```scheme
; Bensoussan
; Céline
; ID: 002144235

;; Contract: ident : no parameters -> string
;; Purpose: print a string with my name, my id, the course code
;; and the assignment number followed by the file name.
;; Definition:
(define ident
  (lambda ()
   (display "
*******************************************************
* Celine Bensoussan ID: 002144235 CSC303 Assignment 1 *
*******************************************************\n\n")

(display "Bens_Celi_assign_1\n")))

;; Contract: count: atom list -> number
;; Purpose: to find 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: twice? : atom list -> boolean
;; Purpose: returns true if a particular atom appears at least twice in a list of atoms
;; Example: (twice? 'c '(a b c d c b a)) should return true
;; Definition:
(define twice?
  (lambda (atom alist)
    (cond ((null? alist) #f)
          ((or (equal? 0 (count atom alist)) (equal? 1 (count atom alist))) #f)
          (else #t))))

;; Contract: remove: atom list -> list
;; Purpose: removes every occurence of an atom in a list
;; Example: (remove 'a '(a b d a c)) should return (b d c)
;; Definition:
(define remove
  (lambda (a ls)
    (cond ((null? ls) ls)
          ((equal? a (car ls)) (remove a (cdr ls)))
          (else (cons (car ls) (remove a (cdr ls)))))))

;; Contract: list_mul: alist -> number
;; Purpose: multiplies all the numbers in the list together
;; Example: (list_mul '(3 2 4)) should return 24
;; Definition:
(define list_mul
  (lambda (alist)
    (cond ((null? alist) 0)
          ((equal? 1 (length alist)) (car alist))
          (else (* (car alist) (list_mul (cdr alist)) )))))

;; Contract: list-index : atom list -> number
;; Purpose: returns the zero-based index of an atom in a list (if atom does not exist it
should return -1)
;; Example: (list-index 'c '(b a c)) should return 2
;; Definition:
```

```scheme
(define list-index
  (lambda (a ls)
    (cond ((or (null? ls) (equal? 0 (count a ls))) -1)
          ((equal? (car ls) a) 0)
          (else (+ 1 (list-index a (cdr ls)))))))

;; Contract: atom? : element -> boolean
;; Purpose: returns true if element is an atom
;; Example: (atom? '(4 5)) should return false
;; Definition:
(define (atom? x)
  (and (not (pair? x)) (not (null? x))))

;; Contract: deep-mul : alist -> number
;; Purpose: multiplies all the numbers of a possibly deep list together
;; Example: (deep-mul '( ((1 2) 3 (((4))) )) ) should return 24
;; Definition:
(define deep-mul
  (lambda (alist)
    (cond ((null? alist) 0)
          ((and (equal? 1 (length alist)) (atom? (car alist))) (car alist))
          ((and (equal? 1 (length alist)) (list? (car alist))) (deep-mul(car alist)))
          ((atom? (car alist)) (* (car alist) (deep-mul (cdr alist))))
          (else (* (deep-mul (car alist))
                   (deep-mul (cdr alist)))))))

;; Contract: duplicate: number atom -> list
;; Purpose: returns a list containing exactly n occurences of the atom
;; Example: (duplicate 3 'foo) should return (foo foo foo)
;; Definition:
(define duplicate
  (lambda(n a)
    (cond ((equal? 0 n) '())
          (else (cons a (duplicate (- n 1) a))))))

;; Contract: compose : function function -> function
;; Purpose: returns a unary function that is the composition of function f and function g
;; Example: (compose zero? sub1) 2) should return false
;; Definition:
(define compose
  (lambda (f g)
    (lambda (x) (f(g x)) )))

;; Contract: how-many : atom list -> list
;; Purpose: increments the frequency of an atom if it exists in the list
;; (if it doesn't, it will insert the pair (a 1) into the list.
;; Example: (how-many 'g '((b 3) (d 34) (g 9) (f 2))) should return ((b 3) (d 34) (g 10) (f
2))
;; Definition:
(define how-many
  (lambda (a alist)
    (cond ((null? alist) (list (list a 1)))
          ((equal? a (caar alist)) (cons (list a (+ 1 (cadar alist))) (cdr alist)))
          (else (cons (car alist) (how-many a (cdr alist)))) )))

;; Contract: list-ref1 : number list -> atom
;; Purpose: returns the (zero-based) nth element in a list
;; Example: (list-ref1 3 '(a b c d e f)) should return d
;; Definition:
(define list-ref1
  (lambda (n ls)
    (cond ((>= n (length ls)) (display "error\n"))
          ((null? ls) ls)
          ((equal? n 0) (car ls))
          (else (list-ref1 (- n 1) (cdr ls))))))

;; Contract: add-end : list datum -> list
;; Purpose: returns a new list where the datum has been added to the end of the list
;; Example: (add-end '(a b c) '(x y)) should return (a b c (x y))
;; Definition:
(define add-end
  (lambda (ls i)
    (append ls (list i))))

;; Contract: union : list list -> list
;; Purpose: returns a new set containing all elements that are either in s0 or s1
;; Example: (union '(a b c) '(b c d)) should return (a b c d) - order does not matter
;; Definition:
(define union
  (lambda (s0 s1)
    (cond ((null? s0) s1)
          ((null? s1) s0)
          ((equal? 0 (count (car s1) s0)) (cons (car s1) (union s0 (cdr s1))))
          (else (union s0 (cdr s1))))))

;; Contract: is-sorted? : list -> boolean
;; Purpose: returns true if they are in ascending order
;; Example: (is-sorted? '(1 45 67 78 79 79 80)) should return true
;; Definition:
(define is-sorted?
