CSA2001
Programming Paradigms Assignment 3
Functional Paradigm

Mark Galea
mgale11@um.edu.mt

21st November 2016

Your task, is to implement a sudoku solver. The aim of this assignment is to strengthen your understanding of the functional programming paradigm, and see its application with a practical example.

1 Preliminaries

This is the third assignment for CSA2001; one more remaining. It carries 10% of the final mark and hence will be scored out of 10 points as follows

- **1 point** for the Report
- **3 points** for the Sudoku Representation Implementation
- **6 points** for the Sudoku Solver Algorithm

The deadline for submission of the complete assignment is at **NOON** on **12th of December 2016** and it must be carried out **individually**.

You are reminded that we take plagiarism seriously. While you are encouraged to discuss ideas with fellow students and perform additional research you cannot copy and steal ideas. Read and follow the information found at `https://www.um.edu.mt/ict/Plagiarism` carefully. Extreme measures might be taken in case of plagiarism. ***Note that you might be randomly selected for a 10 minute interview to discuss your implementation***.

2 Task

In this assignment you will implement a solver for Sudoku which only requires the *singleton/lone number* solving technique.

The objective of Sudoku is to fill up a 9×9 grid of numbers such that each row, column and 3×3 block has all the digits ranging from 1 to 9. If you are unfamiliar with the game I encourage you to *play* a few games at `http://www.sudoku-solutions.com`. *Note that you will only be required to solve simple puzzle types.*

3 Solution Outline

Given an initial (partially solved) sudoku puzzle S_0, we will explore the state-space until we reach an end state S_n representing the solved sudoku puzzle. Given the intermediate steps $S_1, S_2 \dots, S_{n-1}$ the walk through the state-space can be represented as $S_0 \to S_1 \to S_2 \to \dots \to S_{n-1} \to S_n$.

Randomly choosing the next state can lead to *random walks* and thus it is essential to choose an appropriate walking strategy. Given a state S_x it is essential that S_x subsumes S_{x-1} i.e. S_x contains all the previously defined cells in S_{x-1}. One strategy which satisfied this property is the *singleton solution* strategy. This solution technique will transition to the next state by looking up cells which only have one single candidate value.

Consider the example below, starting from state S_0, we can transition to state S_1 by noting that the hypothesis set at cell x contains only one element - $\{1\}$. *Note that the column values* $\{2,5,8\}$, *row values* $\{3,9,4,6,7\}$ *and block values* $\{2,8,3,9\}$ *can be eliminated from the hypothesis set* $\{1 \dots 9\}$.

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

Unsolved Sudoku - State S_0

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

Partial Solution - State S_1

3.1 Sudoku Representation

The class `Sudoku` will represented the sudoku puzzle as a *list* of rows each containing a *list* of *integers* i.e. `List[List[Int]]`. Unknown cells will be represented by the number 0.

```
package mt.edu.um
class Sudoku(val grid: List[List[Int]]) {
  def row (row: Int): Set[Int] = ???
  def column(column: Int): Set[Int] = ???
  def block(block: Int): Set[Int] = ???
  override def toString() = ???
}
```

You will start by creating valid implementations for the `row`, `column` and `block` functions. These functions take an index as parameter and return the content of the row, column, or block respectively. To help you visualise the problem you will also implement the `toString()` function. Given the instantiation `ex1`,

```scala
val ex1 = new Sudoku(List(
   List(0, 0, 5, 0, 0, 6, 3, 0, 0),
   List(0, 0, 0, 0, 0, 0, 4, 0, 0),
   List(9, 8, 0, 7, 4, 0, 0, 0, 5),
   List(1, 0, 0, 0, 7, 0, 9, 0, 0),
   List(0, 0, 9, 5, 0, 1, 6, 0, 0),
   List(0, 0, 8, 0, 2, 0, 0, 0, 7),
   List(6, 0, 0, 0, 1, 8, 0, 9, 3),
   List(0, 0, 1, 0, 0, 0, 0, 0, 0),
   List(0, 0, 4, 2, 0, 0, 5, 0, 0)
 ))
```

println(ex1) should print out the puzzle ***exactly*** as follows[1].

