```lisp
"""#-----------------------------------------------------------------------------------#"""
"""#----------------------      METU Cognitive Sciences         ----------------------#"""
"""#----------------------          Turgay Yıldız               ----------------------#"""
"""#----------------------      yildiz.turgay@metu.edu.tr       ----------------------#"""
"""#-----------------------------------------------------------------------------------#"""


"""#-----------------------------------------------------------------------------------#"""
"""#----------------------          Exercise 4.1                ----------------------#"""
"""#-----------------------------------------------------------------------------------#"""

; Construct the lists formed by the below expressions, using only CONS, elements,
; and NIL – do not forget the quotes where needed.

; (a)(list 'a 'b 'c)

(cons 'a (cons 'b (cons 'c nil) ) )

; (A B C)

; (b)(list 'a 'b NIL)

(cons 'a (cons 'b (cons nil nil) ) )

"""#-----------------------------------------------------------------------------------#"""
"""#----------------------          Exercise 4.2                ----------------------#"""
"""#-----------------------------------------------------------------------------------#"""

; Write forms consisting only of CONS, NIL, ', A, B, C, D, which evaluate to the lists below.

; (a)

(cons 'a nil)

; (A B C D)

(cons 'a (cons 'b (cons 'c (cons 'd nil))))

; (b)(A (B (C D)))

(cons 'a ( cons  (cons 'b  (cons  (cons 'c   (cons 'd nil) )  () ) )   ()  ) )

; (c)(A (B (C) D))

(cons 'a   (cons   (cons 'b   (cons   (cons 'c nil) (cons 'd nil) ) ) nil ) )

; (d)(((A (B (C) D))))

( cons (cons (cons 'a  (cons   (cons 'b   (cons  (cons 'c nil)   (cons 'd nil) ) )  nil) ) nil) nil)

"""#-----------------------------------------------------------------------------------#"""
"""#----------------------          Exercise 4.3                ----------------------#"""
"""#-----------------------------------------------------------------------------------#"""

; Give the sequences of car’s and cdr’s needed to get x in the following expressions;
; for convenience name the list under discussion as 1'st – the first one is answered to
; clarify the question:

; (a b x d)

(let  ( (     list2  (cons 'a (cons 'b (cons 'x (cons 'd nil))))      ) )

    (car (cdr  (cdr list2) ) )
)

; (a (b (x d))  )

(let  ((   list3    ( cons 'a      (cons    (cons 'b (cons (cons 'x  (cons 'd nil) ) nil ) )  nil  )) ))

    (print "(a (b (x d)))")
    (print list3)

    (car (car (cdr (car (cdr list3)  ) ) ) )
)

"""important : (cdr list3 )  is   ((B (X D)))"""


""" CDRs get out with parentheses
    you have to get rid of parentheses by using CAR """

; (((a (b (x) d))))

(let    ((  list4   (cons (cons  (cons 'a  (cons  (cons 'b (cons   (cons 'x nil)  (cons 'd nil) ) )  nil) ) nil) nil) ))

    (print "(((a (b (x) d))))")
    (print list4)

    (car (car (cdr (car (cdr (car (car list4)) )) )))
)

"""#-----------------------------------------------------------------------------------#"""
"""#----------------------          Exercise 4.4                ----------------------#"""
"""#-----------------------------------------------------------------------------------#"""

; Given the list ((A B) (C D) (E F))
; 1. Write what you would get from it by applying the following in order,

; (a)CAR                :  (A B)
; (b)CDR CDR            :  ( (E F) )
; (c)CAR CDR            :  (B)
; (d)CDR CAR            :  (C D)
; (e)CDR CDR CAR        :  (E F)
; (f) CDR CAR CDR CDR   :   nil

(let ((    list1     (cons   (cons 'a (cons 'b nil))    (cons (cons 'c (cons 'd nil))     (cons (cons 'e  (cons 'f nil) ) nil) ) )   ))

    (print "((A B) (C D) (E F))")
    (print list1)
```

```
    (print "-----------------------------------------")
    (print (car list1))
    (print "My anser : (A B)")
    (print "-----------------------------------------")
    (print (cdr  (cdr list1)) )
    (print "My anser : ((E F))")
    (print "-----------------------------------------")
    (print (cdr (car list1)) )
    (print "My anser : (B)")
    (print "-----------------------------------------")
    (print (car (cdr list1)))
    (print "My anser : (C D)")
    (print "-----------------------------------------")
    (print (car (cdr (cdr list1))) )
    (print "My anser : (E F)")
    (print "-----------------------------------------")
    (print (cdr (cdr (car (cdr list1)))))
    (print "My anser : (nil)")
    (print "-----------------------------------------")
)



; Given the list ((A B) (C D) (E F))
; 2. Which sequences of CARs and CDRs would get you A, B and F?

