Homework 2: Functional Programming in Haskell

See Webcourses2 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 you turn in problems you worked on in a Webcourses2 group.)
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 Webcourses2) 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.

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

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].

1. [Concepts]

   (a) (5 points) In Haskell, which of the following is equivalent to the list `[1,1,2]`?

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

   (b) (10 points) Suppose that `abba` is the list `['a', 'b', 'b', 'a']` and that `queen` is the list `"queen"`. 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. `abba ++ queen`
      2. `abba:'q'`
      3. `abba:queen`
      4. `'a':queen`
      5. `['a']:queen`

   (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.

```
dismember lst =
    let first = head lst
    in let rest = tail lst
       in (lst, first:rest)
```

   What is the result of the call `dismember [4,0,2,0]`?

   (d) (5 points) Supposed you implemented linked lists in C or Java, and you wrote functions corresponding to **`head`** and **`tail`** functions in that language. Would these functions mutate (i.e., dynamically change) the argument list passed to them when called?

2. [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 1 smaller than the corresponding element in the argument list.

   (a) (5 points) Write the function

   ```
   sub1_list_comp :: [Integer] -> [Integer]
   ```

   that solves the above problem by using a list comprehension.

   (b) (5 points) Write the function

   ```
   sub1_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 `Sub1ListTests.hs`, which is shown in Figure 1 on the following page. Our tests are written using the `Testing.lhs` file, which is included in hw2-tests.zip; this module is shown in Figure 2 on page 5 and Figure 3 on page 6.

   To run our tests, use the `Sub1ListTests.hs` file. To make that work, you have to put your code in a module `Sub1List`, which will need to be in a file named `Sub1List.hs` (or `Sub1List.lhs`), in the same directory as `Sub1ListTests.hs`. Your file `Sub1List.hs` should thus start as follows.

   ```
   module Sub1List where
   sub1_list_rec :: [Integer] -> [Integer]
   sub1_list_comp :: [Integer] -> [Integer]
   ```

   Then run our tests by running the `main` function in `Sub1ListTests.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 Webcourses2, and the test output should be pasted in to the Comments box for that assignment.)

```
-- $Id: Sub1ListTests.hs,v 1.1 2013/08/17 22:23:58 leavens Exp $
module Sub1ListTests where
import Testing
import Sub1List  -- you have to put your solutions in module Sub1List

version = "Sub1ListTests $Revision: 1.1 $"
recursive_tests = (tests sub1_list_rec)
comprehension_tests = (tests sub1_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 sub1_list_comp
test_comprehension :: IO()
test_comprehension = dotests version comprehension_tests

-- do test_recursive to test just sub1_list_rec
test_recursive :: IO()
test_recursive = dotests version recursive_tests

tests :: ([Integer] -> [Integer]) -> [TestCase [Integer]]
tests f =
    [(eqTest (f []) "==" [])
    ,(eqTest (f (1:[])) "==" (0:[]))
    ,(eqTest (f (3:1:[])) "==" (2:0:[]))
    ,(eqTest (f [1,5,7,1,7]) "==" [0,4,6,0,6])
    ,(eqTest (f [7 .. 21]) "==" [6 .. 20])
    ,(eqTest (f [8,4,-2,3,1,10000000,10])
          "==" [7,3,-3,2,0, 9999999, 9])
    ]
```

Figure 1: Tests for problem 2. In these tests `f` is one of your solutions.

```
$Id: Testing.lhs,v 1.4 2013/01/27 22:13:18 leavens Exp leavens $

> module Testing where

The type constructor TestCase is a representation of tests in Haskell.
To make a TestCase, you write, for example:
     eqTest (1 + 2) "==" 3
     gTest subset s1 "`subset`" s2
For eqTest, the first argument is the Haskell code for the test,
the second is printed as a connective (although == is used), and
the third is data that gives the expected result of the test.
For gTest, the first argument is a function,
which is used to compare the result of the result of the second argument
with the fourth argument (the expected result).
and the third argument is printed as a connective.

> data TestCase a =
>       Test (a -> a -> Bool) a String a

The following is convenient for making test cases that use equality.

> eqTest :: (Show a, Eq a) => a -> String -> a -> TestCase a
> eqTest = Test (==)

> gTest :: (Show a) => (a -> a -> Bool) -> a -> String -> a -> TestCase a
> gTest = Test

The following are for making assertions

> assertTrue :: Bool -> (TestCase Bool)
> assertTrue code = eqTest code "==" True

> assertFalse :: Bool -> (TestCase Bool)
> assertFalse code = eqTest code "==" False

For running a single test case, use the following.
The number returned is the number of test cases that failed.
For example, you can write
  run_test (eqTest (1 + 2) "==" 3)
    >>= (\i -> putStrLn ((show i) ++ " errors"))
and this will run the test.

