Homework 3: Higher-Order Functional Programming

See Webcourses and the syllabus for due dates.

Purpose

In this homework you will learn more advanced techniques of functional programming such as using higher-order functions to abstracting from programming patterns, and using higher-order functions to model infinite 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 other sources.
For coding problems, 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@UCF) 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 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 10-14 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. See also the course code examples page (and the course resources page).

Problems

Combinations of Previous Techniques

1. (20 points) [UseModels] In various contests the contestants are awarded places based on some score, and a list of winners is produced. For example, ebird.org maintains lists of the top 100 birders in Florida this year. In such ranked lists, contestants that have the same score are considered tied; for example, if Josh and Darcy have both seen 295 bird species this year, then they are considered tied, and both are listed as being in (say) fourth place. In this scenario, the next birder, with 294 species, is listed as being in sixth place, as Josh and Darcy take places 4 and 5 together, even though they are listed as tied for fourth place.

   In this problem you will write an general ranking function

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

   which for any type `a` that is an instance of the `Ord` class, takes a list of elements of type `a`, `things`, and returns a list of pairs of `Int`s and `a` elements. The result is sorted (in non-decreasing order) on the `a` elements of `things`, and the `Int` in each pair is the rank of the element in the pair. There are test cases contained in the file `RankTests.hs`, which is shown in Figure 1 on the following page. To run our tests, use the `RankTests.hs` file. To make that work, you have to put your code in a module `Rank`, which will need to be in a file named `Rank.hs` (or `Rank.lhs`), in the same directory as `RankTests.hs` and `Testing.lhs`.

   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.)

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

```
-- $Id: RankTests.hs,v 1.4 2013/09/24 12:40:12 leavens Exp leavens $
module RankTests where
import Rank
import Testing
main :: IO ()
main = dotests2 "$Revision: 1.4 $" testsString testsBirders
testsString :: [TestCase [(Int, String)]]
testsString =
    [(eqTest (rank []) "==" [])
    ,(eqTest (rank ["one"]) "==" [(1,"one")])
    ,(eqTest (rank ["one","one"]) "==" [(1,"one"),(1,"one")])
    ,(eqTest (rank ["two","one","one"]) "==" [(1,"one"),(1,"one"),(3,"two")])
    ,(eqTest (rank ["abel", "charlie", "baker", "abel", "charlie", "delta", "echo"])
      "==" [(1,"abel"), (1,"abel"), (3,"baker"),
            (4,"charlie"), (4,"charlie"), (6,"delta"), (7,"echo")])
     ,(eqTest (rank ["baker", "baker", "abel", "baker", "baker"])
       "==" [(1,"abel"),(2,"baker"),(2,"baker"),(2,"baker"),(2,"baker")])
    ]
data Birder = Person String Int deriving (Eq, Show)
-- The following Ord instance makes the person with the highest count least
instance Ord Birder where
  (Person _ count1) < (Person _ count2) = (count1 > count2) -- yes, backwards!
  compare (Person _ count1) (Person _ count2) = compare count2 count1
flBirders :: [Birder]
flBirders =  -- data from ebird.org 9/23/13
    [(Person "Audrey" 305),(Person "Graham" 319),(Person "John" 293)
    ,(Person "Scott" 269),(Person "Ron" 269),(Person "Tom" 267),(Person "Thomas" 225)
    ,(Person "Steven & Darcy" 295),(Person "David" 294),(Person "Chris" 312)
    ,(Person "Rangel" 281),(Person "Charles" 280),(Person "Andy" 276)
    ,(Person "Angel & Mariel" 274),(Person "Mark" 273),(Person "Kevin" 270)
    ,(Person "josh" 295),(Person "Jonathan" 290),(Person "adam" 286)
    ,(Person "Gary" 223),(Person "Brian" 257),(Person "Janet" 256)
    ,(Person "Michael" 266),(Person "Steven" 263),(Person "Eric" 261)
    ,(Person "Nancy" 223),(Person "Carlos" 224),(Person "Peter" 225)
    ]
testsBirders :: [TestCase [(Int, Birder)]]
testsBirders =
    [(eqTest (rank []) "==" [])
    ,(eqTest (rank [(Person "Tom" 532),(Person "Pat" 532)]) "=="
                [(1,(Person "Tom" 532)),(1, (Person "Pat" 532))])
    ,(eqTest (rank [(Person "Pat" 532),(Person "Tom" 532)]) "=="
                [(1,(Person "Pat" 532)),(1, (Person "Tom" 532))])
    ,(eqTest (rank [(Person "Pat" 532),(Person "Tom" 532),(Person "Neil" 703)])
      "==" [(1,(Person "Neil" 703)),(2,(Person "Pat" 532)),(2, (Person "Tom" 532))])
    ,(eqTest (rank flBirders)
      "==" [(1,Person "Graham" 319),(2,Person "Chris" 312),(3,Person "Audrey" 305)
           ,(4,Person "Steven & Darcy" 295),(4,Person "josh" 295),(6,Person "David" 294)
           ,(7,Person "John" 293),(8,Person "Jonathan" 290),(9,Person "adam" 286)
           ,(10,Person "Rangel" 281),(11,Person "Charles" 280),(12,Person "Andy" 276)
           ,(13,Person "Angel & Mariel" 274),(14,Person "Mark" 273),(15,Person "Kevin" 270)
           ,(16,Person "Scott" 269),(16,Person "Ron" 269),(18,Person "Tom" 267)
           ,(19,Person "Michael" 266),(20,Person "Steven" 263),(21,Person "Eric" 261)
           ,(22,Person "Brian" 257),(23,Person "Janet" 256),(24,Person "Thomas" 225)
           ,(24,Person "Peter" 225),(26,Person "Carlos" 224)
           ,(27,Person "Gary" 223),(27,Person "Nancy" 223)])
    ]
```

