Introduction

This lab is designed to introduce you to the concept of “state space search.” In this lab, you will implement several search algorithms in Lisp, utilizing the “helper” functions that you implemented in the previous lab. Note that there is a solution set for the first lab available -- I will distribute it by email upon request. Note that at the end of this lab assignment, you will actually have a working 8-puzzle program.

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:  

<table>
<thead>
<tr>
<th>2</th>
<th>8</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>6</td>
<td>4</td>
</tr>
<tr>
<td>7</td>
<td>5</td>
<td></td>
</tr>
</tbody>
</table>

Goal State:  

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>6</td>
<td>5</td>
</tr>
</tbody>
</table>

Data Representation:

(the first few data elements are the same as in the last lab)

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 7 e 5)
```

and the goal state is:

```
(1 2 3 8 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 (<move-direction> <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 5 6 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)))

(here’s the new stuff…)

The **open list** (or frontier) will be a list of lists, of the form (<path 1> <path 2> <path 3> … <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.

**Function Specifications**

**10 pts make-open-init**: make-open-init takes one argument, the initial state, and returns this formatted as an open list. For example:

(make-open-init '(2 8 3 1 6 4 7 e 5))
returns (((NIL (2 8 3 1 6 4 7 e 5)))

Note the number of parentheses. This is because open list is a list of paths, paths are a list of moves, and moves are lists containing a direction and a state. In this case, we only have one path, which consists of a single “move” (the move representing the initial state). Be sure that you match this output.
(20 pts) extend-path: extend-path takes one argument, a path, and returns a list of all possible extensions to the path. It should use remove-redundant to only give extensions that do not visit previous states. For example:

```
(extend-path (first (make-open-init '2 8 3 1 6 4 7 e 5))))
```

returns `((((U (2 8 3 1 e 4 7 6 5)) (NIL (2 8 3 1 6 4 7 e 5)))
((L (2 8 3 1 6 4 e 7 5)) (NIL (2 8 3 1 6 4 7 e 5)))
((R (2 8 3 1 6 4 7 5 e)) (NIL (2 8 3 1 6 4 7 e 5))))
```

This has taken the first path off of the result of (make-open-init…), and extended it in all possible ways, making a list of three paths (all two moves long). Note that this is a legal open list all by itself.

Here’s an example of how remove-redundant makes a difference in extend-path (note that the setq is just used to set up a test value within the interpreter):

```
(setq test-path (first (extend-path (first (make-open-init '('2 8 3 1 6 4 7 e 5))))))
```

This sets test-path to the first path in the list shown above:

```
((U (2 8 3 1 e 4 7 6 5)) (NIL (2 8 3 1 6 4 7 e 5)))
```

Now we call extend-path to extend test-path:

```
(extend-path test-path)
```

which results in:

```
(((U (2 e 3 1 8 4 7 6 5)) (U (2 8 3 1 e 4 7 6 5)) (NIL (2 8 3 1 6 4 7 e 5)))
((L (2 8 3 e 1 4 7 6 5)) (U (2 8 3 1 e 4 7 6 5)) (NIL (2 8 3 1 6 4 7 e 5)))
((R (2 8 3 1 4 e 7 6 5)) (U (2 8 3 1 e 4 7 6 5)) (NIL (2 8 3 1 6 4 7 e 5)))
```

Note that the path that would have resulted from applying down to test-path has been eliminated, since it would have resulted in the same state as the initial state.

(20 pts) search-bfs: search-bfs takes one argument, an open list, and returns the path to the goal state by using a breadth-first search. For example:

```
(search-bfs (make-open-init '2 8 3 1 6 4 7 e 5))
```

returns `U U L D R`

search-bfs should check for failure (empty open list), and then get the first path off of the open list and check for success. If successful, it will return the path (using the path function). If we haven’t found our goal state yet, it should recurse on a new open list with the successors of that first path on the end of the old open list. Search-bfs will use the function extend-path in forming the new open list for the recursive call.
(20 pts) search-dfs-fd: search-dfs-fd will take two arguments, an open list and a depth bound. It will make the same checks as search-bfs, but then checks whether the first path on the open list exceeds (is $>$ than, not is $\geq$ to) the depth bound. Remember that in depth-first search, the successors are added to the beginning of the open list.

$$\text{(search-dfs-fd (make-open-init '(2 8 3 1 6 4 7 e 5)) 7)}$$
returns (U U L D R)

Be sure when you’re testing to set your depth bound to something relatively low, or you will end up with very deep recursion (and segmentation faults).

(20 pts) search-id: search-id implements the iterative deepening algorithm for searching. Search-id will take one argument (although you might want an optional argument representing the current path depth), an open list. It will use search-dfs-fd to do iterative deepening.

$$\text{(search-id (make-open-init '(2 8 3 1 6 4 7 e 5)))}$$
returns (U U L D R)

(10 pts) sss: sss stands for “state space search.” It takes one argument, the initial state, and two keyword arguments (remember, keyword arguments are the named parameters). One keyword argument is :type – this argument indicates which type of search should be used. The possible values for :type are ‘BFS ‘DFS and ‘ID (the default should be ‘BFS). The other keyword is :depth, which only makes sense with DFS. It should default to 7. sss should check whether it is at the goal state, and if not, call one of search-bfs, search-dfs-fd, or search-id depending on its keyword arguments. For example:

$$\text{(setq i ‘(2 8 3 1 6 4 7 e 5))}$$
$$\text{(setq g ‘(1 2 3 8 e 4 7 6 5))}$$
$$\text{(sss g)}$$
returns nil because the path to the goal is nil!
$$\text{(sss i)}$$
returns (U U L D R) – the search method defaults to BFS.
$$\text{(sss i :type ‘BFS)}$$
makes the BFS option explicit
$$\text{(sss i :type ‘DFS)}$$
uses DFS with the default depth bound (7)
$$\text{(sss i :type ‘DFS :depth 14)}$$
gives it a deeper depth bound
$$\text{(sss i :type ‘DFS :depth 15)}$$
finds a long path! (U U L D D R U L D R …)

Why did this happen?
$$\text{(sss i :type ‘ID)}$$ – iterative deepening will always find the optimal path

Comparing the search methods: After implementing your search functions, you should spend some time comparing them. You can do this by using the time function. For example:

$$\text{[2]> (time (sss ‘(2 8 3 1 6 4 7 e 5) :type ‘DFS))}$$
Real time: 0.001071 sec.
Run time: 0.0 sec.
Space: 4296 Bytes
(U U L D R)

Compare the time and space requirements for multiple runs of the algorithm – do you see the trends that were identified in class? You don’t need to turn anything in on your comparisons, but
after doing all of the work of implementing the search methods, you should definitely “play” with them a bit.

**Hints:** Just a few things that you might find useful for this assignment. If you’re using many of the same test cases over and over, you can set up your setq assignments in your program file (outside of any functions). This way, when you load your file, your test variables will be automatically set up for you. Please be sure to remove these test statements from your file before submitting it.

Lisp will truncate the display of results past a certain depth in a list and past a certain length in a list. You can control these limits by setting the values of *PRINT-LEVEL* and *PRINT-LENGTH* and then displaying your results using pprint (pretty print) or just print. For example:

```
[18]> (setq i '(3 4 5 (6 7) ((8)) 9) 10))
(3 4 5 (6 7) ((8)) 9) 10)
[19]> (setq *PRINT-LEVEL* 2)
2
[20]> (pprint i) ; will only print elements at first two levels of list with # representing lower-level elements
(3 4 5 (6 7) (# 9) 10)
[21]> (setq *PRINT-LENGTH* 3)
3
[22]> (pprint i) ; will only print first three elements with … representing missing elements
(3 4 5 ...)
[27]> (setq *PRINT-LENGTH* 5) ; two levels, 5 elements
5
[28]> (pprint i)
(3 4 5 (6 7) (# 9) ...)
```

Here’s a simple (but hopefully helpful) hint for the fixed-depth depth-first search. You can use the built-in length function to take the length of a list (like a path) to see how many elements it has. The length of a path will tell you how many moves it has in it, which is how deep it is in the tree. Be careful to subtract one, though, since the initial state would count as depth 0.

Also, here are some other states that can be solved (because many of the random states that you will try are unsolvable). These states should be useful for testing.

- (2 8 1 E 6 3 7 5 4) which will yield (U R R D D L U L U R D)
- (3 1 2 e 8 4 7 6 5) which will yield (U R R D L L U R D R U L L D R)
- (7 1 3 e 8 4 6 5 2) which will yield (U R R D D L L U R U L D)

**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!