Introduction
This assignment is designed to introduce you to the concept of heuristic or informed “state space search.”

Remember the visual representation of the problem is to get the game board from some initial configuration to the goal configuration (state) shown below:

Example Initial State: | Goal State:
---|---
2 | 1
8 | 2
3 | 3
1 | 8
6 | 4
4 |
7 | 7
5 | 6

Data Representation:
(This remains the same from the previous assignments)
Our representation of the 8-puzzle will be a list of elements, read from left-to-right, top-to-bottom. We will use “e” to represent the empty square in the list. So our representation of the initial state above would be:

\[(2 \ 8 \ 3 \ 1 \ 6 \ 4 \ e \ 7 \ e \ 5)\]

and the goal state is:

\[(1 \ 2 \ 3 \ e \ 4 \ 7 \ 6 \ 5)\]

Moves will be represented as a list containing the move direction of the empty space (up, down, left or right) and the resulting state. So the template is \(<\text{move-direction}> <\text{resulting-state}>\). Depending on the position of the empty square, it could move in 2, 3, or 4 directions, labeled U, D, L, and R, for up, down, left and right, respectively. So a sample of what a move would look like is:

\[(U \ (1 \ 2 \ 3 \ 8 \ e \ 4 \ 7 \ 5 \ e \ 7))\]
The path is a list of moves with the most recent move appearing first in the list and the last move in the list representing the initial state of the board (the direction in this initial move will be represented as NIL). If you look at the example below, you should be able to start at the initial state and see how each move in the list would yield the resulting state.

\[
((R \ (1 \ 2 \ 3 \ 8 \ e \ 4 \ 7 \ 6 \ 5))
\ (D \ (1 \ 2 \ 3 \ e \ 8 \ 4 \ 7 \ 6 \ 5))
\ (L \ (e \ 2 \ 3 \ 1 \ 8 \ 4 \ 7 \ 6 \ 5))
\ (U \ (2 \ e \ 3 \ 1 \ 8 \ 4 \ 7 \ 6 \ 5))
\ (NIL \ (2 \ 8 \ 3 \ 1 \ e \ 4 \ 7 \ 6 \ 5)))
\]

The open list will be a list of lists, of the form \(<path 1> <path 2> <path 3> \ldots <path n>\), where path is as described above (the initial state is at the end of the list with a move that is the symbol NIL). So an example open list is:

\[
((\ (U \ (2 \ e \ 3 \ 1 \ 8 \ 4 \ 7 \ 6 \ 5)) \ (NIL \ (2 \ 8 \ 3 \ 1 \ e \ 4 \ 7 \ 6 \ 5)))
\ (D \ (2 \ 8 \ 3 \ 1 \ 6 \ 4 \ 7 \ e \ 5)) \ (NIL \ (2 \ 8 \ 3 \ 1 \ e \ 4 \ 7 \ 6 \ 5)))
\ (L \ (2 \ 8 \ 3 \ e \ 1 \ 4 \ 7 \ 6 \ 5)) \ (NIL \ (2 \ 8 \ 3 \ 1 \ e \ 4 \ 7 \ 6 \ 5)))
\ (R \ (2 \ 8 \ e \ 1 \ 4 \ e \ 7 \ 6 \ 5)) \ (NIL \ (2 \ 8 \ 3 \ 1 \ e \ 4 \ 7 \ 6 \ 5)))
\]

Note that the last move is the same in each path (since each path will originate from the same starting board configuration). This open list has four paths in it, each with one move from the initial state – there are four moves since the initial state had the empty space (“e”) as the center tile.

Programming Rules
As before, you may not use SET, SETF, or SETQ in writing your functions – you will be writing recursive functions without these assignment statements. Such statements will be useful (and necessary) to test your code, but they do not belong in your functions themselves. Also off-limits are iteration constructs such as DO, DOTIMES, DOLIST and LOOP. The goal is to learn Lisp as a purely functional language, rather than trying to make Lisp behave in the ways that we are used to programming.

Supplied Code
You may request a copy of the previous solution sets via email.

Function Specifications
(10 pts) out-of-place: out-of-place takes one argument, a state of the 8-puzzle, and returns the number of tiles that are out of place with respect to the goal state. Do not count the “e” as out of place! For example:

\[
(out-of-place \ '((2 \ 8 \ 3 \ 1 \ 6 \ 4 \ 7 \ e \ 5))
\]
returns 4
\[
(out-of-place \ '((1 \ 2 \ 3 \ 8 \ e \ 4 \ 7 \ 6 \ 5))
\]
return 0 (since this is the goal state!)
(5 pts) out-of-place-f: out-of-place-f takes one argument, a path, and returns the f-value associated with that path. Remember that the f-value of a path is its g-cost (its history) plus its h-cost (the value of the current state). A path’s h-cost is the value returned by out-of-place for the state of the most recent move. A path’s g-cost is its depth in the search tree (the length of the path). For example:

```lisp
(out-of-place-f '((D (1 2 3 e 8 4 7 6 5))
                 (R (e 2 3 1 8 4 7 6 5))
                 (U (1 2 3 4 8 7 e 6 5))
                 (nil (1 2 3 e 8 7 4 6 5))))
```

will return 4. The h-cost of the most recent move is 1 because there is 1 tile out of place. The length of the path is 3 (the initial state doesn’t count). So the total f-cost is $1 + 3 = 4$.

