PoP 8g - Mastermind

Carl Dybdahl, Emil Chr. Søderblom, Patrick Hartvigsen

December 12, 2016

1 Preface

We have been tasked with writing a Mastermind program in F♯, and have been given a series of requirements. The requirements include various items, including which types the program should use, that it should have the option of both AI and human players, and that it should be properly documented and tested. We have made sure to fulfill all of the requirements, and in addition have implemented various optional things, including a relatively good AI.

2 Problem Analysis

We had been given the following requirements:

- The program must be an implementation of the game Mastermind.
- Both user vs user, AI vs AI, and user vs AI must be available. In the last case, the user must be allowed to pick whether they have to choose the code or guess the code.
- The program must implement the folllowing structure:

```
1  type codeColor =
2    Red | Green | Yellow | Purple | White | Black
3  type code = codeColor list
4  type answer = int * int
5  type board = (code * answer) list
6  type player = Human | Computer
7  // let the given player type choose a task for their opponent
8  makeCode : player -> code
9  // let the given player type make a guess based on game history
10 guess : player -> board -> code
11 // compute the answer given a guess and the correct code
12 validate : code -> code -> answer
```

- It must be possible to play this program in the console, rather than with a GUI.
- The program must be tested and documented according to the F♯ standard.
- The solution must be documented with a detailed report at most 20 pages long written in LaTeX.

3 Architecture and Design

We have decided to make this program as functional (in the sense of the programming paradigm) as possible. We can't make it entirely functional, as we have to do IO, but we have avoided loops with mutable variables and have separated the IO from the business logic in various cases.

We could've written some imperative portions, but this has the disadvantage that F♯ is not a primarily imperative language and therefore lack constructs such as `break`. For example, the main loop of the program is the following tail recursive function:

8g.fsx - mainLoop

This is written as a tail recursive function. If we wanted to write it iteratively, we'd need a `break` construct or an additional local variable to end the loop when the guess is correct.

A lot of our architecture has been decided in the requirements, that describe various functions we should implement and types we should use. What remains after that is essentially how we wire up the code to form a program.

Initially in the code file, we have a binding that contains a list of all `code`s, named `makeCodes`. Next, we have some functions which handle user input, namely `charListToCode`, `checkStringCode`, `getUserInput`, `playerFormat` and `codeFormat`. The rest of the file contains the business logic:

The function `makeCode`, which is right after the previously mentioned bindings, is one of the functions we were required to implement. It handles generating a `code` that must be guessed. Then comes `validate`, another function mentioned in the requirements, which computes the answer that the player sees when they have made a guess. Next, we have three functions `guessQuality`, `guessBot` and `guess` which together implement the logic for letting the user or AI guess the code.

Last, we have `gameInit`, `gameStart` and `main`, which tie the functions together into a full game.

4 Program and Algorithms

4.1 Codes

The total number of different possible secret codes is $6^4 = 1296$, which is sufficiently small that we can keep a list of all the possible codes and easily iterate through it. We exploit this in a number of ways. For example, to generate a random code, we simply pick a random index in our table of codes. It is an important observation that the number of codes is tractably small, because the AI algorithm uses this heavily.

4.2 User Input

In order to read user input, we have defined a function `getUserInput` which separates the reading loop from the validation of user input. It takes a parameter `format : string -> 'a option` which it uses to determine whether a given line of input is valid: if the function returns `Some x`, the input is valid, otherwise `getUserInput` queries the user again.

The `format` parameter combines validation and parsing: it can return a value of an arbitrary type, which is the value that `getUserInput` will return.

4.3 AI Algorithm

The first step in the AI is to figure out what codes are potentially the correct ones, based on the current state of the board. It does so by taking all possible codes and filtering so that it gets a list of only the codes that match the answers we have seen so far.

From this list, it tries to pick the code that, once it sees the answer for the code, gives as much information about the correct code as possible.

To do this, we need to define how much information a guess yields. Let S be the set of code that, given the current state of the board, could potentially be the secret code. Observe that a guess g partitions the set S into set of codes that cannot be distinguished based on the answers that the guess could potentially yield. That is, two codes $a, b \in S$ are in the same petition if the answer that you get when you guess g is the same regardless of whether a or b is the true secret code.