Figure 1: Tests for problem 1.

2. [UseModels] This two-part question deals with Artificial Neural Networks (ANNs).

   (a) (10 points) An artificial neuron that can process a vector of n inputs can be represented by a list of n weights, where each weight is a Double.

   ```
   type Neuron = [Weight]
   type Vector = [Double]
   type Weight = Double
   ```

   In this part you will write a function

   ```
   applyNeuron :: Neuron -> Vector -> Double
   ```

   that takes a `Neuron`, which is represented as a list of weights `[`$w_1, \ldots, w_n$`]`, and an input vector `[`$i_1, \ldots, i_n$`]` of the same length, and returns $\tanh(\Sigma_{j=1}^{n} w_j \cdot i_j)$, where $\tanh$ is the hyperbolic tangent function. (You can use the built-in Haskell function **`tanh`** to compute $\tanh$.)

   (b) (20 points) An artificial neural network can be represented as a list of lists of Neurons.

   ```
   type NeuralNetwork = [NeuralLayer]
   type NeuralLayer = [Neuron]
   ```

   A *properly formed neural network of input size* m has the property that if we number its layers from 1 to n, then a layer $j > 1$ is such that it takes an input from each of the neurons in layer $j - 1$, and thus each neuron in layer j has ℓ_{j-1} weights, where ℓ_{j-1} is the number of neurons in layer $j - 1$. The neurons in the first layer each have m weights. (For example, m would be 3 if the inputs are RGB tuples in a pixel.) Properly formed neural networks are constructed by `contructNN`, which is found in Figure 2 on the next page.

   In this part you will write a function

   ```
   applyNetwork :: NeuralNetwork -> Vector -> Vector
   ```

   which takes a properly formed neural network of input size m, `nn`, and an input vector, `iv`, of size m and returns a vector of outputs from the last layer of `nn` that is the result of applying each neuron to the vector of the outputs of the neurons in the previous layer. Let us again number the layers from 1 to n. Then the output from the first layer of `nn` has as its ith element the result of `(applyNeuron ni iv)`, where `ni` is the ith neuron in the first layer. The output of the jth layer is computed in the same way, using the output of the $j - 1$st layer instead of the input vector. Finally, the result of the whole computation is the result of the last layer.

