Course Project: Sudoku

ITI0212

Due: 05/31/2024

1 Introduction

The purpose of this project is to implement a Sudoku game using the Idris programming language. Sudoku is a popular puzzle game that involves filling in a 9x9 grid with digits so that each column, each row, and each of the nine 3x3 sub-grids contain all of the digits from 1 to 9. The goal of the game is to fill in the entire grid with the correct digits.

For this project, you will define a representation for the Sudoku table, and implement some basic functions that act on the table. Then, you will create a parser and a pretty-printer that can read and write Sudoku tables to files. Finally, you will be guided towards writing a fully automatic solver for Sudoku tables.

1.1 About the project

These tasks are intentionally less precisely specified than those in the labs and homeworks. You will need to use good judgement and programming practice to decompose each task into parts (types, functions, interfaces, etc.) that work together in a logical, modular, efficient, and elegant way to solve the problem at hand. This leaves you with quite a lot of freedom, and you may take any approach you like provided that it conforms to the specifications and the spirit of the assignment.

For example, it would violate the spirit of the assignment if you were to find an already implemented solution online, import it, and simply call the relevant functions. Thus, except for in the last task you must obtain permission from an instructor if you wish to use modules other than those found in the standard library.

Your solution to each of the following tasks should include a comment in your source file specifying the task number and briefly explaining the structure of your solution. If there are some parts of your program that are likely to be confusing, are unimplemented, or that contain known bugs, you should explain them in comments as well.

In short, use good coding practice so that readers of your program (including your future self) can understand as easily as possible how it works. In any event, your submitted source file(s) should load without syntax or type errors. Submit your project by pushing it to your course GitLab repository in a directory called `project`. Your submission should include a plain-text `README` file containing any information needed to load and run your program as intended.

The assignment will be marked out of 100 points. As with the homework assignments, approximately 2/3 of this will be for correctness and functionality, with the other 1/3 for programming style (clarity, concision, efficiency).

2 Example

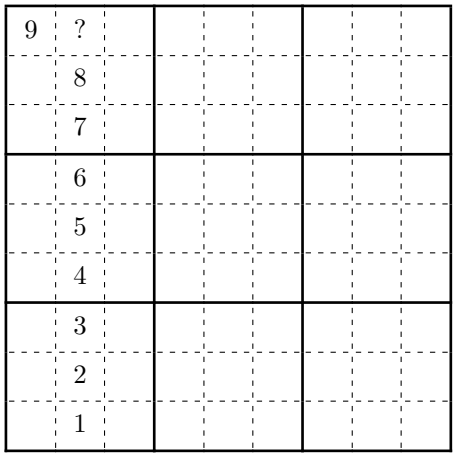
Here is an example of a possible input Sudoku table:

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

In this example some of the cells are filled in with numbers, while others are empty. The goal of the game is to fill in the empty cells with the correct numbers so that each row, column, and 3x3 sub-grid contains all of the digits from 1 to 9. In this particular grid, the solution is:

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

Notice how each row, column, and 3x3 sub-grid contains all of the digits from 1 to 9 *exactly once*. Note that not all Sudokus have a unique solution, and that some Sudokus have no solution at all. We give an example for this last case:



Here the Sudoku is *valid* (i.e., there are no duplicate numbers in any row, column or subsquare), but there is no way to fill the cell indicated with a `'?'`.

3 Tasks

In this section we introduce the main points of the project that you should implement. In order to get the maximum grade, you should implement all the tasks in this section. If you implement fewer tasks than the ones required, your score will be proportional to the difficulty of the tasks you successfully solved and the quality of your solutions. You might find it helpful to start working through the tasks in order, since they build on each other.

Task 1 (Representation)**.** You will need to define a representation `SudokuTable` for the entire Sudoku table and all the numbers inside a table. You might choose any representation to your liking, but it should eventually be able to support the functions defined below. *(Hint: A good table representation should be able to represent both a partially filled table as well as a complete table.)*

Task 2 (Operations on tables)**.** With this representation, you must also define the following functions:

1. A function `index` such that `index s i j` evaluates to the number in the table `s : SudokuTable` at position `(i, j)`. The function should be able to deal with out of bound indices; you can deal with this problem however you prefer.