Let $P_g(S)$ be the set of partitions that g yields. A guess that yields optimal information would make all the partitions the same size. However, this is not always possible. For this reason, we need to define a qualitative measure of how much information a guess yields. We have decided that the information $I(g)$ is defined by:

$$I(g) = \sum_{s \in P_g(S)} \log(\varepsilon + |s|)$$

for some ε. (We used $\varepsilon = 1.0$, but according to our tests the exact value made very little difference.) The reasoning behind this is that maximizing the sum of a sublinear function of the set sizes will encourage the AI to make the sets approximately equally big. The epsilon is used to avoid having infinities break things when the size of the partitions is zero.

4.4 Validation Algorithm

The validation algorithm must, given two codes `c` and `g`, find the number of white and black pins when one guesses `g` while `c` is the correct code.

The number of white pins is defined to be the number of correct colors in the code that are not placed in the correct spot, whereas the number of black pins is defined to be the number of correct colors in the code that are placed in the correct spot.

Computing the number of black pins is trivial: we simply count the number of spots where the two colors match.

It is more complex to compute the number of white pins. To do this, we actually compute the sum of the number of black and the number of white pins, and then subtract the number of black pins.

The sum of the numbers of pins is simply the number of correct colors, regardless of whether they are in the correct spot or not. We find this number by considering each color and counting its number of occurences in `c` and `g`. Each occurence in `c` that is matched by an occurence in `g` is a correct color, so we simply take the minimum of the two numbers of occurences. By summing over these minimums, we find the total number of pins.

5 User Guide

When you start the program, you will be prompted for what kind of game you want to play. Write `1` if you wish to guess the code made by the computer, `2` if you wish to play against another human, `3` if you wish to have the computer try to guess your code and `4` if you wish to test the computer against itself.

If you chose mode `2` or `3`, you will be queried for the code that must be guessed. The code must be 4 letters long and consists of only the characters `rgypwb`, standing for red, green, yellow, purple, white and black. Once you've entered the code, the guessing will begin.

In mode `1` and `2`, you will be repeatedly queried for input guesses. At each query, you will be shown the board of previous guesses. This is rendered as a list of codes and answers, where each answer is the number of white and black pins respectively.

You have 30 turns to guess the hidden code. If you don't manage to guess it in this time, you lose the game.

6 Testing

We have both tested the main functions `makeCode`, `guess` and `validate` separately and tested the full program. These tests can be found in the `8gTest.fsx`. In order to test the user input, we made a test hook named `readUserLine` which we could override to programmatically control the input in our tests.

The tests for `makeCode`, `guess` and `validate` were all successful. In the part of `makeCode` and `guess` where the player type was `Human`, the function converted the input string from the user to a parsed value of type `code` correctly. When player type was `Computer`, it returned a `code` correctly. `makeCode` returned a random code and `guess` returned a guess on a possible code.

When the program asks for user input the code checks if the input is valid. We tried to giving it wrong inputs to see if it would handle them correctly and it did.

The test of the full program were done by calculating the average number of turns it took `Computer` to guess a code. We also tested for worst and best case number of turns. We expected `Computer` to do about as well as other algorithms, such as Donald Knuths algorithm that uses 4.340 guesses on average.

We tried varying the epsilon defined in section 4.3. As can be seen on the following table, this had little effect:

AI	average number of guesses
$\varepsilon = 0.01$	4.458
$\varepsilon = 0.1$	4.457
$\varepsilon = 1.0$	4.455
$\varepsilon = 2.0$	4.455
$\varepsilon = 5.0$	4.456
$\varepsilon = 10.0$	4.458
no AI	5.021

In this table we also included a test without any AI, where the code simply picks the first code that is compatible with all previous guesses. While ε had little effect, we chose $\varepsilon = 1.0$ because it did slightly better than the alternatives. This lets it guess the correct code in 4.455 guesses on average, which is reasonably close to 4.340.

7 Conclusion

We have written a Mastermind game in F♯ using various algorithms that we have documented in this report. The game works as we want it to, and our AI can on average guess a secret code in 4.455 guesses.

