Fall, 2019 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 9.
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 (in the Prelude), including: the constructor functions for lists `(:)` and tuples `(,)`, `map`, `filter`, `length`, `sum`. 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 and modules mentioned in the problem.)

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 `map` and `filter` whenever possible, then you will have to write less code on this 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	10	15	15	20	20	100
Score:									

1. (5 points) [Concepts] In Haskell, which of the following is equivalent to the list `[4,0,2,0]`? Circle the letter of the correct answer.

   A. `(((4 ++ 0) ++ 2) ++ 0)`

   B. `(4 ++ (0 ++ (2 ++ 0)))`

   C. `(4:(0:(2:(0:[]))))`

   D. `(4:0) ++ 2:(0:[])`

   E. `((((]:4):0):2):0)`

2. (5 points) [Concepts] [UseModels] Consider the data type `Suit` defined in the module `Suits` below

```
module Suits where
data Suit = Spade | Club | Diamond | Heart deriving (Eq, Show)
```

   In the card game "500", the suits are ordered so that bids in spades are lower than those in clubs, which are lower than bids in diamonds, and bids in diamonds are lower than bids in hearts; that is:

```
Spade < Club < Diamond < Heart
```

   Consider a function

```
lowerSuit :: Suit -> Suit -> Bool
```

   that takes a two `Suit` values, `s1` and `s2` and returns **`True`** just when `s1` is *strictly* lower (in the ordering given above) than `s2`. The following are examples, written using the `Testing` module from the homework. (Note that `assertFalse` expects the answer to be **`False`**, the opposite of `assertTrue`.)

```
module LowerSuitTests where
import Testing
import Suits
import LowerSuit
main = dotests "LowerSuitTests $Revision: 1.1 $"
       [assertFalse (lowerSuit Spade Spade)
       ,assertTrue (lowerSuit Spade Club)
       ,assertTrue (lowerSuit Spade Diamond)
       ,assertTrue (lowerSuit Spade Heart)
       ,assertFalse (lowerSuit Club Spade)
       ,assertFalse (lowerSuit Club Club)
       ,assertTrue (lowerSuit Club Diamond)
       ,assertTrue (lowerSuit Club Heart)
       ,assertFalse (lowerSuit Diamond Spade)
       ,assertFalse (lowerSuit Diamond Club)
       ,assertFalse (lowerSuit Diamond Diamond)
       ,assertTrue (lowerSuit Diamond Heart)
       ,assertFalse (lowerSuit Heart Spade)
       ,assertFalse (lowerSuit Heart Club)
       ,assertFalse (lowerSuit Heart Diamond)
       ,assertFalse (lowerSuit Heart Heart)  ]
```

(This problem continues on the next page...)

Which of the following (assuming it is in a module named `LowerSuit`) is a correct implementation of the `lowerSuit` function? (Circle the correct answer.)

A.
```
import Suits
lowerSuit :: Suit -> Suit -> Bool
lowerSuit Spade Spade = False
lowerSuit Spade _ = True
lowerSuit Club s = case s of Diamond -> True
                             Heart -> True
                             _ -> False
lowerSuit Diamond Heart = True
lowerSuit Diamond _ = False
lowerSuit Heart _ = False
```

B.
```
import Suits
lowerSuit :: Suit -> Suit -> Bool
lowerSuit Heart _ = False
lowerSuit Diamond Heart = True
lowerSuit Diamond _ = False
lowerSuit Club s = s == Diamond || s == Heart
lowerSuit Spade _ = True
lowerSuit Spade Spade = False
```

C.
```
import Suits
lowerSuit :: Suit -> Suit -> Bool
lowerSuit s1 s2 = case (s1,s2) of
                    (Heart, _) -> True
                    (Diamond, s) -> s /= Heart
                    (Club, Club) -> False
                    (_, _) -> True
```

D.
```
import Suits
lowerSuit :: Suit -> Suit -> Bool
lowerSuit s1 s2 =
    case s1 of
      Spade -> True
      Club -> case s2 of
                Diamond -> True
                _ -> False
      Diamond -> case s2 of
                   Spade -> True
                   Club -> False
                   Diamond -> False
                   Heart -> False
      Heart -> False
```

3. (10 points) [Concepts] This is a question about tail recursion. Consider writing a function

```
average :: [Double] -> Double
```

that takes a non-empty list of doubles and returns their average. The following are examples. The following are examples. (Note that `withinTest`, from `FloatTesting`, checks approximate equality.)

```
module AverageTests where
import Testing
import FloatTesting -- defines approximate equality test: ~=~
import Average
main = dotests "AverageTests $Revision: 1.3 $"
         [withinTest (average [1.0]) "~=~" 1.0
         ,withinTest (average [2.0,4.0]) "~=~" 3.0
         ,withinTest (average [1.0 .. 10.0]) "~=~" 5.5
         ,withinTest (average ([1.0 .. 10.0] ++ [1.0 .. 10.0])) "~=~" 5.5
         ,withinTest (average [1.0, 10.0, 100.0, 1000.0]) "~=~" 277.75
         ,withinTest (average [0.0, 3.14, -3.14, 2.78, -2.78]) "~=~" 0.0
         ,withinTest (average [-100.0 .. 100.0]) "~=~" 0.0   ]
```

Which of the following is a *correct* implementation of this function (assuming it was in a module named `Average`) that is *tail recursive*?

