

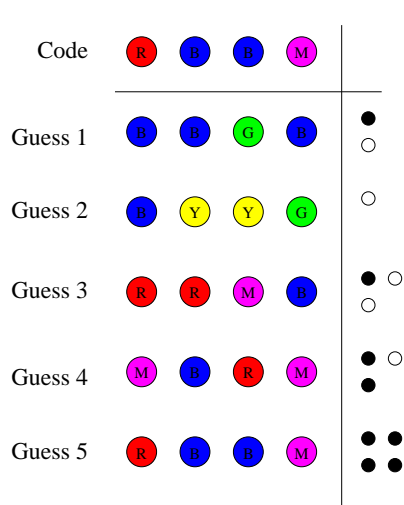
School of Computer Science and Software Engineering
CSE3322 Programming Languages and Implementation

Assignment 2

Due 12 noon Monday 30th August

This assignment covers material from lectures ML1 to ML8. Its purpose is to practice basic and advance ML programming. In order to keep the assignment interesting but relatively short you are allowed (actually, strongly encouraged) to use any function which is part of the Standard ML Basis Library. Particularly useful are those in structures such as List, String, Integer, TextIO, and Random.

Problem Description



The aim of this project is to build a computer-based MastermindTM game. Mastermind is a game for two players invented in 1970-71 by Mordecai Meirowitz. The first player chooses as a code a sequence of four coloured pegs, and places them behind a screen. The second player makes a guess at the code. The first player then scores the code with black and white scoring pegs. For each peg in the guess which is the correct colour in the correct position the first player places a black scoring peg in the scoring box, and for each peg which is the correct colour but in the incorrect position, the first player places a white scoring peg in the scoring box. The second player then makes a new guess taking into account this information, and this continues until the second player correctly guesses the code. The aim is for the second player to guess the code as quickly as possible.

We will play Mastermind with variations in the size of the board (i.e., the number of columns) and the possible colours of pegs. For the game to be *interesting* it must have at least 3 columns and use at least 3 colours. For the game to be *manageable* it must have no more than 6 columns and use no more than 2 more colours than it has columns. A game is *playable* if it is interesting and manageable.

- (a) [0.5 mark] The first task of the Mastermind program is to decide whether the size of the board and the number of colours chosen by the user yields an interesting game. Write an ML function `playable = fn : int -> int -> bool` which given a size of board and a number of colours (in that order) returns `true` if the game is playable and `false` otherwise. For example:

Expression	Answer
<code>playable 4 3</code>	<code>True</code>
<code>playable 5 2</code>	<code>False</code>
<code>playable 3 6</code>	<code>False</code>
<code>playable 6 3</code>	<code>True</code>

- (b) [1 mark] The second task of the Mastermind program is to choose a hidden code for the player to guess. A row of the board is represented by a list of colours, so we assume the following type definitions:

```
type colour = string;
type row = colour list;
```

We will assume the code is calculated from a list of random integers ranging from 1 to the number of colours, the length of the list being the same as the size of the board.

Write an ML function `code = fn : colour list -> int list -> row` which will, given a list of allowed colours and a list of integers with length the size of board and with elements ranging from 1 to the number of allowed colours, return a code, which is a list of colours of length the size of the board. For example

Expression	Answer
<code>code ["red", "blue", "tan"] [3,2,1,2]</code>	<code>["tan", "blue", "red", "blue"]</code>
<code>code ["i", "b", "m", "fnord", "e", "s", "w"] [7,4,2,3]</code>	<code>["w", "fnord", "b", "m"]</code>

- (c) [1 mark] Next the Mastermind program has to check that the guesses entered by the player are valid. A guess is *valid* if it has the right length for the size of the game and each colour in the guess is one of the allowed colours. Define a function `validguess = fn : int -> colour list -> row -> bool` which will, given a size of board, a list of allowed colours, and a guess, return `true` if the guess is valid and `false` otherwise. Example expressions and their answers are:

Expression	Answer
<code>validguess 4 ["red", "blue", "tan"] ["tan", "blue", "red", "blue"]</code>	<code>True</code>
<code>validguess 5 ["red", "blue", "tan"] ["tan", "blue", "red", "blue"]</code>	<code>False</code>
<code>validguess 4 ["red", "blue", "tan"] ["tan", "yellow", "tan", "red"]</code>	<code>False</code>

- (d) [3 marks] The next important task of the Mastermind program is to determine the score for a guess in terms of white and black pegs. Consider the game shown in the previous figure. The code is `[red, blue, blue, mauve]`. The first guess has one blue in the correct position, which gets one black peg. There are two other blues. Since neither is in a correct position the score gets one white peg to indicate one of the pegs other than that indicated by the black is the correct colour but not in the correct position. Notice that the second player can now deduce that the code has at least one blue since either the black or the white score must refer to a blue peg (the other could refer to a green).