(let ((    x     (cons   (cons 'a (cons 'b nil))    (cons (cons 'c (cons 'd nil))     (cons (cons 'e  (cons 'f nil) ) nil) ) )   ))

        (car (car x))

)


(let ((    x     (cons   (cons 'a (cons 'b nil))    (cons (cons 'c (cons 'd nil))     (cons (cons 'e  (cons 'f nil) ) nil) ) )   ))

        (car (cdr (car x)))

)

(let ((    x     (cons   (cons 'a (cons 'b nil))    (cons (cons 'c (cons 'd nil))     (cons (cons 'e  (cons 'f nil) ) nil) ) )   ))

        (car (cdr (car (cdr (cdr x)))) )

)


"""#------------------------------------------------------------------------------------#"""
"""#----------------------          Exercise 4.5                ----------------------#"""
"""#------------------------------------------------------------------------------------#"""

; Write down what the following expressions evaluate to; work them out before try-
; ing on the computer. Some expressions might cause an error; just mark them as an
; error, no need to specify the error itself.


; 1.    (cons 2)            ->      error           takes two elements

; 2.    (cons 2 NIL)        ->      (2)

; 3.    (cons 3 '(2))       ->      (3 2)

; 4.    (cons 3 (2))        ->      error           GRE, searches for a procedure

; 5.    (cons NIL NIL)      ->      (nil)

; 6.    (cons (1 2) NIL)    ->      error           GRE

; 7.    (cons '(1 2) NIL)   ->      (  (1 2 )  )

; 8.    (cons (A B) NIL)    ->      error           GRE

; 9.    (cons ('A 'B) NIL)  ->      error           GRE

; 10.   (cons '(A B) NIL)   ->  (  (A B)  )


; 11.   (cons '(A B) '(C D))    ->  ( (A B) C D)

; 12.   (list 1 4)              ->  (1 4)           makes outputs as a list's elements

; 13.    (list 1 '4)            ->  (1 4)

; 14.   (list '1 4)             ->  (1 4)

; 15.   (list 'A B)             ->  ERROR           B returns unbound error

; 16.   (list 'A 4)             ->  (A 4)

; 17.   (list 'A 'B)            ->  (A B)

; 18.   ('list 1 4)             ->  ERROR           GRE

; 19.   (+ 3 '4)                ->  7               ->  VERY IMPORTANT !!!

""" Because ' quote does not turn them into strings. """

; 20.    ('+ 1 4)               ->  ERROR           GRE

; 21.    (list 3 'times '(- 5 2) 'is 9)     ->  (3 TIMES (- 5 2) IS 9)

; 22.    (list 3 'times (- 5 2) 'is '9)     ->  (3 TIMES 3 IS 9)



"""#------------------------------------------------------------------------------------#"""
"""#----------------------          Exercise 4.6                ----------------------#"""
"""#------------------------------------------------------------------------------------#"""
```

```lisp
; Write down what the following expressions evaluate to; work them out before trying on the computer.

; 1.

(if (listp  '(list 1 2))
        'ok
        'not-really
)

; since '(list 1 2) is a list, do the first
; repl : OK

; note that "(list 1 2)" does the same


;  2.

(if     (null (nil))
    'vice
    'versa
)

; (nil)  will give error... GRE
; (null '() )   will give  TRUE
; (null '(nil) ) =  (null '(()) )   will give NIL

;   3.

(and    (listp (if (> 2 4) (- 2 4) (+ 2 4))  )

    (if (> 2 4) (- 2 4) (+ 2 4)))

; this "(if (> 2 4) (- 2 4) (+ 2 4)) " will give 6. However, 6 is NOT a list !!!
