```
Connect-5 Implementation
Ian Varley
ian@ianvarley.com

Wed. night lab

This implementation uses matrices as the internal representation of playing boards.
The general strategy is:
 - A list of all candidate boards is created
 - Each board is given a score
 - The board with the highest score wins

The scoring process is described in detail below.
```

```
(define ME 'X)
(define YOU 'O)
(define BLANK '-)
```

```
Matrix functionality

The following functions all provide functionality for the matrix type, such as
creation, indexers, etc. It is modeled after the built-in Vector type, with
several additions such as matrix-apply.
```

```
;; build-matrix : number number (number number --> any) --> matrix
;; build a matrix of given size, using the provided function to construct each spot
(define (build-matrix rows cols fn)
  ;; build the outer vector with a function that builds the inner ones
  (build-vector
   rows
   (lambda (i)
     (build-vector
      cols
      (lambda (j) (fn i j))
      )
     )
   )
  )

;; matrix-ref : matrix number number -> any
;; reference into a matrix
(define (matrix-ref mat row col)
  (vector-ref (vector-ref mat row) col))

;; matrix-set! : matrix number number any --> void
;; set a particular cell in the matrix to a value
(define (matrix-set! mat row col item)
  (vector-set! (vector-ref mat row) col item))

;; copy-matrix : matrix -> matrix
;; returns a copy of the given matrix
(define (copy-matrix mat)
  (build-matrix
   (matrix-rows mat)
   (matrix-cols mat)
   ;; function that simply returns the item from the original matrix
   (lambda (i j) (matrix-ref mat i j))))

;; matrix-copy-set : matrix row col item --> matrix
;; deep copies the given matrix, sets the item in the copy, and returns the copy
(define (matrix-copy-set mat row col item)
  (local
      {
       (define result (copy-matrix mat))
       }
    (begin
      (matrix-set! result row col item)
      result
      )
    )
  )


;; matrix-rows : matrix --> number
;; return number of rows in this matrix
;; (since this is implemented as a matrix of matrices, we simply return the length
;; of the enclosing matrix as "rows)
(define (matrix-rows mat)
  (vector-length mat))

;; matrix-cols : matrix --> number
;; returns the number of cols in the matrix
(define (matrix-cols mat)
  (vector-length (vector-ref mat 0)))


;; List->matrix : (listof (listof item)) --> matrix
;; return a matrix representation of this list
(define (List->matrix a-list)
  (build-matrix
   (length a-list) ;; rows
   (length (first a-list)) ;; cols
   ;; retrieval function for items in the list format
   (lambda (row col)
     ;; recur until row is zero; then until col is zero
     (local {(define (findRow a-list row)
               (cond
                 [(empty? a-list) (error "Matrix index out of bounds!")]
                 [(zero? row) (findCol (first a-list) col)]
                 [else (findRow (rest a-list) (sub1 row))])
               )
             (define (findCol a-list col)
               (cond
                 [(empty? a-list) (error "Matrix index out of bounds!")]
                 [(zero? col) (first a-list)]
                 [else (findCol (rest a-list) (sub1 col))]
                 )
               )
             }
       (findRow a-list row)
       )
     )
   )
  )

;; Matrix->List : matrix -> list
;; change a matrix to list representation
(define (Matrix->List mat)
  (apply-matrix
   mat
   (lambda (item row col) (matrix-ref mat row col))
   true
   )
  )

;; pivot-matrix : matrix -> matrix
;; returns a copy of the matrix rotated 90 degrees clockwise
;; NOTE: this only works on a square matrix
(define (pivot-matrix mat)
  (List->matrix
   (apply-matrix
    mat
    (lambda (item row col) (matrix-ref mat col row)) ; reverse references
    true
    )
   )
  )

;; apply-matrix : matrix (number number --> any) bool --> (listof any)
;; applies the given function (which takes to every item in the given matrix, returning
;; a list of each result)
(define (apply-matrix mat fn hierarchical?)
  (local
      {
       (define joinfn (cond [hierarchical? cons] [else append]))
       (define (apply-matrix-rows mat fn row)
         (cond
           [(>= row (vector-length mat)) empty]
           [else
            (joinfn
             (apply-matrix-cols (vector-ref mat row) fn row 0)
             (apply-matrix-rows mat fn (add1 row)))]
           )
         )
       (define (apply-matrix-cols vec fn row col)
         (cond
           [(>= col (vector-length vec)) empty]
           [else
            (cons
             (fn (vector-ref vec col) row col)
             (apply-matrix-cols vec fn row (add1 col)))]
           )
         )
       }
    (apply-matrix-rows mat fn 0)
    )
  )
```

This section contains program logic specific to working with game boards

```
;; a Board is:
(define-struct Board (matrix score))
;; matrix is a matrix representation of the squares on the board
;; score is a number representing the score of this board

;; possible-moves: board --> (listof board)
;; return a list containing one board for each possible move from this board
(define (possible-moves a-board)
  (create-boards
   (filter
    (lambda (x) (not (empty? x)))
    (apply-matrix
     a-board
     (lambda (item row col)
       (cond
         [(equal? item BLANK)
          (matrix-copy-set a-board row col ME)]
         [else empty]
         )
       )
     false ; we want a flat list, not a hierarchical one
     )
    )
   )
  )
```

```
A NOTE ABOUT SCORING IMPLEMENTATION / STRATEGY:

In order to determine the "score" of a given board, a couple simple rules are
followed:
1. Each row is given a score on its own merit, either positive (desireable) or
   negative (undesireable).