2. A function `update` such that, given an input table `s : SudokuTable`, the expression `update s i j n` evaluates to a table `s' : SudokuTable` such that `index s' i j == n`, and the rest of `s'` is the the same as `s`. In other words, the function updates the cell at position (i, j) with the given value. You may again decide how to deal with filling already filled cells, and out of bound indices.

3. A function `difficulty` which, given a table `s : SudokuTable`, returns the *difficulty* of a Sudoku grid. The *difficulty* of a table is defined as the sum of the number of empty cells in the table divided by the total number of cells in the grid.

4. A function `isValid` that checks if the given table `s : SudokuTable` is *valid*, meaning that each digit from 1 to 9 appears at most once in each row, column, and 3x3 sub-grid (so in particular, a completely empty grid should be valid).

5. A function `isComplete` that checks if the table is *complete*, meaning that the table is *valid* and all cells have been filled with digits.

Before diving into coding, we encourage you to take a pen and paper, and define specifications for you project, by answering questions like:

- What is a good representation for a Sudoku table that is easy to manipulate and work with?
- How should we represent cells and their contents in a Sudoku table?
- What would/should happen if a user tried to access an out-of-bound index in the table, etc.

Task 3 (Parser)**.** You will need to implement a parser that can read in a Sudoku table from a file. The parser should be able to handle files that contain partially filled tables as well as complete tables. You are free to choose whatever format you like for the input file, but ideally it should be human-readable and easy to edit. *You should also include a sample input file in your submission along with a simple testing function*, so that we can more easily see that your parser works. You are free to deal with invalid input files however you prefer.

Task 4 (Pretty-printer)**.** You will need to implement a pretty-printer that can output a Sudoku table to a file or to a terminal in a human-readable format. We gave you full freedom to choose a nice representation for the *input* table; for the pretty-printer cell, however, we ask you to output the table in a very specific format:

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

should be printed as:

```
+-------+-------+-------+
| 5 3   |   7   |       |
| 6     | 1 9 5 |       |
|   9 8 |       |   6   |
+------ +-------+-------+
| 8     |   6   |     3 |
| 4     | 8   3 |     1 |
| 7     |   2   |     6 |
+------ +-------+-------+
|   6   |       | 2 8   |
|       | 4 1 9 |     5 |
|       |   8   |   7 9 |
+-------+-------+-------+
```

(Thus, an empty cell should be represented by a space).

Task 5 (Interactive game)**.** Write an interactive game in which the user can progressively fill the Sudoku table, with the program checking whether the numbers and the positions inserted by the user are valid. For instance, a possible game interaction could happen as follows:

```
SudokuProject> Current grid, 1 cell(s) missing:
+-------+-------+-------+
| 5   4 | 6 7 8 | 9 1 2 |
| 6 7 2 | 1 9 5 | 3 4 8 |
| 1 9 8 | 3 4 2 | 5 6 7 |
+-------+-------+-------+
| 8 5 9 | 7 6 1 | 4 2 3 |
| 4 2 6 | 8 5 3 | 7 9 1 |
| 7 1 3 | 9 2 4 | 8 5 6 |
+-------+-------+-------+
| 9 6 1 | 5 3 7 | 2 8 4 |
| 2 8 7 | 4 1 9 | 6 3 5 |
| 3 4 5 | 2 8 6 | 1 7 9 |
+-------+-------+-------+
Enter (i, j)-coordinates:
>>2 1
SudokuProject> Enter the number to fill the cell (2, 1):
>>5
SudokuProject> Impossible to fill the cell (2, 1) with 5. Enter another number:
>>3
SudokuProject> Congratulations, the grid is now solved.
+-------+-------+-------+
| 5 3 4 | 6 7 8 | 9 1 2 |
| 6 7 2 | 1 9 5 | 3 4 8 |
| 1 9 8 | 3 4 2 | 5 6 7 |
+-------+-------+-------+
| 8 5 9 | 7 6 1 | 4 2 3 |
| 4 2 6 | 8 5 3 | 7 9 1 |
| 7 1 3 | 9 2 4 | 8 5 6 |
+-------+-------+-------+
| 9 6 1 | 5 3 7 | 2 8 4 |
| 2 8 7 | 4 1 9 | 6 3 5 |
| 3 4 5 | 2 8 6 | 1 7 9 |
+-------+-------+-------+
```

