Fall, 2019 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 11.
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`, `concat`, `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. You may recall the following definitions (from the Haskell Prelude).

```
map :: (a -> b) -> [a] -> [b]
map f []     = []
map f (x:xs) = f x : map f xs

filter :: (a -> Bool) -> [a] -> [a]
filter p []                 = []
filter p (x:xs) | p x       = x : filter p xs
                | otherwise = filter p xs

concat :: [[a]] -> [a]
concat xss = foldr (++) [] xss

foldr :: (a -> b -> b) -> b -> [a] -> b
foldr f z []     = z
foldr f z (x:xs) = x `f` foldr f z xs

concatMap :: (a -> [b]) -> [a] -> [b]
concatMap f = foldr (\e res -> (f e) ++ res) []
```

For Grading

Question:	1	2	3	4	5	6	7	Total
Points:	10	10	10	20	15	25	10	100
Score:								

1. (10 points) [UseModels] Consider the function

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

that for all types a and b, takes a function, f, from a to b, and a list of lists, lla, whose elements are of type a, and returns a list of lists whose elements are the lists resulting from mapping f over the lists of type a inside lla, in order. The following are examples, written using the Testing module from the homework.

```
module MapInsideTests where
import MapInside; import Testing;
import Data.Char -- defines fromEnum :: Char -> Int
main = dotests "MapInsideTests $Revision: 1.1 $" tests
tests :: [TestCase [[Int]]]
tests =
    [eqTest (mapInside (+1) []) "==" []
    ,eqTest (mapInside (+1) [[]]) "==" [[]]
    ,eqTest (mapInside (+1) [[5,10]]) "==" [[6,11]]
    ,eqTest (mapInside (+3) [[5],[],[10,20,3]]) "==" [[8],[],[13,23,6]]
    ,eqTest (mapInside (\i->(i*2+1)*i) [[],[4],[9,8,3,5],[]])
         "==" [[],[36],[171,136,21,55],[]]
    ,eqTest (mapInside (*2) [[1 .. 10000],[3],[],[4]])
         "==" [[2,4 .. 20000],[6],[],[8]]
    ,eqTest (mapInside fromEnum [['c','d']]) "==" [[99,100]]
    ]
```

Which of the following correctly uses foldr to write mapInside? Circle the correct choice.

A. `mapInside f lla = foldr (\ls res -> res : (map f ls)) lla []`

B. `mapInside f lla = foldr (\ls res -> res : (map f ls)) [] lla`

C. `mapInside f lla = foldr (\ls res -> map f ls : res) [] lla`

D. `mapInside f lla = foldr (\ls res -> [map f ls] : res) [] lla`

E.
```
mapInside f lla = foldr
                  (\ls res -> (foldr (\e r -> [f e] : r) [] ls) : res)
                  []
                  lla
```

F. `mapInside f lla = foldr (\ls res -> (map f ls) ++ res) [] lla`

G. None of the above is correct.

2. (10 points) [UseModels] Consider the higher-order `tailrec` function below, which abstracts the pattern of tail recursion.

```
module Tailrec where
tailrec :: (a -> Bool) -> (a -> b) -> (a -> a) -> a -> b
tailrec isDone extract transform s = loop s
  where loop s = if (isDone s) then (extract s)
                 else loop (transform s)
```

Using the `tailrec` function, in this problem we will consider ways to write the function

```
unzip :: [(a,b)] -> ([a],[b])
```

The function `unzip` is such that for all types `a` and `b`, it takes a list of pairs, `abs` (of type `(a,b)`), and returns a pair of lists, where the first list is the first elements of `abs` in order and the second list is the list of the second elements of `abs` in order. The following are examples, written using the `Testing` module from the homework.

```
module UnzipTests where
import Prelude hiding(unzip); import Unzip; import Testing
main = dotests "UnzipTests $Revision: 1.1 $" tests
tests :: [TestCase ([Int],[Char])]
tests = [eqTest (unzip []) "==" ([],[])
        ,eqTest (unzip [(3,'c'),(4,'a')]) "==" ([3,4],['c','a'])
        ,eqTest (unzip [(9,'b'),(3,'c'),(4,'a')]) "==" ([9,3,4],['b','c','a'])
        ,eqTest (unzip [(9,'U'),(6,'C'),(4,'F')]) "==" ([9,6,4],"UCF")
        ,eqTest (unzip [(1,'C'),(9,'E'),(6,'C'),(4,'S')])
             "==" ([1,9,6,4],"CECS")    ]
```

