You are given the LISP code for a program that plays the game OWARI (oh-WAH-ree). Owari is actually a family of stone-and-pit games with ancient African origins, but we will look at one particular such game; the rules are described below. Your task is to add the routine that chooses a move for the computer, using the MINIMAX algorithm described in class.

1. Owari: the rules of the game

Each player has six pits lined up horizontally in front of him or her, and a "goal" pit on the side to the right. When you play against the computer, the board will be represented as a picture:

```
   3 3 3 3 3 3
  0 0 0 0 0 0
   3 3 3 3 3 3
```

with your pits at the bottom, your goal pit at the right, and the computer's pits at the top and goal pit at the left. The picture above shows the start position: each pit has 3 stones in it, except for the goal pits which are empty.

To make a move, a player chooses one of the six pits on his or her side of the board (the chosen pit must have stones in it) and redistributes (or "sows") the stones one-by-one going counter-clockwise around the board, starting with the pit following the one picked. The opponent's goal pit, if reached, is skipped. For the purposes of this implementation, the pits on the player's side are numbered 0 to 5, with 6 being the player's goal pit, and the computer's pits are numbered 7 to 12 from right to left (i.e. continuing counter-clockwise) with the computer's goal pit number 13. If on your first turn you choose to move from pit number 4, the resulting position would be:

```
   3 3 3 3 4
  0 0 1 0 0
   3 3 3 0 4
```

Capturing: If the last stone of a player's move falls into an empty pit on the moving player's side of the board, then any stones in the pit opposite to it are captured and placed in the moving player's goal pit. For instance, if it were your turn in the position shown above, you could choose to move from pit number 1 (remember that the leftmost pit on your side is number 0) and the resulting position would be:

```
   3 3 3 4
  0 0 1 0
   3 0 4 1 4
```

in which the stones from pit number 8 have been captured and placed in your goal pit.
Ending the game: When either player empties all six pits on his or her side of the board, the game is over. The other player takes all the remaining stones from his or her own side, and places them in his or her goal pit. Players count the stones in their goal pits. The player with the most stones is the winner.

2. What we give you

In the file ~lib220/asst4/owari.l is the LISP code for managing a game of owari between a human player (herein called the player), and a computer opponent (the computer). Read through the lisp code, which is appended to the back of this document, for more information on each of the included functions.

The top level function, OWARI, is called with no arguments, and manages the game by asking repeatedly for moves from the player and computer, and printing the board between moves. The move from the player is entered from the terminal as a number from 0 to 5. In the file we've supplied, the routine for the computer's move has been replaced with a routine that lets you enter a move directly from the terminal; this is so you can play some games against yourself to learn how the game works. (Your task will be to write the function that determines the computer's move. See below.)

The complete state of the game at any time is described by the BOARD (which in this implementation is simply a list of 14 numbers, each of which represents how many stones are in one of the 14 holes) along with knowledge of whose turn it is. See the program for more information on the representation issues.

The other supplied function you are likely to use (from within your solution to this assignment) is %DO-MOVE. This function takes three arguments: an atom specifying whose turn it is (either 'player or 'computer), a number specifying the hole to move out of, and an initial board position. It returns a new board position, the result of applying the given move on behalf of the given player. Since %DO-MOVE does not change the actual board position stored by the system, it can be used by your routines to test new board positions as part of the minimax algorithm.

There are also a few utility functions and constants you might want to use; these are listed and documented in the provided source code. You are free to call any function you like, except the top-level function itself, from inside your routines; none of the calls will interfere with the proper running of the game since there are no changing global variables to worry about. (There are some global constants, documented in the program.)

A programming note: all of the constants and routines in the supplied file (except for the top-level function OWARI and the GET-MOVE function you must rewrite) begin with the % character. Thus you needn't worry about the names you give your functions, since as long as they don't start with a % there will be no naming conflict.

3. What you must write

You will rewrite the routine "GET-MOVE". It will be called from within the supplied routines, as it is now, and it will be passed one argument -- the current board position. Your version of GET-MOVE must return the number of the hole from which the computer must move.
(The list of potentially legal moves for the computer is in the global constant %C-HOLE-NUMS. This list does not change based on board position; if a hole has no stones in it then a move from that hole is not legal at that time, and your algorithms must check for that.)

GET-MOVE and the associated functions you write must be an implementation of the minimax routine. A complete solution will also include alpha-beta pruning. You will need to write a set of functions to properly implement this routine, but GET-MOVE must be the top-level function and must work as stated above. If you take advantage of LISP's recursive nature you may find that this problem is very simple to program. A single recursive function, or a pair of mutually recursive functions, can perform nearly all of the work of maintaining the search tree for you with no actual tree data structure required.

