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:

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

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

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

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

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

And we can define functions which take functions:

\[
\text{map'} :: (a \rightarrow b) \rightarrow [a] \rightarrow [b] \\
\text{map'} _[:] = [] \\
\text{map'} f (x:xs) = f x : \text{map } f \text{ xs}
\]
Haskell lets us pass functions as the arguments to other functions:

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

\[
\text{doubleAll :: (Num a) \Rightarrow [a] \rightarrow [a]}\n\]
\[
doubleAll xs = \text{map double} \hspace{1mm} xs\n\]

And we can define functions which take functions:

\[
\text{map' :: (a \rightarrow b) \rightarrow [a] \rightarrow [b]}\n\]
\[
\text{map'} \hspace{1mm} [] = []\n\]
\[
\text{map'} \hspace{1mm} f \hspace{1mm} (x:xs) = f\hspace{1mm}x\hspace{1mm}:\hspace{1mm}\text{map}\hspace{1mm}f\hspace{1mm}xs\n\]

We see Haskell treats functions as a \textbf{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 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
subtractMinutes :: Int -> Int -> Int
subtractMinutes n x = (x - n) `mod` 60
```

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

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

-- define our doubleAll using a partially applied
-- prefix function (and currying!)
\[
\text{doubleAll = map (multiplyBy 2)}
\]
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'

zipWith' :: (a -> b -> c) -> [a] -> [b] -> [c]
zipWith' _ [] _ = []
zipWith' _ _ [] = []
zipWith' f (x:xs) (y:ys) = f x y : 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) \to (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.
Maybe Lambda Makes This Cleaner

Perhaps a lambda can make it more clear we are returning another function. Consider the \texttt{flip} function (in Haskell's standard library) which takes a function and returns a new one with the arguments flipped:

\[
\texttt{flip'} :: (a \to b \to c) \to b \to a \to c
\]
\[
\texttt{flip'} f \; x \; y = f \; y \; x
\]
Maybe Lambda Makes This Cleaner

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:

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

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

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