Coverage & Logistics

This homework covers material through the seventh chapter of *Haskell: The Craft of Functional Programming* (HCFP). It is officially due in class on **Thursday, February 16**, but it comes with an automatic extension: anything submitted to the CIS 252 bin near CST 4-226 by **noon on Friday, February 17** will be accepted as being on time.

You may work singly or in pairs on this assignment.

What to turn in

The same general grading criteria from Assignment 2 applies for this assignment as well. You should submit hard copies of (1) your source code and (2) a clean transcript demonstrating convincingly that your code is correct. As always, include a completed disclosure cover sheet.

Exercises

To receive maximal credit, ***do not use*** `fst`, `snd`, `head`, or `tail` for these functions. Instead, use Haskell's pattern-matching capabilities as discussed in lecture. *Include an `import Data.Char` directive at the top of your file (you'll need it for `shout`).*

1. Write a **recursive** Haskell function

   ```
   myProduct :: [Integer] -> Integer
   ```

   such that `myProduct ns` returns the product of all of the numbers in `ns`; following by mathematical convention, the product of zero numbers is 1. For example, `myProduct [3,6,2,10]` returns `360`, `myProduct [7,18,2,0,9]` returns `0`, and `myProduct []` returns `1`.

2. Write a **recursive** Haskell function

   ```
   shout :: String -> String
   ```

   such that `shout str` returns the string obtained by replacing all lowercase letters in `str` by their uppercase equivalents; all other characters are unchanged. For example, `shout "Let's go, Orange!"` returns `"LET'S GO, ORANGE!"`.

3. Write a **recursive** Haskell function

   ```
   zap :: Char -> String -> String
   ```

   such that `zap c cs` returns the string obtained by removing all occurrences of `c` from `cs`. For example, `zap 'a' "abbadaab"` returns `"bbdb"`.

4. Write a **recursive** Haskell function

   ```
   pairUp :: [a] -> [(a,a)]
   ```

   such that `pairUp xs` returns the list obtained by pairing up the first two elements of `xs`, then the third and fourth elements, and so on; if `xs` has an odd number of elements, the final element is paired with itself.

   For example, `pairUp [3,5,2,9]` returns `[(3,5),(2,9)]`, whereas `pairUp "abcde"` returns `[('a','b'),('c','d'),('e','e')]`.

5. Write a **recursive** Haskell function

   ```
   neighbors :: [a] -> [(a,a)]
   ```

   such that `neighbors xs` returns a list containing all pairs of "neighboring" elements from `xs`. For example, `neighbors [3,5,2,9]` returns `[(3,5),(5,2),(2,9)]`, whereas `neighbors "abcde"` returns `[('a','b'),('b','c'),('c','d'),('d','e')]`.

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

   In the problems that follow, you must maintain the following two *invariants* (that is, you should assume that these properties hold of any bag your functions receive as input, but you must ensure that your functions' results also satisfy these properties):

   - **No character appears in two or more pairs within the same bag.**

     Thus `[('a',2),('x',3),('a',1)]` is not a valid bag, because the character `'a'` appears in two distinct pairs. In contrast, `[('a',3),('x',3)]` is a valid bag (intuitively, it represents a bag with three copies of `'a'` and three copies of `'x'`).

- **No number smaller than** 1 **appears in any pair of a bag.** 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 equally valid (both represent a bag that contains three copies of `'a'` and five copies of `'x'`).

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

```
bag1, bag2, bag3 :: [(Char,Int)]
bag1 = [('z',1), ('e', 2), ('k',1)]
bag2 = [('y',2), ('a',1), ('n',1), ('c',1), ('e',1)]
bag3 = [('j',1), ('o',1), ('u',1), ('l',1), ('e',1)]
```

Important note: don't use cut-and-paste to copy these into your code, because you'll run into problems. If you want to use these definitions, you should type them in by hand.

(a) Write a **recursive** Haskell function

```
bagCount :: [(Char,Int)] -> Int
```

such that `bagCount bag` returns the total number of items in `bag`. For example:

```
Main> bagCount bag1
4
Main> bagCount bag2
6
Main> bagCount bag3
5
```

(b) Write a **recursive** Haskell function

```
addToBag :: Char -> [(Char,Int)] -> [(Char,Int)]
```

such that `addToBag ch bag` and returns the bag obtained by adding one copy of `ch` to `bag`. For example:

```
Main> addToBag 'y' bag1
[('z',1),('e',2),('k',1),('y',1)]
Main> addToBag 'e' bag1
[('z',1),('e',3),('k',1),('y',1)]
```

You should assume that the initial input is a valid bag, and you must ensure that result is also a valid bag.

(c) Write a **recursive** Haskell function

```
removeFromBag :: Char -> [(Char,Int)] -> [(Char,Int)]
```

such that `removeFromBag ch bag` returns the bag obtained by removing one copy of `ch` from `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)]
```

You should assume that the initial input is a valid bag, and you must ensure that result is also a valid bag.

None of these functions requires more than five lines of code (not counting the type declaration), and most of them can be written using only two or three lines. If your answers are significantly longer than that, then your approach is too complicated: please ask for assistance.