A first look at Haskell functions

Functions are fundamental to Haskell, and it provides us with many ways to define them. First, we must look at how to define their types.

The basic form of a function type definition:

```
f :: a -> b
```

which says that `f` is a function that takes objects of type `a` to objects of type `b`. For example, we might see

```
f :: Int -> (Int, String)
```

which means, of course, that `f` takes an integer variable and returns an ordered pair whose first entry is an integer and whose second entry is a string. Now is a good time to list some of the built-in types that Haskell has:

`Int`	integer in limited range, approximately -2 billion to 2 billion
`Integer`	integer of unlimited size
`Bool`	a boolean value, either `True` or `False`
`Float`	floating-point number
`Char`	a single character
`String`	a list of `Char` values, i. e. `String` is `[Char]`

Now is also a good time to mention that types begin with capital letters, but functions and most other structures begin with small letters.

A more mysterious-looking type definition is something such as

```
exOr ::  Bool -> Bool -> Bool
```

This literally means

```
exOr ::  Bool -> (Bool -> Bool)
```

This uses the exponential law that says that functions of two variables $X \times Y \to Z$ correspond to functions $X \to \text{Functions}(Y, Z)$, by thinking of a two-variable function f as a function that takes x to the function that sends y to $f(x, y)$.

In some sense, *all functions in Haskell are functions of one variable, which may take values that are functions.* Thus a type

```
f :: a -> b -> c -> d
```

means `f :: a -> (b -> (c -> d))`, a function that takes an element of type `a` to a function which takes an element of type `b` to a function of type `c -> d`.

Let's explore this further by examining some of the ways that we and define `exOr`, the "exclusive or" function. One way to complete the definition started above is:

```
exOr ::  Bool -> Bool -> Bool
exOr x y = ( x || y ) && not ( x && y )
```

Here, `||`, `&&`, and `not` are the Haskell notations for the logical "or", the logical "and", and the logical "not". The effect of this definition is that Haskell will evaluate an expresssion such as `exOr True False` as `( True || False ) && not ( True && False )`, eventually simplifying to the value `True`.

Since the type `Bool -> Bool -> Bool` really means `Bool -> (Bool -> Bool)`, we can think of `exOr` as a function that takes a `Bool` value and gives us a function of type `Bool -> Bool`. That is, each of the two values `exOr True` and `exOr False` is a function of type `Bool -> Bool`, so we can define `exOr` by giving the two *functions* `exOr True` and `exOr False`.

```
exOr ::  Bool -> Bool -> Bool
exOr True x = not x
exOr False x = x
```

Haskell provides several different syntactic styles that can improve readability. For example, there is an `if ... then` form:

```
exOr ::  Bool -> Bool -> Bool
exOr x y = if x == True then not y else y
```

Note that logical equality is given by double equals signs "==".

A very nice format that I tend to use whenever possible is the use of "guards". This basically allows you to give case-by-case definitions. We can define `exOr` using guards by

```
exOr ::  Bool -> Bool -> Bool
exOr x y
  | x == True  = not y
  | otherwise  = y
```

You can list any number of cases. It is necessary to indent the vertical bars at least one space (two spaces seems to be a fairly common indentation for Haskell). Unlike many languages, Haskell is sensitive to the use of blank space in its layout. This allows it to parse without requiring lots of braces and semicolons to demarcate the statements.

A very useful syntactic device is the "lambda" notation for functions. This is essentially the "mapsto" symbol $\mapsto$, and allows us to define a function without giving it a name. The general format is `\x -> f(x)`. Thus we write `exOr` without naming it as

```
\x y -> ( x || y ) && not ( x && y )
```

And we can define `exOr` purely at the functional level by writing

```
exOr ::  Bool -> Bool -> Bool
exOr = \x y -> ( x || y ) && not ( x && y )
```

A very convenient and readable syntax for defining complicated functions is the `where` construction. For example, suppose you want a function that takes two integers and returns their maximum, along with the number of times it appears, as an ordered pair. It would be convenient to define functions that take the maximum (of course, Haskell has a built-in maximum function `max`, but pretend that it did not) and count its number of occurrences. If we do not need this for anything else, it is simpler just to make these "local" definitions. The `where` construction enables us to do this as follows:

```
maxCount ::  Int -> Int -> (Int,Int)
maxCount a b = (maxTwo a b, num a b)
  where
  maxTwo a b
    |a > b     = a
    |otherwise = b
  num a b
    |a == b    = 2
    |otherwise = 1
```

The `where` must be indented, and everything within the `where` definitions must be indented at least that amount. Notice that when defining the auxiliary functions `maxTwo` and `num` within the `where` structure, we do not need to give their types. These definitions will be local to the `where` region, so will not be defined anywhere else in your program.