```lisp
;Michael Amann
;CSE 240
;Lab 4 Richard Whitehouse
;TTH @ 3:15-4:30

;Write a Lisp function power-of-two which takes a number as a parameter
;and returns the nth powerof 2.
;         (power-of-two '8)                              ==> 256

;Pre-conditions: The argument list will only accept a positive number
;Post-conditions: The number two is multiplied by the argument as a
;power of two.

(defun power-of-two (x)
      (if (numberp x)
            (cond ((equal x '0) 1)
            (t (* 2 (power-of-two(- x 1))))
            )
      'IMPROPER_ARGUMENT_LIST
      )
)

;Write a Lisp function replicate which takes two arguments, the first
;an expression and the second a non-negative integer. It returns a list
;containing the expression
;copied the given number of times.

;     (replicate 'a 4)                             ==> (a a a a)
;     (replicate '(magic) 0)                       ==> nil
;     (replicate '(1 2) 3)                         ==> ((1 2) (1 2) (1 2))

;Pre-conditions: The functions takes two arguments, an expression such
;as 'a or 'nil, and a number.
;Post-conditions: The functions returns a list containing the number of
;instances of that expression which is determined by the second
;argument.

(defun replicate(x y)
      (if (and (and x y)
               (and (numberp y) (or (> y  0) (equal y 0))))
                 (cond ((zerop y) nil)
                   ((equal x nil)nil)
                       (t (cons x(replicate x (- y 1))))
                  )
      'IMPROPER_ARGUMENT_LIST
      )
)


;Write a Lisp function non-nil which takes a list as a parameter,
;and returns a transformed version of the list such that all nil
;elements are changed to 0 and all non-nil elements are changed to 1.

;     (non-nil '(a nil (b) (nil) 2))                  ==> (1 0 1 1 1)
```

```
;Pre-conditions: The function requires a list of atoms.
;Post-conditions: The function returns a list containing 1's and zeros.
;The zero's replace and nil element, and all other elements are
;replaced by 1's.

(defun non-nil (x)
      (if (listp x)
            (cond ((equal x nil) nil)
                  ((equal (first x) nil) (cons '0 (non-nil (rest x))))
                  ((not (equal (first x)nil))(cons '1 (non-nil (rest
x))))
            )
            'IMPROPER_ARGUMENT_LIST
      )
)


;Write a Lisp function count-atoms that counts all the atoms in a list
;passed as the parameter.
;         (count-atoms '(a b c d))                        ==> 4
;         (count-atoms '(a (b c (d e) f) (g h)))          ==> 8
;         (count-atoms '(a (b c) d))                      ==> 4

;Pre-conditions: The functions requires a list of atoms and or lists.
;Post-conditions: The functions traverses the binary tree data
;structure and counts all the atoms in the list. It then returns a
;number indicating how many atoms were
;contained in that list.

(defun count-atoms (x)
      (if (listp x)
            (cond ((null x) 0)
                  ((atom (first x)) (+ 1 (count-atoms (rest x))))
                  ((listp (first x)) (+ (count-atoms (first x)) (count-
atoms (rest x))))
            )
            'X_MUST_BE_A_LIST
      )
)


;Write a Lisp function flatten which returns a list of all the atoms in
;x. The argument x can be an atom of a list whose components, can be
;atoms or lists.
;         (flatten '(a (b (c d)) e))                       ==> (a b c d e)

;Pre-conditions: The function requires a list containing atoms and/or
;lists
;Post-conditions: The function returns a list of all the atoms in the
;list, including any atoms in a list.

(defun flatten (x)
      (if  (listp x)
            (cond ((equal x  nil) nil)
                  ((atom (first x)) (cons (first x) (flatten (rest
x))))
```

```
                  ((listp (first x)) (append  (flatten (first x))
(flatten (rest x))))
            )
      'X_MUST_BE_A_LIST
      )
 )

;Write a Lisp function my_member that works the same as the lisp member
;function (do not use the member function in your solution).

;Pre-conditions: The function takes a list of elements and a test case
;argument. The argument list must consist of an atom for x and a list
;for y. Post-conditions: The function compares the test case element to
;the list and searches for a match of the test case. If none is found,
;nil is returned. If a match is found, that element along with the
;remaining elements after the match are returned.

(defun my_member (x y)
      (if (and (atom x) (listp y))
            (cond       ((equal y  nil) nil)
                  ((and (atom (first y)) (equal (first y) x))
                        (cons (first y)  (rest y)))
                  ((listp (first y))(my_member x (rest y)))
                  ((and (atom (first y))(not(equal(first y) x)))
                        (my_member x (rest y)))
            )
      'IMPROPER_ARGUMENT_LIST
         )
      )

;Write a Lisp function sub-splice that takes three parameters: the new
;item, the old item to be changed and the list to be edited. Your
;function will return a new version of the list with all the
;occurrences of the old item replaced with the new item.

 ;(sub-splice 3 1 '(1 2 (1 2 (1 2))))            ==> (3 2 (3 2 (3 2)))
 ;(sub-splice '(1 2) 'b '(a b c))                ==> (a 1 2 c)
 ;(sub-splice '(1 2) 'b '(a (b c) d))            ==> (a (1 2 c) d)

;Pre-conditions: The function takes three parameters. They are the new
;item, the old item to be changed and the list to be edited. The first
;parameter can be a list or an atom. The second parameter must be an
;atom. The third parameter must be a list. Post-conditions: The
;function returns a new version of the list to be changed with all the
;occurrences of the old item replaced with the new item.

(defun sub-splice (x y z)
      (if   (or (and (atom x)(atom y)(listp z))
            (and (listp x)(atom y)(listp z)))
      (cond ((equal z nil) nil)
            ((and (atom (first z))(not(equal(first z)y)))
                  (cons (first z)(sub-splice x y (rest z))))
            ((and (atom (first z)) (equal (first z) y)(listp x))
                  (cons (first x) (cons (first(rest x)) (sub-splice x y
(rest z)))))
            ((and (atom (first z)) (equal (first z)y))
                  (cons x (sub-splice x y (rest z))))
```

```
                    ((listp (first z)) (cons (sub-splice x y (first z))
                                             (sub-splice x y (rest z))))

          )
          'NIL
          )
)
```