Tutorial 3: Maps, Filters, Comprehensions

In this tutorial we will practice the higher-order functions `map` and `filter`, as well as list comprehensions. The function `map` takes as its first argument a function `f :: a -> b` and applies it to each element in a list, its second argument.

```
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
```

A list comprehension is a convenient notation for list operations. The syntax is designed after the way sets are described in mathematics, e.g. the set of all primes:

$$\{n \mid n \in \mathbb{N}, n \text{ is prime}\}$$

For simplicity we take the predicate "is prime" as given. In Haskell, if we are given a function `prime :: Int -> Bool` that tells us whether a given number is prime, we can get the list of all prime numbers by a list comprehension as follows:

```
primes :: [Int]
primes = [ n | n <- [1..] , prime n ]
```

This gives the list of integers `n` that are **drawn from** the infinite list `[1..]` and that satisfy the condition `prime n`. Here, we collect just the numbers `n`, but any function over `n` is permitted. A list comprehension is constructed as follows:

```
[ <exp> | <qualifier_1> , ... , <qualifier_n> ]
```

where each qualifier can be a **generator** `x <- xs` that **draws** elements `x` from a list `xs` (such as `n <- [1..]` above), or a boolean **guard** (such as `prime n` above). List comprehensions are intimately related to maps and filters:

```
map f (filter p xs)   ⇔   [ f x | x <- xs , p x ]
```

Exercise 1: Write three versions of each of the following functions, one using recursion, one using map and filter, and one using a list comprehension.

a) A function `halves` that divides every number in a list by half. You can use the function `half` to halve each number.

b) A function `evens` that removes any odd numbers from a list (use the built-in `even`).

c) A function `halveEvens` that halves every even number in a list.

```
*Main> evens1 [1..10]
[2,4,6,8,10]
*Main> halves1 [1..10]
[0,1,1,2,2,3,3,4,4,5]
*Main> halveEvens1 [1..10]
[1,2,3,4,5]
```

Exercise 2:

a) Complete the function `shorts` that removes all strings longer than 5 characters from a list. Write two versions: one using a list comprehension, and one using `filter`. For the latter, define the filter function in a where-clause.

b) Complete the function `squarePositives` that takes all positive integers in a list and squares them. Write one version with a list comprehension, and one with `map` and `filter`. Use a where-clause to define any auxiliary functions you might need.

c) Complete the function `oddLengthSums` that given a list of integer lists, returns for each odd-length list its sum. Write a version using list-comprehension and one using `map` and `filter`. In both versions you may use `odd`, `length`, and `sum`.

```
*Main> shorts1 ["The","Lannisters","send","their","regards"]
["The","send","their"]
*Main> squarePositives1 [-3,4,1,-2,0,3]
[16,1,9]
*Main> oddLengthSums1 [[1],[1,2],[1,2,3],[1..4],[1..5]]
[1,6,15]
```

d) **Optional:** remove the where-clauses in the above functions by using **anonymous functions** and **sections**.

Exercise 3:

a) Complete the function `everyother` that takes every other element from a list (of any type), starting with the first. Use a list comprehension, with the `zip` function to count elements and `odd` to select those to keep. **Optional**: see if you can improve the solution with the `cycle` function.

```
*Main> everyother [1..10]
[1,3,5,7,9]
```

b) Complete the function `same` that takes two lists and returns a list of the positions where their elements coincide. For instance, the strings `"Winterfell"` and `"Kings Landing"` have the same 2nd and 3rd characters, so `same` should return the list `[2,3]`. Use a list comprehension and the function

`zip3 :: [a] -> [b] -> [c] -> [(a,b,c)]` .

Another kind of map

Continuing the development of our text adventure game, we will take another look at the game data of the previous tutorial.

Exercise 4: Write the functions `oneWay` and `bothWays` from the previous tutorial once more, this time using list comprehensions.

```
*Main> oneWay 5 theMap
[6]
*Main> bothWays 5 theMap
[2,3,6]
```

Hint: for `oneWay` use a list comprehension to find all locations `y` such that `(x,y)` is in `theMap` and `x` is equal to the given location. For `bothWays`, use two comprehensions, one for each direction.

The game will need to remember where your party is, at any given time, and it should display (in a nice way) where you can travel and who is with you. We would like to see:

```
*Main> start
You are in Winterfell.
You can travel to
  1. Castle Black
  2. The Twins
With you are
  3. Ned Stark
  4. Robert Baratheon
You can see
  5. Bran
  6. Catelyn
  7. Hodor
  8. Sansa
```

Exercise 5: First, we create the enumeration. Complete the function `enumerate` that, given an index `n` and a list of strings, numbers each string from `n` up and formats it nicely: two spaces, the number, a period and another space, and the string. Use a list comprehension with the functions `zip` and `show`, which displays Haskell objects as strings. Finally, apply the function `unlines` to the list comprehension to make it into one string.
(Below, `'\n'` is the **newline** character, and `putStr` prints a string to the console.)

```
*Main> enumerate 1 ["Bran","Rickon","Arya","Sansa"]
"  1. Bran\n  2. Rickon\n  3. Arya\n  4. Sansa\n"
*Main> putStr it
  1. Bran
  2. Rickon
  3. Arya
  4. Sansa
```

Data types

We want our "current game" to be represented by a `Node`, your current location, your `Party`, and the various characters at each location, of type `[Party]`. We could use a triple, but instead we'll start practicing a new skill, data types:

```
data Game = Game Node Party [Party]
```

This creates a genuinely new type `Game`, with a single **constructor** `Game` with the pattern `(Game n p ps)`. This pattern is used to build objects of type `Game`, as the example `start` shows, and also to decompose them, as the functions in this part of the tutorial will do. First, however, note that if you ask for `start` in the terminal, it will say:

```
*Main> start
<interactive> error:
    * No instance for (Show Game) ...
```

The type class `Show` collects those types `a` for which the function `show :: a -> String` is defined. This function is used by GHCi to display the answers it gives you. When we build a new data type, we can make our own `show` function for it. It is already in your file: all you need to do is uncomment the following lines. Then reload, and try `start` again.

```
instance Show Game where
    show (Game n p ps) = "You are in " ++ (locations !! n)
```

Exercise 6: Complete `showGame` so that it gives a nice representation of the state of the game, as above, using your `enumerate`. The character `'\n'` creates a new line. When it works, change the instance of `Show` for `Game` into `show = showGame`.

Exercise 7: Complete the function `go` that takes a `Game` and an integer `i` corresponding to one of the accessible locations, and moves the game to that location. Other inputs `i` may be ignored. Use `(!!)` to get the ith element of a list, but note that it starts at zero!

```
*Main> go start 2
You are in The Twins.
You can travel to
  1. Kings Landing
  2. Saltpans
  3. Winterfell
With you are
  4. Ned Stark
  5. Robert Baratheon
You can see
  6. Walder Frey

*Main> go it 1
You are in Kings Landing.
You can travel to
  1. The Twins
With you are
  2. Ned Stark
  3. Robert Baratheon
You can see
  4. Jaime
  5. Queen Cersei
```