Spring, 2014 Name: ______________________

(Please *don't* write your id number!)

COP 4020 — Programming Languages I

Test on Advanced Functional Programming

Special Directions for this Test

This test has 7 questions and pages numbered 1 through 10.
This test is open book and notes, but no electronics.
If you need more space, use the back of a page. Note when you do that on the front.
Before you begin, please take a moment to look over the entire test so that you can budget your time.
Clarity is important; if your programs are sloppy and hard to read, you may lose some points. Correct syntax also makes a difference for programming questions. Take special care with indentation and capitalization in Haskell.
When you write Haskell code on this test, you may use anything we have mentioned in class that is built-in to Haskell. But unless specifically directed, you should not use imperative features (such as the IO type).
You are encouraged to define helping functions whenever you wish. Note that if you use functions that are not in the standard Haskell Prelude, such as `map`, `filter`, `foldr`, `maximum`, `sum`, etc., then you must write them into your test. (That is, your code may not import modules other than the Prelude.)

Hints

If you use functions like `filter`, `map`, `concatMap`, and `foldr` whenever possible, then you will have to write less code on the test, which will mean fewer chances for making mistakes and will leave you more time to be careful.

For Grading

Question:	1	2	3	4	5	6	7	Total
Points:	10	15	10	15	15	10	25	100
Score:								

1. (10 points) [UseModels] Using `foldr`, write the function

```
sumLL :: (Num t) => [[t]] -> t
```

that takes a list of lists of numbers `llt` and returns the sum of all the elements in `llt`. The following are examples, written using the `Testing` module from the homework.

```
tests :: [TestCase Integer]
tests =
    [eqTest (sumLL []) "==" 0
    ,eqTest (sumLL [[]]) "==" 0
    ,eqTest (sumLL [[5,10]]) "==" 15
    ,eqTest (sumLL [[5],[10,20]]) "==" 35
    ,eqTest (sumLL [[1 .. 10],[100 .. 200]]) "==" 15205
    ,eqTest (sumLL [[1 .. 1000],[3],[4]]) "==" 500507
    ,eqTest (sumLL [[-55 .. 55],[-1,0,1],[-2000 .. 2000]]) "==" 0
    ,eqTest (sumLL [[-100 .. 99],[100,5]]) "==" 5
    ]
```

Your solution must use `foldr` in an essential way, so you must write it by filling in the remainder of the following. You must not use explicit recursion or a list comprehension in your solution. However, you may use helping functions and built-in functions.

```
sumLL llt = foldr
```

2. (15 points) [UseModels] Using `foldr`, write the function:

```
frMap2 :: ((a,b) -> c) -> [(a,b)] -> [c]
```

that for any types `a`, `b`, and `c`, takes a function, `f`, of type `((a,b) -> c)` and a list of pairs, `lp`, of type `[(a,b)]` and returns a list of elements of type `c` that is the result of applying `f` to each pair in `lp` and forming a list of the results, in an order that corresponds to the order of `lp`. The following are examples, written using the `Testing` module from the homework.

```
tests_int :: [TestCase [Integer]]
tests_int =
    [eqTest (frMap2 fst []) "==" []
    ,eqTest (frMap2 fst [(-1,3),(-5,4)]) "==" [-1,-5]
    ,eqTest (frMap2 snd [(-1,3),(-5,4)]) "==" [3,4]
    ,eqTest (frMap2 (\(x,y) -> x+y) [(3,4),(5,6),(9,2)]) "==" [7,11,11]
    ,eqTest (frMap2 (\(x,y) -> x-y) [(3,4),(5,6),(9,2)]) "==" [-1,-1,7]
    ,eqTest (frMap2 (\(x,y) -> x*y) [(3,4),(5,6),(9,2)]) "==" [12,30,18]
    ,eqTest (frMap2 (\(x,y) -> 3*x+y-1) [(1,0),(0,0),(0,1),(2,3)])
     "==" [2,-1,0,8]
    ]
```

