Project Task 12:
Minimax Part 2
By Carrie Corcoran

This is the twelfth task for my semester-long project of creating mancala playing machines. It is the second task related to minimax, in which I attempt to create a list of all possible moves from a given state. This proved unexpectedly frustrating. There are exponentially more unique starting moves than I had anticipated. I'm unable to get an exact count, but I estimate at least 3000. A recursive function will reach stack overflow long before it can find each unique starting move, even when dividing the problem into subsets. Even if it was possible to find each starting move, it would be incredibly computationally expensive to run a minimax tree on such a move. Therefore, further minimax efforts will be conducted with a subset of possible moves.

A minimax mancala player with the first move and ability to examine every move would almost certainly win every game. A winning game requires 25 points, and while every move hasn't been exhaustively examined, a cursory examination reveals an initial move that nets 18 points. It's possible that a 25 point move exists.

Code:

```
(defun valid-move-sequence (state move-list)
  ;set player and state info
  (load-state (nth 1 state))
  (setf player (nth 0 state))
  (cond
    ((eql player 'a)
      (setf side *side-a-copy*)
    )
     (t
       (setf side *side-b-copy*)
     )
  )
  (setf current-move (car move-list))
  (setf future-moves (cdr move-list))
  (cond
    ((member current-move (non-empty-spaces player 0 '() t))
      (cond
         ((=(length future-moves)0)
            t
         )
          ((single-move-test player current-move)
            (setf new-state (create-node player (save-state nil)
current-move))
```

```
          (valid-move-sequence new-state future-moves)
          )
       )
     )
      (t
        nil
      )
  )
)

(defun move-again-sequence (state move-list)

  ;set player and state info
  (load-state (nth 1 state))
  (setf player (nth 0 state))
  (cond
    ((eql player 'a)
      (setf side *side-a-copy*)
    )
     (t
       (setf side *side-b-copy*)
     )
  )
  (setf current-move (car move-list))
  (setf future-moves (cdr move-list))
  (setf again (single-move-test player current-move))
  (cond
     ((=(length future-moves)0)
       again
     )
    (t
      (setf new-state (create-node player (save-state nil)
current-move))
      (move-again-sequence new-state future-moves)
    )
  )
)
```

```
(defun valid-next-moves (state move-list)

  ;set player and state info
  (load-state (nth 1 state))
  (setf player (nth 0 state))
  (cond
    ((eql player 'a)
      (setf side *side-a-copy*)
    )
     (t
       (setf side *side-b-copy*)
     )
  )
  (setf current-move (car move-list))
  (setf future-moves (cdr move-list))
  (cond
    ((=(length move-list)0)
      (non-empty-spaces player 0 '() t)
     )
    (t
       (single-move-test player current-move)
      (setf new-state (create-node player (save-state nil)
current-move))
      (valid-next-moves new-state future-moves)
    )
  )
)
```

```
(defun find-moves (state complete-moves incomplete-moves max)
  ;set player and state info
  (load-state (nth 1 state))
  (setf player (nth 0 state))
  (cond
    ((eql player 'a)
       (setf side *side-a-copy*)
     )
     (t
       (setf side *side-b-copy*)
     )
  )
  (cond
    ((or( = (length complete-moves) max)(= (length incomplete-
moves) 0))
       complete-moves
     )
     (t
       (setf this-move (car incomplete-moves))
       (setf incomplete-moves (cdr incomplete-moves))
       (cond
         ((move-again-sequence state this-move)
            (setf next-moves (valid-next-moves state this-move))
            (setf new-moves (loop for x in next-moves collect
              (snoc x this-move))
            )
            (setf incomplete-moves (append incomplete-moves new-
moves))
            (find-moves state complete-moves incomplete-moves
max)
          )
          (t
            (setf complete-moves (cons this-move complete-
moves))
            (find-moves state complete-moves incomplete-moves
max)
          )
        )
     )
  )
)
```

```
(defun move-list-test()
  (format t "Getting some starting moves: ~%~A~%" (find-moves
(create-node 'a (save-state t) 'a1) '() '((a1-copy) (a2-
copy)(a3-copy)(a4-copy)(a5-copy)(a6-copy)) 50))
)
```

Demo:

```
[65]> (move-list-test)
Number of moves: 50
Getting some starting moves:
((A2-COPY A1-COPY A5-COPY A2-COPY) (A2-COPY A2-COPY A1-COPY A1-
COPY) (A2-COPY A2-COPY A1-COPY A3-COPY)
 (A2-COPY A2-COPY A1-COPY A6-COPY) (A2-COPY A2-COPY A2-COPY A1-
COPY) (A2-COPY A2-COPY A2-COPY A4-COPY)
 (A2-COPY A2-COPY A2-COPY A5-COPY) (A2-COPY A2-COPY A3-COPY A2-
COPY) (A2-COPY A2-COPY A3-COPY A3-COPY)
 (A6-COPY A3-COPY A3-COPY) (A6-COPY A3-COPY A4-COPY) (A6-COPY
A3-COPY A5-COPY) (A6-COPY A6-COPY A1-COPY)
 (A6-COPY A6-COPY A4-COPY) (A6-COPY A6-COPY A5-COPY) (A5-COPY
A1-COPY A1-COPY) (A5-COPY A1-COPY A2-COPY)
 (A5-COPY A1-COPY A3-COPY) (A5-COPY A1-COPY A4-COPY) (A5-COPY
A1-COPY A6-COPY) (A5-COPY A3-COPY A1-COPY)
 (A5-COPY A3-COPY A2-COPY) (A5-COPY A3-COPY A3-COPY) (A5-COPY
A3-COPY A4-COPY) (A5-COPY A3-COPY A6-COPY)
 (A5-COPY A6-COPY A2-COPY) (A5-COPY A6-COPY A4-COPY) (A3-COPY
A5-COPY A1-COPY) (A3-COPY A5-COPY A3-COPY)
 (A3-COPY A5-COPY A4-COPY) (A3-COPY A5-COPY A6-COPY) (A2-COPY
A1-COPY A2-COPY) (A2-COPY A1-COPY A6-COPY)
 (A2-COPY A2-COPY A4-COPY) (A2-COPY A2-COPY A6-COPY) (A6-COPY
A1-COPY) (A6-COPY A4-COPY) (A6-COPY A5-COPY)
 (A5-COPY A2-COPY) (A5-COPY A4-COPY) (A3-COPY A1-COPY) (A3-COPY
A2-COPY) (A3-COPY A4-COPY) (A3-COPY A6-COPY)
 (A2-COPY A3-COPY) (A2-COPY A4-COPY) (A2-COPY A5-COPY) (A2-COPY
A6-COPY) (A4-COPY) (A1-COPY))
```