Notes from 10/26

"Lambda" notation

A lambda is an anonymous function. Normally when you define a function, you write its *name* first, then parameter, then the body.

```
square1 x = x*x
```

To make it anonymous, you omit the name, and instead use a backslash (which looks a bit like a Greek lowercase lambda: λ).

```
\x -> x*x
```

This is an *expression* not a definition – you wouldn't write just that at the top-level in a Haskell file. However you could still bind the expression to a name:

```
square2 = \x -> x*x
```

You can consider this to be an alternative syntax for a function definition. But the lambda notation is most useful for passing to higher-order functions.

For example, when we do something like these examples:

```
ghci> map square2 [2..4]
[4,9,16]
ghci> map (*8) [2..4]
[16,24,32]
```

We are passing functions to `map`. In the first example, it's a named function, `square1`. In the second example we use a *section* on an operator to produce a function. A more general way to specify a function in that position is a lambda expression:

```
ghci> map (\x -> 2*x*x + 3*x - 4) [5..10]
[61,86,115,148,185,226]
```

Here, the lambda expression represents the polynomial $2x^2 + 3x - 4$, and we have evaluated for each x in the given range.

The operator *section* notation can now be seen as just a shortcut for a lambda expression:

- `(*2)` is equivalent to `(\x -> x*2)`
- `(3-)` is equivalent to `(\x -> 3-x)`

Monad operations

A type is a monad `m` if it provides the following two operations:

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

The operator (>>=) is pronounced as "bind", "andThen", or sometimes "flatMap". The most well-understood monad types are `Maybe` and lists. If we specialize the types above to `Maybe` and `[]`, they are:

```
return :: a -> Maybe a
(>>=) :: Maybe a -> (a -> Maybe b) -> Maybe b

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

We saw the `Maybe` version of this in Assignment 4, where I had you implement it as `andThen`. It's used to sequence together operations that may fail.

Maybe monad

Here's a rather contrived example where someone wants to log in to their bank account, check their balance, and then transfer some money. Each of those operations can fail, so each function returns `Maybe`.

```haskell
login :: String -> String -> Maybe Int  -- returns session ID
login user password =
  if user == reverse password  -- password is your username, reversed
  then Just (length user)      -- fake a session ID
  else Nothing

checkBalance :: Int -> Maybe Float
checkBalance sessionId =
  if odd sessionId        -- suppose even session IDs are offline
  then Just 215.38
  else Nothing

transfer :: Float -> Float -> Maybe Float
transfer amount balance
  | amount <= balance = Just (balance - amount)
  | otherwise = Nothing
```

So we sequence them together as follows, and each can fail for different reasons.

```
-- Works:
ghci> login "joe" "eoj" >>= checkBalance >>= transfer 50
Just 165.38

-- Fails because Joe doesn't have $500 to transfer
ghci> login "joe" "eoj" >>= checkBalance >>= transfer 500
Nothing

-- Fails because Fran's session ID is offline
ghci> login "fran" "narf" >>= checkBalance >>= transfer 50
Nothing

-- Fails because Bob has wrong password
ghci> login "bob" "secret" >>= checkBalance >>= transfer 50
Nothing
```

Of course, the unfortunate thing about `Maybe` in this example is that it can't directly tell us which step in the pipeline went wrong. But `Either` also forms a Monad, so you could try using `Left` to display error messages!

List monad

Here's a simple example of using `(>>=)` on a list of strings.

```
ghci> ["hello world", "this is a test", "bye for now"] >>= words
["hello","world","this","is","a","test","bye","for","now"]
```

Compare it to using `map`, where each word list stays together

```
ghci> map words ["hello world", "this is a test", "bye for now"]
[["hello","world"],["this","is","a","test"],["bye","for","now"]]
```

and you perhaps you can see why it's sometimes called `flatMap`.

State monad

Last week when we worked with threading state through a calculation or traversal, we defined a state function with a type like `s -> (r, s)` where `s` is the state and `r` is the result. It turns out that this function type is *itself* a monad.

Substitute `m a` in the monad operations with `s -> (a, s)` and you get:

```
return :: a -> s -> (a, s)
(>>=) :: (s -> (a, s)) -> (a -> s -> (b, s)) -> s -> (b, s)
```

We can actually write these functions, but we'll name them a little differently so they don't conflict with the real ones:

```
returnState :: a -> s -> (a,s)
returnState result state = (result, state)

bindState :: (s -> (a, s)) -> (a -> s -> (b, s)) -> s -> (b, s)
bindState action1 action2 state0 = (result2, state2)
  where (result1, state1) = action1 state0
        (result2, state2) = action2 result1 state1
```

Notice the usual careful state-passing in the where clause of the `bindState` function. That little bit of state-passing logic turns is sufficient to encode everything we need to build more elaborate stateful functions.

For example, the function `threeRandoms` becomes:

```
threeRandoms :: Seed -> ([Integer], Seed)
threeRandoms =
  rand `bindState` \r1 ->
  rand `bindState` \r2 ->
  rand `bindState` \r3 ->
  returnState [r1, r2, r3]
```

This has the same type as the `threeRandoms` we did list week, but all the state-passing is hidden. It's done transparently by `bindState` and `returnState`, and we get exactly the same results.

```
ghci> threeRandoms (Seed 3453)
([58034571,429459059,225867046],Seed {unSeed = 225867046})

data Seed = Seed { unSeed :: Integer }
  deriving (Eq, Show)

rand :: Seed -> (Integer, Seed)
rand (Seed s) = (s', Seed s')
  where
    s' = (s * 16807) `mod` 0x7FFFFFFF

main = putStrLn "OK"
```