COL 100M - Lab Exam 3

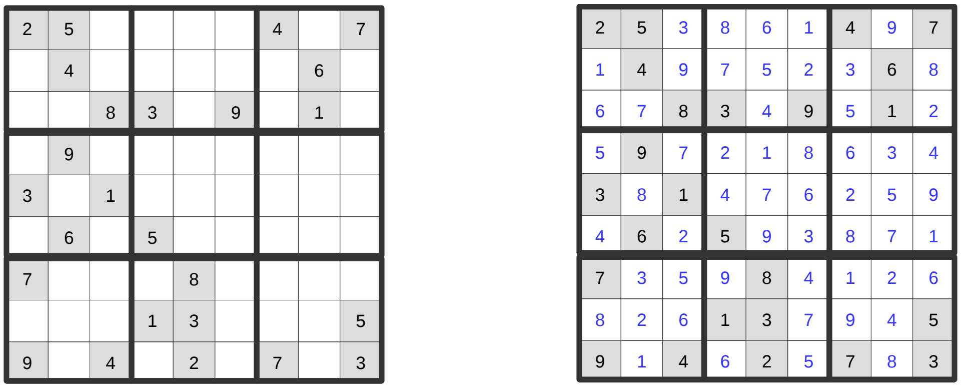
April 7, 2018

Instructions

- This exam consists of five questions, each with sub parts.
- No notes, phones, local or internet resources are allowed.
- Submit the file : **test.ml**
- You may “Run” or “Evaluate” your code.
  - When you “Run” your code, it is executed against a few test cases and both the test case and the output of your program are visible to you. This is to make it easier for you to debug your code, and does not affect your grade. Note that even if your code passes all the “Run” test cases, this does not guarantee that your code is 100 % correct. Use the 🚀 button to “Run” your code.
  - When you “Evaluate”, your code is run against hidden test cases and your grade is updated according to the output.
- A password will be announced in class that you can use to submit code on Moodle. You will be allowed to submit / evaluate your code at most 20 times. No limit on ‘Run’.

1 Sudoku

Sudoku is a hugely popular logic puzzle game that involves filling in numbers on a square grid such that certain constraints are obeyed. In the 9×9 puzzle for example, there are 9 3×3 sub-grids, or *boxes*, and each row, column, and box must contain each number in $1 \ldots 9$ exactly once. The Sudoku puzzle is a partially filled grid. For instance, the puzzle in Figure 1a has a solution in Figure 1b.



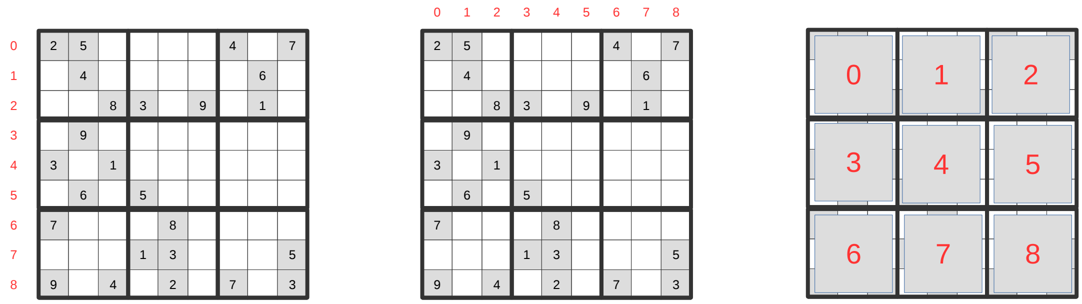
2	5					4		7
	4						6	
		8	3		9		1	
	9							
3		1						
	6		5					
7				8				
			1	3				5
9		4		2		7		3

(a) Sudoku Puzzle

2	5	3	8	6	1	4	9	7
1	4	9	7	5	2	3	6	8
6	7	8	3	4	9	5	1	2
5	9	7	2	1	8	6	3	4
3	8	1	4	7	6	2	5	9
4	6	2	5	9	3	8	7	1
7	3	5	9	8	4	1	2	6
8	2	6	1	3	7	9	4	5
9	1	4	6	2	5	7	8	3

(b) Sudoku Solution

Figure 1: Sudoku puzzle



(a) Row Numbers

(b) Column Numbers

(c) Box Numbers

Figure 2: Row, Column, and Box Numbers in the Sudoku grid

2 Exam

In this exam, you will write a program to solve a sudoku puzzle. Your program will implement a 'humanistic' method, which uses strategies commonly applied by people to eliminate choices and fill cells in the grid. In addition, you will implement a back-tracking technique to fill the sudoku grid. Because the humanistic techniques outlined here may not be enough to solve all sudoku puzzles, we will employ the following approach. In the first step, we fill as many cells, and eliminate as many choices as possible using only humanistic methods. In the second step, we fill whatever cells remain using the back-tracking method, which effectively tries every possible combination of candidates in unfilled cells until it arrives at a solution.

