

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

- A. ((([[]]:5):0):2):1)
- B. (5:(0:(2:(1:[])))
- C. (5:0) ++ (2:(1:[]))
- D. (((5 ++ 0) ++ 2) ++ 1)
- E. (5 ++ (0 ++ (2 ++ 1)))

2. (10 points) [UseModels] In Haskell, write the function:

```
scale :: Integer -> [Integer] -> [Integer]
```

that takes an Integer, factor, and a list of Integers, lst, and returns a list of Integers that is like lst, except that each element in the result is factor times the corresponding element of lst. The following are examples, written using the Testing module from the homework.

```
tests :: [TestCase [Integer]]
tests = [(eqTest (scale 32 []) "==" [])
        ,(eqTest (scale 17 [10]) "==" [170])
        ,(eqTest (scale 6 [1,2,1]) "==" [6,12,6])
        ,(eqTest (scale 50 [1 .. 7]) "==" [50, 100, 150, 200, 250, 300, 350])
        ,(eqTest (scale 3 [1,2 .. 100]) "==" [3,6 .. 300])
        ,(eqTest (scale 31 [10,11 .. 1000]) "==" [310,341 .. 31000])
        ]
```

3. (10 points) [Concepts] [UseModels] Consider the data type Outcome defined below.

```
data Outcome = StrictlyIncreasing | AllEqual | StrictlyDecreasing | Unordered
               deriving (Eq, Show)
```

In Haskell, write the function

```
order :: (Integer, Integer, Integer) -> Outcome
```

which takes a triple of integers, (i, j, k), and returns an Outcome that says how i, j, and k are ordered from left to right. That is: it returns StrictlyIncreasing if $i < j$ and $j < k$, it returns AllEqual if $i == j$ and $j == k$, it returns StrictlyDecreasing if $i > j$ and $j > k$, and otherwise it returns Unordered. The following are examples, written using the Testing module from the homework.

```
tests :: [TestCase Outcome]
tests = [(eqTest (order (3,4,10)) "==" StrictlyIncreasing)
        ,(eqTest (order (7,7,7)) "==" AllEqual)
        ,(eqTest (order (7,2,0)) "==" StrictlyDecreasing)
        ,(eqTest (order (9999,100,10000000)) "==" Unordered)
        ,(eqTest (order (99999,99999,99999)) "==" AllEqual)
        ,(eqTest (order (3,7,4)) "==" Unordered)    ]
```

4. [UseModels] In this problem you will write a function in two different ways. The function you are to write in Haskell is

```
averages :: [(Double, Double, Double)] -> [Double]
```

which takes a list of triples of Doubles, `lst`, and returns a list of Doubles such that each element of the result is the average (i.e., the arithmetic mean) of the corresponding triple. The following are examples, written using the `FloatTesting` module from the homework.

```
tests :: [TestCase [Double]]
tests =
  [(vecWithin (averages []) "~=~" [])
  ,(vecWithin (averages [(3.0,4.0,5.0)]) "~=~" [4.0])
  ,(vecWithin (averages [(4.0,5.0,3.0)]) "~=~" [4.0])
  ,(vecWithin (averages [(4.0,5.0,3.0),(5.0,15.0,10.0)]) "~=~" [4.0,10.0])
  ,(vecWithin (averages [(4.0,5.0,3.0),(5.0,15.0,10.0)]) "~=~" [4.0,10.0])
  ,(vecWithin (averages [(4.0,5.0,3.0),(5.0,15.0,10.0),(81.3,86.0,99.0)])
    "~=~" [4.0,10.0,(81.3+86.0+99.0)/3.0])
  ,(vecWithin (averages [(81.3,86.0,99.0),(107.0,100.0,105.3),(0.0,0.0,0.0),
    (22.5,34.4,9.0),(8.0,16.0,32.0),(1.0,-0.5,-0.25)])
    "~=~" [(81.3+86.0+99.0)/3.0,(107.0+100.0+105.3)/3.0,0.0,
    (22.5+34.4+9.0)/3.0,(8.0+16.0+32.0)/3.0,0.25/3.0]) ]
```

- (a) (5 points) Write `averages` using a list comprehension.

