Spring, 2014 Name: ____________________

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

COP 4020 — Programming Languages I

Test on Haskell and Functional Programming

Special Directions for this Test

This test has 8 questions and pages numbered 1 through 8.
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, including: the constructor functions for lists `(:)` and tuples `(,)`, `filter`, `fromIntegral`, `length`, `map`, `sum`, `toLower`, and `toUpper`. But unless specifically directed, you should not use imperative features (such as the IO type), or the monadic syntax.
You are encouraged to define functions not specifically asked for if they are useful to your programming; however, if they are not in the standard Haskell Prelude, then you must write them into your test. (That is, your code may not import modules other than the Prelude.)

Hints

You do not need to write out module declarations or import statements on this test; just write the function requested.
If you use functions like `filter` and `map` 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. The problem will note explicitly if you are prohibited from using such functions, but by default you can.

For Grading

Question:	1	2	3	4	5	6	7	8	Total
Points:	5	5	10	15	15	15	15	20	100
Score:									

1. (5 points) [Concepts] In Haskell, which of the following is equivalent to the list `[4,7,5,8]`? Circle the letter of the correct answer.

   A. `(4 ++ (7 ++ (5 ++ 8)))`

   B. `(((4 ++ 7) ++ 5) ++ 8)`

   C. `(((([]:4):7):5):8)`

   D. `(4:(7:(5:(8:[]))))`

   E. `(4:7) ++ 5:(8:[]))`

2. (5 points) [Concepts] [UseModels] Consider the data type `TrafficColor` defined below.

```
data TrafficColor = Red | Yellow | Green
                  deriving (Eq,Show)
```

In Haskell, write a function

```
trafficNext :: TrafficColor -> TrafficColor
```

which takes a `TrafficColor`, `c`, and returns the next color that a traffic light (i.e., a stop light) would change to when it currently has color `c`. That is: it returns `Green` if `c` is `Red`, it returns `Yellow` if `c` is `Green`, and it returns `Red` if `c` is `Yellow`. The following are examples, written using the `Testing` module from the homework.

```
tests = [(eqTest (trafficNext Red) "==" Green)
        ,(eqTest (trafficNext Green) "==" Yellow)
        ,(eqTest (trafficNext Yellow) "==" Red) ]
```

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

```
capitalize :: [Char] -> [Char]
```

that takes a list of characters, `str`, and returns a list of the same length with the corresponding upper case letter in place of each lower case letter.

In your solution, use the built-in function `toUpper`, which takes a `Char` and returns the corresponding upper case letter for a lower case letter argument; it returns all other `Char`s unchanged.

The following are examples, written using the `Testing` module from the homework.

```
tests :: [TestCase [Char]]
tests = [(eqTest (capitalize []) "==" [])
        ,(eqTest (capitalize ['a']) "==" ['A'])
        ,(eqTest (capitalize "adp") "==" "ADP")
        ,(eqTest (capitalize "usa wins") "==" "USA WINS")
        ,(eqTest (capitalize "slow down!") "==" "SLOW DOWN!")
        ,(eqTest (capitalize "stop") "==" "STOP")        ]
```

4. (15 points) [UseModels] In Haskell, without using ++, write a function

   ```
   join :: ([t],[t]) -> [t]
   ```

   which takes a pair of lists of some type `t`, `(xs,ys)` and returns a single list that contains all the elements of `xs` followed by all the elements of `ys`, in order. The following are examples, written using the `Testing` module from the homework (assume that the prelude's function `join` is hidden).

   ```
   testsInt :: [TestCase [Int]]
   testsInt = [(eqTest (join ([],[])) "==" [])
              ,(eqTest (join ([],[5,6,5])) "==" [5,6,5])
              ,(eqTest (join ([7],[5,6,5])) "==" [7,5,6,5])
              ,(eqTest (join ([3,7],[5,6,5])) "==" [3,7,5,6,5])
              ,(eqTest (join ([1,3,7],[5,6,5])) "==" [1,3,7,5,6,5]) ]
   testsInt :: [TestCase [Char]]
   testsChar = [(eqTest (join ("","happy")) "==" "happy")
               ,(eqTest (join ("be","happy")) "==" "behappy")
               ,(eqTest (join ("Haskell, be ","happy")) "==" "Haskell, be happy")
               ,(eqTest (join ("Haskell, be ","")) "==" "Haskell, be ")
               ,(eqTest (join ("just ","recurse!")) "==" "just recurse!") ]
   ```

   Note: you are prohibited from using ++ in your code.