There are test cases contained in the file `ANNTests.hs`, which is shown in Figure 2 on the following page. (These tests use functions imported from the module `FloatTesting`, which we saw in the previous homework, and which is included in the testing files.) To run our tests, use the `ANNTests.hs` file. To make that work, you have to put your code in a module `ANN`, which will need to be in a file named `ANN.hs` (or `ANN.lhs`), in the same directory as `ANNTests.hs`, `Testing.lhs`, and `FloatTesting.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.)

```
-- $Id: ANNTests.hs,v 1.6 2013/09/25 11:00:01 leavens Exp leavens $
module ANNTests where
import ANN
import Testing
import FloatTesting
main = dotests2 "ANNTests $Revision: 1.6 $" ntests nntests
test_applyNeuron = dotests "Testing applyNeuron $Revision: 1.6 $" ntests
test_applyNetwork = dotests "Testing applyNetwork $Revision: 1.6 $" nntests
-- construction of Neural Networks, for testing, not for you to implement
af = tanh -- hyperbolic tangent is the activation function
initWeight = 1.0 -- initial weight
type Pattern = [Integer] -- used for construction
constructNN :: Pattern -> Integer -> NeuralNetwork
constructNN pattern m = createNN m pattern []
  where createNN :: Integer -> Pattern -> NeuralNetwork -> NeuralNetwork
        createNN _ [] acc = reverse acc
        createNN m (len:densities) acc =
            createNN len densities
                        ([[initWeight | _<-[1..m]] | _<- [1..len]]:acc)
-- data for testing below
nn321 = constructNN [3,2,1] 3
demo = [[[1.0,1.0,1.0,1.0],[2.0,2.0,2.0,2.0],[3.0,3.0,3.0,3.0]]]
demo1 = [[[1.0,1.0,1.0],[2.0,2.0,2.0],[3.0,3.0,3.0]]]
demo2 = [[[1.0,1.0,1.0],[2.0,2.0,2.0],[3.0,3.0,3.0]],
         [[4.0,4.0,4.0],[5.0,5.0,5.0]]]
iv5678 = [5.0,6.0,7.0,8.0]
iv567 = [5.0,6.0,7.0]
ntests :: [TestCase Double]
ntests = [withinTest (applyNeuron [1.0,1.0] [0.0,0.0])
          "~=~" (af (1.0*0.0 + 1.0*0.0))
         ,withinTest (applyNeuron [1.05,1.0,0.95] [3.0,4.0,5.0])
          "~=~" (af (1.05*3.0 + 1.0*4.0 + 0.95*5.0))
         ,withinTest (applyNeuron [1.05,2.0,0.95] [3.0,4.0,5.0])
          "~=~" (af (1.05*3.0 + 2.0*4.0 + 0.95*5.0))
         ,withinTest (applyNeuron [1.0,1.0,1.0,1.0] [1.0,1.0,1.0,1.0])
          "~=~" (af 4.0) ]
nntests :: [TestCase [Double]]
nntests =
    [vecWithin (applyNetwork demo iv5678)
     "~=~" [applyNeuron (demo!!0!!0) iv5678
           ,applyNeuron (demo!!0!!1) iv5678
           ,applyNeuron (demo!!0!!2) iv5678]
    ,vecWithin (applyNetwork demo1 iv567)
     "~=~" [applyNeuron (demo1!!0!!0) iv567
           ,applyNeuron (demo1!!0!!1) iv567
           ,applyNeuron (demo1!!0!!2) iv567]
    ,vecWithin (applyNetwork demo2 iv567)
     "~=~" (let layer1res = [applyNeuron (demo2!!0!!0) iv567
                            ,applyNeuron (demo2!!0!!1) iv567
                            ,applyNeuron (demo2!!0!!2) iv567]
            in [applyNeuron (demo2!!1!!0) layer1res
               ,applyNeuron (demo2!!1!!1) layer1res])
    ,vecWithin (applyNetwork nn321 [4.3,2.1,0.5]) "~=~" [0.9633220964607272]
    ,vecWithin (applyNetwork nn321 [0.0,0.0,0.0]) "~=~" [0.0]
    ,vecWithin (applyNetwork nn321 [1.0,1.0,1.0]) "~=~" [0.9633007043762163]
    ,vecWithin (applyNetwork nn321 [10.0,10.0,10.0]) "~=~" [0.9633221051195399]
    ,vecWithin (applyNetwork nn321 [-1.0,-1.0,-1.0]) "~=~" [-0.9633007043762163] ]
```

Figure 2: Tests for problem 2.

Higher-Order Functions

These problems are intended to give you an idea of how to use and write higher-order functions.

3. (5 points) [UseModels] In cryptography, one would like to apply functions defined over the type `Int` to data of type `Char`. However, in Haskell, these two types are distinct. In Haskell, write a function

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

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

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

   There are test cases contained in the file `ToCharFunTests.hs`, which is shown in Figure 3 on the next page.

   To run our tests, use the `ToCharFunTests.hs` file. To make that work, you have to put your code in a module `ToCharFun`, which will need to be in a file named `ToCharFun.hs` (or `ToCharFun.lhs`), in the same directory as `ToCharFunTests.hs` and `Testing.lhs`.

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

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

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

   There are test cases contained in the file `MapInsideTests.hs`, which is shown in Figure 4 on page 8.

