Assignment 2

Haskell

Due Date: Monday, February 6th 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. Use your group from assignment 1.

Deliverables

For this assignment, turn in the following files:

- Vectors.hs
- ApplyToList.hs
- BinaryRelationOps.hs
- Rotate.hs
- Hep.hs
- ListMin.hs
- WhatIndex.hs
- Solution.pdf

For problems that require an English answer, put your solution in Solution.pdf. For coding problems, put your solution in the appropriate Haskell source file (named with ".hs" or ".lhs" suffix).

Problem 1 (15 pts)

Complete the module Vectors found in the file Vectors.hs (provided in the assignment2-tests.zip file), by writing function definitions in the indicated places that implement the functions: `scale`, `add`, and `sub`. This module represents Vectors by lists of Doubles. The functions you are to implement are as follows.

(a) (5 pts) The function

```
scale :: Double -> Vector -> Vector
```

takes a Double y, and a Vector, v, and returns a new Vector that is just like v, except that each coordinate is y times the corresponding coordinate in v.

(b) (5 pts) The function

```
add :: Vector -> Vector -> Vector
```

takes two vectors and adds them together, so that each coordinate of the result is the sum of the corresponding coordinates of the argument Vectors. Your code should assume that the two vector arguments have the same length.

(c) (5 pts) The function

```
dot :: Vector -> Vector -> Double
```

takes two vectors and computes their dot product (or inner product), which is the sum of the products of the corresponding elements. Your code should assume that the two vector arguments have the same length.

There are test cases contained in VectorsTests.hs.

To run our tests, use the VectorsTests.hs file. To make that work, edit your code into the provided file Vectors.hs. Our tests use the FloatTesting module from assignment2-tests.zip.

You can use test_scale, test_add, or test_dot to test individual functions.

Problem 2 (20 pts)

This problem will have you write two functions that deal with the application of binary relations to keys (i.e., the look up of the values associated with a given key). In this problem binary relations are represented as lists of pairs, as described in the file BinaryRelation.hs:

```
-- $Id: BinaryRelation.hs
module BinaryRelation where
-- Binary relations are represented as lists of pairs
type BinaryRelation a b = [(a,b)]
```

In a BinaryRelation, the first part of a pair is called a "key" and the second part of a pair is called a "value".

Your code for the following two functions should go in a module named ApplyToList that imports the BinaryRelation module. Thus it should start as follows.

```
module ApplyToList where
import BinaryRelation
```

The functions you are to write are the following.

(a) (10 pts) Using a list comprehension, write the function

```
applyRel :: (Eq k) => k -> (BinaryRelation k v) -> [v]
```

When given a key value, ky, of some equality type k, and a BinaryRelation pairs, of type `(BinaryRelation k v)` the result is a list of values (of type v) that are the values from all the pairs whose key is ky. Note that values in the result appear in the order in which they appear in pairs.

(b) (10 pts) Using recursion (that is, without using a list comprehension or library functions), write the function

```
applyToList :: (Eq k) => [k] -> (BinaryRelation k v) -> [v]
```

When given a list of keys, keys, of some equality type k, and a BinaryRelation, `pairs`, the result is a list of values from all pairs in the relation `pairs` whose key is one of the keys in `keys`. Note that the order of the answer is such that all the values associated with the key in keys appear before any of the values associated with a later key, and similarly the values associated with other keys appear before later keys in keys. (Hint: You may use `applyRel` in your solution for this problem.)

There are test cases contained in ApplyToListTests.hs. That file imports Relations.hs.

To run our tests, use the ApplyToListTests.hs file. To make that work, you have to put your code in a module ApplyToList.

Problem 3 (30 pts)

In this problem you will implement 4 functions that operate on the type BinaryRelation, which is defined in the file BinaryRelation.hs:

```
module BinaryRelation where
-- Binary relations are represented as lists of pairs
type BinaryRelation a b = [(a,b)]
```

Your code should be written in a module named BinaryRelationOps, which should import the BinaryRelation module, and thus should start as follows.

```
module BinaryRelationOps where
import BinaryRelation
```

You are to write the following functions:

(a) (5 pts) The function

```
project1 :: (BinaryRelation a b) -> [a]
```

projects a binary relation on its first column. That is, it returns a list of all the keys of the relation (in their original order).