> run_test :: (Show a) => TestCase a -> IO Integer
> run_test (Test comp code connective expected) =
>   do if result
>      then do { putStrLn (show code) }
>      else do { putStrLn (failure ++ (show code))}
>      putStr arrow
>      putStrLn (show expected)
>      return (if result then 0 else 1)
>    where failure = "FAILURE: "
>          arrow   = "      " ++ connective ++ " "
>          result = code `comp` expected
```

Figure 2: Part 1 of our testing module.

```
The following will run an entire list of tests.
For example, you can write
   run_tests [eqTest (1 + 2) "==" 3,
              eqTest (1 + 2) "==" 4]

> run_tests :: (Show a) => [TestCase a] -> IO ()
> run_tests ts =
>   do errs <- run_test_list 0 ts
>      doneTesting errs

A version of run_tests with more labeling.

> dotests :: (Show a) => String -> [TestCase a] -> IO ()
> dotests name ts =
>   do startTesting name
>      run_tests ts

> run_test_list :: (Show a) => Integer -> [TestCase a] -> IO Integer
> run_test_list errs_so_far [] =
>   do return errs_so_far
> run_test_list errs_so_far (t:ts) =
>   do err_count <- run_test t
>      run_test_list (errs_so_far + err_count) ts

To be able to create tests interactively, need an instance of Show for TestCase

> instance (Show a) => Show (TestCase a) where
>   show (Test _ _ connective expected) =
>             "(Test (" ++ connective ++ ") <code> \"" ++ connective ++ "\" "
>              ++ (show expected) ++ ")"

Print a newline and a message that testing is beginning.

> startTesting :: String -> IO ()
> startTesting name =
>    do putChar '\n'
>       putStrLn ("Testing " ++ name ++ "...")

> doneTesting :: Integer -> IO ()
> doneTesting fails =
>    do putStr "Finished with "
>       putStr (show fails)
>       putChar ' '
>       putStr (case fails of
>                   1 -> "failure!"
>                   _ -> "failures!")
>       putChar '\n'
```

Figure 3: Part 2 of our testing module.

3. (10 points) [UseModels] In Haskell, write the function:

   `squareEvens :: [Integer] -> [Integer]`

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

   There are examples in Figure 4.

```
-- $Id: SquareEvensTests.hs,v 1.1 2013/08/22 19:37:47 leavens Exp leavens $
module SquareEvensTests where
import SquareEvens
import Testing
main = dotests "SquareEvensTests $Revision: 1.1 $" tests
tests :: [TestCase [Integer]]
tests = [eqTest (squareEvens []) "==" []
        ,eqTest (squareEvens [3]) "==" [3]
        ,eqTest (squareEvens [4]) "==" [16]
        ,eqTest (squareEvens [4,3]) "==" [16,3]
        ,eqTest (squareEvens [1,2,3,4,5,6]) "==" [1,4,3,16,5,36]
        ,eqTest (squareEvens [3,22,3,95,600,0,-2]) "==" [3,484,3,95,360000,0,4]
        ]
```

Figure 4: Tests for problem 3.

4. [UseModels] Complete the module `Polynomials` found in the file `Polynomials.hs` (provided in the `hw2-tests.zip` file), by writing function definitions in the indicated places that implement the functions: `scaleBy`, `add`, and `sub`. This module represents Polynomials by lists of Doubles. A list such as `[9.0,12.0,4.0]` represents the polynomial $4x^2 + 12x + 9$. The `evaluate` function (which you do not need to implement), shows how the value of such a Polynomial is calculated. The functions you are to implement are as follows.

   (a) (5 points) The function

   ```
   scaleBy :: Double -> Polynomial -> Polynomial
   ```

   takes a Double `y`, and a polynomial, `p`, and returns a new Polynomial that is just like `p`, except that each coefficient is `y` times the corresponding coefficient in `p`.

   (b) (5 points) The function

   ```
   add :: Polynomial -> Polynomial -> Polynomial
   ```

   takes two polynomials and adds them together, so that each coefficient of the result is the sum of the corresponding coefficients of the argument Polynomials. Missing coefficients in one of the arguments are treated as `0.0`; thus the two arguments need not be of the same length.

   (c) (5 points) The function

   ```
   sub :: Polynomial -> Polynomial -> Polynomial
   ```

   takes two polynomials and subtracts the second argument from the first, so that each coefficient of the result is the difference between the corresponding coefficient of the first argument and the second. Again, missing coefficients in one of the arguments are treated as `0.0`.

   There are test cases contained in `PolynomialsTests.hs`, which is shown in Figure 5 on the following page.

   To run our tests, use the `PolynomialsTests.hs` file. To make that work, edit your code into the provided file `Polynomials.hs`. Our tests use the `FloatTesting` module shown in Figure 6 on page 10.

   You can use `test_scaleBy`, `test_add`, or `test_sub` to test individual functions. Then run all our tests by running the `main` function in `PolynomialsTests.hs`, and turn in both your code file and the output of our `main` test. (As usual, upload your code file to Webcourses2, and paste the test output into the Comments box for the assignment corresponding to this problem.)

