COP 4020 — Programming Languages I February 6, 2015

Homework 2: Functional Programming in Haskell

See Webcourses and the syllabus for due dates.

Purpose

In this homework you will learn basic techniques of recursive programming over various types of (recursively-structured) data [UseModels] [Concepts]. Many of the problems exhibit polymorphism [UseModels] [Concepts]. The problems as a whole illustrate how functional languages work without hidden effects [EvaluateModels].

Directions

Answers to English questions should be in your own words; don't just quote text from the textbook.
We will take some points off for: code with the wrong type or wrong name, duplicated 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.) Make sure your code has the specified type by including the given type declaration with 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.
Since the purpose of this homework is to ensure skills in functional programming, we suggest that you work individually. (However, per the course's grading policy you can work in a group if you wish, provided that carefully follow the policy on cooperation described in the course's grading policy.)
Don't hesitate to contact the staff if you are stuck at some point.

What to Turn In

For each problem that requires code, turn in (on Webcourses) your code and output of testing with our test cases. Please upload code as a plain (text) file with the name given in the problem or testing file and with the suffix `.hs` or `.lhs` (that is, do *not* give us a Word document or a PDF file for the code). Also paste the output from our tests into the Comment box for that "assignment". For English answers, please paste your answer into the assignment as a "text answer" in the problem's "assignment" on Webcourses. For a problem with a mix of code and English, follow both of the above.
For all Haskell programs, you must run your code with GHC. See the course's Running Haskell page for some help and pointers on getting GHC installed and running. Your code should compile properly (and thus type check); if it doesn't, then you probably should keep working on it. Email the staff with your code file if you need help getting it to compile or have trouble understanding error messages. If you don't have time to get your code to compile, at least tell us that you didn't get it to compile in your submission.
You are encouraged to use any helping functions you wish, and to use Haskell library functions, unless the problem specifically prohibits that.

What to Read

Besides reading chapters 1-7 of the recommended textbook on Haskell [Tho11], you may want to read some of the Haskell tutorials. Use the Haskell 2010 Report as a guide to the details of Haskell.
Also read "Following the Grammar with Haskell" [Lea13] and follow its suggestions for planning and organizing your code. You may also want to read a tutorial on the concepts of functional programming

languages, such as Hudak's computing survey article mentioned in the syllabus. See also the course code examples page (and the course resources page).

Problems

Functions on Tuples

1. (5 points) [UseModels] The function you are to write in Haskell is

   ```
   average3 :: (Double,Double,Double) -> Double
   ```

   which takes a triple of `Double`s, `(x,y,z)`, and returns a `Double` that is the average (i.e., the arithmetic mean) of `x`, `y`, and `z`. The following are examples, written using the `Testing` and `FloatTesting` modules which are included in the hw2-tests.zip file.

   ```
   main = dotests "Average3Tests $Revision: 1.1 $" tests
   tests :: [TestCase Double]
   tests = [withinTest (average3 (0.0,0.0,0.0)) "~=~" 0.0
           ,withinTest (average3 (0.0,1.0,2.0)) "~=~" 1.0
           ,withinTest (average3 (75.0,100.0,50.0)) "~=~" 75.0
           ,withinTest (average3 (-30.2,10.1,55.7)) "~=~" 11.866666666666667
           ,withinTest (average3 (62.4,98.6,212.532)) "~=~" 124.51066666666668
           ,withinTest (average3 (10.0,100.0,3.14)) "~=~" 37.71333333333333
           ]
   ```

   To run our tests, use the `Average3Tests.hs` file. To make that work, you have to put your code in a module `Average3`, which will need to be in a file named `Average3.hs` (or `Average3.lhs`), in the same directory as `Average3Tests.hs`. Your file `Average3.hs` should thus start as follows.

   ```
   module Average3 where
   average3 :: (Double,Double,Double) -> Double
   ```

   Then run our tests by running the `main` function in `Average3Tests.hs`. Our tests are written using the `Testing.lhs` and `FloatTesting.hs` files, which are included in hw2-tests.zip.

```
-- $Id: Average3Tests.hs,v 1.1 2015/01/21 14:52:27 leavens Exp leavens $
module Average3Tests where
import Average3
import Testing
import FloatTesting

main = dotests "Average3Tests $Revision: 1.1 $" tests
tests :: [TestCase Double]
tests = [withinTest (average3 (0.0,0.0,0.0)) "~=~" 0.0
        ,withinTest (average3 (0.0,1.0,2.0)) "~=~" 1.0
        ,withinTest (average3 (75.0,100.0,50.0)) "~=~" 75.0
        ,withinTest (average3 (-30.2,10.1,55.7)) "~=~" 11.866666666666667
        ,withinTest (average3 (62.4,98.6,212.532)) "~=~" 124.51066666666668
        ,withinTest (average3 (10.0,100.0,3.14)) "~=~" 37.71333333333333
        ]
```

Figure 1: Tests for problem 1.