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

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

```
-- $Id: ToCharFunTests.hs,v 1.1 2013/02/11 02:51:06 leavens Exp leavens $
module ToCharFunTests where
import ToCharFun  -- your solution goes in this module
import Testing

main = dotests "ToCharFunTests $Revision: 1.1 $" tests
tests :: [TestCase Char]
tests = [eqTest (toCharFun (+3) 'a') "==" 'd'
        ,eqTest (toCharFun (+1) 'b') "==" 'c'
        ,eqTest (toCharFun (+7) 'a') "==" 'h'
        ,eqTest (toCharFun (\c -> 10*c `div` 12) 'h') "==" 'V'
        ]
```

Figure 3: Tests for problem 3.

```
-- $Id: MapInsideTests.hs,v 1.2 2013/09/24 18:55:42 leavens Exp leavens $
module MapInsideTests where
import Testing
import ToCharFun -- used for testing
import MapInside  -- you have to put your solutions in module MapInside

version = "MapInsideTests $Revision: 1.2 $"

-- do main to run our tests
main :: IO()
main = do startTesting version
          errs_ints <- run_test_list 0 int_tests
          total_errs <- run_test_list errs_ints string_tests
          doneTesting total_errs

int_tests :: [TestCase [[Int]]]
int_tests =
    [eqTest (mapInside (+1) []) "==" []
    ,eqTest (mapInside (+1) [[]]) "==" [[]]
    ,eqTest (mapInside (*2) [[3,4,5],[4,0,2,0],[],[8,7,6]])
      "==" [[6,8,10],[8,0,4,0],[],[16,14,12]]
    ,eqTest (mapInside (*2) [[1 .. 10], [2,4 .. 20], [7]])
      "==" [[2,4 .. 20], [4,8 .. 40], [14]]
    ,eqTest (mapInside (3*) [[0 .. 10], [0,2 .. 10], [7]])
      "==" [[0,3 .. 30], [0,6 .. 30], [21]]
    ,eqTest (mapInside (\n -> 3*n + 1) [[0,7,17,27], [94,5]])
      "==" [[1,22,52,82],[283,16]]
    ]

string_tests :: [TestCase [[Char]]]
string_tests =
    [eqTest (mapInside (toCharFun (+1)) ["A string", "is a list!"])
            "==" ["B!tusjoh","jt!b!mjtu\""]
    ,eqTest (mapInside (toCharFun (\x -> x-7)) ["UCF","CS","is","great"])
            "==" ["N<?","<L","bl","`k^Zm"]
    ,eqTest (mapInside (toCharFun (+7)) ["N<?","<L","bl","`k^Zm"])
            "==" ["UCF","CS","is","great"]
     ]
```

Figure 4: Tests for problem 4.

Functions as Data and Abstract Data Types

5. (15 points) [UseModels] Write a function

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

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

   Hint: note that `compose []` is the identity function.

   There are test cases contained in the file `ComposeListTests.hs`, which is shown in Figure 5.

```
-- $Id: ComposeListTests.hs,v 1.2 2013/09/24 18:55:42 leavens Exp leavens $
module ComposeListTests where
import ComposeList
import Testing

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

tests :: [TestCase Bool]
tests =
    [assertTrue ((composeList [] [1,2,3]) == [1,2,3])
    ,assertTrue ((composeList [(*5),(+2)] 4) == 30)
    ,assertTrue ((composeList [tail,tail,tail] [1,2,3,4,5]) == [4, 5])
    ,assertTrue ((composeList [(3*),(4+),(10*)] 1) == (3*(4+10)))
    ,assertTrue ((composeList [(\x -> 3:x),(\y -> 4:y)] []) == 3:(4:[]))
    ,assertTrue ((composeList [(\x -> 'a':x),(\y -> ' ':y)] "star") == "a star")
    ,assertTrue ((composeList (map (+) [1 .. 1000]) 0) == (sum [1 .. 1000]))
    ]
```

Figure 5: Tests for problem 5.

---

6. (30 points) [UseModels] In this problem you will write operations for an abstract data type `Matrix`, by writing definitions for the module `Matrix` that completes the module definition in Figure 6.

```
module Matrix (Matrix, fillWith, fromRule, numRows, numColumns,
               at, mtranspose, mmap) where

newtype Matrix a = Mat ((Int,Int), (Int,Int) -> a)

fillWith :: (Int,Int) -> a -> (Matrix a)
fromRule :: (Int,Int) -> ((Int,Int) -> a) -> (Matrix a)
numRows :: (Matrix a) -> Int
numColumns :: (Matrix a) -> Int
at :: (Matrix a) -> (Int, Int) -> a
mtranspose :: (Matrix a) -> (Matrix a)
mmap :: (a -> b) -> (Matrix a) -> (Matrix b)
```

Figure 6: Start of the module `Matrix`.

