# Notes from 11/30

## Administrative

Take-home exam will be provided on Dec 14th (Thu), and due at 23:59 on Dec 20th (Wed). No class on Dec 21st.

## Module header

Here are the language extensions and module imports that I use in these notes.

```
{-# LANGUAGE OverloadedStrings #-}

module N20171130 where
import Control.Monad
import Data.ByteString (ByteString)
import Data.String
import Data.String.Conversions (cs)
import Data.Text (Text)
import qualified Data.ByteString as BS
import qualified Data.Map as M
import qualified Data.Text as T
```

For `Data.String.Conversions` to work, you probably need to run:

```
stack install string-conversions
```

outside of GHCi.

## String types

Built-in strings are just lists of chars. This is convenient because it allows us to use list functions like `length`, `map`, `take`/`drop`, but it's a pretty inefficient representation. If you have a large piece of text, just imagine all the memory and pointers you'd need to represent it as a linked list of individual characters.

So there are libraries containing more efficient representations, such as `ByteString` and `Text`. Always import them *qualified*, as I did above, because they contain names that conflict with names in the standard Prelude (and with each other).

- `ByteString` holds arbitrary sequences of bytes. Each byte is an 8-bit number. This is what you want if you're handling purely binary data.
- `Text` is intended for strings of text, where each element in the sequence is a *character*. Characters can actually be multi-byte, in an encoding such as UTF-8.

Haskell has an extension called `OverloadedStrings` that makes it easy to use the same string syntax (double quotes) for different string types:

```
s1 :: String
s1 = "HelloStr"
s2 :: Text
s2 = "HelloTxt"
s3 :: ByteString
s3 = "HelloBS"
```

We needed the `{-# LANGUAGE OverloadedStrings #-}` code at the top of this file to make that work. Also, when you're trying to write such strings directly in GHCi, you need to turn on the extension there too:

```
ghci> :set -XOverloadedStrings
```

Even though the values `s1`, `s2`, and `s3` are specified the same way they are different, incompatible types. For example, we need to use different functions to get their length:

```
ghci> length s1
8
ghci> T.length s2
8
ghci> BS.length s3
7

ghci> length s2
<interactive>:43:8-9: error:
    • Couldn't match expected type ‘[a0]’ with actual type ‘Text’

ghci> T.length s3
<interactive>:44:10-11: error:
    • Couldn't match expected type ‘Text’ with actual type ‘ByteString’
```

There are customized versions of most of the list/sequence functions within the `Text` and `ByteString` modules. Here is `take`, for example, and then append:

```
ghci> take 5 s1
"Hello"
ghci> T.take 5 s2
"Hello"
ghci> BS.take 5 s3
"Hello"

ghci> s1 ++ s1
"HelloStrHelloStr"
ghci> T.append s2 s2
"HelloTxtHelloTxt"
ghci> BS.append s3 s3
"HelloBSHelloBS"
```

We're not allowed to append strings of mismatching types, although string literals (double quotes) can automatically work with any:

```
ghci> s1 ++ s2
<interactive>:64:7-8: error:
    • Couldn't match expected type ‘[Char]’ with actual type ‘Text’

ghci> T.append s2 s3
<interactive>:65:13-14: error:
    • Couldn't match expected type ‘Text’ with actual type ‘ByteString’

ghci> s1 ++ "OK"
"HelloStrOK"
ghci> T.append s2 "OK"
"HelloTxtOK"
```

Another nice trick is that all the string types implement the `Monoid` class, so if you import `Data.Monoid` you can use the <> (`mappend`) operator. But you still cannot mix and match types:

```
ghci> s1 <> s1
"HelloStrHelloStr"
ghci> s2 <> s2
"HelloTxtHelloTxt"
ghci> s3 <> s3
"HelloBSHelloBS"

ghci> s1 <> s2
<interactive>:64:7-8: error:
    • Couldn't match type ‘Text’ with ‘[Char]’
```

```
ghci> s2 <> s3
<interactive>:65:13-14: error:
    • Couldn't match expected type ‘Text’ with actual type ‘ByteString’
```