Circle the correct, tail-recursive, choice.

A.
```
average :: [Double] -> Double
average ds = average_iter ds (toInteger (length ds))

average_iter :: [Double] -> Integer -> Double
average_iter [] len = 0.0
average_iter (d:ds) len =
    (average_iter ds len) + (d / (fromInteger len))
```

B.
```
average :: [Double] -> Double
average ds = average_iter ds

average_iter :: [Double] -> Double
average_iter ds = (total ds) / (fromInteger (len ds))

total :: [Double] -> Double
total [] = 0.0
total (d:ds) = d + (total ds)

len :: [a] -> Integer
len [] = 0
len (_:xs) = 1 + (len xs)
```

C.
```
average :: [Double] -> Double
average ds = average_iter ds 0 0.0

average_iter :: [Double] -> Integer -> Double -> Double
average_iter [] count total = total / (fromInteger count)
average_iter (d:ds) count total = average_iter ds (1+count) (d+total)
```

D.
```
average :: [Double] -> Double
average ds = average_iter ds 0

average_iter :: [Double] -> Integer -> Double
average_iter [] count = 0.0
average_iter (d:ds) count = (d + average_iter ds (1+count))
                          / (fromInteger  (1+count))
```

4. (10 points) [UseModels] In Haskell, write the function:

```
threeXPlus1Each :: [Integer] -> [Integer]
```

that takes a list of Integers, `ints`, and returns a list of the same length with each integer x replaced by the integer $3 \cdot x + 1$. The following are examples, written using the `Testing` module from the homework.

```
module ThreeXPlus1EachTests where
import Testing
import ThreeXPlus1Each
main = dotests "ThreeXPlus1EachTests $Revision: 1.1 $"
         [eqTest (threeXPlus1Each []) "==" []
         ,eqTest (threeXPlus1Each (1:[])) "==" (4:[])
         ,eqTest (threeXPlus1Each [2,1]) "==" [7,4]
         ,eqTest (threeXPlus1Each [0,5,10,5,-1]) "==" [1,16,31,16,-2]
         ,eqTest (threeXPlus1Each [1, 2 .. 10]) "==" [4, 7 .. 31]
         ,eqTest (threeXPlus1Each [2, 3 .. 30]) "==" [7, 10 .. 91]
         ]
```

Make sure your solution does not have: syntax errors, type errors, incorrect code, extra cases, or repeated code.

5. (15 points) [Concepts] [UseModels] Consider the data type

```
data Stack a = Empty | Push a (Stack a) deriving (Eq, Show)
```

Without using any library functions, write in Haskell a function

```
stackCombine :: (Stack t) -> (Stack t) -> (Stack t)
```

which for all types `t`, takes two `(Stack t)` arguments, `s1`, and `s2`, and returns a stack that has all the elements of `s1` pushed on top of the elements of `s2`, in order. The following are examples using the `Testing` module from the homework.

```
module StackCombineTests where
import Testing
import StackCombine
main = dotests "StackCombineTests $Revision: 1.2 $"
         [eqTest (stackCombine Empty (Push 'e' Empty))
                 "==" (Push 'e' Empty)
         ,eqTest (stackCombine (Push 'c' Empty) (Push 'e' Empty))
                 "==" (Push 'c' (Push 'e' Empty))
         ,eqTest (stackCombine (Push '+' (Push 'c' Empty)) (Push 'e' Empty))
                 "==" (Push '+' (Push 'c' (Push 'e' Empty)))
         ,eqTest (stackCombine (Push '9' (Push '3' Empty))
                               (Push '4' (Push '0' (Push '4' Empty))))
                 "==" (Push '9' (Push '3' (Push '4' (Push '0' (Push '4' Empty)))))
         ,eqTest (stackCombine (Push 'l' (Push 'a' (Push 'n' Empty)))
                               (Push 'g' (Push 'u' (Push 'a' (Push 'g' (Push 'e' Empty))))))
                 "==" (Push 'l' (Push 'a' (Push 'n' (Push 'g'
                         (Push 'u' (Push 'a' (Push 'g' (Push 'e' Empty)))))))) ]
```

6. (15 points) [UseModels] In Haskell, with the type `(BinaryRelation a b)` defined as below, write the function

```
type BinaryRelation a b = [(a,b)]
replaceAt :: (Eq a) => a -> b -> (BinaryRelation a b) -> (BinaryRelation a b)
```

