$\lambda.5$

COURSEWORK I: MASTERMIND

The aim of this coursework is to implement the board game *Mastermind* in Haskell with the help of some skeleton code. The game is played by exactly two players: a *codemaker* and a *codebreaker*. At the start of the game, the codemaker makes up a code consisting of four coloured pegs. Pegs are also referred to as symbols. For example:

Yellow, Green, Green, Blue

Each colour (symbol) may be used any number of times in the code, as long as the code has no more than four pegs. There are six colours to choose from. The code is *not* disclosed to the codebreaker, whose objective it is to figure out what the code is. The codebreaker does this by repeatedly *guessing* what the code might be. For example, to start the codebreaker might guess the following code at random:

Green, Red, Blue, Blue

The codemaker then scores the guess according to the following rules:

- For each peg that is in the correct position and has the right colour, the codebreaker scores one coloured marker.
- For each peg that is the right colour but in an incorrect position, the codebreaker scores one white marker.

For example, for the above guess, the codebreaker would score one white marker for the green peg that is in the wrong position and one coloured marker for the blue peg that is in the right position. The codebreaker does *not* score a white marker for the second blue peg. In other words, at most one point is awarded for each peg in the code. The codebreaker then has to use this score to come up with a new guess for the code, which is then scored again, and so on. Once the codebreaker scores four coloured markers, the game is over and the two players switch roles.

5.1 Getting started

In order to get started with the coursework, you need to get hold of the skeleton code and ensure that it compiles successfully.

5.1.1 Obtaining the skeleton code

There are three different ways in which you can obtain the skeleton code for this coursework, which are all explained below alongside their advantages and disadvantages:

Option A: Private fork By following the GitHub Classroom link below, you can create a private fork of our git repository with the skeleton code. This requires a GitHub account, but has the advantage that you have your own private copy of our repository on GitHub that you can write to. That would allow then you to work easily share your work between machines in the labs and at home:

`https://classroom.github.com/a/gD7o5fXq`

Once you have accepted the assignment, you can then clone your fork of the skeleton code to your machine with the usual `git clone` command where `[username]` is your GitHub username:

```
$ git clone https://github.com/fpclass/1819-cswk1-[username]
```

Option B: Clone If you do not wish to create a GitHub account or host a copy of your repository there, then you could instead just clone our repository with:

```
$ git clone https://github.com/fpclass/cswk1
```

You will be able to `git commit` changes to your local copy of the repository, but you will not be able to `git push` them. This is sufficient if you are only planning to work on the coursework from one place (*e.g.* only the lab machines but not your personal computer).

Option C: Archive If GitHub should be unavailable or you do not have `git` installed your machine, you can download a `.zip` file with the skeleton code from the module website.

5.1.2 Working with the skeleton code

You may wish to verify that the code compiles and that all tests fail by entering the `cswk1` directory that was created and running `stack test`:

```
$ cd cswk1
$ stack test
```

Running `stack test` will compile your code, run a bunch of unit tests on it, and give you a rough indication of how complete your solution is (the more tests pass, the more complete it is). Running `stack bench` will run a set of benchmarks on your code. You can also use `stack build` to just compile your code and then `stack exec mastermind` to run the program. Alternatively, you can run `stack repl` to load up the REPL, which is useful for debugging.

The skeleton code contains a bunch of files, most of which you do not need to touch. The most important file is `src/Game.hs` which contains the definitions you will need to complete in order to implement the game. There are some definitions to get you started. Firstly, the number of pegs per code is defined as:

```
pegs :: Int
pegs = 4
```

Ideally, your solution should still work even if this number is modified. We represent colours using characters from `a` to `f` and refer to them as symbols:

```
type Symbol = Char

symbols :: [Symbol]
symbols = ['a'..'f']
```

Again, your solution should continue to work even if you modify how many symbols there are and which characters are used to represent them. A code is a list of symbols:

```
type Code = [Symbol]
```

Codes are scored using coloured and white markers. We define scores to be pairs of integers where the first component of the pair represents the number of coloured markers and the second component represents the number of white markers:

```
type Score = (Int, Int)
```

A player is either human or a computer:

```
data Player = Human | Computer
```

The initial codemaker is defined as a constant:

```
codemaker :: Player
codemaker = Human
```

You can change this value to determine who goes first. Finally, the computer's first guess is defined as:

```
firstGuess :: Code
firstGuess = "aabb"
```

You can change this value to change the computer's first guess, but note that values other than `"aabb"` may cause the computer to take more guesses to crack the code.