That is, you are to complete the module by writing a function definition for each function declared in the module. To explain these, note that an $m \times n$ matrix is one with m rows and n columns. Element indexes range from 1 to the number of rows or columns (unlike the convention in C or Haskell). With that convention you are to implement the following functions:

- `fillWith` takes a pair (m, n) and an element e and produces an $m \times n$ matrix each of whose elements are e.
- `fromRule` takes a pair (m, n) and a function g (the rule), and produces an $m \times n$ matrix whose (i, j)th element is $g(i, j)$.
- `numRows` takes an $m \times n$ matrix and returns the number of rows in the matrix, m.
- `numColumns` takes an $m \times n$ matrix and returns the number of columns in the matrix, n.
- `at` takes an $m \times n$ matrix and a pair of `Int`s, (i, j), and returns the (i, j)th element of the matrix, provided that $1 \leq i \leq m$ and $1 \leq j \leq n$. If either index is outside of those ranges, an error occurs (at runtime).
- `mtranspose` takes an $m \times n$ matrix and returns an $n \times m$ matrix where the (i, j)th element of the result is the (j, i)th element of the argument matrix.
- `mmap` takes an $m \times n$ matrix and a function f and returns an $m \times n$ matrix whose (i, j)th element is the result of applying f to the (i, j)th element of the argument matrix.

There are test cases contained in `MatrixTests.hs`, which is shown in Figure 7 on the next page.

To aid in testing, we have also provided code to make `Matrix` an instance of the Haskell type classes `Show` and `Eq`. These instances are found in the file `MatrixInstances.hs`.

To make the tests work, you have to put your code in a module named `Matrix`. As specified on the first page of this homework, turn in both your code file and the output of your testing.

```
-- $Id: MatrixTests.hs,v 1.2 2013/09/24 12:42:54 leavens Exp $
module MatrixTests where
import Matrix
import MatrixInstances
import Testing

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

-- helpful definitions follow, NOT for you to implement!
allIndexes :: (Int,Int) -> [(Int,Int)]
allIndexes (m,n) = [(i,j) | i <- [1..m], j <- [1..n]]
initial = (fillWith (2,3) "initial")
m10x3 = fillWith (10,3) "u"
m5x7 = fromRule (5,7) (\(i,j) -> show (i,j))
m10ipj = fromRule (5,7) (\(i,j) -> show (10*i+j))

tests :: [TestCase String] -- the actual tests
tests = (map (\(i,j) -> eqTest (initial `at` (i,j)) "==" "initial")
         (allIndexes (2,3)))
        ++ (map (\(i,j) -> eqTest (m10x3 `at` (i,j)) "==" "u")
            (allIndexes (10,3)))
        ++ (map (\(i,j) -> eqTest (m5x7 `at` (i,j)) "==" (show (i,j)))
            (allIndexes (5,7)))
        ++ (map (\(i,j) -> eqTest (m10ipj `at` (i,j)) "==" (show (10*i+j)))
            (allIndexes (5,7)))
        ++ (map (\(i,j) -> eqTest ((mtranspose m5x7) `at` (i,j))
                           "==" (show (j,i)))
            (allIndexes (7,5)))
        ++ (map (\(i,j) -> eqTest ((mmap reverse m10ipj) `at` (i,j))
                           "==" (reverse (show (10*i+j))))
            (allIndexes (5,7)))
```

Figure 7: Tests for problem 6. In these tests `f` is one of your solutions.

7. (20 points) [UseModels] Complete the module definition in Figure 8 below, by defining the functions `sameShape`, `pointwiseApply`, `add`, and `sub`.

```
-- $Id: MatrixAdd.hs,v 1.1 2013/02/11 14:53:44 leavens Exp leavens $
module MatrixAdd where
import Matrix
import MatrixInstances

sameShape :: (Matrix a) -> (Matrix a) -> Bool
pointwiseApply :: (a -> a -> b) -> (Matrix a) -> (Matrix a) -> (Matrix b)
add :: (Num a) => (Matrix a) -> (Matrix a) -> (Matrix a)
sub :: (Num a) => (Matrix a) -> (Matrix a) -> (Matrix a)
```

Figure 8: Beginning of the module `MatrixAdd`, for you to complete.

The predicate `sameShape` takes arguments of type `Matrix a` and returns **`True`** when they have the same dimensions. The function `pointwiseApply` takes a curried function `op` of two arguments and two matricies, `m1`, and `m2`, which have the same shape and whose elements are the same type as the argument types of `op`, and returns a matrix of the same shape as `m1` and `m2`, in which the `(i,j)`th element is `(m1 `at` (i,j)) `op` (m2 `at` (i,j))`. If the two matrices do not have the same shape, then an error should be raised (using Haskell's `error` function). The `add` and `sub` operations are the usual pointwise addition and subtraction of matrices, and must be defined by using `pointwiseApply`.

