Higher-order functions

One of the most powerful ideas in Haskell, and functional programming in general, is that **functions are first-class values**. This means that we can use functions in the same places where we use other kinds of values: We can put them in lists, we can store them in variables, and so on. Most importantly, we can *pass them as parameters to functions*, and we can also *return them as the results from functions*.

We focus on the first part of this here: *functions that take other functions as parameters*.

Map

The best place to start this journey is with functions that operate on lists. One of the most important such functions is map:

> map creates a new list by applying a provided function to each element of a list‘.

For example, if double x = 2∗x, then map double [1, 2 , 3] will result in [2, 4, 6]. In other words, it behaves exactly like the list comprehension [double x | x <– [1, 2, 3]]. In fact, we can define map as a list comprehension:

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

We can also define it via a standard recursive pattern:

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

Note the second case. We call map f xs to obtain the result for the tail of our list. Then we also compute f x and put it at the front of the list.

Take a moment to think about the type of the map function: It takes a a–>b function (the parentheses are important), and a list of a values, and produces a list of b values.

You might wonder why you should prefer the map function over list comprehensions. There are two reasons. The first is the idea of *curried functions* and *partial application* which we will discuss later. The other is the fact that map can actually be extended to work with other collection types, not just lists, and more or less in the same way.

Practice

1. Use map to write a function that converts a string into uppercase. You may use the function Data.Char.toUpper which takes a character and converts it to uppercase, if possible.

2. Use map to build a list of all the characters based on their integer codes, starting from the one with code 32 and ending with the one with code 128. The function Data.Char.chr returns the character corresponding to a code.
3. Define the length function for lists using map and sum.

Filter

Another important higher-order function is filter. filter takes a predicate, which is a function of type a -> Bool. Then it takes a list of values, applies the predicate to them, and only returns those for which the predicate is True. In effect:

> filter keeps only those elements from the list that satisfy a provided condition.

With list comprehensions, we could write filter like this:

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

Or we can use recursion:

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

As an example, we can keep only the digits from a string by doing: filter Data.Char.isDigit lst.

Practice Problems

To practice thinking about higher-order functions, here are some practice problems to work on.

1. Write a function zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]. It takes a function that turns an a and a b into a value of type c, and also takes a list of as and a list of bs. It then forms a list out of the result of applying the function to the corresponding pairs of elements.
2. Write a function takeWhile :: (a -> Bool) -> [a] -> [a] which takes as input a predicate and a list and retains the elements from the list as long as the predicate is true.
3. Write a function dropWhile :: (a -> Bool) -> [a] -> [a] which takes as input a predicate and a list and drops the elements from the list as long as the predicate is true.
4. Write a function splitWith :: (a -> Bool) -> [a] -> ([a], [a]) which takes as input a predicate and a list, and separates the list in two lists, with the first list containing those elements for which the predicate is True and the second list containing those elements for which the predicate is False. The order of elements must be maintained within each list.