```
-- $Id: PolynomialsTests.hs,v 1.1 2013/08/18 16:35:26 leavens Exp $
module PolynomialsTests where
import Testing
import FloatTesting
import Polynomials  -- you have to put your solutions in module Polynomials
version = "PolynomialsTests $Revision: 1.1 $"
-- do main to run our tests
main :: IO()
main = do startTesting version
          errs_scaleBy <- run_test_list 0 scaleBy_tests
          errs_add <- run_test_list errs_scaleBy add_tests
          total_errs <- run_test_list errs_add sub_tests
          doneTesting total_errs
-- The following will test one function each
test_scaleBy, test_add, test_sub :: IO()
test_scaleBy = dotests "Testing scaleBy $Revision: 1.1 $" scaleBy_tests
test_add = dotests "Testing add $Revision: 1.1 $" add_tests
test_sub = dotests "Testing sub $Revision: 1.1 $" sub_tests

scaleBy_tests :: [TestCase Polynomial]
scaleBy_tests =
    [(vecWithin (scaleBy 3.14 []) "~=~" [])
    ,(vecWithin (scaleBy 10.0 [1.0, 2.0, 4.0]) "~=~" [10.0, 20.0, 40.0])
    ,(vecWithin (scaleBy 5.3 [1.0 .. 10.0]) "~=~" [5.3, 10.6 .. 53.0])
    ,(vecWithin (scaleBy 2.0 [1.0 .. 100.0]) "~=~" [2.0, 4.0 .. 200.0])
    ,(vecWithin (scaleBy 3.5 [4.0]) "~=~" [3.5*4.0])
    ]
add_tests :: [TestCase Polynomial]
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` [1.0 .. 110.0])
                    "~=~" ([2.0, 4.0 .. 20.0] ++ [11.0 .. 110.0]))
    ,(vecWithin ([3.5] `add` []) "~=~" [3.5])
    ,(vecWithin ([3.5] `add` [7.2,9.6,13.1,15.5]) "~=~" [10.7,9.6,13.1,15.5])
    ,(vecWithin ([] `add` [40.20]) "~=~" [40.20])
    ]
sub_tests :: [TestCase Polynomial]
sub_tests =
    [(vecWithin ([] `sub` []) "~=~" [])
    ,(vecWithin ([0.0, 100.0, 200.0] `sub` [1.0, 2.0, 4.0])
                    "~=~" [-1.0, 98.0, 196.0])
    ,(vecWithin ([1.0 .. 10.0] `sub` [100.0 .. 109.0])
                    "~=~" (take 10 [-99.0, -99.0 ..]))
    ,(vecWithin ([1.0 .. 10.0] `sub` [1.0 .. 110.0])
                    "~=~" ((take 10 [0.0, 0.0 ..]) ++ [-11.0, -12.0 .. -110.0]))
    ,(vecWithin ([3.5] `sub` []) "~=~" [3.5])
    ,(vecWithin ([3.5] `sub` [7.2,9.6,13.1,15.5]) "~=~" [-3.7,-9.6,-13.1,-15.5])
    ,(vecWithin ([] `sub` [40.20]) "~=~" [-40.20])
    ]
```

Figure 5: Tests for problem 4.

```
-- $Id: FloatTesting.hs,v 1.3 2013/02/22 16:05:21 leavens Exp leavens $
module FloatTesting where
import Testing

withinMaker :: (RealFloat a) => a -> a -> a -> Bool
withinMaker eps x y = abs(x - y) < eps

relativeMaker eps x y = abs(x - y) < eps*abs(y)

(~=~) :: (RealFloat a, Tolerance a) => a -> a -> Bool
(~=~) = withinMaker hwTolerance
(~~~) :: (RealFloat a, Tolerance a) => a -> a -> Bool
(~~~) = relativeMaker hwTolerance

withinTest :: (Show a, RealFloat a, Tolerance a) =>
                a -> String -> a -> TestCase a
withinTest = gTest (~=~)

vecWithin :: (Show a, RealFloat a, Tolerance a) =>
                [a] -> String -> [a] -> TestCase [a]
vecWithin = gTest (\xs ys -> length xs == length ys
                             && all (uncurry (~=~)) (zip xs ys))

class (RealFloat a) => Tolerance a where
  hwTolerance :: a

instance Tolerance Float where
  hwTolerance = 1.0e-5

instance Tolerance Double where
  hwTolerance = 1.0e-9
```

Figure 6: Testing for problems using floating-point numbers.

5. [UseModels] This problem will have you write two functions that deal with deletion from binary relations. In this problem binary relations are represented as lists of pairs:

```
-- $Id: BinaryRelation.hs,v 1.1 2013/08/20 02:03:43 leavens Exp $
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."

