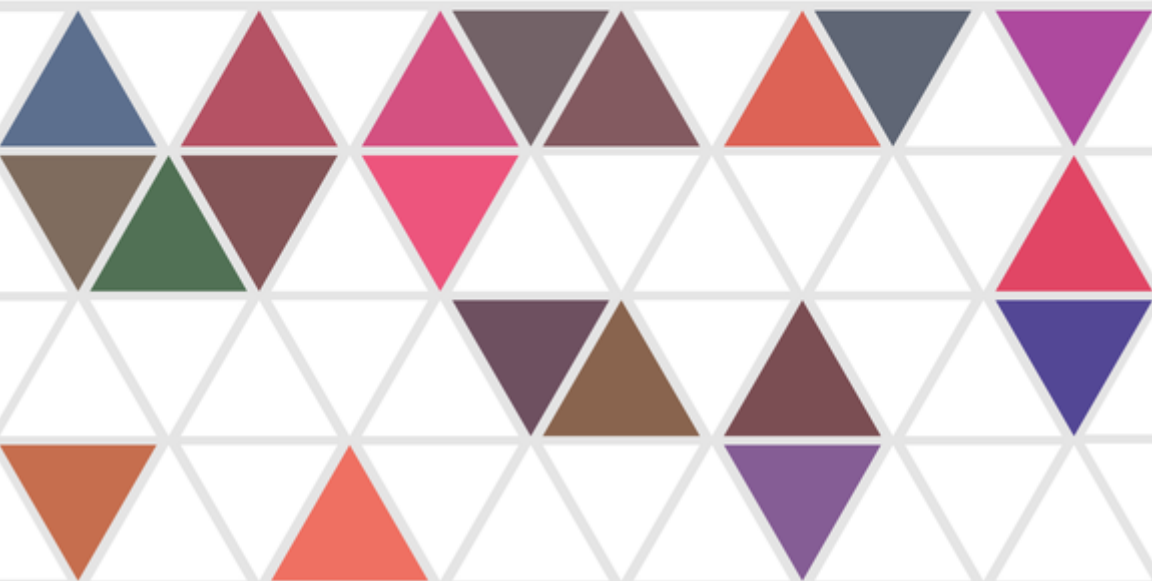
Maybe Haskell

by Pat Brisbin

thoughtbot



Maybe Haskell

Pat Brisbin

May 4, 2015

Contents

Introduction

As a programmer, I spend a lot of time dealing with the fallout from one specific problem: partial functions. A partial function is one that can't provide a valid result for all possible inputs. If you write a function (or method) to return the first element in an array that satisfies some condition, what do you do if no such element exists? You've been given an input for which you can't return a valid result. Aside from raising an exception, what can you do?

The most popular way to to deal with this is to return a special value that indicates failure. Ruby has `nil`, Java has `null`, and many C functions return `-1` in failure cases. This is a huge hassle. You now have a system in which any value at any time can either be the value you expect or `nil`, always.

For instance, if you try to find a `User`, and then treat the value you get back as if it's a `User` but it's actually `nil`, you get a `NoMethodError`. What's worse, that error may not happen anywhere near the problem's source. The line of code that created that `nil` may not even appear in the eventual backtrace. The result is various "`nil` checks" peppered throughout the code. Is this the best we can do?

The problem of partial functions is not going away. User input may be invalid, files may not exist, networks may fail. We will always need a way to deal with partial functions. What we don't need is `null`.

An Alternate Solution

In languages with sufficiently expressive type systems, we have another option: we can state in the types that certain values may not be present. Functions that

typically are written in a partial way can instead be defined to return a type that captures any potential non-presence. Not only does this make it explicit and "type checked" that you have code to handle the case when a value isn't present, it also means that if a value is *not* of this special "nullable" type, you can feel safe in assuming the value is really there. In short: no `nil` checks are required.

The focus of this book will be how Haskell implements this idea via the `Maybe` data type. This type and all the functions that deal with it are not built-in, language-level constructs. Instead, this is all implemented as libraries, written in a very straightforward way. In fact, we'll write most of that code ourselves over the course of this short book.

Haskell is not the only language to have such a construct. For example, Scala has a similar `Option` type and Swift has `Optional` with various built-in syntax elements to make its usage more convenient. Many of the ideas implemented in these languages were lifted directly from Haskell. If you happen to use one of them, it can be good to learn where the ideas originated.

Required Experience

I'll assume no prior Haskell experience. I expect that those reading this book will have programmed in other, more traditional languages, but I'll also ask that you *actively combat* your prior programming experience.

For example, you're going to see code like this:

```
countEvens = length . filter even
```

This is a function definition written in an entirely different style than you may be used to. Even so, I'll bet you can guess what it does, and even get close to how it does it: `filter even` probably takes a list and filters it for only even elements. `length` probably takes a list and returns the number of elements it contains.

Given those fairly obvious facts, you might guess that putting two things together with `(.)` must mean you do one and then give the result to the other. That makes this expression a function that must take a list and return the number of even elements it contains. Without mentioning any actual argument, we can directly assign this function the name `countEvens`. There's no need in Haskell to say that

count-evens *of some x* is to take the length after filtering for the even values *of that x*. We can state directly that count-evens is taking the length after filtering for evens.

This is a relatively contrived example, but it's an indication of the confusion that can happen at any level: if your first reaction is "such weird syntax! What is this crazy dot thing!?", you're going to have a bad time. Instead, try to internalize the parts that make sense while getting comfortable with *not* knowing the parts that don't. As you learn more, the various bits will tie together in ways you might not expect.

Structure

We'll spend this entire book focused on a single *type constructor* called `Maybe`. We'll start by quickly covering the basics of Haskell, but only far enough that we can see the opportunity for such a type and can't help but invent it ourselves. With that type defined, we'll quickly see that it's cumbersome to use. This is because Haskell has taken an inherently cumbersome concept and put it right in front of us by naming it, and by requiring we deal with it at every step.

From there, we'll explore three *type classes* whose presence will make our lives far simpler. We'll see that `Maybe` has all the properties required to call it a *functor*, an *applicative functor*, and even a *monad*. These terms may sound scary, but we'll go through them slowly, each concept building on the one before. These three *interfaces* are crucial to how I/O is handled in a purely functional language such as Haskell. Understanding them will open your eyes to a whole new world of abstractions and demystify some notoriously opaque topics.

Finally, with a firm grasp on how these concepts operate in the context of `Maybe`, we'll discuss other types that share these qualities. This is to reinforce the fact that these ideas are *abstractions*. They can be applied to any type that meets certain criteria. Ideas like *functor* and *monad* are not limited to, or specifically tied to, the concept of partial functions or nullable values. They apply much more broadly to things like lists, trees, exceptions, and program evaluation.

What This Book is Not

I don't intend to teach you Haskell. Rather, I want to show you *barely enough* Haskell that I can wade into some more interesting topics. I want to show how this `Maybe` data type can add safety to your code base while remaining convenient, expressive, and powerful. My hope is to show that Haskell and its "academic" ideas are not limited to PhD thesis papers. These ideas can result directly in cleaner, more maintainable code that solves practical problems.

I won't describe how to set up a Haskell programming environment, show you how to write and run complete Haskell programs, or dive deeply into every language construct that we'll encounter. If you are interested in going further and actually learning Haskell (and I hope you are!), then I recommend following Chris Allen's great learning path.

Finally, a word of general advice before you get started:

The type system is not your enemy. It's your friend. It doesn't slow you down; it keeps you honest. Keep an open mind. Haskell is simpler than you think. Monads are not some mystical burrito. They're a simple abstraction that, when applied to a variety of problems, can lead to elegant solutions. Don't get bogged down in what you don't understand. Instead, dig deeper into what you do. And above all, take your time.

Haskell Basics

When we declare a function in Haskell, we first write a type signature:

```
five :: Int
```

We can read this as `five` *of type* `Int`.

Next, we write a definition:

```
five = 5
```

We can read this as `five` *is* `5`.

In Haskell, `=` is not variable assignment, it's defining equivalence. We're saying here that the word `five` *is equivalent to* the literal `5`. Anywhere you see one, you can replace it with the other and the program will always give the same answer. This property is called *referential transparency* and it holds true for any Haskell definition, no matter how complicated.

It's also possible to specify types with an *annotation* rather than a signature. We can annotate any expression with `:: <type>` to explicitly tell the compiler the type we want (or expect) that expression to have.

```
almostThird = (3 :: Float) / 9
-- => 0.3333334

actualThird = (3 :: Rational) / 9
-- => 1 % 3
```

We can read these as `almostThird` is `3`, *of type* `Float`, divided by `9` and `actualThird` is `3`, *of type* `Rational`, divided by `9`.

Type annotations and signatures are usually optional, as Haskell can almost always tell the type of an expression by inspecting the types of its constituent parts or seeing how it is eventually used. This process is called *type inference*. For example, Haskell knows that `actualThird` is a `Rational` because it saw that `3` is a `Rational`. Since you can only use `(/)` with arguments of the same type, it *enforced* that `9` is also a `Rational`. Knowing that `(/)` returns the same type as its arguments, the final result of the division must itself be a `Rational`.

Good Haskellers will include a type signature on all top-level definitions anyway. It provides executable documentation and may, in some cases, prevent errors that occur when the compiler assigns a more generic type than you might otherwise want.

Arguments

Defining functions that take arguments looks like this:

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

The type signature can be confusing because the argument types are not separated from the return type. There is a good reason for this, but I won't go into it yet. For now, feel free to mentally treat the thing after the last arrow as the return type.

After the type signature, we give the function's name (`add`) and names for any arguments it takes (`x` and `y`). On the other side of the `=`, we define an expression using those names.

Higher-order functions

Functions can take and return other functions. These are known as higher-order functions. In type signatures, any function arguments or return values must be surrounded by parentheses:

```
twice :: (Int -> Int) -> Int -> Int
twice f x = f (f x)

twice (add 2) 3
-- => 7
```

`twice` takes as its first argument a function of type `(Int -> Int)`. As its second argument, it takes an `Int`. The body of the function applies the first argument (`f`) to the second (`x`) twice, returning another `Int`. The parentheses in the definition of `twice` indicate grouping, not application. In Haskell, applying a function to some argument is simple: stick them together with a space in between. In this case, we need to group the inner `(f x)` so the outer `f` is applied to it as single argument. Without these parentheses, Haskell would think we were applying `f` to two arguments: another `f` and `x`.

You also saw an example of *partial application*. The expression `add 2` returns a new function that itself takes the argument we left off. Let's break down that last expression to see how it works:

```
-- Add takes two Ints and returns an Int
add :: Int -> Int -> Int
add x y = x + y

-- Supplying only the first argument gives us a new function that will add 2 to
-- its argument. Its type is Int -> Int
add 2 :: Int -> Int

-- Which is exactly the type of twice's first argument
twice :: (Int -> Int) -> Int -> Int
twice f x = f (f x)

twice (add 2) 3
-- => add 2 (add 2 3)
-- => add 2 5
-- => 7
```

It's OK if this doesn't make complete sense now. I'll talk more about partial application as we go.

Operators

In the definition of `add`, I used something called an *operator*: `(+)`. Operators like this are not in any way special or built-in; we can define and use them like any other function. That said, operators have three additional (and convenient) behaviors:

1. They are used *infix* by default, meaning they appear between their arguments (i.e. `2 + 2`, not `+ 2 2`). To use an operator *prefix*, it must be surrounded in parentheses (as in `(+) 2 2`).
2. When defining an operator, we can assign custom associativity and precedence relative to other operators. This tells Haskell how to group expressions like `2 + 3 * 5 / 10`.
3. We can surround an operator and *either* of its arguments in parentheses to get a new function that accepts whichever argument we left off. Expressions like `(+ 2)` and `(10 /)` are examples. The former adds `2` to something and the latter divides `10` by something. Expressions like these are called *sections*.