3.1 Solver

Finally, we will guide you towards implementing a Sudoku solver that can provide a solution for a partially filled Sudoku table.

There is a number of approaches that can be taken to tackle the problem of solving a Sudoku table. Among these:

(A) You might want to implement a brute-force search, in which you simply test all possible numbers for each square, and test whether the final table is a valid solution;

(B) You could implement a more sophisticated algorithm that uses backtracking: try to place a number in a given square, and whenever the table cannot be filled you go back and try another with number;

(C) You can imagine starting by putting in each hole the list of numbers that would fit, and then progressively remove from each list the numbers that are already present until a solution is reached.

We present you two implementation guides that you can choose to follow: one is based on approach B and implemented using the `List` monad; the other follows approach C using progressive refinements of the table.

You can choose to follow either approach to implement the solver, we only require you to implement *one* of the two approaches listed below.

3.2 Approach B: Backtracking in the List monad

One of the interpretation for the `List` monad is that computation in this monad represents *non-deterministic computation*.

```
do
-- ... some previous computation
x <- xs
-- ... whatever computation follows will be performed
-- for any possible choice of x in xs.
```

An example: let's compute the Cartesian product of two lists:

```
l1 = [1, 2]
l2 = ['a', 'b', 'c']
prod = do
x <- l1
y <- l2
pure (x, y)

-- prod == [(1, 'a'), (1, 'b'), (1, 'c'), (2, 'a'), (2, 'b'), (2, 'c')]
```

This definition of `prod` can be read as follows: first take any element `x` from `l1`, then take any element `y` from `l2`, then put them together in a tuple.

A function returning a list `l` represents a nondeterministic computation returning any of the elements in `l`. In particular, `pure` represents the monadic computation that terminates with precisely one result. Furthermore, a computation returning the empty list is a computation that has no result at all.

This behaviour allows to write computations in which you have to make choices when you are unsure of the choice to be made, or when you have to return the result of all possible choices in the process. You make a choice, and if it turns out to be wrong, you return the empty list. The `List` monad takes care of the backtracking for you.

Let us find all the pairs of integers from 0 to 3 such that their sum is even:

```
even : Integer -> Bool
even n = n `mod` 2 == 0

l3 = [0, 1, 2, 3]
evenSums = do
```

```
x <- l3
y <- l3 -- take two elements from l3
if (even (x + y)) -- if their sum is even
then pure (x, y) -- they are a good pick, so return them
else [] -- else return nothing

-- evenSums == [(0, 0), (0, 2), (1, 1), (1, 3), (2, 0), (2, 2), (3, 1), (3, 3)]
```

A way to obtain all possible solutions to a sudoku then is the following:

- if the sudoku is not valid, that is, the digits that were already inserted do not respect the rules, than it has *no solution*;
- otherwise, if the sudoku is completely filled, it has only one solution, that is, itself;
- finally, if the sudoku is not completely filled, we take the first hole, then we *nondeterministically* pick a digit to insert in the hole, we update the sudoku and we solve the updated version.