**Aside:** the story about `Text` and `ByteString` is even a little more complicated, because each of them have both *lazy* and *strict* implementations. I'm ignoring that here and using the default, but in some programs you might need to distinguish between:

```
import qualified Data.ByteString as BS
import qualified Data.ByteString.Lazy as LBS
```

### String conversions

So how do we convert between different string types? There are custom techniques for each kind of conversion, but I find them hard to remember. Generally, when going from list-of-characters to an efficient representation, it's called pack (and the reverse is unpack):

```
ghci> T.unpack s2 :: String
"HelloTxt"
ghci> T.pack s1 :: Text
"HelloStr"
```

But I find it really convenient to use the `string-conversions` package, which contains the module `Data.String.Conversions` and a function `cs` (stands for convert string). It can convert pretty much any string type to any other, automatically. So then we can append different string types.

```
ghci> s1 ++ cs s2
"HelloStrHelloTxt" :: String
ghci> T.append (cs s1) s2
"HelloStrHelloTxt" :: Text
```

Here's a peculiarity you'll notice between `Text` and `ByteString`. Remember that `Text` is a packed representation of (potentially multi-byte) characters, but `ByteString` is an arbitrary sequence of bytes. So if we create a `Text` string containing non-ASCII characters:

```
nonASCII :: Text
nonASCII = "é¶ß  "
```

Its length is sensible:

```
ghci> T.length nonASCII
5
```

But if we convert to a `ByteString`, it expands!

```
ghci> BS.length (cs nonASCII)
10
ghci> cs nonASCII :: BS.ByteString
"\195\169\194\182\195\159\206\187\206\147"
```

Those backslashed numbers indicate byte values (in base ten). So the character é, known to Unicode as LATIN SMALL LETTER E WITH ACUTE is encoded in UTF-8 as the two bytes 0xC3 0xA9. They're specified in hexadecimal there, but they're the same values that Haskell produced for the first character.

```
ghci> 0xc3
195
ghci> 0xa9
169
```

### User-defined string type

You can also use `OverloadedStrings` with your own types. Just import `Data.String` and hen declare your type as an instance of `IsString`. You need to provide the function `fromString`.

```haskell
data Color = R | G | B deriving (Eq, Show)

instance IsString Color where
  fromString "R" = R
  fromString "r" = R
  fromString "G" = G
  fromString "g" = G
  fromString "B" = B
  fromString "b" = B
  fromString _ = error "No such color"
```

```
ghci> R == "R"
True
ghci> G == "B"
False
```

Those expressions are not type errors, because the string literal can be converted automatically to any type that implements `IsString`. So they are converted to `Color` to match the left side of the equals.

## Rationals

Haskell also has overloaded numbers, and there are many different numeric types:

```
ghci> 1+2 :: Int
3
ghci> 1+2 :: Integer
3
ghci> 1+2 :: Float
3.0
ghci> 1+2 :: Double
3.0
```

Another numeric type we haven't encountered yet is `Rational`. A rational number is a ratio of two integers. In Haskell the ratio is expressed with a percent sign:

```
ghci> 1+2 :: Rational
3 % 1
```

So $1 + 2$ expressed as a ratio is three-to-one, or $\frac{3}{1}$. The division operator (`/`) can produce either floats, doubles, or rationals:

```
ghci> 4/6 :: Float
0.6666667
ghci> 4/6 :: Double
0.6666666666666666
ghci> 4/6 :: Rational
2 % 3
```

Notice it simplified $\frac{4}{6}$ into $\frac{2}{3}$. The nice thing about rationals is you get an exact representation, which float and double are not. You of course can't represent *irrational* numbers, such as $\pi$ or $\sqrt{2}$, but floats don't represent them precisely either!