8 Appendix

8.1 Code

8g.fsx

```
1  //#############
2  //Opgave 8g - Mastermind
3  //#############
4
5  //Program types
6  type codeColor =
7      Red | Green | Yellow | Purple | White | Black
8  type code = codeColor list
9  type answer = int * int
10 type board = ( code * answer ) list
11 type player = Human | Computer
12
13 /// <summary>
14 ///  Produce a list of all possible combination of code(code have 4 elements).
15 /// </summary>
16 /// <example>
17 ///   <code>
18 ///     let s = makeCodes
19 ///   </code>
20 /// </example>
21 /// <returns>code with length of 4.</returns>
22 let makeCodes =
23     let colors = [Black; White; Purple; Yellow; Green; Red]
24     let rec codesOfLength = function
25         | 0 -> [[]]
26         | n ->
27             let subCodes = codesOfLength (n - 1)
28             colors |> List.collect (fun col ->
29                 subCodes  |> List.map ((fun x xs -> x :: xs) col))
30     codesOfLength 4
31
32 /// <summary>
33 ///  Converts a char-list to code. (with chars 'r','g,'y','p','w,'b')
34 /// </summary>
35 /// <remarks> All other chars than r','g,'y','p' and 'w' will give Black
36 /// </remarks>
37 /// <example>
38 ///   <code>
39 ///     charListToCode (List.ofSeq "rrrr")
40 ///   </code>
41 /// </example>
42 /// <param name="code"> A list of chars.</param>
43 /// <returns> Returns code, the example above til return [Red;Red;Red;Red].
44 /// </returns>
45 let rec charListToCode (code : char list) : code =
46     match code with
47     | [] -> []
48     | x :: xs when x = 'r' -> Red :: charListToCode xs
```

```
49      | x :: xs when x = 'g' -> Green :: charListToCode xs
50      | x :: xs when x = 'y' -> Yellow :: charListToCode xs
51      | x :: xs when x = 'p' -> Purple :: charListToCode xs
52      | x :: xs when x = 'w' -> White :: charListToCode xs
53      | x :: xs (*   'b'  *) -> Black :: charListToCode xs
54
55  /// <summary>
56  ///  Checks if a string is the length of 4
57  ///  and only contaions chars:'r','g,'y','p','w,'b'.
58  /// </summary>
59  /// <example>
60  ///   <code>
61  ///    checkStringCode "rrrr"
62  ///   </code>
63  /// </example>
64  /// <param name="code"> A string. </param>
65  /// <returns>
66  ///  A boolean, if string is valid, it returns true else false.
67  ///  The example above wil return true.
68  /// </returns>
69  let checkStringCode (code : string) : bool =
70      let checkChars (str : string) =
71          str |> String.forall (fun ch -> "rgypwb" |> String.exists ((=) ch))
72      match code.Length with
73      | 4 -> checkChars code
74      | _ -> false
75
76  // This variable is used as a hook, for testing funktions with user inputs in 8gTests.fsx.
77  let mutable readUserLine = fun () -> System.Console.ReadLine ()
78
79  /// <summary>
80  ///  Gets input from user as a string. And checks if string is valid
81  ///  with a format function. If string is valid then return the string
82  ///  else gets new input from user and checks if valid.
83  /// </summary>
84  /// <example>
85  ///   <code>
86  ///    getUserInput playerFormat
87  ///   </code>
88  /// </example>
89  /// <param name="format"> A format function for what is allowed to pass.</param>
90  /// <returns> Returns option type with string. </returns>
91  let rec getUserInput(format : string -> 'a option) =
92      printf "> "
93      match format <| readUserLine () with
94      | None ->
95          printfn "Invalid input."
96          getUserInput format
97      | Some x -> x
98
99  /// <summary>