Task 6 (Check for holes)**.** Your sudoku representation should allow tables with hole, that is, cells that have not been filled with a digit. Write a function `findHole` that given a sudoku returns `Nothing` if it is completely filled, and `Just (i, j)` if it finds a hole at coordinates i, j. *Hint: better not to scan the whole table if a hole is found soon enough.*

Task 7 (Solve the sudoku!)**.** Write a function `solve` that takes a sudoku and applies the approach hinted above to return a `List` of all possible solutions. You should use the `findHole` function you just wrote, and the `isValid` function that you should have by now.

3.3 Approach C: Progressive refinement

In this section, we will guide you through developing a solver by adopting a slightly different representation for tables. The core idea to write a solver is the following: use a clever representation that represents holes in the table with *list of (valid) possibilities*.

To each cell in the grid, we associated it a list whose entries should represent a possible number that hasn't appeared in either the corresponding row, column or 3x3 square in the grid, in such a way that the cell could be filled with any of the values in the list and still be a valid Sudoku grid.

Example 1. Let's make an example. Consider the following table:

5		4	6		8	9	1	2
6	7	2	1	9		3		8
	9	8	3	4	2	5	6	
8	5	9	7	6	1	4	2	3
		6	8	*c*	3	7	9	1
7	1	3	9	2	4	8	5	6
9	6		5		7	2	8	
2	8	7		1		6		5
t		5	2		6		7	9

Take t to be the left-most cell of the Sudoku on the last row. The list of possibilities for t are [1,4,3], since those are the only *valid* numbers that can be placed in the cell; if you put any other number there, the table would become invalid.

Or, consider the cell c placed at the very center of the board. The list of possibilities for c is just [5], since every other number is present in either its corresponding row, column, or 3x3 square. Since the list of possibilities for c is a singleton, we can conclude that c *must* be 5.

Since we now know that 5 must be in the center of the board, we can *remove* 5 from the list of possibilities for all the cells in the same row, column and 3x3 square as c. This will eventually cause other lists to become singletons, which we can propagate again, and so on until we reach a point where the table does not change anymore.

You might find it useful to define the following functions, which we provide as small subtasks.

Task 8 (Constraint propagation)**.** Define a function which, given a (valid) position in the table (i, j) and a number n, removes the number n from the possibility lists of all the cells in the corresponding i-th row, in the corresponding j-th column, and in the corresponding 3x3 square of the Sudoku where (i, j) is contained.
Note: you should not remove n from the possibility list of the cell (i, j) itself.

Task 9 (Propagate all constraints)**.** Define a function which applies the constraint propagation function to each cell where there is only one number in the list of possibilities.

The idea behind the solver is the following: we start with a table where each number that is already present in the input table is represented as a singleton list with the number itself, and each hole is represented initially as the list of all possibilities [1,2,3,4,5,6,7,8,9]. Then, you repeat the process of propagating constraints on singleton cells, so that the lists on each hole decreases in size. We keep repeating this until the table does not change anymore, at which point one of three possible cases will be reached:

- There is only *one possibility* (singleton list) for each cell inside the table, so the table has been successfully completed;

- There are still some cells which have more than one possibility: this means that the initial input table does not have an unique solution, and so it is *underspecified*. It is okay to simply tell the user that the table is underspecified, so you are not required to output every possible solution of the Sudoku.
- There's at least one cell with *no possibilities* (empty list), so the table is impossible to solve!

Task 10 (Solve the sudoku!)**.** Write a function which solves a Sudoku table by iterating the *propagate-all-constraints* function until the table does not change anymore. One of the three states described above will be reached.

4 Project submission

Your project should include the following:

- A fully documented Idris code that implements the requirements specified above.
- A README file that describes how to run your program and any dependencies that are required.
- A sample input file that contains a partially filled Sudoku table.
- A sample `Game.idr` Idris file whose `main` function allows you to play the interactive game described in 5.
- A sample `TestSolve.idr` Idris file whose `main` function reads the input Sudoku file you gave in the previous bullet point, pretty-prints the input Sudoku to the terminal, and uses your solver on the Sudoku.
- The rest of the code can be structured and placed in your folder the way you prefer, as long as all tasks are implemented.