Tutorial 3

James and Aaron

1 Lambda and Currying

```haskell
max1 a b = if a < b then b else a

max2 :: Ord a => a -> a -> a
max2 = \a b -> if a < b then b else a

max3 :: Ord a => a -> a -> a
max3 = \a -> \b -> if a < b then b else a
```

All the above are the same.

```
(max1 2 4) == ((max1 2) 4)

mymax = max1 2
mymax 4
```

2 List comprehensions

Lists can be specified by specifying ranges (`[start..end]`and `[start..]`). The range has a step of 1 and includes both the start and end values. When no end value is specified it is infinite.

```
*Main> [0..10]
[0,1,2,3,4,5,6,7,8,9,10]
```

We can specify the size of the step by giving the first value, the next value, and then optionally the end value like `[first,next..end]`or `[first,next..]`. In these cases the step will be $next - first$.

For example:

```
*Main> [0, 2..20]
```

```
[0,2,4,6,8,10,12,14,16,18,20]

*Main> [10,9..0]
[10,9,8,7,6,5,4,3,2,1,0]
```

Lists can also be generated using list comprehensions in the form `[value|generators,filters]`. For example:

```
*Main> [x | x <- [1..10], x < 5]
[1,2,3,4]
*Main> [(x,y) | x <- [1..10], y <- [1..10], x < 5, y < 5]
[(1,1),(1,2),(1,3),(1,4),(2,1),(2,2),(2,3),(2,4),(3,1),(3,2),(3,3),(3,4),(4,1),(4,2),(4,
```

Note using an infinite list as a generator will cause the list to be infinite even if it would be logically bound by one of the conditions.

3 List functions

3.a Filter

The filter function is used to filter out items in a list, by specifying which items should remain in the list.

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

To get a list of only values greater than zero, we can write:

```
*Main> filter (>0) [-2, 100, -30, 4.4, 0, 0.1]
[100.0,4.4,0.1]
```

To get a list of palindromes:

```
filter (\x -> x == reverse x) ["string","racecar","tacocat", "palindrome"]
["racecar","tacocat"]
```

3.b Map

The map function takes a function and maps a given list using that function to a new list.

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

Some examples:

```
*Main> map length ["hi", "hello", "tacocat"]
[2,5,7]

*Main> map (>0) [1,3,4,-4,5,2,0]
[True,True,True,False,True,True,False]
```

Note there is another function `and`that folds (explained later) lists using logical and.

```
*Main> :type and
and :: Foldable t => t Bool -> Bool
```

Those two functions can be used to see whether a property holds for the entire list.

```
*Main> and (map (>0) [1,3,4,-4,5,2,0])
False
```

Note this can be more efficently done using a single fold (covered further down).

You can also map lists to lists of functions, for example:

```
*Main> map (max3) [1,3,4,4,5,2,1]
```

Note this will throw an error in the console as functions cannot be printed (No show function), however you can still use it within expressions.

```
*Main> (map (max3) [1,3,4,4,5,2,1]) 5
5
```

3.c Fold

Fold functions are used to combine the items of a list (or any foldable type) into a single value. You can start the fold from the left or right side.

```
*Main> :type foldl
foldl :: Foldable t => (b -> a -> b) -> b -> t a -> b
*Main> :type foldr
foldr :: Foldable t => (a -> b -> b) -> b -> t a -> b
```

Note when the function used to combine the items is not symmetric the direction of the fold matters.

```
foldl (-) 0 [1,2,3]
((0 - 1) - 2) - 3
-6

foldr (-) 0 [1,2,3]
1 - (2 - (3 - 0))
```

```
1 - (2 - 3)
1 - (-1)
2
```

3.d Zip

The zip function combines two lists into a list of pairs.

```
*Main> :type zip
zip :: [a] -> [b] -> [(a, b)]
```

Here is an example.

```
*Main> zip ["James", "Greenwood", "-", "Thessman"] [3, -5, 20, 2]
[("James",3),("Greenwood",-5),("-",20),("Thessman",2)]
```

When the two lists are not the same length, the returned list will be length of the shortest list.

Another function `zipWith`allows for a function to be specified for how the each pair is combined.

```
*Main> :type zipWith
zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]

*Main> zipWith max [1,5,0] [4,2,4]
[4,5,4]
*Main> zipWith (\x y -> (x,y)) [1,5,0] [4,2,4]
[(1,4),(5,2),(0,4)]
```

Zip can implemented as the following.

```
zip1 x [] = []
zip1 [] y = []
zip1 (x:xs) (y:ys) = (x,y):zip1 xs ys
```

3.e Take

The take function returns a list of the first n items in the given list. This is useful for dealing with infinite lists.

```
*Main> :type take
take :: Int -> [a] -> [a]
```

For example:

```
*Main> take 3 [1,2,3,4,5]
[1,2,3]
*Main> take 3 [0..]
```

```
[0,1,2]
```

3.f Drop

The drop function drops the first n items in the given list.

```
*Main> :type drop
drop :: Int -> [a] -> [a]
```

For example:

```
*Main> drop 5 [1,2,3,4,5,6,7,8,9]
[6,7,8,9]
*Main> drop 5 [1..20]
[6,7,8,9,10,11,12,13,14,15,16,17,18,19,20]
```

Take and drop often work well together.

```
*Main> take 5 (drop 5 [1..])
[6,7,8,9,10]
```