# Higher order functions - map, fold and filter

Higher order functions are functions which take in other functions as input, or return a function as output. `takeWhile` and `dropWhile` which I covered in the page on Prelude functions are examples of higher order functions. But the three most useful higher order functions are map, fold and filter.

### 1 - map

map is the easiest to understand of the three. It takes in two inputs - a function, and a list. It then applies this function to every element in the list. You can basically do the same thing with a list comprehension however.

`map function list` does exactly the same thing as `[function x | x <- list]`

Why do we use it then? Mainly because map is easier to understand and more compact in writing.

```
Main> map (+1) [1..5]
[2, 3, 4, 5, 6]
Main> map (toLower) "abcDEFG12!@#"
"abcdefg12!@#"
Main> map (`mod` 3) [1..10]
[1, 2, 0, 1, 2, 0, 1, 2, 0, 1]
```

### 2 - fold

fold takes in a function and *folds* it in between the elements of a list. It's a bit hard to understand at first - try this example.

Let's add up the first 5 numbers with fold. The command for this is

`foldl (+) 0 [1..5]`

How does this work? First, take the elements of ths list and write them out separately with a space between each.

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | 1 | | 2 | | 3 | | 4 | | 5 |

And now we *fold* the function into the elements. To do this, we write the function in the empty spaces between the elements.

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | + | 1 | + | 2 | + | 3 | + | 4 | + | 5 |

Note that there's nothing before the first (+). That's because we can specify a starting point. So that's the second input foldl asks for - a starting point. In this case, we said zero, so we'll put it in there.

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| 0 | + | 1 | + | 2 | + | 3 | + | 4 | + | 5 |

And all this gives us the final result

```
Main> foldl (+) 0 [1..5]
15
```

And if we change the starting point, we get the corresponding answer.

```
Main> foldl (+) 1 [1..5]
16
Main> foldl (+) 10 [1..5]
25
```

fold is NOT limited to 'infix' functions! (Infix functions are those where you write the function between the inputs. Examples are 'mod', `div`, + and - ) With a non-infix function you won't write the function between the elements like the example above - you'll write it to the left of a pair of elements, and put lots of brackets everywhere. For example,

```
foldl (max) 0 [3,10,14,6]
```

is the same as

```
(max (max (max (max 0 3) 10) 14) 6)
```

(This is basically how the inbuilt function `maximum` is implemented in Prelude.hs)

We can also go through the list from right to left, rather than left to right. That's why we use `foldl` and `foldr`. In the example with (+) we get exactly the same result since `(0 + 1 + 2 + 3 + 4 + 5)` is exactly the same as `(0 + 5 + 4 + 3 + 2 + 1)` However, there are cases where folding from the left gives us different results to folding from the right. It all depends on what function you're folding. For example,

```
Main> foldr (div) 7 [34,56,12,4,23]
8
Main> foldl (div) 7 [34,56,12,4,23]
0
```

Now, the fact that I've written heaps about fold doesn't mean it's the most important of the three - just the hardest to explain! I actually use fold a lot less than I use map and filter.

**3 - filter**

filter is easy. It takes in a 'test' and a list, and it chucks out any elements of the list which don't satisfy that test.

```
Main> filter (isAlpha) "$#!+abcDEF657"
"abcDEF"
Main> filter (even) [1..10]
[2, 4, 6, 8, 10]
Main> filter (>5) [1..10]
[6, 7, 8, 9, 10]
```

A lot of the time, you'll want to write your own tests. The other day I saw a list comprehension which didn't work.

```
allThings:: [Content] -> [Content]
allThings contents = [ x | x <- contents, x == (Thing _)]
```

This is basically an attempt at filtering with a list comprehension. This didn't work - remember that you can only use the underscore (the _) on the left hand side of a pattern match, and never on the right hand side of = . What you normally do to work around this problem is to write your own test, like so.

```
isThing:: Content -> Bool
        isThing Thing _  = True
isThing _         = False
```

And then you can rewrite `allThings`

```
allThings:: [Content] -> [Content]
allThings contents = filter (isThing) contents
```

Go get some practice!

---

Back to Supp. Notes