Homework 7: Sudoku

For this assignment, you will write a Sudoku solver. To do so, you will (1) use Scala collections extensively, (2) implement a *backtracking search* algorithm, and (3) implement *constraint propagation*.

Preliminaries

We assume you know how to play Sudoku. If you don't, you should play a few games by hand, before attempting this assignment.

The support code for this assignment is in the `hw.sudoku` package. You should create a directory-tree that looks like this:

```
./sudoku
 ├─ project
 │   └─ plugins.sbt
 └─ src
     ├─ main
     │   └─ scala ........................................ Your solution goes here
     └─ test
         └─ scala ........................................ Yours tests go here
```

The `project/plugins.sbt` file must have exactly this line:

```
addSbtPlugin("edu.umass.cs" % "compsci220" % "1.0.2")
```

Overview

A Sudoku board is a 9-by-9 grid, where each cell is either blank or has a value in the range 1–9. For this assignment, we'll use strings of length 81 to represent Sudoku boards, where each block of nine characters represents a successive row. For example, the following string:

```scala
val puzzle =
  "....8.3...6..7..84.3.5..2.9...1.54.8.........4.27.6...3.1..7.4.72..4..6...4.1...3"
```

Represents the following board:

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

Here are some more examples:

```
".43.8.25.6.............1.949....4.7....6.8....1.2....382.5.............5.34.9.71."
"2...8.3...6..7..84.3.5..2.9...1.54.8.........4.27.6...3.1..7.4.72..4..6...4.1...3"
"..3.2.6..9..3.5..1..18.64....81.29..7.......8..67.82....26.95..8..2.3..9..5.1.3.."
"1..92....524.1...........7..5...81.2.........4.27...9..6...........3.945....71..6"
```

Your first task is to parse strings that represent Sudoku boards. You may assume that all strings represent solvable boards and that the string has exactly 81 characters. This part of the assignment should be trivial.

Solving Sudoku puzzles is much harder, but we'll walk you through it.
Backtracking search is a recursive algorithm that operates as follows. Given a a Sudoku board, B:

- If B is already filled completely and a solution, return the solution.
- If B is an invalid board (e.g., two 2s in a row), abort.
- Otherwise, generate a list of all boards that fill in one more square of B. For each generated board, recursively apply the search function:
  - If any board produces a valid solution, return that board
  - If no board produces a solution, abort and return

This approach will work in principle. But, in practice there are too many boards to search: there are 81 squares, and each square can hold 10 values (a digit or blank). Therefore, there are 10^{81} possible boards, which exceeds the number of atoms in the observable universe.

To actually solve Sudoku problems, we need to combine backtracking search with *constraint propogation*. When you play Sudoku yourself, every time you fill a digit into a cell, you can eliminate that digit from several other cells. For example, if you fill 2 into the top-left corner of the board above, you can eliminate 2 from the first row, first column, and first box. i.e., there is no point even trying to place 2 in those spots, since the 2 in the corner *constrains* those cells.

We'll augment backtracking search with constraint propagation, which implements this intuition. The key idea is to store not the value at a cell, but the *set of values* that may be placed in a cell. For example, on the empty board the values 1—9 may be placed at any cell:

123456789	123456789	123456789	123456789	123456789	123456789	123456789	123456789	123456789
123456789	123456789	123456789	123456789	123456789	123456789	123456789	123456789	123456789
123456789	123456789	123456789	123456789	123456789	123456789	123456789	123456789	123456789
123456789	123456789	123456789	123456789	123456789	123456789	123456789	123456789	123456789
123456789	123456789	123456789	123456789	123456789	123456789	123456789	123456789	123456789
123456789	123456789	123456789	123456789	123456789	123456789	123456789	123456789	123456789
123456789	123456789	123456789	123456789	123456789	123456789	123456789	123456789	123456789
123456789	123456789	123456789	123456789	123456789	123456789	123456789	123456789	123456789
123456789	123456789	123456789	123456789	123456789	123456789	123456789	123456789	123456789

With this representation, when we place a value at a cell, we eliminate it from the other cells in the same row, column, and box (collectively known as the *peers* of a cell). For example, if we place 5 at the top- left corner of the empty board, we can eliminate 5 from the peers of the top-left corner:

5	1234 6789	1234 6789	1234 6789	1234 6789	1234 6789	1234 6789	1234 6789	1234 6789
1234 6789	1234 6789	1234 6789	123456789	123456789	123456789	123456789	123456789	123456789
1234 6789	1234 6789	1234 6789	123456789	123456789	123456789	123456789	123456789	123456789
1234 6789	123456789	123456789	123456789	123456789	123456789	123456789	123456789	123456789
1234 6789	123456789	123456789	123456789	123456789	123456789	123456789	123456789	123456789
1234 6789	123456789	123456789	123456789	123456789	123456789	123456789	123456789	123456789
1234 6789	123456789	123456789	123456789	123456789	123456789	123456789	123456789	123456789
1234 6789	123456789	123456789	123456789	123456789	123456789	123456789	123456789	123456789
1234 6789	123456789	123456789	123456789	123456789	123456789	123456789	123456789	123456789

This procedure will significantly reduce the number of boards that need to be visited.