As specified on the first page of this homework, turn in both your code file and the output of your testing. (The code file should be uploaded to Webcourses, and the test output should be pasted in to the Comments box for that assignment.)

Recursion over Flat Lists

These problems are intended to give you an idea of how to write recursions by following the grammar for flat lists [Lea13].

2. [Concepts]

(a) (5 points) In Haskell, which of the following is equivalent to the list `[3,5,3]`?

1. `(3,5,3)`
2. `3:(5:3)`
3. `(3:5):3`
4. `(((3:5):3):[])`
5. `3:(5:(3:[]))`

(b) (10 points) Suppose that ohno is the list `['o', 'h', 'n', 'o']` and that `yikes` is the list `"yikes"`. For each of the following, say whether it is legal or illegal in Haskell, and if it is illegal, say why it is illegal.

1. `ohno:'y'`
2. `ohno ++ yikes`
3. `ohno:yikes`
4. `['o']:yikes`
5. `'o':yikes`

(c) (5 points) Haskell has built in functions **head** and **tail** defined as follows.

```
head             :: [a] -> a
head (x:_)       =  x
head []          =  error "Prelude.head: empty list"

tail             :: [a] -> [a]
tail (_:xs)      =  xs
tail []          =  error "Prelude.tail: empty list"
```

For example, **head** `[1 ..]` equals `1` and **tail** `[1 ..]` equals `[2 ..]`. Consider the following function.

```
rip lst =
    let back = tail lst
    in let front = head lst
       in (lst, front:back)
```

What is the result of the call `rip [3,4,7,5,8]`?

(We suggest that you think about it first, and only use the Haskell system to check your answer.)

3. [UseModels] This problem will have you write a solution in 2 ways. The problem is to write a function that takes a list of Integers and returns a list that is just like the argument but in which every element is 10 greater than the corresponding element in the argument list.

   (a) (5 points) Write the function

   ```
   add10_list_comp :: [Integer] -> [Integer]
   ```

   that solves the above problem by using a list comprehension.

   (b) (5 points) Write the function

   ```
   add10_list_rec :: [Integer] -> [Integer]
   ```

   that solves the above problem by writing out the recursion yourself; that is, without using a list comprehension and without using `map` or any other higher-order library function.

   There are test cases contained in `Add10ListTests.hs`, which is shown in Figure 2.

```
-- $Id: Add10ListTests.hs,v 1.1 2015/01/21 14:06:28 leavens Exp leavens $
module Add10ListTests where
import Testing
import Add10List  -- you have to put your solutions in module Add10List

version = "Add10ListTests $Revision: 1.1 $"
recursive_tests = (tests add10_list_rec)
comprehension_tests = (tests add10_list_comp)

-- do main to run our tests
main :: IO()
main = do startTesting version
          errs_comp <- run_test_list 0 comprehension_tests
          total_errs <- run_test_list errs_comp recursive_tests
          doneTesting total_errs

-- do test_comprehension to test just add10_list_comp
test_comprehension :: IO()
test_comprehension = dotests version comprehension_tests

-- do test_recursive to test just add10_list_rec
test_recursive :: IO()
test_recursive = dotests version recursive_tests

tests :: ([Integer] -> [Integer]) -> [TestCase [Integer]]
tests f =
    [(eqTest (f []) "==" [])
    ,(eqTest (f (1:[])) "==" (11:[]))
    ,(eqTest (f (3:1:[])) "==" (13:11:[]))
    ,(eqTest (f [1,5,7,1,7]) "==" [11,15,17,11,17])
    ,(eqTest (f [7 .. 21]) "==" [17 .. 31])
    ,(eqTest (f [8,4,-2,3,1,10000000,10])
          "==" [18,14,8,13,11,10000010,20])
    ]
```

Figure 2: Tests for problem 3. In the definition of `tests f` stands for one of the two functions you are to write.

To run our tests, use the `Add10ListTests.hs` file. To make that work, you have to put your code in a module `Add10List`, which will need to be in a file named `Add10List.hs` (or `Add10List.lhs`), in the same directory as `Add10ListTests.hs`. Your file `Add10List.hs` should thus start as follows.

```
module Add10List where
add10_list_rec :: [Integer] -> [Integer]
add10_list_comp :: [Integer] -> [Integer]
```

Then run our tests by running the `main` function in `Add10ListTests.hs`.

