Homework 5: Generalized Tic Tac Toe

For this assignment, you will write a program that given an $n \times n$ tic-tac-toe board, determines which player will win, or if the game will be a draw. You're going to make significant use of Scala collections and learn the the *Minimax algorithm*, which is a form of *backtracking search*.

1 Preliminaries

You should create a directory-tree that looks like this:

```
./tictactoe
├── 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" % "1.0.0")
```

You can use the code in fig. 17.1 as a template for your solution.

2 Representing a Tic Tac Toe Board

We assume you know how to play Tic Tac Toe. This section talks about the representation of tic-tac-toe boards that you will use. All the types mentioned below are in the `hw.tictactoe` package.

The `sealed trait Player` has two constructors, `O` and `X`, that represent the two players.

A typical 3×3 board can be thought of as a 3×3 matrix, where $(0, 0)$ is the coordinate of the top-left corner and $(2, 2)$ is the coordinate of the bottom-right corner:

```
import hw.tictactoe._

class Game(/* add fields here */) extends GameLike[Game] {
  def isFinished(): Boolean = ???
  /* Assume that isFinished is true */
  def getWinner(): Option[Player] = ???
  def nextBoards(): List[Game] = ???
}

object Solution extends MinimaxLike {
  type T = Game // T is an "abstract type member" of MinimaxLike
  def createGame(turn: Player, dim: Int, board: Map[(Int, Int), Player]): Game = ???
  def minimax(board: Game): Option[Player] = ???
}
```

Figure 17.1: Template for Generalized Tic Tac Toe.

$(0, 0)$	$(1, 0)$	$(2, 0)$
$(0, 1)$	$(1, 1)$	$(2, 1)$
$(0, 2)$	$(1, 2)$	$(2, 2)$

The `Solution.createGame` function, which you need to implement, takes as input the player who makes the first move, the value n that specifies the dimensions of the board, and a map from coordinates to `Player`s that indicates where the pieces are.

For example:

- In generalized tic-tac-toe, either player may make the first move. Therefore, given an empty board:

  We can call the function in two ways:

  ```
  Solution.createGame(O, 3, Map())
  Solution.createGame(X, 3, Map())
  ```

- This board:

X		
		O

  Can be represented as:

  ```
  Solution.createGame(X, 3, Map((0, 0) -> X, (2, 2) -> O))
  ```

  Alternatively, we could have O make the next move.

- This board:

X		
X		O

  Can be represented as:

  ```
  Solution.createGame(X, 3, Map((0, 0) -> X, (0, 2) -> X, (2, 2) -> O))
  ```

  Alternatively, we could have O make the next move.

- This board, which is 4×4:

X			
		O	
			X

  Can be represented as:

  ```
  Solution.createGame(O, 4, Map((0, 0) -> X, (2, 2) -> O, (3, 3) -> X))
  ```

  Alternatively, we could have X start first too.

3 The Minimax Algorithm

Minimax is an algorithm to determine who will win (or draw) a two-player game, if both players are playing perfectly. To do so, Minimax searches all possible game-states that are reachable from a given inital state. Here is an outline of a recursive implementation of Minimax:

```
def minimax(game: Game): Some[Player] = {

  /*
  If it is Xs turn:

    1. If X has won the game, return Some(X).
    2. If the game is a draw, return None. (If all squares are filled
       and nobody has won, then the game is a draw. However, you are
       free to detect a draw earlier, if you wish.)
    3. Recursively apply minimax to all the successor states of game
       - If any recursive call produces X, return Some(X)
       - Or, if any recursive call produces None, return None
       - Or, return Some(O)

  The case for Os turn is similar.
  */

}
```

You can find several other descriptions of Minimax on the Web. But, Minimax is a very straightforward function to write, if you follow the programming directions below and implement (and test) everything leading up to Minimax.

4 Programming Task

Your task is to implement a representation of boards, by implementing the `GameLike` trait, provided in the template code. Your code must be able to implement arbitrary $n \times n$ boards for all $n > 2$. However, your implementation of the Minimax algorithm (the `MinimaxLike` trait) only needs to be fast enough for $n \leq 4$.[1]

I recommend proceeding in the following way, using `Solution.scala` as a template:

1. Add fields to the `Game` class to represent the state of the game and fill in the body of the `Solution.createGame(turn, dim, board)` function. You may assume that `dim >= 2` and that all the pieces described in `board` are within bounds. However, *The board may be in an arbitrary, even illegal state.* For example, the board may have seven Xs. Similarly, the `turn` could be either X or O.

2. Implement the `Game.isFinished` method. This method should produce `true` when there are three Xs or Os in a row or the game ends in a draw.

3. Implement the `getWinner` method.

[1] If you want to do better, lookup *alpha-beta pruning* on the web.

4. Implement the `nextBoards` method, which returns a list of boards that represent all the moves the next player could make.

   For example, if the current board looks like this:

X	X	
	O	
X	O	O

   And if it is O's turn, then these are the three possible next boards:

X	X	
O	O	
X	O	O

X	X	
	O	O
X	O	O

X	X	O
	O	
X	O	O

As you implement each successive step, you may need to revisit design decisisions you made earlier.

5 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.