In Haskell, any function with a name made up entirely of punctuation (where The Haskell Report states very precisely what "punctuation" means) behaves like an operator. We can also take any normally named function and treat it like an operator by surrounding it in backticks:

```
-- Normal usage of an elem function for checking if a value is present in a list
elem 3 [1, 2, 3, 4, 5]
-- => True

-- Reads a little better infix
3 `elem` [1, 2, 3, 4, 5]
-- => True

-- Or as a section, leaving out the first argument
intersects xs ys = any (`elem` xs) ys
```

Lambdas

The last thing we need to know about functions is that they can be *anonymous*. Anonymous functions are called *lambdas* and are most frequently used as argu-

ments to higher-order functions. Often these functional arguments exist for only a single use and giving them a name is not otherwise valuable.

The syntax is a back-slash, followed by the arguments to the function, an arrow, and finally the body of the function. A back-slash is used because it looks similar to the Greek letter λ.

Here's an example:

```
twice (\x -> x * x + 10) 5
-- => 1235
```

If you come across a code example using a lambda, you can always rewrite it to use named functions. Here's the process for this example:

```
-- Grab the lambda
\x -> x * x + 10

-- Name it
f = \x -> x * x + 10

-- Replace "\... ->" with normal arguments
f x = x * x + 10

-- Use the name instead
twice f 5
```

Our Own Data Types

We're not limited to basic types like `Int` or `String`. As you might expect, Haskell allows you to define custom data types:

```
data Person = MakePerson String Int
--                       |      |
--                       |      ` The persons's age
--                       |
--                       ` The person's name
```

To the left of the `=` is the *type* constructor and to the right can be one or more *data* constructors. The type constructor is the name of the type, which is used in type signatures. The data constructors are functions that produce values of the given type. For example, `MakePerson` is a function that takes a `String` and an `Int`, and returns a `Person`. Note that I will often use the general term "constructor" to refer to a *data* constructor if the meaning is clear from the context.

When working with only one data constructor, it's quite common to give it the same name as the type constructor. This is because it's syntactically impossible to use one in place of the other, so the compiler makes no restriction. Naming is hard. So when you have a good name, you might as well use it in both contexts.

```
data Person = Person String Int
--   |        |
--   |        ` Data constructor
--   |
--   ` Type constructor
```

Once we have declared the data type, we can now use it to write functions that construct values of this type:

```
pat :: Person
pat = Person "Pat" 29
```

Pattern Matching

To get the individual parts back out again, we use pattern matching:

```
getName :: Person -> String
getName (Person name _) = name

getAge :: Person -> Int
getAge (Person _ age) = age
```

In the definitions above, each function is looking for values constructed with `Person`. If it gets an argument that matches (which is guaranteed since that's

the only way to get a `Person` in our system so far), Haskell will use that function body with each part of the constructed value bound to the variables given. The `_` pattern (called a *wildcard*) is used for any parts we don't care about. Again, this is using `=` for equivalence (as always). We're saying that `getName`, when given `(Person name _)`, *is equivalent to* `name`. It works similarly for `getAge`.

Haskell offers other ways to do this sort of thing, but we won't get into those here.

Sum Types

As mentioned earlier, types can have more than one data constructor. These are called *sum types* because the total number of values you can build of a sum type is the sum of the number of values you can build with each of its constructors. The syntax is to separate each constructor by a `|` symbol:

```
data Person = PersonWithAge String Int | PersonWithoutAge String

pat :: Person
pat = PersonWithAge "Pat" 29

jim :: Person
jim = PersonWithoutAge "Jim"
```

Notice that `pat` and `jim` are both values of type `Person`, but they've been constructed differently. We can use pattern matching to inspect how a value was constructed and accordingly choose what to do. Syntactically, this is accomplished by providing multiple definitions of the same function, each matching a different pattern. Each definition will be tried in the order defined, and the first function to match will be used.

This works well for pulling the name out of a value of our new `Person` type:

```
getName :: Person -> String
getName (PersonWithAge name _) = name
getName (PersonWithoutAge name) = name
```

But we must be careful when trying to pull out a person's age:

```
getAge :: Person -> Int
getAge (PersonWithAge _ age) = age
getAge (PersonWithoutAge _) = -- uh-oh
```

If we decide to be lazy and not define that second function body, Haskell will compile, but warn us about a *non-exhaustive* pattern match. What we've created at that point is a *partial function*. If such a program ever attempts to match `getAge` with a `Person` that has no age, we'll see one of the few runtime errors possible in Haskell.

A person's name is always there, but their age may or may not be. Defining two constructors makes both cases explicit and forces anyone attempting to access a person's age to deal with its potential non-presence.

Kinds and Parameters

Imagine we wanted to generalize this `Person` type. What if people were able to hold arbitrary things? What if what that thing is (its type) doesn't really matter, if the only meaningful thing we can say about it is if it's there or not? What we had before was a *person with an age* or a *person without an age*. What we want now is a *person with a thing* or a *person without a thing*.

One way to do this is to *parameterize* the type:

```
data Person a = PersonWithThing String a | PersonWithoutThing String
--          |                          |
--          |                          ` we can use it as an argument here
--          |
--          ` By adding a "type variable" here
```

The type we've defined here is `Person a`. We can construct values of type `Person a` by giving a `String` and an `a` to `PersonWithThing`, or by giving only a `String` to `PersonWithoutThing`. Notice that even if we build our person using `PersonWithoutThing`, the constructed value still has type `Person a`.

The `a` is called a *type variable*. Any lowercase value will do, but it's common to use `a` because it's short, and a value of type `a` can be thought of as a value of *any* type.

Rather than hard-coding that a person has an `Int` representing their age (or not), we can say a person is holding some thing of type `a` (or not).

We can still construct people with and without ages, but now we have to specify in the type that in this case the `a` is an `Int`:

```
patWithAge :: Person Int
patWithAge = PersonWithThing "pat" 29

patWithoutAge :: Person Int
patWithoutAge = PersonWithoutThing "pat"
```

Notice how even in the case where I have no age, we still specify the type of that thing that I do not have. In this case, we specified an `Int` for `patWithoutAge`, but values can have (or not have) any type of thing:

```
patWithEmail :: Person String
patWithEmail = PersonWithThing "pat" "pat@thoughtbot.com"

patWithoutEmail:: Person String
patWithoutEmail = PersonWithoutThing "pat"
```

We don't have to give a concrete `a` when it doesn't matter. `patWithoutAge` and `patWithoutEmail` are the same value with different types. We could define a single value with the generic type `Person a`:

```
patWithoutThing :: Person a
patWithoutThing = PersonWithoutThing "pat"
```

Because `a` is more general than `Int` or `String`, a value such as this can stand in anywhere a `Person Int` or `Person String` is needed:

```
patWithoutAge :: Person Int
patWithoutAge = patWithoutThing

patWithoutEmail :: Person String
patWithoutEmail = patWithoutThing
```

Similarly, functions that operate on people can choose whether they care about what the person's holding–or not. For example, getting someone's name shouldn't be affected by whether they hold something, so we can leave it unspecified:

```
getName :: Person a -> String
getName (PersonWithThing name _) = name
getName (PersonWithoutThing name) = name

getName patWithAge
-- => "pat"

getName patWithoutEmail
-- => "pat"
```

But a function that does care must specify both the type *and* account for the case of non-presence:

```
doubleAge :: Person Int -> Int
doubleAge (PersonWithThing _ age) = 2 * age
doubleAge (PersonWithoutThing _) = 1

doubleAge patWithAge
-- => 58

doubleAge patWithoutAge
-- => 1

doubleAge patWithoutThing
-- => 1

doubleAge patWithoutEmail
-- => Type error! Person String != Person Int
```

In this example, `doubleAge` had to account for people that had no age. The solution it chose was a poor one: return the doubled age or `1`. A better choice is to not return an `Int`; instead, return some type capable of holding both the doubled age and the fact that we might not have had an age to double in the first place. What we need is `Maybe`.

Maybe

Haskell's `Maybe` type is very similar to our `Person` example:

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

The difference is that we're not dragging along a name this time. This type is only concerned with representing a value (of any type) that is either *present* or *not*.

We can use this to take functions that otherwise would be *partial* and make them *total*:

```
-- | Find the first element from the list for which the predicate function
--   returns True. Return Nothing if there is no such element.
find :: (a -> Bool) -> [a] -> Maybe a
find _ [] = Nothing
find predicate (first:rest) =
    if predicate first
        then Just first
        else find predicate rest
```

This function has two definitions matching two different patterns: if given the empty list, we immediately return `Nothing`. Otherwise, the (non-empty) list is deconstructed into its `first` element and the `rest` of the list by matching on the `(:)` (pronounced *cons*) constructor. Then we test whether applying the `predicate` function to `first` returns `True`. If it does, we return `Just` that. Otherwise, we recurse and try to find the element in the `rest` of the list.

Returning a `Maybe` value forces all callers of `find` to deal with the potential `Nothing` case:

```
--
-- Warning: this is a type error, not working code!
--
findUser :: UserId -> User
findUser uid = find (matchesId uid) allUsers
```

This is a type error since the expression actually returns a `Maybe User`. Instead, we have to take that `Maybe User` and inspect it to see if something's there or not. We can do this via `case`, which also supports pattern matching:

```
findUser :: UserId -> User
findUser uid =
    case find (matchesId uid) allUsers of
        Just u  -> u
        Nothing -> -- what to do? error?
```

Depending on your domain and the likelihood of Maybe values, you might find this sort of "stair-casing" propagating throughout your system. This can lead to the thought that `Maybe` isn't really all that valuable over some *null* value built into the language. If you need these `case` expressions peppered throughout the code base, how is that better than the analogous "`nil` checks"?

Don't Give Up

The above might leave you feeling underwhelmed. That code doesn't look all that better than the equivalent Ruby:

```
def find_user(uid)
  if user = all_users.detect? { |u| u.matches_id?(uid) }
    user
  else
    # what to do? error?
  end
end
```

First of all, the Haskell version is type safe: `findUser` must always return a `User` since that's the type we've specified. I'd put money on most Ruby developers returning `nil` from the `else` branch. The Haskell type system won't allow that and that's a good thing. Otherwise, we have these values floating throughout our system that we assume are there and in fact are not. I understand that without spending time programming in Haskell, it's hard to see the benefits of ruthless type safety

employed at every turn. I assure you it's a coding experience like no other, but I'm not here to convince you of that – at least not directly.

The bottom line is that an experienced Haskeller would not write this code this way. `case` is a code smell when it comes to `Maybe`. Almost all code using `Maybe` can be improved from a tedious `case` evaluation using one of the three abstractions we'll explore in this book.

Let's get started.

Functor

In the last chapter, we defined a type that allows any value of type `a` to carry with it additional information about whether it's actually there or not:

```
data Maybe a = Nothing | Just a

actuallyFive :: Maybe Int
actuallyFive = Just 5

notReallyFive :: Maybe Int
notReallyFive = Nothing
```

As you can see, attempting to get at the value inside is dangerous:

```
getValue :: Maybe a -> a
getValue (Just x) = x
getValue Nothing = error "uh-oh"

getValue actuallyFive
-- => 5

getValue notReallyFive
-- => Runtime error!
```

At first, this seems severely limiting: how can we use something if we can't (safely) get at the value inside?

Choices

When confronted with some `Maybe a`, and you want to do something with an `a`, you have three choices:

1. Use the value if you can, otherwise throw an exception
2. Use the value if you can, but still have some way of returning a valid result if the value's not there
3. Pass the buck and return a `Maybe` result yourself

The first option is a non-starter. As you saw, it is possible to throw runtime exceptions in Haskell via the `error` function, but you should avoid this at all costs. We're trying to eliminate runtime exceptions, not add them.