100 ///  Format for checking input when asking for who is playing.
101 ///  Checks if input is "1","2","3" or "4".
102 /// </summary>
```

```
103 /// <example>
104 ///   <code>
105 ///     getUserInput playerFormat
106 ///   </code>
107 /// </example>
108 /// <param name=""> A string.</param>
109 /// <returns>
110 ///  Returns option type, if string not valid than it returns None
111 ///  else it returns players.
112 /// </returns>
113 let playerFormat = function
114     | "1" -> Some (Human, Computer)
115     | "2" -> Some (Human, Human)
116     | "3" -> Some (Computer, Human)
117     | "4" -> Some (Computer, Computer)
118     | _   -> None
119
120 /// <summary>
121 ///  Format for checking input when asking for a for a string-code.
122 ///  checks if input is allowed with checkStringCode.
123 /// </summary>
124 /// <example>
125 ///   <code>
126 ///     getUserInput codeFormat
127 ///   </code>
128 /// </example>
129 /// <param name=""> A string.</param>
130 /// <returns>
131 ///  Returns option type, if string not valid than it returns None
132 ///  else it returns code.
133 /// </returns>
134 let codeFormat = function
135     | x when x |> checkStringCode -> Some (List.ofSeq x |> charListToCode)
136     | _ -> None
137
138 /// <summary>
139 ///  makes a code of length 4. If player is Human then it ask user for input.
140 ///  else if player is Computer produce random code.
141 /// </summary>
142 /// <example>
143 ///   <code>
144 ///     makeCode Computer
145 ///   </code>
146 /// </example>
147 /// <param name="p"> A player type.</param>
148 /// <returns> Returns code </returns>
149 let makeCode (p : player) : code =
150     match p with
151     | Human ->
152         printfn "Choose color code with length of 4 with: \
153         r=Red, g=Green, y=Yellow, p=Purple, w=White, b=Black."
154         getUserInput codeFormat
155     | Computer ->
156         let r = System.Random ()
```

```
157         makeCodes.[(r.Next (0,1297))]
158 
159 /// <summary>
160 ///  Validate your guess with the correct code.
161 /// </summary>
162 /// <example>
163 ///   <code>
164 ///     validate [Red;Red;Red;Red] [Red;Red;Red;Green]
165 ///   </code>
166 /// </example>
167 /// <param name="c"> The correct code.</param>
168 /// <param name="g"> Guess from player.</param>
169 /// <returns>
170 ///  Returns a tuple of white and black pins.
171 ///  White corrrect color, black correct color and position.
172 /// </returns>
173 let validate (c : code) (g : code) : answer =
174     let codeColourLst = [Red; Green; Yellow; Purple; White; Black]
175     let white = List.sum (List.map (fun col ->
176         let countG = List.length (List.filter ((=) col) g)
177         let countC = List.length (List.filter ((=) col) c)
178         min countG countC
179         ) codeColourLst)
180     let black = List.sum (List.map (fun (colC,colG) ->
181         if colC = colG then 1 else 0) (List.zip c g))
182     (white-black,black)
183 
184 /// <summary>
185 ///  Determines the best guess out of possible guesses.
186 /// </summary>
187 /// <example>
188 ///   <code>
189 ///     s |> List.maxBy (guessQuality s)
190 ///   </code>
191 /// </example>
192 /// <param name="option"> Possible code guesses. </param>
193 /// <param name="guess">  Checks quality of guess. </param>
194 /// <returns>
195 ///  Returns a float, a higher number means a better guess.
196 /// </returns>
197 let guessQuality (options : code list) (guess : code) : float =
198     let answers: answer list = options |> List.map (validate guess)
