values first, we can use `unlines` on the `err` value, and then `putStrLn` on the value that returns.

```
display :: Username -> Password -> IO ()
display name password =
    case makeUser name password of
        Failure err -> putStrLn (unlines (errorCoerce err))
        Success (User name password) -> _
```

That takes care of our error messages. Now we can think through how to reward a user who presents worthy input.

[1]We are doing it this way very much with a specific purpose in mind, and that is introducing you to the `coerce` function in `base`. You will see `coerce` used a lot in `newtype`-heavy code. It is safe, does not break type inference, imposes no runtime cost, and so it is very convenient. However, we wanted to show you how easy it is to write that function as well. So many things are like this in Haskell's `base` package: yes, someone wrote it for you, but we could just as well have written it ourselves.

9.6 Handling success

In the `Success` case, we will be returning a `User` value that has both a `Username` and a `Password` value. However, we typically do not want to display the password value on the screen, so we don't want to display that whole `User` value. What we might want to do, though, is welcome our user by name, so that's what we'll work on.

We'll concatenate the name with a welcome message and then use `putStrLn` again to display it to the screen.

```
display :: Username -> Password -> IO ()
display name password =
    case makeUser name password of
        Failure err -> putStrLn (unlines (errorCoerce err))
        Success (User name password) ->
            putStrLn ("Welcome, " ++ name)
```

That doesn't quite work.

```
error:
   • Couldn't match expected type ‘[Char]’ with actual
       type ‘Username’
   • In the second argument of ‘(++)’, namely ‘name’
     In the first argument of ‘putStrLn’, namely
       ‘("Welcome, " ++ name)’
     In the expression: putStrLn ("Welcome, " ++ name)
  |
  |       putStrLn ("Welcome, " ++ name)
  |                                ^^^^
```

The compiler error tells us what's wrong:

1. The type error:

   - `expected type ‘[Char]’`
   - `actual type ‘Username’`

2. Where it occurred:

```
       putStrLn ("Welcome, " ++ name)
                                ^^^^
```

`(++)` takes two `String` inputs, but the name we want to display here is a `String` wrapped in a `Username` constructor. It's a very similar problem as we had with the errors, but we'll take a different approach this time.

```
display :: Username -> Password -> IO ()
display name password =
    case makeUser name password of
        Failure err -> putStrLn (unlines (errorCoerce err))
        Success (User (Username name) password) ->
            putStrLn ("Welcome, " ++ name)
```

It's really the same approach we took before, just that we were in a context where we were pattern matching already, and so it was convenient just to alter the pattern instead of introducing a new function.

We have achieved our goal with this; the only remaining thing to do is incorporate this into `main` so that running the program includes the printing instructions in `display`.

9.7 The final `main`

As we've discussed, `main` has the type `IO ()` because it is expected to only produce effects when the program is run, not to return a value the way a pure function does. We have been using `print` in the last statement of `main` in order to return the `User` value as an `IO` action that prints it to the terminal, but now `display` (by using `putStrLn` internally) is doing that. So we replace the combination of `print` and a pure function with `display`.

```
main :: IO ()
main =
  do
    putStrLn "Please enter a username."
    username <- Username <$> getLine
    putStrLn "Please enter a password."
    password <- Password <$> getLine
    display username password
```

Here we find failure:

```
*Main> main
Please enter a username.
```

```
>
Please enter a password.
>
Invalid username:
Cannot be empty.
Invalid password:
Cannot be empty.
```

And here we find success.

```
*Main> main
Please enter a username.
julie
Please enter a password.
julielovesbooks
Welcome, julie
```

Appendix A

API reference

A.1 Types

A.1.1 `Maybe`

```
data Maybe a = Nothing | Just a

instance Functor Maybe
instance Applicative Maybe
instance Monad Maybe
instance Show a => Show (Maybe a)
```

A.1.2 `Either`

```
data Either a b = Left a | Right b

instance Functor (Either a)
instance Applicative (Either a)
instance Monad (Either a)
instance (Show a, Show b) => Show (Either a b)
```

A.1.3 Validation

```
data Validation err a = Failure err | Success a

instance Functor (Validation err)
instance Semigroup err => Applicative (Validation err)

bindValidation :: Validation err a
               -> (a -> Validation err b)
               -> Validation err b
```

