CIS 194: Homework 2

Due Wednesday, February 4

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.

Mastermind

Figure 1: A typical Mastermind game board

Mastermind is a two-player code breaking game. One player is the *codemaker* and the other player is the *codebreaker*. At the start of the game, the codemaker chooses a secret code comprised of four colored pegs. Each peg is one of six colors: red, green, blue, yellow, orange, or purple. The codebreaker must figure out the code in as few turns as possible. Each turn, the codebreaker guesses the secret code. The codemaker then tells the codebreaker how many pegs in his guess are exact matches (the color appears in the same place in the secret code) and how many are non-exact matches (the color appears in a different position in the secret code. More information about the history and rules of Mastermind are available here `http://en.wikipedia.org/wiki/Mastermind_(board_game)`.

In this section, you will implement an algorithm that plays Mastermind. 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. Once again, you are encouraged to use functions from the `Data.List` module in order to make your code cleaner.

Exercise 1 In this exercise you will implement a function that returns the number of exact matches between the secret code and the codebreaker's guess.

```
exactMatches :: Code -> Code -> Int
```

Example: `exactMatches [Red, Blue, Green, Yellow] [Blue, Green, Yellow, Red] == 0`

Example: `exactMatches [Red, Blue, Green, Yellow] [Red, Purple, Green, Orange] == 2`

Exercise 2 Now you will write a function that returns the number of total matches between the secret code and the guess. This is a little bit more complicated than finding exact matches because we have to take in to account duplicate colors. For example, the sequences `[Red, Red, Blue, Blue]` and `[Red, Red, Green, Green]` only have 2 matches even though both of the `Red` pegs in the secret match both of the `Red` pegs in the guess. For this reason, we can't just scan the first list and count the occurences of each element in the second list. Instead, we will count the number of times that each peg appears in both lists and sum the minimum values for each color. In the example above, `Red` and `Blue` both occur twice and all of the other colors never appear in the first list and `Red` and `Green` occur twice in the second list. The number of matches is therefore 2 since `Red` occurs twice in both lists and all the other colors are not in both of the lists.

Before you can count the number of matches, you should implement the helper function:

```
countColors :: Code -> [Int]
```

This function takes in a `Code` and returns a list containing the numbers of times that each color in the list `colors` appears in the `Code`. The counts should appear in the same order that they occur in the list `colors`. As a sanity check, output should always have length 6 and the sum of all the entries should be equal to `length Code`.

Example: `countColors [Red, Blue, Yellow, Purple] == [1, 0, 1, 1, 0, 1]`

Example: `countColors [Green, Blue, Green, Orange] == [0, 2, 1, 0, 1, 0]`

Now you are ready to implement the main function:

```
matches :: Code -> Code -> Int
```

Example: `matches [Red, Blue, Yellow, Orange] [Red, Orange, Orange, Blue] == 3`

Exercise 3 A `Move` is a new datatype that is constructed with a `Code` and two `Int`s. The first `Int` is the number of *exact* matches that the `Code` has with the secret and the second `Int` is the number of *non-exact* matches [1]. Implement the function:

[1] Note that this value is slightly different than the one you calculated in Exercise 2, but it is related

```
getMove :: Code -> Code -> Move
```

The first `Code` is the secret, the second `Code` is guess, and the output is the resulting `Move`.

Example: `getMove [Red, Blue, Yellow, Orange] [Red, Orange, Orange, Blue] ==`
`Move [Red, Orange, Orange, Blue] 1 2`

Exercise 4 We will now define a concept that will be important in playing the Mastermind game. This is the concept of *consistency*; we say that a `Code` is consistent with a `Move` if the `Code` could have been the secret that generated that move. In other words, if the guess inside the `Move` has the same number of exact and non-exact matches with the provided `Code` as it did with the actual secret, then the `Code` is consistent with the `Move`. Define the function:

```
isConsistent :: Move -> Code -> Bool
```

Example: `isConsistent (Move [Red, Red, Blue, Green] 1 1) [Red, Blue, Yellow, Purple] == True`

Example: `isConsistent (Move [Red, Red, Blue, Green] 1 1) [Red, Blue, Red, Purple] == False`

Exercise 5 Now that we have the concept of consistency, we can filter a list of `Code`s to only contain those that are consistent with a given `Move`. This will be useful to us since our game solver will start with a list of all possible codes and gradually filter the list based on each new move until there is only one code left. Implement the function:

```
filterCodes :: Move -> [Code] -> [Code]
```

Exercise 6 As mentioned in Exercise 5, the final algorithm will start with a list of all possible codes and filter out the inconsistent ones. In order to do this, we first need to be able to generate a list of all the codes, ie all length n combinations of the 6 colors. In general, Mastermind games use codes of length 4, however in theory the code could be any length. We have not yet made any assumptions about the lengths of the codes, so why start now? Your function should take in a length[2] and return all `Code`s of that length:

[2] Haskell's type system is strong enough to encode the length of a list in its type, however it uses an advanced feature of Haskell called Generalized Algebraic Datatypes which is beyond the scope of this homework. Using this feature, we could write `allCodes` without giving the length as an input.

```
allCodes :: Int -> [Code]
```

Hint: This exercise is a bit tricky. Try using a helper function that takes in all the codes of length $n-1$ and uses it to produce all codes of length n. You may find the `concatMap` function helpful.

Exercise 7 We are now finally ready to write our Mastermind solver! There are many algorithms to solve a game of Mastermind, but most of them work in roughly the same way; start with all of the codes and keep making guesses until only one possible code remains. The tricky bit is actually *how* you choose your guesses. In this exercise, you will implement a fairly dumb algorithm that keeps track of the remaining consistent `Codes` in a list and always chooses the first element in the list as the next guess.

Is this algorithm guaranteed to converge to a single, correct answer? Yes! In fact, there is a simple one-line proof[3]. Every guess is either correct or incorrect. If it is correct, then the game is over, and if it is incorrect, then it will be filtered out. So, at least one `Code` is filtered out every round, and there are a finite number of `Code`s therefore the game will end in a finite number of rounds.

[3] "One-line proof" is a term used by CIS and math professors to signify that a proof is so trivial that it can fit on to one line. In fact, any proof can fit on to one line if enough details are omitted.

Your solver function should take in a secret `Code` and output a list of `Moves` that the computer used as clues to figure out the secret. Always start by guessing `[Red, Red, Red, ..., Red]`. This will make it easier for us to test your outputs.

```
solve :: Code -> [Move]
```

You will most likely need to write a helper function in order for this to work correctly.

Exercise 8 **(Optional)** In the previous exercise you implemented a relatively dumb algorithm for Mastermind. We proved that this algorithm will terminate after a *finite* number of turns, however in a real game, you would want to choose your guesses more carefully in order to minimize the number of turns it takes to guess the secret code.

There is an algorithm due to Donald Knuth that is guaranteed to win in five moves or fewer. A detailed description of the algorithm is available here `http://en.wikipedia.org/wiki/Mastermind_(board_game)#Five-guess_algorithm`. For kudos, you can implement this algorithm in Haskell.

Note, although the Five Guess algorithm terminates in fewer *turns* than the naïve one, it is much more computationally intensive and will take much longer to run in most cases. If you want test cases that run quickly, choose secret codes that are very close to the first guess.