Your solution must use `foldr` in an essential way, so write it by filling in the remainder of the following. You must not use explicit recursion or a list comprehension in your solution. Your solution is also not allowed to use `map`.

```
frMap2 f lp = foldr
```

Matrix Type and Examples for Testing

The following type definitions, which are in a module `Mat`, are used in the next few problems.

```
-- $Id: Mat.hs,v 1.1 2015/03/23 00:03:47 leavens Exp $
module Mat where
data Mat t = S t
           | D [Mat t]
             deriving (Eq, Show)
```

Figure 1: N-dimensional Matrix type, for use in later problems. The `S` constructor is for scalars (or simple matricies). The `D` constructor is for higher dimensions.

These type definitions are intended to model n-dimensional matricies. The definitions given in the module `MatExamples` are examples of matricies, which are used in later tests.

```
-- $Id: MatExamples.hs,v 1.2 2015/03/24 10:31:36 leavens Exp leavens $
module MatExamples where
import Mat
-- DON'T IMPLEMENT THESE, THEY ARE JUST EXAMPLES!
-- vec takes a list and makes a 1-dimensional Mat out of it
vec :: [t] -> Mat t
vec xs = D (map S xs)
-- twoD makes a 2-dimensional matrix from a [[t]] value
twoD :: [[t]] -> Mat t
twoD llt = D (map vec llt)
-- fromRule2 assumes that the arguments (m,n) are both at least 1.
fromRule2 :: (Int,Int) -> ((Int,Int) -> t) -> Mat t
fromRule2 (m,n) f = D (map (\i -> D (row i)) [1 .. m])
    where row i = (map (\j -> S (f (i,j))) [1 .. n])
-- similarly for fromRule3, which makes 3-dimensional matricies
fromRule3 :: (Int,Int,Int) -> ((Int,Int,Int) -> t) -> Mat t
fromRule3 (m,n,o) f = D (map (\i -> (d2 i)) [1 .. m])
    where d2 i = D (map (\j -> D (d1 i j)) [1 .. n])
          d1 i j = (map (\k -> S (f (i,j,k))) [1 .. o])
-- 2-dimensional matricies of a given size
unit2 (m,n) = fromRule2 (m,n) (\(i,j) -> if i == j then 1 else 0)
zero2 (m,n) = fromRule2 (m,n) (\(_,_) -> 0)
ten2 (m,n) = fromRule2 (m,n) (\(i,j) -> (10*i)+j)
-- 3-dimensional matricies of a given size
unit3 (m,n,o) = fromRule3 (m,n,o) (\(i,j,k) -> if i == j && j == k then 1 else 0)
zero3 (m,n,o) = fromRule3 (m,n,o) (\(_,_,_) -> 0)
ht3 (m,n,o) = fromRule3 (m,n,o) (\(i,j,k) -> (100*i)+(10*j)+k)
```

Figure 2: Matrix examples, for use in later tests.

---

3. (10 points) [UseModels] This is a problem about the matrix type in Figure 1 on the preceding page. In Haskell, write a function

```
maxMat :: (Ord t) => (Mat t) -> t
```

that for any ordered type `t` takes a non-empty `Mat t` value, `m`, and returns the largest element of `m`. You may assume that `m` is non-empty, and that, in particular, every list in every dimension of `m` is also non-empty. Be sure to follow the grammar implied by the types in Figure 1 on the previous page! The following are examples, written using the `Testing` module from the homework and using the definitions given in Figure 2 on the preceding page.

```
-- $Id: MaxMatTests.hs,v 1.1 2015/03/23 00:03:47 leavens Exp $
module MaxMatTests where
import Mat; import MatExamples; import MaxMat; import Testing
main = dotests "MaxMatTests $Revision: 1.1 $" (maxMat_tests maxMat)
maxMat_tests :: (Mat Int -> Int) -> [TestCase Int]
maxMat_tests mm = -- the argument mm is a solution to the problem
    [eqTest (mm (S 5)) "==" 5
    ,eqTest (mm (ten2 (5,6))) "==" 56
    ,eqTest (mm (fromRule2 (100,300) (\(i,j) -> if i == 35 && j == 42 then 99999 else i-j))) "==" 99999
    ,eqTest (mm (ht3 (5,6,7))) "==" 567
    ,eqTest (mm (fromRule3 (10,20,30) (\(i,j,k) -> (40*i)-(30*j)-k))) "==" 369 ]
```