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

```
-- $Id: MatrixAddTests.hs,v 1.1 2013/02/11 14:53:44 leavens Exp leavens $
module MatrixAddTests where
import Matrix
import MatrixInstances
import MatrixAdd
import Testing

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

-- helpers for testing below, NOT something you have to implement
allIndexes :: (Int,Int) -> [(Int,Int)]
allIndexes (m,n) = [(i,j) | i <- [1..m], j <- [1..n]]
zeros = fillWith (4,3) 0
id4x3 = fromRule (4,3) (\(i,j) -> if i == j then 1 else 0)
m4x3 = fromRule (4,3) (\(i,j) -> 10*i+j)
m9 = fillWith (4,3) 9

tests :: [TestCase (Matrix Int)]
tests =
    [eqTest (zeros `add` id4x3) "==" id4x3
    ,eqTest (m9 `sub` zeros) "==" m9
    ,eqTest (m9 `sub` id4x3)
     "==" (fromRule (4,3) (\(i,j) -> if i == j then 8 else 9))
    ,eqTest (m9 `sub` m4x3)
     "==" (fromRule (4,3) (\(i,j) -> 9 - (10*i+j)))
    ,eqTest (m9 `add` m4x3)
     "==" (fromRule (4,3) (\(i,j) -> 9 + (10*i+j)))
    ,eqTest (m9 `add` m4x3)
     "==" (fromRule (4,3) (\(i,j) -> 9 + (10*i+j)))
    ]
```

Figure 9: Tests for problem 7.

8. (30 points) [Concepts] [UseModels] A bag (or multiset) can be described by a "characteristic function" (whose range is **Integer**) that determines the number of times an element occurs in the bag. For example, the function ϕ such that $\phi($"coke"$) = 6$ and for all other String arguments x, $\phi(x) = 0$ is the characteristic function for a bag containing the String "coke" 6 times and nothing else. Allowing the user to construct a bag from a characteristic function gives one the power to construct bags that may "contain" an infinite number of elements (such as a bag where all numbers n occur $n + 1$ times).

   In a module named InfBag, you will declare a polymorphic type constructor Bag, which can be declared something like as follows:

```
type Bag a = ...
-- or perhaps something like --
data Bag a = ...
```

   Hint: think about using a function type as part of your representation of bags.

   Then fill in the operations of the module InfBag, which are described informally as follows.

   1. The function

      ```
      fromFunc :: (a -> Integer) -> (Bag a)
      ```

      takes a characteristic function, f and returns a bag such that each value x (of type a) occurs in the bag fx times. Assume that the argument f never returns a negative integer for any argument of type a.

   2. The function

      ```
      minusBag :: Bag a -> Bag a -> Bag a
      ```

      takes two bags as arguments, with characteristic functions f and g and returns a bag that contains each value x of type a the number of times it occurs in the first argument minus the number of times it occurs in the second argument (that is, $fx - gx$ times). However, occurences must always be nonnegative numbers, so in no case can an element occur in a bag a negative number of times.

   3. The function

      ```
      unionBag :: Bag a -> Bag a -> Bag a
      ```

      takes two bags, with characteristic functions f and g, and returns a bag such that each value x (of type a) occurs $(fx) + (gx)$ times.

   4. The function

      ```
      intersectBag :: Bag a -> Bag a -> Bag a
      ```

      takes two bags, with characteristic functions f and g, and returns a bag such that each value x (of type a) occurs the number of times that x occurs in both bags.

   5. The function

      ```
      number :: a -> Bag a -> Integer
      ```

      returns how many times the first argument occurs in the bag represented by the second argument. (The number returned should never be negative.)

   Tests for this are given in the Figure 10 on the following page.

