CSCI 5582
Problem Set 2
Handed Out: Wednesday, Oct. 1
Due Back: Wednesday, Oct. 29

Extra problem (for 10 points credit on Problem Set 1)
This is an optional problem -- the points earned will be added to your score on problem set 1.

Write a procedure named make-number-or-list-procedure. This should take as its argument a numerical function f, and return a procedure which will "expand" the original function f so that it applies to either numbers or arbitrarily-nested lists of numbers. The newly-returned procedure behaves like f when called on a number; when called on a list of numbers, it returns a new list whose elements are just the result of calling f on each element of the argument list. Arbitrarily nested lists behave similarly—the nesting structure of the argument list is preserved, and f is applied to each numerical element of the original list.

Here are a couple of examples:

```
(define double-on-number-or-list
  (make-number-or-list-procedure double))

>>>(double-on-number-or-list 4)
8

>>>(double-on-number-or-list '(2 3 4))
(4 6 8)

>>>(double-on-number-or-list '(2 (3 4) (5) 6))
(4 (6 8) (10) 12)

>>>((make-number-or-list-procedure 1+) '(((1)) 2 3))
(((2)) 3 4)
```

Problem 1. (30 pts)
(a) In this problem, we return to the simple peg-jumping puzzle from Problem 2 of the first problem set. As a preliminary step, calculate how many distinct states there are of the (7-hole) puzzle. Write an expression for the number of distinct states there are in the (2n+1) - hole version of the puzzle.

(b) Use procedures for both depth-first search and breadth-first search to solve the puzzle. (If need be, you can use the data structure and procedures provided in your answers to the previous problem set.) What are the characteristics of these two search procedures applied to this puzzle? Note that in actually writing programs to solve the puzzle, you have to make some decisions about generating next moves: for instance, a very naive depth-first search procedure might "loop" by following a search path in which a single peg is moved back and forth forever.

(c) How would you construct an A* search strategy for this puzzle? What corresponds to the "least cost" and what might correspond to a viable estimation function?

(d) How do the various search strategies that you have considered perform as the puzzle gets extremely large? How does the length of the solution path increase with the size of the puzzle? (Use order-of-growth notation in answering the second question.)

Problem 2. (70 pts)
Note: You are encouraged to work in two-person teams for this problem.

In this problem, you will write a procedure to play the game "Connect-4". The game is played on a board of six rows by seven columns. Each column may be thought of as a "tube" into which a player might drop a (black or white) marble; the marble falls to the lowest empty position in the tube. Two players (call them W and B) take alternate moves, at each move dropping a single marble into one column. (Assume, as in chess, that white plays first.) The winner is the first player to achieve four marbles of his or her color in a horizontal, vertical, or diagonal row.

Our data structure for this game will be a list of seven lists, each representing a current column of the game board. Thus, the initial state of the game is represented by:

'(() () () () () () ())

Here are the first few moves of a sample play of the game:

```
'(      ()
        ()
        ()
        ()
        ()
        ()
        ()
)
'(      ()
        ()
        (W)
        ()
        ()
        ()
        ()
)
'(      ()
        ()
        (W)
        ()
        ()
        (B)
        ()
)
'(      ()
        (W)
        (W)
        ()
        ()
        (B)
        ()
)
'(      ()
        (W)
        (W)
        (B)
        ()
        (B)
        ()
)
'(      ()
        (W)
        (W W)
        (B)
        ()
        (B)
        ()
)
'(      ()
        (W B)
        (W W)
        (B)
        ()
        (B)
        ()
)
```

Note that a player's marble is added to the end of a list representing a column. Also note that a player cannot add a seventh marble to a column that already has six marbles in it.

(a) (0 pts; warmup) Play a few games of Connect-4 with a friend (or your teammate for the problem) to familiarize yourself with the rules and strategy.

(b) Peform some preliminary analysis of the game. What is the branching factor for the search tree for this game? Make a rough estimate of the number of distinct Connect-4 games that can be played; also, calculate the number of distinct possible "full" game boards (with 21 of each marble type) that could exist (ignore the fact that many of these game boards will correspond to games that would have ended before the board was filled up). This latter quantity is a (somewhat hazy) ballpark estimate of the number of possible states that could be encountered in a full space of legal game positions.

(c) Write a procedure named CN-[yourinitials] of the following form: the procedure should take two arguments, a symbol (either W or B) and a game-state. Your procedure should return as its result a new game-state representing the chosen move of the given player (W or B), or the symbol FAIL if you wish to concede. Thus, a sample call to my procedure might look like this (note that I've written the game-board in a more readable format):

```
>>> (CN-ME 'W
      '(   (W B)
           (B W)
           (W)
           ()
           (B)
           ()
           (W B)
      ))

'(   (W B)
     (B W)
     (W W)
     ()
     (B)
     ()
     (W B)
     )
```

You should send in your procedure (and all subsidiary procedures) by email. It is probably a good idea to give all the procedures in your code a special suffix (e.g., your initials) to ensure that your procedure names do not clash with those of the other player.

All working procedures will be entered in a classwide Connect-4 tournament (and yes, there will be prizes for the most successful procedures!). The rules of the tournament are as follows:

(i) Your procedure should be callable as either cnproc1 or cnproc2 from the following (skeletal) tournament procedure:

```
(define (tournament-game cnproc1  cnproc2)
   (game-play cnproc1 cnproc2 'W '(() () () () () () ())))


(define (game-play cnproc1 cnproc2 color-to-play game-so-far)
  (cond ((game-over? game-so-far) ; the other guy made 4 in a row
         (print-winner (switch-color color-to-play) game-so-far))
        (else
          (let ((next-game
                  (if (eq? color-to-play 'W)
                      (cnproc1 color-to-play game-so-far)
                      (cnproc2 color-to-play game-so-far))))
             (cond ((eq? next-game 'fail)
                    (print-winner (switch-color color-to-play)
                                  game-so-far))
                   (else
                    (game-play cnproc1 cnproc2
                               (switch-color color-to-play)
                               next-game)))))))
```

If you wish to test your procedure in the program above, you should write your own versions of the procedures game-over?, print-winner, and switch-color.

(ii) If your procedure causes the tournament program to halt with an error, or if it supplies an illegal move (e.g., adding two marbles at a time) it will be disqualified from the tournament. Also, all procedures will be tested against a random (legal) move-generator; any procedure requiring more than 4 minutes or so to make a move will be disqualified.

(iii) Your program should include documentation indicating the strategy behind your procedure. Explain any ideas that you may have used to build a static evaluation function, and what types of move-generating strategy you used. Procedures will be graded on a combination of their (failure-free) participation in the tournament; their sophistication; and the overall quality of the code, documentation, and accompanying discussion.

Due Wed. Oct. 22:
Hand in a one-page transcript showing your program in competition with that of another team.