So lazy as always

Chapter 11: Functors, Applicative Functors and Monoids:

Summary:
Functors in essence are things that can be mapped over (e.g. lists, Maybes, trees…). All functors follows two laws where the first is if you map the id function over a functor, the functor that you get back should be the original functor. The second is that composing two functions and then mapping the resulting function over a functor should be the same as first mapping one function over the functor and then mapping the other one, which is very similar to composition. Now comes the Applicative typeclass that defines two methods, pure and <*> which you have to define yourself. They are useful because you can combine different computations (I/O, non-deterministic, and etc.…) by using the applicative style.

Notes:

Functors redux:

Functors are things that can be mapped over, like lists, Maybes, trees, and such. In Haskell, they're described by the typeclass Functor, which has only one typeclass method, namely fmap, which has a type of fmap :: (a -> b) -> f a -> f b.

short summary is that it asks for a function taking in an 'a' and returns a 'b' and a box with an 'a' inside it, and it'll give you a box with a b inside of it.

If we want to make a type constructor an instance of Functor, it has to have a kind of * -> *, which means that it has to take exactly one concrete type as a type parameter.

If some value has a type of, say, IO String, that means that it's an I/O action that, when performed, will go out into the real world and get some string for us, which it will yield as a result. We can use <-
in do syntax to bind that result to a name.

If we look at what fmap's type would be if it were limited to IO, it would be fmap :: (a -> b) -> IO a -> IO b. fmaptakes a function and an I/O action and returns a new I/O action that's like the old one, except that the function is applied to its contained result.

Another instance of Functor that we've been dealing with all along but didn't know was a Functor is (->) r.

This makes the revelation that using fmap over functions is just composition sort of obvious.

if we write fmap :: (a -> b) -> (f a -> f b), we can think of fmap not as a function that takes one function and a functor and returns a functor, but as a function that takes a function and returns a new function that's just like the old one, only it takes a functor as a parameter and returns a functor as the result. It takes an a -> b function and returns a function f a -> f b. This is called lifting a function.

You can think of fmap as either a function that takes a function and a functor and then maps that function over the functor, or you can think of it as a function that takes a function and lifts that function so that it operates on functors

Functor Laws:

The first functor law states that if we map the id function over a functor, the functor that we get back should be the same as the original functor.

The second law says that composing two functions and then mapping the resulting function over a functor should be the same as first mapping one function over the functor and then mapping the other one.
that means that fmap (f . g) = fmap f . fmap g. Or to write it in another way, for any functor F, the following should hold:fmap (f . g) F = fmap f (fmap g F).

All the Functor instances in the standard library obey these laws.
We can also look at functors as things that output values in a context. For instance, Just 3 outputs the value 3 in the context that it might or not output any values at all.
If you think of functors as things that output values, you can think of mapping over functors as attaching a transformation to the output of the functor that changes the value.

Looking at it this way gives us some intuition as to why using fmap on functions is just composition (fmap (+3) (*3) equals(+3) . (*3), which equals \x -> ((x*3)+3))
The result is still a function, only when we give it a number, it will be multiplied by three and then it will go through the attached transformation where it will be added to three. This is what happens with composition.

<u>Applicative Functors:</u>
As you know, functions in Haskell are curried by default, which means that a function that seems to take several parameters actually takes just one parameter and returns a function that takes the next parameter and so on.

We see how by mapping "multi-parameter" functions over functors, we get functors that contain functions inside them.

Meet the Applicative typeclass. It lies in the Control.Applicative module and it defines two methods, pure and <*>. It doesn't provide a default implementation for any of them, so we have to define them both if we want something to be an applicative functor. The class is defined like so:

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

It starts the definition of the Applicative class and it also introduces a class constraint. It says that if we want to make a type constructor part of the Applicative typeclass, it has to be in Functor first. That's why if we know that if a type constructor is part of the Applicative typeclass, it's also inFunctor, so we can use fmap on it.

pure should take a value of any type and return an applicative functor with that value inside it. We take a value and we wrap it in an applicative functor that has that value as the result inside it.

A better way of thinking about pure would be to say that it takes a value and puts it in some sort of default (or pure) context—a minimal context that still yields that value.