   Note (hint, hint) that the following equations must hold, for all f, g, and x of appropriate types.

```
number x (fromFunc f) == f x
number x ((fromFunc f) `minusBag` (fromFunc g)) == max 0 ((f x) - (g x))
number x ((fromFunc f) `unionBag` (fromFunc g)) == (f x) + (g x)
number x ((fromFunc f) `intersectBag` (fromFunc g)) == min (f x) (g x)
```

```
-- $Id: InfBagTests.hs,v 1.1 2013/09/24 14:34:31 leavens Exp leavens $
module InfBagTests where
import InfBag
import Testing
main :: IO ()
main = dotests "InfBagTests $Revision: 1.1 $" tests
tests :: [TestCase Integer]
tests =
    [(eqTest (number "coke" coke6) "==" 6)
    ,(eqTest (number "pepsi" coke6) "==" 0)
    ,(eqTest (number "pepsi" pepsi12) "==" 12)
    ,(eqTest (number 'e' eBag) "==" 999573)
    ,(eqTest (number 'a' eBag) "==" 0)
    ,(eqTest (number 'a' letterBag) "==" 99)
    ,(eqTest (number 10 squareBag) "==" 100)
    ,(eqTest (number (-5) squareBag) "==" 25)
    ,(eqTest (number 9999999 squareBag) "==" (9999999^2))
    ,(eqTest (number 100000000 np1Bag) "==" 100000001)
    ,(eqTest (number "coke" (pepsi12 `minusBag` coke6)) "==" 0)
    ,(eqTest (number "pepsi" (pepsi12 `minusBag` coke6)) "==" 12)
    ,(eqTest (number "pepsi" (pepsi12 `minusBag` (simpleBag "pepsi" 3))) "==" 9)
    ,(eqTest (number "pepsi" (pepsi12 `minusBag` pepsi12)) "==" 0)
    ,(eqTest (number "pepsi" (pepsi12 `unionBag` coke6)) "==" 12)
    ,(eqTest (number "pepsi" (pepsi12 `unionBag` pepsi12)) "==" 24)
    ,(eqTest (number "coke" (pepsi12 `unionBag` coke6)) "==" 6)
    ,(eqTest (number "coke" (coke6 `unionBag` coke6)) "==" 12)
    ,(eqTest (number "sprite" (pepsi12 `unionBag` coke6)) "==" 0)
    ,(eqTest (number "sprite" (pepsi12 `unionBag` coke6)) "==" 0)
    ,(eqTest (number 'e' (eBag `unionBag` letterBag)) "==" (999573 + 4020))
    ,(eqTest (number "pepsi" (pepsi12 `intersectBag` coke6)) "==" 0)
    ,(eqTest (number "coke" (pepsi12 `intersectBag` coke6)) "==" 0)
    ,(eqTest (number "sprite" (pepsi12 `intersectBag` coke6)) "==" 0)
    ,(eqTest (number "pepsi" (pepsi12 `intersectBag` pepsi12)) "==" 12)
    ,(eqTest (number 'e' (eBag `intersectBag` letterBag)) "==" 4020)
    ]
  where simpleBag what num = fromFunc (\x -> if x == what then num else 0)
        coke6 = simpleBag "coke" 6
        pepsi12 = simpleBag "pepsi" 12
        eBag = simpleBag 'e' 999573
        letterBag = fromFunc (\c -> if c == 'e' then 4020 else if c == 'a' then 99 else 0)
        squareBag = fromFunc (\i -> i*i)
        np1Bag = fromFunc (\n -> n+1)
```

Figure 10: Tests for the module `InfBag`.

Functional Abstractions of Programming Patterns

9. (10 points) [UseModels] [Concepts] Using Haskell's built-in `foldr` function, write the polymorphic function

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

   This function can be considered to be an abstraction of problems like `deleteAll` from Homework 2. An application such as `(concatMap f ls)` applies `f` to each element of `ls`, and concatenates the results of those applications together (preserving the order). Note that application of `f` to an element of type `a` returns a list (of type `[b]`), and so the overall process collects the elements of these lists together into a large list of type `[b]`. Your solution must have the following form:

   ```
   module ConcatMap where
   import Prelude hiding (concatMap)
   concatMap :: (a -> [b]) -> [a] -> [b]
   concatMap f ls = foldr ...
   ```

   where the "`...`" is where you will put the arguments to `foldr` in your solution. Note: your code in `...` should not call `concatMap` (let `foldr` do the recursion).

   There are test cases contained in `ConcatMapTests.hs`, which is shown in Figure 11.

```
-- $Id: ConcatMapTests.hs,v 1.2 2013/09/24 15:05:46 leavens Exp leavens $
module ConcatMapTests where
import Prelude hiding (concatMap)
import ConcatMap
import Testing

main :: IO()
main = dotests "ConcatMapTests $Revision: 1.2 $" tests

-- some definitions using concatMap, for testing, not for you to implement
deleteAll toDel ls = concatMap (\e -> if e == toDel then [] else [e]) ls
duplicate ls = concatMap (\e -> [e,e]) ls

tests :: [TestCase Bool]
tests =
    [assertTrue ((deleteAll 'c' "abcdefedcba") == "abdefedba")
    ,assertTrue ((deleteAll 3 [3,3,3,7,3,9]) == [7,9])
    ,assertTrue ((deleteAll 3 []) == [])
    ,assertTrue ((duplicate "") == "")
    ,assertTrue ((duplicate "okay") == "ookkaayy")
    ,assertTrue ((duplicate "balon") == "bbaalloonn")
    ]
```

Figure 11: Tests for problem 9.

