Haskell Programming

Practice Programming Test

Mastermind

Note: The seven parts carry 2, 3, 4, 4, 3, 6 and 3 marks respectively. An additional 5 marks will be awarded for the appropriate use of Haskell's higher-order functions and list comprehensions.

IMPORTANT: If your script fails to compile THREE marks will be deducted from your total. If you cannot get a function to compile by the end of the test you should comment it out prior to submission. All commented-out code will be marked.

This question concerns the game of Mastermind in which a person tries to work out a secret code represented as a collection of coloured pegs. The exercise requires you to build some of the supporting functions that would be required in a full Haskell implementation of the game.

> **The Game** is generally played with two people. The first sets a secret (hidden) code using a specified number of coloured pegs. In this version of the game there are three colours: red (R), blue (B) and green (G) and the secret code comprises an arbitrary number, n say, of these. Duplicates are allowed so, for n=3 for example, valid codes include RRG, RBG, BBB and so on. The second player then tries to guess the code and each guess is marked using *black* and *white* scoring sticks. If they guess a colour which is correct and in the correct position they score one black stick; if they guess a colour which is correct but in the wrong place they score one white stick. Black sticks are scored first and the corresponding pegs in both the secret code and the guess are eliminated before scoring the white sticks. For example, if the code is RGR and the guess is RBB the score is 1 black and 0 white, written (1,0). A guess of RBG would score (1,1), RRR (2,0) and so on.

The colours can be represented as an enumerated data type in Haskell thus:

```
data Colour = R | B | G deriving (Eq,Show)
```

The secret code and each guess can both be represented as a list of colours and the result of a given guess as a pair comprising the guess and its corresponding score, a score being the count of the number of black and white sticks respectively. Thus:

```
type Colours = [Colour]

type Score = (Int, Int)

type Result = (Colours, Score)
```

a Define a function `colour :: Int -> Colour` that will convert a given `Int` to a `Colour`, assuming the mapping 0 ⟶ red (R), 1 ⟶ blue (B) and 2 ⟶ green (G). You may assume the precondition that the given integer is 0, 1 or 2.

b Define a function `base3 :: Int -> Int -> [Int]` which given two numbers `x` and `n` will convert `x` to its base 3 representation (this is like binary except each digit may assume the value 0, 1 or 2). For example, given the integer `32` the function should return the list `[1,0,1,2]` i.e. $1\times3^3+0\times3^2+1\times3^1+2\times3^0$. The final list should contain exactly `n` digits so, for example, `base3 0 4` should deliver `[0,0,0,0]`. A precondition is that `n` is at least as large as the number of digits required to represent `x`.

c Define a function `blacks :: Colours -> Colours -> Int`, which given a guess and a secret code will determine the number of black sticks the guess should score. For example, given the Haskell lists `[R,B,G]` and `[R,R,G]` representing the guess and the secret respectively, the result should be `2`. A precondition is that the two lists are of the same length.

d If `s` is the secret code and `g` is a guess then the number of pegs in `g` that score neither a black stick nor a white stick can be computed using Haskell's list difference operator,

viz. `g \\ s`. (The `\\` operator is defined in the List module, but this has been imported for you via the `import` statement in the given template.) From this you can work out the combined total number of black and white sticks in the score and hence, using the function `blacks` from part c, the number of white sticks. Using this define a function
`score :: Colours -> Colours -> Result` which given a guess and a secret code will return the correct `Result` for the guess. For example,
`score [G,G,B] [B,G,G]` should return `([G,G,B],(1,2))`, i.e. one black and two white sticks.

e Using the functions `base3` and `colour` define a function
`allGuesses :: Int -> [Colours]` which given an integer `n` will generate all possible guesses of length `n` using the three colours R, B and G (including duplicates). For example, `allGuesses 2` should generate, in some order, the list
`[[R,R],[R,B],[R,G],[B,R],[B,B],[B,G],[G,R],[G,B],[G,G]]`.
Hint: each guess will be a colour encoding of the base-3 representation of an integer between 0 and 3^n-1 inclusive.

f Suppose we (ultimately) want the computer to guess the secret code. A simple way to construct a next guess is to try all possible guesses (part e) and strike out those that are inconsistent with the given list of scores. It works like this: for each guess, g, assume that it's correct, i.e. that it's the same as the secret code. Then re-score each of the previous guesses using g as the secret code. If the scores produced are the same as those previously recorded then g is *consistent* with those scores and could thus be the right answer. Thus, define a function `consistent :: Colours -> [Result] -> Bool` which given a guess, and the list of results from all previous guesses, returns `True` if the guess would have produced the given list of results; `False` otherwise. A precondition is that the list of results is correct with respect to the secret code.

g Using the function `consistent` from part f define a function
`strike :: [Colours] -> [Result] -> [Colours]` to implement the striking-out described in part f. Again, a precondition is that each (previous) guess has been correctly scored. For example, if the secret code is `[R,G,R]` and the guesses `[R,B,B]` and `[R,G,G]` have been scored correctly (`(1,0)` and `(2,0)` respectively) then the expression `strike (allGuesses 3) ([R,B,B],(1,0)), ([R,G,G],(2,0))]` should yield the list `[[R,R,G],[R,G,R]]`, i.e. the only two codes that can possibly be correct given the list of `Results`.