5.2 Five-guess algorithm

Donald Knuth described an algorithm for Mastermind which, for every code with four pegs, takes a computer no more than five guesses to solve. The algorithm works as follows:

1. Let S be a set of all possible codes (`"aaaa"`, `"aaab"`, ..., `"ffff"`).
2. Let the first guess be `"aabb"`.
3. Get the codemaker to score your guess.
4. If the score has four coloured markers, then the guess was correct.
5. Otherwise, remove all codes from S which would result in a different score. In other words, we know that the code is somewhere in S, so it can only be one which results in the same score for the guess as the one we got from the codemaker.
6. Find the next guess as follows. If there is only one code left in S, use it. Otherwise, for every possible code c (not just those left in S):
   (a) For each possible score s ($(0,1)$, $(0,2)$, ..., $(4,0)$):
      i. Determine how many other codes would be eliminated from S. That is, if the next guess were c and it would get a score of s, how many codes would that eliminate from S – i.e. how many codes with different scores would there be?

Choose the code which is guaranteed to eliminate the most options from S. This is calculated in the above step by calculating the minimum of eliminations for each code across all the possible scores it might get. In the case of multiple codes producing the same number of guaranteed eliminations, a code which is still a member of S should be picked over one which is not.

7. Go to Step 3.

5.3 Task

Complete all definitions in `src/Game.hs` so that the game works as described above and that the computer never takes more than five guesses to figure out a code. The following function stubs in `src/Game.hs` need to be implemented:

1. `correctGuess :: Score -> Bool`
   This function should determine whether a `Score` value represents a winning guess – *i.e.* one where the number of coloured markers matches `pegs` and there are no white markers.

2. `validateCode :: Code -> Bool`
   This function should determine whether a given `Code` value is valid: the code should contain `pegs`-many symbols and all the symbols should be elements of `symbols`.

3. `codes :: [Code]`
   This list should contain all possible codes of length `pegs` using elements from `symbols`. There should be no duplicates.

4. `results :: [Score]`
   This list should contains all possible scores for codes of length `pegs`. There should be no duplicates.

5. `score :: Code -> Code -> Score`
   This function should score a code according to the rules described above. This function should be commutative, so that it does not matter whether the code or the guess is given as first argument and vice-versa.

6. `nextGuess :: [Code] -> Code`
   This function should determine the next guess, given the current S represented as a list of codes.

7. `eliminate :: Score -> Code -> [Code] -> [Code]`
   This function should eliminate all codes from a given S, represented as a list

of codes, with the help of the most recent guess (the `Code` argument) and the score which was obtained for it from the codemaker (the `Score` argument).

5.4 Marking & submission

This coursework is worth 15% of the overall module mark. It will be marked out of 100% as follows:

- 20% for *correctness*. You gain full marks here if all parts of the coursework have been attempted and are correct. You may use `stack test` as a rough indication for whether this is the case, but there are some things the unit tests do not test for, so you should play the game and ensure that everything works as described.
- 20% for *documented understanding*. You should document your code with comments and explain how it works. You gain full marks if all code is documented and explained sufficiently well so that someone who is unfamiliar with your code can understand it.
- 20% for *elegance*. Definitions should be concise and readable, new functions should be introduced where needed, existing library functions used when applicable, etc.
- 20% for *performance and efficiency*. You will do well here if you use sensible data structures and your functions perform as little redundant computation as possible. You can test performance by running `stack bench` on different versions of your code to see how they compare.
- 20% for *improvements and extensions*. This is an opportunity for you to demonstrate creativity and advanced understanding. You could achieve this in many different ways, such as adding additional unit tests, functionality, improved algorithms, etc. You may wish to modify `exe/Main.hs` as well as other source files or even add new ones. You could also prove some properties about your game on paper. The amount of marks awarded will depend on the complexity and creativity of your extension(s) and improvement(s).

Submit a `.zip` or `.tar.gz` archive of the whole, completed project (not just `Game.hs`) through Tabula by noon on 7 February 2019:

`https://tabula.warwick.ac.uk/coursework/submission/1905b143-7b68-4f61-bf2d-5288934c6253`