which for any types `a` and `b` such that `a` is an instance of `Eq`, takes a `key` of type `a`, a value `newVal` of type `b`, and a binary relation, `r`, of type `(BinaryRelation a b)` and returns a new binary relation that is just like `r`, except that for every pair `(k,v)` in `r`, if `key` equals `k`, then the corresponding pair in the result is `(k, newVal)`. That is, in the result, `newVal` replaces the second element of each pair in `r` whose first element equals `key`. The following are examples.

```
,eqTest (replaceAt "Don" "Adams" [("Don", "Jose")])
     "==" [("Don", "Adams")]
,eqTest (replaceAt "Fred" "Smith"
                   [("Donna", "Jones"), ("Freddie", "Mercury")])
     "==" [("Donna", "Jones"), ("Freddie", "Mercury")]
,eqTest (replaceAt "Fred" "Smith"
                   [("Fred", "Frank"), ("Donna", "Jones"), ("Freddie", "Mercury")])
     "==" [("Fred", "Smith"), ("Donna", "Jones"), ("Freddie", "Mercury")]
,eqTest (replaceAt "Joan" "Craw"
                   [("Joan", "Jones"), ("Joan", "d'Arc"), ("Joan", "Baptist")])
     "==" [("Joan", "Craw"), ("Joan", "Craw"), ("Joan", "Craw")]
,eqTest (replaceAt "One" "Ton"
                   [("One", "World"), ("Joan", "d'Arc"), ("One", "Thing"), ("One", "Big")])
     "==" [("One", "Ton"), ("Joan", "d'Arc"), ("One", "Ton"), ("One", "Ton")]     ]
```

Your answer should not contain extra, unnecessary or repeated code.

7. (20 points) [UseModels] Consider the data type `(Triple a)` defined as follows

```
module Triple where
data Triple t = Three t t t deriving (Eq, Show)
```

In Haskell, write a function

```
-- recall that pattern matching works from the top of the page downwards
```

that for any type a takes a list, `lst`, and returns a list of all consecutive Triples that can be formed from consecutive elements of `lst`. (If the argument `lst` has two or fewer elements, then the result is the empty list of triples.) The following are examples, using the `Testing` module from the homework.

```
[eqTest (combine3 []) "==" []
,eqTest (combine3 [1]) "==" []
,eqTest (combine3 [1,2]) "==" []
,eqTest (combine3 [1,2,3]) "==" [(Three 1 2 3)]
,eqTest (combine3 [1,2,3,4]) "==" [(Three 1 2 3), (Three 2 3 4)]
,eqTest (combine3 [1,2,3,4,5])
 "==" [(Three 1 2 3), (Three 2 3 4), (Three 3 4 5)]
,eqTest (combine3 [1,2,3,4,5,6])
 "==" [(Three 1 2 3), (Three 2 3 4), (Three 3 4 5), (Three 4 5 6)]
,eqTest (combine3 [1 .. 10])
 "==" [(Three 1 2 3), (Three 2 3 4), (Three 3 4 5), (Three 4 5 6),
       (Three 5 6 7), (Three 6 7 8), (Three 7 8 9), (Three 8 9 10)]
,eqTest (combine3 [1, 2, 1, 3, 1, 4, 1, 5])
 "==" [(Three 1 2 1), (Three 2 1 3), (Three 1 3 1), (Three 3 1 4),
        (Three 1 4 1), (Three 4 1 5)]           ]
```

Hint: don't hesitate to use helping functions in your solution if you wish.

8. (20 points) [UseModels] This problem also uses the data type `(Triple a)` defined as follows

```
module Triple where
data Triple t = Three t t t deriving (Eq, Show)
```

In Haskell, write a function

```
smooth :: [(Triple Double)] -> [Double]
```

that takes a list of Triple values, whose elements are of type **Double**, `triples`, and returns a list of the average of the three elements of each triple. The following are examples, using the `Testing` and `FloatTesting` modules from the homework. (Note that `vecWithin` tests for approximate equality of lists of **Double**s.)

```
module SmoothTests where
import Testing
import FloatTesting -- defines vecWithin for approximate equality of [Double]: ~=~
import Triple
import Smooth
main = dotests "SmoothTests $Revision: 1.3 $"
         [vecWithin (smooth []) "~=~" []
         ,vecWithin (smooth [(Three 1.0 2.0 3.0)]) "~=~" [2.0]
         ,vecWithin (smooth [(Three 2.0 1.0 3.0)]) "~=~" [2.0]
         ,vecWithin (smooth [(Three 2.0 3.0 1.0)]) "~=~" [2.0]
         ,vecWithin (smooth [(Three 3.0 2.0 1.0)]) "~=~" [2.0]
         ,vecWithin (smooth [(Three 3.0 1.0 2.0)]) "~=~" [2.0]
         ,vecWithin (smooth [(Three 1.0 3.0 2.0)]) "~=~" [2.0]
         ,vecWithin (smooth [(Three 1.0 3.0 2.0), (Three 86.0 6.0 99.0)])
              "~=~" [2.0, 63.666666666666664]
         ,vecWithin (smooth [(Three 1.0 3.0 2.0), (Three 3.0 2.0 4.0),
                             (Three 2.0 4.0 6.0)])
           "~=~" [2.0, 3.0, 4.0]
         ,vecWithin (smooth [(Three 1.0 3.0 2.0), (Three 3.0 2.0 4.0),
                             (Three 2.0 4.0 6.0), (Three 4.0 6.0 8.0)])
           "~=~" [2.0, 3.0, 4.0, 6.0]      ]
```

Hint: don't hesitate to use helping functions in your solution if you wish.