CMSC 15100 **Introduction to Computer Science** **Project 3**
Autumn 2009 **November 30**

Tic-tac-toe (part 2)

Due: December 7; 10pm

Introduction

In this final project, your task is to implement a "smart' computer player for the tic-tac-toe game that you wrote for Project 2. To implement a smart computer player, we will use the *minimax* algorithm.

You should use the *Advanced Student* 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 — Project 2 fixes

The first step is to address any outstanding issues with your Project 2 code. Add missing test cases and correct any errors in your code.

Step 2 — The minimax player

For this part, you are to define a new computer player that uses the Minimax algorithm to pick the best move.

```
;; minimax-player : board -> board
;; pick the best move for the current player using the minimax
;; algorithm and return the resulting board.
(define (minimax-player board) ...)
```

The basic idea of the Minimax algorithm is as follows. Assume that we have two players: A and B (which is the computer does not matter for this discussion). If it is player A's turn and there is a move that wins the game for A in one move, that move is A's *best* move. Likewise, if player B knows that one move will lead to A's winning the game, whereas another move will lead to a draw, then the move that leads to the draw is B's best move. The minimax algorithm finds the best move by assuming that A is trying to *maximize* his/her chances of winning, while B is trying to *minimize* A's chances of winning.

We can make the notion of minimization and maximization concrete by using integer values to *score* a board position. For 3×3 tic-tac-toe, we will be able to use exhaustive search and so we only need three values for board positions: -1 for a losing position, 0 for a draw, and $+1$ for a winning position. The `minimax-player` function then needs to pick the move that results in a board with