Programming Task

Your task is to implement the `SolutionLike` and `BoardLike` traits. You can use the template in fig. 27.1 to do so.
We recommend proceeding in this order and testing as you go along:

1. Implement `Solution.peers`. `peers(r, c)` produces the coordinates of all cells in the same row as `r`, same column as `c`, and same block as `(r,c)`.

   Do not include `(r, c)` in the set of its own peers. i.e., the value of the expression `peers(r,c).contains((r, c))` should be `false`.

```scala
import hw.sudoku._

object Solution extends SudokuLike {
  type T = Board

  def parse(str: String): Board = ???

  // You can use a Set instead of a List (or, any Iterable)
  def peers(row: Int, col: Int): List[(Int, Int)] = ???
}

// Top-left corner is (0,0). Bottom-right corner is (8,8). Feel free to
// change the fields of this class.
class Board(val available: Map[(Int, Int), List[Int]]) extends BoardLike[Board] {

  def availableValuesAt(row: Int, col: Int): List[Int] = {
    // Assumes that a missing value means all values are available. Feel
    // free to change this.
    available.getOrElse((row, col), 1.to(9).toList)
  }

  def valueAt(row: Int, col: Int): Option[Int] = ???

  def isSolved(): Boolean = ???

  def isUnsolvable(): Boolean = ???

  def place(row: Int, col: Int, value: Int): Board = {
    require(availableValuesAt(row, col).contains(value))
    ???
  }

  // You can return any Iterable (e.g., Stream)
  def nextStates(): List[Board] = {
    if (isUnsolvable()) {
      List()
    }
    else {
      ???
    }
  }

  def solve(): Option[Board] = ???
}
```

Figure 27.1: Template for Sudoku

2. Implement `Solution.parse` You should assume that the input string has exactly 81 characters, where each character is a digit or a period. Each successively block of nine characters represents a row (i.e., row-major order).

   As the template suggests, you need to store the set of available values at each cell instead of the value at the cell. You'll need to define the empty board as the map with all values available at each cell and implement `parse` using `peers` as a helper function.

3. Implement `Board.valueAt`. You should produce the digit stored at the given row and column or `None` if it is blank.

4. Implement `Board.isSolved` and `Board.isUnsolvable`. You may assume that the constraints represented by `available` are valid. Therefore, a board is solved if every cell is constrained to exactly one value. Similarly, a board is unsolvable if any cell is constrained to the empty set of values (i.e., nothing can be placed in that cell).

5. Implement `Board.place`. The application `place(row, col, value)` produces a new board with `value` placed at `(row, col)` of `this` board. You may assume that the predicate `availableValuesAt(row, col).contains(value)` is true.

   For the new board to be valid, you'll have to:

   (a) Remove `value` from set of available values of each peer (i.e., `peers(row, col)`).

   (b) While doing (1), if a peer is constrained to exactly 1 value, remove that value from its peers.

6. Implement `nextStates`, which returns the list of all boards that have exactly one additional value placed on the board.

   You should sort the returned list: ensure that boards with fewer available values occur earlier in the list.

7. Implement `solve`. If `this.isSolved` is true, then return `Some(this)`. If not, iterate through the list of `nextStates`, applying `solve` to board. Return the first solution that you find. If no solution is found, return `None`.

Solvable Boards

Your solver will not be able to solve arbitrary Sudoku boards. But, here are some boards that should work:

```
val fromCS121_1 =
  "85....4.1......67...21....3..85....7...982...3....15..5....43...37......2.9....58"
val fromCS121_2 =
  ".1.....2..3..9..1656..7...33.7..8..........89....6......6.254..9.5..1..7..3.....2"
val puz1 =
  ".43.8.25.6.............1.949....4.7....6.8....1.2....382.5.............5.34.9.71."
val puz2 =
  "2...8.3...6..7..84.3.5..2.9...1.54.8.........4.27.6...3.1..7.4.72..4..6...4.1...3"
```

There are several sources of Sudoku puzzles on the Web. There are 50 purportedly easy puzzles on Peter Norvig's webpage:
http://norvig.com/easy50.txt
You can use this terminal command to translate them into the format for this assignment:

```
curl -s http://norvig.com/easy50.txt | \
  tr '\n' ' ' | \
  tr '0' '.' | \
  sed 's/ //' | \
  sed 's/ ======== /\'$'\n/g'
```

In addition, we encourage you to share solvable puzzles with each other on Piazza.

Check Your Work

Here is a trivial test suite that simply checks to see if you've defined the `Solution` object with the right type:

```
class TrivialTestSuite extends org.scalatest.FunSuite {

  test("The solution object must be defined") {
    val obj : hw.sudoku.SudokuLike = Solution
  }
}
```

You should place this test suite in **`src/test/scala/TrivialTestSuite.scala`**. This test suite must run as-is, or you may receive no credit for your solution.

Hand In

From the **`sbt`** console, run the command **`submit`**. The command will create a file called **`submission.tar.gz`** in your assignment directory. Submit this file using Moodle.

For example, if the command runs successfully, you will see output similar to this:

```
Created submission.tar.gz. Upload this file to Moodle.
[success] Total time: 0 s, completed Jan 17, 2016 12:55:55 PM
```

Note: The command will not allow you to submit code that does not compile. If your code doesn't compile, you will receive no credit for the assignment.