You will need to write a function which "expands" a node, but notice that all you need to do is check which moves are legal and then run %DO-MOVE on each of them, passing in the current player (which will be different at each level of the tree), the hole number of the legal move, and the board position of the node you are expanding. (Don't send %DO-MOVE an illegal move! It won't check, and anything might happen!)

Since minimax relies on a heuristic evaluator to avoid searching all the way to the end of the game, you will also need to write such an evaluator. Unless you find yourself enchanted by the prospect, you needn't spend too much time worrying about what would make the best evaluator. Come up with a simple one at first, and if you have time, then think about how it could be improved. Just make sure that it bears some relation to the rules of the game, and, most importantly, that it evaluates to a very large positive number (outside your normal range) when you reach a winning position, and to a very large negative number for a losing position.

You must decide how deep your program will search, but this decision should be encapsulated in a global constant that can easily be changed. You should set it to search at least 3-ply deep, but feel free to set it higher if the program speed is reasonable.

4. What to hand in

(a) The file containing all the functions you have written to solve the assignment. (Electronically using "submit", and on paper)

(b) A runtime session showing yourself playing one complete game against the computer opponent you've created. (Don't be surprised if you lose to your creation.) (On paper)

(c) Another runtime session with (TRACE) turned on for your major routines and your static evaluator. Just run this for two moves of your game, then quit out of the game (in Unix, hit Control-C a few times until you are into the debugger, then you can use (QUIT). (On paper)

A basic minimax routine without alpha-beta pruning can receive a maximum grade of 93% (A-). To receive a 94% to 100% (A), your solution must include alpha-beta pruning.
5. Optional extension activities

(a) 5%. Compare the search efficiency of standard minimax search with alpha-beta pruning by keeping track (in a global variable, perhaps?) of the total number of nodes expanded by each algorithm during a game (be sure to play the same moves!) and write up your results. Is the gain bigger towards the beginning of the game, or in the middle, or near the end? Or is it the same throughout?

(b) 5%. Experiment with more advanced heuristic algorithms. Rewrite the human-player function to accept moves from a second computer opponent, so that you can pit the computer against itself using alternative heuristics to see which ones appear to be the best.

(c) 5%. Experiment with a non-constant depth-of-search. For instance, you could keep track of the number of nodes searched instead of the total depth, allowing your program to go deeper when the search tree has fewer branches (typically towards the end of the game).
OWARI is played on a board with 14 holes arranged as two rows of 6
with two holes in the middle on the ends of the rows. Complete rules
of the game are in the accompanying assignment document.

The state of the game is very simple to describe. All you need
to keep track of are the number of stones in each of the fourteen
holes (and whose turn it is, but that can be done automatically).
Let's represent the board as a list of 14 numbers. The first
six will be the player's holes, the seventh will be the
player's goal hole, the next six will be the computer's holes
(counting counter-clockwise around the board) and the 14th will
be the computer's goal hole. So the list
'(1 3 0 2 2 6 4 2 1 0 8 1 11)
would represent the board position:

1 8 0 1 2 4
11 6
1 3 0 2 2 2

For the purposes of this program, holes are numbered from 0 to 13,
beginning with the player's first hole and going counterclockwise.
So above, there is 1 stone in hole number 0, 3 in hole number 1,
4 in number 7, 11 in number 13, etc.

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; Function: OWARI
; Takes no arguments. OWARI is the top-level call. It sets the
; board to the initial position, and then iteratively prints it,
; asks the player for a move, handles the player's move, asks the
; computer for a move, handles the computer's move, and repeats.
; It also checks for the end-of-game condition after every move
; by each side, and prints final statistics if the end is reached.
; It is at this top level ONLY that the real board is maintained.
; Although all functions manipulate "boards", unless they return
; their values to this top-level function, the board will not change,
; and this is as it should be. Thus, BOARD is not a global variable.

(defun owari ()
  (let ((BOARD '(3 3 3 3 3 0 3 3 3 3 3 0)))
    (loop
     (%print-board BOARD)
     (setq BOARD (%player-turn BOARD))
     (if (%end-position BOARD) (return))
     (%print-board BOARD)
     (setq BOARD (%computer-turn BOARD))
     (if (%end-position BOARD) (return)))
    (%print-final-board BOARD)))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; A set of utility functions and constants (global variables) used to
; make the rest of the code more readable and understandable. Names
; beginning with P refer to the human player; C refers to the computer.

