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

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", but for simplicity we will assume that we have a specific state type, namely integers. We can then make the following definitions:

```
type State = Int
newtype ST a = S (State -> (a, State))
```

So a value of type `ST a` is a 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:

```
app :: ST a -> State -> (a, State)
app (S st) x = st x
-- Could also have done: app (S st) = st
```

TODO

Functors

TODO

Applicatives

TODO

Monads

TODO