The second option is possible only in certain scenarios. You need to have some way to handle an incoming `Nothing`. That may mean skipping certain aspects of your computation or substituting another appropriate value. Usually, if you're given a completely abstract `Maybe a`, it's not possible to determine a substitute because you can't produce a value of type `a` out of nowhere.

Even if you did know the type (say you were given a `Maybe Int`) it would be unfair to your callers if you defined the safe substitute yourself. In one case `0` might be best because we're going to add something, but in another `1` would be better because we plan to multiply. It's best to let them handle it themselves using a utility function like `fromMaybe`:

```
fromMaybe :: a -> Maybe a -> a
fromMaybe x Nothing = x
fromMaybe _ (Just x) = x

fromMaybe 10 actuallyFive
-- => 5

fromMaybe 10 notReallyFive
-- => 10
```

Option 3 is actually a variation on option 2. By making your own result a `Maybe` you always have the ability to return `Nothing` yourself if the value isn't present. If the

value *is* present, you can perform whatever computation you need to and wrap what would be your normal result in `Just`.

The main downside is that now your callers also have to consider how to deal with the `Maybe`. Given the same situation, they should again make the same choice (option 3), but that only pushes the problem up to their callers–which means any `Maybe` values tend to go *viral*.

Eventually, probably at some UI boundary, someone will need to "deal with" the `Maybe`, either by providing a substitute or skipping some action that might otherwise take place. This should happen only once, at that boundary. Every function between the source and the final use should pass along the value's potential non-presence unchanged.

Even though it's safest for every function in our system to pass along a `Maybe` value, it would be extremely annoying to force them all to actually take and return `Maybe` values. Each function separately checking whether it should go ahead and perform its computations will become repetitive and tedious. Instead, we can completely abstract this "pass along the `Maybe`" concern using higher-order functions and something called *functors*.

Discovering a Functor

Imagine we had a higher-order function called `whenJust`:

```
whenJust :: (a -> b) -> Maybe a -> Maybe b
whenJust f (Just x) = Just (f x)
whenJust _ Nothing = Nothing
```

It takes a function from `a` to `b` and a `Maybe a`. If the value's there, it applies the function and wraps the result in `Just`. If the value's not there, it returns `Nothing`. Note that it constructs a new value using the `Nothing` constructor. This is important because the value we're given is type `Maybe a` and we must return type `Maybe b`.

This allows the internals of our system to be made of functions (e.g. the `f` given to `whenJust`) that take and return normal, non-`Maybe` values, but still "pass along the `Maybe`" whenever we need to take a value from some source that may fail and manipulate that value in some way. If it's there, we go ahead and manipulate it, but return the result as a `Maybe` as well. If it's not, we return `Nothing` directly.

```
whenJust (+5) actuallyFive
-- => Just 10

whenJust (+5) notReallyFive
-- => Nothing
```

This function exists in Haskell's Prelude[1] as `fmap` in the `Functor` type class.

About Type Classes

Haskell has a concept called *type classes*. These are not at all related to the classes used in Object-oriented programming. Instead, Haskell uses type classes for functions that may be implemented in different ways for different data types. These are more like the *interfaces* and *protocols* you may find in other languages. For example, we can add or negate various kinds of numbers: integers, floating points, rational numbers, etc. To accommodate this, Haskell has a `Num` type class that includes functions like `(+)` and `negate`. Each concrete type (`Int`, `Float`, etc) then defines its own version of the required functions.

Type classes are defined with the `class` keyword and a `where` clause listing the types of any *member functions*:

```
class Num a where
    (+) :: a -> a -> a

    negate :: a -> a
```

Being an *instance* of a type class requires that you implement any member functions with the correct type signatures. To make `Int` an instance of `Num`, someone defined the `(+)` and `negate` functions for it. This is done with the `instance` keyword and a `where` clause that implements the functions from the class declaration:

```
instance Num Int where
    x + y = addInt x y

    negate x = negateInt x
```

[1] The module of functions available without `importing` anything.

Usually, but not always, *laws* are associated with these functions that your implementations must satisfy. Type class laws are important for ensuring that type classes are useful. They allow us as developers to reason about what will happen when we use type class functions without having to understand all of the concrete types for which they are defined. For example, if you negate a number twice, you should get back to the same number. This can be stated formally as:

```
negate (negate x) == x -- for any x
```

Knowing that this law holds gives us a precise understanding of what will happen when we use `negate`. Because of the laws, we get this understanding without having to know how `negate` is implemented for various types. This is a simple example, but we'll see more interesting laws with the `Functor` type class.

Functor

Haskell defines the type class `Functor` with a single member function, `fmap`:

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

Type constructors, like `Maybe`, implement `fmap` by defining a function where that `f` is replaced by themselves. We can see that `whenJust` has the correct type:

```
--          (a -> b) -> f     a -> f     b
whenJust :: (a -> b) -> Maybe a -> Maybe b
```

Therefore, we could implement a `Functor` instance for `Maybe` with the following code:

```
instance Functor Maybe where
    fmap = whenJust
```

In reality, there is no `whenJust` function; `fmap` is implemented directly:

```
instance Functor Maybe where
    fmap f (Just x) = Just (f x)
    fmap _ Nothing = Nothing
```

This definition is exactly like the one we saw earlier for `whenJust`. The only difference is we're now implementing it as part of the `Functor` instance declaration for `Maybe`. For the rest of this book, I'll be omitting the `class` and `instance` syntax. Instead, I'll state in prose when a function is part of some type class but show its type and definition as if it was a normal, top-level function.

The Functor Laws

As mentioned, type class laws are a formal way of defining what it means for implementations to be "well-behaved." If someone writes a library function and says it can work with "any `Functor`", that code can rely both on that type having an `fmap` implementation, and on its operating in accordance with these laws.

The first Functor law

The first Functor law states:

```
fmap id x == id x
--
-- for any value x, of type f a (e.g. Maybe a)
--
```

Where `id` is the *identity* function, one which returns whatever you give it:

```
id :: a -> a
id x = x
```

Since pure functions always give the same result when given the same input, it's equally correct to say the functions themselves must be equivalent, rather than applying them to "any `x`" and saying the results must be equivalent. For this reason, the laws are usually stated as:

```
fmap id == id
```

This law says that if we call `fmap id`, the function we get back should be equivalent to `id` itself. This is what "well-behaved" means in this context. If you're familiar with the common `map` function on lists, you would expect that applying `id` to every element in a list (as `map id` does) gives you back the exact same list. That is exactly what you expect to get if you apply `id` directly to the list itself. That `map` function is actually `fmap` specialized to the `[]` type. Hence, that behavior follows from the first law.

Let's go through the same thought exercise for `Maybe` so you can see that this law holds for its implementation as well. We'll use our two example values `actuallyFive` and `notReallyFive` from earlier:

```
actuallyFive :: Maybe Int
actuallyFive = Just 5

notReallyFive :: Maybe Int
notReallyFive = Nothing
```

What do we get by applying the identity function to each of these?

```
id actuallyFive
-- => Just 5

id notReallyFive
-- => Nothing
```

Not too surprising. Now let's look at `fmap id`:

```
fmap id actuallyFive
```

Remember the definition of `fmap` for `Maybe` values:

```
fmap :: (a -> b) -> Maybe a -> Maybe b
fmap f (Just x) = Just (f x)
fmap _ Nothing = Nothing
```

Since `actuallyFive` matches the `Just` case, `fmap` will apply `id` to `5`, then re-wrap the result in `Just`:

```
fmap id actuallyFive
-- => fmap id (Just 5) = Just (id 5)
-- =>                  = Just 5
```

And for `notReallyFive`?

```
fmap id notReallyFive
```

Since `notReallyFive` is `Nothing`, `fmap` will return a new `Nothing`:

```
fmap id notReallyFive
-- => fmap _ Nothing = Nothing
-- =>                = Nothing
```

As expected, both results are the same as applying `id` directly.

The second Functor law

The second law has to do with order of operations. It states:

```
fmap (f . g) == fmap f . fmap g
```

Where `(.)` is a function that takes two functions and *composes* them together:

```
(.) :: (b -> c) -> (a -> b) -> a -> c
(f . g) x = f (g x)
```

What this law says is that if we compose two functions together, then `fmap` the resulting function, we should get a function that behaves the same as when we `fmap` each function individually, then compose the two results. Let's prove again that this law holds for `Maybe` by walking through an example with `actuallyFive` and `notReallyFive`.

First, let's define two concrete functions, `f` and `g`

```
f :: Int -> Int
f = (+2)

g :: Int -> Int
g = (*3)
```

We can *compose* these two functions to get a new function, and call that `h`:

```
h :: Int -> Int
h = f . g
```

Given the definition of `(.)`, this is equivalent to:

```
h :: Int -> Int
h x = f (g x)
```

This new function takes a number and gives it to `(*3)`, then it takes the result and gives it to `(+2)`:

```
h 5
-- => 17
```

We can give this function to `fmap` to get one that works with `Maybe` values:

```
fmap h actuallyFive
-- => Just 17

fmap h notReallyFive
-- => Nothing
```

Similarly, we can give each of `f` and `g` to `fmap` separately to produce functions that can add 2 or multiply 3 to a `Maybe Int` and produce another `Maybe Int`. The resulting functions can also be composed with `(.)` to produce a new function, `fh`:

```
fh :: Maybe Int -> Maybe Int
fh = fmap f . fmap g
```

Again, given the definition of `(.)`, this is equivalent to:

```
fh :: Maybe Int -> Maybe Int
fh x = fmap f (fmap g x)
```

This function will call `fmap g` on its argument, which will multiply by 3 if the number's there or return `Nothing` if it's not. Then it will give that result to `fmap f`, which will add 2 if the number's there, or return `Nothing` if it's not:

```
fh actuallyFive
-- => Just 17

fh notReallyFive
-- => Nothing
```

You should convince yourself that `fh` and `fmap h` behave in exactly the same way. The second functor law states that this must be the case if your type is to be a valid `Functor`.

Because Haskell is referentially transparent, we can freely replace functions with their implementations. It may require some explicit parentheses here and there, but the code will always give the same answer. Doing so brings us back directly to the statement of the second law:

```
(fmap f . fmap g) actuallyFive
-- => Just 17

fmap (f . g) actuallyFive
-- => Just 17

(fmap f . fmap g) notReallyFive
-- => Nothing

fmap (f . g) notReallyFive
-- => Nothing

-- Therefore:
fmap (f . g) == fmap f . fmap g
```

Not only can we take normal functions (those that operate on fully present values) and give them to `fmap` to get functions that can operate on `Maybe` values, but this law states we can do so in any order. We can compose our system of functions together *then* give that to `fmap` or we can `fmap` individual functions and compose *those* together. Either way, we're guaranteed to get the same result. We can rely on this fact whenever we use `fmap` for any type that's in the `Functor` type class.

Why Is This Useful?

OK, enough theory. Now that we know how it works, let's see how it's used. Say we have a lookup function to get from a `UserId` to the `User` for that id. Since the user may not exist, it returns a `Maybe User`:

```
findUser :: UserId -> Maybe User
findUser = undefined
```

I've left the implementation of `findUser` as `undefined` because this doesn't matter for our example. I'll do this frequently throughout the book. `undefined` is a function with type `a`. That allows it to stand in for any expression. If your program ever tries to evaluate it, it will raise an exception. Still, it can be extremely useful while developing because we can build our program incrementally, but have the compiler check our types as we go.

Next, imagine we want to display a user's name in all capitals:

```
userUpperName :: User -> String
userUpperName u = map toUpper (userName u)
```

The logic of getting from a `User` to that capitalized `String` is not terribly complex, but it could be. Imagine something like getting from a `User` to that user's yearly spending on products valued over $1,000. In our case the transformation is only one function, but realistically it could be a whole suite of functions wired together. Ideally, none of these functions should have to think about potential non-presence or contain any "nil-checks," as that's not their purpose; they should all be written to work on values that are fully present.

