CSCI 365 Module 3

due Wednesday, 16 February 2022

Specification

- To receive **Level 1** credit for this module:
  - Complete exercises 1, 2, 5–7, and 9–12.
  - Your code must adhere to the style guide linked from the course web page.
  - You must complete Quiz 3 for credit, which will test your facility with folds.
- To receive **Level 2** credit for this module:
  - Complete everything required for Level 1.
  - Complete *any two* of exercises 3, 4, and 8.
  - Read part of John Backus's 1977 Turing Award lecture, write a short response, and solve exercise 13, as explained at the end of this document.

List folds

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 (Level 2) Implement `foldl` using `foldr`. That is, complete the definition

```
myFoldl :: (b -> a -> b) -> b -> [a] -> b
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
```

Hint 2: Ask me if you want another hint.

Exercise 4 (Level 2) 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 = Empty
            | 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))
```

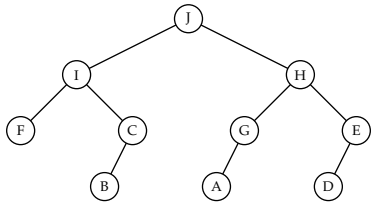

Figure 1: A balanced tree

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.

General folds

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
  RNode :: a -> [Rose a] -> Rose a
```

Exercise 5 Implement a function

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

Exercise 6 Implement a fold for `Rose a`. [1]

Exercise 7 Using your fold, implement a function

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

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.

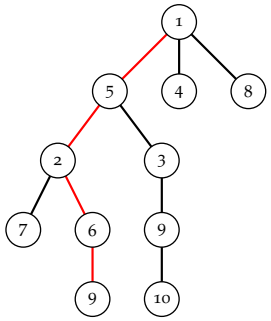

Figure 2: A tree with height 4

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

Exercise 8 (Level 2) 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 might 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.

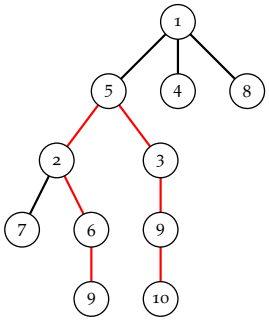

Figure 3: A tree with width 6

Representing data via folds

It turns out that having a fold for some data is just as good as having the data itself! That is, a fold over some data captures all possible

information about it. (We will make heavy use of this principle later, when we encode data in the lambda calculus.)

Exercise 9 Begin by implementing a fold for the type `Nat`, defined by

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

Exercise 10 Using what you know about parametricity, describe the set of all possible functions of type `forall a. (a -> a) -> a -> a`. How many different functions are there of this type? How would you characterize their behavior in English?

Hint: there are more than 2. Please ask for help if you are stuck!

Exercise 11 Implement a function with the type

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

Be sure that your function gives a *different* function of type `forall a. (a -> a) -> a -> a` for each `Nat`. (*Hint:* think about your answers to the previous two questions.)

Exercise 12 Now implement a function which goes the other way, that is, a function of type

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

which is inverse to your function from the previous exercise. Notice that since the `forall a` is *inside* the parentheses, this means you are *given* a function which has to work for any choice of `a`—which means that *you* get to pick the type `a` when you call it.

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

Reading (Level 2) 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`.

Backus delivered this lecture 45 years ago. In what ways do his remarks still apply today, and in what ways are they outdated? Give two specific examples of each.

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 13 (Level 2) Translate the Innerproduct function, defined

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