4. (15 points) [UseModels] This is a problem about the matrix type in Figure 1 on page 4. In Haskell, write the function

```
scaleMat :: (Num t) => t -> (Mat t) -> (Mat t)
```

which for any numeric type `t` takes a value of type `t`, `n`, and a matrix `m` of elements of type `t`, and returns a matrix that is just like `m` but in which every element has been multiplied by `n`. Be sure to follow the grammar given by the types in Figure 1 on page 4! The following are examples, written using the definitions in Figure 2 on page 4.

```
-- $Id: ScaleMatTests.hs,v 1.1 2015/03/23 00:03:47 leavens Exp $
module ScaleMatTests where
import Mat; import MatExamples; import ScaleMat; import Testing
main = dotests "ScaleMatTests $Revision: 1.1 $" (scaleMat_tests scaleMat)
scaleMat_tests :: (Int -> (Mat Int -> Mat Int)) -> [TestCase (Mat Int)]
scaleMat_tests sm = -- the argument sm is a solution to the problem
    [eqTest (sm 5 (S 5)) "==" (S 25)
    ,eqTest (sm 10 (ten2 (5,6))) "==" (fromRule2 (5,6) (\(i,j) -> 10*(10*i+j)))
    ,eqTest (sm 100 (ht3 (7,8,9))) "==" (fromRule3 (7,8,9) (\(i,j,k) -> 100*((100*i)+(10*j)+k)))
    ,eqTest (sm 0 (ht3 (7,8,9))) "==" (zero3 (7,8,9))
    ,eqTest (sm 1 (ht3 (5,6,7))) "==" (ht3 (5,6,7))
    ,eqTest (sm 3 (unit3 (5,5,5))) "==" (fromRule3 (5,5,5)
                                          (\(i,j,k) -> if i==j&&j==k then 3 else 0))    ]
```

5. (15 points) This is also a problem about the matrix type in Figure 1 on page 4. In Haskell, write the function

```
mapMat :: (t -> r) -> (Mat t) -> (Mat r)
```

that for types `t` and `r`, takes function, `f`, of type `(t -> r)` and a matrix, `m`, whose elements have type `t`, and returns a matrix that is like `m`, except that the value of each element of the result is the result of applying `f` to the corresponding element of `m`. Be sure to follow the grammar given by the types in Figure 1 on page 4 in your solution! The following are examples written using the definitions in Figure 2 on page 4.

```
-- $Id: MapMatTests.hs,v 1.1 2015/03/24 00:45:06 leavens Exp leavens $
module MapMatTests where
import Mat; import MatExamples; import MapMat; import Testing
main = dotests "MapMatTests $Revision: 1.1 $" (mapMat_tests mapMat)
mapMat_tests :: ((Int -> Int) -> (Mat Int) -> Mat Int) -> [TestCase (Mat Int)]
mapMat_tests mm = -- the argument mm is a solution to the problem
    [eqTest (mm (5+) (S 5)) "==" (S 10)
    ,eqTest (mm (10+) (ten2 (5,6))) "==" (fromRule2 (5,6) (\(i,j) -> 10+(10*i+j)))
    ,eqTest (mm (\n -> 2*(100+n)) (ht3 (7,8,9)))
                "==" (fromRule3 (7,8,9) (\(i,j,k) -> 2*(100+((100*i)+(10*j)+k))))
    ,eqTest (mm (+0) (ht3 (7,8,9))) "==" (ht3 (7,8,9))
    ,eqTest (mm (1+) (ht3 (5,6,7))) "==" (fromRule3 (5,6,7) (\(i,j,k) -> 1+(100*i)+(10*j)+k))
    ,eqTest (mm (3+) (unit3 (5,5,5))) "==" (fromRule3 (5,5,5)
                                          (\(i,j,k) -> if i==j&&j==k then 4 else 3))    ]
```