(setq %p-goal-num 6)
(setq %c-goal-num 13)
;;;; The values in the goal holes
(defun %p-goal-val (board) (nth %p-goal-num board))
(defun %c-goal-val (board) (nth %c-goal-num board))

;;;; The numbers of the other holes
(setq %p-hole-nums '(0 1 2 3 4 5))
(setq %c-hole-nums '(7 8 9 10 11 12))

;;;; The values in the other holes
(defun %p-hole-vals (board)
  (%extract %p-hole-nums board))
(defun %c-hole-vals (board)
  (%extract %c-hole-nums board))

;;;; %EXTRACT takes a list of position numbers, and a list, and returns
;;;; the elements of the list which occur at each of the positions.
(defun %extract (positions list)
  (cond
   ((null positions) nil)
   (t (cons (nth (car positions) list)
            (%extract (cdr positions) list)))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;; %END-POSITION takes a BOARD, and returns NIL if the game is not over
;;;; (that is, if both players still have stones on their side), T otherwise.
;;;; Since COND returns nil if none of the conditions is met, the final
;;;; line of this function, (t nil), is not strictly necessary. However,
;;;; some people include this line to make it clear that the function is
;;;; in fact supposed on occasion to reach that line.
;;;;
;;;; Programming note: Look up the function APPLY in the textbooks. It
;;;; can be a very useful tool. Make sure you understand how this function
;;;; works. Notice the trick: since none of the board positions can have
;;;; a negative number of stones, then as long as the sum of the values is 0,
;;;; all the individual values must be zero as well.
(defun %end-position (board)
  (cond
   ((or (equal 0 (apply '+ (%p-hole-vals board)))
        (equal 0 (apply '+ (%c-hole-vals board))))
    t)
   (t nil)))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;; %PRINT-FINAL-BOARD prints the board, calculates the final score,
;;;; and prints that as well. It assumes that checking has already
;;;; been done to ensure that the game really is over. This function
;;;; returns the atom 'PLAYER or 'COMPUTER, according to whether the
;;;; player or the computer won, or 'TIE if the game was a tie.
(defun %print-final-board (board)
  (let ((p-score 0) (c-score 0))
    (terpri)
    (princ "The game is over")
    (terpri)
    (setq p-score (+ (%p-goal-val board)
        (apply '+ (%p-hole-vals board))))
    (setq c-score (+ (%c-goal-val board)
        (apply '+ (%c-hole-vals board))))
    (princ "Final score: ")
    (princ "Player: ")
    (princ p-score)
    (princ " Computer: ")
    (princ c-score)
    (terpri)
    (cond
      ((equal p-score c-score) (princ "It's a tie!") 'TIE)
      ((> p-score c-score) (princ "You have won!") 'PLAYER)
      ((< p-score c-score) (princ "The computer has won!") 'COMPUTER)))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Function: %PRINT-BOARD
;;; Notice how dolist is used. This function does not make best use
;;; of the game-specific utility functions.

(defun %print-board (BOARD)
  (terpri)
  (princ "     ")
  (dolist (index (reverse %c-hole-nums))
    (princ (nth index BOARD))
    (princ "  ")
  )
  (terpri)
  (princ (%c-goal-val BOARD))
  (princ "                  ")
  (princ (%p-goal-val BOARD))
  (terpri)
  (princ "  ")
  (dolist (index %p-hole-nums)
    (princ (nth index BOARD))
    (princ "  "))
  (terpri))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; %PLAYER-TURN takes a board, asks the player for a move, verifies
;;; that the move is legal, and returns a new board by calling %do-move
;;; with the given board and the move entered from the terminal.

(defun %player-turn (BOARD)
  (let ((hole-num))
    (loop
      (princ "Move stones from which hole (0 - 5)? ")
      (setq hole-num (read))
      (if (and (member hole-num %p-hole-nums)
            (< 0 (nth hole-num BOARD)))
        (return (%do-move 'player hole-num BOARD))
        (princ "Illegal move! ")')))
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;; Function: %DO-MOVE
;;; Arguments:
;;;    who: an atom, either 'player or 'computer, indicating whose move
;;;    it is (for the purpose of deciding which holes to skip and
;;;    whether the move ended on the current players' side.
;;;    hole-num: the number of the hole from which to move;
;;;    %DO-MOVE assumes error-checking has already been done.
;;;    BOARD: the old board position.
;;; Returns: a new board, which is the board resulting from moving out
;;;    of hole-num on the old board.
;;; Notice that when it calls %do-move-aux, it passes it the board with
;;;    the stones already removed from the hole we start at.

