Flolac 2008
Functional programming in Haskell
Assignment 2, Due date: July 3

1. (List comprehension, 30%)

(a) (5%) Using list comprehensions, define a function, `countNeg`, for counting the number of negative numbers in a list of numbers.

```
countNeg :: [Int] ->  Int
countNeg [1, -2, 3, -5] = 2
```

(b) (10%) Define x^n using a list comprehension. Name the function as `raise`:

```
raise :: Int ->  Int ->  Int
raise 2 4= 16
```

(c)(15%) **Pascal's triangle** is a triangle of numbers

```
     1
    1  1
   1  2  1
  1  3  3 1
 1  4  6  4  1
1  5  10  10  5  1
.............
```

computed as follows: (1) The first row just contains a 1.

(2) The following rows are computed by adding together adjacent numbers in the row above, and adding a 1 at the beginning and at the end.

Write a function, `pascal`, using list comprehension and ++, which maps a positive integer *n* to the *nth* row of Pascal numbers.

For example, `pascal 5 = [1, 4, 6, 4, 1].`

Hint: define an auxiliary function `pairs` which construct pairs from two consecutive integer in a list.

2. (Higher-order functions and list comprehension 30%)

(a) (10%) Study the code fragment below and identify the operation it provides. Then rewrite it using `map` and `filter`.

```
q1f1 :: [Int] -> [Int]
q1f1 [] = []
q1f1 (x:xs) | x < 3     = q1f1 xs
            | x > 10    = q1f1 xs
            | otherwise = x * 3 : q1f1 xs
```

(b) (10%) Now rewrite `q1f1` using *list comprehensions and name it as* `q1f1b`.

(c) (10%) Express the comprehension

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

using the functions **map** and **filter.** Call the function **compre:**

`compre xs f p =` … --using map and filter

3. **(Function composition, 20%)** Use the functions, *remdup*, *elemOcc*, and *map* to define a function **occurrences** that receives a list and returns a list of pairs of an element and the number of its occurrences in the input list.

```
remdup :: Eq a => [a] -> [a] --or [Char]->[Char]
remdup [] = []
remdup (x:xs) = x : remdup (filter ?? xs)

elemOcc :: Eq a => a -> [a] -> Int --or [Char]->Int
elemOcc x = length . (filter ??)

occurences :: Eq a => [a] -> [(a,Int)]
occurences xs = map (??) (??)
```

Example: >occurrences [‘a’, ‘c’, ‘d’, ‘a’, ‘c’]
[(‘a’, 2), (’c’, 2), (‘d’, 1)]

4. (fold 20%)

(a) Use **foldr** to define ***map f***. To avoid confusion, please rename it as **myMap**.

```
myMap :: (a->b)->[a]->[b]
myMap f = foldr ...
```

(b) The “unwords” function creates a string from a *list of strings* by inserting a space character between the original strings. For examples:

```
unwords :: [String] -> String
unwords ["aa","bb","cc","dd","ee"]
= “aa bb cc dd ee”
```

Please define unwords in terms of “***foldr1***”.