Exercises
Higher-order functions

When writing the following functions, you should try to eliminate as many unecessary arguments as possible.

Exercise 1:
Show how the list comprehension

```
[f x | x <- xs, p x]
```

can be re-expressed using the higher-order functions ***map*** and ***filter***.

Solution 1:

```
map f (filter p xs)
```

Exercise 2:
Without looking at the definitions from the standard Prelude, define the following higher-order library functions on lists. To use the ***Prelude*** name, use

```
import Prelude hiding (all, any, takeWhile, dropWhile)
```

1. Decide if all elements of a list satisfy a predicate:

   ```
   all :: (a -> Bool) -> [a] -> Bool
   ```

2. Decide if all elements of a list satisfy a predicate:

   ```
   any :: (a -> Bool) -> [a] -> Bool
   ```

3. Select elements from a list while they satisfy a predicate:

   ```
   takeWhile' :: (a -> Bool) -> [a] -> [a]
   ```

4. Remove elements from a list while they satisfy a predicate

   ```
   dropWhile :: (a -> Bool) -> [a] -> [a]
   ```

Solution 2:
Note: We use *all'* etc so as not to clash with Prelude defined functions.

1. Decide if all elements of a list satisfy a predicate:

```
all' :: (a -> Bool) -> [a] -> Bool
all' p = and . map p
```

2. Decide if all elements of a list satisfy a predicate:

```
any' :: (a -> Bool) -> [a] -> Bool
any' p = or . map p
```

3. Select elements from a list while they satisfy a predicate:

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

4. Remove elements from a list while they satisfy a predicate

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

Exercise 3:
Redefine the functions

```
map f
```

and

```
filter p
```

using

```
foldr
```

Solution 3:

```
map f = foldr (\x xs -> f x : xs) []
```

```
filter :: (a -> Bool) -> [a] -> [a]
filter p = foldr (\x acc -> if p x then x : acc else acc) []
```

Exercise 4:
Noting that *String* is the same as [*Char*]. Define a function *capitalises*, of type

```
capitalises :: String -> String
```

which takes a list of characters as its argument and returns the same list as its value except that each lower-case letter has been replaced by its upper-case equivalent. Thus,
capitalises "Bohemian Rhapsody" = "BOHEMIAN RHAPSODY".
Hint: Use *toupper* which returns the uppercase of a letter and *map*. You should try to eliminate arguments where possible.

Solution 4:

```
capitalises :: String -> String
capitalises = map toUpper
```

Exercise 5:

Define a function $squareall :: [Int] \rightarrow [Int]$ which takes a list of integers and produces a list of the squares of those integers. For example,
$squareall[6, 1, (-3)] = [36, 1, 9]$.
Hint: Using map, You should try to eliminate arguments where possible.

Solution 5:

```
squareall :: [Int] -> [Int]
squareall = map (\x -> x*x)
```

Exercise 6:

Define a function *nestedreverse* which takes a list of strings as its argument and reverses each element of the list and then reverses the resulting list. Thus, *nestedreverse* ["in", "the", "end"] = ["dne", "eht", "ni"].

Hint: Using map, you should try to eliminate arguments where possible.

Solution 6:

```
nestedreverse :: [String] -> [String]
nestedreverse = reverse . map reverse
```

Exercise 7:

Define a function $atfront :: a \rightarrow [[a]] \rightarrow [[a]]$ which takes an object and a list of lists and prepends the object at the front of every component list. For example,
$atfront\ 7\ [[1, 2], [], [3]] = [[7, 1, 2], [7], [7, 3]]$.
Hint: Using map, you should try to eliminate arguments where possible.

Solution 7:

```
atfront :: a -> [[a]] -> [[a]]
atfront x = map (x:)
```

Exercise 8:

Define a function lengths which takes a list of strings as its argument and returns the list of their lengths. For example,
lengths ["the", "end", "is","nigh"] = [3, 3, 2, 4].
Hint: You should try to eliminate arguments where possible.

Solution 8:

```
lengths :: [String] -> [Int]
lengths = map length
```

Exercise 9:

Using the higher-order function map define a function *sumsq* which takes an integer n as its argument and returns the sum of the squares of the first n integers. That is to say, *sumsq* $n = 1^2 + 2^2 + 3^2 + \ldots + n^2$

Solution 9:

```
square :: Num a => a -> a
square x = x * x
sumsq :: Integral a => a -> a
sumsq n = sum (map square [1..n])
```

Exercise 10:

The function filter can be defined in terms of *concat* and *map*:

```
filter p = concat.map box where box x = ...
```

Write down the definition of box x

Solution 10:

```
filter1 :: (a -> Bool) -> [a] -> [a]
filter1 p = concat.map box
where box x
          | p x = [x]
          | otherwise = []
```

Exercise 11:

Define a function *wvowel* (without vowels) which removes every occurrence of a vowel from a list of characters.

Solution 11:

```
wvowel :: String -> String
wvowel xs = filter f xs where f x = not (x == 'a' ||
                                         x == 'e' ||
                                         x == 'i' ||
                                         x == 'o' ||
                                         x == 'u' )
```

Exercise 12:

Define a function wiv (without internal vowels) which takes a list of strings as its argument and removes every occurrence of a vowel from each element. For example,

```
wiv ["the", "end", "is", "nigh"] = ["th", "nd", "s", "ngh"]
```

Solution 12:

```
wiv :: [String] -> [String]
wiv = map wvowel
```