A.1.4 List

```
instance Semigroup [a]

(++) :: [a] -> [a] -> [a]

all :: (a -> Bool) -> [a] -> Bool

map :: (a -> b) -> [a] -> [b]

null :: [a] -> Bool

reverse :: [a] -> [a]

unlines :: [String] -> String
```

A.1.5 IO

```
instance Applicative IO
instance Functor IO
instance Monad IO

print :: Show a => a -> IO ()
print x = putStrLn (show x)

putStrLn :: String -> IO ()

getLine :: IO String
```

A.2 Typeclasses

A.2.1 Functor

```
class Functor (f :: * -> *) where
    fmap :: (a -> b) -> f a -> f b

(<$>) = fmap
```

A.2.2 Applicative

```
class Functor f => Applicative (f :: * -> *) where
    pure :: a -> f a
    (<*>) :: f (a -> b) -> f a -> f b
    (*>) :: f a -> f b -> f b
    (<*) :: f a -> f b -> f a
```

A.2.3 Monad

```
class Applicative m => Monad (m :: * -> *) where
    (>>=) :: m a -> (a -> m b) -> m b
```

A.2.4 Semigroup

```
class Semigroup (a :: *) where
    (<>) :: a -> a -> a
```

A.2.5 Show

```
class Show (a :: *) where
    show :: a -> String
```

A.2.6 Coercible

```
coerce :: Coercible a b => a -> b
```

Appendix B

Solutions to exercises

Exercise 1 – Absolute value

```
absVal x =
    case (x < 0) of
        True -> negate x
        False -> x
```

Exercise 2 – (Bool, Bool)

```
validateUsernamePassword username password =
    case (null username, null password) of
        (False, False) -> "Okay"
        (True, False) -> "Empty username"
        (False, True) -> "Empty password"
        (True, True) -> "Empty username and password"
```

Exercise 3 – A question of types The reason this doesn't work is that the return types of the two "cases" have different types, and the return type of the function only matches the case where we return `x`.

This case returns a value of type `[a]`.

```
head' [] = []
```

Even though it's an empty list, it's still a value from the list datatype. The other case returns a value of type `a`, namely the first value in the list. The type of the function says we will return a value of type `a` from the list, not the list itself.

It's similar in this regard to the situation with `if-then-else` expressions, where the return types of the expression in the `then` and in the `else` must match.

Exercise 4 – Maybe for safety

```
tail' [] = Nothing
tail' (x:xs) = Just xs
```

```
head' [] = Nothing
head' (x:xs) = Just x
```

Exercise 5 – Palindromes

```
import Data.Char (isAlpha, toLower)
```

```
main :: IO ()
main =
  do
    putStrLn "Please enter a word."
    word <- getLine
    print (checkPalindrome word)
```

```
checkPalindrome :: String -> String
checkPalindrome word =
    case (isWord word) of
        Nothing -> "The word is invalid."
        Just word ->
            case (isPalindrome word) of
                False -> "This word is not palindrome."
                True -> "This word is a palindrome."
```

```
isPalindrome :: String -> Bool
isPalindrome word = (word == reverse word)
```

A basic `isWord` that rejects empty input:

```
isWord :: String -> Maybe String
isWord [] = Nothing
isWord word = Just word
```

A more advanced `isWord` implementation:

```
isWord word =
    case (filter isAlpha word) of
        [] -> Nothing
        word -> Just (map toLower word)
```

This second version

1. Removes non-letter characters;
2. Rejects the word if the result is empty; and
3. Converts all of the letters to lower case.

Exercise 6 – Hacker voice This is one possible way to accomplish this task:

```
import Data.Char (toLower)
```

```
substituteChar :: Char -> Char
substituteChar c =
    case c of
        'e' -> '3'
        'a' -> '4'
        'o' -> '0'
        't' -> '7'
        _   -> c
```

You might have added more cases to translate more characters.

