MasterMind

Andres Löh

Difficulty: easy

The purpose of this assignment is to reimplement the Mastermind game – see the Wikipedia article at

```
http://en.wikipedia.org/wiki/Mastermind_%28board_game%29
```

for more information about the game.

The game is sufficiently small to fit into a single Haskell module, distributed in a file `MasterMind.hs`. The file contains a skeleton program: it runs and typechecks, but doesn't do anything really useful yet.

Several functions in the program are missing or only partially implemented. The places where you have to modify or add stuff are marked using calls to the dummy function `tODO`. In the final program, you should have replaced all the occurrences of `tODO` with meaningful code.

Overview

Mastermind is a game for two players, called the codemaker and the codebreaker. The codemaker's role is played by the computer in our case. The codemaker devises a code made up of four positions, each position being one of six colors (represented by the numbers 1 to 6). The codebreaker (played by the human user of your program) must guess the code in as few turns as possible. After each guess, two scores are determined for the guess. The *black* score says how many positions of your code match the solution. Once the black score is 4, the codebreaker has determined the correct code and the game is over. The *white* score indicates that a position of the codebreaker's code used a color (number) contained in the solution, but in the wrong position. It is easy to see that the sum of black and white score never exceeds 4, the number of positions in the code.

Here are two protocols of possible games:

code	guess	score
3 4 6 6	1 1 2 2	0 black, 0 white
3 4 6 6	3 3 4 4	1 black, 1 white
3 4 6 6	3 5 3 6	2 black, 0 white
3 4 6 6	3 4 6 6	4 black, 0 white

code	guess	score
5 1 1 4	1 2 3 4	1 black, 1 white
5 1 1 4	1 3 5 6	0 black, 2 white
5 1 1 4	5 2 1 5	2 black, 0 white
5 1 1 4	5 2 4 1	1 black, 2 white
5 1 1 4	5 4 1 1	2 black, 2 white
5 1 1 4	5 1 1 4	4 black, 0 white

And here is how the game looks being played using a Haskell program:

```
? 2 2 3 3
0 black, 1 white
? 4 4 5 5
0 black, 1 white
? 5 2 1 6
1 black, 3 white
? 5 1 6 2
4 black, 0 white
Congratulations.
```

The game interactively prompts the user to type in a guess using a `?` symbol. The user types in a whitespace-separated sequence of four numbers between 1 and 6, and the game responds with the score. If a black score of 4 is reached, the game stops – otherwise, it asks for another guess.

A bottom-up approach

To solve the problem of implementing the game, we will implement several functions in an incremental fashion. If all functions are implemented correctly, you should be able to play the game yourself.

1. Read the skeleton. Compile the skeleton and run it. See what the program does (and more importantly, what it doesn't do). Find all the positions marked `tODO`. Apart from the definition of `tODO` itself, there are seven such positions.

2 (medium)**.** Let us start with the function `black`. Both a guess and a solution are represented by a list of integers `[Int]`. We introduce abbreviations `Guess` and `Solution` for this type, so that our type signatures can be more descriptive. Currently, the function `black` always returns `0`. Redefine the function so that it computes the `black` score correctly. Test the function on inputs of your choice – for example, on the inputs from the game protocols given above:

```
*Main> black [5,1,1,4] [1,2,3,4]
1
*Main> black [3,4,6,6] [3,5,3,6]
2
```

3 (difficult)**.** Note that computing the white score is much more difficult than computing the black score. Don't feel forced to do the exercises in this order – if you get stuck, try the rest first and come back to this exercise later.

Write function `white` which, given the solution and a guess, computes the white score. Again, test your function on examples of your choice, for instance

```
*Main> white [5,1,1,4] [1,2,3,4]
1
*Main> white [3,4,6,6] [3,5,3,6]
0
```

4. Extend the definition of `check` such that the third component tests if the guess was all-correct and the game could be finished. Again, test your function on inputs of your choice:

```
*Main> check [5,1,6,2] [5,2,1,6]
(1,3,False)
*Main> check [5,1,6,2] [5,1,6,2]
(4,0,True)
```

5. The function `report` takes the result of `check` and assembles a piece of text that can be presented to the user. It should indicate the score the user has achived, and also give a message in the case the game was won. The output doesn't have to match the examples here exactly, but should be approximately as follows:

```
*Main> report (check [5,1,6,2] [5,2,1,6])
"1 black, 3 white"
*Main> report (check [5,1,6,2] [5,1,6,2])
"4 black, 0 white\nCongratulations."
```

Note that you can use the function `putStrLn` to print a string on the screen, thereby interpreting escape sequences such as the newline `\n`:

```
*Main> putStrLn it
4 black, 0 white
Congratulations.
```

6 (medium)**.** The function `input` has type `IO Guess` – it should read a whitespace-separated sequence of numbers from the screen, turn it into a list of integers, and return that list as a guess. Currently, it reads a line into the string `l`, but doesn't transform the string, and instead returns the empty list. Fix this problem by using the prelude function `words` (check what it does again) and the given function `readInt` (read the documentation and test it in the interpreter to see how it works). Finally, test your function:

```
*Main> input
? 3 4 5 5
```

```
[3,4,5,5]
*Main> input
? 7 hello
[7,-1]
```

Note that the function currently does not check its input to be valid. The user can place non-numeric inputs, enter colors that are outside the range from 1 to 6, or specify less or more than four colors.

7 (medium)**.** Now we can assemble all our work in the function `loop`. The function should call `check` on the input, use `report` to generate output for the player, and it should repeat the loop unless the given guess was correct. Once you've implemented that function, the game is playable. The `main` function produces a random code, and then calls `loop` with that solution. The function `generateSolution` that generates the random code is given in the skeleton, so you can test the function `loop` by running `main` from the interpreter or by compiling your program (see in the beginning) into an executable and running that.

Bonus exercises

8 (bonus)**.** Implement the function `valid` that, given a guess as produced by `input`, decides if the guess is valid according to the rules of the game. I.e., the guess should contain `width` numbers, and the numbers should be between `1` and `colors` – we store the game parameters in constants so that they're easier to parameterize over at a later point in the development. Then change `input` such that it calls `valid` on the guess before returning it. If the user input isn't valid, it should complain and ask for new input rather than returning the invalid guess.

9 (medium, bonus)**.** Improve the game experience: Count the number of turns the player requires and print it in the end. Allow the user to give up, and in that case, print what the correct solution would have been.

10 (difficult, bonus)**.** Try to implement an algorithm for playing the game. Two such algorithms are given on the Wikipedia page. Modify the game so that the computer plays against itself, and prints the protocol.