2.1 The Cell type

The Sudoku will be modeled as a 2 dimensional `Array` of type `cell`, which is defined for you as the following variant:

```
type cell =
| Value of int
| PossibleValues of int list
```

The `cell` type is defined in the file `cell.ml`. Use `Open Cell` in your submission file `test.ml` to include the variant type into your code.

If a `cell` is filled, it is a `Value` cell, otherwise it is a `PossibleValues` cell. The Sudoku will be initialized for you, with each unfilled cell being `PossibleValues [1 ...9]`. Your final goal is to return the Sudoku solution, a 2 dimensional `Array` of type `cell`, where each entry is of type `Value`.

2.2 The Sudoku Grid

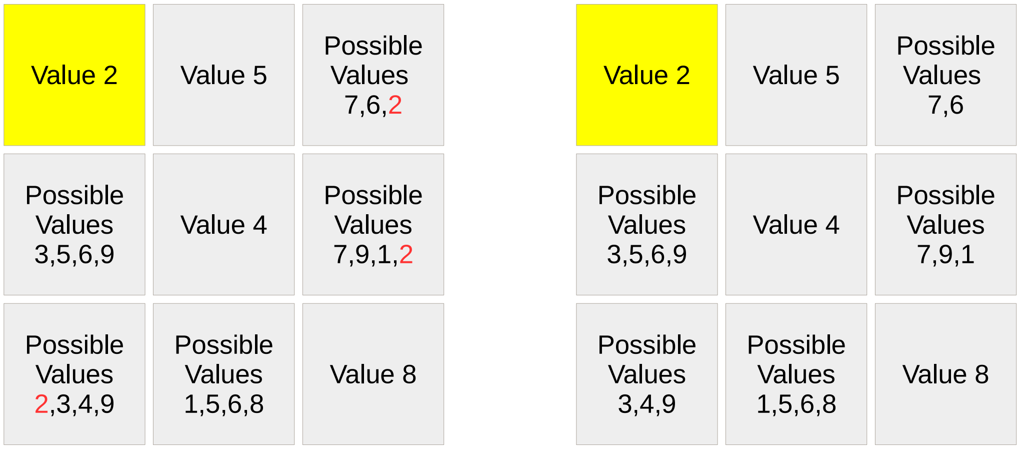
Assume that the sudoku is a 9×9 **grid.** Rows, columns, and boxes in the Sudoku grid are numbered from $0 \ldots 8$. When asked to perform operations on a particular row, column, or box, refer to Figure 2. For instance, row 0 is the top row in the grid, while column 8 is the right most column. Box 4 is the middle box in the grid i.e. it contains all cells with both row and column numbers in (3,4,5).

3 Humanistic Methods [4.95 marks]

3.1 Elimination

Elimination is the most frequently used heuristic. It removes the value of a filled cell from the set of possible candidates in the cell's row, column, and box. For example, in Figure 3a, 2 can be removed as a possible value from all other cells since it is already filled in one of the cells in the box.

- Write a function `eliminateValueRow : cell array array -> int -> int -> bool` such that `eliminateValueRow sudoku v r` removes `v` as a possible candidate in all `PossibleValues` cells in the row `r` of `sudoku`. It returns `true` if anything changes in the `sudoku` grid (i.e., if any elimination does actually happen), and `false` otherwise.



Value 2	Value 5	Possible Values 7,6,2
Possible Values 3,5,6,9	Value 4	Possible Values 7,9,1,2
Possible Values 2,3,4,9	Possible Values 1,5,6,8	Value 8

(a) Initial

Value 2	Value 5	Possible Values 7,6
Possible Values 3,5,6,9	Value 4	Possible Values 7,9,1
Possible Values 3,4,9	Possible Values 1,5,6,8	Value 8

(b) Final

Figure 3: Eliminating 2 from the box

- Write a function `eliminateValueCol : cell array array -> int -> int -> bool` such that `eliminateValueCol sudoku v c` removes `v` as a possible candidate in all `PossibleValues` cells in the column `c` of `sudoku`. It returns `true` if anything changes in the `sudoku` grid, and `false` otherwise.

