**Coverage & Logistics**

This homework covers material through the seventh chapter of *Haskell: The Craft of Functional Programming* (HCFP).

This homework is officially due in class on **Thursday, March 3**. However, it comes with an automatic extension: anything submitted to the CIS 252 bin near CST 4-226 by **noon on Friday, March 4** will be accepted as on time.

You may work singly or in pairs on this assignment.

**What to turn in:** You should turn in a hard copy of your source code and a transcript that demonstrates convincingly that your code is correct.

**Background (Read carefully before starting)**

*An important caveat:* If you try to cut-and-paste code from this writeup, you’ll likely run into problems because of differing character encodings. If you run into problems, be prepared to type things from scratch.

A **bag** is an abstract data type that is like a set, except that it can contain multiple copies of the same element. We can represent a bag in Haskell by a list of pairs: each pair contains the element, plus a positive integer indicating how many copies of that element are in the bag. For the purposes of this assignment, we’ll consider only bags of characters, and we introduce the following type synonym:

\[
\text{type Bag = [(Char, Int)]}
\]

(You will need to include this type synonym near the top of your code.)

In the problems that follow, you must maintain the following two invariants (that is, if your function’s argument satisfies these properties, then the function’s result should also satisfy these properties):

- No bag contains two pairs with the same character.
  
  Thus \([(‘a’,2),(‘x’,5),(‘a’,1)]\) is not a valid bag, but \([(‘a’,3),(‘x’,5)]\) is.

- No bag contains a pair with a number smaller than 1.
  
  Thus \([(‘a’,0),(‘x’,5)]\) is not a valid bag, but \([(‘x’,5)]\) is.

The order in which unique elements appear is unimportant, however: both \([(‘a’,3),(‘x’,5)]\) and \([(‘x’,5),(‘a’,3)]\) are valid.

In the examples that follow, I make use of the following definitions (you don’t need to use them):

\[
\begin{align*}
\text{bag1, bag2 :: Bag} \\
\text{bag1 = [(‘z’,1), (‘e’, 2), (‘k’,1)]} \\
\text{bag2 = [(‘y’,2), (‘a’,1), (‘n’,1), (‘c’,1), (‘e’,1)]}
\end{align*}
\]

### Exercises

Some of these functions are easier to write recursively, and others may be easier to write using list comprehensions or built-in (or previously defined!) functions. Unless otherwise specified, you may use whichever strategy you prefer. However, you should be striving for simple, easy-to-read code: if your code is unduly complicated, you may lose some points.

1. Write a Haskell function `addToBag::Char -> Bag -> Bag` that takes a character and a bag, and returns the result of adding one copy of that character to the bag. For example:

   - Main> addToBag ‘y’ bag1
   - \([(‘z’,1), (‘e’,2), (‘k’,1), (‘y’,1)]\)
   - Main> addToBag ‘y’ bag2
   - \([(‘y’,3), (‘a’,1), (‘n’,1), (‘c’,1), (‘e’,1)]\)

2. Write a Haskell function `removeFromBag::Char -> Bag -> Bag` that takes a character and a bag, and returns the result of removing one copy of that character from the bag. For example:

   - Main> removeFromBag ‘e’ bag1
   - \([(‘z’,1), (‘e’,1), (‘k’,1)]\)
   - Main> removeFromBag ‘e’ bag2
   - \([(‘y’,2), (‘a’,1), (‘n’,1), (‘c’,1)]\)
   - Main> removeFromBag ‘a’ bag1
   - \([(‘z’,1), (‘e’,2), (‘k’,1)]\)

3. Write a Haskell function `catalog::String -> Bag` that takes a string and returns a bag containing all of the characters (including spaces and punctuation) that occur in the string.
For example, your function should have the following behavior (the order in which elements appear is unimportant):

```
Main> catalog "She sells seashells down by the seashore."
[(' ',1),('e',7),('r',1),('o',2),('h',4),('s',7),
 ('a',2),('t',6),('t',1),('y',1),('b',1),('n',1),
 ('w',1),('d',1),('l',4),('S',1)]
```

4. Write a Haskell function `unique :: String -> String` such that `unique cs` returns a string that contains those characters that occur exactly once in `cs`.

For example, your function should have the following behavior (again, the order of characters in the resulting string is unimportant):

```
Main> unique "Zeroes mean so much"
"hcunarZ"
```

5. Write a Haskell function `validBag :: [(Char,Int)] -> Bool` such that `validBag pairs` determines whether `pairs` satisfies both of the bag invariants described previously (i.e., no two pairs with the same character, and all numbers greater than 0).

For example, your function should have the following behavior:

```
Main> validBag bag1
True
Main> validBag bag2
True
Main> validBag [( 'a' ,2),('x',5),('a',1)]
False
Main> validBag [( 'a',0),('x',5)]
False
```

6. Write a Haskell function `combineBags :: Bag -> Bag -> Bag` that takes two bags and combines all of their elements into a single bag.

For example, your function should have the following behavior (once again, the order of items in the resulting bag is unimportant):

```
Main> combineBags bag1 bag2
[( 'y',2),('a',1),('n',1),('c',1),('e',3),('z',1),('k',1)]
```