6. (10 points) [Concepts] [UseModels] Suppose we want to generalize the previous three problems involving the matrix type from Figure 1 on page 4. That is, suppose we want to have a function:

```
foldMat :: (t -> r) -> ([r] -> r) -> (Mat t) -> r
```

This function should be such that, for any type `t` and desired result type `r`, it takes two function arguments, `sf`, of type `(t -> r)`, and `lf`, of type `([r] -> r)`, and a matrix, `m`, whose elements have type `t`, and returns a value of type `r`. This function is an abstraction of the pattern of recursion over values of type `(Mat t)`, and thus should use `sf` for the `S` case and `lf` on the result of the recursion over all elements in the `D` case. The following are test cases, written using the data in Figure 2 on page 4 and the tests cases in the previous 3 problems.

```
-- programming the examples using foldMat for testing purposes
maxMat = foldMat id maximum
scaleMat n = foldMat (\x -> S (n*x)) D
mapMat f = foldMat (\x -> S (f x)) D
-- parameterizing the tests to try out the above
tests_mm = maxMat_tests maxMat
tests_sm = scaleMat_tests scaleMat
tests_mpm = mapMat_tests mapMat
```

Your task in this problem is to choose which one of the following, if any, is a declaration that correctly implements `foldMat`. The correct implementation should have the type and behavior described above and satisfy the test cases given above. (So don't ask us why some choice has a type error or is incorrect during the test — it's because it is the wrong answer!) Circle the letter of the correct choice.

A.
```
foldMat sf lf m = lf (map sf m)
```

B.
```
foldMat sf (S x) = S (sf x)
foldMat sf (D xs) = D (map sf xs)
```

C.
```
foldMat sf _ (S x) = sf x
foldMat sf lf (D xs) = lf (map (foldMat sf lf) xs)
```

D.
```
foldMat sf _ (S x) = S (sf x)
foldMat sf lf (D xs) = D (concatMap (lf . sf) xs)
```

E.
```
foldMat sf _ (S x) = sf (S x)
foldMat sf lf (D xs) = lf (D (map (foldMat sf lf) xs))
```

F.
```
foldMat sf lf (S x) = lf [sf x]
foldMat sf lf (D xs) = (foldMat sf lf (D (lf xs)))
```

G.
```
foldMat sf lf (S x) = D (lf [sf x])
foldMat sf lf (D xs) = S (foldMat sf lf (D (lf xs)))
```

H.
```
foldMat sf _ x = sf x
```

I. None of the above purported solutions are correct.

Decimal Fraction Problem

7. [UseModels] [Concepts] In this problem you will implement an abstract data type `DecFrac`. Abstractly, a `DecFrac` is an infinite decimal fraction between 0 and 1. We have decided to represent the type `DecFrac` using the type declarations at the beginning of the module `DecFrac` as follows.

```
module DecFrac where
type Digit = Int  -- only the numbers 0 .. 9
type PosInteger = Integer -- only positive integers, i.e., 1 ..
data DecFrac = Digits (PosInteger -> Digit)
```

We are assuming for this problem that values of the type `Digit` are between 0 and 9 (inclusive) and that values of type `PosInteger` are always strictly positive. In this problem you will implement:

(a) (9 points) The function

```
fromRule :: (PosInteger -> Digit) -> DecFrac
```

takes a function, `f`, of type `(PosInteger -> Digit)`, and returns a `DecFrac` value that represents $\sum_{j=1}^{\infty}(\texttt{f}\ j) \times 10^{-j}$ (i.e., the jth decimal digit is given by `f` applied to j).