---

10. (30 points) [UseModels] [Concepts] In this problem you will write a function

```
foldWindowLayout :: ((String,Int,Int) -> r) -> ([r] -> r) -> ([r] -> r)
                 -> WindowLayout -> r
```

that abstracts from all the `WindowLayout` examples we have seen (such as those in homework 2 and on the course examples page). For each type `r`, the function `foldWindowLayout` takes 3 functions: `wf`, `hf`, and `vf`, which correspond to the three variants in the grammar for `WindowLayout`. In more detail:

- `wf`, operates on a tuple of the information from a `Window` variant and returns a value of type `r`,
- `hf`, takes a list of the results of mapping `(foldWindowLayout wf hf vf)` over the list in a `Horizontal` variant, and
- `vf`, takes a list of the results of mapping `(foldWindowLayout wf hf vf)` over the list in a `Vertical` variant.

There are test cases contained in `FoldWindowLayoutTests.hs`, which is shown in Figure 12 on the following page and Figure 13 on page 19.

Points

This homework's total points: 200.

References

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

```
-- $Id: FoldWindowLayoutTests.hs,v 1.2 2013/09/24 18:55:42 leavens Exp leavens $
module FoldWindowLayoutTests where
import WindowLayout
import FoldWindowLayout
import Testing

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

-- uses of foldWindowLayout for testing purposes, not for you to implement
watching' = foldWindowLayout (\(wn,_,_) -> [wn]) concat concat
changeChannel new old =
    let changeName new old nm = if nm == old then new else nm
    in foldWindowLayout
           (\(nm,w,h) -> Window {wname = changeName new old nm,
                                 width = w, height = h})
           Horizontal
           Vertical
doubleSize = foldWindowLayout
              (\(wn,w,h) -> Window {wname = wn, width = 2*w, height = 2*h})
              Horizontal
              Vertical
addToSize n = foldWindowLayout
               (\(wn,w,h) -> Window {wname = wn, width = n+w, height = n+h})
               Horizontal
               Vertical
multSize n = foldWindowLayout
               (\(wn,w,h) -> Window {wname = wn, width = n*w, height = n*h})
               Horizontal
               Vertical
totalWidth = foldWindowLayout
               (\(_,w,_) -> w)
               sum
               sum

-- a WindowLayout for testing
hlayout =
    (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 = "track", width = 100, height = 60})
                  ,(Window {wname = "golf", width = 70, height = 30})])
     ,(Vertical [(Window {wname = "Star Trek", width = 40, height = 100})
                ,(Window {wname = "olympics", width = 80, height = 33})
                ,(Window {wname = "news", width = 20, height = 10})])])
```

Figure 12: Tests for problem 10, part 1.

```
tests :: [TestCase Bool]
tests =
    [assertTrue ((totalWidth hlayout) == 1760)
    ,assertTrue ((doubleSize hlayout) == (multSize 2 hlayout))
    ,assertTrue ((watching' hlayout)
                 == ["Tempest","Othello","Hamlet","baseball","track","golf",
                     "Star Trek","olympics","news"])
    ,assertTrue
     ((changeChannel
       "pbs" "news"
       (Vertical [(Window {wname = "news", width = 10, height = 5})
                 ,(Window {wname = "golf", width = 50, height = 25})
                 ,(Window {wname = "news", width = 30, height = 70})]))
      ==
      (Vertical [(Window {wname = "pbs", width = 10, height = 5})
                ,(Window {wname = "golf", width = 50, height = 25})
                ,(Window {wname = "pbs", width = 30, height = 70})]))
    ,assertTrue
     ((addToSize 100 hlayout)
      ==
      (Horizontal
       [(Vertical [(Window {wname = "Tempest", width = 300, height = 200})
                  ,(Window {wname = "Othello", width = 300, height = 177})
                  ,(Window {wname = "Hamlet", width = 1100, height = 700})])
       ,(Horizontal [(Window {wname = "baseball", width = 150, height = 140})
                    ,(Window {wname = "track", width = 200, height = 160})
                    ,(Window {wname = "golf", width = 170, height = 130})])
       ,(Vertical [(Window {wname = "Star Trek", width = 140, height = 200})
                  ,(Window {wname = "olympics", width = 180, height = 133})
                  ,(Window {wname = "news", width = 120, height = 110})])]) )
    ,assertTrue
     ((doubleSize hlayout)
      ==
      (Horizontal
       [(Vertical [(Window {wname = "Tempest", width = 400, height = 200})
                  ,(Window {wname = "Othello", width = 400, height = 154})
                  ,(Window {wname = "Hamlet", width = 2000, height = 1200})])
       ,(Horizontal [(Window {wname = "baseball", width = 100, height = 80})
                    ,(Window {wname = "track", width = 200, height = 120})
                    ,(Window {wname = "golf", width = 140, height = 60})])
       ,(Vertical [(Window {wname = "Star Trek", width = 80, height = 200})
                  ,(Window {wname = "olympics", width = 160, height = 66})
                  ,(Window {wname = "news", width = 40, height = 20})])]) )
    ]
```

Figure 13: Tests for problem 10, part 2.