Haskell: Higher Order Functions (Part I)

Principles of Programming Languages
Colorado School of Mines

https://lambda.mines.edu
Haskell lets us pass functions as the arguments to other functions:

\[
\text{double} :: (\text{Num } a) \Rightarrow a \to a
\]
\[
\text{double } x = x \times 2
\]

\[
\text{doubleAll} :: (\text{Num } a) \Rightarrow [a] \to [a]
\]
\[
\text{doubleAll } xs = \text{map} \text{ double } xs
\]

We see Haskell treats functions as a first-class citizen; that is, we can pass them around just like any other type of data.
Review: Functions are Data

Haskell lets us pass functions as the arguments to other functions:

\[
double :: (\text{Num } a) \Rightarrow a \rightarrow a
\]
\[
double \ x = x \times 2
\]

\[
doubleAll :: (\text{Num } a) \Rightarrow [a] \rightarrow [a]
\]
\[
doubleAll \ xs = \text{map} \ double \ xs
\]

And we can define functions which take functions:

\[
\text{map'} :: (a \rightarrow b) \rightarrow [a] \rightarrow [b]
\]
\[
\text{map'} \ _ \ [] = []
\]
\[
\text{map'} \ f \ (x:xs) = f \ x : \text{map} \ f \ xs
\]

We see Haskell treats functions as a \textit{first-class citizen}; that is, we can pass them around just like any other type of data.
Review: Functions are Data

Haskell lets us pass functions as the arguments to other functions:

\[
\text{double} :: (\text{Num } a) \Rightarrow a \to a \\
\text{double } x = x \times 2
\]

\[
\text{doubleAll} :: (\text{Num } a) \Rightarrow [a] \to [a] \\
\text{doubleAll } xs = \text{map} \ \text{double} \ \text{xs}
\]

And we can define functions which take functions:

\[
\text{map'} :: (a \to b) \Rightarrow [a] \to [b] \\
\text{map'} \ _ \ [] = [] \\
\text{map'} \ f \ (x:xs) = f \ x : \text{map} \ f \ xs
\]

We see Haskell treats functions as a first-class citizen; that is, we can pass them around just like any other type of data.
Haskell takes advantage of **currying** to support functions with multiple arguments. That is, functions take a single argument and return a function ready to take the next argument.

We call the function ready to take the next argument a **partially applied function**.

```haskell
-- define a function which subtracts
subtract45 = subtractMinutes 45
```
Haskell takes advantage of \textit{currying} to support functions with multiple arguments. That is, functions take a single argument and return a function ready to take the next argument.

We call the function ready to take the next argument a \textit{partially applied function}.

\texttt{subtractMinutes :: Int -> Int -> Int}
\texttt{subtractMinutes n x = (x - n) `mod` 60}

-- define a function which subtracts
-- 45 minutes every time
-- subtract45 30 --> 45
\texttt{subtract45 = subtractMinutes 45}
Partially Applied Prefix Functions

\[
\text{multiplyBy} :: (\text{Num } a) \Rightarrow a \rightarrow a \rightarrow a
\]
\[
\text{multiplyBy } x y = x \times y
\]

-- define our doubleAll using a partially applied prefix function (and currying!)
\[
doubleAll = \text{map } (\text{multiplyBy } 2)
\]
Write a partially complete infix function in parentheses to create a partially applied infix function.

-- define our doubleAll using a partially applied infix function (and currying!)
doubleAll = map (2 *)

-- also valid
doubleAll = map (* 2)
zipWith is a really useful function in Haskell’s standard library. It takes a function that takes two arguments, and applies it to two each of the elements from two lists. For example:

```
GHCi> zipWith (+) [1,2,3] [10,20,30]
[11,22,33]
GHCi> zipWith max [1..5] (reverse [1..5])
[5,4,3,4,5]
```
Let's Implement zipWith

\[
\text{zipWith'} :: (a \to b \to c) \to [a] \to [b] \to [c]
\]

\[
\text{zipWith'} _ [] _ = []
\]

\[
\text{zipWith'} _ _ [] = []
\]

\[
\text{zipWith'} f (x:xs) (y:ys) = f x y : \text{zipWith'} f xs ys
\]
Anonymous Functions (Lambdas)

Haskell provides a notation to write functions inline without a name:

```
-- twistTuples [(1,2),(3,4)] --> [(2,1),(4,3)]
twistTuples xs = map (\ (a,b) -> (b,a)) xs
```
Anonymous Functions (Lambdas)

Haskell provides a notation to write functions inline without a name:

```
-- twistTuples [(1,2),(3,4)] --> [(2,1),(4,3)]
twistTuples xs = map (\ (a,b) -> (b,a)) xs
```

Why do we have lambdas? Perhaps there is a case where writing a lambda might be cleaner than another function, a `let` or `where` binding, or partial application.
Perhaps a lambda can make it more clear we are returning another function. Consider the flip function (in Haskell’s standard library) which takes a function and returns a new one with the arguments flipped:

\[
\text{flip'} :: (a \rightarrow b \rightarrow c) \rightarrow b \rightarrow a \rightarrow c \\
\text{flip'} f \, x \, y = f \, y \, x
\]
Perhaps a lambda can make it more clear we are returning another function. Consider the flip function (in Haskell’s standard library) which takes a function and returns a new one with the arguments flipped:

\[
\text{flip'} :: (a \rightarrow b \rightarrow c) \rightarrow b \rightarrow a \rightarrow c
\]
\[
\text{flip'} f \ x \ y = f \ y \ x
\]

Is it immediately obvious this function is supposed to return another (partially applied) function? Compare to this definition:

\[
\text{flip'} :: (a \rightarrow b \rightarrow c) \rightarrow b \rightarrow a \rightarrow c
\]
\[
\text{flip'} f = \ lambda \ x \ y . f \ y \ x
\]
Quiz Prep Time

With your learning groups, everyone take turns taking your quizzes you designed. Once finished, we will start Quiz 2.

More on higher order functions next time.