As specified on the first page of this homework, turn in both your code file and the output of your testing. (The code file should be uploaded to Webcourses, and the test output should be pasted in to the Comments box for that assignment.)

4. (10 points) [UseModels] In Haskell, write the function:

   ```
   cubeOdds :: [Integer] -> [Integer]
   ```

   that takes a list of Integers, `lst`, and returns a list of Integers that is just like `lst`, except that each odd element of `lst` is replaced by the cube of that element. In your solution, you might find it helpful to use the built-in predicate `odd`.

   There are examples in Figure 3.

```
-- $Id: CubeOddsTests.hs,v 1.1 2015/01/21 14:16:07 leavens Exp leavens $
module CubeOddsTests where
import CubeOdds
import Testing
main = dotests "CubeOddsTests $Revision: 1.1 $" tests
tests :: [TestCase [Integer]]
tests = [eqTest (cubeOdds []) "==" []
        ,eqTest (cubeOdds [3]) "==" [27]
        ,eqTest (cubeOdds [4]) "==" [4]
        ,eqTest (cubeOdds [4,3]) "==" [4,27]
        ,eqTest (cubeOdds [1,2,3,4,5,6]) "==" [1,2,27,4,125,6]
        ,eqTest (cubeOdds [3,10,3,5,600,0,-2,-1,-3])
                "==" [27,10,27,125,600,0,-2,-1,-27]
        ]
```

Figure 3: Tests for problem 4.

5. (10 points) [UseModels] Write the function

   ```
    deleteNth :: (Eq a) => Int -> a -> [a] -> [a]
   ```

   that takes a positive Int, `n`, an element, `toDelete`, of some equality type `a`, and a list, `as`, of type `[a]`, and returns a list that is just like `as`, but which does not contain the nth occurrence (in `as`) of the element `toDelete`.

   Your solution must *not* use any Haskell library functions. You may assume that `n` is strictly greater than 0.

   There are test cases contained in `DeleteNthTests.hs`, which is shown in Figure 4 on the following page.

   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.

6. (5 points) [Concepts] Is it possible to use a list comprehension to solve problem 5 in an easy, direct way? Briefly explain.

```haskell
-- $Id: DeleteNthTests.hs,v 1.1 2015/01/21 16:33:59 leavens Exp $
module DeleteNthTests where
import Testing
import DeleteNth

-- do main to run our tests
main :: IO()
main = dotests "DeleteNthTests $Revision: 1.1 $" tests

tests :: [TestCase [Int]]
tests =
    [(eqTest (deleteNth 1 3 []) "==" [])
    ,(eqTest (deleteNth 1 3 (1:[])) "==" (1:[]))
    ,(eqTest (deleteNth 1 1 (1:[])) "==" [])
    ,(eqTest (deleteNth 1 3 (3:1:3:[])) "==" (1:3:[]))
    ,(eqTest (deleteNth 2 3 (3:1:3:[])) "==" (3:1:[]))
    ,(eqTest (deleteNth 3 3 (3:1:3:[])) "==" (3:1:3:[]))
    ,(eqTest (deleteNth 1 3 (3:9:3:7:3:[])) "==" (9:3:7:3:[]))
    ,(eqTest (deleteNth 3 3 (3:9:3:7:3:[])) "==" (3:9:3:7:[]))
    ,(eqTest (deleteNth 2 1 (3:1:5:1:4:[])) "==" (3:1:5:4:[]))
    ,(eqTest (deleteNth 1 7 (3:1:[])) "==" (3:1:[]))
    ,(eqTest (deleteNth 1 7 [1,5,7,1,7]) "==" [1,5,1,7])
    ,(eqTest (deleteNth 2 7 [1,5,7,1,7]) "==" [1,5,7,1])
    ,(eqTest (deleteNth 4 9 [9,2,9,3,9,10,9,5,6]) "==" [9,2,9,3,9,10,5,6])
    ,(eqTest (deleteNth 2 8 [8,8,8,8,8,8]) "==" [8,8,8,8,8])
    ,(eqTest (deleteNth 17 8 [8,8,8,8,8,8]) "==" [8,8,8,8,8,8])
    ,(eqTest (deleteNth 18 99 [8,8,8,8,8,8]) "==" [8,8,8,8,8,8])
    ,(eqTest (deleteNth 3 8 [8,2,8,4,8,8,8,8,8]) "==" [8,2,8,4,8,8,8,8])
    ,(eqTest (deleteNth 2 20 ([1 .. 50] ++ (reverse [1 .. 50])))
      "==" ([1 .. 50] ++ (reverse ([1 .. 19] ++ [21 .. 50]))))
    ]
```

Figure 4: Tests for problem 5.