(b) (6 points) The function

```
digit :: DecFrac -> PosInteger -> Digit
```

takes a `DecFrac`, `d`, and positive integer, `n`, and returns the nth decimal digit of `d`.

(c) (10 points) The function

```
gt :: PosInteger -> DecFrac -> DecFrac -> Bool
```

takes a positive integer, `lim`, and two `DecFrac`s, `x` and `y`, and returns True just when the fraction formed from the first `lim` decimal digits of `x` is strictly greater than the fraction formed from the first `lim` decimal digits of `y`, and False otherwise. (The limit ensures that the comparison is always well-defined.)

There are test cases in Figure 3 on the next page. Complete the implementation of the module `DecFrac` that was started above by implementing these three functions.

```
fromRule :: (PosInteger -> Digit) -> DecFrac
digit :: DecFrac -> PosInteger -> Digit
gt :: PosInteger -> DecFrac -> DecFrac -> Bool
```

```
module DecFracTests where
import DecFrac; import Testing
main = dotests2 "DecFracTests $Revision: 1.1 $" tests_d tests_b
-- definitions for testing (and tests of fromRule), not for you to implement
fromList :: [Digit] -> DecFrac
fromList ds = fromRule (\n -> if n <= (toInteger (length ds))
                              then ds!!(fromInteger (n-1)) else 0)
rep :: [Digit] -> DecFrac  -- form repeated decimal
rep ds = fromRule (\n -> ds!!(fromInteger ((n-1) `mod` (toInteger (length ds)))))
seventh = (rep [1,4,2,8,5,7])
eighth = (fromList [1,2,5])
ninth = (fromRule (\n -> 1))
tenth = (fromList [1])
eleventh = (rep [0,9])
gt800 = gt 800  -- 800 digits of comparison version of gt
-- testcases themselves
tests_d :: [TestCase Digit]
tests_d = [eqTest (digit seventh 1) "==" 1
          ,eqTest (digit seventh 2) "==" 4
          ,eqTest (digit seventh 3) "==" 2
          ,eqTest (digit seventh 8) "==" 4
          ,eqTest (digit seventh 601) "==" 1
          ,eqTest (digit eighth 1) "==" 1
          ,eqTest (digit eighth 2) "==" 2
          ,eqTest (digit eighth 3) "==" 5
          ,eqTest (digit eighth 4) "==" 0
          ,eqTest (digit eighth 99999999211345) "==" 0
          ,eqTest (digit ninth 9999999) "==" 1
          ,eqTest (digit tenth 1) "==" 1
          ,eqTest (digit tenth 2) "==" 0
          ,eqTest (digit tenth 3) "==" 0
          ,eqTest (digit tenth 44449921) "==" 0
          ,eqTest (digit eleventh 1) "==" 0
          ,eqTest (digit eleventh 2) "==" 9
          ,eqTest (digit eleventh 3) "==" 0
          ,eqTest (digit eleventh 6) "==" 9      ]
tests_b :: [TestCase Bool]
tests_b = [assertTrue (seventh `gt800` eighth)
          ,assertTrue (eighth `gt800` ninth)
          ,assertTrue (ninth `gt800` tenth)
          ,assertTrue (tenth `gt800` eleventh)
          ,assertFalse (eleventh `gt800` tenth)
          ,assertFalse (tenth `gt800` tenth)
          ,assertFalse (eighth `gt800` eighth)    ]
```

Figure 3: Tests for the problem on the previous page. To read the code, it may be useful to recall that `ls!!n` is the nth element of the list `ls`, when `n` is an `Int`, and that `length` returns an `Int`. In the code above think of `fromInteger` as having type `Integer -> Int`, and `toInteger` as having the type `Int -> Integer`.