## List comprehensions

I may have covered this before, but it's an extremely convenient notation for using nested loops and conditions to generate a list. Here's an example:

```
ghci> [(c,i) | c <- "zob", i <- [3..6]]
[('z',3),('z',4),('z',5),('z',6),('o',3),('o',4),
 ('o',5),('o',6),('b',3),('b',4),('b',5),('b',6)]

ghci> [replicate i c | c <- "zob", i <- [3..6]]
```

```
["zzz","zzzz","zzzzz","zzzzzz",
 "ooo","oooo","ooooo","oooooo",
 "bbb","bbbb","bbbbb","bbbbbb"]
```

Here's an example with a condition:

```
ghci> [x*y | x <- [2..4], y <- [5..7], even (x+y)]
[12,15,21,24]
```

The results represent `[2*6, 3*5, 3*7, 4*6]` because out of the nine pairs you can draw from the two x and y generators, only those four combinations have even sums.

## Probability monad

We'll represent discrete probability distributions as a list of each possibly value paired with its probability, as a rational number.

```haskell
data Prob a = Prob { toList :: [(a, Rational)] }
  deriving (Show)
```

So if there's only one possible value, the probability is 1.

```haskell
always :: a -> Prob a
always a = Prob [(a, 1)]
```

We can represent a coin flip as a probability of a Boolean value (True for heads, False for tails, let's say).

```haskell
coinFlip :: Prob Bool
coinFlip = Prob [(True, 1/2), (False, 1/2)]
```

Or here's a more general technique, that works for selecting a value from any non-empty list, with equal probability.

```haskell
oneOf :: [a] -> Prob a
oneOf as = Prob (map g as)
  where g a = (a, 1 / toRational(length as))
```

So rolling a 6-sided die can be represented as:

```haskell
diceRoll :: Prob Int
diceRoll = oneOf [1..6]
```

The sum of probabilities in a distribution should always be 1.

```
sumProb :: Prob a -> Rational
sumProb (Prob xs) = sum (map snd xs)
```

```
ghci> sumProb coinFlip
1 % 1
ghci> sumProb diceRoll
1 % 1
ghci> sumProb $ always "FOO"
1 % 1
ghci> sumProb $ oneOf "My impressive character collection"
1 % 1
```

### Optimizing

That last example seems a little fishy though. Take a look at the result without the `sumProb`:

```
ghci> oneOf "My impressive character collection"
Prob {toList = [('M',1 % 34),('y',1 % 34),(' ',1 % 34),('i',1 % 34),
('m',1 % 34),('p',1 % 34),('r',1 % 34),('e',1 % 34),('s',1 % 34),
('s',1 % 34),('i',1 % 34),('v',1 % 34),('e',1 % 34),(' ',1 % 34),
('c',1 % 34),('h',1 % 34),('a',1 % 34),('r',1 % 34),('a',1 % 34),
('c',1 % 34),('t',1 % 34),('e',1 % 34),('r',1 % 34),(' ',1 % 34),
('c',1 % 34),('o',1 % 34),('l',1 % 34),('l',1 % 34),('e',1 % 34),
('c',1 % 34),('t',1 % 34),('i',1 % 34),('o',1 % 34),('n',1 % 34)]}
```

The list enumerates every single character, each with a $\frac{1}{34}$ chance, but the same character can appear multiple times. We'll treat that as valid, but it's much more efficient if we can eliminate duplicates. A good way to do that is a `Map`. (I imported `Data.Map as M`). But `M.fromList` isn't good enough:

```
ghci> M.fromList (toList (oneOf "My impressive character collection"))
fromList [(' ',1 % 34),('M',1 % 34),('a',1 % 34),('c',1 % 34),
('e',1 % 34),('h',1 % 34),('i',1 % 34),('l',1 % 34),('m',1 % 34),
('n',1 % 34),('o',1 % 34),('p',1 % 34),('r',1 % 34),('s',1 % 34),
('t',1 % 34),('v',1 % 34),('y',1 % 34)]

ghci> sumProb (Prob (M.toList (M.fromList
     (toList (oneOf "My impressive character collection")))))
1 % 2
```

Now each character only appears once, but the probabilities don't add to one! When there are duplicate keys, `M.fromList` just uses the last one and discards the rest.