; therefore, (and    nil   ... )  if and found a nil output is nil.


;   4.

(or   (listp    (if (> 2 4) (- 2 4) (+ 2 4)))

        (if (> 2 4) (- 2 4) (+ 2 4)))

; 6     Because "or" just searches for a non-nil


;   5.

(or     (and (or 'or) 'and)     'or)


; and


""" This is from old version of pdf     """

; The Collatz sequence (see Exercise 3.6) of a positive integer is the sequence starting
; with the number itself and ending with 1, where the numbers in-between are the
; results of Collatz steps. For instance the Collatz sequence of 3 is 3 10 5 16 8 4 2 1.
; Given a non-negative integer, compute the count of even and odd numbers in
; its Collatz sequence. Return the result as a list of two numbers, the first is the even
; count and the second is the odd count. The solution for 3 will be (5 3).



(defun collatz (x)

    (cond   (   (= x 1)     1)
        (   (evenp x)   (/ x 2)  )
        (   (oddp  x)   (+  (* 3 x)  1 )  )
    )
)




(defun countEO (x &key (even-count 0) (odd-count 0) )



    (if     (= x 1)

        (list even-count  (+ odd-count 1)   )


        (if (evenp x)
            (countEO  (collatz x)   :even-count (+ even-count 1)    :odd-count  odd-count)
            (countEO  (collatz x)   :even-count even-count          :odd-count  (+ odd-count 1)  )
        )
    )
)


"""#-------------------------------------------------------------------------------------#"""
"""#----------------------          Exercise 4.7               ----------------------#"""
"""#-------------------------------------------------------------------------------------#"""


(defun insert-2nd   (x y)

    (and  (print   (cons    (car y)     (cons       x           (cdr   y  )    )    )    )   t)


)


"""#-------------------------------------------------------------------------------------#"""
"""#----------------------          Exercise 4.8               ----------------------#"""
"""#-------------------------------------------------------------------------------------#"""
```

```lisp
(defun replace-2nd   (x y)

    (and (print (cons (car y)  (cons x (cdr (cdr y))) ) ) t)
)


"""#----------------------------------------------------------------------------------#"""
"""#----------------------          Exercise 4.9                ----------------------#"""
"""#----------------------------------------------------------------------------------#"""

(defun swap (y)

    (and   (print  (cons   (car (cdr y) )    (cons  (car y)  nil)   )  )  t)
)


"""#----------------------------------------------------------------------------------#"""
"""#----------------------          Exercise 4.10               ----------------------#"""
"""#----------------------------------------------------------------------------------#"""

(defun append1  (x y)

    (and (print  (append  y    (cons x  nil)  ) ) t)
)


"""#----------------------------------------------------------------------------------#"""
"""#----------------------          Exercise 4.11               ----------------------#"""
"""#----------------------------------------------------------------------------------#"""

(defun append2   (x   y)

    (and (print   (cons   (car x)      (cons     (car (cdr x))   y) ) ) t)
)


"""#----------------------------------------------------------------------------------#"""
"""#----------------------          Exercise 4.12               ----------------------#"""
"""#----------------------------------------------------------------------------------#"""


(defun list-4th  (x)

    (and (print ( cadddr x) ) t)
)


"""#----------------------------------------------------------------------------------#"""
"""#----------------------          Exercise 4.13               ----------------------#"""
"""#----------------------------------------------------------------------------------#"""

(defun after-first (x y)

    (and (print ( cons (car x)   (append  y (cdr x) )      )     )  t)
)

"""------------------------------------------------------------------------------------"""
"""-----------------                   Exercise 4.14                  -----------------"""
"""------------------------------------------------------------------------------------"""


(defun get-last-n-elm (x n)     ; n  =  index  +  1

    (if (=  n   0)
        x
        (get-last-n-elm  (cdr x)  (-  n  1) )
    )
)

; (get-last-n-elm  '(a b c d e f g h ) 5)  will give  (f g h)

"""------------------------------------------------------------------------------"""

(defun get-reverse-of-first-n-elm  (x   n   storage)   ;  n  =  index + 1

        (if (=  n  0)
            storage
            (get-reverse-of-first-n-elm  (cdr x)  (-  n  1)   (cons  (car  x)  storage))
        )
)
; (get-reverse-of-first-n-elm '(a b c d e f g h )  5  nil)   will give   (E D C B A)

