CSCI 490 problem set 3

Due Tuesday 9 February, 2016

Revision 1: compiled Tuesday 2nd February, 2016 at 13:04

Exercise 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.

Exercise 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.

Exercise 3 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 size (total number of nodes) of the binary tree at that node.

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

For this exercise, write a function, implemented using `foldr`,

```
mkBalancedTree :: [a] -> Tree a
mkBalancedTree = foldr ...
```

which generates a balanced binary tree from a list of values.

For example, one sample output might be the following, also visualized at right in Figure 1:

```
foldTree "ABCDEFGHIJ" ==
  Node 10
    (Node 4
      (Node 1 Leaf 'F' Leaf)
      'I'
      (Node 2 (Node 1 Leaf 'B' Leaf) 'C' Leaf))
    'J'
    (Node 5
      (Node 2 (Node 1 Leaf 'A' Leaf) 'G' Leaf)
      'H'
      (Node 2 (Node 1 Leaf 'D' Leaf) 'E' Leaf))
```

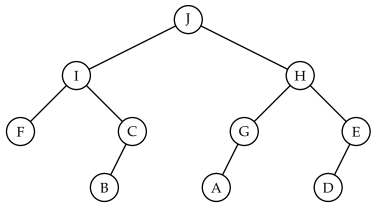

Figure 1: A balanced tree

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

Exercise 4 Implement a fold for the type `Nat`, defined by

```
data Nat where
  Z :: Nat
  S :: Nat -> Nat
```

Exercise 5 Describe the set of all possible functions of type `forall a. (a -> a) -> a -> a`.

Exercise 6 Implement (sensible) functions with types

```
Nat -> (forall a. (a -> a) -> a -> a)
```

and

```
(forall a. (a -> a) -> a -> a) -> Nat.
```

Note carefully where the parentheses are in the above types. We haven't specifically discussed types like this, but see if you can figure out what they mean; ask me if you need a hint.

You will need to enable the `Rank2Types` extension, by putting `{-# LANGUAGE Rank2Types #-}` at the top of your `.hs` file.

The following exercises concern the type of *rose trees*, where each node contains a value of some type and *any number* of children (including the possibility of *zero* children):

```
data Rose a where
  Node :: a -> [Rose a] -> Rose a
```

Exercise 7 Implement a function

```
mapRose :: (a -> b) -> Rose a -> Rose b.
```

Exercise 8 Implement a fold for `Rose a`.

I'm not telling you the type; that's part of the fun.

Exercise 9 Using your fold, implement a function

```
height :: Rose a -> Integer.
```

Again, the height of a tree is defined as the length of the deepest path from the root to any leaf. The height of a leaf (a node with no children) is therefore zero. An example is shown in Figure 2.

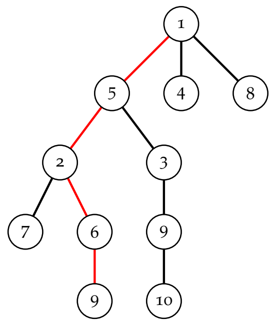

Figure 2: A tree with height 4

Exercise 10 The *width* of a tree is defined as the length of the longest path between two leaves. (That is, a path between two leaves starts at a leaf, goes up the tree for a while, and then goes back down to another leaf.) Be careful to note that, as illustrated in Figure 3, the maximum-width path may not pass through the root of the tree!

Use your fold to implement a function

```
width :: Rose a -> Integer.
```

Note: this is much trickier than it may first appear. You will probably need a helper function. Please ask for hints if you are stuck.

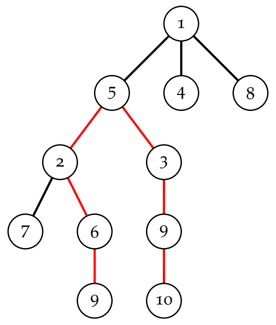

Figure 3: A tree with width 6

Read sections 1–5 of John Backus's 1977 Turing Award lecture, *Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs*, available from `http://worrydream.com/refs/Backus-CanProgrammingBeLiberated.pdf`.

The rest of this lecture has lots of cool stuff in it too; you're welcome to read more if you find it interesting, although beware that it starts getting extremely hairy about halfway through.

Exercise 11 Backus delivered this lecture almost 40 years ago. In what ways do his remarks still apply today, and in what ways are they outdated? Give two specific examples of each.

Exercise 12 Translate the Innerproduct function, defined in section 5.2, into Haskell. (You may use the standard `transpose` function, defined in the `Data.List` module.)