199     let allAnswers: answer list =
200         [0 .. 4] |> List.collect (fun w ->
201             [0 .. 4 - w] |> List.map (fun b ->
202                 (w, b)))
203     let initCounts = allAnswers |> List.map (fun x -> (x, 0)) |> Map.ofList
204     let counts = answers |> (List.fold (fun q a ->
205         let count = q |> Map.find a |> (+) 1
206         Map.add a count q
207     ) initCounts)
208     let eps = 1.0
209     counts |> Map.toList |> List.map(snd >> float >> (+) eps >> log) |> List.sum
210 
```

```
211 /// <summary>
212 ///  Reduce possible guesses with use of previously
213 ///  guesses and answers, and returns a guess.
214 /// </summary>
215 /// <example>
216 ///   <code>
217 ///     guessBot [((0,3),[Red;Red;Red;Red]);((0,3),[Red;Red;Red;Green])]
218 ///   </code>
219 /// </example>
220 /// <param name="b1"> The board, previously guesse and answers.</param>
221 /// <returns> Returns a guess as type code. </returns>
222 let guessBot (b1 : board) : code =
223     match b1 with
224     | [] -> [Red;Red;Green;Green]
225     | _  ->
226         let s = List.foldBack (fun (c, (w, b)) ->
227             List.filter (fun x -> (validate x c) = (w,b))) b1 makeCodes
228         s |> List.maxBy (guessQuality s)
229
230 /// <summary>
231 ///  Returns a guess of code. If player is Human gets input from user.
232 /// If player is Computer it will 'calculate' a guess of code.
233 /// </summary>
234 /// <example>
235 ///   <code>
236 ///     guess Human [((0,3),[Red;Red;Red;Red]);((0,3),[Red;Red;Red;Green])]
237 ///   </code>
238 /// </example>
239 /// <param name="p"> Player type.</param>
240 /// <param name="b"> The board, previously guesse and answers.</param>
241 /// <returns> Returns a guess as type code. </returns>
242 let guess (p : player) (b : board) : code =
243     match p with
244     | Human ->
245         printfn "Previous guesses: %A" b
246         printfn "Try a guess on the color code: \
247         r=Red, g=Green, y=Yellow, p=Purple, w=White, b=Black."
248         getUserInput codeFormat
249     | Computer -> guessBot b
250
251 /// <summary>
252 ///  Initialize game variabels, player types and the correct-code.
253 /// </summary>
254 /// <example>
255 ///   <code>
256 ///     gameInit()
257 ///   </code>
258 /// </example>
259 /// <returns> Returns player types and code </returns>
260 let gameInit() =
261     printfn "Welcome to Mastermind SUPER TEXT 0.3x Supreme digital edition."
262     printfn "The guesser have 30 tries to guess a color-code made by the coder."
263     printfn "Choose how to play (guesser vs coder):
264     1: Human vs Computer
```

```
    2: Human vs Human
    3: Computer vs Human
    4: Computer vs Computer"
    let p1, p2 = getUserInput playerFormat
    (p1,p2, makeCode p2)

/// <summary>
///  Starts the game, and the game loop.
/// </summary>
/// <example>
///   <code>
///     gameStart()
///   </code>
/// </example>
/// <returns> Returns unit (prints information out). </returns>
let gameStart() =
    let (p1,p2,colorCode) = gameInit()
    let maxTurns = 30
    printfn "\nGame starts!\n"
    let rec gameLoop (B : board) (turns : int) =
        let G = guess p1 B
        let A = validate colorCode G
        match A with
        | (0,4) -> printfn "\nYou won in %d turns! :D" turns
        | _ when turns <= 30 ->
            printfn "\nNext turn"
            gameLoop ((G,A) :: B) (turns + 1)
        | _ -> printfn "\nYou have used your 30 turns, you lose"
    gameLoop [] 0

[<EntryPoint>]
let main args =
    gameStart()
    0
```