7. [UseModels] Complete the module `Vectors` found in the file `Vectors.hs` (provided in the `hw2-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 points) 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 points) 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 points) 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`, which is shown in Figure 5 on the next page.

   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 `hw2-tests.zip`.

   You can use `test_scale`, `test_add`, or `test_dot` to test individual functions. Then run all our tests by running the `main` function in `VectorsTests.hs`, and turn in both your code file and the output of our `main` test. (As usual, upload your code file to Webcourses, and paste the test output into the Comments box for the assignment corresponding to this problem.)

```
-- $Id: VectorsTests.hs,v 1.1 2015/01/21 18:43:36 leavens Exp $
module VectorsTests where
import Testing
import FloatTesting
import Vectors  -- you have to put your solutions in module Vectors
version = "VectorsTests $Revision: 1.1 $"
-- do main to run our tests
main :: IO()
main = do startTesting version
          errs_scale <- run_test_list 0 scale_tests
          errs_add <- run_test_list errs_scale add_tests
          total_errs <- run_test_list errs_add dot_tests
          doneTesting total_errs
-- The following will test one function each
test_scale, test_add, test_dot :: IO()
test_scale = dotests "Testing scale $Revision: 1.1 $" scale_tests
test_add = dotests "Testing add $Revision: 1.1 $" add_tests
test_dot = dotests "Testing dot $Revision: 1.1 $" dot_tests

scale_tests :: [TestCase Vector]
scale_tests =
    [(vecWithin (scale 3.14 []) "~=~" [])
    ,(vecWithin (scale 10.0 [1.0, 2.0, 4.0]) "~=~" [10.0, 20.0, 40.0])
    ,(vecWithin (scale 5.3 [1.0 .. 10.0]) "~=~" [5.3, 10.6 .. 53.0])
    ,(vecWithin (scale 2.0 [1.0 .. 100.0]) "~=~" [2.0, 4.0 .. 200.0])
    ,(vecWithin (scale 3.5 [4.0]) "~=~" [3.5*4.0])
    ]
add_tests :: [TestCase Vector]
add_tests =
    [(vecWithin ([] `add` []) "~=~" [])
    ,(vecWithin ([0.0, 100.0, 200.0] `add` [1.0, 2.0, 4.0])
                    "~=~" [1.0, 102.0, 204.0])
    ,(vecWithin ([1.0 .. 10.0] `add` [100.0 .. 109.0])
                    "~=~" [101.0, 103.0 .. 119.0])
    ,(vecWithin ([1.0 .. 10.0] `add` [11.0 .. 20.0])
                    "~=~" ([12.0, 14.0 .. 30.0]))
    ,(vecWithin ([3.5] `add` [10.0]) "~=~" [13.5])
    ,(vecWithin ([3.5,6.2,8.2,5.99] `add` [7.2,9.6,13.1,15.5]) "~=~"
                    [10.7,15.8,21.299999999999997,21.490000000000002])
    ,(vecWithin ([-1.0] `add` [40.20]) "~=~" [39.20])
    ]
dot_tests :: [TestCase Double]
dot_tests =
    [(withinTest ([] `dot` []) "~=~" 0.0)
    ,(withinTest ([0.0, 100.0, 200.0] `dot` [1.0, 2.0, 4.0])
                    "~=~" 1000.0)
    ,(withinTest ([1.0 .. 10.0] `dot` [100.0 .. 109.0])
                    "~=~" 5830.0)
    ,(withinTest ([1.0 .. 10.0] `dot` [11.0 .. 20.0])
                    "~=~" 935.0)
    ,(withinTest ([3.5] `dot` [4.7]) "~=~" (3.5*4.7))
    ,(withinTest ([3.5,1.0,10.1,599.25] `dot` [7.2,9.6,13.1,15.5]) "~=~" 9455.485)
    ,(withinTest ([-1.0] `dot` [40.20]) "~=~" (-40.2))
    ]
```

Figure 5: Tests for problem 7.

8. [UseModels] 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`:

```haskell
-- $Id: BinaryRelation.hs,v 1.2 2015/01/21 20:31:14 leavens Exp leavens $
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.

