```lisp
;*****************************************************************************
; Programming Assignment #1 Solutions
; MCS 380/590 Intro to AI
;*****************************************************************************

;*****************************************************************************
;2.2
;*****************************************************************************

(defun test_conses ()
    (print '(cons 'a '(b c)))
    (print (cons 'a '(b c)))
    (print '(cons 'a (cons 'b '(c))))
    (print (cons 'a (cons 'b '(c))))
    (print '(cons 'a (cons 'b (cons 'c nil))))
    (print (cons 'a (cons 'b (cons 'c nil)))))

;*****************************************************************************
;2.7
;*****************************************************************************

(defun nested_lisp (x)
    (cond
        ; check if arg, itself, is a list . . .
        ((not (listp x)) (listp x)) ; just return nil if true

        ; if first arg isn't null, test to see if its a list
        ((listp (car x))) ; return t if it's a list

        ; if not a list either, then keep going
        ((null (cdr x)) nil) ;end of input, just return nil

        ; still members left, keep traversing
        (t (nested_lisp (cdr x)))))


;*****************************************************************************
;3.2
;*****************************************************************************

(defun un (a b)
    (cond
        ((null b) a)
        ((member (car b) a) (un a (cdr b)))
        (t (un (append a (list (car b))) (cdr b)))))

;*****************************************************************************
;3.4
;*****************************************************************************

; why does the following return null?

; (member '(a) '((a) (b)))

; ex. test out an our-member function first to see details of member

(defun our-member (obj lst)
  (if (null lst)
     nil
```

```
    (if (eql (car lst) obj)
        lst
        (our-member obj (cdr lst)))))


; the member function, after it takes a car of the 2nd argument = (A), will
; break down further to the predicate (eql '(a) '(a)) which returns nil.
; eql is just a test which returns true iff object1 and object2 are eq, or
; the same character, or numbers that would look the same when printed.  If
; this instead relied upon the predicate similiar to 'equal', which compares
; s-expressions, then this comparison would return true.

;*******************************************************************************
;3.5a recursive
;*******************************************************************************

(defun pos_rec+ (arg0 &optional (argv 0))
    (progn (if (not arg0) arg0
        (progn (setf (car arg0) (+ (car arg0) argv))
          (pos_rec+ (cdr arg0) (+ argv 1)))) arg0))

;*******************************************************************************
;3.5b iteratively
;*******************************************************************************

(defun pos_iter+ (lst)
    (let ((pos 0))
        (dolist (obj lst)
            (setf (nth pos lst) (+ pos obj))
            (setf pos (+ pos 1))) lst))

;*******************************************************************************
;3.5c using mapcar
;*******************************************************************************

(defun pos_mapcar+ (arg) (mapcar #'(lambda (x) (+ x (position x arg))) arg))


;*******************************************************************************
;5.7a recursively
;*******************************************************************************

; define a function that takes a list of numbers and returns true iff the
; difference b/w each successive pair of them is 1.