"""------------------------------------------------------------------------------"""

(defun get-first-n-elm   (x   n)                 ; n   =   index + 1

        (get-reverse-of-first-n-elm (get-reverse-of-first-n-elm  x   n   nil) n nil)
)
; (get-first-n-elm '(a b c d e f g h )  5  nil)   will give    (A B C D E)

"""------------------------------------------------------------------------------"""

(defun insert-after-nth (x y index)

    (append  ( get-first-n-elm x (+ index 1) )      (append  y   (get-last-n-elm  x (+ index 1) ) ) )
)

; (insert-after-nth '(a b c d e f g h) '(x) 4)    will give   (A B C D E X F G H)


"""------------------------------------------------------------------------------------"""
"""-----------------                   Exercise 4.15                  -----------------"""
"""------------------------------------------------------------------------------------"""

(defun above-TH  (x  th  storage)       ; ( (A 100) (B 150) (C 200))

    (if (null x)
        storage
```

```
    (if  (>=  (cadar x)  th)
        (above-TH   (cdr x)  th   (cons  (cadar x)   storage) )
        (above-TH   (cdr x)  th     storage)
    )
  )
)



"""-----------------------------------------------------------------------------------------"""
"""-----------------                   Exercise 4.16                   ----------------"""
"""-----------------------------------------------------------------------------------------"""


(defun order (x y)              ; (member 'x '(a b c x d e f))   -> (x d e f )

  (if (member x  y)
      (+   (-   (length y)   (length (member x y)) )    1)
      nil
  )
)

"""-----------------------------------------------------------------------------------------"""
"""-----------------                   Exercise 4.17                   ----------------"""
"""-----------------------------------------------------------------------------------------"""

(defun  add_list (x   storage)      ; storage initially = 0
                                    ; x -> (a b c 11 23 45 bg ... )
  (if (null  x)
      storage
      (if (numberp   (car x) )
          (add_list  (cdr x)      (+  storage  (car x)) )
          (add_list  (cdr x)      storage)
      )
  )
)


(defun add-list2  (x)

  (if (null x)
      0
      (if (numberp (car x) )
          (+    (car x) (add-list2 (cdr x) ) )
          (add-list2 (cdr x))
      )
  )
)


"""-----------------------------------------------------------------------------------------"""
"""-----------------                   Exercise 4.18                   ----------------"""
"""-----------------------------------------------------------------------------------------"""


(defun find_max  (x    max_value)   ; initially zero

  (if (endp x)
      max_value
      (if (and   (numberp   (car  x) )   (>   (car x)  max_value))
          (find_max   (cdr x)   (car x) )
          (find_max   (cdr x)   max_value)
      )
  )
)

""" ----------------       second way      ----------------   """


(defun make_number (x storage)

  (if (endp x)
      storage
      (if (numberp (car x))
          (make_number   (cdr x)   (cons (car x)    storage) )
          (make_number   (cdr x)            storage)
      )
  )
)

(defun  find_max_2  (x)

  (let    ((  y   (make_number x nil) ) )

          (if (= (length y) 1)
              (car y)
              (if (>  (car y)  (cadr y) )
                  (find_max_2 (cons (car y)  (cddr y) ))
                  (find_max_2 (cons  (cadr y) (cddr y)))
              )
          )
  )
)



"""-----------------------------------------------------------------------------------------"""
"""-----------------                   Exercise 4.19                   ----------------"""
"""-----------------------------------------------------------------------------------------"""


(defun sec_largest  (x   sec_val   max_val)    ; initially 0 and 1

  (if (endp x)
      sec_val
      (if (>  (car x)    max_val)
          (sec_largest   (cdr x)   max_val    (car x) )
          (if (>  (car x)    sec_val)
              (sec_largest   (cdr x)    (car x)    max_val)
```

```lisp
                (sec_largest   (cdr x)    sec_val    max_val )
            )
        )
    )
)


"""-------------------------------------------------------------------------------------"""
"""-----------------                   Exercise 4.20                  -----------------"""
"""-------------------------------------------------------------------------------------"""


(defun find-smallest  (x    smallest-value   index   pseudo-index)      ; initially smallest-value  = a big number
                                                                        ; index and pseudo-index are initially 0.
