Discrete Math & Functional Programming— CSCI 054— Spring 2024
Instructor: Osborn
Homework 4 (29 point(s))
Due: 10:00PM on Sunday

- The template file `week04-ps-template.hs` has the types for each of the functions. There is a comment above each that simply names the function. Make sure to augment that comment with a description of what the function does. This is in addition to other comments you might want to include in your file. And please don't change the type declarations!
- You are free to use functions that are built-in to Haskell (e.g. `length` or `map`) or that are in `Data.Char` (e.g. `toUpper`, or `ord` or `chr`). (Yes, those are all hints!)
- You may not import any modules (other than `Data.Char`, which is already imported in the template). **You may not use list comprehensions.**

1. **[19 point(s)] Substitution Ciphers, part 2**

   For the last problem you wrote a function to encrypt using Caesar ciphers, which use a simple shift of the alphabet. For this problem we'll look at a slightly more sophisticated system.

   Suppose you and your fellow spy want to exchange secret messages. You agree on a pangram, a sentence that contains all the letters of the alphabet, to use as a key. The order of the letters in the pangram will specify the letter-for-letter substitution you will use. Here are a few examples of pangrams:

   ```
   A QUICK BROWN FOX JUMPS OVER THE LAZY DOG
   CRAZY FREDRICK BOUGHT MANY VERY EXQUISITE OPAL JEWELS
   GRUMPY WIZARDS MAKE A TOXIC BREW FOR THE JOVIAL QUEEN
   ```

   Suppose you chose the third of these. Then to encode your message, you would replace `A` with `G`, replace `B` with `R`, and so forth. Pangrams that use each letter only once are rare and hard to remember, so we have to eliminate duplicate letters. For the third "grumpy" pangram, we would have the following translation.

   ABCDEFGHIJKLMNOPQRSTUVWXYZ␣

   GRUMPY␣WIZADSKETOXCBFHJVLQN

   Now let's write code that allows us to encrypt and decrypt a message using this system. To do this you should implement the following functions:

- Write a function `keepFirst :: [a] -> [a]` which removes duplicates from a list by keeping the *first* occurence of each item. For example, `keepFirst [1,3,2,1]` should evaluate to `[1,3,2]`.
- Use `keepFirst` to write a function `subst :: String -> [(Char, Char)]` that takes a pangram and returns a list of pairs of characters. Each pair will specify one substitution in the cipher. For example, using the "GRUMPY" pangram:

  `subst "GRUMPY...QUEEN"` would return

  `[('A','G'),('B','R'),...,('␣','N')]`

  You may assume that the given pangram is presented as a string with only uppercase letters and spaces.
- We are now ready to assemble the parts of a substitution cipher. Write a function `substEncipher :: String -> String -> String` which takes a pangram key as the first parameter and a plaintext message as the second, then returns the encrypted string. The pangram key and plaintext may be in either lower or uppercase and they might include punctuation, so you will want to `sanitize` them.
- Write the corresponding `substDecipher :: String -> String -> String` which takes a pangram key as the first parameter and an encoded string as the second parameter and returns the plaintext message. The pangram key will be the one used to encipher. All you have to do is invert the substitution.

  Again, the pangram key and plaintext may be in either lower or uppercase, so you will want to `sanitize` them.

Once your functions are working properly, you will be able to encrypt a message with the key and then decrypt it to get the original message back.

2. **[10 point(s)] More higher-order fun(ctions)**

   Implement the following functions using map, filter, or fold (i.e., without using recursion directly).

   (a) `double :: [Integer] -> [Integer]`, which doubles every number in a list.

   (b) `countOdd :: [Integer] -> Int`, which counts how many odd numbers are in a list of numbers. You can do this with a fold or a filter and another function.

   (c) `filter'`, implementing filter using fold.

   (d) `maxInt`, our old friend from week 1, using fold. Hint: Define your function like `maxInt (l:ls) = ...` so you know you have at least one item.