

## Game lab

The objective for this Laboratory is to show you how to implement a game in lisp. I have most of the game functions included in this exercise, but you will need more for the game of ACHI (see the next assignment). This lab will show you how to input the first set of core functions and test them. At the end of this lab, you will be responsible for implementing two functions for testing a win position **on the diagonal right and diagonal left**.

**At the end of the lab, turn in checks for winning the game diagonally right, diagonally left.**

The core game functions are:

- (*print-board board*) - where board is the name of the board
- (*movegen board player*) – which generates possible moves for a player, given a particular board
- (*opposite player*) – given ‘x’ it will give you ‘o’
- (*player-moves player board position*) – actually moves the player to a particular position
- (*won? board player*) – returns a true or false if a particular player wins

First, we will need to make some functions that help us set-up the game. I have set up my game as an array. So I need to.....

- a. Write a function that makes an array
- b. Write a function that prints a board

The function to create an array is easy....You’ll set up an array (or vector) and initialize it to strings of nothing “<space>” The make-array function is a built in function that creates an array. The game has 9 spaces, so your array will need 9 elements (which really means 0 – 8).

```
(defun make-board ()  
  (make-array '(3 3) :initial-element " ")); this has a space between the quotes
```

The print board is a little more complicated. It is going to require the use of the *format* built-in function. The *format* function is: (*format default <print information>*) It operates similar to the *printf* function in c++. The special character ~% is used to indicate a line feed. The actual thing that is printed is surrounded by quotes. So, the print function for my board is: (WATCH YOUR SPACES! – this looks ugly in word)

```

(defun print-board (board)
  (format t "~% 1 2 3 ")
  (format t "~% +---+---+---+")
  (dotimes (row 3)
    (progn
      (format t "~%| | | |~%|")
      (dotimes (column 3)
        (progn
          (format t " ~a |" (aref board (- 2 row) column))))
        (format t "~%| | | |")
        (format t "~% +---+---+---+")))))

```

This may look a little strange, but it's the best I can do without a lot of neat graphic stuff.

So now you can create your board by calling the `make-board` function like the following (`setf my-board (make-board)`).

Once you do that, you can print the board by typing (`print-board my-board`).

You will need to add a copy-array function for the `movegen` function. You need this because you want to generate all the possible moves from a given position...and still keep the original board. This is easy, but I'll just give you the code.

```

(defun copy-array (array)
  (let ((b (make-board)))
    (dotimes (r 3)
      (dotimes (c 3)
        (setf (aref b r c) (aref array r c))))
    b))

```

The next function is designed to move the player to an actual position. The function needs arguments for player, board, and location. That is, move a player on a board to a specific location...

```

;;; returns a list of all moves (next board states) for player from
;;; board

```

```

(defun player-moves (player board row column)
  (let ((b (copy-array board)))
    (setf (aref b row column) player)
    b))

```

Test your function...(*player-moves "x" my-board 0 2*) ; remember that the array starts at 0.

This board is not permanent. (Which is what you want when you play the game and generate the moves. Test this by typing (*print-board my-board*) ) So, if you want to make it permanent while you are testing the next function, then you need to do the following:

```
(setf new-board (player-moves "x" my-board 0 2))
```

Then you can (*print-board my-board*) and see the new board that is created.

The next function generates ALL the player's possible moves. It takes a board and a player and generates all the possible '**next moves**' for that player.....The *find nil function* finds all the empty spaces on the board and passes them to *movegen*. *movegen then creates a list where it puts the player into ONE of those empty spaces*. You are going to need more move

```
(defun movegen (board player)
  (mapcan
   #'(lambda (m)
       (list (player-moves player board (car m) (car (cdr m))))))
   (find-nil board)))
```

```
(defun find-nil (board)
  (let ((emptycells nil))
    (dotimes (r 3)
      (dotimes (c 3)
        (if (string-equal (aref board r c) " ")
            (setf emptycells (cons (list r c) emptycells))))
      emptycells)))
```

First, test the find-nil function by typing (*find-nil my-board*). It should return a list of lists of all the empty positions. Now test the movegen function – and notice the differences.... (*movegen my-board "o"*) . This should return a list of boards for "o" given the "x" board.

We also need a function that will allow us to alternate between the players – *other-player*. *Other-player* works by passing one player into the function, and getting the other one back. We need this function to generate the WHOLE GAME TREE...So we need to alternate giving movegen "x" and "o".

```

(defun other-player (player)
  (if (string-equal player "x")
      "o"
      "x"))

```

You can test this function by calling it with one player – (*other-player "x"*) It should return “o”....etc.

We now need a *won* function. There are four ways to win in Achi, just like tic-tac-toe. You win 3 in a row, 3 in a column, 3 diagonal (both ways). So the won function takes a board and a player and checks for all possible wins.

```

(defun won? (board player)
  (or (horizontal board player)
      (vertical-line board player)
      (diagonal-right board player)
      (diagonal-left board player)))

```

Here are the two horizontal and vertical wins

```

(defun horizontal (board player)
  (let ((truth nil))
    (dotimes (i 3)
      (let ((j 0))
        (cond ((and (string-equal player (aref board i j))
                    (string-equal player (aref board i (+ j 1)))
                    (string-equal player (aref board i (+ j 2))))
              (setf truth t))))))
    truth))

```

```

(defun vertical (board player)
  (let ((truth nil))
    (dotimes (j 3)
      (setq i 0)
      (cond ((and (string-equal player (aref board i j))
                  (string-equal player (aref board (+ i 1) j))
                  (string-equal player (aref board (+ i 2) j)))
            (setf truth t))))
    truth))

```

You can test these two functions, but you will need to set up some “test” boards. I would create a few boards using the (setf test-board (make-board)) (setf test2-board (make-board)). Now you can populate those boards using the code that is similar below....

This produces a board that has 3 “x” in a row.

```
(defun test (board)
  (progn
    (setf board (player-moves "x" board 0 2))
    (setf board (player-moves "x" board 0 1))
    (setf board (player-moves "x" board 0 0))) board)
```

TEST

```
(setf test-board (test test-board))
```

Now you can test the horizontal function ...(*horizontal test-board* “x”). Now do the same for the vertical function, and finally the won function...

Now right the diagonal left, diagonal right win functions and hand them into the “*achilab*” project.