Instead, let's use `M.fromListWith` which allows us to specify what to do with the values of duplicate keys. We'll add them.

```
optimize :: Ord a => Prob a -> Prob a
optimize (Prob xs) =
  Prob (M.toList (M.fromListWith (+) xs))
```

```
ghci> optimize (oneOf "My impressive character collection")
Prob {toList = [(' ',3 % 34),('M',1 % 34),('a',1 % 17),('c',2 % 17),
('e',2 % 17),('h',1 % 34),('i',3 % 34),('l',1 % 17),('m',1 % 34),
('n',1 % 34),('o',1 % 17),('p',1 % 34),('r',3 % 34),('s',1 % 17),
('t',1 % 17),('v',1 % 34),('y',1 % 34)]}
ghci> sumProb (optimize (oneOf "My impressive character collection"))
1 % 1
```

### Functor

We want to be able to do calculations on probability distributions – transforming them, sequencing them, etc. The nicest way to do it is to implement the appropriate type-classes: `Functor`, `Applicative`, and `Monad`.

A functor `f` must support a function

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

In the case of the probability distribution, that's

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

So that means we're modifying the values, but preserving probabilities. Here's the implementation:

```
instance Functor Prob where
  fmap f (Prob xs) = Prob (map g xs)
    where
      g (a,p) = (f a, p) -- Apply f to value a, preserve probability p
```

We can use this, for example, to double the value of a dice roll:

```
ghci> fmap (*2) diceRoll
Prob {toList = [(2,1 % 6),(4,1 % 6),(6,1 % 6),(8,1 % 6),(10,1 % 6),(12,1 % 6)]}
ghci> (*2) <$> diceRoll
Prob {toList = [(2,1 % 6),(4,1 % 6),(6,1 % 6),(8,1 % 6),(10,1 % 6),(12,1 % 6)]}
```

(`fmap` can also be written as the infix operator `<$>`.) So we can get the values `2,4,6,8,10,12`, but they're all equally likely with 1-in-6 probability.

What if we integer-divide the dice roll?

```
ghci> (`div` 2) <$> diceRoll
Prob {toList = [(0,1 % 6),(1,1 % 6),(1,1 % 6),(2,1 % 6),(2,1 % 6),(3,1 % 6)]}
```

Now we end up with some duplicate values, so optimize it:

```
ghci> optimize $ (`div` 2) <$> diceRoll
Prob {toList = [(0,1 % 6),(1,1 % 3),(2,1 % 3),(3,1 % 6)]}
```

The result is that we get 1 or 2 each with probability $\frac{1}{3}$, or 0 or 3 each with probability $\frac{1}{6}$. The sum is still 1.

As another example, we can ask whether the dice roll is even or odd.

```
ghci> optimize $ even <$> diceRoll
Prob {toList = [(False,1 % 2),(True,1 % 2)]}
```

It ends up being the same distribution as a coin flip!

### Applicative

We didn't talk much about Applicative yet, but it's like a weaker version of Monad. You can sequence computations, but they must be *independent*. The second computation in a sequence doesn't know the result of the first. (But then both results can be merged together later.)

An applicative requires a very simple function `pure` which is just the same thing as the `return` in Monad:

```
pure :: Applicative m => a -> m a
```

It 'lifts' a single value into the applicative/monadic container. So for us, that's the same thing as `always`.

The more interesting function is the operator `<*>` (sometimes pronounced "splat", or just "apply"). Its type:

```
(<*>) :: Applicative m => m (a -> b) -> m a -> m b
```

So if you ignore the `m` containers, it takes a function `a -> b` and then an argument `a`, and returns `b`. So it's really just applying a function! The only trick is that everything is done *within* the applicative/monadic container, `m`.

Here's our implementation for the probability distribution:

```haskell
instance Applicative Prob where
  pure = always
  Prob fs <*> Prob xs =
    Prob [(f x, p*q) |(f,p) <- fs, (x,q) <- xs]
```

Basically, we have a probability distribution over *functions* (`fs`), and a probability distribution over *values* (`xs`). We use a list comprehension to select all pairs of functions and values. We apply them `f x` but also take into account their probabilities. The function `f` appears with probability `p`; the value `x` appears with probability `q`. So the result of `f x` has probability `p*q`.

