CIS 194: Homework 2

Due Friday, September 12

Be sure to write functions with exactly the specified name and type signature for each exercise (to help us test your code). You may create additional helper functions with whatever names and type signatures you wish.

You are allowed (and *encouraged*) to use functions in the `Data.List` standard library. You can find a list of these functions at `http://hackage.haskell.org/package/base-4.7.0.1/docs/Data-List.html`. The type signatures tell you much about what these functions do. To experiment with them, just say `import Data.List` in GHCi and start trying out expressions. Remember that a `String` is just a list of characters (that is, a `[Char]`), and so provides conveniently-written test data.

You will see that some of the type signatures have something like `Eq a =>` appearing in them. For now, completely ignore this bit of the type signature, called a *typeclass constraint*. You'll learn much more about these constraints later.

Local variables

The introduction in the first class meeting did not discuss local variables. Here, we see a few examples of local variables in case these constructs are useful for you in writing your homework solutions.

`let` *expressions*

To define a local variable scoped over an expression, use `let`:

```
strLength :: String -> Int
strLength []     = 0
strLength (_:xs) = let len_rest = strLength xs in
                   len_rest + 1
```

In this case, the use of the local variable is a little silly (better style would just be `1 + strLength xs`), but it demonstrates the use of `let`. Don't forget the `in` after you've defined your variable!

`where` *clauses*

To define a local variable scoped over multiple guarded branches, use `where`:

```
frob :: String -> Char
frob []  = 'a'   -- len is NOT in scope here
```

```
frob str
  | len > 5   = 'x'
  | len < 3   = 'y'
  | otherwise = 'z'
  where
    len = strLength str
```

Note that the `len` variable can be used in any of the alternatives immediately above the `where` clause, but not in the separate top-level pattern `frob [] = 'a'`. In idiomatic Haskell code, `where` is somewhat more common than `let`, because using `where` allows the programmer to get right to the point in defining what a function does, instead of setting up lots of local variables first.

Haskell Layout

Haskell is a *whitespace-sensitive* language. This is in stark contrast to most other languages, where whitespace serves only to separate identifiers. (Haskell shares this trait with Python, which is also whitespace-sensitive.) Haskell uses indentation level to tell where certain regions of code end, and where new statements appear. The basic idea is that, when a so-called *layout herald* appears, GHC looks at the next thing it sees and remembers its indentation level. A later line that begins at the exact same indentation level is considered another member of the group, and a later line that begins at a lesser (more to the left) indentation level is not part of the group.

The layout heralds are `where`, `let`, `do`, `of`, and `\ case`. Because Haskell modules begin with `module Name where`, that means that the layout rule is in effect over the declarations in the file. This means that the following is no good:

```
x :: Int
x =
5
```

The problem is that the `5` is at the same indentation level (zero) as other top-level declarations, and so GHC considers it to be a new declaration instead of part of the definition of `x`.

The layout rule is often a source of confusion for newcomers to Haskell. But, if you get stuck, return to this decription (or, any of the many online) and re-read — often, if you think carefully enough about it, you'll see what's going on.

When calculating indentation level, tabs in code (you don't have any of these, do you?!?) are considered with tab stops 8 characters apart, regardless of what your editor might show you. This potential confusion is why tabs are a terrible, terrible idea in Haskell code.

Accumulators

Haskell's one way to repeat a computation is recursion. Recursion is a natural way to express the solutions to many problems. However, sometimes a problem's structure doesn't exactly match Haskell's structure. For example, say we have a list of numbers — that is, a `[Int]`. We wish to sum the elements in the list, but only until the sum is greater than 20. After that, the rest of the numbers should be ignored. Because recursion over a list builds up the result from the end backward, a naive recursion will not work for us. What we need is to keep track of the running sum as we go deeper into the list. This running sum is called an *accumulator*.

Here is the code that solves the stated problem:

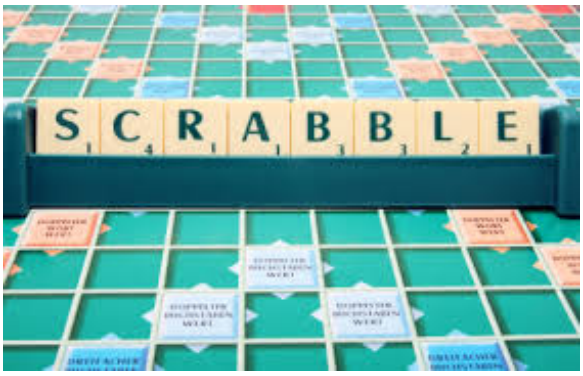
```
sumTo20 :: [Int] -> Int
sumTo20 nums = sumTo20Helper 0 nums   -- the acc. starts at 0

sumTo20Helper :: Int -> [Int] -> Int
sumTo20Helper acc [] = acc   -- empty list: return the accumulated sum
sumTo20Helper acc (x:xs)
  | acc >= 20 = acc
  | otherwise = sumTo20Helper (acc + x) xs
```

Example: `sumTo20 [4,9,10,2,8] == 23`

This technique of using an accumulator may be helpful in this assignment.

Scrabble

It's time to have some fun!



You will be writing functions to help in a computer player for a Scrabble game. Though familiarity with the rules is *not* assumed for this assignment, you may wish to read them at `http://www.hasbro.com/scrabble/en_US/discover/rules.cfm`.

We will be using some type definitions to make this work nicely. Please download the HW02.hs file (linked from the "Lectures" page) and edit that file. You will also need the file Words.hs, which defines a list of all possible Scrabble words. Make sure these files are in the same directory, so that the files can see each other.

Many of the functions return lists of words. The order of the words within the list does not matter.