Write an ML function `score = fn : row -> row -> (int * int)` which takes a row with the hidden code and a row with the players guess, and returns a pair containing the number of black pegs and the number of white pegs; scoring the players guess against the code. The function can assume the guesses are valid. Example expressions and their answers are:

Expression	Answer
<code>score ["red", "blue", "blue", "mauve"] ["blue", "blue", "green", "blue"]</code>	<code>(1,1)</code>
<code>score ["red", "blue", "blue", "mauve"] ["blue", "yellow", "yellow", "green"]</code>	<code>(0,1)</code>
<code>score ["red", "blue", "blue", "mauve"] ["red", "red", "mauve", "blue"]</code>	<code>(1,2)</code>

- (e) [3 marks] We will now put it all together in a function `play = fn : int -> colour list -> unit` which takes the size of the board and the allowed colours the player wants to use. It will then check the game is playable (print an error if it is not), select a hidden code, and initialise the history of the game to be empty. It then enters a loop which asks the user for a guess (must be entered with colours separated by spaces), reads the guess, checks it is valid (if not prints an error and re-enters the loop), computes its score, adds the guess and its associated score to the past history, and prints the newly computed history. The loop will end when the guess is equal to the hidden code (the score is perfect). For example, consider the following execution:

```
- play 5 ["red", "blue", "green", "tan", "mauve", "purple"];
Please enter a guess:
```

```

red red red red red
Past history (from most recent):
red red red red red -- 1 0
Please enter a guess:
red blue blue blue blue
Past history (from most recent):
red blue blue blue blue -- 1 1
red red red red red -- 1 0
Please enter a guess:
green red blue green green
Past history (from most recent):
green red blue green green -- 0 3
red blue blue blue blue -- 1 1
red red red red red -- 1 0
Please enter a guess:
tan blue green red tan
Past history (from most recent):
tan blue green red tan -- 1 3
green red blue green green -- 0 3
red blue blue blue blue -- 1 1
red red red red red -- 1 0
Please enter a guess:
tan green red blue mauve
Past history (from most recent):
tan green red blue mauve -- 1 4
tan blue green red tan -- 1 3
green red blue green green -- 0 3
red blue blue blue blue -- 1 1
red red red red red -- 1 0
Please enter a guess:
mauve green tan red blue
Past history (from most recent):
mauve green tan red blue -- 1 4
tan green red blue mauve -- 1 4
tan blue green red tan -- 1 3
green red blue green green -- 0 3
red blue blue blue blue -- 1 1
red red red red red -- 1 0
Please enter a guess:
mauve tan green blue red
You won with past history:
mauve tan green blue red -- 5 0
mauve green tan red blue -- 1 4
tan green red blue mauve -- 1 4
tan blue green red tan -- 1 3
green red blue green green -- 0 3
red blue blue blue blue -- 1 1
red red red red red -- 1 0
val it = () : unit

```

In selecting the hidden code, a list of random numbers must be generated. This can be done using the `Random` structure in SML which is, unfortunately, not very well documented. The structure provides a type `rand` which is the internal state of a random number generator. To create a generator one must

provide an initial seed (a pair of integers) to the `Random.rand` function. After that, calls to the function `Random.randRange` will generate random numbers on a given integer range.

- (e) [0.5 mark] Define a `Mastermind` structure which groups all the above functions and the `colour` and `row` type definitions, and hides everything else (all the auxiliary functions, etc).
- (f) [1 mark] The remaining functions are used to build a Mastermind player, i.e., a version where the computer plays against itself (of course, without looking into the hidden code). Most strategies for this include initially computing the entire range of possible codes, and then using each guess to eliminate possibilities which are not longer possible until the hidden code is found Write a function `all_codes = fn : colour list -> int -> colour list list` which takes a list of allowed colours and the size of the board, and returns the list of all possible codes in any order.

g (Optional) [2 marks] The optional question aims to build a mastermind player. This is a function

```
player = fn : int -> colour list -> row option -> unit
```

which takes a size of board, the list of allowed colours, and optionally a hidden code. Then it starts a game aiming to find the correct answer.

