CIS 194: Homework 4

Due Monday, February 11

What to turn in: you should turn a single `.hs` (or `.lhs`) file, which **must type check**.

Exercise 1: Wholemeal programming

Reimplement each of the following functions in a more idiomatic Haskell style. Use *wholemeal programming* practices, breaking each function into a pipeline of incremental transformations to an entire data structure. Name your functions `fun1'` and `fun2'` respectively.

1.
```
fun1 :: [Integer] -> Integer
fun1 []     = 1
fun1 (x:xs)
  | even x    = (x - 2) * fun1 xs
  | otherwise = fun1 xs
```

2.
```
fun2 :: Integer -> Integer
fun2 1 = 0
fun2 n | even n    = n + fun2 (n `div` 2)
       | otherwise = fun2 (3 * n + 1)
```

Hint: For this problem you may wish to use the functions `iterate` and `takeWhile`. Look them up in the Prelude documentation to see what they do.

Exercise 2: Folding with trees

Recall the definition of a *binary tree* data structure. The *height* of a binary tree is the length of a path from the root to the deepest node. For example, the height of a tree with a single node is 0; the height of a tree with three nodes, whose root has two children, is 1; and so on. A binary tree is *balanced* if the height of its left and right subtrees differ by no more than 1, and its left and right subtrees are also balanced.

http://en.wikipedia.org/wiki/Binary_tree

You should use the following data structure to represent binary trees. Note that each node stores an extra `Integer` representing the height at that node.

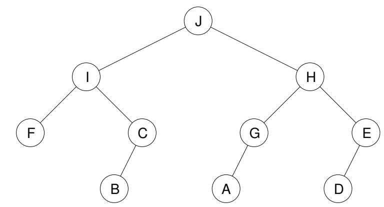
```
data Tree a = Leaf
            | Node Integer (Tree a) a (Tree a)
  deriving (Show, Eq)
```

For this exercise, write a function

```
foldTree :: [a] -> Tree a
foldTree = ...
```

which generates a balanced binary tree from a list of values using `foldr`.

For example, one sample output might be the following, also visualized at right:




```
foldTree "ABCDEFGHIJ" ==
  Node 3
    (Node 2
      (Node 0 Leaf ’F’ Leaf)
      ’I’
      (Node 1 (Node 0 Leaf ’B’ Leaf) ’C’ Leaf))
    ’J’
    (Node 2
      (Node 1 (Node 0 Leaf ’A’ Leaf) ’G’ Leaf)
      ’H’
      (Node 1 (Node 0 Leaf ’D’ Leaf) ’E’ Leaf))
```

Your solution might not place the nodes in the same exact order, but it should result in balanced trees, with each subtree having a correct computed height.

Exercise 3: More folds!

1. Implement a function

   ```
   xor :: [Bool] -> Bool
   ```

   which returns `True` if and only if there are an odd number of `True` values contained in the input list. It does not matter how many `False` values the input list contains. For example,

   `xor [False, True, False] == True`

   `xor [False, True, False, False, True] == False`

   Your solution must be implemented using a fold.

2. Implement `map` as a fold. That is, complete the definition

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

   in such a way that `map’` behaves identically to the standard `map` function.

3. **(Optional)** Implement `foldl` using `foldr`. That is, complete the definition

```
myFoldl :: (a -> b -> a) -> a -> [b] -> a
myFoldl f base xs = foldr ...
```

in such a way that `myFoldl` behaves identically to the standard `foldl` function.

Hint: Study how the application of `foldr` and `foldl` work out:

```
foldr f z [x1, x2, ..., xn] == x1 ‘f‘ (x2 ‘f‘ ... (xn ‘f‘ z)...)
foldl f z [x1, x2, ..., xn] == (...((z ‘f‘ x1) ‘f‘ x2) ‘f‘...) ‘f‘ xn
```

Exercise 4: Finding primes

Read about the *Sieve of Sundaram*. Implement the algorithm using function composition. Given an integer n, your function should generate all the odd prime numbers up to $2n + 2$.

http://en.wikipedia.org/wiki/Sieve_of_Sundaram

```
sieveSundaram :: Integer -> [Integer]
sieveSundaram = ...
```

To give you some help, below is a function to compute the *Cartesian product* of two lists. This is similar to `zip`, but it produces all possible pairs instead of matching up the list elements. For example,

```
cartProd [1,2] ['a','b'] == [(1,'a'),(1,'b'),(2,'a'),(2,'b')]
```

It's written using a *list comprehension*, which we haven't talked about in class (but feel free to research them).

```
cartProd :: [a] -> [b] -> [(a, b)]
cartProd xs ys = [(x,y) | x <- xs, y <- ys]
```