5. (15 points) [Concepts] [UseModels] In Haskell, without using any library functions, write a function

```
joinAll :: [[t]] -> [t]
```

which for all types `t`, takes a list of lists of elements of type `t`, `lst`, and returns a list of all the `t` elements within `lst`. In other words, `joinAll` joins all the lists in `lst` together. The following are tests.

```
testsInt :: [TestCase [Int]]
testsInt = [(eqTest (joinAll []) "==" [])
           ,(eqTest (joinAll [[]]) "==" [])
           ,(eqTest (joinAll [[3]]) "==" [3])
           ,(eqTest (joinAll [[5],[6,5],[]]) "==" [5,6,5])
           ,(eqTest (joinAll [[7],[],[5,6,5]]) "==" [7,5,6,5])
           ,(eqTest (joinAll [[3,7],[],[5,6,5]]) "==" [3,7,5,6,5])
           ,(eqTest (joinAll [[1],[2],[],[4],[],[8,9],[5]])
             "==" [1,2,4,8,9,5]) ]
testsChar = [(eqTest (joinAll [""]) "==" "")
            ,(eqTest (joinAll ["","happy"]) "==" "happy")
            ,(eqTest (joinAll ["worry, ", "be ","happy"])
              "==" "worry, be happy")
            ,(eqTest (joinAll ["Haskell ", "is ","cool!"])
              "==" "Haskell is cool!") ]
```

In your answer you are prohibited from using any library functions (aside from the list constructor functions). However, you may use the `join` function from the previous problem.

6. (15 points) [UseModels] In Haskell, write the function

```
increasingSums :: (Real t) => [(t,t,t)] -> [t]
```

which takes a list of triples of some type `t`. The type `t` has both `+` and `<` functions, due to its being an instance of the type class `Real` (and hence of both `Ord` and `Num`). When `increasingSums` is applied to a list of triples `triples` it returns a list of the sums of all the elements of the triples that are in strictly increasing order (as defined by <). The result is a list of elements of type `t` that are the sums of the elements of the triples in the list `triples` that satisfy this condition. The following are examples.

```
tests :: [TestCase [Integer]]
tests = [(eqTest (increasingSums []) "==" [])
        ,(eqTest (increasingSums [(0,1,2),(1,1,10)]) "==" [0+1+2])
        ,(eqTest (increasingSums [(3,4,5),(0,1,2),(1,1,10)]) "==" [12,3])
        ,(eqTest (increasingSums [(2,1,3)]) "==" [])
        ,(eqTest (increasingSums [(3,2,1)]) "==" [])
        ,(eqTest (increasingSums [(3,3,3)]) "==" [])
        ,(eqTest (increasingSums [(5,6,7),(3,2,1)]) "==" [5+6+7])
        ,(eqTest (increasingSums [(9,9,9),(5,6,7),(3,2,1)]) "==" [5+6+7])
        ,(eqTest (increasingSums [(9,10,20),(5,6,7),(3,2,1)]) "==" [39,18])
        ,(eqTest (increasingSums [(9,10,20),(5,6,7),(3,2,1)]) "==" [39,18])
        ,(eqTest (increasingSums [(9,10,20),(5,6,7),(3,2,1)]) "==" [39,18])  ]
```

7. (15 points) [UseModels] This problem uses the type BinaryRelation

```
type BinaryRelation a b = [(a,b)]
```

In Haskell, write the function

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

This function takes a key (of some equality type a), and a binary relation (i.e., a list of pairs), br, and returns a binary relation that is just like br except that it does not contain any pair (k,v) in br such that key == k. The following are examples.

```
tests :: [TestCase (BinaryRelation String Integer)]
tests =
  [(eqTest (deleteKey "happy" []) "==" [])
  ,(eqTest (deleteKey "happy" [("happy",3),("sad",2)])
    "==" [("sad",2)])
  ,(eqTest (deleteKey "S" [("S",2010),("C",2006),("B",2002),("S",2014)])
    "==" [("C",2006),("B",2002)])
  ,(eqTest (deleteKey "Voyager"
                  [("Cassini",12000000000),("Voyager",7),
                   ("Curiosity",60000000),("Voyager",2),("Rover",2)])
    "==" [("Cassini",12000000000),("Curiosity",60000000),("Rover",2)])
  ,(eqTest (deleteKey "k" [("k",1),("k",2),("k",3),("k",4)]) "==" [])   ]
```

8. (20 points) [UseModels] In Haskell, write the function

```
variance :: [Float] -> Float
```

that takes a non-empty list of Floats, measures, and returns its variance. The *variance* of a non-empty list $[m_1, m_2, \ldots, m_n]$ is defined as $(1/n) \cdot \Sigma_{i=1}^{n}(m_i - \mu)^2$, where μ is the mean (i.e., the average) of $m_1, m_2, \ldots, m_n$. That is, it is the sum of the squares of the deviations of each measure from the mean, divided by the number of measures. The following are examples, written using the FloatTesting module from the homework.

```
main = dotests "VarianceTests $Revision: 1.1 $" tests
tests :: [TestCase Float]
tests = [(withinTest (variance [1.0]) "~=~" 0.0)
        ,(withinTest (variance [10,20,30,20]) "~=~" 50.0)
        ,(withinTest (variance [3,4,5]) "~=~" 0.6666667)
        ,(withinTest (variance [3.0,3.1,2.9]) "~=~" 6.6666543e-3)
        ,(withinTest (variance [1.2,1.2,1.3]) "~=~" 2.222218e-3)
        ,(withinTest (variance [1.2,1.2,1.3,1.1]) "~=~" 4.999996e-3)  ]
```