`lowercaseWord` is not strictly necessary, but we didn't write cases to translate uppercase letters:

```
lowercaseWord :: String -> String
lowercaseWord xs = map toLower xs
```

Mapping the character substitution function over a list of characters:

```
translateWord :: String -> String
translateWord xs = map substituteChar (lowercaseWord xs)
```

Another possibility is lower-casing and translating in one pass with function composition:

```haskell
translateWord :: String -> String
translateWord xs = map (substituteChar . toLower) xs
```

```haskell
main :: IO ()
main =
  do
    word <- getLine
    print (translateWord word)
```

Exercise 7 – Minimum length

```haskell
checkPasswordLength password =
    case (length password < 10 || length password > 20) of
        True -> Nothing
        False -> Just password
```

Exercise 8 – Combine them all Answered in chapter 4.

Exercise 9 – Vacuous truth `all isAlphaNum ""` returns `True`, so `requireAlphaNum ""` returns `Just ""`. There aren't any characters in an empty string to test, so it is true that *all* of the characters in the string are alphanumeric. Alternatively, think of it this way: It *isn't* true that any of them is *not* alphanumeric. We call such a statement about all of nothing a "vacuous truth."

Exercise 10 – Doing I/O

```haskell
reverseDo :: IO ()
reverseDo =
  do
    word <- getLine
    print (reverse word)
```

Exercise 11 – The bind function for Maybe You need to pattern match on `Maybe` to handle both cases, either by defining `bindMaybe` piecemeal across two definitions:

```haskell
bindMaybe Nothing func = Nothing
bindMaybe (Just x) func = func x
```

or by using a `case` expression:

```
bindMaybe m func =
  case m of
    Nothing -> Nothing
    Just x -> func x
```

If you got the infinite type error, like this:

```
• Occurs check: cannot construct the infinite type:
        b ~ Maybe b
```

Welcome to the club! It means that your implementation says that `b` is `Maybe b` and since `Maybe b` contains `b`, this loop would never end. It probably means you wrote something like this:

```
bindMaybe Nothing func = Nothing
bindMaybe (Just x) func = Just (func x)
```

Since `func x` is already producing a `Maybe b` value, by applying that extra `Just` constructor to it, you've said that the `b` in `Maybe b` is itself `Maybe b`, thus implying that the type of `b` is `Maybe (Maybe (Maybe (Maybe (Maybe ...))))`.

Exercise 12 – The bind function for "StringOrValue" Notice the similarities to the `bindMaybe` function from the previous exercise.

```
bindStringOrValue
    :: StringOrValue a
    -> (a -> StringOrValue b)
    -> StringOrValue b
bindStringOrValue (Str xs) func = Str xs
bindStringOrValue (Val x) func = func x
```

Exercise 13 – Converting to Either

```
requireAlphaNum :: String -> Either String String
requireAlphaNum password =
    case (all isAlphaNum password) of
        False -> Left "Your password cannot contain \
                      \white space or special characters."
        True -> Right password
```

```haskell
cleanWhitespace :: String -> Either String String
cleanWhitespace "" = Left "Your password cannot be empty."
cleanWhitespace (x : xs) =
    case (isSpace x) of
        True -> cleanWhitespace xs
        False -> Right (x : xs)
```

Exercise 15 – Testing

```haskell
test = printTestResult $
  do
    eq 1 (checkPasswordLength "") (Right "")
    eq 2 (checkPasswordLength "julielovesbooks")
         (Right "julielovesbooks")
    eq 3 (checkPasswordLength "abcdefghijklmnopqrstuvwxyz")
         (Left "Your password cannot be longer \
               \than 20 characters.")
    eq 4 (requireAlphaNum "") (Right "")
    eq 5 (requireAlphaNum "julielovesbooks")
         (Right "julielovesbooks")
    eq 6 (requireAlphaNum "julie loves books")
         (Left "Your password cannot contain white \
               \space or special characters.")
    eq 7 (cleanWhitespace "     ")
         (Left "Your password cannot be empty.")
    eq 8 (cleanWhitespace "      julielovesbooks")
         (Right "julielovesbooks")
```

Exercise 21 – Fitting in

```haskell
makeUserTmpPassword name =
    User <$> validateUsername ""
         <*> Success (Password "temporaryPassword")
```

Exercise 23 – Applicative I/O

```haskell
main =
  do
    result <- checkAnagram <$> promptWord1 <*> promptWord2
    print result
```