```haskell
module ApplyToList where
import BinaryRelation
```

The functions you are to write are the following.

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

```haskell
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 points) Using recursion (that is, without using a list comprehension or library functions), write the function

```haskell
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 the 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 first 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`, which is shown in Figure 6 on the following page. That file imports `Relations.hs`, which is shown in Figure 7 on page 11.

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

As specified on the first page of this homework, turn in both your code file and the output of your testing. (The code file should be uploaded to Webcourses, and the test output should be pasted in to the Comments box.)

```
-- $Id: ApplyToListTests.hs,v 1.1 2015/01/21 20:31:14 leavens Exp leavens $
module ApplyToListTests where
import Testing
import BinaryRelation
import Relations  -- some test inputs
import ApplyToList  -- you have to put your solutions in this module

version = "ApplyToListTests $Revision: 1.1 $"

-- do main to run our tests
main :: IO()
main = do startTesting version
          errs_wk <- run_test_list 0 applyRel_tests
          total_errs <- run_test_list errs_wk applyToList_tests
          doneTesting total_errs
-- do test_applyRel to test just applyRel
test_applyRel :: IO()
test_applyRel = dotests ("deleteWithValue " ++ version) applyRel_tests
-- do test_applyToList to test just applyToList
test_applyToList :: IO()
test_applyToList = dotests ("applyToList " ++ version) applyToList_tests

applyRel_tests :: [TestCase [String]]
applyRel_tests =
    [(eqTest (applyRel "Timbuktu" []) "==" [])
    ,(eqTest (applyRel "Ames" us_cities) "==" ["Iowa"])
    ,(eqTest (applyRel "Chicago" us_cities) "==" ["Illinois"])
    ,(eqTest (applyRel "bar" bar_stuff)
                 "==" ["mitzva", "stool", "tender", "keeper"])
    ,(eqTest (applyRel "salad" bar_stuff) "==" ["bar"])
    ,(eqTest (applyRel "foo" bar_stuff) "==" [])
    ]

applyToList_tests :: [TestCase [String]]
applyToList_tests =
    [(eqTest (applyToList ([]::[String]) ([]::(BinaryRelation String String))) "==" [])
    ,(eqTest (applyToList ["foo"] []) "==" [])
    ,(eqTest (applyToList ["foo","bar"] bar_stuff)
                 "==" ["mitzva", "stool", "tender", "keeper"])
    ,(eqTest (applyToList ["salad","bar"] bar_stuff)
                 "==" ["bar", "mitzva", "stool", "tender", "keeper"])
    ,(eqTest (applyToList ["Beijing", "Guangzhou", "Shenzhen", "Tokyo"] city_country)
                 "==" ["China", "China", "China", "Japan"])
    ,(eqTest (applyToList ["Buenos Aires", "Delhi", "Osaka"] city_country)
                 "==" ["Argentina", "India", "Japan"])
    ]
```

Figure 6: Tests for problem 8.