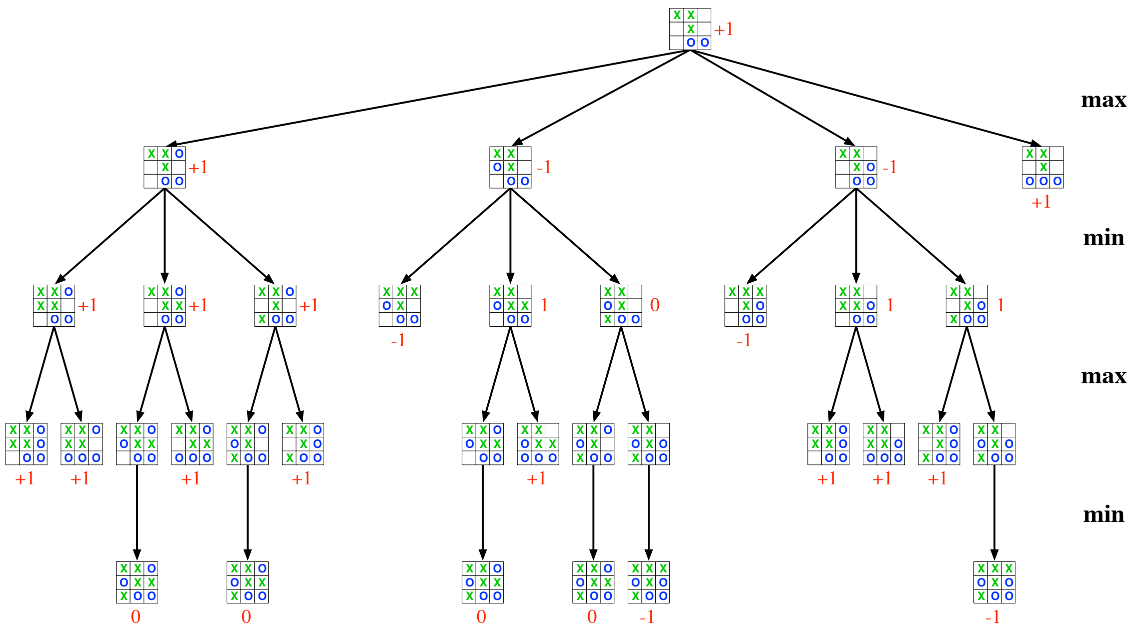

Figure 1: Minimax scoring of a winning board position for O

the maximum score (if there are multiple moves with the best score, then one should be picked at random). To score a board that is not a win or draw, the computer picks the minimum score of any board that it can reach (*i.e.*, it assumes the human player will pick his/her best move). We generalize this approach to a recursive scoring algorithm that alternatively maximizes and minimizes the board score at each level. Figure 1 shows how the scores propagate up from the terminal board positions. The fact that the root has a score of $+1$ means that it is a winning position for the O player. In this example, the O player has two winning moves: the rightmost wins immediately, while the leftmost leads to a win in O's following turn.

You will probably find it best to break this part of the code into two pieces: the minimax algorithm that computes a score for a board and the `minimax-player` function that picks the best move from a given board.

Step 3 — User-interface changes

In the early game, the minimax search can take several seconds to complete. Since the UI code from Project 2 plays both the human and computer moves (in the `update` callback function), there can be a significant delay from when the user clicks a move to when the move is displayed. To address this problem, you will need to rework the user interface.

This new interface will move the computation of the computer's response from the mouse-event callback to a tick callback. We also want to ensure that the game is redrawn in response to the player's move before we compute the computer's response. Therefore, you will need to include additional state in the world representation to communicate information between callback functions. Your interface will also display informative messages just below the board.

Step 3a — New data structures

You will define a new data structure to represent the game state.

```
;; a world consists of a board, a game state, and a message to display
(define-struct world (board gstate msg))

;; a game state is used to determine how the callbacks should work.
;; The possible states are
;;    'X    -- waiting for the player
;;    'O    -- waiting for the computer
;;    'draw -- waiting for a redraw
;;    'done -- game over
```

The game state is used to pass information between the callbacks. Your code will handle the game state according to the state diagram in Figure 2. Note that the "*Waiting for redraw*" state and `on-redraw` callback appear twice in the diagram. In these cases, the current player determines the next game state (*e.g.*, if the player is `'X`, then the next state after the `on-redraw` callback will be "*Waiting for user input*").

Step 3b — Render the game world

In addition to displaying the board, you will add a message area at the bottom, where you will display the current message. The messages might include

- "`It's your turn`" when waiting for the player's move,
- "`I'm thinking`" when the computer is calculating its move, and
- "`You win!`" when the human wins.

Your existing `render-board` function is largely unchanged, but you will need to define a new function for rendering the world.

```
;; render-world : world -> scene
;; render the board and current message as a scene
(define (render-world w) ...)
```

You may observe that the `render-world` function returns a scene, but is also responsible for changing the game state from `'draw` to `'X` or `'O`. Thus, you will need to use side effects in your `render-world` function to update the game state.[1]

You can test your `render-world` function using the `random-player` from Project 2 first, since the `minimax-player` will be less responsive.

Step 3c — Handling user input

You will have to modify the update function from Project 2 to match its new contract.

[1]Note that the `render-world` function is the only place where you need side effects, but you may use them to update the world representation elsewhere. You should *not* use side effects outside on any structures other than the world representation.