Given `userUpperName`, which works only on present values, we can use `fmap` to apply it to a value that may not be present to get back the result we expect with the same level of *present-ness*:

```
maybeName :: Maybe String
maybeName = fmap userUpperName (findUser someId)
```

We can do this repeatedly with every function in our system that's required to get from `findUser` to the eventual display of this name. Because of the second functor law, we know that if we compose all of these functions together then `fmap` the result, or if we `fmap` any individual functions and compose the results, we'll always get the same answer. We're free to design our system as we see fit, but still pass along the `Maybe`s everywhere we need to.

If we were doing this in the context of a web application, this maybe-name might end up being interpolated into some HTML. It's at this boundary that we'll have to "deal with" the `Maybe` value. One option is to use the `fromMaybe` function to specify a default value:

```
template :: Maybe String -> String
template mname = "<span class=\"username\">" ++ name ++ "</span>"

  where
    name = fromMaybe "(no name given)" mname
```

Curried Form

Before moving on, I need to pause briefly and answer a question I dodged in the Haskell Basics chapter. You may have wondered why Haskell type signatures don't separate a function's argument types from its return type. The direct answer is that all functions in Haskell are in *curried* form. This is an idea developed by and named for the same logician as Haskell itself.

A curried function is one that *conceptually* accepts multiple arguments by actually accepting only one, but returning a function. The returned function itself will also be curried and use the same process to accept more arguments. This

process continues for as many arguments as are needed. In short, all functions in Haskell are of the form `(a -> b)`. A (conceptually) multi-argument function like `add :: Int -> Int -> Int` is really `add :: Int -> (Int -> Int)`; this matches `(a -> b)` by taking `a` as `Int` and `b` as `(Int -> Int)`.

The reason I didn't talk about this earlier is that we can mostly ignore it when writing Haskell code. We define and apply functions as if they actually accept multiple arguments and things work as we intuitively expect. Even partial application (a topic I hand-waved a bit at the time) can be used effectively without realizing this is a direct result of curried functions. It's when we dive into concepts like `Applicative` (the focus of the next chapter) that we need to understand a bit more about what's going on under the hood.

The Case for Currying

In the implementation of purely functional programming languages, there is value in having all functions taking exactly one argument and returning exactly one result. Haskell is written this way, so users have two choices for defining "multi-argument" functions.

We could rely solely on tuples:

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

This results in the sort of type signatures you might expect, where the argument types are shown separate from the return types. The problem with this form is that partial application can be cumbersome. How do you add 5 to every element in a list?

```
f :: [Int]
f = map add5 [1,2,3]

  where
    add5 :: Int -> Int
    add5 y = add (5, y)
```

Alternatively, we could write all functions in curried form:

```
--
--     / One argument type, an Int
--     |
--     |       / One return type, a function from Int to Int
--     |       |
add :: Int -> (Int -> Int)
add x = \y -> x + y
--  |   |
--  |   ` One body expression, a lambda from Int to Int
--  |
--  ` One argument variable, an Int
--
```

This makes partial application simpler. Since `add 5` is a valid expression and is of the correct type to pass to `map`, we can use it directly:

```
f :: [Int]
f = map (add 5) [1,2,3]
```

While both forms are valid Haskell (in fact, the `curry` and `uncurry` functions in the Prelude convert functions between the two forms), the curried version was chosen as the default and so Haskell's syntax allows some things that make it more convenient.

For example, we can name function arguments in whatever way we like; we don't have to always assign a single lambda expression as the function body. In fact, these are all equivalent:

```
add = \x -> \y -> x + y
add x = \y -> x + y
add x y = x + y
```

In type signatures, `(->)` is right-associative. This means that instead of writing:

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

We can write the less-noisy:

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

And it has the same meaning.

Similarly, function application is left-associative. This means that instead of writing:

```
six :: Int
six = ((addThree 1) 2) 3
```

We can write the less-noisy:

```
six :: Int
six = addThree 1 2 3
```

And it has the same meaning as well.

These conveniences are why we don't actively picture functions as curried when writing Haskell code. We can define `addThree` naturally, as if it took three arguments, and let the rules of the language handle currying. We can also apply `addThree` naturally, as if it took three arguments and again the rules of the language will handle the currying.

Partial Application

Some languages don't use curried functions but do support *partial application*: supplying only some of a function's arguments to get back another function that accepts the arguments that were left out. We can do this in Haskell too, but it's not "partial" at all, since all functions truly accept only a single argument.

When we wrote the following expression:

```
maybeName = fmap userUpperName (findUser someId)
```

What really happened is that `fmap` was first applied to the function `userUpperName` to return a new function of type `Maybe User -> Maybe String`.

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

userUpperName :: (User -> String)

fmap userUpperName :: Maybe User -> Maybe String
```

This function is then immediately applied to `(findUser someId)` to ultimately get that `Maybe String`. This example shows that Haskell's curried functions blur the line between partial and total application. The result is a natural and consistent syntax for doing either.

Recap

So far, we've seen an introduction to Haskell functions and to Haskell's type system. We then introduced the `Maybe` type as a new and powerful way to use that type system to describe something about your domain–that some values may not be present–and a type class (`Functor`) that allows for strict separation between value-handling functions and the need to apply them to values that may not be present.

We then saw some real-world code that takes advantage of these ideas and discussed type class laws as a means of abstraction and encapsulation. These laws give us a precise understanding of how our code will behave without having to know its internals. Finally, we took a brief detour into the world of currying, a foundational concept responsible for many of the things we'll explore next.

In the next chapter, we'll talk about *applicative functors*. If we think of a *functor* as a value in some context, supporting an `fmap` operation for applying a function to that value while preserving its context, *applicative functors* are functors where the value itself *can be applied*. In simple terms: it's a function. These structures must then support another operation for applying that function from within its context. That operation, combined with currying, will grant us more power and convenience when working with `Maybe` values.

Applicative

In the last section we saw how to use `fmap` to take a system full of functions that operate on fully present values, free of any `nil`-checks, and employ them to safely manipulate values that may in fact not be present. This immediately makes many uses of `Maybe` more convenient, while still being explicit and safe in the face of failure and partial functions.

There's another notable case where `Maybe` can cause inconvenience, one that can't be solved by `fmap` alone. Imagine we're writing some code using a web framework. It provides a function `getParam` that takes the name of a query parameter (passed as part of the URL in a GET HTTP request) and returns the value for that parameter as parsed out of the current URL. Since the parameter you name could be missing or invalid, this function returns `Maybe`:

```
getParam :: String -> Params -> Maybe String
getParam = undefined
```

Let's say we also have a `User` data type in our system. `User`s are constructed from their name and email address, both `String`s.

```
data User = User String String
```

How do we build a `User` from query params representing their name and email?

The most direct way is the following:

```
userFromParams :: Params -> Maybe User
userFromParams params =
    case getParam "name" params of
        Just name -> case getParam "email" params of
            Just email -> Just (User name email)
            Nothing -> Nothing
        Nothing -> Nothing
```

`Maybe` is not making our lives easier here. Yes, type safety is a huge implicit win, but this still looks a lot like the tedious, defensive coding you'd find in any language:

```
def user_from_params(params)
  if name = get_param "name" params
    if email = get_param "email" params
      User.new(name, email)
    end
  end
end
```

Hiding Details

So how do we do this better? What we want is code that looks as if there is no `Maybe` involved (because that's convenient) but correctly accounts for `Maybe` at every step along the way (because that's safe). If no `Maybe`s were involved, and we were constructing a normal `User` value, the code might look like this:

```
userFromValues :: User
userFromValues = User aName anEmail
```

An ideal syntax would look very similar, perhaps something like this:

```
userFromMaybeValues :: Maybe User
userFromMaybeValues = User <$> aMaybeName <*> aMaybeEmail
```

The `Applicative` type class and its `Maybe` instance allow us to write exactly this code. Let's see how.

Follow The Types

We can start by trying to do what we want with the only tool we have so far: `fmap`.

What happens when we apply `fmap` to `User`? It's not immediately clear because `User` has the type `String -> String -> User` which doesn't line up with `(a -> b)`. Fortunately, it only *appears* not to line up. Remember, every function in Haskell takes one argument and returns one result: `User`'s actual type is `String -> (String -> User)`. In other words, it takes a `String` and returns a function, `(String -> User)`. In this light, it indeed lines up with the type `(a -> b)` by taking `a` as `String` and `b` as `(String -> User)`.

By substituting our types for `f`, `a`, and `b`, we can see what the type of `fmap User` is:

```
fmap :: (a -> b) -> f a -> f b

--      a      ->  b
User :: String -> (String -> User)

--           f     a      -> f      b
fmap User :: Maybe String -> Maybe (String -> User)
```

So now we have a function that takes a `Maybe String` and returns a `Maybe (String -> User)`. We also have a value of type `Maybe String` that we can give to this function, `getParam "name" params`:

```
getParam "name" params             :: Maybe String

fmap User                          :: Maybe String -> Maybe (String -> User)

fmap User (getParam "name" params) ::                 Maybe (String -> User)
```

The `Control.Applicative` module exports an operator synonym for `fmap` called `(<$>)` (I pronounce this as *fmap* because that's what it's a synonym for). The reason this synonym exists is to get us closer to our original goal of making expressions look as if there are no `Maybe`s involved. Since operators are placed between their arguments, we can use `(<$>)` to rewrite our expression above to an equivalent one with less noise:

```
User <$> getParam "name" params :: Maybe (String -> User)
```

This expression represents a "`Maybe` function". We're accustomed to *values* in a context: a `Maybe Int`, `Maybe String`, etc; and we saw how these were *functors*. In this case, we have a *function* in a context: a `Maybe (String -> User)`. Since functions are things that *can be applied*, these are called *applicative functors*.

By using `fmap`, we reduced our problem space and isolated the functionality we're lacking, functionality we'll ultimately get from `Applicative`:

We have this:

```
fmapUser :: Maybe (String -> User)
fmapUser = User <$> getParam "name" params
```

And we have this:

```
aMaybeEmail :: Maybe String
aMaybeEmail = getParam "email" params
```

And we're trying to ultimately get to this:

```
userFromParams :: Params -> Maybe User
userFromParams params = fmapUser <*> aMaybeEmail
```

We only have to figure out what that `(<*>)` should do. At this point, we have enough things defined that we know exactly what its type needs to be. In the next section, we'll see how its type pushes us to the correct implementation.

Apply

The `(<*>)` operator is pronounced *apply*. Specialized to `Maybe`, its job is to apply a `Maybe` function to a `Maybe` value to produce a `Maybe` result.

In our example, we have `fmapUser` of type `Maybe (String -> User)` and `aMaybeEmail` of type `Maybe String`. We're trying to use `(<*>)` to put those together and get a `Maybe User`. We can write that down as a type signature:

```
(<*>) :: Maybe (String -> User) -> Maybe String -> Maybe User
```

With such a specific type, this function won't be very useful, so let's generalize it away from `String`s and `User`s:

```
(<*>) :: Maybe (a -> b) -> Maybe a -> Maybe b
```

This function is part of the `Applicative` type class, meaning it will be defined for many types. Therefore, its actual type signature is:

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

Where `f` is any type that has an `Applicative` instance (such as `Maybe`).

It's important to mention this because it is the type signature you're going to see in any documentation about `Applicative`. Now that I've done so, I'm going to go back to type signatures using `Maybe` since that's the specific instance we're discussing here.

The semantics of our `(<*>)` function are as follows:

- If both the `Maybe` function and the `Maybe` value are present, apply the function to the value and return the result wrapped in `Just`
- Otherwise, return `Nothing`

We can translate that directly into code via pattern matching:

```
(<*>) :: Maybe (a -> b) -> Maybe a -> Maybe b
Just f <*> Just x = Just (f x)
_      <*> _      = Nothing
```

With this definition, and a few line breaks for readability, we arrive at our desired goal:

```
userFromParams :: Params -> Maybe User
userFromParams params = User
    <$> getParam "name" params
    <*> getParam "email" params
```

The result is an elegant expression with minimal noise. Compare *that* to the staircase we started with!

Not only is this expression elegant, it's also safe. Because of the semantics of `fmap` and `(<*>)`, if any of the `getParam` calls return `Nothing`, our whole `userFromParams` expression results in `Nothing`. Only if they all return `Just` values, do we get `Just` our user.

As always, Haskell's being referentially transparent means we can prove this by substituting the definitions of `fmap` and `(<*>)` and tracing how the expression expands given some example `Maybe` values.

If the first value is present, but the second is not:

```
User <$> Just "Pat" <*> Nothing
-- => fmap User (Just "Pat") <*> Nothing        (<$> == fmap)
-- => Just (User "Pat")      <*> Nothing        (fmap definition, first pattern)
-- => Nothing                                   (<*> definition, second pattern)
```

If the second value is present but the first is not:

```
User <$> Nothing <*> Just "pat@thoughtbot.com"
-- => fmap User Nothing <*> Just "pat@thoughtbot.com"
-- => Nothing           <*> Just "pat@thoughtbot.com"   (fmap, second pattern)
-- => Nothing
```

Finally, if both values are present:

```
User <$> Just "Pat" <*> Just "pat@thoughtbot.com"
-- => fmap User (Just "Pat") <*> Just "pat@thoughtbot.com"
-- => Just (User "Pat")      <*> Just "pat@thoughtbot.com"
-- => Just (User "Pat" "pat@thoughtbot.com")            (<*>, first pattern)
```

Chaining

One of the nice things about this pattern is that it scales up to functions that, conceptually at least, can accept any number of arguments. Imagine that our `User` type had a third field representing their age:

```
data User = User String String Int
```

Since our `getParam` function can only look up parameters of type `String`, we'll also need a `getIntParam` function to pull the user's age out of our `Params`:

```
getIntParam :: String -> Params -> Maybe Int
getIntParam = undefined
```

With these defined, let's trace through the types of our applicative expression again. This time, we have to remember that our new `User` function is of type `String -> (String -> (Int -> User))`:

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

```
User <$> getParam "name" params :: Maybe (String -> (Int -> User))
```

```
User <$> getParam "name" params <*> getParam "email" params :: Maybe (Int -> User)
```

This time, we arrive at a `Maybe (Int -> User)`. Knowing that `getIntParam "age" params` is of type `Maybe Int`, we're in the exact same position as last time when we first discovered a need for `(<*>)`. Being in the same position, we can do the same thing again:

```
userFromParams :: Params -> Maybe User
userFromParams params = User
    <$> getParam "name" params
    <*> getParam "email" params
    <*> getIntParam "age" params
```

As our pure function (`User`) gains more arguments, we can continue to apply it to values in context by repeatedly using `(<*>)`. The process by which this happens may be complicated, but the result is well worth it: an expression that is concise, readable, and above all safe.

Applicative In the Wild

This pattern is used in a number of places in the Haskell ecosystem.

JSON parsing

As one example, the aeson package defines a number of functions for parsing things out of JSON values. These functions return their results wrapped in a `Parser` type. This is very much like `Maybe` except that it holds a bit more information about *why* the computation failed, not only *that* the computation failed. Not unlike our `getParam`, these sub-parsers pull basic types (`Int`, `String`, etc.) out of JSON values. The `Applicative` instance for the `Parser` type can then be used to combine them into something domain-specific, like a `User`.

Again, imagine we had a rich `User` data type:

```
data User = User
    String        -- Name
    String        -- Email
    Int           -- Age
    UTCTime       -- Date of birth
```

We can tell aeson how to create a `User` from JSON, by implementing the `parseJSON` function. That takes a JSON object (represented by the `Value` type) and returns a `Parser User`:

```
parseJSON :: Value -> Parser User
parseJSON (Object o) = User
    <$> o .: "name"
    <*> o .: "email"
    <*> o .: "age"
    <*> o .: "birth_date"

-- If we're given some JSON value besides an Object (an Array, a Number, etc) we
-- can signal failure by returning the special value mzero
parseJSON _ = mzero
```

Each individual `o .: "..."` expression is a function that attempts to pull the value for the given key out of a JSON `Object`. Potential failure (missing key, invalid type, etc) is captured by returning a value wrapped in the `Parser` type. We can combine the individual `Parser` values together into one `Parser User` using `(<$>)` and `(<*>)`.

If any key is missing, the whole thing fails. If they're all there, we get the `User` we wanted. This concern is completely isolated within the implementation of `(<$>)` and `(<*>)` for `Parser`.

Option parsing

Another example is command-line options parsing via the optparse-applicative library. The process is very similar: the library exposes low-level parsers for primitive types like `Flag` or `Argument`. Because this may fail, the values are wrapped in another `Parser` type. (Though it behaves similarly, this is this library's own `Parser` type, not the same one as above.) The `Applicative` instance can again be used to combine these sub-parsers into a domain-specific `Options` value:

```
-- Example program options:
--
-- - A bool to indicate if we should be verbose, and
-- - A list of FilePaths to operate on
--
data Options = Options Bool [FilePath]

parseOptions :: Parser Options
parseOptions = Options
    <$> switch (short 'v' <> long "verbose" <> help "be verbose")
    <*> many (argument (metavar "FILE" <> help "file to operate on"))
```

You can ignore some of the functions here, which were included to keep the example accurate. What's important is that `switch (...)` is of type `Parser Bool` and `many (argument ...)` is of type `Parser [FilePath]`. We use `(<$>)` and `(<*>)` to put these two sub-parsers together with `Options` and end up with an overall `Parser Options`. If we add more options to our program, all we need to do is add more fields to `Options` and continue applying sub-parsers with `(<*>)`.

Monad

So far, we've seen that as `Maybe` makes our code safer, it also makes it less convenient. By making potential non-presence explicit, we now need to correctly account for it at every step. We addressed a number of scenarios by using `fmap` to "upgrade" a system full of normal functions (free of any `nil`-checks) into one that can take and pass along `Maybe` values. When confronted with a new scenario that could not be handled by `fmap` alone, we discovered a new function `(<*>)` which helped ease our pain again. This chapter is about addressing a third scenario, one that `fmap` and even `(<*>)` cannot solve: dependent computations.

Let's throw a monkey wrench into our `getParam` example from earlier. This time, let's say we're accepting logins by either username or email. The user can say which method they're using by passing a `type` param specifying "username" or "email".

Note: this whole thing is wildly insecure, but bear with me.

Again, all of this is fraught with `Maybe`-ness and again, writing it with straight-line `case` matches can get very tedious:

```
loginUser :: Params -> Maybe User
loginUser params = case getParam "type" of
    Just t -> case t of
      "username" -> case getParam "username" of
          Just u -> findUserByUserName u
          Nothing -> Nothing
      "email" -> case getParam "email" of
          Just e -> findUserByEmail e
```

```
        Nothing -> Nothing
    _ -> Nothing
  Nothing -> Nothing
```

Yikes.

More Power

We can't clean this up with `(<*>)` because each individual part of an `Applicative` expression doesn't have access to the results from any other part's evaluation. What does that mean? If we look at the `Applicative` expression from before:

```
User <$> getParam "name" params <*> getParam "email" params
```

Here, the two results from `getParam "name"` and `getParam "email"` (either of which could be present or not) are passed together to `User`. If they're both present we get a `Just User`, otherwise `Nothing`. Within the `getParam "email"` expression, you can't reference the (potential) result of `getParam "name"`.

We need that ability to solve our current conundrum because we need to check the value of the "type" param to know what to do next. We need... *monads*.

And Then?

Let's start with a minor refactor. We'll pull out a `loginByType` function:

```
loginUser :: Params -> Maybe User
loginUser params = case getParam "type" params of
    Just t -> loginByType params t
    Nothing -> Nothing

loginByType :: Params -> String -> Maybe User
loginByType params "username" = case getParam "username" params of
    Just u -> findUserByUserName u
```

```
    Nothing -> Nothing

loginByType params "email" = case getParam "email" params of
    Just e -> findUserByEmail e
    Nothing -> Nothing

loginByType _ _ = Nothing
```

Things seem to be following a pattern now: we have some value that might not be present and some function that needs the (fully present) value, does some other computation with it, but may itself fail.

Let's abstract this concern into a new function called `andThen`:

```
andThen :: Maybe a -> (a -> Maybe b) -> Maybe b
andThen (Just x) f = f x
andThen _ _ = Nothing
```

We'll use the function infix via backticks for readability:

```
loginUser :: Params -> Maybe User
loginUser params =
    getParam "type" params `andThen` loginByType params

loginByType :: Params -> String -> Maybe User
loginByType params "username" =
    getParam "username" params `andThen` findUserByUserName

loginByType params "email" =
    getParam "email" params `andThen` findUserByEmail

-- Still needed in case we get an invalid type
loginByType _ _ = Nothing
```

This cleans things up nicely. The concern of "passing along the `Maybe`" is completely abstracted away behind `andThen` and we're free to describe the nature of *our* computation. If only Haskell had such a function...

Bind

If you haven't guessed it, Haskell does have exactly this function. Its name is *bind* and it's defined as part of the `Monad` type class. Here is its type signature:

```
(>>=) :: m a -> (a -> m b) -> m b
--
-- where m is the type you're saying is a Monad (e.g. Maybe)
--
```

Again, you can see that `andThen` has the correct signature:

```
--         m     a    (a -> m     b) -> m     b
andThen :: Maybe a -> (a -> Maybe b) -> Maybe b
```

Chaining

`(>>=)` is defined as an operator because it's meant to be used infix. It's also given an appropriate fixity so it can be chained together intuitively. This is why I chose the name `andThen` for my fictitious version: it can sometimes help to read `x >>= y >>= z` as *x and-then y and-then z*. To see this in action, let's walk through another example.

Suppose we are working on a system with the following functions for dealing with users' addresses and their zip codes:

```
-- Returns Maybe because the user may not exist
findUser :: UserId -> Maybe User
findUser = undefined

-- Returns Maybe because Users aren't required to have an address on file
userZip :: User -> Maybe ZipCode
userZip = undefined
```

Let's also say we have a function to calculate shipping costs by zip code. It employs `Maybe` to handle invalid zip codes:

```
shippingCost :: ZipCode -> Maybe Cost
shippingCost = undefined
```

We could naively calculate the shipping cost for some user given their Id:

```
findUserShippingCost :: UserId -> Maybe Cost
findUserShippingCost uid =
    case findUser uid of
        Just u -> case userZip u of
            Just z -> case shippingCost z of
                Just c  -> Just c

                -- User has an invalid zip code
                Nothing -> Nothing

            -- Use has no address
            Nothing -> Nothing

        -- User not found
        Nothing -> Nothing
```

This code is offensively ugly, but it's the sort of code I write every day in Ruby. We might hide it behind three-line methods each holding one level of conditional, but it's there.

How does this code look with `(>>=)`?

```
findUserShippingCost :: UserId -> Maybe Cost
findUserShippingCost uid = findUser uid >>= userZip >>= shippingCost
```

You have to admit, that's quite nice. Hopefully even more so when you look back at the definition for `andThen` to see that that's all it took to clean up this boilerplate.

Do Notation

There's one more topic I'd like to mention related to monads: *do-notation*.

This bit of syntactic sugar is provided by Haskell for any of its `Monad`s. The reason is to allow functional Haskell code to read like imperative code when building compound expressions using `Monad`. This is valuable because monadic expressions, especially those representing interactions with the outside world, are often read best as a series of imperative steps:

```
f = do
    x <- something
    y <- anotherThing
    z <- combineThings x y

    finalizeThing z
```

That said, this sugar is available for any `Monad` and so we can use it for `Maybe` as well. We can use `Maybe` as an example for seeing how *do-notation* works. Then, if and when you come across some `IO` expressions using *do-notation*, you won't be as surprised or confused.

De-sugaring *do-notation* is a straightforward process followed out during Haskell compilation. It can be understood best by doing it manually. Let's start with our end result from the last example. We'll translate this code step by step into the equivalent *do-notation* form, then follow the same process backward, as the compiler would do if we had written it that way in the first place.