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

```
deleteWithKey :: (Eq a) => a -> (BinaryRelation a b) -> (BinaryRelation a b)
```

When given a key value, `k`, of some equality type `a`, and a BinaryRelation `pairs`, the result is a BinaryRelation that is just like `pairs`, except that it does not contain any pair `(x,y)` such that `x == k`.

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

```
deleteWithValue :: (Eq b) => b -> (BinaryRelation a b) -> (BinaryRelation a b)
```

When given a value, `v`, of some equality type `b`, and a BinaryRelation `pairs`, the result is a BinaryRelation that is just like `pairs`, except that it does not contain any pair `(x,y)` such that `y == v`.

There are test cases contained in `DeleteFromRelationTests.hs`, which is shown in Figure 7 on the following page. That file imports `Relations.hs`, which is shown in Figure 8 on page 13.

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

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 Webcourses2, and the test output should be pasted in to the Comments box.)

```
-- $Id: DeleteFromRelationTests.hs,v 1.3 2013/08/21 21:11:27 leavens Exp $
module DeleteFromRelationTests where
import Testing
import BinaryRelation
import Relations
import DeleteFromRelation  -- you have to put your solutions in this module

version = "DeleteFromRelationTests $Revision: 1.3 $"

-- do main to run our tests
main :: IO()
main = do startTesting version
          errs_wk <- run_test_list 0 deleteWithKey_tests
          total_errs <- run_test_list errs_wk deleteWithValue_tests
          doneTesting total_errs
-- do test_deleteWithKey to test just deleteWithKey
test_deleteWithKey :: IO()
test_deleteWithKey = dotests ("deleteWithKey " ++ version) deleteWithKey_tests
-- do test_deleteWithValue to test just deleteWithValue
test_deleteWithValue :: IO()
test_deleteWithValue = dotests ("deleteWithValue " ++ version) deleteWithValue_tests

deleteWithKey_tests :: [TestCase (BinaryRelation String String)]
deleteWithKey_tests =
    [(eqTest (deleteWithKey "foo" []) "==" [])
    ,(eqTest (deleteWithKey "bar" [("bar", "mitzva"),("bar", "stool")])
       "==" [])
    ,(eqTest (deleteWithKey "bar" bar_stuff) "==" [("salad", "bar")])
    ,(eqTest (deleteWithKey "salad" bar_stuff)
       "==" [("bar", "mitzva"),("bar", "stool"), ("bar", "tender")
            ,("bar", "keeper")])
    ,(eqTest (deleteWithKey "TimbukTu" city_country) "==" city_country)
    ]
deleteWithValue_tests :: [TestCase (BinaryRelation String String)]
deleteWithValue_tests =
    [(eqTest (deleteWithValue "Japan" []) "==" [])
    ,(eqTest (deleteWithValue "Iowa" us_cities)
      "==" [("Chicago", "Illinois"),("Miami","Florida"),("Orlando", "Florida")])
    ,(eqTest (deleteWithValue "Illinois" us_cities)
      "==" [("Miami","Florida"),("Ames", "Iowa"),("Orlando", "Florida")
           ,("Des Moines", "Iowa")])
    ,(eqTest (deleteWithValue "island"
                              [("Coney", "island"),("Merritt","island")
                              ,("Curry Ford", "road"),("Orlando","Florida")
                              ,("Long","island")])
      "==" [("Curry Ford", "road"),("Orlando","Florida")])
    ,(eqTest (deleteWithValue "Vatican City" city_country) "==" city_country)
    ]
```

Figure 7: Tests for problem 5.

```
-- $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 8: Test data for relation problems, the file `Relations.hs`.

6. (5 points) [UseModels] Write the function

   ```
   deleteFirst :: (Eq a) => a -> [a] -> [a]
   ```

   that takes 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 first occurrence (in `as`) of the element `toDelete`.

   Your solution must *not* use any Haskell library functions.

   There are test cases contained in `DeleteFirstTests.hs`, which is shown in Figure 9.

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

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

tests :: [TestCase [Int]]
tests =
    [(eqTest (deleteFirst 3 []) "==" [])
    ,(eqTest (deleteFirst 3 (1:[])) "==" (1:[]))
    ,(eqTest (deleteFirst 1 (1:[])) "==" [])
    ,(eqTest (deleteFirst 3 (3:1:[])) "==" (1:[]))
    ,(eqTest (deleteFirst 3 (3:1:3:[])) "==" (1:3:[]))
    ,(eqTest (deleteFirst 3 (3:3:3:[])) "==" (3:3:[]))
    ,(eqTest (deleteFirst 1 (3:1:[])) "==" (3:[]))
    ,(eqTest (deleteFirst 1 (1:3:1:[])) "==" (3:1:[]))
    ,(eqTest (deleteFirst 7 (3:1:[])) "==" (3:1:[]))
    ,(eqTest (deleteFirst 7 [1,5,7,1,7]) "==" [1,5,1,7])
    ,(eqTest (deleteFirst 1 [1,5,7,1,7]) "==" [5,7,1,7])
    ,(eqTest (deleteFirst 8 [8,8,8,8,8,8]) "==" [8,8,8,8,8])
    ,(eqTest (deleteFirst 8 [8,2,8,8,8,8,8,8]) "==" [2,8,8,8,8,8,8])
    ,(eqTest (deleteFirst 20 ([1 .. 50] ++ (reverse [1 .. 50])))
      "==" ([1 .. 19] ++ [21 .. 50] ++ (reverse [1 .. 50])))
    ]
```

Figure 9: Tests for problem 6.

