Tutorial 3: Map, filter, zip

We will practice the **higher-order** functions `map`, `filter`, `zip`, and `zipWith`. The function `map` takes as its first argument a function `f :: a -> b`, as its second argument a list `xs :: [a]`, and applies `f` to each element in `xs` to return a list of type `[b]`.

```
map :: (a -> b) -> [a] -> [b]
map _ []     = []
map f (x:xs) = f x : map f xs
```

The function `filter` takes a **property** `p :: a -> Bool` (something that is true or false for a value of type `a`) and selects those elements from a list for which the property is true.

```
filter :: (a -> Bool) -> [a] -> [a]
filter _ []     = []
filter p (x:xs)
    | p x       = x : filter p xs
    | otherwise =     filter p xs
```

The function `zip` combines two lists into a list of pairs. If one list is longer than the other, the remaining elements are discarded.

```
zip :: [a] -> [b] -> [(a,b)]
zip _ [] = []
zip [] _ = []
zip (x:xs) (y:ys) = (x,y) : zip xs ys
```

The function `zipWith` is similar, except that instead of forming pairs, it combines the two lists with an arbitrary function.

```
zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]
zipWith f _ [] = []
zipWith f [] _ = []
zipWith f (x:xs) (y:ys) = f x y : zip xs ys
```

Exercise 1: Write two versions of each of the following functions, one using **recursion** and one using **map** and **filter**.

a) A function `doubles` that multiplies every number in a list by two. It may be helpful to create a function `double` to double a single number.

b) A function `multiplesOfThree` that removes any numbers from a list that are not a multiple of three. Use the built-in `mod`, and define an auxiliary function to test divisibility by three.

c) A function `doubleMultiplesOfThree` that takes the multiples of three from a list (removing the others) and then doubles them.

```
ghci> doubles [1..10]
[2,4,6,8,10,12,14,16,18,20]
ghci> multiplesOfThree [1..10]
[3,6,9]
ghci> doubleMultiplesOfThree [1..10]
[6,12,18]
```

Where-clauses

When you use auxiliary functions such as `double` that are only used by one function, and not needed elsewhere in your program, it is good practice to put them in a **where-clause**. For instance, for a function that squares each number in a list:

```
squareAll :: [Int] -> [Int]
squareAll xs = map square xs
    where
        square x = x*x
```

The function `square` is now **local** to the definition of `squareAll`. Other functions cannot use it, nor can you use it when you load this into `ghci`. Firstly, this tidies up your code for better readability. Secondly, if your function uses the same value multiple times, if you put it in a where-clause then you are certain it will be computed only once.

A where-clause is part of the syntax of function definitions. Each pattern-matching case can have its own where-clause. Guards do not have separate where-clauses, but a single one for all guards in a given pattern-matching case.

Exercise 2:

a) Tidy up the functions of the previous exercise by putting any auxiliary functions in where-clauses.

b) Complete the function `shorts` that removes all strings of 4 characters or more from a list. Use `filter`, and a where-clause to define the filtering property.

c) Complete the function `incrementPositives` that takes all positive integers in a list and adds one. Use `map` and `filter`, with a where-clause to define any auxiliary functions you might need.

d) Complete the function `difference` which takes two strings, and removes every character occurring in the second, from the first. Use `filter`, and `elem` and `not` may be helpful.

e) Complete the function `oddLengthSums` that given a list of integer lists, returns for each odd-length list its sum. Use `map` and `filter`, and `odd`, `length`, and `sum` may be helpful.

f) Give the functions above their most general type using type variables and type classes.

```
ghci> shorts ["one","two","three","four","five","six","seven"]
["one","two","six"]
ghci> incrementPositives [-3,4,1,-2,0,3]
[5,2,4]
ghci> difference "difference" "ef"
"dirnc"
ghci> oddLengthSums [[1],[1,2],[1,2,3],[1..4],[1..5]]
[1,6,15]
```

Zip and zipWith

Exercise 3:

a) Complete the function `numbered` that indexes the elements in a list by pairing them with numbers, counting up from 1. Use the function `zip`.

b) Complete the function `everyother` that takes every other element from a list, starting with the first. Use `numbered` to count elements, `filter`, `fst`, and `odd` to select those pairs to keep, and `map` and `snd` to remove the indexing again.

c) Complete the function `same` that takes two lists and returns a list of the positions where their elements coincide. For instance, the strings `"Mary"` and `"Jane"` have the same 2nd characters, so `same` should return the list `[2]`. Use the functions `filter`, `map`, `fst`, `snd`, `zip` (or `numbered`), and `zipWith`.

```
ghci> numbered "days"
[(1,'d'),(2,'a'),(3,'y'),(4,'s')]
ghci> everyother "Elizabeth"
"Eiaeh"
ghci> same "Charles" "Charlotte"
[1,2,3,4,5]
```

Extra

Partial application, sections, function composition

Haskell has several further techniques to help readability (or to completely obfuscate your code, depending on how you use them).

Partial application is when you provide a function with some of its arguments, but not all. For instance, `max :: Int -> Int -> Int` is a function taking two integers; `max 3 5` returns the integer `5`; and `max 3 :: Int -> Int` is a function that takes one more integer, and then returns an integer.

To put this another way: function application takes one argument at a time, and is left-associative:

```
max 3 5  ==  (max 3) 5  ==  5
```

Sections are partially-applied infix operators, written by putting parentheses around the operator and any given argument. For instance, `(+3)` is the function that adds three to any number; `(5*)` multiplies by five; `(<10)` checks if a number is less than ten; `(/=0)` if it is non-zero. With zero arguments, a section makes a prefix function: e.g. `(+)` is addition.

```
1 < 2  ==  (<) 1 2  ==  (1<) 2  ==  (<2) 1  ==  True
```

One can use sections with normal functions by making them infix with backticks, `` `...` ``:

```
mod 5 3  ==  (`mod` 3) 5  ==  (5 `mod`) 3  ==  2
```

Function composition, written `f . g` to match the mathematical notation $f \circ g$, applies first `g` and then `f` to an argument. For instance, `(+1).length` is the function that takes the length of a list and adds one to that. It is defined as follows:

```
(.) :: (b -> c) -> (a -> b) -> a -> c
(f . g) x  =  f (g x)
```

Exercise 4:

a) Use partial application, sections, and function composition to simplify your answers to the previous exercises.