; progression function

(defun pro_rec (x)
  (cond
    ((endp x))
    ((null (second x)))
    ((= (1+ (car x)) (second x)) (pro_rec (cdr x)))
    (t nil)))

;*******************************************************************************
;5.7b iteratively with do
;*******************************************************************************

(defun pro_iter (x)
```

```
  (do ((lst x (cdr lst)))
    ((or (endp lst) (null (second lst))) t)
      (cond
        ((= (1+ (car lst)) (second lst)))
        (t (return nil)))))


;*****************************************************************************
;5.7c using mapc and return
;*****************************************************************************

(defun pro_mapc (x)
    (setf tmp t)
    (setf cpy (copy-list x))
    (pop cpy)
    (mapc #'(lambda (x cpy)
                (if (equal (abs (- x cpy)) 1) t (setf tmp nil))) x cpy)
    (if (equal z t) (block nil (return t))))


(defun test_pro ()
    (print '(pro_rec '(1 2 3 4)))
    (print (pro_rec '(1 2 3 4)))
    (print '(pro_rec '(1 2 3 5)))
    (print (pro_rec '(1 2 3 5)))
    (print '(pro_rec '(1 2 3 4 10)))
    (print (pro_rec '(1 2 3 4 10)))
    (print '(pro_rec '(1 3)))
    (print (pro_rec '(1 3)))
    (print '(pro_rec '(1 2)))
    (print (pro_rec '(1 2)))
    (print '(pro_rec '(1 2 3 4 5)))
    (print (pro_rec '(1 2 3 4 5)))

    (print '(pro_iter '(1 2 3 4)))
    (print (pro_iter '(1 2 3 4)))
    (print '(pro_iter '(1 2 3 5)))
    (print (pro_iter '(1 2 3 5)))
    (print '(pro_iter '(1 2 3 4 10)))
    (print (pro_iter '(1 2 3 4 10)))
    (print '(pro_iter '(1 3)))
    (print (pro_iter '(1 3)))
    (print '(pro_iter '(1 2)))
    (print (pro_iter '(1 2)))
    (print '(pro_iter '(1 2 3 4 5)))
    (print (pro_iter '(1 2 3 4 5))))

;*****************************************************************************
;6.3
;*****************************************************************************

(defun ca (&rest args)
  (length args))


(defun test_ca ()
    (print '(ca))
    (print (ca))
    (print '(ca 'a))
```

```
    (print (ca 'a))
    (print '(ca 'a 'b 'c))
    (print (ca 'a 'b 'c))
    (print '(ca 'a '(b c)))
    (print (ca 'a '(b c)))
    (print '(ca '(a '(b c))))
    (print (ca '(a '(b c))))
    (print '(ca 'a '(b '(c d))))
    (print (ca 'a '(b '(c d))))
    (print '(ca '(a b c d '(e f g))))
    (print (ca '(a b c d '(e f g))))
    (print '(ca '(a b c '(d e '(f g)))))
    (print (ca '(a b c '(d e '(f g)))))
    (print '(ca '(a b c '(d '(e f)))))
    (print (ca '(a b c '(d '(e f))))))

;****************************************************************************
;8.1 - is it possible for two symbols to have the same name but not be eql?
;****************************************************************************

; Yes, as demonstrated by the following example

; USER(30): (setq one-ref (make-symbol "X"))
; #:X
; USER(31): (setq two-ref (make-symbol "X"))
; #:X
; USER(32): (eql one-ref two-ref)
; NIL

; There are other possible cases dealing with scoping of variables that are
; also correct.  There also exists cases that are possibly implementation
; dependent.

;****************************************************************************
;x.x
;****************************************************************************

; Write a function to compute the binomial coefficients

(defun C (n k)
  (cond
    ((= k 0) 1)
    ((= k n) 1)
    (t (+ (C (- n 1) k) (C (- n 1) (- k 1))))))

(defun test_binomial ()
   (print '(C 5 0))
   (print (C 5 0))
   (print '(C 5 1))
   (print (C 5 1))
   (print '(C 5 2))
   (print (C 5 2))
   (print '(C 5 3))
   (print (C 5 3)))

;****************************************************************************
; demonstrate the code
;****************************************************************************
```

```
(defun make ()
    (test_conses)
    (print '(nested_lisp '(a b c d e f g)))
    (print (nested_lisp '(a b c d e f g)))
    (print '(nested_lisp 'a))
    (print (nested_lisp 'a))
    (print '(nested_lisp '(a b c d (e f (g h i j)))))
    (print (nested_lisp '(a b c d (e f (g h i j)))))
    (print '(un '(a b c) '(b a d)))
    (print (un '(a b c) '(b a d)))
    (print '(pos_rec+ '(9 8 7 6 5 4 3 2 1)))
    (print (pos_rec+ '(9 8 7 6 5 4 3 2 1)))
    (print '(pos_iter+ '(9 8 7 6 5 4 3 2 1)))
    (print (pos_iter+ '(9 8 7 6 5 4 3 2 1)))
    (print '(pos_mapcar+ '(7 6 5 4 3 2 1)))
    (print (pos_mapcar+ '(7 6 5 4 3 2 1)))
    (test_pro)
    (test_ca)
    (test_binomial)
    (print 'END_OF_TEST_OUTPUT))
```