Hint, look at the test cases in Figure 9 carefully.

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.

7. (5 points) [Concepts] Is it possible to use a list comprehension to solve problem 6 in the same direct way you could use a list comprehension to solve part (a) of problem 5? Briefly explain.

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

```
-- $Id: BinaryRelation.hs,v 1.1 2013/08/20 02:03:43 leavens Exp $
module BinaryRelation where
-- Binary relations are represented as lists of pairs
type BinaryRelation a b = [(a,b)]
```

   (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 10 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.

```
-- $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 10: Tests for problem 8. These tests use the relations defined in `Relations.hs` (see Figure 8 on page 13).

9. (5 points) [Concepts] [UseModels] Consider the data type `Amount` defined below.

```
-- $Id: Amount.hs,v 1.1 2013/08/22 19:07:43 leavens Exp $
module Amount where
data Amount = Zero | One | Two
```

In Haskell, write the polymorphic function

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

which takes an Amount, `amt`, and a triple of elements of some type, `(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 `(x,y,z)` is returned unchanged; when `amt` is `One`, then `(z,x,y)` is returned; finally, when `amt` is `Two`, then `(y,z,x)` is returned. There are examples in Figure 11.

```
-- $Id: RotateTests.hs,v 1.1 2013/08/22 19:07:43 leavens Exp $
module RotateTests where
import Testing
import Amount
import Rotate -- your code should go in this module
main = dotests "RotateTests $Revision: 1.1 $" tests
tests :: [TestCase Bool]
tests =
    [assertTrue ((rotate Zero (1,2,3)) == (1,2,3))
    ,assertTrue ((rotate One (1,2,3)) == (3,1,2))
    ,assertTrue ((rotate Two (1,2,3)) == (2,3,1))
    ,assertTrue ((rotate Two ("jan","feb","mar")) == ("feb","mar","jan"))
    ,assertTrue ((rotate One ("jan","feb","mar")) == ("mar","jan","feb"))
    ,assertTrue ((rotate Zero (True,False,True)) == (True,False,True)) ]
```

Figure 11: Tests for problem 9.

---

10. (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`, and
    - `great` is replaced by `gr8`.

    This list is complete (for this problem).

    The examples in Figure 12 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 1.1 2013/08/22 19:37:47 leavens Exp leavens $
module HepTests where
import Testing
import Hep

main = dotests "HepTests $Revision: 1.1 $" 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","will","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","will","brb"])
    ]
```

Figure 12: Tests for problem 10.

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

11. (10 points) [UseModels]

    In Haskell, write a polymorphic function

    ```
    listMax :: (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 maximum element from `lst`. That is, the result should be an element of `lst` that is no greater than any other element of `lst`.

    Your code must *not* use any library functions.

    In your code, you can assume that the argument list is non-empty and finite. There are test cases contained in `ListMaxTests.hs`, which is shown in Figure 13.

    Hint: it may be useful to use tail recursion for this problem.

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

main = do startTesting "ListMaxTests $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 (listMax (1:1:1:1:1:[])) "==" 1)
    ,(eqTest (listMax (26:[])) "==" 26)
    ,(eqTest (listMax (1:[])) "==" 1)
    ,(eqTest (listMax (1:26:[])) "==" 26)
    ,(eqTest (listMax (26:1:[])) "==" 26)
    ,(eqTest (listMax (1:2:3:4:1:3:5:26:27:[])) "==" 27)
    ,(eqTest (listMax (4:0:2:0:[])) "==" 4)
    ,(eqTest (listMax (86:99:12: -3:[])) "==" 99)
    ,(eqTest (listMax (100000:8600000:12222: -999999:[])) "==" 8600000)
    ]

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

Figure 13: Tests for `listMax`.

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.

12. (10 points) [UseModels]

    In Haskell, 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 occurence of `sought` in `lst`. However, if `sought` does not occur in `lst`, it returns -1.

    Your code must not use any library functions.

    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 14.

    Hint: it may be useful to use tail recursion for this problem.

```
-- $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 14: 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.

Recursion over Grammars

In the following problems, you can use whatever Haskell library functions you wish.

13. (20 points) [UseModels] 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

`iconify :: WindowLayout -> WindowLayout`

that takes a ⟨WindowLayout⟩, `wl`, and returns a ⟨WindowLayout⟩ that is just like `wl`, except that in each ⟨Window⟩ record, the value of each `width` and `height` field is replaced by 2.

The file `IconifyTests.hs` contains examples, see Figure 15.

