Worksheet 2: *Lists*

Template file:	Worksheet2.hs
Labs:	Friday 10th February, 2017
Hand-in:	Tuesday 14st February, 2017at 15:00hr
Topics:	Lists. Map and filter. List comprehension.

Message: (1) register your attendance, (2) each sub-question is worth 10 points, (3) scripts that don't compile properly loose 20% points, (4) don't forget to put your name on your script, (5) don't try to find solutions on the web, you learn by doing it yourself! The answers are generally short. (6) Take care that the layout is pleasing (7) have fun!

1. An phone book for storing names and telephone numbers can be implemented in Haskell as follows.

```
type Name = String
type PhoneNumber = Int
type Person  = (Name, PhoneNumber)
type PhoneBook = [Person]
```

   (a) Write the function `add::Person -> PhoneBook -> PhoneBook` that adds an entry to the phone book at the beginning of the list.

   (b) Write the function `delete::Name -> PhoneBook -> PhoneBook` that given the name of a person deletes all entries in the phone book with that name.

   (c) Write the function `find::Name -> PhoneBook -> [PhoneNumber]` that gives the list of all telephone numbers of a certain person.

   (d) Write the function `update::Name -> PhoneNumber -> PhoneNumber -> PhoneBook -> PhoneBook` that given the name and old phone number of a person updates that entry in the phone book with that the new phone number. (You may assume that the phonebook does not contain multiple entries of the same data.)

2. A Bank stores details on its customers via their national insurance number, their age, and their balance. This gives the following type definitions.

```
type NI = Int
type Age = Int
type Balance = Float
type Customer  = (NI,Age, Balance)
type Bank = [Customer]
```

(a) Define a function `retired :: Customer -> Bool` which returns true if the person is, or is over, 60 years.

(b) Define a function `deposit :: Customer -> Float -> Customer` which adds a given amount to the person's balance.

(c) Define a function `withdraw :: Customer -> Float -> Customer` which removes a given amount from the person's balance, but only if the remaining total is positive!

(d) Define a function `credit :: Bank -> [Customer]` which returns those people who are not overdrawn.

3. Using list comprehension define the function `cubeOdds :: [Int] -> [Int]` which takes a list of integers as input and returns a list consisting of the cube of only the odd numbers, eg `cubeOdds [3,6,4,5] = [27,125]`. Define a function `cubeOdds2` which has the same effect as `cubeOdds` but which is defined using map and filter instead of list comprehension.

4. Define a function `repChar :: (Char, Char) -> String -> String` which takes as input a pair of characters and a string (a list of characters) and returns the result of replacing the first character by the second in the string. For instance, `repChar ('a','o') "Too many aaaaaaaaas"` gives `"Too mony ooooooooos"`.

5. (a) Define a function `zap :: [Int] -> [Int] -> [(Int,Int)]` that given two lists of integers produces one list of pairs of integers, as long as the shortest input list.

   For example: `zap [1,2,3,4] [10,20,30] = [(1,10),(2,20),(3,30)]`

   (b) Define a function `addIndex :: [Int] -> [(Int,Int)]` that given a list $[n_1, n_2, \ldots n_k]$ of integers produces the list $[(1, n_1), (2, n_2), \ldots, (k, n_k)]$ which is a list of pairs of integers.

   For example `addIndex [2,2,3,1] -> [(1,2),(2,2),(3,3),(4,1)]`

   (c) Define a function `extend :: Int -> String -> String` that given an integer k and a string provide k is larger than the length of the string extends the string with blanks, so that the resulting string has length k. And otherwise when k is less or equal than the length of the given string, this string is outputted without any change.

   For example

   `extend 5 "abc" = "abc␣␣"`
   `extend 1 "abc" = "abc"`
   where ␣ is used to indicate blank space.