Lesson 4: Higher Order Functions

What's It All About?

1. Lambda in Haskell
2. Mystery1
3. Mystery2
4. Implementations of MAP
5. Implementations of FILTER
6. Zip With (map2)
7. Zip
8. Folding Functions
9. Left Folding
10. Right Folding

Lambda in Haskell

Demo

In the following demo, a nameless function to square a given number is applied just one time. Then a nameless function to determine whether or no a string is shorter than five characters in length is applied twice. Finally, three variants on a function to square a first number and add a second number to it are considered. The first of the two nameless variants is explicitly curried, while the other two are implicitly curried, as is the nature of functions in Haskell.

```
>>> ( \x -> x*x ) 5
25

>>> ( \x -> ( length x ) < 5 ) "red"
True
>>> ( \x -> ( length x ) < 5 ) "purple"
False

>>> ( \x -> \y -> x ^ 2 + y ) 4 3
19
>>> ( \ x y -> x ^ 2 + y ) 3 4
19
>>> fun x y = x ^ 2 + y
>>> fun 4 3
19
```

```
>>> :t ( \x -> \y -> x ^ 2 + y )
( \x -> \y -> x ^ 2 + y ) :: Num a => a -> a -> a
>>> :t ( \ x y -> x ^ 2 + y )
( \ x y -> x ^ 2 + y ) :: Num a => a -> a -> a
>>> :t fun
fun :: Num a => a -> a -> a
>>>
```

Mystery1

Partitioned Demo

```
>>> mystery1 f xs = [ f x | x <- xs ]

>>> mystery1 succ [1,2,3,4,5]
[2,3,4,5,6]
>>> mystery1 ( \x -> x*x ) [1,2,3,4,5]
[1,4,9,16,25]
>>> mystery1 length ["red","white","blue"]
[3,5,4]

>>> map succ [1,2,3,4,5]
[2,3,4,5,6]
>>> map ( \x -> x*x ) [1,2,3,4,5]
[1,4,9,16,25]
>>> map length ["red","white","blue"]
[3,5,4]
>>>
```

Mystery2

Partitioned Demo

```
>>> mystery2 p xs = [ x | x <- xs, p x]

>>> mystery2 even [1..10]
[2,4,6,8,10]
>>> mystery2 ( \x -> length x < 5 ) ["red","yellow","blue"]
["red","blue"]

>>> filter even [1..10]
```

```
[2,4,6,8,10]
>>> filter ( \x -> length x < 5 ) ["red","yellow","blue"]
["red","blue"]
>>>
```

Implementations of MAP

Two implementations of the `map` function from the standard Prelude are presented. The first features list comprehensions. The second features recursion.

Code

```
map' f xs = [ f x | x <- xs ]

map'' f [] = []
map'' f (x:xs) = f x : map'' f xs
```

Demo

```
>>> map (+10) [1..10]
[11,12,13,14,15,16,17,18,19,20]
>>> map' (+10) [1..10]
[11,12,13,14,15,16,17,18,19,20]
>>> map'' (+10) [1..10]
[11,12,13,14,15,16,17,18,19,20]

>>> :type map
map :: (a -> b) -> [a] -> [b]
>>>
```

Implementations of FILTER

Two implementations of the `filter` function from the standard Prelude are presented. The first features list comprehensions. The second features recursion.

Code

```
filter' f xs = [ x | x <- xs, f x]
```

```
filter'' f [] = []
filter'' f (x:xs) = if ( f x ) then ( x : y ) else y
                    where y = filter'' f xs
```

Demo

```
>>> filter even [1..10]
[2,4,6,8,10]
>>> filter ( \x -> x /= ' ' ) "red white blue"
"redwhiteblue"
>>> filter' even [1..10]
[2,4,6,8,10]
>>> filter' ( \x -> x /= ' ' ) "red white blue"
"redwhiteblue"
>>> filter'' even [1..10]
[2,4,6,8,10]
>>> filter'' ( \x -> x /= ' ' ) "red white blue"
"redwhiteblue"

>>> :type filter
filter :: (a -> Bool) -> [a] -> [a]
>>>
```

Zip With

The function `map` operates according to the following specification:

- first argument: a function of one argument (:: a -> b)
- second argument: a list (:: [a])
- result: a list (:: [b]) obtained by gathering the results of applying the first argument to each element of the second argument

Sometimes it is useful to have an analogous function, say `map2`, which operates according to the following specification:

- first argument: a function of two argument (:: a -> b -> c)
- second argument: a list (:: [a])
- third argument: a list (:: [b])
- result: a list (:: [c]) obtained by gathering the results of applying the first argument to each successive pair of elements found in the second and third arguments

Here is a recursive implementation of `map2`:

```
map2 f [] [] = []
map2 f (x:xs) (y:ys) = f x y : map2 f xs ys
```

It turns out that there a function defined in the standard prelude called `zipWith` that operates just like the `map2` function.

