CMSC 15100 **Introduction to Computer Science** **Project 2**
Autumn 2009 **November 16**

Tic-tac-toe

Due: November 23; 10pm

Introduction

Your task in this project is to implement the game tic-tac-toe, including a simple computer player. The game is played on a 3×3 board; the human player goes first and places an "X" on the board, then the computer places an "O", and so on, until either one player gets three in a row (a winning postion) or there are no more moves (a draw).

You will use the *Intermediate Student with Lambda* language and the `world.ss` teachpack for this project. You should follow the steps outlined below and test the code for each step before moving onto the next step.

Step 1 — data structures

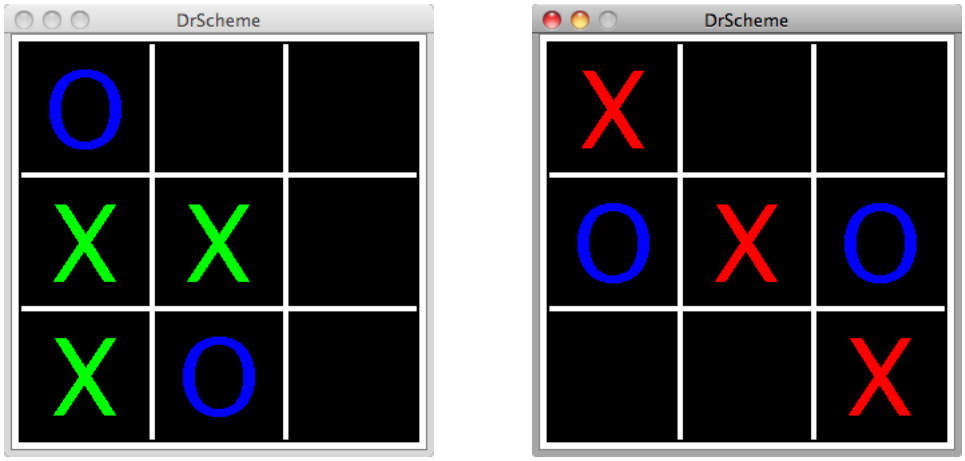
As in Project 1, the playing area (or board) is divided up into square cells. We refer to a cell by its row and column ($\langle r, c \rangle$), with cell $\langle 0, 0 \rangle$ being the upper-left corner of the board and cell $\langle 2, 2 \rangle$ being the lower-right corner. The first step is to define the constants and data structures that are used to represent the state of the game.

```
;; a board consists of the current player, the opponent, and the
;; state of the game, where the state of the game is either 'X or
;; 'O when there is a winner, 'draw when the game is drawn, or
;; 'none when the game is in progress.
(define-struct board (player opponent winner))

;; a player is a name ('X or 'O) and a sorted list of pieces
(define-struct player (name pieces))

;; a cell is a row and column
(define-struct cell (row col))
```

Notice that the board representation uses the order of the players to distinguish which player gets the next move. After each move, the players are swapped.



A game in progress. Player “X” wins!

Figure 1: Screen shots of the game board

Step 2 — testing for a winning position

Notice that representation of a player keeps the pieces in sorted order. You will need to define a total ordering on cells and an insertion function for inserting a cell into an ordered list of cells. The reason for this representation is that it makes testing for a winning position easier. Define a list of the possible winning positions:

```
;; There are eight winning configurations
(define winners ...)
```

Each configuration is a three-element list of cells in sorted order. Then testing for a winning position just requires testing to see if any of the winning configurations is embedded in the list of pieces for the player:[1]

```
;; winner? : (listof cells) -> (union (listof cells) false)
;; is there a winning configuration embedded in the sorted list of pieces?
;; Return the winning configuration, if it exists, otherwise return false.
(define (winner? pieces) ...)
```

Step 3 — rendering the board

Once you have the board data structures defined and tested, you can write the code to render the board as a scene. As before, the position of the center of the cell $\langle r, c\rangle$ will be $\langle wc + \frac{w}{2}, wr + \frac{w}{2}\rangle$, where w is the width in pixels of a cell (we suggest 100 pixels for w). To render the pieces, you should use the `text` function to draw `"X"` and `"O"` strings. A text size of 80 points will work well for 100×100 cells. Note that the pinhole for a text image is its upper-left corner. Figure 1 shows what the board should look like. The lines separating the cells are drawn as rectangles.

[1] We say that list l_1 is *embedded* in list l_2 if the elements of l_1 appear in l_2 in the same order as they are in l_1. For example, `'(a b c)` is embedded in `'(x a b y z c w)`, but not in `'(x a c y b z)`.

Step 4 — updating the board with a move

Once you can render the board, ou need to provide a way to update the board by placing a piece on it. Define the following function for this purpose:

```
;; do-move : board cell -> board
;; update the board with the given move by the current player and
;; swap the players
(define (do-move board cell) ...)
```

This function must also test for winning and drawn positions.

Step 5 — handling player input

The player will use the mouse to make his or her move. You will need to test for `'button-down` events and then convert the mouse coordinates to a cell. Before playing the user's move, you must check that the cell in question is unoccupied. Once the user's move has been played, your program should play the computer's response. To allow different implementations of the computer player, abstract it into a functional argument. This design results in the following interface to the update function:

```
;; update : (board -> board) -> (board number number mouse-event -> board)
;; update the board by first handling the mouse event and then, if necessary,
;; playing the opponent's move
(define (update opponent) ...)
```

Step 6 — running the game

To run the game, you will use the `big-bang` function from the `world.ss` teachpack to define a `run` function.

```
;; run : (board -> board) -> true
;; run the game with the specified amount of food.
(define (run opponent)
  (and
   (big-bang board-wid board-ht 1/20
     (make-board (make-player 'X '()) (make-player 'O '())) 'none)
   (on-redraw render-board)
   (on-mouse-event (update opponent))
   (stop-when (lambda (b) ...))))
```

Note that since there is no animation, this code does not provide a `tick` function. The game should stop when either player wins or the game is a draw.

You can test your code using the `identity` function as the opponent, which will allow you to play both sides.

```
(run identity)
```

Step 7 — a random computer player

The final step of the project is to define a computer player that plays randomly.

```
;; random-player : board -> board
;; Play a random move
(define (random-player b) ...)
```

The random player will randomly pick one of the possible moves. Once this player is defined, you can run the game as

```
(run random-player)
```

Hints and guidelines

Here are some hints and guidelines for the project

- Do not leave this project to the last minute!
- Use library functions. The PLT Scheme system defines many useful functions (especially on lists) that you can use to reduce your coding effort.
- Do not leave "magic numbers" (*i.e.* constants) floating around your code. These should be defined at the top of the file.
- Each step described above corresponds to a logical part of the system that you are implementing. You should organize your code into sections based on this outline, with the main function of each section presented first. Put the test cases for each section last.
- Make sure your code is commented and clear.
- The easiest way to deal with highlighting the winning moves is to redraw those cells after the whole board is rendered.