```
findUserShippingCost :: UserId -> Maybe Cost
findUserShippingCost uid = findUser uid >>= userZip >>= shippingCost
```

First, let's add some arbitrary line breaks so the eventual formatting aligns with what someone might write by hand:

```
findUserShippingCost :: UserId -> Maybe Cost
findUserShippingCost uid =
    findUser uid >>=
    userZip >>=

    shippingCost
```

Next, let's name the arguments to each expression via anonymous functions, rather than relying on partial application and their curried nature:

```
findUserShippingCost :: UserId -> Maybe Cost
findUserShippingCost uid =
    findUser uid >>= \u ->
    userZip u >>= \z ->

    shippingCost z
```

Next, we'll take each lambda and translate it into a *binding*, which looks a bit like variable assignment and uses `(<-)`. You can read `x <- y` as "x from y":

```
findUserShippingCost :: UserId -> Maybe Cost
findUserShippingCost uid =
    u <- findUser uid
    z <- userZip u

    shippingCost z
```

Finally, we prefix the series of "statements" with `do`:

```
findUserShippingCost :: UserId -> Maybe Cost
findUserShippingCost uid = do
    u <- findUser uid
    z <- userZip u

    shippingCost z
```

Et voilà, you have the equivalent *do-notation* version of our function. When the compiler sees code written like this, it follows (mostly) the same process we did, but in reverse:

Remove the `do` keyword:

```
findUserShippingCost :: UserId -> Maybe Cost
findUserShippingCost uid =
```

```
    u <- findUser uid
    z <- userZip u

    shippingCost z
```

Translate each binding into a version using `(>>=)` and lambdas:

```
findUserShippingCost :: UserId -> Maybe Cost
findUserShippingCost uid =
    findUser uid >>= \u ->
    userZip u >>= \z ->

    shippingCost z
```

The compiler can stop here as all remaining steps are stylistic changes only. To get back to our exact original expression, we only need to *eta-reduce*[1] the lambdas:

```
findUserShippingCost :: UserId -> Maybe Cost
findUserShippingCost uid =
    findUser uid >>=
    userZip >>=

    shippingCost
```

And remove our arbitrary line breaks:

```
findUserShippingCost :: UserId -> Maybe Cost
findUserShippingCost uid = findUser uid >>= userZip >>= shippingCost
```

Wrapping Up

And thus ends our discussion of monads. This also ends our discussion of `Maybe`. You've now seen the type itself and three of Haskell's most important abstractions,

[1] The process of simplifying `\x -> f x` to the equivalent form `f`.

which make its use convenient while still remaining explicit and safe. To highlight the point that these abstractions (`Functor`, `Applicative`, and `Monad`) are *interfaces* shared by many types, the next and final section will briefly show a few other useful types that also have these three interfaces.

Other Types

The three abstractions you've seen all require a certain kind of value. Specifically, a value with some other bit of information, often referred to as its *context*. In the case of a type like `Maybe a`, the `a` represents the value itself and `Maybe` represents the fact that it may or may not be present. This potential non-presence is that other bit of information, its context.

Haskell's type system is unique in that it lets us speak specifically about this other bit of information without involving the value itself. In fact, when defining instances for `Functor`, `Applicative` and `Monad`, we were defining an instance for `Maybe`, not for `Maybe a`. When we define these instances we're not defining how `Maybe a`, a value in some context, behaves under certain computations, we're actually defining how `Maybe`, the context itself, behaves under certain computations.

This kind of separation of concerns is difficult to understand when you're only accustomed to languages that don't allow for it. I believe it's why topics like monads seem so opaque to those unfamiliar with a type system like this. To strengthen the point that what we're really talking about are behaviors and contexts, not any one specific *thing*, this chapter will explore types that represent other kinds of contexts and show how they behave under all the same computations we saw for `Maybe`.

Either

Haskell has another type to help with computations that may fail:

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

Traditionally, the `Right` constructor is used for a successful result (what a function would have returned normally) and `Left` is used in the failure case. The value of type `a` given to the `Left` constructor is meant to hold information about the failure: why did it fail? This is only a convention, but it's a strong one that we'll use throughout this chapter. To see one formalization of this convention, take a look at Control.Monad.Except. It can appear intimidating because it is so generalized, but Example 1 should look a lot like what I'm about to walk through here.

With `Maybe a`, the `a` was the value and `Maybe` was the context. Therefore, we made instances of `Functor`, `Applicative`, and `Monad` for `Maybe` (not `Maybe a`). With `Either` as we've written it above, `b` is the value and `Either a` is the context, therefore Haskell has instances of `Functor`, `Applicative`, and `Monad` for `Either a` (not `Either a b`).

This use of `Left a` to represent failure with error information of type `a` can get confusing when we start looking at functions like `fmap`. Here's why: the generalized type of `fmap` talks about `f a` and I said our instance would be for `Either a` making that `Either a a`, but they aren't the same `a`!

For this reason, we can imagine an alternate definition of `Either` that uses different variables. This is perfectly reasonable since the variables are chosen arbitrarily anyway:

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

When we get to `fmap` (and others), things are clearer:

```
--      (a -> b)    f        a -> f        b
fmap :: (a -> b) -> Either e a -> Either e b
```

ParserError

As an example, consider some kind of parser. If parsing fails, it would be nice to include some information about what triggered the failure. To accomplish this, we first define a type to represent this information. For our purposes, it's the line and column where something unexpected appeared, but it could be much richer than that including what was expected and what was seen instead:

```
data ParserError = ParserError Int Int
```

From this, we can make a domain-specific type alias built on top of `Either`. We can say a value that we parse may fail. If it does, error information will appear in a `Left`-constructed result. If it succeeds, we'll get the `a` we originally wanted in a `Right`-constructed result.

```
--              Either e           a    = Left e           | Right a
type Parsed a = Either ParserError a -- = Left ParserError | Right a
```

Finally, we can give an informative type to functions that may produce such results:

```
parseJSON :: String -> Parsed JSON
parseJSON = undefined
```

This informs callers of `parseJSON` that it may fail and, if it does, the invalid character and line can be found:

```
jsonString = "..."

case parseJSON jsonString of
   Right json ->                 -- do something with json
   Left (ParserError ln col) ->  -- do something with the error information
```

Functor

You may have noticed that we've reached the same conundrum as with `Maybe`: often, the best thing to do if we encounter a `Left` result is to pass it along to our own callers. Wouldn't it be nice if we could take some JSON-manipulating function and apply it directly to something that we parse? Wouldn't it be nice if the "pass along the errors" concern were handled separately?

```
-- Replace the value at the given key with the new value
replace :: Key -> Value -> JSON -> JSON
replace = undefined

--
```

```
-- This is a type error!
--
--  replace "admin" False is (JSON -> JSON), but parseJSON returns (Parsed JSON)
--
replace "admin" False (parseJSON jsonString)
```

`Parsed a` is a value in some context, like `Maybe a`. This time, rather than only present-or-non-present, the context is richer. It represents present-or-non-present-with-error. Can you think of how this context should be accounted for under an operation like `fmap`?

```
--      (a -> b) -> f        a -> f        b
--      (a -> b) -> Either e a -> Either e b
fmap :: (a -> b) -> Parsed   a -> Parsed   b
fmap f (Right v) = Right (f v)
fmap _ (Left e)  = Left e
```

If the value is there, we apply the given function to it. If it's not, we pass along the error. Now we can do something like this:

```
fmap (replace "admin" False) (parseJSON jsonString)
```

If the incoming string is valid, we get a successful `Parsed JSON` result with the `"admin"` key replaced by `False`. Otherwise, we get an unsuccessful `Parsed JSON` result with the original error message still available.

Knowing that `Control.Applicative` provides `(<$>)` as an infix synonym for `fmap`, we could also use that to make this read a bit better:

```
replace "admin" False <$> parseJSON jsonString
```

Speaking of `Applicative`...

Applicative

It would also be nice if we could take two potentially failed results and pass them as arguments to some function that takes normal values. If any result fails, the overall result is also a failure. If all are successful, we get a successful overall result. This sounds a lot like what we did with `Maybe`. The only difference is that we're doing it for a different kind of context.

```
-- Given two json objects, merge them into one
merge :: JSON -> JSON -> JSON
merge = undefined

jsonString1 = "..."

jsonString2 = "..."

merge <$> parseJSON jsonString1 <*> parseJSON jsonString2
```

`merge <$> parseJSON jsonString1` gives us a `Parsed (JSON -> JSON)`. (If this doesn't make sense, glance back at the examples in the Applicative chapter.) What we have is a *function* in a `Parsed` context. `parseJSON jsonString2` gives us a `Parsed JSON`, a *value* in a `Parsed` context. The job of `(<*>)` is to apply the `Parsed` function to the `Parsed` value and produce a `Parsed` result.

Defining `(<*>)` starts out all right: if both values are present we'll get the result of applying the function wrapped up again in `Right`. If the second value's not there, that error is preserved as a new `Left` value:

```
--       f        (a -> b) -> f        a -> f        b
--       Either e (a -> b) -> Either e a -> Either e b
(<*>) :: Parsed   (a -> b) -> Parsed   a -> Parsed   b
Right f <*> Right x = Right (f x)
Right _ <*> Left e = Left e
```

Astute readers may notice that we could reduce this to one pattern by using `fmap`. This is left as an exercise.

What about the case where the first argument is `Left`? At first this seems trivial: there's no use inspecting the second value because we know something has already failed, so let's pass that along, right? Well, what if the second value was also an error? Which error should we keep? Either way we discard one of them. Any potential loss of information should be met with pause.

It turns out, it doesn't matter, at least not as far as the Applicative Laws are concerned. If choosing one over the other had violated any of the laws, we would have had our answer. Beyond those, we don't know how this instance will eventually be used by end-users and we can't say which is the "right" choice standing here now.

Given that the choice is arbitrary, I present the actual definition from `Control.Applicative`:

```
Left e <*> _ = Left e
```

Monad

When thinking through the `Monad` instance for our `Parsed` type, we don't have the same issue of deciding which error to propagate. Remember that the extra power offered by monads is that computations can depend on the results of prior computations. When the context involved represents failure (which may not always be the case!), any single failing computation must trigger the omission of all subsequent computations (since they could be depending on some result that's not there). This means we only need to propagate that first failure.

Let's say we're interacting with a JSON web service for getting blog post content. The responses include the body of the post as a string of HTML:

```
{
  "title": "A sweet blog post",
  "body": "<p>The post content...</p>"
}
```

Parsing JSON like this includes parsing the value at the `"body"` key into a structured `HTML` data type. For this, we can re-use our `Parsed` type:

```
parseHTML :: Value -> Parsed HTML
parseHTML = undefined
```

We can directly parse a `String` of JSON into the `HTML` present at one of its keys by binding the two parses together with `(>>=)`:

```
-- Grab the value at the given key
at :: Key -> JSON -> Value
at = undefined

parseBody :: String -> Parsed HTML
parseBody jsonString = parseJSON jsonString >>= parseHTML . at "body"
```

First, `parseJSON jsonString` gives us a `Parsed JSON`. This is the `m a` in `(>>=)`'s type signature. Then we use `(.)` to compose a function that gets the value at the `"body"` key and passes it to `parseHTML`. The type of this function is `(JSON -> Parsed HTML)`, which aligns with the `(a -> m b)` of `(>>=)`'s second argument. Knowing that `(>>=)` will return `m b`, we can see that that's the `Parsed HTML` we're after.

If both parses succeed, we get a `Right`-constructed value containing the `HTML` we want. If either parse fails, we get a `Left`-constructed value containing the `ParserError` from whichever failed.

Allowing such a readable expression (*parse JSON and then parse HTML at body*), requires the following straightforward implementation for `(>>=)`:

```
--       m        a -> (a -> m        b) -> m        b
--       Either e a -> (a -> Either e b) -> Either e b
(>>=) :: Parsed   a -> (a -> Parsed   b) -> Parsed   b
Right v >>= f = f v
Left e >>= _ = Left e
```

Armed with instances for `Functor`, `Applicative`, and `Monad` for both `Maybe` and `Either e`, we can use the same set of functions (those with `Functor f`, `Applicative f` or `Monad m` in their class constraints) and apply them to a variety of functions that may fail (with or without useful error information).

This is a great way to reduce a project's maintenance burden. If you start with functions returning `Maybe` values but use generalized functions for (e.g.) any `Monad m`, you can later upgrade to a fully fledged `Error` type based on `Either` without having to change most of the code base.

List

At various points in the book, I relied on most programmers having an understanding of arrays and lists of elements to ease the learning curve up to `Maybe` and particularly `fmap`. In this chapter, I'll recap and expand on some of the things I've said before and then show that `[a]` is more than a list of elements over which we can map. It also has `Applicative` and `Monad` instances that make it a natural fit for certain problems.

Tic-Tac-Toe and the Minimax algorithm

For this chapter's example, I'm going to show portions of a program for playing Tic-Tac-Toe. The full program is too large to include, but portions of it are well-suited to using the `Applicative` and `Monad` instances for `[]`. The program uses an algorithm known as minimax to choose the best move to make in a game of Tic-Tac-Toe.

In short, the algorithm plays out all possible moves from the perspective of one player and chooses the one that maximizes their score and minimizes their opponent's, hence the name. Tic-Tac-Toe is a good game for exploring this algorithm because the possible choices are small enough that we can take the naive approach of enumerating all of them, then choosing the best.

To model our Tic-Tac-Toe game, we'll need some data types:

```
-- A player is either Xs or Os
data Player = X | O

-- A square is either open, or taken by one of the players
data Square = Open | Taken Player

-- A row is top, middle, or bottom
data Row = T | M | B

-- A column is left, center, or right
data Column = L | C | R
```

```
-- A position is the combination of row and column
type Position = (Row, Column)

-- A space is the combination of position and square
type Space = (Position, Square)

-- Finally, the board is a list of spaces
type Board = [Space]
```

And some utility functions:

```
-- Is the game over?
over :: Board -> Bool
over = undefined

-- The opponent for the given player
opponent :: Player -> Player
opponent = undefined

-- Play a space for the player in the given board
play :: Player -> Position -> Board -> Board
play = undefined
```

Applicative

One of the things this program needs to do is generate a `Board` with all `Square`s `Open`. We could do this directly:

```
openBoard :: Board
openBoard =
    [ ((T, L), Open), ((T, C), Open), ((T, R), Open)
    , ((M, L), Open), ((M, C), Open), ((M, R), Open)
    , ((B, L), Open), ((B, C), Open), ((B, R), Open)
    ]
```

But that approach is tedious and error-prone. Another way to solve this problem is to create an `Open` square for all combinations of `Row`s and `Column`s. We can do exactly this with the `Applicative` instance for `[]`:

```
openSpace :: Row -> Column -> Space
openSpace r c = ((r, c), Open)

openBoard :: Board
openBoard = openSpace <$> [T, M, B] <*> [L, C, R]
```

Let's walk through the body of `openBoard` to see why it gives the result we need. First, `openSpace <$> [T, M, B]` maps the two-argument `openSpace` over the list `[T, M, B]`. This creates a list of partially applied functions. Each of these functions has been given a `Row` but still needs a `Column` to produce a full `Space`. We can show this as a list of lambdas taking a `Column` and building a `Space` with the `Row` it has already:

```
(<$>) :: (a -> b) -> f a -> f b

--           a       b
openSpace :: Row -> (Column -> Space)

--                        f   b
openSpace <$> [T, M, B] :: [] (Column -> Space)

openSpace <$> [T, M, B]
-- => [ (\c -> ((T, c), Open))
-- => , (\c -> ((M, c), Open))
-- => , (\c -> ((B, c), Open))
-- => ]
```

Like the `Maybe` example from the `Applicative` chapter, we've created a function in a context. Here we have the function `(Column -> Space)` in the `[]` context: `[(Column -> Space)]`. Separating the type constructor from its argument and writing `[(Column -> Space)]` as `[] (Column -> Space)` shows how it matches the `f b` in `(<$>)`s type signature. How do we apply a function in a context to a value in a context? With `(<*>)`.

Using `(<*>)` with lists means applying every function to every value:

```
openSpace <$> [T, M, B] <*> [L, C, R]
-- => [ (\c -> ((T, c), Open)) L        (first function applied to each value)
```

```
-- => , (\c -> ((T, c), Open)) C
-- => , (\c -> ((T, c), Open)) R
-- => , (\c -> ((M, c), Open)) L        (second function applied to each value)
-- => , (\c -> ((M, c), Open)) C
-- => , (\c -> ((M, c), Open)) R
-- => , (\c -> ((B, c), Open)) L        (third function applied to each value)
-- => , (\c -> ((B, c), Open)) C
-- => , (\c -> ((B, c), Open)) R
-- => ]
--
-- => [ ((T, L), Open)
-- => , ((T, C), Open)
-- => , ((T, R), Open)
-- => , ((M, L), Open)
-- => , ((M, C), Open)
-- => , ((M, R), Open)
-- => , ((B, L), Open)
-- => , ((B, C), Open)
-- => , ((B, R), Open)
-- => ]
```

Monad and non-determinism

The heart of the minimax algorithm is playing out a hypothetical future where each available move is made to see which one works out best. The `Monad` instance for `[]` is perfect for this problem when we think of lists as representing one non-deterministic value rather than a list of many deterministic ones.

The list `[1, 2, 3]` represents a single number that is any one of `1`, `2`, or `3` at once. The type of this value is `[Int]`. The `Int` tells us the type of the value we're dealing with and the `[]` tells us that it's many values at once.

Under this interpretation, `Functor`'s `fmap` represents *changing* probabilities: we have a number that can be any of `1`, `2`, or `3`. When we `fmap (+1)`, we get back a number that can be any of `2`, `3`, or `4`. We've changed the non-determinism without changing *how much* non-determinism there is. That fact, that `fmap` can't increase or decrease the non-determinism, is actually guaranteed through the Functor laws.

```
fmap (+1) [1, 2, 3]
-- => [2, 3, 4]
```

`Applicative`'s `(<*>)` can be thought of as *combining* probabilities. Given a function that can be any of `(+1)`, `(+2)`, or `(+3)` and a number that can be any of `1`, `2`, or `3`, `(<*>)` will give us a new number that can be any of the combined results of applying each possible function to each possible value.

```
[(+1), (+2), (+3)] <*> [1, 2, 3]
-- => [2,3,4,3,4,5,4,5,6]
```

Finally, `Monad`'s `(>>=)` is used to *expand* probabilities. Looking at its type again:

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

And specializing this to lists:

```
--       m  a -> (a -> m  b) -> m  b
(>>=) :: [] a -> (a -> [] b) -> [] b
```

We can see that it takes an `a` that can be one of many values, and a function from `a` to `[b]`, i.e. a `b` that can be one of many values. `(>>=)` applies the function `(a -> [b])` to every `a` in the input list. The result must be `[[b]]`. To return the required type `[b]`, the list is then flattened. Because the types are so generic, this is the only implementation this function can have. If we rule out obvious mistakes like ignoring arguments and returning an empty list, reordering the list, or adding or dropping elements, the only way to define this function is to map, then flatten.

```
xs >>= f = concat (map f xs)
```

Given our same number, one that can be any of `1`, `2`, or `3`, and a function that takes a (deterministic) number and produces a new set of possibilities, `(>>=)` will expand the probability space:

```
next :: Int -> [Int]
next n = [n - 1, n, n + 1]

[1, 2, 3] >>= next
-- => concat (map next [1, 2, 3])
-- => concat [[1 - 1, 1, 1 + 1], [2 - 1, 2, 2 + 1], [3 - 1, 3, 3 + 1]]
-- => [0,1,2,1,2,3,2,3,4]
```

We can continue expanding by repeatedly using `(>>=)`:

```
[1, 2, 3] >>= next >>= next
-- => [-1,0,1,0,1,2,1,2,3,0,1,2,1,2,3,2,3,4,1,2,3,2,3,4,3,4,5]
```

If we picture the `next` function as a step in time, going from some current state to multiple possible next states, we can think of `>>= next >>= next` as looking two steps into the future, exploring the possible states reachable from possible states.

The Future

If the theory above didn't make complete sense, that's OK. Let's get back to our Tic-Tac-Toe program and see how this works in the context of a real-word example.

When it's our turn (us being the computer player), we want to play out the next turn for every move we have available. For each of those next turns, we want to do the same thing again. We want to repeat this process until the game is over. At that point, we can see which choice led to the best result and use that one.

One thing we'll need, and our first opportunity to use `Monad`, is to find all available moves for a given `Board`:

```
available :: Board -> [Position]
available board = do
    (position, Open) <- board

    return position
```

In this expression, we're treating a `Board` as a list of `Space`s. In other words, it's one `Space` that is all of the spaces on the board at once. We're using `(>>=)`, through *do-notation*, to map, then flatten, each `Space` to its `Position`. We're using *do-notation* to take advantage of the fact that if we use a pattern in the left-hand side of `(<-)`, but the value doesn't match the pattern, it's discarded. This expression is a concise map-filter that relies on `(>>=)` to do the mapping and pattern matching to do the filtering.

Return

The `return` function, seen at the end of `available`, is not like `return` statements you'll find in other languages. Specifically, it does not abort the computation, presenting its argument as the return value for the function call. Nor is it always required at the end of a monadic expression. `return` is another function from the `Monad` type class. Its job is to take some value of type `a` and make it an `m a`. Conceptually, it should do this by putting the value in some default or minimal context. For `Maybe` this means applying `Just`. For `[]`, we put the value in a singleton list:

```
--        a -> m  a
return :: a -> [] a
return x = [x]
```

In our example, `position` is of type `Position` (i.e. `a`) and we need the expression to have type `[Position]` (i.e. `m a`), so `return position` does that.

Another `Monad`-using function of the minimax algorithm is one that expands a given board into the end-states reached when each player plays all potential moves:

```
future :: Player -> Board -> [Board]
future player board = do
    if over board
        then return board
        else do
            space <- available board

            future (opponent player) (play player space board)
```

First we check if the `Board` is `over`. If that's the case, the future is a singleton list of only that `Board`–again, `return board` does that. Otherwise, we explore all available spaces. For each of them, we explore into the future again, this time for our opponent on a `Board` where we've played that space. This process repeats until someone wins or we fill the board in a draw. To fit this function into our Tic-Tac-Toe-playing program, we would score each path as we explore it and play the path with the best score.

While a full program like this is very interesting, it quickly gets complicated with things not important to our discussion. To see a complete definition of a minimax-using Tic-Tac-Toe-playing program written in Ruby, check out Never Stop Building's Understanding Minimax.

IO

So far, we've seen three types: `Maybe a`, `Either e a`, and `[a]`. These types all represent a value with some other bit of information: a *context*. If `a` is the `User` you're trying to find, the `Maybe` says if she was actually found. If the `a` is the `JSON` you're attempting to parse, the `Either e` holds information about the error when the parse fails. If `a` is a number, then `[]` tells you it is actually many numbers at once, and how many.

For all these types, we've seen the behaviors that allow us to add them to the `Functor`, `Applicative`, and `Monad` type classes. These behaviors obey certain laws which allow us to reason about what will happen when we use functions like `fmap` or `(>>=)`. In addition to this, we can also reach in and manually resolve the context. We can define a `fromMaybe` function to reduce a `Maybe a` to an `a` by providing a default value for the `Nothing` case. We can do a similar thing for `Either e a` with the `either` function. Given a `[a]` we can resolve it to an `a` by selecting one at a given index (taking care to handle the empty list).

The `IO` type, so important to Haskell, is exactly like the three types you've seen so far in that it represents a value in some context. With a value of type `IO a`, the `a` is the thing you want and the `IO` means some input or output will be performed in the real word as part of producing that `a`. The difference is that the only way we can combine `IO` values is through their `Functor`, `Applicative`, and `Monad` interfaces. In fact, it's really only through its `Monad` interface since the `Applicative` and `Functor` instances are defined in terms of it. We can't ourselves resolve an `IO a` to an `a`.

This has many ramifications in how programs must be constructed.

Effects in a pure world

One question I get asked a lot is, "how is it that Haskell, a *pure* functional programming language, can actually do anything? How does it create or read files? How does it print to the terminal? How does it serve web requests?"

The short answer is, it doesn't. To show this, let's start with the following Ruby program:

```ruby
def main
  print "give me a word: "

  x = gets

  puts x
end
```

When you run this program with the Ruby interpreter, does anything happen? It depends on your definition of *happen*. Certainly, no I/O will happen, but that's not *nothing*. Objects will be instantiated, and a method has been defined. By defining this method, you've constructed a blue-print for some actions to be performed, but then neglected to perform them.

Ruby expects (and allows) you to invoke effecting methods like `main` whenever and wherever you want. If you want the above program to do something, you need to call `main` at the bottom. This is a blessing and a curse. While the flexibility is appreciated, it's a constant source of bugs and makes methods and objects impossible to reason about without looking at their implementations. A method may look "pure", but internally it might access a database, pull from an external source of randomness, or fire nuclear missiles. Haskell doesn't work like that.

Here's a translation of the Ruby program into Haskell:

```haskell
main :: IO ()
main = do
  putStr "give me a word: "
```

```
  x <- getLine

  putStrLn x
```

Much like the Ruby example, this code doesn't *do* anything. It defines a function `main` that states what should happen when it's executed. It does not execute anything itself. Unlike Ruby, Haskell does not expect or allow you to call `main` yourself. The Haskell runtime will handle that and perform whatever I/O is required for you. This is how I/O happens in a pure language: you define the blue-print, a *pure* value that says *how* to perform any I/O, then you give that to a separate runtime, which is in charge of actually performing it.

Statements and the curse of do-notation

The Haskell function above used *do-notation*. I did this to highlight that the reason do-notation exists is for Haskell code to look like that equivalent, imperative Ruby, on which it was based. This fact has the unfortunate consequence of tricking new Haskell programmers into thinking that `putStr` (for example) is an imperative statement that actually puts the string to the screen when evaluated.

In the Ruby code, each statement is implicitly combined with the next as the interpreter sees them. There is some initial global state, statements modify that global state, and the interpreter handles ensuring that subsequent statements see an updated global state from all those that came before. If Ruby used a semicolon instead of white space to delimit statements, we could almost think of `(;)` as an operator for combining statements and keeping track of the global state between them.

In Haskell, there are no statements, only expressions. Every expression has a type and compound expressions must be combined in a type-safe way. In the case of `IO` expressions, they are combined with `(>>=)`. The semantic result is very similar to Ruby's statements. It's because of this that you may hear `(>>=)` referred to as a *programmable semicolon*. In truth, it's so much more than that. It's a first-class function that can be passed around, built on top of, and overloaded from type to type.

To see how this works, let's build an equivalent definition for `main`, only this time no do-notation, only `(>>=)`.

Typed puzzles

Starting with the type of `main`, we immediately see something worth explaining:

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

The type of `main` is pronounced *IO void*. `()` itself is a type defined with a single constructor. It can also be thought of as an empty tuple:

```
data () = ()
```

It's used to stand in when a computation affects the *context*, but produces no useful *result*. It's not specific to `IO` (or monads for that matter). For example, if you were chaining a series of `Maybe` values together using `(>>=)` and under some condition you wanted to manually trigger an overall `Nothing` result, you could insert a `Nothing` of type `Maybe ()` into the expression.

This is exactly how the guard function works. When specialized to `Maybe`, its definition is:

```
guard :: Bool -> Maybe ()
guard True = Just ()
guard False = Nothing
```

It is used like this:

```
findAdmin :: UserId -> Maybe User
findAdmin uid = do
    user <- findUser uid

    guard (isAdmin user)

    return user
```

If you're having trouble seeing why this expression works, start by de-sugaring from *do-notation* to the equivalent expression using `(>>=)`, then use the `Maybe`-specific definitions of `(>>=)`, `return`, and `guard` to reduce the expression when an admin is found, a non-admin is found, or no user is found.

Next, let's look at the individual pieces we'll be combining into `main`:

```
putStr :: String -> IO ()

putStrLn :: String -> IO ()
```

`putStr` also doesn't have any useful result so it uses `()`. It takes the given `String` and returns an action that *represents* printing that string, without a trailing newline, to the terminal. `putStrLn` is exactly the same, but includes a trailing newline.

```
getLine :: IO String
```

`getLine` doesn't take any arguments and has type `IO String` which means an action that represents reading a line of input from the terminal. It requires `IO` and presents the read line as its result.

Next, let's review the type of `(>>=)`:

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

In our case, `m` will always be `IO`, but `a` and `b` will be different each time we use `(>>=)`. The first combination we need is `putStr` and `getLine`. `putStr "..."` fits as `m a`, because its type is `IO ()`, but `getLine` does not have the type `() -> IO b` which is required for things to line up. There's another operator, built on top of `(>>=)`, designed to fix this problem:

```
(>>) :: m a -> m b -> m b
ma >> mb = ma >>= \_ -> mb
```

It turns its second argument into the right type for `(>>=)` by wrapping it in a lambda that accepts and ignores the `a` returned by the first action. With this, we can write our first combination:

```
main = putStr "..." >> getLine
```

What is the type of this expression? If `(>>)` is `m a -> m b -> m b` and we've got `m a` as `IO ()` and `m b` as `IO String`. This combined expression must be `IO String`. It

represents an action that, *when executed*, would print the given string to the terminal, then read in a line.

Our next requirement is to put this action together with `putStrLn`. Our current expression has type `IO String` and `putStrLn` has type `String -> IO ()`. This lines up perfectly with `(>>=)` by taking `m` as `IO`, `a` as `String`, and `b` as `()`:

```
main = putStr "..." >> getLine >>= putStrLn
```

This code is equivalent to the do-notation version I showed before. If you're not sure, try to manually convert between the two forms. The steps required were shown in the do-notation sub-section of the Monad chapter.

Hopefully, this exercise has convinced you that while I/O in Haskell may appear confusing at first, things are quite a bit simpler:

- Any function with an `IO` type *represents* an action to be performed
- Actions are not executed, only combined into larger actions using `(>>=)`
- The only way to get the runtime to execute an action is to assign it the special name `main`

From these rules and the general requirement of type-safety, it emerges that any value of type `IO a` can only be called directly or indirectly from `main`.

Other instances

Unlike previous chapters, here I jumped right into `Monad`. This was because there's a natural flow from imperative code to monadic programming with do-notation, to the underlying expressions combined with `(>>=)`. As I mentioned, this is the only way to combine `IO` values. While `IO` does have instances for `Functor` and `Applicative`, the functions in these classes (`fmap` and `(<*>)`) are defined in terms of `return` and `(>>=)` from its `Monad` instance. For this reason, I won't be showing their definitions. That said, these instances are still useful. If your `IO` code doesn't require the full power of monads, it's better to use a weaker constraint. More general programs are better; weaker constraints on what kind of data your functions can work with makes them more generally useful.

Functor

`fmap`, when specialized to `IO`, has the following type:

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

It takes a function and an `IO` action and returns another `IO` action, which represents applying that function to the *eventual* result returned by the first.

It's common to see Haskell code like this:

```
readInUpper :: FilePath -> IO String
readInUpper fp = do
    contents <- readFile fp

    return (map toUpper contents)
```

All this code does is form a new action that applies a function to the eventual result of another. We can say this more concisely using `fmap`:

```
readInUpper :: FilePath -> IO String
readInUpper fp = fmap (map toUpper) (readFile fp)
```

As another example, we can use `fmap` with the Prelude function `lookup` to write a safer version of `getEnv` from the `System.Environment` module. `getEnv` has the nasty quality of raising an exception if the environment variable you're looking for isn't present. Hopefully this book has convinced you it's better to return a `Maybe` in this case. The `lookupEnv` function was eventually added to the module, but if you intend to support old versions, you'll need to define it yourself:

```
import System.Environment (getEnvironment)

-- lookup :: Eq a => a -> [(a, b)] -> Maybe b
--
-- getEnvironment :: IO [(String, String)]

lookupEnv :: String -> IO (Maybe String)
lookupEnv v = fmap (lookup v) getEnvironment
```

Applicative

Imagine a library function for finding differences between two strings:

```
data Diff = Diff [Difference]
data Difference = Added | Removed | Changed

diff :: String -> String -> Diff
diff = undefined
```

How would we run this code on files from the file system? One way, using `Monad`, would look like this:

```
diffFiles :: FilePath -> FilePath -> IO Diff
diffFiles fp1 fp2 = do
    s1 <- readFile fp1
    s2 <- readFile fp2

    return (diff s1 s2)
```

Notice that the second `readFile` does not depend on the result of the first. Both `readFile` actions produce values that are combined *at once* using the pure function `diff`. We can make this lack of dependency explicit and bring the expression closer to what it would look like without `IO` values by using `Applicative`:

```
diffFiles :: FilePath -> FilePath -> IO Diff
diffFiles fp1 fp2 = diff <$> readFile fp1 <*> readFile fp2
```

As an exercise, try breaking down the types of the intermediate expressions here, like we did for `Maybe` in the Follow the Types sub-section of the Applicative chapter.

Learning more

There are many resources online for learning about `Monad` and `IO` in Haskell. I recommend reading them all. Some are better than others and many get a bad rap

for using some grandiose analogy that only makes sense to the author. Be mindful of this, but know that no single tutorial can give you a complete understanding because that requires looking at the same abstract thing from a variety of angles. Therefore, the best thing to do is read it all and form your own intuitions.

If you're interested in the origins of monadic I/O in Haskell, I recommend Tackling the Awkward Squad: monadic input/output, concurrency, exceptions, and foreign-language calls in Haskell by Simon Peyton Jones and Comprehending Monads by Philip Wadler.

What's Next

At the start of this book, I said my intention was not to teach you Haskell. Instead, my goal was to give you a sense of writing realistic code that takes advantage of an uncommon feature found in the Haskell language. I wanted to strike a balance between short sound-bites without much depth and the large investment of time required to become proficient in the language.

The central theme, the `Maybe` type, is an example of Haskell's principled stance resulting in tangible benefit for programmers. The frustration caused by Tony Hoare's self-proclaimed "billion-dollar mistake" is something I'll gladly live without. This same principled stance has lead to many similar outcomes in the Haskell language. From monadic I/O to advanced concurrency primitives, Haskell is full of constructs only made possible through slow and thoughtful language design. My hope is that by seeing–and really understanding–the relatively small example that is `Maybe`, you'll be motivated to explore the language further.

If you are so inclined, there are many resources online for getting up and running, what to read, etc. I mentioned Chris Allan's learning path already. There's also Haskell in 5 steps. I'm a huge fan of Learn You a Haskell for Great Good! and Real World Haskell if you're looking for books. If you prefer an exercise-based approach, we have a Haskell Fundamentals trail on Upcase. Finally, don't be afraid to read academic papers. They are dense sources of very good information.