Lambdas

Lambdas are essentially *nameless functions*. They can be confusing to look at initially, but are quite useful once you get the hang of them.

Recall Ruby's blocks:

```
(1..10).each { |x|
    puts x + 1
}
```

You can think of the block `{ |x| puts x + 1 }` as an anonymous function of sorts – or, perhaps slightly more accurately, as an anonymous `Proc`. It's a syntactic structure that accepts some number of inputs and operates on those inputs, but doesn't have a name associated with it.

Lambda's are kind of similar to blocks, or at least as similar as anything in Haskell can be to something in Ruby. Consider the following line:

```
\x -> x + 1
```

This is a simple example of a lambda function – the result of that line is a function itself, but it doesn't have a name. We can get the type of that thing in `ghci`:

```
> :t (\x -> x + 1)
(\x -> x + 1) :: Num a => a -> a
```

Even though the function doesn't have a name, the **arguments** are named (in this case, we just have one argument: `x`. The `\` says that we're about to define a lambda function, then we provide a space-separated list of arguments, then a `->` followed by the function body.

We could use the lambda function by (you can run this in `ghci`):

```
> (\x -> x + 1) 5
6
```

So we create a function with a single argument, then we provide the argument `5` to that function.

If we wanted multiple arguments:

```
(\x y -> x + y + 1)
```

And we could call it by:

```
> (\x y -> x + y + 1) 3 4
8
```

What's the point?

The previous examples were fairly trivial/contrived. However, consider the following function definition:

```
zipWith' :: (a -> b -> c) -> [a] -> [b] -> [c]
zipWith' f xs ys = map f' $ zip xs ys
  where
    f' (x, y) = f x y
```

Notice that we had to use a `where` clause to define the `f'` function. However, we're only using this function once, and it's relatively simple, so why give it a name?

Let's redefine `zipWith'` to use an anonymous function:

```
zipWith' :: (a -> b -> c) -> [a] -> [b] -> [c]
zipWith' f xs ys = map (\(x, y) -> f x y) $ zip xs ys
```

Note that we can do pattern matching on the arguments of the lambda function, just like we can with any other function. In this case, we're pattern matching on the tuple constructor.

Function Composition

Here's a weird function:

```
(.) :: (b -> c) -> (a -> b) -> a -> c
(f . g) x = f (g x)
```

The easiest way to think about this function is that it takes in two functions as arguments, and returns a new function which is the composition of the input functions. Following that reasoning, it's informative to rewrite the type signature as:

```
(.) :: (b -> c) -> (a -> b) -> (a -> c)
```

```
minList :: Ord a => [a] -> a
minList xs = head $ sort xs
```

Think of `head $ sort x` as "apply the function `head` to the result of applying the function `sort` to the list `xs`".

We could also write `minList` as:

```
minList :: Ord a => [a] -> a
minList xs = head . sort $ xs
```

Think of `head . sort $ x` as "create a new function `head . sort`, then apply that new function to the list `xs`". `(.)` has *higher priority* than `($)`, so you can think of the order of operations as `(head . sort) $ xs`. `head` is the first argument to the `(.)` function, `sort` is the second, and `xs` is the argument we supply to the resulting function `head . sort`.

Or, equivalently, we could leave the `xs` out altogether:

```
minList :: Ord a => [a] -> a
minList = head . sort
```

This is similar to the previous case, but with a shorter explanation: "`minList` is the result of composing the `head` and `sort` functions".

Note that **all 3 of the above definitions of `minList` are equivalent**. They're simply meant to help you think about how `(.)` works.

A common design pattern in haskell is to combine multiple functions with composition to create a new function:

```
maxList :: Ord a => [a] -> a
maxList = head . reverse . sort
```

`(.)` is right-associative, so `head . reverse . sort` is equivalent to `head . (reverse . sort)` – `reverse . sort` will result in a function with the type `Ord a => [a] -> [a]`, then we compose `head` with that function.

This will sort a list from smallest to largest, reverse the list, then grab the first element.

Typeclasses

Overview

Recall the two types of polymorphism in Haskell: *parametric polymorphism*, and *ad-hoc polymorphism*. Ad-hoc polymorphism is provided by Haskell's typeclasses.

