Mastermind: Final Report

Amy Zal - Programming Languages Final Project

Table of Contents

(You can also click on a section name for quick access to it.)

Introduction

Mastermind is a game I used to play as a kid. The game goes as follows: There is a codemaster (played by the computer) and a guesser (the user). The code master makes a “code” out of 4 colored pegs and hides it from the guesser. Then the guesser guesses the code. The master then grades the guesser by telling them how many of the colors are correct and how many of the pegs are placed in the correct spot. The guesser guesses again, and the game continues on. Simply put, it has the same logic to Wordle, just with colored pegs instead of words.

Visual Example

This is what Mastermind looks like from the code master’s perspective. On the right side you can see the evaluation of the guesses. Red pegs represent completely correct guesses and white pegs represent partially correct guesses (the color is correct but not located in the correct place.)

How to Run

1. Load the file and run
   a. Run *ghci* in the terminal to open GHCI
   b. Once in ghci, type *:l mastermind.hs* to load the file
   c. Type *main* to start the program

2. The program will prompt you for inputs.

a. **Guesses**: Guesses should be a four letter input, with each letter matching one of the acceptable colors. Inputs can be uppercase or lowercase.
b. **Error handling:** The program should handle both invalid characters and guesses of incorrect length.
c. **Testing/Cheats**: During the program, if users at any point enter *cheat* then the secret code will be outputted to the user. This is only to be used for testing purposes.
d. **Ending the program:** During the program, if users at any point enter *quit* then the program will quit, outputting the previous guesses and the Secret code.

Project Outline

Imports

The following imports are used:

import System.Random (randomRIO)

- Useful for generating and using random values
- **In my code:** generating the secret code and random colors

import Control.Monad (replicateM)

- Control.Monad is useful for functions relating to Monads
- **In my code:** replicateM generates a list of random colors

Datatypes and Parsing

The datatypes will be Color and Code. Color will represent all the potential color options and Code will represent the codes for the answer and the guess. Parsing will take the guess and split each letter from the code into its corresponding Color data type.

Colors

data Color = Red | Green | Blue | Yellow | Orange | Purple deriving (Show, Eq, Enum)

Codes

```
type Code = [Color]

codeLength :: Int
codeLength = 4
```

Guess Results

```
data GuessResult = GuessResult
  { guess :: Code
  , correctPosition :: Int
  , correctColor :: Int
  }
```

(used to store guess results)

```
instance Bounded Color where
        minBound = Red
        maxBound = Purple
```

(The Bounded type class is used for any type that has a min and max value, it is useful for my code to easily get random values of any color)

In the method getRandomColor it is used as follows:

```
index <- randomRIO (fromEnum (minBound :: Color), fromEnum (maxBound :: Color))
```

Function Implementation

Here is a list of the implemented functions with their types and a brief explanation.

1. **generateSecretCode:: IO Code**

- generates a random secret code for the answer
- Explanation: uses *replicateM* to create a list of random colors of *codeLength* length.

2. **getRandomColor :: IO Color**
    - Helper function for generateSecretCode, generates a random color for each individual peg
    - Explanation: uses randomRIO to get a random color within the bounds
3. getUserGuess :: Code ⟶ IO Code:
    - gets the user guess/input
4. colorFromCode :: Char ⟶ Maybe Color
    - parses the guess, if the input is invalid it returns nothing
5. **evaluateGuess :: Code ⟶ Code → (Int, Int)**
    - tells the player how they did
    - Explanation: returns a tuple of two Ints: correctPosition and correctColor
        - The functions uses a zip to pair those, filters them to see if they overlap and counts the total correct colors (in position or not)
6. countColor:: Code → Color → Int
    - helper function- counts the occurrences of a color in the code
7. displayGuesses :: GuessResult → IO ()
    - Function to display a single guess w/ its result
8. pastGuesses :: [GuessResult] → IO ()
    - Function to display past guesses w/ results
9. intro :: IO ()
    - Function to display an introduction w/ color options and rules (displayed in full below)
10. **play :: Code → [GuessResult] → IO ()**
    - organizes everything, main game loop (displayed in full below)
    - Explanation: takes as input the secret code and a list of the past guesses and plays accordingly, keeping record of past guesses
11. Main:: IO ()

- handles input and output (displayed in full below)

Main Function(s)

Main

```haskell
main :: IO ()
main = do
  intro
  secretCode <- generateSecretCode
  play secretCode []
```

Intro

```haskell
intro :: IO ()
intro = do
  putStrLn " "
  putStrLn "Welcome to Mastermind!"
  putStrLn " "
  putStrLn "Color Options: R (Red), G (Green), B (Blue), Y (Yellow), O (Orange), P (Purple)"
  putStrLn " "
  putStrLn "Rules: Try to guess the secret code, consisting of four colors chosen from the options above."
  putStrLn " "
  putStrLn "For each guess, you will receive feedback on the folllowing:"
  putStrLn "  -# of completely correct colors (colors that are correctly in their space)"
  putStrLn "  -# of colors that are partially corect (colors that are in the secret code but not in the correct space)"
  putStrLn " "
  putStrLn "Type 'quit' to end the game at any time."
  putStrLn " "
```

Play

```haskell
play :: Code -> [GuessResult] -> IO ()
play secretCode pastGuessesList = do
  guess <- getUserGuess secretCode
  if null guess
```

```
    then do
      pastGuesses pastGuessesList
      putStrLn $ "Secret Code: " ++ show secretCode
      putStrLn " "
      putStrLn "Quitting the game. Goodbye!"
    else do
      let (correctPosition, correctColor) = evaluateGuess secretCode guess
      let newGuessResult = GuessResult guess correctPosition correctColor
      pastGuesses (pastGuessesList ++ [newGuessResult])
      if correctPosition == codeLength
        then putStrLn "Congratulations! You've guessed the secret code."
        else play secretCode (pastGuessesList ++ [newGuessResult])
```

Discussion

Difficult Aspects and Solutions

1. Keeping track of past guesses- This is where I created the GuessResult data type and kept track of it by passing the pastGuessList in during each iteration of play.
2. Bounded Colors - I didn't know what the easiest way to make the datatype easily boundable, and discovered the Bounded type class. This made it easy to create a min and max value when using randomRIO
3. Figuring out importing randoms- Luckily I didn't have any big issue with this, but I did have to try 2 different methods. (Check the helpful resources section for the link I used to set mine up.)

Proud Moments

I didn't procrastinate on this project!! I actually put in a decent amount of time and effort from the beginning so I didn't have to scramble to fix everything at the last minute.

Conclusion

Summary

I found it interesting to implement something from my childhood. I think delving deeper into IO was interesting to me, since we didn't spend as much time on that in class as other subjects.

In Retrospective / Future Implementations

If I were to continue this project, I would like to create a GUI instead of relying on terminal IO. I think it would be a fun challenge, but I had a lot going on so I didn't have the time to tackle that beast. Even though it would have been nice, I'm still proud of what I did since I did accomplish everything I promised through the proposal.

Helpful Sources

Here are some of the places I got help from. If someone in the future were to implement this project these sources would be helpful.

The Haskell Cabal | Overview (basic info on cabal and installing it)

How to install modules (discussion forum on specifically installing System.Random)

Hoogle (haskell.org) (search data types, methods, packages, etc)