```
-- $Id: IconifyTests.hs,v 1.1 2013/08/22 19:28:18 leavens Exp $
module IconifyTests where
import Iconify
import WindowLayout
import Testing
main = dotests "IconifyTests $Revision: 1.1 $" tests
tests :: [TestCase WindowLayout]
tests =
    [eqTest (iconify Window {wname="castle", width=1280, height=740})
     "==" (Window {wname="castle", width=2, height=2})
    ,eqTest (iconify (Horizontal [Window {wname="castle", width=1280, height=740},
                                  Window {wname="bball", width=900, height=900}]))
     "==" (Horizontal [Window {wname="castle", width=2, height=2},
                       Window {wname="bball", width=2, height=2}])
    ,eqTest (iconify (Vertical [])) "==" (Vertical [])
    ,eqTest (iconify (Horizontal [])) "==" (Horizontal [])
    ,eqTest (iconify (Vertical [Horizontal [Window {wname="castle", width=1280, height=740},
                                            Window {wname="bball", width=900, height=900}],
                                Vertical [Window {wname="csi", width=1000, height=500}]]))
     "==" (Vertical [Horizontal [Window {wname="castle", width=2, height=2},
                                 Window {wname="bball", width=2, height=2}],
                     Vertical [Window {wname="csi", width=2, height=2}]])
    ,eqTest (iconify (Horizontal [Vertical [Window {wname="csi", width=1280, height=740},
                                            Window {wname="daily", width=900, height=900}],
                                  Horizontal [Window {wname="news", width=1000, height=500},
                                              Horizontal [Window {wname="pbs", width=800,height=400}]]]))
     "==" (Horizontal [Vertical [Window {wname="csi", width=2, height=2},
                                            Window {wname="daily", width=2, height=2}],
                                  Horizontal [Window {wname="news", width=2, height=2},
                                              Horizontal [Window {wname="pbs", width=2,height=2}]]]) ]
```

Figure 15: Tests for problem 13.

Be sure to follow the grammar!

14. (15 points) [UseModels]

    This is another problem about Window Layouts. Write a function

    ```
    changeChannel :: String -> String -> WindowLayout -> WindowLayout
    ```

    that takes two strings, `new` and `old` and a Window Layout, `wl`, and returns a Window Layout that is just like `wl`, except that all windows whose `wname` field's value is (== to) `old` in the argument `wl` are changed to `new` in the result.

    Figure 16 on the following page shows examples.

    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: ChangeChannelTests.hs,v 1.2 2013/08/22 19:59:54 leavens Exp leavens $
module ChangeChannelTests where
import WindowLayout
import qualified Data.Set as HSet
import ChangeChannel
import Testing
main = dotests "ChangeChannelTests $Revision: 1.2 $" tests
tests :: [TestCase WindowLayout]
tests =
 [(eqTest (changeChannel "goldfinger" "olympics"
                         (Window {wname = "olympics", width = 50, height = 33}))
   "==" (Window {wname = "goldfinger", width = 50, height = 33}))
 ,(eqTest (changeChannel "masterpiece" "dancing" (Horizontal [])) "==" (Horizontal []))
 ,(eqTest (changeChannel "nova" "Star Trek" (Vertical [])) "==" (Vertical []))
 ,(eqTest (changeChannel "masterpiece" "local news"
            (Horizontal [(Window {wname = "olympics", width = 80, height = 33})
                        ,(Window {wname = "local news", width = 20, height = 10})]))
   "==" (Horizontal [(Window {wname = "olympics", width = 80, height = 33})
                    ,(Window {wname = "masterpiece", width = 20, height = 10})]))
 ,(eqTest (changeChannel "Dr. No" "olympics"
           (Vertical [(Window {wname = "olympics", width = 80, height = 33})
                     ,(Window {wname = "local news", width = 20, height = 10})]))
   "==" (Vertical [(Window {wname = "Dr. No", width = 80, height = 33})
                  ,(Window {wname = "local news", width = 20, height = 10})]))
 ,(eqTest (changeChannel "Sienfeld" "local news"
           (Vertical [(Window {wname = "Star Trek", width = 40, height = 100})
                     ,(Window {wname = "olympics", width = 80, height = 33})
                     ,(Window {wname = "Sienfeld", width = 20, height = 10})]))
   "==" (Vertical [(Window {wname = "Star Trek", width = 40, height = 100})
                  ,(Window {wname = "olympics", width = 80, height = 33})
                  ,(Window {wname = "Sienfeld", width = 20, height = 10})]))
 ,(eqTest (changeChannel "Sienfeld" "local news"
         (Horizontal
            [(Vertical [(Window {wname = "Tempest", width = 200, height = 100})
                       ,(Window {wname = "Othello", width = 200, height = 77})
                       ,(Window {wname = "Hamlet", width = 1000, height = 600})])
            ,(Horizontal [(Window {wname = "baseball", width = 50, height = 40})
                         ,(Window {wname = "local news", width = 100, height = 60})
                         ,(Window {wname = "equestrian", width = 70, height = 30})])
            ,(Vertical [(Window {wname = "Star Trek", width = 40, height = 100})
                       ,(Window {wname = "olympics", width = 80, height = 33})
                       ,(Window {wname = "local news", width = 20, height = 10})])
            ]))
   "==" (Horizontal
            [(Vertical [(Window {wname = "Tempest", width = 200, height = 100})
                       ,(Window {wname = "Othello", width = 200, height = 77})
                       ,(Window {wname = "Hamlet", width = 1000, height = 600})])
            ,(Horizontal [(Window {wname = "baseball", width = 50, height = 40})
                         ,(Window {wname = "Sienfeld", width = 100, height = 60})
                         ,(Window {wname = "equestrian", width = 70, height = 30})])
            ,(Vertical [(Window {wname = "Star Trek", width = 40, height = 100})
                       ,(Window {wname = "olympics", width = 80, height = 33})
                       ,(Window {wname = "Sienfeld", width = 20, height = 10})])
            ])) ]
```