2. The scores for all the rows (including the columns, by way of a pivoted version
   of the original matrix) are added together to get a final score.
3. Some rows are so important that they are automatically desireable (or to be
   avoided at all costs). These are worth more points by an order of magnitude,
   so that regardless of the other costs, they'll win out.

A more general implementation of this program should allow for other goals, such as
connect-6, connect-7 ... connect-n. In the interest of efficiency and simplicity,
however, this program contains only patterns for connect-5. In order to change this
behavior for different goals, all that need happen is the substitution of another
set of patterns.
```

```
(define WIN 9999)
;; LOSE is an order of magnitude smaller than win because win should always take
;; precedence (i.e. it's more important to sieze a win than avert a loss)
(define LOSE -999)
(define VERYGOOD 10)
(define GOOD 5)
(define OK 2)
(define EH -2)
(define BAD -5)
(define VERYBAD -10)
(define DRAW 0)

;; score-by-count : board --> number
;; iterate through each row of the board (and the 90 degree rotated board)
;; and compute the score, adding them
(define (score-by-count mat)
  (local
      {
       (define (score-by-count-recursive counter)
         (cond
           [(>= counter (matrix-rows mat)) 0]
           [else
            (+
             (score-by-count-row (vector-ref mat counter))
             (score-by-count-recursive (add1 counter)))]
           ))
       }
     (score-by-count-recursive 0)

    )
  )

;; score-by-count-row : vector --> number
;; This function actually implements the scorings for various configurations, though
;; the literal values for different outcomes are stored above as constants
(define (score-by-count-row row)
  (local
      {
       (define MEs (count-squares ME row))
       (define YOUs (count-squares YOU row))
       (define BLANKs (count-squares BLANK row))
       }
    (cond
      [(and
        (> MEs 0)
        (> YOUs 0)) DRAW]
      [(= MEs 5) WIN]
      [(= MEs 4) VERYGOOD]
      [(= MEs 3) GOOD]
      [(= MEs 2) OK]
      [(= MEs 1) 1]
      [(= YOUs 4) LOSE]
      [(= YOUs 3) VERYBAD]
      [(= YOUs 2) BAD]
      [else DRAW]
      )
    )
  )

;; count-squares : any vector -> number
;; count how many of the given item are in this vector
(define (count-occurrences item vec)
  (local
      {
       (define (count-occurrences-recursive counter)
         (cond
           [(>= counter (vector-length vec)) 0]
           [(equal? (vector-ref vec counter) item)
            (+ 1 (count-occurrences-recursive (add1 counter)))]
           [else
            (count-occurrences-recursive (add1 counter))]
           ))
       }
    (count-occurrences-recursive 0))
  )

"Testing count-occurrences:"
(= 4 (count-occurrences ME (make-vector 4 ME)))
(= 0 (count-occurrences ME (make-vector 4 YOU)))
(= 4 (count-occurrences YOU (make-vector 4 YOU)))

;; score : board -> number
;; return the combined score from the original and pivoted version of the board
(define (score a-board)
  (+
   (score-by-count a-board)
   (score-by-count (pivot-matrix a-board))
   )
  )

;; create-boards : (listof matrix) -> (listof board)
;; transform a list of matrices into a list of boards, with scores assigned
(define (create-boards loms)
  (map
   (lambda (a-matrix)
     (make-Board a-matrix (score a-matrix)))
   loms
   )
  )

;; best : (listof Board) -> Board
;; pick out the best of the boards
;; note: this favors the lower right, all things being equal
(define (best list-of-boards)
  (foldr
   ;; fn to return whichever board has the higher score
   (lambda (b1 b2)
     (cond
       [(empty? b2) b1]
       [(> (Board-score b1) (Board-score b2)) b1]
       [else b2]))
   empty
   list-of-boards
   )
  )
```

```
;; make-move (listof (listof any)) -> (listof (listof any))
;; return child board-list with best score
;; This function acts as an interface to the internal matrix representation
(define (make-move a-board-list)
  (Matrix->List
   (Board-matrix
    (best
     (possible-moves
      (List->matrix a-board-list)
      )
     )
    )
   )
  )



"Testing make-move:"
(define sample-board
  (list (list '- '- '- '- '-)
        (list 'X 'X 'X 'X '-)
        (list '- 'O 'O 'O 'O)
        (list '- '- '- '- '-)
        (list '- '- '- '- '-))
  )
(make-move sample-board)

(define TEST2
  (list
   (list '- '- '- '- '-)
   (list '- 'O 'O 'O 'X)
   (list '- 'O 'X '- 'O)
   (list '- 'O '- '- '-)
   (list '- 'X 'X 'X '-)))

(make-move TEST2)
```