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

To make sure you got the correct implementation consider trying out the following test cases on instance ex1.

Test	**Value**
ex1.row(0)	Set(5, 6, 3)
ex1.row(1)	Set(4)
ex1.row(2)	Set(5, 9, 7, 8, 4)
ex1.column(8)	Set(5, 7, 3)
ex1.column(0)	Set(9, 1, 6)
ex1.column(1)	Set(8)
ex1.column(2)	Set(5, 1, 9, 8, 4)
ex1.block(0)	Set(5, 9, 8)
ex1.block(1)	Set(6, 7, 4)
ex1.block(3)	Set(1, 9, 8)
ex1.block(8)	Set(9, 3, 5)

3.2 Sudoku Solver

Now that you have your representation set up you can focus on the solver. As you have seen in other functional problems it is beneficial to decompose the problem into smaller ones. Solving a Sudoku puzzle is no different. For this problem you need a way to:

[1]Under the hood Scala will call your toString method.

1. Determine all the possible candidates for each empty element in the grid so as to form a hypothesis set.
2. Analyse all the hypothesis sets and assign values to the empty cells whose hypothesis set cardinality is 1.
3. Check whether the puzzle is complete.

In order to achieve the above functionality, you need to complete the class functions below:

```
package mt.edu.um
class SingletonSolver {
  def solve(sudoku: Sudoku): Sudoku = {
    def hypothesis(r: Int, c: Int): Set[Int] = ???
    def allHypothesis(): List[(Int, Int, Set[Int])] = ???
    def complete(): Boolean = ???
    def step(): Sudoku = ???
    if (complete()) sudoku else solve(step())
  }
}
```

3.2.1 Determining hypothesis

The `hypothesis()` function returns all the possible candidates for a given element in the grid. This function will provide the basis for implementing the `allHypothesis()` function which returns a list of tuples; each tuple will contain the row, column and hypothesis set relevant to that particular row and column.

To make sure you got the correct implementation consider checking the following values on instance `ex1`.

Test	**Value**
`allHypothesis()(0)`	(0,0,Set(2, 7, 4))
`allHypothesis()(1)`	(0,1,Set(1, 2, 7, 4))
`allHypothesis()(2)`	(0,3,Set(1, 9, 8))
`hypothesis(0, 0)`	Set(2,7,4)
`hypothesis(0, 1)`	Set(1,2,4,7)
`hypothesis(2, 2)`	Set(2,3,6)
`hypothesis(0, 3)`	Set(1,8,9)

3.2.2 Hypothesis analysis

The `step()` function will analyse all hypothesis sets and filter those with cardinality 1. Using this (filtered) subset you will take a step in the state-space by instantiating a new instance of the Sudoku problem while updating the relevant cell values.

3.2.3 Completion of the puzzle

The `complete()` function will determine whether the puzzle has been solved. *Note that there are various ways how this can be express.*

4 Notes

1. **Do not** change the given function definitions.
2. **Do not** use any software packages or libraries that provide out-of-the-box solutions.
3. **Do not** use the `var` keyword since this implies mutation of state.
4. **Do not** use loops or other iterative constructs. Use recursive functions.
5. Comment your code when necessary but **do not** comment the obvious.

5 Testing

Once you have implemented `Sudoku` and `SingletonSolver` consider checking your solution with the following puzzles.

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

Sudoku Puzzle 1 - Unsolved

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

Sudoku Puzzle 1 - Solved

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

Sudoku Puzzle 2 - Unsolved

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

Sudoku Puzzle 2 - Solved

6 Report

This assignment is to be accompanied by a concise report - **2 pages max** - outlining the algorithm adopted and implementation details of your Sudoku representation and solver.

7 Submissions

All submissions have to take place by the stipulated deadline.

1. Submit a hard copy of the report and the plagiarism declaration form to Ms. Vanessa Borg.
2. Package all the source code together with the report and submit through VLE.
3. Make sure that the files and functions are named exactly as specified in this document.

8 Conclusion

If you have reasonable questions please contact me on my email address. Obviously I will not provide implementation details however if you require some clarification feel free to contact me. Dedicate enough time for completing this assignment since no **extensions** will be provided. Good luck and happy sudokuing!