CSE 262: Programming Languages

Fall 2016

Homework 6: **Due on Nov 4th at 9pm on CourseSite.**

READINGS: Learn You a Haskell: Chapter 6. Real World Haskell: Chapter 4.

SUBMISSION REQUIREMENTS: Submit a text file (.hs/.txt) of your code. **Do not submit a screenshot** of it, you will not receive credit if you do. Your assignment must work on the sunlab machines.

NOTES: Argument order of functions must be in the order specified by the question definition.

1. (10 points): Write the recursive function extract. Given two elements of type x, "min" and "max', and a list of x as input. Return the elements of the list that are not within the range from [min, max] (inclusive) in their original order.

   ```
   extract 3 5 [1,5,4,6,3,2,1] -> [1,6,2,1]
   extract 'a' 'e' "apple" -> "ppl"
   ```

2. (10 points): Write the function superimpose. Given two lists of ints, for each pair of elements, return the weighted average of the pair, such that we take (2*largest) + smaller and divide by 3. This should be a single line, using some sort of map/mapWith/zip/zipWith (not including function definition).

   ```
   superimpose [1,2,3,4] [7,8,9,10] -> [5.0,6.0,7.0,8.0]
   superimpose [6,4,5] [9,10,2] -> [8.0,8.0,4.0]
   ```

3. (10 points): Write the function supersuperimpose. Given two lists of lists (essentially, a matrix) of ints, do the same thing as superimpose. Same rules, single line, etc.

   ```
   supersuperimpose [[1,2,3,4],[6,4,5,6],[1,2,3,4]]
       [[7,8,9,10],[9,10,2,9],[7,8,9,10]] ->
   [[5.0,6.0,7.0,8.0],[8.0,8.0,4.0,8.0],[5.0,6.0,7.0,8.0]]
   ```

4. (10 points): Write the recursive function maxAndCount. Given a list, return a tuple with the maximum element of that list as the first element of the tuple and the total number of occurences as the second element. Do not create a helper function for this. You will want to use a where statement.

```
maxAndCount [] -> *** Exception: Empty List
maxAndCount "apples" -> ('s',1)
maxAndCount "apple" -> ('p',2)
maxAndCount [1..20] -> (20,1)
maxAndCount [mod x 4 | x <- [1..50]] -> (3,12)
maxAndCount [1,10,2,10,3,4] -> (10,2)
maxAndCount [1,10,2,10,3,4,10] -> (10,3)
```

5. (10 points): Define the length of a list function (as lengthLambda) using only lambda expressions. lengthLambda should not define any arguments.

```
lengthLambda "apple" -> 5
lengthLambda [3,5..20] -> 9
```

6. (10 points): Write the recursive function adjuster. Given a list of type x, an int and an element of type x, either remove from the front of the list until it is the same length as int, or append to the end of the list until it is the same length as the value specified by the int.

```
adjuster [1..10] (-2) 2 -> *** Exception: Invalid Size
adjuster [1..10] 0 2 -> []
adjuster "apple" 10 'b' -> "applebbbbb"
adjuster "apple" 5 'b' -> "apple"
adjuster "apple" 2 'b' -> "le"
adjuster [] 3 (7,4) -> [(7,4),(7,4),(7,4)]
```

7. (10 points): Write the recursive function insertSort. Given a list of type x and an element of type x insert the element into the list such that the list is in ascending order. Duplicates are allowed. The input list will be properly sorted.

```
insertSort [] 3 -> [3]
insertSort "btt" 'u' -> "bttu"
foldl insertSort [] [] -> []
foldl insertSort [] [3,1,5,5,3,0,1,8,4] ->
[0,1,1,3,3,4,5,5,8]
```

8. (10 points): Write the recursive function wordMaker. Given a String, return a list of Strings containing each space separated word from the input String. You only have to handle the space character and you will not be given more than one space in a row. Ideally, do this without looking at the String more than once.

```
wordMaker "" -> []
wordMaker "app" -> ["app"]
wordMaker "apple core" -> ["apple","core"]
wordMaker "apple core maker" -> ["apple","core","maker"]
```

9. (10 points): Write lengthFold, only using some fold function that you provide a lambda function, solve for the length of a list.

```
lengthFold "apple" -> 5
lengthFold [3,5..20] -> 9
```

10. (10 points): Write the recursive function depth. Given our Tree datatype, return the depth of the tree. An empty tree should return 0. A single root node Tree should return 1.

```
let treeCons x = (\x -> foldl (flip insertTree) Empty x) x
depth (treeCons []) -> 0
depth (treeCons [5,4,6,3,7,1]) -> 4
depth (treeCons [1,2,5,8,9,4,7]) -> 5
depth (treeCons [5,4,6,3,7,1,2,5,8,9,4,7,8,5,3,4]) -> 6
```