Figure 16: Tests for problem 14.

15. (25 points) [UseModels] Consider the data type of quantified Boolean expressions defined as follows, in the file `QBExp.hs`.

```
-- $Id: QBExp.hs,v 1.1 2013/08/22 19:28:18 leavens Exp $
module QBExp where
data QBExp = Varref String | QBExp `Or` QBExp | Exists String QBExp
```

Your task is to write a 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 *occurs as a free variable reference* if it occurs as a variable reference in a subexpression that is outside of any expression of the form `(Exists s e)`, which declares `s`.

In the examples given in Figure 17, 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).

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

```
-- $Id: FreeQBExpTests.hs,v 1.1 2013/08/22 19:28:18 leavens Exp leavens $
module FreeQBExpTests where
import QBExp
import FreeQBExp
import Testing
main = dotests "FreeQBExpTests $Revision: 1.1 $" tests
tests :: [TestCase [String]]
tests = [setEq (freeQBExp (Varref "x")) "==" ["x"]
        ,setEq (freeQBExp ((Varref "x") `Or` (Varref "y"))) "==" ["x","y"]
        ,setEq (freeQBExp ((Varref "y") `Or` (Varref "x"))) "==" ["y","x"]
        ,setEq (freeQBExp (((Varref "y") `Or` (Varref "x"))
                           `Or` ((Varref "x") `Or` (Varref "y"))))
                    "==" ["y","x"]
        ,setEq (freeQBExp (Exists "y" (Varref "y"))) "==" []
        ,setEq (freeQBExp (Exists "y" ((Varref "y") `Or` (Varref "z"))))
                    "==" ["z"]
        ,setEq (freeQBExp (Exists "z" (Exists "y" ((Varref "y") `Or` (Varref "z")))))
                    "==" []
        ,setEq (freeQBExp ((Varref "z")
                           `Or` (Exists "z" (Exists "y" ((Varref "y") `Or` (Varref "z"))))))
                    "==" ["z"]
        ,setEq (freeQBExp (((Varref "z") `Or` (Varref "q"))
                           `Or` (Exists "z" (Exists "y" ((Varref "y") `Or` (Varref "z"))))))
                    "==" ["z","q"]  ]
    where setEq = gTest setEqual
          setEqual los1 los2 = (length los1) == (length los2)
                               && subseteq los1 los2
          subseteq los1 los2 = all (\e -> e `elem` los2) los1
```

Figure 17: Tests for problem 15.

---

16. (10 points) [UseModels]

    The problem uses the types `Statement` and `Expression` [Lea13], shown below.

```
-- $Id: StatementsExpressions.hs,v 1.1 2013/08/22 19:37:47 leavens Exp leavens $
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