(20 pts) manhattan: manhattan takes one argument, a state, and computes the “Manhattan distance” of that state from the goal. Recall that the Manhattan distance metric computes the length of the shortest path between two locations. So, for the 8-puzzle, this means summing the distance from its goal location in terms of vertical and horizontal moves for each tile. For example, the Manhattan distance of the state (2 1 4 8 6 5 3 e 7) from the goal state of (1 2 3 8 e 4 7 6 5) is $(1 + 1 + 0 + 1 + 1 + 4 + 2) = 11$ (don’t count the e as out of place).

(5 pts) manhattan-f: manhattan-f takes one argument, a path, and returns the f-value associated with that path. As in out-of-place-f, a path’s g-cost is its depth in the search tree (the length of the path). A path’s h-cost is the value returned by manhattan for the state of the most recent move. For example:

```lisp
(manhattan-f '((D (1 2 3 e 8 4 7 6 5))
                (R (e 2 3 1 8 4 7 6 5))
                (U (1 2 3 4 8 7 e 6 5))
                (nil (1 2 3 e 8 7 4 6 5))))
```

will return 4. The h-cost of the most recent move is 1 because there is 1 tile out of place, and it is one position out of place. The length of the path is 3 (the initial state doesn’t count). So the total f-cost is $1 + 3 = 4$.

(10 pts) better: better takes one argument -- an evaluation function (f). It returns a lambda expression (a function) that takes two paths as arguments. The function “f” (the one that is passed in) is intended to be a path evaluation function such as out-of-place-f or manhattan-f that returns a LOWER number for better paths. The lambda expression returned by better will take two paths as arguments and use funcall to call f on each path argument, returning T if the first is better than or equal to the second, NIL otherwise.

```lisp
(setq p1 '((U (2 8 3 1 e 4 7 6 5))
          (NIL (2 8 3 1 6 4 7 e 5)))))
(setq p2 '((U (1 2 3 8 6 e 7 5 4))
          (NIL (1 2 3 8 6 4 7 5 e)))))
(setq p3 '((L (2 1 4 8 e 7 3 6 5))
          (NIL (1 2 4 8 7 e 3 6 5)))))
```

```lisp
(funcall (better #’out-of-place-f) p1 p2)
```
(funcall (better #'out-of-place-f) p2 p1)
T (because evaluations are the same!)
(funcall (better #'out-of-place-f) p3 p2)
NIL (because more tiles are out of place in p3)

(35 pts) search-a*: This function takes two arguments – an open list and a heuristic evaluation function that works on paths. Search-A* implements the A* algorithm. It will look a lot like your breadth-first search routine, but the new open list in the recursive call will have to be sorted each time, instead of simply adding the new paths from extend-path to the back of the open list.

There is a built-in Lisp function sort that can be used to do the sorting. Sort takes two arguments: a list and a predicate which compares the two items on the list. For example:

```
(sort '(3 1 4 1 5 9) #'<)
returns (1 1 3 4 5 9)
```

You will want to sort the paths according to the heuristic evaluation function that was passed in as an argument. The function that you can create with better will be useful here, since it compares two paths according to a heuristic evaluation function.

If you want to be more efficient, instead of re-sorting portions of an already sorted list, you can just sort the new paths produced by extend-path and then merge them into the already sorted open list. Merge is a built-in Lisp function which takes a desired return type, two sorted lists, and a predicate and merges the two arguments based on the predicate. For example:

```
(merge 'list '(1 3 5 7 9) '(2 4 6 8 10) #'<)
returns (1 2 3 4 5 6 7 8 9 10)
```

The overall search-a* function works like this:

```
(setq s1 '(2 8 1 e 6 3 7 5 4))
(search-a* (make-open-init s1) #'out-of-place-f))
returns (U R R D D L U L U R D)
```

(5 pts) Modify SSS: You need to modify your SSS function by adding the keyword argument “F” – this parameter will take the heuristic evaluation function to be used in A*. Also add A* as a search type within SSS. The default values for the keyword parameters should be BFS for :type, 7 for :depth and #'out-of-place-f for :f. The first argument to SSS should still be the mandatory initial state.

General coding hint Don’t forget about the “let” construct for setting up local variables. If you use let*, you can do a lot of the up-front processing in steps, using the variables as you declare them. So you might have:

```
(let* ((v1 value1) (v2 value2) (v3 (+ value1 value2)))
  (cond
    ...))
```
And, in fact, you can use a let inside a cond (nested lets and conds – sounds like fun, right?!) So if you got to the “t” case in your cond and had some complex processing still left to do, you might use a let statement within the “t” case to set up some variables. Of course, if you get to this point, you might just want to define another helper function!

**Submission notes:** I will be loading my own solutions for the previous labs to use in testing. Any changes that you make to previously required functions will be overridden, so assume that the functions work exactly as previously specified.

**Acknowledgement:** Many thanks to Gary Cottrell for giving me permission to use his functional breakdown and description of the 8 puzzle problem as the basis for this assignment!