```haskell
-- $Id: Relations.hs,v 1.2 2013/08/21 21:11:27 leavens Exp $
module Relations where
import BinaryRelation
bar_stuff :: BinaryRelation String String
us_cities :: BinaryRelation String String
city_country :: BinaryRelation String String
city_population :: BinaryRelation String Int
city_areakm2 :: BinaryRelation String Int
country_population :: BinaryRelation String Int
bar_stuff = [("bar","mitzva"),("bar","stool"), ("bar","tender"),("salad","bar"),("bar","keeper")]
us_cities = [("Chicago","Illinois"),("Miami","Florida"),("Ames","Iowa")
            ,("Orlando","Florida"),("Des Moines","Iowa")]
-- The following data are from Wikipedia.org, accessed August 19, 2013
city_country =
    [("Beijing","China"),("Buenos Aires","Argentina"),("Cairo","Egypt"),("Delhi","India")
    ,("Dhaka","Bangladesh"),("Guangzhou","China"),("Istanbul","Turkey"),("Jakarta","Indonesia")
    ,("Karachi","Pakistan"),("Kinshasa","Democratic Republic of the Congo"),("Kolkata","India")
    ,("Lagos","Nigeria"),("Lima","Peru"),("London","United Kingdom"),("Los Angeles","United States")
    ,("Manila","Philippines"),("Mexico City","Mexico"),("Moscow","Russia"),("Mumbai","India")
    ,("New York City","United States"),("Osaka","Japan"),("Rio de Janeiro","Brazil")
    ,("Sao Paulo","Brazil"),("Seoul","South Korea"),("Shanghai","China"),("Shenzhen","China")
    ,("Tehran","Iran"),("Tianjin","China"),("Tokyo","Japan")]
city_population =
   [("Tokyo", 37239000),("Jakarta", 26746000),("Seoul", 22868000)
   ,("Delhi", 22826000),("Shanghai", 21766000),("Manila", 21241000)
   ,("Karachi", 20877000),("New York City", 20673000),("Sao Paulo", 20568000)
   ,("Mexico City", 20032000),("Beijing", 18241000),("Guangzhou", 17681000)
   ,("Mumbai", 17307000),("Osaka", 17175000),("Moscow", 15788000)
   ,("Cairo", 15071000),("Los Angeles", 15067000),("Kolkata", 14399000)
   ,("Buenos Aires", 13776000),("Tehran", 13309000),("Istanbul", 12506000)
   ,("Lagos", 12090000),("Rio", 10183000),("London", 9576000)
   ,("Lima", 9400000),("Kinshasa", 9387000),("Tianjin", 9277000)
   ,("Chennai", 9182000),("Chicago", 9104000),("Bengaluru", 9044000)
   ,("Bogota", 9009000)]
city_areakm2 = -- area is measured in square km
    [("Tokyo", 8547) ,("Jakarta", 2784) ,("Seoul", 2163)
    ,("Delhi", 1943) ,("Shanghai", 3497) ,("Manila", 1437)
    ,("Karachi", 803) ,("New York City", 11642) ,("Sao Paulo", 3173)
    ,("Mexico City", 2046) ,("Beijing", 3497) ,("Guangzhou", 3173)
    ,("Mumbai", 546) ,("Osaka", 3212) ,("Moscow", 4403)
    ,("Cairo", 1658) ,("Los Angeles", 6299) ,("Kolkata", 1204)
    ,("Bangkok", 2331) ,("Dhaka", 324) ,("Buenos Aires", 2642)
    ,("Tehran", 1360) ,("Istanbul", 1347) ,("Shenzhen", 1748)
    ,("Lagos", 907) ,("Rio de Janeiro", 2020) ,("Paris", 2845)
    ,("Nagoya", 3820) ,("London", 1623) ,("Lima", 648)
    ,("Kinshasa", 583) ,("Tianjin", 1684) ,("Chennai", 842)
    ,("Chicago", 6856) ,("Bengaluru", 738) ,("Bogota", 414)]
country_population = -- from Wikipedia.org access August 21, 2013
    [("China",1359470000),("India",1232830000),("United States",316497000),("Indonesia",237641326)
    ,("Brazil",193946886),("Pakistan",184013000),("Nigeria",173615000),("Bangladesh",152518015)
    ,("Russia",143400000),("Japan",127350000),("Mexico",117409830),("Philippines",98234000)]
```

Figure 7: Test data for relation problems, the file `Relations.hs`.

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