- (b) (10 points) Write `averages` without using a list comprehension, by writing out the recursion explicitly (without using `map` or other higher-order functions).

5. (15 points) [Concepts] [UseModels] In Haskell write a function

```
duplicate :: [a] -> [a]
```

which for all types `a`, takes a list of elements of type `a`, `lst`, and returns a list that is twice as long as `lst` and in which each element of `lst` is duplicated, and so appears twice in the result. The following are tests. In the tests, recall that `Strings` in Haskell are lists of `Chars`, so it is the `Chars` in the list that are to be duplicated.

```
tests :: [TestCase String] -- recall String = [Char]
tests =
  [(eqTest (duplicate []) "==" [])
  ,(eqTest (duplicate ['a','b','a']) "==" ['a','a','b','b','a','a'])
  ,(eqTest (duplicate "acid") "==" "aacciidd")
  ,(eqTest (duplicate "mississippi") "==" "mmiissssiissssiipppii")
  ,(eqTest (duplicate "even spaces!") "==" "eevveenn ssppaacceess!!")
  ]
```

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

```
count :: Char -> [Char] -> Integer
```

which takes a Char, what, and a list of Chars (i.e., a String), s, and returns the number of times that what occurs in s. The following are examples.

```
tests :: [TestCase Integer]
tests = [(eqTest (count 'c' []) "==" 0)
        ,(eqTest (count 'c' ['d','a','d','a']) "==" 0)
        ,(eqTest (count 'c' "cade") "==" 1)
        ,(eqTest (count 'c' "brocade") "==" 1)
        ,(eqTest (count 'c' "succinct") "==" 3)
        ,(eqTest (count 'x' "xylem") "==" 1)
        ,(eqTest (count 'i' "mississippi") "==" 4)
        ,(eqTest (count 's' "mississippi") "==" 4)
        ,(eqTest (count ' ' "now is the time for all") "==" 5) ]
```

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

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

In this problem you will write the function

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

This function take a key (of some equality type `a`), and a val (of some type `b`), and binary relation (i.e., a list of pairs), `rel`, and returns a binary relation that is just like `rel` except that all pairs (k, v) in `rel` such that `key == k` are replaced in the result by (key, val) . The following are examples.

```
tests :: [TestCase (BinaryRelation String Integer)]
tests =
  [(eqTest (update "happy" 6000 []) "==" [])
  , (eqTest (update "happy" 6000 [("happy",3),("sad",2)])
    "==" [("happy",6000),("sad",2)])
  , (eqTest (update "Scott" 2014 [("Scott",2010),("Crist",2006),("Bush",2002)])
    "==" [("Scott",2014),("Crist",2006),("Bush",2002)])
  , (eqTest (update "Voyager" 130000000000
    [("Cassini",120000000000),("Voyager",7),("Curiosity",60000000)])
    "==" [("Cassini",120000000000),("Voyager",130000000000),("Curiosity",60000000)])
  ]
```

8. (15 points) [UseModels] In Haskell, write the function

```
sumSublists :: [[Integer]] -> Integer
```

that takes a list of lists of Integers, `nss`, and returns the sum of all the Integers in `nss`. The following are examples.

```
tests :: [TestCase Integer]
```

```
tests =
```

```
  [(eqTest (sumSublists []) "==" 0)
  ,(eqTest (sumSublists [[]], []) "==" 0)
  ,(eqTest (sumSublists [[4,0],[2,0]]) "==" 6)
  ,(eqTest (sumSublists [[2,0]]) "==" 2)
  ,(eqTest (sumSublists [[1..10],[10..20]]) "==" 220)
  ,(eqTest (sumSublists [[5,-10,15],[35,30,25,20],[55,50,40,45,10]]) "==" 320)
  ,(eqTest (sumSublists [[1],[2],[3],[],[4],[5,6],[7,8,9,10]]) "==" 55)  ]
```