Exercise 1 The first function you will write tests whether a certain word is formable from the tiles in a Scrabble hand. That is, given a `String` and a list of `Char`s, can the `String` be formed from the `Char`s, taking any duplicates into account?

As described in *HW02.hs*, we use a type synonym `type Hand = [Char]` to talk about Scrabble hands, to make it very clear where order matters (in words) and where it does not (in hands).

```
formableBy :: String -> Hand -> Bool
```

Example: `formableBy "fun" ['x','n','i','f','u','e','l'] == True`

Example: `formableBy "haskell" ['k','l','e','h','a','l','s'] == True`

Example: `formableBy "haskell" ['k','l','e','h','a','y','s'] == False`

Hint: Start by thinking what this should do if the string to be matched is empty. Then, what should it do if the string is non-empty? The `elem` and `delete` functions from `Data.List` may be helpful here.

Exercise 2 Now, using `formableBy`, write a function `wordsFrom` that gives a list of all valid Scrabble words formable from a certain hand. The `Words` module (imported by the *HW02.hs* you downloaded) allows you to use `allWords :: [String]`, which contains all valid Scrabble words.

```
wordsFrom :: Hand -> [String]
```

Example: `wordsFrom ['a','b','c','d'] == ["ab","ad","ba","bad","cab","cad","dab"]`

Example:
```
wordsFrom ['h','e','l','l','o'] ==
    [ "eh","el","ell","he","hell","hello","helo"
    , "ho","hoe","hole","lo","oe","oh","ole" ]
```

This function will likely require a helper function in order to process all of the elements in `allWords`.

Exercise 3 Most plays in Scrabble do not build completely fresh words from the tiles in one's hand. All Scrabble plays (except the first) have to build on existing tiles. Often, there is a place that a player wants to make a word, but that player must figure out if a word can fit. Your next functions will help to solve this part of the problem.

First, we must have the idea of a *template*. A template is a string containing some letters and some question marks. The question marks represent open spaces on the board that will get filled in by the letter in a player's hand. The letters in the template represent letters that already appear on the board. They must appear in exactly the same positions in the final words produced. So, the template `??r?` represents a board position where the third letter of a four-letter word must be `r`. If you have the letters available, you could play `care` or `burp` there (among many other words).

Write a function `wordFitsTemplate` that checks to see if a given word matches a template, given a set of tiles available:

```
wordFitsTemplate :: Template -> Hand -> String -> Bool
```

Example: `wordFitsTemplate "??r?" ['c','x','e','a','b','c','l'] "care" == True`

Example: `wordFitsTemplate "??r?" ['c','x','e','w','b','c','l'] "care" == False`

Example: `wordFitsTemplate "??r?" ['c','x','e','a','b','c','l'] "car" == False`

Example: `wordFitsTemplate "let" ['x','x'] "let" == True`

Exercise 4 Now, using that function, write another one that produces all valid Scrabble words that match a given template using a hand of available tiles. This will be similar to `wordsFrom`.

```
wordsFittingTemplate :: Template -> Hand -> [String]
```

Example: `wordsFittingTemplate "??r?" ['c','x','e','a','b','c','l'] ==`
`["acre","bare","carb","care","carl","earl"]`

Exercise 5 Now we must think about scoring, as not all words in Scrabble are created equal! The `Words` module, along with providing

`allWords`, provides `scrabbleValue :: Char -> Int` that gives the point value of any letter. Use that function to write a new function that gives the point value of any word.

```
scrabbleValueWord :: String -> Int
```

Example: `scrabbleValueWord "care" == 6`

Example: `scrabbleValueWord "quiz" == 22`

Exercise 6 You will use the `scrabbleValueWord` function to write a filtering function that takes a list of words and selects out only those that have the maximum point value. Note that there may be many words tied for the most points; your function must return all of them.

```
bestWords :: [String] -> [String]
```

Example: `bestWords (wordsFittingTemplate "??r?" ['c','x','e','a','b','c','l']) == ["carb"]`

Example: `bestWords ["cat", "rat", "bat"] == ["bat","cat"]`

Example: `bestWords [] == []`

A helper function with an accumulator may come in handy here, but there are other possible solutions.

Exercise 7 A Scrabble board is not a completely blank canvas. There are four kinds of special squares: double-letter, triple-letter, double-word, and triple-word. A letter played on a double- or triple-letter square gets its point value multiplied, and if any letter is played on a double- or triple-word square, the whole word's value gets multiplied. The effects multiply with each other, as appropriate. So, a play on both a double-word and a triple-word gets multiplied by 6. If one tile is on a double-letter and another is on a double-word, then that letter's value is multiplied by 4.

To represent these special squares, we use a new type `STemplate` (the `S` is for "square"). A stemplate is like a template, but it uses `'D'` and `'T'` to mark double- and triple-letter squares, respectively, and it uses `'2'` and `'3'` to mark double- and triple-word squares, respectively. Thus, the stemplate `?e??3` represents a place on the board where there is room for a 5-letter word, the second letter of which is an `e`, and the last letter of which falls on a triple-word square.

Write a function `scrabbleValueTemplate` that computes the value of playing a given word on a given template. In this function, you may assume that the word actually matches the template.

```
scrabbleValueTemplate :: STemplate -> String -> Int
```

Example: `scrabbleValueTemplate "?e??3" "peace" == 27`

Example: `scrabbleValueTemplate "De?2?" "peace" == 24`

Example: `scrabbleValueTemplate "??Tce" "peace" == 11`

Challenge: Extensible templates

(This problem is not worth credit, but is included for fun.)

The templates used above are a little silly, given the way Scrabble works. For example, if you could use the template `?e???`, then you could certainly use `?e??` and leave the last square blank.

Write a version of `wordFitsTemplate` that does not require that the word takes up the entire template.