The aim is to build a player that finds the correct answer as quickly as possible. Programs that do not consistently solve a game of size 4 with 5 colours within 12 steps will gain no bonus marks. The three submissions which consistently find the correct answer in the least possible number of steps without taking too much computation time will be awarded a prize containing chocolate.

The following is a sample execution:

```

player 5 ["red","blue","green","tan","mauve","purple"] (SOME(["red", "blue","green","red","mauve"]));
Past history (from most recent):
red red red red red -- 2 0
Past history (from most recent):
red red blue blue blue -- 1 2
red red red red red -- 2 0
Past history (from most recent):
red blue red green green -- 2 2
red red blue blue blue -- 1 2
red red red red red -- 2 0
Past history (from most recent):
red blue green red tan -- 4 0
red blue red green green -- 2 2
red red blue blue blue -- 1 2
red red red red red -- 2 0
Won with past history:
red blue green red mauve -- 5 0
red blue green red tan -- 4 0
red blue red green green -- 2 2
red red blue blue blue -- 1 2
red red red red red -- 2 0
val it = () : unit

```

Submission Instructions

The above exercises contribute 10% to your total CSE3322 mark. Optional marks will help cover missing marks in this or other assignments/exam.

If possible, you should get your assignment marked by the tutor in one of the optional tutorial classes for CSE3322. Regardless of whether your assignment has been marked or not by the tutor, you should submit your assignment electronically. You should use `/cs/cc/bin/submit` to submit a file called `mastermind.ml` containing the above functions and appropriate documentation. The assignment name is `ass2`.

Furthermore, your submission must include the following declaration at the top of submitted file (failure to do so will result in 0 marks).

(* MONASH UNIVERSITY, School of Computer Science and Software Engineering Student Declaration for CSE3322 Submission I #YOUR NAME#, ID: #YOUR ID NUMBER# declare that this submission is my own work and has not been copied from any other source without attribution. I acknowledge that severe penalties exist for any copying of code without attribution, including a mark of 0 for this assessment. *)

Assignments handed in after the due date will attract a late penalty of 10% per day unless special consideration applies or there has been prior agreement in writing from the lecturer. **No submission will be accepted after 12 noon Tuesday 31st of August.** This is because Maria goes overseas on Friday the 3rd of September for a month and she wants to (a) return the marked assignments, and (b) hand the solution while she is still in Melbourne

No assignment-related questions will be answered by e-mail to students who have not first tried to solve their questions during consultation hours. No assignment-related questions will be answered after 2pm Friday 27th of August.

Marking guide

Your programs will be marked on correctness, style, efficiency, clarity, consistent indenting, appropriate selection of names, and documentation.

In particular, here are some general things that get you a zero in the associated question:

- Having the wrong function name and/or type
- Not doing a function or having one that does not work
- Having compilation errors (syntax errors, etc)
- Having lines longer than 80 characters (very difficult to read, and thus mark)
- Having insufficient comments
- Having too many comments (be informative but succinct please!)
- Having comments within the body of a function (all related comments must appear *before* each function, otherwise they are very difficult to read, and thus mark)
- Using `xs@[x]`.

Some of the things (not all) that will lower your mark in the associated question are:

- Using `hd` or `tl` when you can use pattern matching (usually, to the left of the '=' during a function definition).
- Not using `let` to break up complex expressions
- Using too many `lets`
- Not realising that equality works with any equality type
- Using "`let (A,B) = ... in (A,B) end`" instead of simply "`(A,B) = ...`"
- Using "`if B then true else false`" instead of simply "`B`"
- Using "`orelse false`"
- Using `[x]@xs` instead of `x::xs`.
- Not using anonymous variables when appropriate
- Not using in-built higher order functions such as `List.map`, `List.foldr`, `List.filter`, `List.exists`, `List.all`, etc, whenever their use results in clear and natural code.
- Defining general predicates which already exist in the `List`, `Integer`, etc structures (such as `List.nth`, `Int.abs`)
- Not using anonymous functions when appropriate (small, once-called functions)