Demo

```
>>> zipWith (*) [2,3,5,7] [2,3,5,7]
[4,9,25,49]
>>> zipWith max [1,3,5,7] [8,6,4,2]
[8,6,5,7]
>>> zipWith (^) [1..10] [2,2..]
[1,4,9,16,25,36,49,64,81,100]
>>> zipWith (\x y -> (x,y)) [1,2,3,4,5,6,7] [7,6,5,4,3,2,1]
[(1,7),(2,6),(3,5),(4,4),(5,3),(6,2),(7,1)]
>>> zipWith (\x y -> (x,y)) "blue" "gold"
[('b','g'),('l','o'),('u','l'),('e','d')]
>>>
```

Zip

It turns out that the idea inherent in the last two example of the demo is worth encapsulating in a function. We might call this function `zip'`, and define it like this:

```
zip' xs ys = zipWith (\x y -> (x,y)) xs ys
```

But there is a function in the standard prelude called `zip` that does just this, so you might as well use that when you need to zip a couple of lists.

Demo

```
>>> zip [1,2,3,4,5,6,7] [7,6,5,4,3,2,1]
[(1,7),(2,6),(3,5),(4,4),(5,3),(6,2),(7,1)]
>>> zip "blue" "gold"
[('b','g'),('l','o'),('u','l'),('e','d')]
>>>
```

Folds

Functions called **folds** allow you to reduce a list to a single value. In view of this, the word “reduce” is often used to describe the behavior of a fold function.

Whenever you want to traverse a list and return a value that depends on each element of the list, chances are that you can frame your problem as a fold.

Specifically, a fold takes a **binary function**, a **starting value**, and a **list** to fold up.

There are two basic varieties of folding. A **left fold** starts the folding from the left side of the list, with the help of the initial value, and works element by element to the right side of the list. A **right fold** starts the folding from the right side of the list, with the help of the initial value, and works element by element to the left side of the list. The featured left fold function in Haskell is called `foldl`. The featured right fold function in Haskell is called `foldr`.

Foldl

The following lines present a couple of informal examples of `foldl` in action. In each case, you can see how, after conceptualization, the initial element is folded with the first element, how the result of the first fold is folded with the second element, and so on. In order to render the string orented example sensible, imaging that a definition called `a2` is in effect to append just two lists (`a2 x y = x ++ y`).

```
foldl (+) 0 [1,2,3,4]
==> conceptualization
( ( ( ( 0 + 1 ) + 2 ) + 3 ) + 4 )
==> definition of addition
( ( ( 1 + 2 ) + 3 ) + 4 )
==> definition of addition
( ( ( 3 + 3 ) + 4 )
==> definition of addition
( ( 6 + 4 )
==> definition of addition
10

foldl a2 "" ["a","bb","ccc"]
==> conceptualization
( ( ( "" a2 "a" ) a2 "bb" ) a2 "ccc" )
==> definition of a2
( ( "a" a2 "bb" ) a2 "ccc" )
==> definition of a2
( "abb" a2 "ccc" )
==> definition of a2
"abbccc"
```

For the record, here is what a recursive definition of `foldl` might look like:

```
foldleft f v [] = v
foldleft f v (x:xs) = foldleft f ( f v x ) xs
```

And here is a demo involving `foldl`, and a related function called `scanl`. This latter function can be used to see the partial fold results going from the left of the list to the right of the list.

```
>>> foldl (+) 0 [1,2,3,4]
10
>>> scanl (+) 0 [1,2,3,4]
[0,1,3,6,10]
>>> foldl a2 "" ["a","bb","ccc"]
```

```
"abbccc"
>>> scanl a2 "" ["a","bb","ccc"]
["","a","abb","abbccc"]
>>>
```

Foldr

The following lines present a couple of informal examples of `foldr` in action. In each case, you can see how, after conceptualization, the last element is folded with the initial element, after which the second to the last element is folded with the result of the previous fold, and so on.

```
foldr (+) 0 [1,2,3,4]
==> conceptualization
( 1 + ( 2 + ( 3 + ( 4 + 0 ) ) ) )
==> definition of addition
( 1 + ( 2 + ( 3 + 4 ) ) )
==> definition of addition
( 1 + ( 2 + 7 ) )
==> definition of addition
( 1 + 9 )
==> definition of addition
10

foldr a2 "" ["a","bb","ccc"]
==> conceptualization
( "a" a2 ( "bb" a2 ( "ccc" a2 "" ) ) )
==> definition of a2
( "a" a2 ( "bb" a2 "ccc" ) )
==> definition of a2
( "a" a2 "bbccc" )
==> definition of a2
"abbccc"
```

For the record, here is what a recursive definition of `foldr` might look like:

```
foldright f v [] = v
foldright f v (x:xs) = f x ( foldright f v xs )
```

And here is a demo involving `foldr`, and a related function called `scanr`. This latter function can be used to see the partial fold results going from the right of the list to the left of the list.

```
>>> foldr (+) 0 [1,2,3,4]
10
>>> scanr (+) 0 [1,2,3,4]
[10,9,7,4,0]
>>> foldr a2 "" ["a","bb","ccc"]
"abbccc"
>>> scanr a2 "" ["a","bb","ccc"]
["abbccc","bbccc","ccc",""]
```

>>>