The <*> function is a beefed up fmap. Whereas fmap takes a function and a functor and applies the function inside the functor, <*> takes a functor that has a function in it and another functor and sort of extracts that function from the first functor and then maps it over the second one. When I say extract, I actually sort of mean run and then extract, maybe even sequence.

Applicative functors and the applicative style of doing pure f <*> x <*> y <*> ... allow us to take a function that expects parameters that aren't necessarily wrapped in functors and use that function to operate on several values that are in functor contexts. The function can take as many parameters as we want, because it's always partially applied step by step between occurences of <*>.

This becomes even more handy and apparent if we consider the fact that pure f <*> x equals fmap f x. This is one of the applicative laws.

This is why Control.Applicative exports a function called <$>, which is just fmap as an infix operator. Here's how it's defined:

```
(<$>) :: (Functor f) => (a -> b) -> f a -> f b
f <$> x = fmap f x
```

By using <$>, the applicative style really shines, because now if we want to apply a function f between three applicative functors, we can write f <$> x <*> y <*> z. If the parameters weren't applicative functors but normal values, we'd writef x y z.

Lists (actually the list type constructor, []) are applicative functors. What a suprise! Here's how [] is an instance of Applicative:

```
instance Applicative [] where
    pure x = [x]
    fs <*> xs = [f x | f <- fs, x <- xs]
```

Because <*> is left-associative, [(+),(*)] <*> [1,2] happens first, resulting in a list that's the same as[(1+),(2+),(1*),(2*)], because every function on the left gets applied to every value on the right. Then,[(1+),(2+),(1*),(2*)] <*> [3,4] happens, which produces the final result.

Another instance of Applicative that we've already encountered is IO. This is how the instance is implemented:

```
instance Applicative IO where
    pure = return
    a <*> b = do
        f <- a
        x <- b
        return (f x)
```

Since pure is all about putting a value in a minimal context that still holds it as its result, it makes sense that pure is just return, because return does exactly that; it makes an I/O action that doesn't do anything, it just yields some value as its result, but it doesn't really do any I/O operations like printing to the terminal or reading from a file

Calling <*> with two applicative functors results in an applicative functor, so if we use it on two functions, we get back a function. So what goes on here? When we do (+) <$> (+3) <*> (*100), we're making a function that will use + on the results of (+3) and (*100) and return that. It makes an 8 and 500 then adds together.

Remember, : is a function that takes an element and a list and returns a new list with that element at the beginning.

Then some stuffs with zipwith and they mention recursion on lists.

Like normal functors, applicative functors come with a few laws. The most important one is the one that we already mentioned, namely that pure f <*> x = fmap f x holds. As an exercise, you can prove this law for some of the applicative functors that we've met in this chapter.The other functor laws are:

- pure id <*> v = v
- pure (.) <*> u <*> v <*> w = u <*> (v <*> w)
- pure f <*> pure x = pure (f x)
- u <*> pure y = pure ($ y) <*> u

In conclusion, applicative functors aren't just interesting, they're also useful, because they allow us to combine different computations, such as I/O computations, non-deterministic computations, computations that might have failed, etc. by using the applicative style. Just by using <$> and <*> we can use normal functions to uniformly operate on any number of applicative functors and take advantage of the semantics of each one.

Summary of these things from ch13:
When we first talked about functors, we saw that they were a useful concept for values that can be mapped over. Then, we took that concept one step further by introducing applicative functors, which allow us to view values of certain data types as values with contexts and use normal functions on those values while preserving the meaning of those contexts.

Chapter 13: Functionally Solving Problems:

Summaries:
Monads can be used to take values with contexts and apply them to functions and how using >>= or do notation allow you to focus on the values themselves while context is handled for you. The Maybe monad added context of possible failure to values and the list monad lets you easily introduce none-determinism into the programs. There was more indepth of the IO monad before we knew what a monad was.

Notes:
Monads are a natural extension of applicative functors and with them we're concerned with this: if you have a value with a context, m a, how do you apply to it a function that takes a normal a and returns a value with a context? That is, how do you apply a function of type a -> m b to a value of type m a? So essentially, we will want this function:

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

If we have a fancy value and a function that takes a normal value but returns a fancy value, how do we feed that fancy value into the function? This is the main question that we will concern ourselves when dealing with monads. We write m a instead of f a because the m stands for Monad, but monads are just applicative functors that support >>=. The >>= function is pronounced as bind.

