Introduction
In this lab, you will write several Lisp functions that will serve as the foundation for your next several assignments. We will be working with a classic AI search problem called the “8-puzzle.” We will be using the 8-puzzle program to examine and compare different search algorithms and heuristics. However, to get started, we need some to write some Lisp code. Note that at the end of this lab assignment, you won’t actually have a working 8-puzzle program yet, but you will have many of the functions that you will need to complete the 8 puzzle program.

The 8-puzzle is a game involving 8 numbered, movable tiles set in a 3x3 frame. You may have seen versions of this game before — the flimsy little plastic versions are always popular as give-aways and birthday party favors. One cell of the frame is always empty so that adjacent tiles can be moved into that spot. For example, when you start playing the game, the configuration might look like the one on the left (below). The idea is to find a series of moves that leads you to the “goal state” shown on the right (below). There are some configurations from which it is impossible to find a solution. Don’t worry — they won’t be among my test cases!

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></td>
<td>5</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></td>
<td>4</td>
</tr>
<tr>
<td>7</td>
<td>6</td>
<td>5</td>
</tr>
</tbody>
</table>
**Data Representation**

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 \(<\text{move-direction}>\ <\text{resulting-state}>\). 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:

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

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.

\[\begin{align*}
(R \ (1 \ 2 \ 3 \ 8 \ e \ 4 \ 7 \ 6 \ 5)) \\
(D \ (1 \ 2 \ 3 \ e \ 8 \ 4 \ 7 \ 6 \ 5)) \\
(L \ (e \ 2 \ 3 \ 1 \ 8 \ 4 \ 7 \ 6 \ 5)) \\
(U \ (2 \ e \ 3 \ 1 \ 8 \ 4 \ 7 \ 6 \ 5)) \\
(NIL \ (2 \ 8 \ 3 \ 1 \ e \ 4 \ 7 \ 6 \ 5))
\end{align*}\]

**Programming Rules**

In writing your functions you may not use SET, SETF, or SETQ – 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.

**Submission**

Be sure to name your functions and order parameters EXACTLY as specified below. Submission will be via AutoLab, and your functions will be tested automatically on submission. There will also be an individual inspection of the code that can result in grade adjustments if the programming rules and specs aren’t followed.
Function Specifications

(10 pts) goal-state: We’ll get you started out on some easy ones. Goal-state takes one argument, a list representing a state of the 8-puzzle (you don’t need to check whether the state is a legal one), and returns T if the argument is the goal state and NIL otherwise. For example:

(goal-state '(2 8 3 1 6 4 7 e 5))
returns NIL

(10 pts) get-direction: Get-direction takes one argument, which is a move (see above – a move is a list consisting of a move, and the resulting state of the board). Get-direction simply returns the direction of the move. So here’s an example:

(get-direction '(L (4 5 6 7 8 e 3 2 1)))
L

(10 pts) get-state: Get-state is the second access function for moves, and it returns the state of the move. Here’s an example:

(get-state '(L (4 5 6 7 8 e 3 2 1)))
(4 5 6 7 8 e 3 2 1)

(10 pts) same-state: Same-state takes two arguments, each one a move as represented above, and returns T if the two states are the same, and NIL otherwise. Note that it doesn’t matter how the state is reached (the direction of the move). So the following function call should return T.

(same-state '(u (1 2 3 8 e 4 7 6 5)) '(d (1 2 3 8 e 4 7 6 5)))

(10 pts) path: Path takes one argument, which is a list of moves in the path format explained in the data representation section. Path should return the list of move directions in the order that they were taken. Since the path is stored in the order of most recent move in the beginning of the list and initial state at the end of the list, you must return the list of move directions in the opposite order from the way that they are stored. Note that the move direction of the initial state (NIL) should not be included in the result. Here’s an example:

(path '((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)))
(U L D R)
(25 pts) remove-redundant: Remove-redundant takes two arguments, both of them lists of moves. Remove-redundant should return the first list with any moves with states (not moves) in the second list removed. For example:

```
(remove-redundant `((R (1 2 3 8 e 4 7 6 5))
                  (U (e 2 3 1 8 4 7 6 5))
                  (D (1 2 3 7 8 4 e 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))
                (U (2 8 3 1 e 4 7 6 5))))
returns ((R (1 2 3 8 e 4 7 6 5)) (D (1 2 3 7 8 4 e 6 5)))
```

```
(remove-redundant `((R (1 2 3 8 e 4 7 6 5))
                  (U (e 2 3 1 8 4 7 6 5))
                  (D (1 2 3 7 8 4 e 6 5)))
              nil)
returns ((R (1 2 3 8 e 4 7 6 5)) (U (e 2 3 1 8 4 7 6 5))
              (D (1 2 3 7 8 4 e 6 5)))
```

This function will be useful in our search routines to avoid paths that return to the same state again.

(25 pts) moves: This is the most complicated function in this assignment. Given a state, this function returns a list of possible moves out of that state. For example:

```
(moves `(2 e 3 4 7 8 1 5 6))
returns
    `((D (2 7 3 4 E 8 1 5 6))
      (L (2 3 4 7 8 1 5 6))
      (R (2 3 E 4 7 8 1 5 6)))
```

(note that it won’t be nicely formatted like this! It will be all on one big line). You need to pay attention to which moves are legal from a given state – for example, there’s no “up” move in the example above because the empty space was already in the top row. You will almost certainly need several helper functions to accomplish this in a readable, understandable fashion.

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!