    (if (null x)
        (cons  smallest-value (cons (- index 1) nil) )

        (if (<   (car x)  smallest-value)
            (find-smallest   (cdr x)   (car x)   (+ pseudo-index 1)   (+ pseudo-index 1)   )
            (find-smallest   (cdr x)   smallest-value     index     (+ pseudo-index 1) )
        )
    )
)

; output will be    (smallest-value    index)
; (find-smallest '(4  5 7 90 2 1 7 9 )  999999999999999  0 0 )            will give (1 5)

(defun ordered (x  ordered-x)  ; ordered-x  initially nil  ()

    (if (null x)
        ordered-x

        (ordered    (append
        (subseq    x   0   (car (cdr (find-smallest  x    999999999   0   0)) ) )
        (subseq    x   (+ 1  (car (cdr (find-smallest  x    999999999   0   0)) ) )    (length x))
                )
        (cons (car (find-smallest  x    999999999   0   0))  ordered-x)
        )
    )
)

;  (ordered '(2 5 7 1 3 0 9 6 3 ) nil)      will give   (9 7 6 5 3 3 2 1 0)

(defun ordered-2   (x  ordered-x)  ; ordered-x  initially nil  ()

    (let (  (smallest      (car (find-smallest  x    999999999   0   0))  )
        (index         (car (cdr (find-smallest  x    999999999   0   0)) ) )
        )

        (if (null x)
            ordered-x

            (ordered-2      (append
            (subseq    x   0     index )
            (subseq    x   (+ 1  index )    (length x))
                )
            (cons smallest  ordered-x)
            )
        )
    )
)


(defun n-th-largest  (x   n)

    (nth   (- n 1)   ( ordered x    nil) )
)

(defun n-th-largest-2  (x  n)

    (if (= n  0)
        (car x)
        (n-th-largest (cdr x)  (- n 1) )
    )
)


"""-------------------------------------------------------------------------------------"""
"""-----------------                   Exercise 4.21                  -----------------"""
"""-------------------------------------------------------------------------------------"""


(defun last1  (x    last_element)

    (if (null x)
        last_element
        (last1  (cdr x)   (car x) )
    )
)

"""-------------------------------------------------------------------------------------"""
"""-----------------                   Exercise 4.22                  -----------------"""
"""-------------------------------------------------------------------------------------"""


(defun multi-member  (x   y)

    (if (null y)
        nil
        (if (listp (car y) )
            (multi-member   x   (append  (car y)  (cdr y) ) )
            (if (equal  x   (car y) )
                t
                (multi-member  x  (cdr y) )
            )
        )
    )
)
```

```
"""-------------------------------------------------------------------------------------------"""
"""------------------                   Exercise 4.23                   -----------------"""
"""-------------------------------------------------------------------------------------------"""


(defun rec-mem  (x   y  counter)

    (cond   (   (endp y)    counter)
            (   (and  (not (listp (car y))) (equal x (car y) )  )                (rec-mem   x    (cdr y)  (+ counter 1) )    )
            (         (not (listp (car y)))                                      (rec-mem   x    (cdr y)      counter )      )
            (   (listp  (car y) )                                                (rec-mem     x    (append (car y) (cdr y))    counter) )
    )
)

; if you use the name "count" instead of "counter" it may give error. Because "count" is an inbuilt function like
; (count 1 '(1 2 3 1 1 ) )  will give   3

"""-------------------------------------------------------------------------------------------"""
"""------------------                   Exercise 4.24                   -----------------"""
"""-------------------------------------------------------------------------------------------"""


(defun level  (x   y  depth)

    (cond
            (   (endp y)                                                    nil)
            (   (and  (not (listp (car y)) )   (equal x (car y))   )        depth)
            (   (not (listp (car y)) )                                      (level x (cdr y) depth)  )
            (   (listp (car y))                                             (level x  (append  (car y)  (cdr y)) (+ depth 1) ) )
    )
)


"""-------------------------------------------------------------------------------------------"""
"""------------------                   Exercise 4.25                   -----------------"""
"""-------------------------------------------------------------------------------------------"""

; convert binary to decimal w/o checking the length

(defun get_reverse  (x storage)

