CISS 445: Programming Languages
Test 01 Programming Part

NAME: ____

There are two parts. Part 1 is close-book and in-class and Part 2 is open-book and take-home. Part 1 and Part 2 are both worth 50% of the midterm grade. No discussion is allowed for either case. Plaigiarism will be dealt with severely. You can only use: your laptop, the class notes, and your brain.

SUBMISSION:
Use alex. Enter t01 for work when submitting using alex.

Part 2

Write an OCAML program that plays the tic-tac-toe game. The function that I should run is main ().

The tic-tac-toe board can (of course) be modeled by a 2-dimensional array of 3x3 strings of length 1. For instance the board

```
O|X|O
-+-+-
X| |O
-+-+-
 |X|
```

can be represented by `[['O';'X';'O'];['X'; ' '; 'O'];[' ';'X';' ']]`. However I leave it up to you if you want to model it in a different way. For instance you can model the tic-tac-toe board as a list of 9 characters: `['O';'X';'O';'X';' ';'O';' ';'X';' ']`.

For this test, when we first run main (), the program waits for an integer value for the size of the board. The standard tic-tac-toe is 3-by-3 (of course), but your program must allow n-by-n where n is the user input.

The human player takes X and and is the first to make a move. Here's an a sample execution:

```
3
-1

 | |
-+-+-
 | |
-+-+-
 | |
row: 0
col: 0
```

As always, I'm using bold to indicate user input. The first input is the board size (which is 3 in this case.) I'll explain the -1 later. The player has chosen (0,0), i.e.row 0, column 0. The program goes on and display:

```
3
-1

 | |
-+-+-
 | |
-+-+-
 | |
row: 0
col: 0

X| |
-+-+-
 |O|
-+-+-
 | |
row:
```

i.e, the program has chosen (1,1) and now waits for a row value from the player. If the user chooses a wrong move, the program waits for another move:

```
3
-1

 | |
-+-+-
 | |
-+-+-
 | |
row: 10
col: 0
row:
```

Here's a win situation:

```
... some output not shown ...
X| |O
-+-+-
X|O|
-+-+-
O| |
row: 2
col: 1

X| |O
-+-+-
X|O|
-+-+-
O|X|
OCAML wins
```

If the player wins, the program prints "you win". Here's a draw situation:

```
... some output not shown ...
O|O|X
-+-+-
X|X|O
-+-+-
O|X|
row: 2
col: 2

O|O|X
-+-+-
X|X|O
-+-+-
O|X|X
it's a draw
```

The intelligence of your `play` function must include at least the following:

- Prevent the human player from making a win whenever possible. (If there are two possible winning positions, the program must try to prevent one).
- Win whenever possible. (If the program has two possible winning positions, then it can take either.)

This implements only a look-ahead of 1 step.

(EXTRA CREDIT: For the game to look ahead to all possible scenarios, you need to do a search tree (or graph). However you may choose to use another data structure. Each node is a tic-tac-toe board.

There are at most 9 possible positions (for 3-by-3) starting with a blank tic-tac-toe board. After the human player has taken the first position, there are in fact only 8. Given a tic-tac-toe board, you need to build a tree to look ahead for all possible scenarios. Note that technically, you don't need to build the whole tree before traversing it. (That's why this is a search problem and not a graph problem). But I leave it to you. The point is that suppose the program is at a particular state (or node) and there are 4 possible moves n1,n2,n3,n4. You should continually build all possible scenarios from these 4 nodes until they all reach a win or a draw. Obviously you want to pick either n1,n2,n3,n4 where all the leaves give the program winning nodes. If this is not possible, then you should pick one that will lead to only wins and draws where the fraction of wins is the largest. And if that's not possible you want to pick one which will give you the largest win to win-draw-lose ratio.)

Now let me explain about the second input of -1 in the first test scenario:

```
3
-1

 | |
-+-+-
 | |
-+-+-
 | |
row:
```

The program waits for a board size, i.e., you can play an n-by-n tic-tac-toe game where n is the first input. The player can also choose to create "holes" in the tic-tac-toe board. In the following the player requests for a 4-by-4 board with two holes, one at (1,2) and another at (3,0). (The -1 is used to terminate the data for "holes".) A "hole" is denoted by '@'. Neither the player nor OCAML can place a piece at a hole.

```
4
1
2
3
0
-1

 | | |
-+-+-+-
 | |@|
-+-+-+-
 | | |
-+-+-+-
@| | |
row: 0
col: 0

X|O| |
-+-+-+-
 | |@|
-+-+-+-
 | | |
-+-+-+-
@| | |
row:
```

(In the above, the player chooses (0,0) and OCAML chooses (0,1).)

The player and OCAML cannot play a piece at a "hole". If a player places his/her piece at a hole, that would be considered invalid – the program must then wait for a new move. Winning moves are still the same – the player (or the OCAML player) wins if the player (or OCAML) creates a complete row or column or diagonal with his/her piece.

That's it!!!