We almost always combine the splat with an fmap, like this:

```
ghci> (,) <$> coinFlip <*> coinFlip
Prob {toList = [((True,True),1 % 4),((True,False),1 % 4),
                ((False,True),1 % 4),((False,False),1 % 4)]}
```

We flipped two coins, paired the results `(,)`, and we have the probability of each result! Just add more splats (and commas) for more coins:

```
ghci> (,,) <$> coinFlip <*> coinFlip <*> coinFlip
Prob {toList = [((True,True,True),1 % 8),((True,True,False),1 % 8),
((True,False,True),1 % 8),((True,False,False),1 % 8),
((False,True,True),1 % 8),((False,True,False),1 % 8),
((False,False,True),1 % 8),((False,False,False),1 % 8)]}
```

Or instead of tupling the results, we can apply some other operation using the fmap. How about flip two coins, what's the probability that *both* are heads? What about add the results of two dice?

```
ghci> optimize $ (&&) <$> coinFlip <*> coinFlip
Prob {toList = [(False,3 % 4),(True,1 % 4)]}
ghci> optimize $ (+) <$> diceRoll <*> diceRoll
Prob {toList = [(2,1 % 36),(3,1 % 18),(4,1 % 12),(5,1 % 9),
(6,5 % 36),(7,1 % 6),(8,5 % 36),(9,1 % 9),(10,1 % 12),
(11,1 % 18),(12,1 % 36)]}
```

You can also pair up dice rolls with coin flips:

```
ghci> (,) <$> diceRoll <*> coinFlip
Prob {toList = [((1,True),1 % 12),((1,False),1 % 12),((2,True),1 % 12),
((2,False),1 % 12),((3,True),1 % 12),((3,False),1 % 12),
((4,True),1 % 12),((4,False),1 % 12),((5,True),1 % 12),
((5,False),1 % 12),((6,True),1 % 12),((6,False),1 % 12)]}
```

Or generate lists. There's a function in `Control.Monad` called `replicateM`:

```
replicateM :: Applicative m => Int -> m a -> m [a]
```

It runs the monadic operation the specified number of times, and returns a list of results. So how about the sum of 10 dice?

```
ghci> optimize $ sum <$> replicateM 4 diceRoll
Prob {toList = [(4,1 % 1296),(5,1 % 324),(6,5 % 648),
(7,5 % 324),(8,35 % 1296),(9,7 % 162),(10,5 % 81),
(11,13 % 162),(12,125 % 1296),(13,35 % 324),(14,73 % 648),
(15,35 % 324),(16,125 % 1296),(17,13 % 162),(18,5 % 81),
(19,7 % 162),(20,35 % 1296),(21,5 % 324),(22,5 % 648),
(23,1 % 324),(24,1 % 1296)]}
```

The one thing we can't do with applicative is let a subsequent action depend on the result of a previous action. For example, we can't roll dice and then flip *that many* coins. To do that, we need the monad bind.

### Monad

Reminder: monadic bind on probabilities should have type:

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

Here's my implementation, which relies on `concatMap`. That's not surprising, because `concatMap` is the bind for the list monad.

```
instance Monad Prob where
  Prob as >>= f =
    Prob (concatMap each as)
    where
      each (a,p) = scale p (f a)
      scale p (Prob bs) = map (\(b,q) -> (b, p*q)) bs
```

Now here's the problem we wanted to solve: roll dice, then flip that many coins. And let's say, count how many heads you get.

```
countHeads :: [Bool] -> Int
countHeads = length . filter id

diceThenCoins :: Prob [Bool]
diceThenCoins = optimize $ do
  n <- diceRoll
  replicateM n coinFlip
```

```
ghci> optimize $ countHeads <$> diceThenCoins
Prob {toList = [(0,21 % 128),(1,5 % 16),(2,33 % 128),(3,1 % 6),
(4,29 % 384),(5,1 % 48),(6,1 % 384)]}
```

Or how often is *every* flip heads?

```
ghci> optimize $ Prelude.all id <$> diceThenCoins
Prob {toList = [(False,107 % 128),(True,21 % 128)]}
```

Fun!

```
main :: IO ()
main = return ()
```