```
-- $Id: BinaryRelation.hs,v 1.2 2015/01/21 20:31:14 leavens Exp leavens $
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 points) 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 points) 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 points) 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 points) 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 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`, which is shown in Figure 8 on the following page. 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.

```
-- $Id: BinaryRelationOpsTests.hs,v 1.2 2013/08/22 19:59:54 leavens Exp leavens $
module BinaryRelationOpsTests where
import Testing; import BinaryRelation; import Relations
import BinaryRelationOps  -- you have to put your solutions in this module
version = "BinaryRelationOpsTests $Revision: 1.2 $"
main :: IO() -- do main to run all our tests
main = do startTesting version
          pj1_errs <- run_test_list 0 project1_tests
          pj2_errs <- run_test_list pj1_errs project2_tests
          select_errs <- run_test_list pj2_errs select_tests
          total_errs <- run_test_list select_errs compose_tests
          doneTesting total_errs
-- do test_f to test just the function named f
test_project1, test_project2, test_select, test_compose :: IO()
(test_project1, test_project2, test_select, test_compose) =
   (runts project1_tests, runts project2_tests, runts select_tests, runts compose_tests)
   where runts :: Show a => [TestCase [a]] -> IO() -- prevents type errors
         runts = dotests version
project1_tests :: [TestCase [String]]
project1_tests =
    [(eqTest (project1 []) "==" [])
    ,(eqTest (project1 bar_stuff) "==" ["bar", "bar", "bar", "salad", "bar"])
    ,(eqTest (project1 city_country)
      "==" ["Beijing","Buenos Aires","Cairo","Delhi","Dhaka","Guangzhou","Istanbul","Jakarta","Karachi"
           ,"Kinshasa","Kolkata","Lagos","Lima","London","Los Angeles","Manila","Mexico City","Moscow"
           ,"Mumbai","New York City","Osaka","Rio de Janeiro","Sao Paulo","Seoul","Shanghai"
           ,"Shenzhen","Tehran","Tianjin","Tokyo"])]
project2_tests :: [TestCase [String]]
project2_tests =
    [(eqTest (project2 []) "==" [])
    ,(eqTest (project2 bar_stuff) "==" ["mitzva", "stool", "tender", "bar", "keeper"])
    ,(eqTest (project2 city_country)
      "==" ["China","Argentina","Egypt","India","Bangladesh","China","Turkey","Indonesia","Pakistan"
           ,"Democratic Republic of the Congo","India","Nigeria","Peru","United Kingdom","United States"
           ,"Philippines","Mexico","Russia","India","United States","Japan","Brazil","Brazil"
           ,"South Korea","China","China","Iran","China","Japan"])]
select_tests :: [TestCase (BinaryRelation String String)]
select_tests =
    [(eqTest (select (\(x,y) -> length x > length y) []) "==" [])
    ,(eqTest (select (\(x,y) -> length x <= length y) us_cities)
      "==" [("Chicago","Illinois"),("Miami","Florida"),("Ames","Iowa"),("Orlando","Florida")])
    ,(eqTest (select (\(_,y) -> y == "Iowa") us_cities) "==" [("Ames","Iowa"),("Des Moines","Iowa")])
    ,(eqTest (select (\(x,y) -> x == "Tokyo" && y == "Japan") city_country) "==" [("Tokyo","Japan")])
    ,(eqTest (select (\(c:city,y:country) -> c == y) city_country)
      "==" [("Mexico City","Mexico"),("Seoul","South Korea")])]
compose_tests :: [TestCase (BinaryRelation String Int)]
compose_tests =
    [(eqTest (compose [] country_population) "==" [])
    ,(eqTest (compose bar_stuff [("stool",3),("tender",16)]) "==" [("bar",3),("bar",16)])
    ,(eqTest (compose city_country country_population)
      "==" [("Beijing",1359470000),("Delhi",1232830000),("Dhaka",152518015)
           ,("Guangzhou",1359470000),("Jakarta",237641326),("Karachi",184013000)
           ,("Kolkata",1232830000),("Lagos",173615000),("Los Angeles",316497000)
           ,("Manila",98234000),("Mexico City",117409830),("Moscow",143400000)
           ,("Mumbai",1232830000),("New York City",316497000),("Osaka",127350000)
           ,("Rio de Janeiro",193946886),("Sao Paulo",193946886),("Shanghai",1359470000)
           ,("Shenzhen",1359470000),("Tianjin",1359470000),("Tokyo",127350000)])]
```

Figure 8: Tests for problem 9. These tests use the relations defined in `Relations.hs` (see Figure 7 on page 11).

10. (5 points) [Concepts] [UseModels] 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,x,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. There are examples in Figure 9.

```
-- $Id: RotateTests.hs,v 1.2 2015/01/21 20:43:49 leavens Exp leavens $
module RotateTests where
import Testing
import Amount
import Rotate -- your code should go in this module
main = dotests "RotateTests $Revision: 1.2 $" tests
tests :: [TestCase Bool]
tests =
    [assertTrue ((rotate Zero (1,2,3,4)) == (1,2,3,4))
    ,assertTrue ((rotate One (1,2,3,4)) == (4,1,2,3))
    ,assertTrue ((rotate Two (1,2,3,4)) == (3,4,1,2))
    ,assertTrue ((rotate Three (1,2,3,4)) == (2,3,4,1))
    ,assertTrue ((rotate Two ("jan","feb","mar","apr")) == ("mar","apr","jan","feb"))
    ,assertTrue ((rotate Three ("jan","feb","mar","apr")) == ("feb","mar","apr","jan"))
    ,assertTrue ((rotate Zero (True,False,True,False)) == (True,False,True,False))
    ,assertTrue ((rotate One (True,False,True,False)) == (False,True,False,True)) ]
```

Figure 9: Tests for problem 10.

11. (10 points) [UseModels]

Write a function

```
hep :: [Word] -> [Word]
type Word = 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).

The examples in Figure 10 are written using the `Testing` module supplied with the homework. They r also found in our testing file `HepTests.hs` which u can get from webcourses (in the zip file attached to problem 1). Be sure to turn in both ur code and the output of our tests on webcourses.