8.2 Test Code

8gTests.fsx

```
//#####################################################
// Test for makeCode, guess and validate.
//#####################################################

// Testing if makeCode() returns correct type-code from string and computer produce a random code.
let testMakeCode() =
    let test (input : string) (expected : code) : unit =
        readUserLine <- fun () -> input
        printfn "User Writes: %A as input" input
        let result = makeCode Human
        if result = expected then
            printfn "[ OK ] makeCode Human %A = %A" input expected
        else
            printfn "[FAIL] makeCode Human %A = %A != %A" input result expected
    test "rbgp" [Red;Black;Green;Purple]
    test "yyww" [Yellow;Yellow;White;White]
```

```
17      printfn ""
18      printfn "makeCode Computer, makes Random output(code): %A" (makeCode Computer)
19      printfn "makeCode Computer, makes Random output(code): %A" (makeCode Computer)
20
21  // Testing if guess() returns correct type-code from string and computer returns a code.
22  let testGuess() =
23      let test (input : string) (input2 : board) (expected : code) : unit =
24          readUserLine <- fun () -> input
25          printfn "User Writes: %A as input" input
26          let result = guess Human input2
27          if result = expected then
28              printfn "[ OK ] makeCode %A = %A" input expected
29          else
30              printfn "[FAIL] makeCode %A = %A != %A" input result expected
31      test "rbgp" [] [Red;Black;Green;Purple]
32      test "yyww" [([Red;Red;Red;Red],(1,1))] [Yellow;Yellow;White;White]
33      printfn ""
34      printfn "guess Computer, gives a possible guess: %A" (guess Computer [])
35      printfn "guess Computer, gives a possible guess: %A"
36          (guess Computer [([Red;Red;Green;Green],(0,0))])
37
38  // Testing if validate returns correct black and white pins.
39  let testValidate() =
40      let test (input : code) (input2 : code) (expected : answer) : unit =
41          let result = validate input input2
42          if result = expected then
43              printfn "[ OK ] validate %A %A = %A" input input2 expected
44          else
45              printfn "[FAIL] validate %A %A = %A != %A"
46                  input input2 result expected
47      test [Red;Red;Red;Red] [Red;Red;Red;Red] (0,4)
48      test [Black;Red;Purple;Green] [Red;Black;Red;Red] (2,0)
49      test [Black;Red;Purple;Green] [Red;Black;Purple;Green] (2,2)
50
51  testMakeCode()
52  printfn "\n"
53  testGuess()
54  printfn "\n"
55  testValidate()
56  printfn "\n"
57
58  //######################################################################
59  //Test for full game. Average turns for Computer to guess the code.
60  //######################################################################
61
62  /// <summary>
63  ///  Calculates the average turns i takes for Computer to guess a code.Cons
64  /// </summary>
65  /// <example>
66  ///   <code>
67  ///     //printfn "%A" average
68  ///   </code>
69  /// </example>
70  /// <returns> Returns float. </returns>
```

```
let average() = makeCodes |> List.averageBy (fun secret ->
    let maxTurns = 30
    let mutable turns = 0
    let mutable res = 0
    let mutable B = []
    printfn "game begin"
    while (maxTurns > turns) do
        turns <- turns + 1
        let G = guess Computer B
        let A = validate secret G
        if A = (0,4) then
            printfn "you win in %A turns! :D" turns
            res <- turns
            turns <- 30
        else
            B <- (G,A) :: B
    float res
)

//printfn "%A" (average())


// Average turns for computer without advance-guess: 5.021
// Average turns for computer with advance-guess where eps is 0.1: 4.457

// Average turns for computer with advance-guess where eps is 1.0: 4.455,
// worst case is 6 turns and best case is 1 turn.

// Average turns for computer with advance-guess where eps is 0.01: 4.458
// Average turns for computer with advance-guess where eps is 10: 4.456
// Average turns for computer with advance-guess where eps is 5.0: 4.456
// Average turns for computer with advance-guess where eps is 2.0: 4.455
```

8.3 Test Results

8gTestResult.txt

```
User Writes: "rbgp" as input
Choose color code with length of 4 with: r=Red, g=Green, y=Yellow, p=Purple, w=White, b=Black.
> [ OK ] makeCode Human "rbgp" = [Red; Black; Green; Purple]
User Writes: "yyww" as input
Choose color code with length of 4 with: r=Red, g=Green, y=Yellow, p=Purple, w=White, b=Black.
> [ OK ] makeCode Human "yyww" = [Yellow; Yellow; White; White]

makeCode Computer, makes Random output(code): [Red; Purple; Red; Yellow]
makeCode Computer, makes Random output(code): [Purple; Yellow; Green; Purple]


User Writes: "rbgp" as input
Previous guesses: []
Try a guess on the color code: r=Red, g=Green, y=Yellow, p=Purple, w=White, b=Black.
> [ OK ] makeCode "rbgp" = [Red; Black; Green; Purple]
User Writes: "yyww" as input
Previous guesses: [([Red; Red; Red; Red], (1, 1))]
Try a guess on the color code: r=Red, g=Green, y=Yellow, p=Purple, w=White, b=Black.
```

```
> [ OK ] makeCode "yyww" = [Yellow; Yellow; White; White]

guess Computer, gives a possible guess: [Red; Red; Green; Green]
guess Computer, gives a possible guess: [Black; Black; White; Purple]


[ OK ] validate [Red; Red; Red; Red] [Red; Red; Red; Red] = (0, 4)
[ OK ] validate [Black; Red; Purple; Green] [Red; Black; Red; Red] = (2, 0)
[ OK ] validate [Black; Red; Purple; Green] [Red; Black; Purple; Green] = (2, 2)
```