Much to no one's surprise, Maybe is a monad.

A value of type Maybe a represents a value of type a with the context of possible failure attached

However, applicatives also have the function wrapped. Maybe is an applicative functor in such a way that when we use <*> to apply a function inside a Maybe to a value that's inside a Maybe, they both have to be Just values for the result to be a Just value, otherwise the result is Nothing.

And now, let's think about how we would do >>= for Maybe. Like we said, >>= takes a monadic value, and a function that takes a normal value and returns a monadic value and manages to apply that function to the monadic value.

In this case, >>= would take a Maybe a value and a function of type a -> Maybe b and somehow apply the function to the Maybe a.

Instead of calling it >>=, let's call it applyMaybe for now. It takes a Maybe a and a function that returns a Maybe b and manages to apply that function to the Maybe a. Here it is in code:

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

The Monad type class:

```
class Monad m where
   return :: a -> m a

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

   (>>) :: m a -> m b -> m b
   x >> y = x >>= \_ -> y

   fail :: String -> m a
   fail msg = error msg
```

every monad is an applicative functor, even if the Monad class declaration doesn't say so.

The first function that the Monad type class defines is return. It's the same as pure

return is nothing like the return that's in most other languages. It doesn't end function execution or anything, it just takes a normal value and puts it in a context.

The next function is >>=, or bind. It's like function application, only instead of taking a normal value and feeding it to a normal function, it takes a monadic value (that is, a value with a context) and feeds it to a function that takes a normal value but returns a monadic value.

Now some example on using >>= repeatedly, look it up.

We couldn't have achieved this by just using Maybe as an applicative. If you try it, you'll get stuck, because applicative functors don't allow for the applicative values to interact with each other very much. They can, at best, be used as parameters to a function by using the applicative style. The applicative operators will fetch their results and feed them to the function in a manner appropriate for each applicative and then put the final applicative value together, but there isn't that much interaction going on between them. Here, however, each step relies on the previous one's result. On every landing, the possible result from the previous one is examined and the pole is checked for balance. This determines whether the landing will succeed or fail.

do notation:
Monads in Haskell are so useful that they got their own special syntax called do notation.
Well, as it turns out, do notation isn't just for IO, but can be used for any monad. Its principle is still the same: gluing together monadic values in sequence.

In a do expression, every line is a monadic value. To inspect its result, we use<-. If we have a Maybe String and we bind it with <- to a variable, that variable will be a String, just like when we used >>= to feed monadic values to lambdas. The last monadic value in a do expression, likeJust (show x ++ y) here, can't be used with <- to bind its result, because that wouldn't make sense if we translated
the do expression back to a chain of>>= applications. Rather, its result is the result of the whole glued up monadic value, taking into account the possible failure of any of the previous ones.

Because do expressions are written line by line, they may look like imperative code to some people. But the thing is, they're just sequential, as each value in each line relies on the result of the previous ones, along with their contexts

The list monad:
```
instance Monad [] where
   return x = [x]
   xs >>= f = concat (map f xs)
   fail _ = []
```

return does the same thing as pure, so we should already be familiar with return for lists. It takes a value and puts it in a minimal default context that still yields that value. In other words, it makes a list that has only that one value as its result.

Then a bunch of examples on this…

Then an example using Knight from chess.

Monad Laws:
left identity:

The first monad law states that if we take a value, put it in a default context with **return** and then feed it to a function by using**>>=**, it's the same as just taking the value and applying the function to it. To put it formally:

- return x >>= f is the same damn thing as f x

Right identity:

The second law states that if we have a monadic value and we use **>>=** to feed it to **return**, the result is our original monadic value. Formally:

- m >>= return is no different than just m

Associativity

The final monad law says that when we have a chain of monadic function applications with **>>=**, it shouldn't matter how they're nested. Formally written:

- Doing (m >>= f) >>= g is just like doing m >>= (\x -> f x >>= g)