(b) (5 pts) The function

```
project2 :: (BinaryRelation a b) -> [b]
```

projects a binary relation on its second column. That is, it returns a list of all the values of the relation (in their original order). (Note that the resulting list may have duplicates even if the original relation had no duplicate tuples.)

(c) (10 pts) The function

```
select :: ((a,b) -> Bool) -> (BinaryRelation a b) -> (BinaryRelation a b)
```

takes a predicate and a binary relation and returns a list of all the tuples in the relation that satisfy the predicate (in their original order). Note that the predicate is a function that takes a single pair as an argument. For those pairs for which it returns **True**, the `select` function should include that pair in the result.

(d) (10 pts) The function

```
compose :: Eq b => (BinaryRelation a b) -> (BinaryRelation b c)
                   -> (BinaryRelation a c)
```

takes two binary relation and returns their relational composition, that is the list of pairs (a, c) such that there is a some pair (a, b) in the first argument binary relation and a pair (b, c) in the second relation argument.

There are test cases contained in BinaryRelationOpsTests.hs. To make this work your code must be in a module named BinaryRelationOps.

As always, after writing your code, run our tests, and turn in your solution and the output of our tests as specified on the first page of this homework.

Problem 4 (5 pts)

Consider the data type Amount defined below.

```
module Amount where
data Amount = Zero | One | Two | Three
```

In Haskell, write the polymorphic function

```
Rotate :: Amount -> (a,a,a,a) -> (a,a,a,a)
```

Which takes an Amount, `amt`, and a 4-tuple of elements of some type, (w,x,y,z), and returns a triple that is circularly rotated to the right by the number of steps indicated by the English word that corresponds to `amt`. That is, when `amt` is Zero, then (w, y, y, z) is returned unchanged; when `amt` is One, then (z, w, x, y) is returned; when `amt` is Two, then (y, z, w, x) is returned; finally when `amt` is Three, then (x, y, z, w) is returned.

Problem 5 (10 pts)

Write a function

```
hep :: [String] -> [String]
```

that takes a list of words (i.e., Strings not containing blanks), `txt`, and returns a list just like `txt` but with the following substitutions made each time they appear as consecutive words in `txt`:

- "you" is replaced by "u",
- "are" is replaced by "r",
- "your" is replaced by "ur",
- the three words "by the way" are replaced by the word "btw",
- the three words "for your information" is replaced by the word "fyi",
- "boyfriend" is replaced by "bf",
- "girlfriend" is replaced by "gf",
- the three words "be right back" are replaced by the word "brb",
- the three words "laughing out loud" are replaced by the word "lol",
- the two words "see you" are replaced by the word "cya",
- the two words "I will" are replaced by the word "I'll", and
- "great" is replaced by "gr8".

This list is complete (for this problem).

Tests can be found at HepTests.hs which you can get from WebCourses.

Problem 6 (10 pts)

In Haskell, using tail recursion, write a polymorphic function

```
listMin :: (Ord a) => [a] -> a
```

that takes a non-empty, finite list, `lst`, whose elements can be compared (hence the requirement in the type that a is an Ord instance), and returns a minimal element from `lst`. That is, the result should be an element of `lst` that is not larger than any other element of `lst`.

Although you are allowed to use the standard min function, your code must *not* use any other library functions.

In your code, you can assume that the argument list is non-empty and finite. There are test cases contained in ListMinTests.hs.

Note: your solution *must use tail recursion*.

Problem 7 (10 pts)

In Haskell, using tail recursion, write a polymorphic function

```
whatIndex :: (Eq a) => a -> [a] -> Integer
```

that takes an element of some Eq type, a, `sought`, and a finite list, `lst`, and returns the 0-based index of the first occurrence of `sought` in `lst`. However, if sought does not occur in `lst`, it returns -1.

Your code must not use any library functions and must be tail recursive.

In your code, you can assume that the argument list is finite. There are test cases contained in WhatIndexTests.hs.