  (lambda (ls)
    (cond ((null? ls) #t)
          ((equal? 1 (length ls)) #t)
          ((<= (car ls) (cadr ls)) (is-sorted? (cdr ls)))
          (else #f))))

;; Contract: sumsqr : list -> number
;; Purpose: returns the sum of the squares of the numbers
;; Example: (sumsqr '(3 4)) should return 25
;; Definition:
(define sumsqr
  (lambda (ls)
    (cond ((null? ls) 0)
          (else (+ (* (car ls) (car ls)) (sumsqr(cdr ls)))))))

;; Contract: func-make : number -> function
;; Purpose: returns x multiplied by its parameter
;; Example: ((func-make 12) 7) should return 84
;; Definition:
(define func-make
  (lambda (x)
    (lambda (y) (* x y))))

;; Contract: edge : list atom -> list
;; Purpose: returns a list with all the edges of a node
;; Example: (edges '((A B)(B C)(A D)(C E)(E F)) 'A) should return (b d)
;; Definition:
(define edges
  (lambda (graph node)
    (cond ((null? graph) '())
          ((equal? node (caar graph)) (cons (cadar graph) (edges (cdr graph) node)))
          (else (edges (cdr graph) node)))))

;; Contract: path? : list atom atom -> boolean
;; Purpose: returns true if there is a path between two given nodes
;; Example: (path? '((A B)(B C)(A D)(C E)(E F)) 'A 'F) should return true
;; Definition:
(define path?
  (lambda (graph node1 node2)
    (cond ((null? graph) #f) ; if graph is empty
          ((equal? node1 node2) #t) ; if node1 == node 2
          ((equal? 1 (count node2 (edges graph node1))) #t) ; if direct path between node1
and node2
          ((null? (edges graph node1)) #f) ; if node1 is not connected to any other nodes
          (else (car (map path? (duplicate (length (edges graph node1)) graph)
                                (edges graph node1)
                                (duplicate (length (edges graph node1)) node2)))))))

;; Contract: pre-order : list -> list
;; Purpose: returns the nodes of a binary search tree in prefix sequence
;; Example: (pre-order '(200 (30 '() '()) (800 '() '()))) should return (200 30 800)
;; Definition:
(define pre-order
  (lambda (ls)
    (cond ((null? ls) ls)
          ((atom? (car ls)) (cons(car ls) (pre-order(cdr ls))))
          (else (append (pre-order(car ls)) (pre-order(cdr ls)))))))

;; Contract: insert : list atom -> list
;; Purpose: inserts a value in a binary search tree
;; Example: (insert '(200 (30 () ()) (800 () ())) 600) should return (200 (30 () ()) (800
(600 () ()) ()))
;; Definition:
(define insert
  (lambda (btree x)
    (cond ((null? btree) (list x '() '()))
          ((< x (car btree)) (list (car btree) (insert (cadr btree) x) (caddr btree)))
          ((> x (car btree)) (list (car btree) (cadr btree) (insert (caddr btree) x)))
          (else btree))))

;; 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? 0 index) (cons value (cdr alist)))
          (else (cons (car alist) (replaceNth (cdr alist) value (- index 1)))))))

;; Contract: setNth: list atom number -> list
;; Purpose: inserts an element at the nth position of a list
;; Example: (setNth '(1 2 3 4) 5 2) should return (1 2 5 4)
;; Definition:
(define setNth
  (lambda (alist value index)
    (cond ((null? alist) alist)
          ((equal? 0 index) (cons value alist))
          ((>= index (length alist)) (append alist (list value)))
          (else (cons (car alist) (setNth (cdr alist) value (- index 1)))))))

;; Contract: put-mat : number list number number -> list
;; Purpose: inserts the value in the cell at row and column of the matrix
;; Example: (put-mat 6 '((1 2 3) (4 5 6) (7 8 9)) 2 2) should return ((1 2 3) (4 5 6) (7 8 9
6))
;; Definition:
```

```scheme
(define put-mat
  (lambda (x matrix row column)
    (cond ((null? matrix) matrix)
          (else (replaceNth matrix (setNth (list-ref matrix row) x column) row)))))


;; Contract: andMap : procedure list -> boolean
;; Purpose: returns true if the application of the procedure to every number of the list
returns true (and false otherwise)
;; Example: (andMap list? '((5,0) (4,6) 0)) should return false
;; Definition:
(define andMap
  (lambda (proc ls)
    (cond ((null? ls) #t)
          ((proc (car ls)) (andMap proc (cdr ls)))
          (else #f))))

;; Contract: insert-ascending : list -> list
;; Purpose: inserts an element in a sorted list
;; Example: (index-ascending 17 '(1 8 16 20 32)) should return (1 8 16 17 20 32)
;; Definition:
(define insert-ascending
  (lambda (x alist)
    (cond ((null? alist) (list x))
          ((<= x (car alist)) (cons x alist))
          (else (cons (car alist) (insert-ascending x (cdr alist)))))))

;; Contract: sort-ascending : list -> list
;; Purpose: sorts a list in ascending order
;; Example: (sort-ascending '(6 5 4 2 1)) should return (1 2 4 5 6)
;; Definition:
(define sort-ascending
  (lambda (alist)
    (cond ((null? alist) alist)
          ((equal? 1 (length alist)) alist)
          (else (insert-ascending (car alist) (sort-ascending(cdr alist)))))))
```