Which of the following correctly implements `unzip` (assuming it is in a module named `Unzip` that imports `Tailrec`)? (Recall that `null` returns True if its argument is the empty list and False otherwise.) Circle the correct choice.

A.
```
unzip lop = tailrec null reverse (:) lop
```

B.
```
unzip lop = tailrec (\(abs,(_,_)) -> null abs)
                    (\(_,(as,bs)) -> ((reverse as),(reverse bs)))
                    (\(((a,b):abs),(as,bs)) ->(abs,((a:as),(b:bs))))
                    (lop,([],[]))
```

C.
```
unzip lop = tailrec null
                    (\(_,(as,bs)) -> ((reverse as),(reverse bs)))
                    (\(((a,b):abs),(as,bs)) ->(abs,((a:as),(b:bs))))
                    (lop,([],[]))
```

D.
```
unzip lop = tailrec (\(abs,(_,_)) -> null abs)
                    snd
                    (\(((a,b):abs),(as,bs)) ->(abs,((a:as),(b:bs))))
                    (lop,[])
```

E.
```
unzip lop = tailrec (\(abs,(_,_)) -> null abs)
                    (\(_,(as,bs)) -> ((reverse as),(reverse bs)))
                    (map unzip)
                    (lop,[])
```

F. None of the above is correct.

File System Types

Consider the types `Directory` and `FSItem` defined by the grammar in the following module.

```
module FileSys where
data Directory = Dir [(Name, FSItem)] deriving (Eq, Show)
data FSItem = F Contents | D Directory deriving (Eq, Show)
type Name = String
type Contents = String
```

Figure 1: File system types for use in later problems. Note that the types `Name` and `Contents` are synonyms for the type `String`.

In a directory, the list of pairs names each `FSItem` in the sense that the `Name` in the pair is the directory's name for the `FSItem` in that pair.

Some examples in the grammar of Figure 1 are given in Figure 2.

```
module FileSysExamples where
import FileSys

emptyDir :: Directory
emptyDir = (Dir [])

books :: Directory
books = (Dir [("first", F "In the beginning...")
             ,("second", F "It was a dark and stormy...")
             ,("third", D emptyDir)])

cardinals :: Directory
cardinals = (Dir [("one", D (Dir [("two", D emptyDir)]))
                 ,("three", F "It was the best of times...")
                 ,("four", D (Dir [("two", F "blah")]))])

volumes :: Directory
volumes = (Dir [("A", D books)
               ,("B", D cardinals)])
```

Figure 2: File system examples for use in later tests.

3. (10 points) [UseModels] Consider a function

   ```
   listNames :: Directory -> [Name]
   ```

   that takes a `Directory`, `dir`, as an argument, and returns a list of all the names in `dir` and its subdirectories in a depth-first traversal. That is, if `e` is the name of a directory in `dir`, then each name contained in `e` is listed before any of the directories that follow `e` in `dir`. The following are examples.

```
module ListNamesTests where
import FileSys; import FileSysExamples; -- defines books, cardinals, volumes
import Testing; import ListNames
main = dotests "ListNamesTests $Revision: 1.2 $" tests
tests :: [TestCase [Name]]
tests = [eqTest (listNames emptyDir) "==" []
        ,eqTest (listNames books) "==" ["first", "second", "third"]
        ,eqTest (listNames cardinals)
             "==" ["one", "two", "three", "four", "two"]
        ,eqTest (listNames volumes)
             "==" ["A", "first", "second", "third",
                   "B", "one", "two", "three", "four", "two"]      ]
```

Figure 3: Tests for `listNames`. Note that these use the examples defined in Figure 2 on the preceding page.