(defun %do-move (who hole-num BOARD)
  (%do-move-aux who
    hole-num
    (nth hole-num BOARD)
    (%remove-stones hole-num BOARD)))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;; %DO-MOVE-AUX is the auxiliary function for %DO-MOVE. It is given:
;;; whose turn it is to play, the hole number we're about to drop
;;; a stone into, how many stones are still in the player's hand, and
;;; the BOARD as it looks so far. When there are no more stones left
;;; in the player's hand, %DO-MOVE-AUX determines whether any stones
;;; have been captured, and finally returns the new BOARD.
;;; Programming note: the formal parameter hole-num is manipulated and
;;; changed (using setq) throughout this program.
;;; Style note: This function is too long to leave with just a header
;;; comment, so I should probably break it into smaller functions. I
;;; have chosen instead to use in-line comments.

(defun %do-move-aux (who hole-num how-many BOARD)
  (cond
    ((equal 0 how-many) ;check if our hand is empty
      (%do-capture who hole-num BOARD)) ;if so, end by calling %do-capture
    (t ;we still have stones, so
      (setq hole-num (1+ hole-num)) ;move to the next hole
      (if (or (and (equal who 'player) ;if we're over the
                  (equal hole-num %c-goal-num)) ;opponent's goal hole,
           (equal who 'computer) ;...
           (equal hole-num %p-goal-num)) ;...
        (setq hole-num (1+ hole-num))) ;skip it
      (if (>= hole-num 14) ;if we've gone all the way around
        (setq hole-num (- hole-num 14))) ;the circle, go back to the start
      (%do-move-aux who ;call this function recursively
        hole-num ;with the new hole number we're over
        (1- how-many) ;and with one fewer stone, which
        (%add-stones hole-num 1 BOARD)))) ;we put in this hole
;; %DO-CAPTURE is called by %DO-MOVE-AUX at the point when
;; all stones have been distributed. At that point, if there is
;; exactly one stone in the last hole, and if that hole is on the
;; current player's side of the board, then any stones on the other
;; side are captured. The arguments are: whose turn it is, the
;; hole number that player finished at, and a board. It returns a
;; new board reflecting the results of the capture, if there is one,
;; or else simply returns the old board.

(defun %do-capture (who hole-num board)
  (cond
    ((and (equal who 'player)
           (member hole-num %p-hole-nums)
           (equal 1 (nth hole-num board)))
     (%add-stones %p-goal-num
      (nth (%opposite hole-num) board)
      (%remove-stones (%opposite hole-num) board))
    ((and (equal who 'computer)
           (member hole-num %c-hole-nums)
           (equal 1 (nth hole-num board)))
     (%add-stones %c-goal-num
      (nth (%opposite hole-num) board)
      (%remove-stones (%opposite hole-num) board))
    (t board)))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;; %OPPOSITE is a utility function which returns the number of the
;;;; hole %opposite the one it is given in its argument.

(defun %opposite (hole-num)
  (- 12 hole-num)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;; %REMOVE-STONES takes a hole number and a BOARD, and returns a
;;;; new board with all the stones removed from the given hole number.

(defun %remove-stones (hole-num BOARD)
  (cond
    ((null BOARD) nil)
    ((equal hole-num 0) (cons 0 (cdr BOARD)))
    (t (cons (car BOARD) (%remove-stones (1- hole-num) (cdr BOARD)))))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;; %ADD-STONES takes a hole number, a number of stones to add, and a
;;;; BOARD, and returns a new board with the stones added to the given
;;;; hole number. Its algorithm is similar to that of %REMOVE-STONES

(defun %add-stones (hole-num how-many BOARD)
  (cond
    ((null BOARD) nil)
    ((equal hole-num 0) (cons (+ how-many (car BOARD)) (cdr BOARD)))
    (t (cons (car BOARD) (%add-stones (1- hole-num) how-many (cdr BOARD)))))))
%COMPUTER-TERM: Asks the computer for its move by calling "GET-MOVE".
GET-MOVE is the top-level function you need to write! As a place-
holder, a function GET-MOVE is included that asks for terminal input,
but you need to replace that function with one of your own. (You will
probably write additional utility functions as well, but they will
not interact directly with the game.)
%COMPUTER-TERM checks to see if the move is legal, but if it isn't then
your function is doing something wrong, since computers should never
make illegal moves. A message will be printed that the move was illegal,
and the computer loses its turn. Of course, this should not happen.

(defun %computer-turn (BOARD)
  (let ((hole-num (get-move board)))
    (cond
      ((and (member hole-num %c-hole-noms)
            (< 0 (nth hole-num BOARD)))
        (%do-move 'computer hole-num BOARD))
      (t (princ "ERROR! The computer tried to move from ")
        (princ hole-num)
        (terpri)
        (princ (princ "ERROR! The computer tried to move from ")
                (princ hole-num)
                (terpri)
        BOARD)))
)

(defun get-move (board)
  (princ "Move stones from which hole (7 - 12)? ")
  (read))