Functors, Applicatives, and Monads

In this section we discuss three important type classes that pertain to container types. They each express a key idea:

- Functors express the idea that we can `map` a function over a container type, by applying the function to its contents.
- Applicatives express the idea of mapping a function over possibly multiple arguments.
- Monads express the idea of sequencing effectful operations.

Reading

- Sections 12.1-12.3
- Practice exercises (12.5): 1, 4, 7
- Optional practice: 2, 3, 8
- https://wiki.haskell.org/State_Monad

Container Types and the State "Monad"

Before we start exploring functors, let us consider three important container types that share certain features. All three types express the idea of "producing a value of a given type", but they do so in different ways:

The Maybe type expresses the idea that the operation we performed may or may not have produced a value (i.e. it may have failed).

[a] Maybe a

The list type expresses the idea that the operation we performed may have returned more than one possible values (but all of the same type). A good example is the list comprehensions we discussed earlier in the course. A list is a convenient way to represent these possibly multiple results.

The `IO` type expresses the idea that our operation interacts with the world in some way, and produces a value of type `a`.

The State "monad" (we will discuss what that means later) is another example. It is meant to somehow maintain and update some "state", and also return a value of type `a`. We will discuss this type now in some more detail.

The idea of the State type is similar to our view of `IO` as a function that changed the "world" in some way and also produced a value of type `a`. The State type makes that more precise. It can work with very generic "states", represented here with the type `s`. We can then make the following definition:

```
newtype ST s a = S (s -> (a, s))
```

So a value of type `ST a` is a *transition* function that takes the current state, and produces a value of type `a` along with a new (updated) state. For technical reasons we place that function inside an `S` tag. We can easily write a function that removes the tag:

```
runState :: ST s a -> s -> (a, s)
runState (S trans) x = trans x
-- Could also have done: runState (S trans) = trans
```

We can also write a function that evaluates a given stateful computation for a state, discards the resulting state and simply returns the final value:

```
evalState :: ST s a -> s -> a
evalState (S trans) st = x'
    where (x', _) = runState (S trans) st
-- Alternative definition: evalState state = fst . runState state
```

In order to meaningfully work with this new state structure though, we will need a couple of things:

- A way to set the state to a new value. This is akin to `putStr` printing something to the screen and hence changing the state of IO.
- A way to read the state, while going through a stateful computation.
- A way to turn a normal value into a stateful computation. We did this for `IO` with a function called `return`, and we will do the same here.
- A way to apply a normal function to the value in the computation, while keeping the state the same.
- A way to chain two computations together, so that the state transfers from the first to the second. In Haskell that operation has a name, `(>>=)`, typically called a "bind" operation.

Let us take a look at each of these. Getting and setting the state is easy:

```
get :: ST s s
get = S (\st -> (st, st))

put :: s -> ST s ()
put st = S (\_ -> ((), st))
```

Now we need functions for returning a normal value as the result of a stateful computation that does not change the state, and also mapping the result values through a normal function:

```
return :: a -> ST s a
return x = S (\st -> (x, st))

fmap :: (a -> b) -> ST s a -> ST s b
fmap f (S trans) = S trans'
    where trans' st = (f x, st')
                  where (x, st') = trans st
```

Now comes the tricky bit: We want to combine two stateful computations into a new stateful computation. Actually what we will do is slightly different: We will combine one stateful computation with a function that takes as input a result of the first computation, and uses it to produce a second stateful computation that is then carried out. It is probably the hardest part of the whole story:

```
(>>=) :: ST s a -> (a -> ST s b) -> ST s b
act1 >>= f = S trans
    where trans st = runState act2 st'
                     where (x, st') = runState act1 st
                           act2     = f x
```

This all looks a bit messy, but we only had to do it and understand it once! Now we can use this chaining instead of having to effectively manually do it every time. The even cooler thing is that this effectively allows us to use the IO-type notation with `do`, and Haskell will unravel that for us. For example, we can build a function that modifies the current state, as follows:

```
modify :: (s -> s) -> ST s ()
modify f = do
    st <- get
    put st
```

And Haskell turns that into:

```
modify f = get >>= (\st -> put (f st))
-- Can also write as:   get >>= (put . f)
```

which is perhaps elegant but somewhat harder to read, especially if it involved more steps. In order for Haskell to be able to carry this out, it must know that our `ST` structure is a "monad". We will discuss what that means later in the chapter, but effectively it just means having the "bind" operation we just defined.

Practice:

1. Using "do" notation, write a function `incr :: ST Int ()` that increments the integer state by 1. Also do it by instead using `modify`.
2. Using "do" notation, write a function `account :: a -> ST Int a` which "accounts" for the computation that produces a value of type `a`, by incrementing the integer state. Your function should simply increment the state and return the provided value.

Functors

TODO

Applicatives

TODO

Monads

TODO