- Write a function `eliminateValueBox : cell array array -> int -> int -> bool` such that `eliminateValueBox sudoku v b` removes `v` as a possible candidate in all `PossibleValues` cells of the box `b` of `sudoku`. It returns `true` if anything changes in the `sudoku` grid, and `false` otherwise.

- Write a function `eliminate : cell array array -> int -> int -> bool` such that `eliminate sudoku i j` checks if cell `[i,j]` is a `Value` cell. If so, it removes the value of cell `[i,j]` as a possible candidate in all `PossibleValues` cells of the corresponding row, column, and box. If not, it does nothing. It returns `true` if anything changes in the `sudoku` grid, and `false` otherwise.

3.2 Lone Cell

The lone cell heuristic looks for `PossibleValues` cells with only one candidate. If a cell has only one candidate value, it may be filled with that value.

Write a function `loneCells : cell array array -> bool` to identify lone cells in the Sudoku puzzle. `loneCells sudoku` iterates over each cell in the Sudoku grid. If the cell `[i,j]` is a `PossibleValues` cell with only one candidate value, it converts this cell into a `Value` cell and then calls `eliminate sudoku i j`. It returns `true` if anything changes in the `sudoku` grid, and `false` otherwise.

Note that the above procedure might create fresh lone cells, but you do not have to identify or assign them again in the same function call.

3.3 Lone Ranger

A lone ranger is a value that appears as a candidate in only one cell in a given row, column, or box. In Figure 4a, 7 is a lone ranger in the given row, since it appears as a possible value in only one cell. A lone ranger can be assigned to the cell it appears as a candidate in.

- Write a function `getCellsRow : cell array array -> int -> int -> (int * int) list` such that `getCellsRow sudoku r v` returns the indexes of the `PossibleValues` cells in row `r` that include `v` as a candidate. The returned list is a list of tuples, each tuple specifies a cell in the grid as a `(row,column)` pair.

- Write a function `getCellsCol : cell array array -> int -> int -> (int * int) list` such that `getCellsCol sudoku c v` returns the indexes of the `PossibleValues` cells in column `c` that include `v` as a candidate.

PV 1,2	PV 3	PV 3,4	PV 1,2,5	V 6	PV 5,7	PV 8,9	PV 8,9,1	V 2

(a) Initial

PV 1,2	PV 3	PV 3,4	PV 1,2,5	V 6	V 7	PV 8,9	PV 8,9,1	V 2

(b) Final

Figure 4: Lone Ranger (PV : `PossibleValues`, V: `Value`)

PV 1,2,4	PV 3	PV 3,4	PV 3,5,7	PV 6	PV 5,6,7	PV 1,2,8	PV 8,9,1	PV 2

(a) Initial

PV 1,2,4	PV 3	PV 3,4	PV 5,7	PV 6	PV 5,7	PV 1,2,8	PV 8,9,1	PV 2

(b) Final

Figure 5: Twins

- Write a function `getCellsBox : cell array array -> int -> int -> (int * int) list` such that `getCellsBox sudoku b v` returns the indexes of the `PossibleValues` cells in box `b` that include `v` as a candidate.

- Write a function `loneRanger : cell array array -> int -> (cell array array -> int -> int -> (int * int) list) -> bool`. The first argument is `sudoku`, the second argument is the row/ column/ box number on which the lone ranger is to be found, and the third argument is a function, one of `getCellsRow, getCellsCol, getCellsBox`, depending on whether the lone ranger is to be found on a row, column, or box. If a lone ranger is found at cell `[i,j]`, it should convert cell `[i,j]` to a `Value` cell and call `eliminate sudoku i j`. `loneRanger sudoku id f` returns `true` if anything changes in the `sudoku` grid, and `false` otherwise.

3.4 Twins

A twin in a row, column, or box is a pair of numbers that appear as candidates in the same two cells, and no where else. In figure 5a, 5 and 7 appear in the same pair of cells and no where else. Therefore, we can eliminate the *remaining possible values* in both cells.