Which of the following correctly implements `listNames` and also follows the grammar for `Directory`?

A.
```
listNames dir = [n | n <- listNames dir]
```

B.
```
listNames dir = [n | (n,_) <- listNames dir]
```

C.
```
listNames (Dir pairs) = concat (map listNames pairs)
```

D.
```
listNames (Dir pairs) = (map (\(n,_) -> n) pairs)
```

E.
```
listNames (n, (F _)) = [n]
listNames (n, (D d)) = n:(listNames d)
listNames (Dir pairs) = (concatMap listNames pairs)
```

F.
```
listNames (Dir pairs) = concatMap listItemNames pairs
listItemNames :: (Name, FSItem) -> [Name]
listItemNames (n, (F _)) = [n]
listItemNames (n, (D d)) = n:(listNames d)
```

G. None of the above are correct and follow the grammar.

4. (20 points) [UseModels] This is another problem about the types defined in Figure 1 on page 4. In this problem we will write the following function

```
rename :: Name -> Name -> Directory -> Directory
```

that takes two Names (i.e., Strings), newName, and oldName, and a Directory, dir, and renames each FSItem whose name is (== to) oldName as newName. That is, the result is a directory that is just like dir except that each name equal to oldName is replaced by newName. The following are examples, written using the Testing module from the homework and using the definitions given in Figure 2 on page 4.

```
module RenameTests where
import FileSys; import FileSysExamples; -- defines books, cardinals, volumes
import Testing; import Rename
main = dotests "RenameTests $Revision: 1.1 $" tests
tests :: [TestCase Directory]
tests = [eqTest (rename "Chuck" "Dickens" emptyDir) "==" emptyDir
        ,eqTest (rename "Bible" "first" books)
             "==" (Dir [("Bible", F "In the beginning...")
                       ,("second", F "It was a dark and stormy...")
                       ,("third", D emptyDir)])
        ,eqTest (rename "Tale 2 Cities" "three" cardinals)
             "==" (Dir [("one", D (Dir [("two", D emptyDir)]))
                       ,("Tale 2 Cities", F "It was the best of times...")
                       ,("four", D (Dir [("two", F "blah")]))])
        ,eqTest (rename "Tale 2 Cities" "three" volumes)
             "==" (Dir [("A", D books)
                       ,("B", D (Dir [("one", D (Dir [("two", D emptyDir)]))
                                     ,("Tale 2 Cities", F "It was the best of times...")
                                     ,("four", D (Dir [("two", F "blah")]))]))])
        ,eqTest (rename "NEW" "two" cardinals)
             "==" (Dir [("one", D (Dir [("NEW", D emptyDir)]))
                       ,("three", F "It was the best of times...")
                       ,("four", D (Dir [("NEW", F "blah")]))])
        ,eqTest (rename "NEW" "two" volumes)
             "==" (Dir ([("A", D books)
                        ,("B", D (Dir [("one", D (Dir [("NEW", D emptyDir)]))
                                      ,("three", F "It was the best of times...")
                                      ,("four", D (Dir [("NEW", F "blah")]))]))]))
        ]
```

Correctly solve this problem in Haskell by filling in the blanks in the outline below (assuming that this code is in the module Rename that has imported the module FileSys). Your answer must follow the grammar and it must use the two helping functions renamePair and renameName.

```
rename newName oldName (Dir pairs) =

    Dir (map ____________________________________________________________

renamePair :: Name -> Name -> (Name, FSItem) -> (Name, FSItem)
renamePair newName oldName (nm, (D dir)) =

    ____________________________________________________________________
renamePair newName oldName (nm, (F contents)) =

    ____________________________________________________________________

renameName :: Name -> Name -> Name -> Name
renameName newName oldName name =
    if ____________________________________________________________________
```

5. (15 points) [Concepts] [UseModels] Consider the data type `Document` defined below

```
module Document where
data Document = Blank | Group String Document
                deriving (Eq, Show)
```