```
-- $Id: HepTests.hs,v 1.2 2015/01/21 21:00:49 leavens Exp leavens $
module HepTests where
import Testing
import Hep

main = dotests "HepTests $Revision: 1.2 $" tests

tests :: [TestCase [String]]
tests =
    [(eqTest (hep []) "==" [])
    ,(eqTest (hep ["you","you","you","you"]) "==" ["u","u","u","u"])
    ,(eqTest (hep ["you","know","I","will","see","you","soon"])
      "==" ["u","know","I'll","cya","soon"])
    ,(eqTest (hep ["by","the","way","you","must","see","my","girlfriend","she","is","great"])
      "==" ["btw","u","must","see","my","gf","she","is","gr8"])
    ,(eqTest (hep (["for","your","information","you","are","a","pig"]
                   ++ ["see","you","later","when","you","find","me","a","boyfriend"]))
      "==" ["fyi","u","r","a","pig","cya","later","when","u","find","me","a","bf"])
    ,(eqTest (hep ["by","the","way","I","will","be","right","back"])
      "==" ["btw","I'll","brb"])
    ]
```

Figure 10: Tests for problem 11.

BTW, we will take some number of points off if u have repeated code in ur solution. U can avoid repeated code by using a helping function or a case-expression. A case-expression would be used in a larger expression to form the result list, like: **case** w **of** ....

Iteration

12. (10 points) [UseModels]

    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`, which is shown in Figure 11.

    Note: your solution *must use tail recursion*.

```
-- $Id: ListMinTests.hs,v 1.1 2013/08/22 18:09:10 leavens Exp $
module ListMinTests where
import Testing
import ListMin

main = do startTesting "ListMinTests $Revision: 1.1 $"
          errs <- run_test_list 0 tests_ints
          total <- run_test_list errs tests_chars
          doneTesting total

tests_ints :: [TestCase Int]
tests_ints =
    [(eqTest (listMin (1:1:1:1:1:[])) "==" 1)
    ,(eqTest (listMin (26:[])) "==" 26)
    ,(eqTest (listMin (1:[])) "==" 1)
    ,(eqTest (listMin (1:26:[])) "==" 1)
    ,(eqTest (listMin (26:1:[])) "==" 1)
    ,(eqTest (listMin (1:2:3:4:1:3:5:26:27:[])) "==" 1)
    ,(eqTest (listMin (4:0:2:0:[])) "==" 0)
    ,(eqTest (listMin (86:99:12:(-3):[])) "==" (-3))
    ,(eqTest (listMin (100000:8600000:12222:(-999999):[])) "==" (-999999))
    ]

tests_chars :: [TestCase Char]
tests_chars =
    [
     (eqTest (listMin "upqieurqoeiruazvzkpsau") "==" 'a')
    ,(eqTest (listMin "see haskell.org for more about Haskell") "==" ' ')
    ]
```

Figure 11: Tests for `listMin`.

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.

13. (10 points) [UseModels] 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`, which is shown in Figure 12.

```
-- $Id: WhatIndexTests.hs,v 1.1 2013/08/22 19:37:47 leavens Exp leavens $
module WhatIndexTests where
import WhatIndex
import Testing

main = dotests "WhatIndexTests $Revision: 1.1 $" tests

tests :: [TestCase Integer]
tests =
    [(eqTest (whatIndex 3 []) "==" (-1))
    ,(eqTest (whatIndex 2 [1,2,3,2,1]) "==" 1)
    ,(eqTest (whatIndex 'a' ['a' .. 'z']) "==" 0)
    ,(eqTest (whatIndex 'b' ['a' .. 'z']) "==" 1)
    ,(eqTest (whatIndex 'c' ['a' .. 'z']) "==" 2)
    ,(eqTest (whatIndex 'q' ['a' .. 'z']) "==" 16)
    ,(eqTest (whatIndex (41,'c') [(42,'c'),(43,'c'),(41,'c'),(3,'c')])
      "==" 2)
    ,(eqTest (whatIndex True [False,False,False]) "==" (-1))
    ,(eqTest (whatIndex True [False,False,False,True]) "==" 3)
    ,(eqTest (whatIndex True [True,False,False,False]) "==" 0)
    ,(eqTest (whatIndex True [True,True,True]) "==" 0)
    ,(eqTest (whatIndex 1000 [1 .. 4000]) "==" 999)
    ]
```

Figure 12: Tests for `WhatIndex`.

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.

Points

This homework's total points: 160.

References

[Lea13] Gary T. Leavens. Following the grammar with Haskell. Technical Report CS-TR-13-01, Dept. of EECS, University of Central Florida, Orlando, FL, 32816-2362, January 2013.

[Tho11] Simon Thompson. *Haskell: the craft of functional programming*. Addison-Wesley, Harlow, England, third edition, 2011.