Assignment 3

Haskell

Due Date: Monday, February 27th 2017, 11:59pm

Directions

Answers to English questions should be in your own words; don't just quote from articles or books. We will take some points off for: code with the wrong type or wrong name, duplicate code, code with extra unnecessary cases, or code that is excessively hard to follow. You should always assume that the inputs given to each function will be well-typed, thus your code should not have extra cases for inputs that are not of the proper type. (Assume that any human inputs are error checked before they reach your code. Avoid duplicating code by using helping functions, library functions (when not prohibited in the problems), or by using syntactic sugars and local definitions (using **let** and **where**). It is a good idea to check your code for these problems before submitting. Be sure to test your programs on Eustis to ensure they work on a Unix system.

This is a group assignment. Make a group of 3-4 in WebCourses and use that to turn in the assignment.

Deliverables

For this assignment, turn in the following files:

- TotalHeight.hs
- SplitScreen.hs
- Improve.hs
- FreeQBExp.hs
- Rank.hs
- ToCharFun.hs
- MapInside.hs
- EncryptWith.hs
- ComposeList.hs

For coding problems, put your solution in the appropriate Haskell source file (named with ".hs" or ".lhs" suffix).

Problem 1 (20 pts)

This problem is about the type WindowLayout, which is defined in the file WindowLayout.hs.

```
module WindowLayout where
data WindowLayout = Window {wname :: String, width :: Int, height :: Int}
                  | Horizontal [WindowLayout]
                  | Vertical [WindowLayout]
                  deriving (Show, Eq)
```

In Haskell, write a function

```
totalHeight :: WindowLayout -> Int
```

that takes a <WindowLayout>, w1, and returns the total height of the window layout. The height is defined by cases as follows. The height of a <WindowLayout> of form

Window {wname = nm, width = w, height = h}

is h. The height of a <WindowLayout> of the form

Horizontal [$wl_1, \ldots, wl_n$]

is 0 if the list is empty, and otherwise is the maximum of the heights of wl_1 through wl_m (inclusive). The height of a <WindowLayout> of the form

Vertical [$wl_1, \ldots, wl_n$]

is the sum of the heights of wl_1 through wl_m (inclusive), which is 0 if the list is empty.

The file TotalHeightTests.hs contains tests that show how the function should work.

Be sure to follow the grammar! In particular, you need to use some helping functions to work on the lists that are part of the <WindowLayout> grammar. We will take off points if you do not follow the grammar (and you will spend more time trying to get your code to work).

Problem 2 (20 pts)

This is another problem about Window Layouts. Write a function

```
splitScreen :: String -> WindowLayout -> WindowLayout
```

that takes in a string, name, and a Window Layout, w1, and returns a Window Layout that is just like w1, except that for each window in w1 whose wname is (== to) name is changed to a horizontal window layout with both windows having the same name and half the width of the previous layout. (Hint: Use Haskell's div operator to do the division.)

Problem 3 (20 pts)

This problem uses the types Statement and Expression, which are found in the file StatementsExpressions.hs.

```
module StatementsExpressions where

data Statement = ExpStmt Expression
               | AssignStmt String Expression
               | IfStmt Expression Statement
                 deriving (Eq, Show)

data Expression = VarExp String
    | NumExp Integer
    | EqualsExp Expression Expression
    | BeginExp [Statement] Expression
      deriving (Eq, Show)
```

Write a function

```
improve :: Statement -> Statement
```

that takes a Statement, `stmt`, and returns a Statement just like `stmt`, except that two simplifications are made:

1. Each Statement of the form `(IfStmt (VarExp "true") s)` is replaced by a simplified version of *s* in the output.
2. Each Expression of the form (BeginExp [] e) is replaced by a simplified version of *e* in the output.

There are test cases contained in ImproveTests.hs.

Problem 4 (25 pts)

Consider the data type of quantified Boolean expressions defined as follows, in the file QBExp.hs.

```
module QBExp where
data QBExp = Varref String | QBExp `Or` QBExp
           | Not QBExp | Exists String QBExp
             deriving (Eq, Show)
```

Your task is to write the function

```
freeQBExp :: QBExp -> [String]
```

that takes a QBExp, qbe, and returns a list containing just the strings that occur as a free variable reference in qbe. The following defines what "occurs as a free variable reference" means. A string s *occurs as a variable reference* in a QBExp if s appears in a subexpression of the form `(Varref s)`. Such a string s occurs as a free variable reference if and only if occurs as a variable reference in a subexpression that is outside of any expression of the form `(Exists s e)`, which declares `s`.

Note that the lists returned by freeQBExp should have no duplicates. In the tests, the setEq function constructs a test case that considers lists of strings to be equal if they have the same elements (so that the order is not important).

Don't use tail recursion on this problem! Instead, use separate helping functions to prevent duplicates.

Problem 5 (20 pts)

In various contests the contestants are awarded places based on some score, and a list of winners is produced. For example, *ebird.org* maintains the lists of the top 100 birders in Florida this year. In such ranked lists, contestants that have the same score are considered tied; for example, if Audrey and Carlos have both seen 187 bird species this year, then they are considered tied, and both are listed as being in (say) 12th place. In this scenario, the next birder, with 186 species, is listed as being in 14th place, as Audrey and Carlos take places 12 and 13 together, even though they are listed as tied for 12th place.

In this problem you will write a general ranking function

rank :: (**Ord** a) => [a] -> [(**Int**, a)]

which for any type a that that is an instance of the Ord class, takes a list of elements of type a, things, and returns a list of pairs of Ints and a elements. The result is sorted (in non-decreasing order) on the a elements of things, and the Int in each pair is the rank of the element in the pair.

Hint: you can use sort from the module Data.List. You may also find it helpful to use a helping function so that you have some additional variables, even if you are not using tail recursion.

Problem 6 (5 pts)

In cryptography, one would like to apply functions defined over the type Int to data of type Char. However, in Haskell, these two types are distinct. In Haskell, write a function.

```
toCharFun :: (Int -> Int) -> (Char -> Char)
```

that takes a function f, of type `Int -> Int`, and returns a function that operates on characters. In your implementation you can use the `fromEnum` and `toEnum` functions that Haskell provides (found in the **`Enum`** instance that is built-in for the type `Char`).

Hint: note that `(fromEnum 'a')` is 97 and `(toEnum 100) :: Char`.

There are test cases contained in the file ToCharFunTests.hs.

Problem 7 (10 pts)

Using Haskell's built-in map function, write the function

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

that for some types a and b takes a function f of type a -> b, and a list of lists of type a, lls, and returns a list of type [[b]] that consists of applying f to each element inside each list in lls, preserving the structure of the lists.

There are test cases contained in the file MapInsideTests.hs.

Note that your code must use map. For full credit, write a solution that does not use any pattern matching.

As specified on the first page of this homework, turn in both your code file and the output of your testing.

Problem 8 (20 pts)

Write a higher-order function

```
encryptWith :: (Int -> Int) -> [String] -> [String]
```

that takes a cryptographic function, f, and a list of strings text, and returns a (poorly) encrypted version of each string in the list by using toCharFun f to encrypt each character.

There are test cases contained in the file EncryptWithTests.hs.

Problem 9 (15 pts)

Write a function

```
composeList :: [(a -> a)] -> (a -> a)
```

that takes a list of functions, and returns a function which is their composition.

Hint: note that composeList [] is the identity function.

Don't use last or init in your solution, as that will make your solution $O(n^2)$.

There are test cases contained in the file ComposeListTests.hs.