```scheme
;; Student: Céline Bensoussan
;; ID: 002144235
;; CSC 303 - Assignment 2
;; Due Tuesday, October 16, 2012

;; The 8-Puzzle is a 3x3 square tray in which are placed eight square tiles, each of which
as a number on it.
;; The ninth square is empty. Any of the tiles adjacent to the empty square can be slid into
that square, resulting
;; in a new configuration.
;; The goal is to transform some given configuration of the puzzle into this one:

;;    * * * * *     This is considered to be the "final" configuration.
;;    * 1 2 3 *     For this assignment, you will implement a function to solve
;;    * 8 0 4 *     this puzzle using any algorithm you like.
;;    * 7 6 5 *
;;    * * * * *

;;**************************************************************************
;;**************************************************************************

;; Contract: terminal: list -> boolean
;; Purpose: Returns true if the configuration we are testing is the final configuration
;; Example: (terminal? '(1 2 3 8 0 4 7 6 5)) should return #t
;; Definition:
(define terminal?
  (lambda (configuration)
    (let ((final '(1 2 3 8 0 4 7 6 5)))
      (cond ((null? configuration) #f)
            ((equal? configuration final) #t)
            (else #f)))))

;; Contract: count: atom list -> number
;; Purpose: Returns the number of occurences of a certain number in a list
;; Example: (count 3 '(5, 4, 3, 2, 3)) should return 2
;; Definition:
(define count
  (lambda (atom alist)
    (cond ((null? alist) 0)
          ((equal? atom (car alist)) (+ 1 (count atom (cdr alist))))
          (else (count atom (cdr alist))))))

;; Contract: replaceNth: list atom number -> list
;; Purpose: Replaces the nth element of a list by another element
;; Example: (replaceNth '(1 2 3 4) 5 2) should return (1 2 5 4)
;; Definition:
(define replaceNth
  (lambda (alist value index)
    (cond ((null? alist) alist)
          ((equal? 1 index) (cons value (cdr alist)))
          (else (cons (car alist) (replaceNth (cdr alist) value (- index 1)))))))

;; Contract: position : atom list -> number
;; Purpose: Returns the one-based index of an atom in a list (if atom does not exist it
should return 0)
;; Example: (position 'c '(b a c)) should return 3
;; Definition:
(define position
  (lambda (a alist)
    (cond ((or (null? alist) (equal? 0 (count a alist))) 0)
          ((equal? (car alist) a) 1)
          (else (+ 1 (position a (cdr alist)))))))
```

```scheme
;; Contract: position : list -> number
;; Purpose: Returns a number representing the position of the blank tile
;; Example: (blank-pos '(1 2 3 8 0 4 7 6 5)) should return 5
;; Definition:
(define blank-pos
  (lambda (configuration)
    (position 0 configuration)))

;; Contract: swap : number number list -> list
;; Purpose: Returns the original state but with the elements in pos1 and pos2 interchanged
;; Example: (swap 1 9 '(1 0 3 8 2 4 7 6 5)) should return (0 1 3 8 3 4 7 6 5)
;; Definition:
(define swap
  (lambda (pos1 pos2 alist)
    (cond ((null? alist) alist)
          ((equal? pos1 pos2) alist)
          ((<= pos2 pos1) (swap pos2 pos1 alist))
          ((equal? 1 pos1) (cons (list-ref alist (- pos2 1))
                                 (replaceNth (cdr alist) (car alist) (- pos2 1))))
          (else (cons (car alist) (swap (- pos1 1) (- pos2 1) (cdr alist)))))))

;; Contract: in-list? : atom list -> list
;; Purpose: Returns true if the atom is found in the given list
;; Example: (in-list? 1 '(3 4 6 1)) should return #t
;; Definition:
(define in-list?
  (lambda (x alist)
    (cond ((null? alist) #f)
          ((equal? x (car alist)) #t)
          (else (in-list? x (cdr alist))))))

;; Contract: move : atom list -> list
;; Purpose: Returns the resulting state after moving the blank correspondingly
;; Example: (move 'D '(1 0 3 8 2 4 7 6 5)) should return (1 2 3 8 0 4 7 6 5)
;; Definition:
(define move
  (lambda (symbol alist)
    (cond ((equal? symbol 'D) (if (in-list? (blank-pos alist) '(7 8 9)) alist
                                  (swap (blank-pos alist) (+ (blank-pos alist) 3) alist)))
          ((equal? symbol 'U) (if (in-list? (blank-pos alist) '(1 2 3)) alist
                                  (swap (blank-pos alist) (- (blank-pos alist) 3) alist)))
          ((equal? symbol 'R) (if (in-list? (blank-pos alist) '(3 6 9)) alist
                                  (swap (blank-pos alist) (+ (blank-pos alist) 1) alist)))
          ((equal? symbol 'L) (if (in-list? (blank-pos alist) '(1 4 7)) alist
                                  (swap (blank-pos alist) (- (blank-pos alist) 1) alist))))))

;; Contract: possib-moves : list -> list
;; Purpose: Returns a list with all the possible moves from a given state
;; Example: (possib-moves '(3 1 8 5 2 4 7 6 0)) should return (U L)
;; Definition:
(define possib-moves
  (lambda (alist)
    (let ((posibilities
           '((R D) (R D L) (D L) (U D R) (U D L R) (U D L) (U R) (U L R) (U L))))
      (list-ref posibilities (- (blank-pos alist) 1)))))
```

```
;; Contract: expand : list -> list
;; Purpose: Returns a list of the states that would result if you applied each of the moves
to the original state
;;          + for each state, the list of all the previous applied to it, and the move that
was just applied.
;; Example: (expand '(D L) '((1 2 0 8 4 3 7 6 5)(r d))) should return (((1 2 3 8 4 0 7 6 5)
(r d d)) ((1 0 2 8 4 3 7 6 5) (r d l)))
;; Definition:
(define expand
  (lambda (states alist)
    (cond ((null? states) '())
          (else (cons (list
                            (move (car states) (car alist))
                            (append (cadr alist) (list(car states))))
                      (expand (cdr states) alist) )))))

;; Contract: breath_solve : list -> list
;; Purpose: Returns a list with all the moves to apply to get the final configuration and
the number of states tested
;;          Keeps track of the number of visited states and returns an error message if the
given configuration does not have a solution
;; Example: (breath_solve '(((1 2 0 8 6 3 7 5 4)())) 0) should return ((d d l u) (34))
;; Definition:
(define breath_solve
  (lambda (to_visit count)
    (cond ((equal? count 5000)
                (display "No solution was found for this configuration after 5000 searches"))
          ((terminal? (caar to_visit)) (list (cadar to_visit) (list count)))
          (else (breath_solve (append (cdr to_visit)
                                      (expand (possib-moves (caar to_visit))
                                              (car to_visit)))
                              (+ count 1))))))

;; Contract: solve : list -> list
;; Purpose: Outputs the sequence of moves needed to go from the original configuration to
the final configuration
;; Example: (solve '(1 2 0 8 6 3 7 5 4)) should return (d d l u)
;; Definition:
(define solve
  (lambda (puzzle)
    (car (breath_solve (list (list puzzle '())) 0))))

;;(solve '(1 2 0 8 6 3 7 5 4))
```