Write a function `getTwin : cell array array -> int -> (cell array array -> int -> int -> (int * int) list) -> bool.` . The first argument is `sudoku`, the second argument is the row/ column/ box number on which twins are to be found, and the third argument is a function, one of `getCellsRow, getCellsCol, getCellsBox`, depending on whether the twins are to be found on a row, column, or box. If a twin is found, it removes all other possible values from both cells. `getTwin sudoku id f` returns `true` if anything changes in the `sudoku` grid, and `false` otherwise.

3.5 Solving the Sudoku

To solve the Sudoku puzzle, the above techniques must be applied repeatedly, until *no unfilled cell remains* (that is, solution has been found) or until it can be determined that *no solution can be found.*

The following algorithm outlines the steps.

```
1. While there is an unfilled cell in sudoku do
   (a) for each cell [i,j] in sudoku
           Call eliminate sudoku i j
   (b) call loneCells sudoku.
   (c) For each row, column, and box
           Call loneRanger sudoku id f
           (* id is the row / column / or box number from 0...8, and f is one of getCellsRow /
           getCellsCol / getCellsBox *)
   (d) For each row, column, and box
           Call getTwin sudoku id f
   (e) if sudoku has not changed in any of the steps in (a) to (d), then exit – no more cells can be filled
       in the sudoku. Otherwise, return to step 1.
```

Write a function `solveHumanistic : cell array array -> unit` which solves the sudoku grid using only humanistic techniques. Even if a solution cannot be found eventually, it should fill in as many cells and eliminate as many choices from unfilled cells as possible using only the outlined humanistic techniques.

4 Back-tracking

Often, the above techniques will be unable to solve the Sudoku puzzle. To solve the remaining parts of the puzzle, we will adopt a brute force technique. The following recursive procedure systematically tries each combination of the candidates in the cells until it arrives at a solution.

- Identify the left most cell in the top most row that is unfilled, say `c`.
- If no such cell exists, solution is found. Return `true`.
- Let `L` be the set of candidates of `c`.
- For each `x in L`
  - If `x` not filled in any other cell in `c`'s row, column, and box
    - fill `x` in `c`
    - Recursively call the procedure on the resulting sudoku.
    - If the call returns `true`, a solution is found. Return `true`.
- Reset `c` to the set of candidates `L`. Return `false`.

[**5.05 marks**] Write a function `solveBruteForce : cell array array -> bool` that solves the sudoku grid using the brute force technique. It should return `true` if a solution was found, `false` otherwise.

5 Appendix

In this 'help' section, you are provided with a few code snippets that will help you with the syntax of arrays and loops. Note that you are free to use recursion or any other data structures internally in your code as long as you conform to the function signatures given in this specification.

5.1 Arrays

1. We create `arr`, a two dimensional array of integers with 8 rows and 4 columns, where each entry is initialized to 0.

   ```
   # let arr = Array.make_matrix 8 4 0;;
   val arr : int array array ...
   ```

2. Accessing array elements : To access the element at the row 5 and column 3 (where numbering starts from 0), we write

   ```
   # arr.(5).(3);;
   - : int = 0
   # get (get arr 5) 3;; (* alternatively *)
   - : int = 0
   # get arr 5;; (* get the entire 5th row *)
   - : int array = [|0; 0; 0; 0|]
   ```

   We can also set array elements. In the following, we set the element in row 5 and column 3 to 15.

   ```
   # set (get arr 5) 3 15;;
   - : unit = ()
   # get arr 5;; (* get the entire 5th row, element 3 should be 15 *)
   - : int array = [|0; 0; 0; 15|]
   ```

5.2 For and While Loops on an Array

Suppose we wanted to set each entry `[i,j]` in `arr` to `i * j`. Then, the following *nested* `for` loop can be used.

```
# for i = 0 to 7 do
    for j = 0 to 3 do
      set (get arr i) j (i*j)
    done
  done;;
- : unit = ()
# get arr 5;;
- : int array = [|0; 5; 10; 15|]
```

Let us say we want to traverse row 5 of `arr`, and return the index of the left most element that is greater than 5.

```
# let x = get arr 5;;
val x : int array = [|0; 5; 10; 15|]
# let i = ref 0 in
  (
    while (x.(!i)  <= 5) do (* (!i) gets the value of ref variable i *)
      i := (!i) + 1
    done;
    (!i) (* The statement returns the value of i*)
  );;
- : int = 2
```