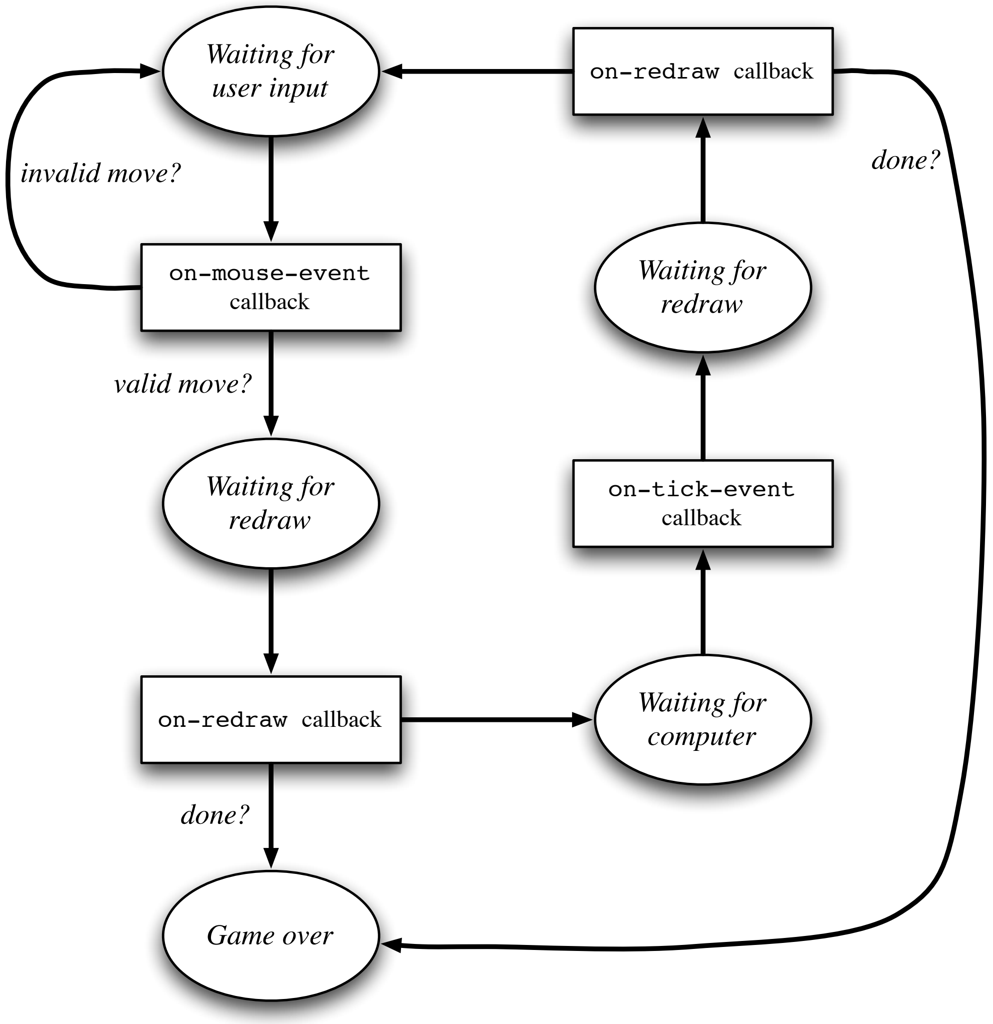

Figure 2: The game states and transitions. The ellipses represent logical states, while the rectangles represent callback functions.

```
;; update : world number number mouse-event -> world
;; handle mouse input by checking for a possible player move and
;; executing the move if it is valid.
(define (update world x y evt) ...)
```

Because the message is stored separate from the game state, you can change it in response to invalid user input. For example, if the user clicks outside the board area, then you could change the message to

```
Invalid move:  try again
```

Step 3d — Running the computer player

The computer player will be run from a `on-tick-event` callback function when the game state is `'O`. Define a higher-order function that takes an player function and returns a `on-tick-event` callback.

```
;; tick : (board -> board) -> (world -> world)
;; define a callback function that runs the opponent function when the
;; game state is 'O
(define (tick opponent) ...)
```

Step 3e — Putting it all together

You will also need a new version of the `run` function that uses the functions defined above.

```
;; run : (board -> board) -> true
;; run the game with the specified computer player
(define (run opponent) ...)
```

Your initial world should be something like the following:

```
(define start-world
  (make-world
   (make-board (make-player 'X '()) (make-player 'O '()) 'none)
   'X
   "It's your turn"))
```

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.
- Make sure your code is commented and clear. As in the previous projects, it should be organized into logical sections with the main function of each section presented first and the test cases at the end of the section. Also, avoid excessively long lines of code; use newlines and indentation to make the structure of your code clear.
- Note that you can code the Minimax search using a single recursive function by exploiting the fact that $\max(a, b) = -\min(-a, -b)$.