Essentially, typeclasses are a way of providing function/operator-overloading. An example of a typeclass is `Eq`:

```
class Eq a where
  (==) :: a -> a -> Bool
```

Just as in the type signatures for polymorphic functions that we've seen, we have a type variable here: `a`. You can read this typeclass definition as: "In order for an arbitrary type `a` to satisfy the `Eq` typeclass, there must exist a function `(==)` for type `a` with the type signature `(==) :: a -> a -> Bool`."

So far, we've only shown how to define the typeclass itself. Now we can define *instances* of that typeclass. For example, if we want `Bool` to satisfy the `Eq` typeclass, we would write:

```
instance Eq Bool where
  True  == True  = True
  False == False = True
  _     == _     = False
```

As an aside, recall that (==) is an infix function, which lets us define it as we did above. So the following two lines are equivalent:

```
True == True = True
-- is the same as
(==) True True = True
```

Now that we've defined the typeclass instance `Eq Bool`, we can use a `Bool` as an argument to any function that imposes the `Eq` *class constraint* on the relevant type variable:

```
listElem :: Eq a => a -> [a] -> Bool
listElem x xs = x `elem` xs
```

Consider the following call to this function:

```
result = listElem True [False, False, True]
```

Looking back at the type signature for `listElem`, we see that the type variable `a` is `Bool` in this specific function call (imagine replacing all of the `a`'s in the type signature of `listElem` with `Bool`). That means the type constraint says that the typeclass instance `Eq Bool` *must* be defined (which we did previously).

Hierarchical Typeclasses

We can also use existing typeclass instances to define new ones. For example, given the `Eq` typeclass above, we might want to say something like "If you have a tuple of values whose types are instances of the `Eq` typeclass, then you can compare the tuples themselves as well." You can do this with:

```
instance (Eq a, Eq b) => Eq (a, b) where
  (x1, y1) == (x2, y2) = x1 == x2 && y1 == y2
```

The `(Eq a, Eq b)` part is another set of class constraints and says that the types `a` and `b` must both be instances of `Eq` in order for this typeclass instance `Eq (a, b)` to work. If you look at the second line, `(x1, y1) == (x2, y2) = x1 == x2 && y1 == y2`, you should see why this is the case: basically, in order to define `(==)` for the tuples, we must be able to call the `(==)` function on the elements of the tuples.

We can also add a class constraint to a typeclass *definition*:

```
class Eq a => Ord a where
  (<), (<=), (>=), (>) :: a -> a -> Bool
  max, min             :: a -> a -> a
```

The `Eq a => Ord a` part says "in order for some type `a` to be an instance of the `Ord` type class, it must also be an instance of the `Eq` typeclass".

`Deriving`

Consider the following data declaration:

```
data Person = Person Name Age
type Name = String
type Age = Int
```

If we loaded this code into `ghci`, created a `Person`, then tried to print it:

```
> bob = Person "Bob" 27
> print bob

<interactive>:6:1: error:
    • No instance for (Show Person) arising from a use of ‘print’
    • In the expression: print person
      In an equation for ‘it’: it = print person
```

The issue here is that `Person` needs to be an instance of the `Show` typeclass so that the `print` knows how to convert a `Person` to a `String`.

We could do this manually:

```
instance Show Person where
  show (Person name age) = "Person " ++ show name ++ " " ++ show age
```

And now we won't get an error:

```
> print person
Person Bob 27
```

Both fields of `Person` (`Name` and `Age`) are both already instances of `Show`. It would be nice if the compiler could infer that we want to simply print out the name of the constructor and each of its values, without having to full define the `Show Person` instance ourselves.

In fact, we can do just that by adding to our data declaration:

```
data Person = Person Name Age
  deriving Show
```

```haskell
type Name = String
type Age = Int
```

The `deriving` keyword tells the compiler to define a default instance of a typeclass. In this case, we're using `deriving Show` to tell the compiler that we want it to automatically create a `Show Person` instance for us. The result will be exactly the same as the instance we defined ourselves previously:

```
> person = Person "Bob" 27
> print person
Person Bob 27
```

We can derive a few other types as well, like `Eq`:

```haskell
data Person = Person Name Age
  deriving (Eq, Show)
```

```
> youngBob = Person "Bob" 27
> oldBob = Person "Bob" 60
> youngBob == oldBob
False
> oldBob == oldBob
True
```

The automatically created `(==)` for the `Person` type simply compares each of the fields, `Name` and `Age`.