Without using any library functions, write in Haskell a function

```
combineDocs :: Document -> Document -> Document
```

which takes two `Document` arguments, `doc1` and `doc2`, and returns a `Document` that has all the strings in `doc1` before the strings in `doc2`, in order. The following are examples using the `Testing` module from the homework.

```
module CombineDocsTests where
import Testing; import Document; import CombineDocs
main = dotests "CombineDocsTests $Revision: 1.1 $"
         [eqTest (combineDocs Blank (Group "Memo" Blank))
                 "==" (Group "Memo" Blank)
         ,eqTest (combineDocs (Group "Schedule" Blank) (Group "Memo" Blank))
                 "==" (Group "Schedule" (Group "Memo" Blank))
         ,eqTest (combineDocs (Group "Evening" (Group "Work" Blank)) (Group "Memo" Blank))
                 "==" (Group "Evening" (Group "Work" (Group "Memo" Blank)))
         ,eqTest (combineDocs (Group "Now" (Group "is" Blank))
                              (Group "the" (Group "time" (Group "for" Blank))))
                 "==" (Group "Now" (Group "is" (Group "the" (Group "time" (Group "for" Blank)))))
         ,eqTest (combineDocs (Group "It" (Group "was" (Group "the" Blank)))
                              (Group "best" (Group "of" (Group "times" (Group "in" (Group "SF" Blank))))))
                 "==" (Group "It" (Group "was" (Group "the" (Group "best"
                         (Group "of" (Group "times" (Group "in" (Group "SF" Blank)))))))))
         ]
```

We will take points off for repeated code or extra, unneeded cases.

6. (25 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 for all positive integers i $\phi(i) = i$ is the characteristic function for a bag containing n of each positive integer n. Allowing the user to construct a bag from a characteristic function allows the user to construct bags that may "contain" an infinite number of elements. (such as a bag where all numbers n occur n^2 times).

   You will implement potentially infinite bags in a module named `PIBag` by filling in the following module declaration.

```
module PIBag where


data Bag a = ____________________________________
             deriving (Eq, Show)

-- implement each of the functions declared below

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



number :: a -> Bag a -> Integer



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



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



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

   Fill in the blank in the data definition for the polymophic type `(Bag a)` in the space indicated above, and also define each of the declared functions, which are described informally as follows.

   - The function

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

     takes a characteristic function, f and returns a bag (of type `(Bag a)`) 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.

   - The function

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

     takes a value x of type a and a potentially infinite bag (of type `(Bag a)` and returns how many times x occurs in the bag. (The number returned should never be negative.)

   - 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.

- 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.

- 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.

Tests for this are given in the Figure 4 on the next 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)
```

```
module PIBagTests where
import PIBag; import Testing
main :: IO ()
main = dotests "PIBagTests $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 4: Tests for the module PIBag.

7. (10 points) In Haskell write a function

```
applyList :: [(t -> t)] -> t -> [t]
```

which for all types `t` takes a list `funs` of functions (of type `(t -> t)`) and a value `x` (of type `t`) and returns a list that is the result of applying each function (separately) to `x`, and forming a list of the results in the same order as the function. The following are examples.

```
module ApplyListTests where
import Testing; import ApplyList
main = dotests "ApplyListTests $Revision: 1.3 $" tests
tests :: [TestCase [Int]]
tests = [eqTest (applyList [] 3) "==" []
        ,eqTest (applyList [(2*)] 3) "==" [6]
        ,eqTest (applyList [(+1),(2*)] 3) "==" [4,6]
        ,eqTest (applyList [(\n->3*n+1),(+1),(2*)] 3) "==" [10,4,6]
        ,eqTest (applyList [(\n->3*n+1),(+1),(2*)] 10) "==" [31,11,20]
        ,eqTest (applyList [(+7),(\n->n-20),(\n->3*n+1),(+1),(2*)] 1)
             "==" [8,-19,4,2,2]
        ,eqTest (applyList [(\_->1),(\n->n),(\n->n*n),(\n->n^3),(\n->n^4)] 2)
             "==" [1,2,4,8,16]    ]
```

Be sure that there are no extra cases in your code or repeated code.