```
eqOptim :: Statement -> Statement
```

    that takes a Statement, `stmt`, and returns a Statement just like `stmt`, except that all expressions of the form `EqualsExp (VarExp` x`) (VarExp` x`))` that occur in `stmt` are replaced by `(VarExp "true")`.

    There are test cases contained in `EqOptimTests.hs`, which is shown in Figure 18 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.

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

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

tests :: [TestCase Statement]
tests =
 [(eqTest (eqOptim (ExpStmt (EqualsExp (VarExp "x") (VarExp "x"))))
          "==" (ExpStmt (VarExp "true")))
 ,(eqTest (eqOptim (ExpStmt (EqualsExp (VarExp "x") (VarExp "yy"))))
          "==" (ExpStmt (EqualsExp (VarExp "x") (VarExp "yy"))))
 ,(eqTest (eqOptim (ExpStmt (NumExp 7))) "==" (ExpStmt (NumExp 7)))
 ,(eqTest (eqOptim (ExpStmt (VarExp "q"))) "==" (ExpStmt (VarExp "q")))
 ,(eqTest (eqOptim (ExpStmt (VarExp "true"))) "==" (ExpStmt (VarExp "true")))
 ,(eqTest (eqOptim (ExpStmt (BeginExp [] (EqualsExp (VarExp "x") (VarExp "x")))))
   "==" (ExpStmt (BeginExp [] (VarExp "true"))))
 ,(eqTest (eqOptim (AssignStmt "y" (EqualsExp (VarExp "jz") (VarExp "jz"))))
   "==" (AssignStmt "y" (VarExp "true")))
 ,(eqTest (eqOptim (IfStmt (EqualsExp (VarExp "c") (VarExp "c"))
                           (AssignStmt "d" (EqualsExp (VarExp "f") (VarExp "f")))))
   "==" (IfStmt (VarExp "true") (AssignStmt "d" (VarExp "true"))))
 ,(eqTest (eqOptim (AssignStmt "g"
                    (BeginExp [(IfStmt (EqualsExp (VarExp "c") (VarExp "c"))
                                       (AssignStmt "d" (EqualsExp (VarExp "f") (VarExp "f"))))
                              ,(AssignStmt "z" (EqualsExp (VarExp "m") (VarExp "m")))]
                     (BeginExp [AssignStmt "e" (EqualsExp (VarExp "y") (NumExp 2))]
                               (EqualsExp (VarExp "a") (VarExp "a"))))))
   "==" (AssignStmt "g"
         (BeginExp [(IfStmt (VarExp "true") (AssignStmt "d" (VarExp "true")))
                   ,(AssignStmt "z" (VarExp "true"))]
          (BeginExp [AssignStmt "e" (EqualsExp (VarExp "y") (NumExp 2))]
                    (VarExp "true")))))
 ]
```

Figure 18: Tests for `EqOptim`.

17. (20 points) [UseModels]

    The problem uses the types `Statement` and `Expression` [Lea13], as in the previous problem.

    Write a function

    ```
    simplify :: 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 `SimplifyTests.hs`, which is shown in Figure 19 on the next 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.

Points

This homework's total points: 245.

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.

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

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

tests :: [TestCase Statement]
tests =
 [(eqTest (simplify (IfStmt (VarExp "true") (ExpStmt (NumExp 7))))
          "==" (ExpStmt (NumExp 7)))
 ,(eqTest (simplify (ExpStmt (BeginExp [] (NumExp 6))))
          "==" (ExpStmt (NumExp 6)))
 ,(eqTest (simplify (ExpStmt (NumExp 7))) "==" (ExpStmt (NumExp 7)))
 ,(eqTest (simplify (ExpStmt (VarExp "q"))) "==" (ExpStmt (VarExp "q")))
 ,(eqTest (simplify (ExpStmt (VarExp "true"))) "==" (ExpStmt (VarExp "true")))
 ,(eqTest (simplify (ExpStmt (BeginExp [] (EqualsExp (VarExp "x") (VarExp "x")))))
   "==" (ExpStmt (EqualsExp (VarExp "x") (VarExp "x"))))
 ,(eqTest (simplify (AssignStmt "y" (EqualsExp (VarExp "jz") (VarExp "jz"))))
   "==" (AssignStmt "y" (EqualsExp (VarExp "jz") (VarExp "jz"))))
 ,(eqTest (simplify (IfStmt (VarExp "true")
                    (AssignStmt "d" (VarExp "true"))))
   "==" (AssignStmt "d" (VarExp "true")))
 ,(eqTest (simplify
           (AssignStmt "g"
            (BeginExp [(IfStmt (VarExp "true")
                               (AssignStmt "d" (BeginExp [] (VarExp "true"))))
                       ,(AssignStmt "z" (EqualsExp (VarExp "m")
                                                   (BeginExp [] (VarExp "m"))))]
             (BeginExp [AssignStmt "e" (EqualsExp (VarExp "y") (NumExp 2))
                        ,(IfStmt (VarExp "true") (ExpStmt (NumExp 3)))]
                        (BeginExp [(IfStmt (VarExp "true")) (ExpStmt (NumExp 1))]
                                  (VarExp "true"))))))
   "==" (AssignStmt "g"
         (BeginExp [(AssignStmt "d" (VarExp "true"))
                   ,(AssignStmt "z" (EqualsExp (VarExp "m") (VarExp "m")))]
          (BeginExp [AssignStmt "e" (EqualsExp (VarExp "y") (NumExp 2))
                     ,(ExpStmt (NumExp 3))]
                     (BeginExp [(ExpStmt (NumExp 1))] (VarExp "true"))))))
 ]
```

Figure 19: Tests for `Simplify`.