        (if (endp x)
            storage
            (get_reverse  (cdr x)  (cons  (car  x)  storage))
        )
)


(defun binary_to_decimal   (y    res_as_dec   power)

    (if (endp y)
        res_as_dec
        (binary_to_decimal   (cdr y)    (+ res_as_dec (* (car  y) (expt  2  power) ) )    (+ power 1) )
    )
)

(defun bin_to_dec   (x)

    (let    (   ( y (get_reverse  x   nil)   )  )

                (binary_to_decimal   y    0    0  )
    )
)


"""-------------------------------------------------------------------------------------------"""
"""------------------                   Exercise 4.26                   -----------------"""
"""-------------------------------------------------------------------------------------------"""


(defun enumerate  (x  counter storage)

    (if (endp x)
        storage
        (enumerate  (cdr x)  (+ counter 1)   (append storage  (list (cons  counter   (cons  (car x)  nil))))  )

    )
)
; add from right to left, first (append  nil  list (0 A) )   =   (  (0 A)  )
;     (append   ( (0 A) )   (list  (1 B) ))    =   (  (0 A)  (1 B) )

"""#----------------------      Second Way      -----------------------#"""

(defun enumerate2  (x   counter)

    (if (null x)
        nil
        (append   (list (cons  counter   (cons  (car x)  nil)))     (enumerate2  (cdr x)   (+  counter 1)  )   )
                ; (  (0 A)  )
    )
)



"""-------------------------------------------------------------------------------------------"""
"""------------------                   Exercise 4.27                   -----------------"""
"""-------------------------------------------------------------------------------------------"""


(defun find_x_position (x listx &optional path)

  (cond     ( (null listx)          nil)                 ; OR will skip the NILs
            ( (eq x listx)          (reverse path) )     ; You will either encounter NIL or X at the end of all path

            ( (listp listx)

                (or
```

```
                (find_x_position x (car listx) (cons 'CAR path))

                (find_x_position x (cdr listx) (cons 'CDR path))
                )
            )
        )
)


;;; The reason why this code works is that "or" searches for a "non-nil"
; most of this , bifurcations, or paths, will end up with nil.
; but, if at least one will reach a non-nil, it will return "path"



"""-----------------------------------------------------------------------------------------"""
"""------------------                   Exercise 4.28                   -----------------"""
"""-----------------------------------------------------------------------------------------"""

(defun nestedp   (x)

    (cond   ( (endp x)              nil)
            ( (listp (car x))       t)
            (t                      (nestedp  (cdr x)) )
    )
)


"""-----------------------------------------------------------------------------------------"""
"""------------------                   Exercise 4.29                   -----------------"""
"""-----------------------------------------------------------------------------------------"""

(defun flatten (x   storage)

    (cond   ( (endp  x)                     (reverse storage)  )
            ( (listp (car x))               (flatten   (append  (car x)  (cdr x) )  storage )  )
            ( t                             (flatten   (cdr x)    (cons (car x)  storage) )  )
    )
)


"""-----------------------------------------------------------------------------------------"""
"""------------------                   Exercise 4.30                   -----------------"""
"""-----------------------------------------------------------------------------------------"""

(defun range (x   storage)

    (cond   ( (eq x 0)              nil)
            ( (eq x 1)              (cons 0 storage) )
            ( t                     (range  (- x 1)  (cons (- x 1) storage) )  )
    )
)


(defun range-2  (x)

    (cond   ( (eq x 0)              nil)
            ( (eq x 1)              (list 0) )
            ( t                     (append  (range-2  (- x 1))   (list  (- x 1))  ) )
    )
)


"""-------------------------------------------------------------------------------------------"""
"""-------------------                  Exercise 4.31                   -------------------"""
"""-------------------------------------------------------------------------------------------"""


(defun sub-sequence (x   start   end    storage)

    (cond   ( (endp x)                                  storage  )
            ( (and (= start 0)(= end 0) )               (cons (car x ) storage) )
            ( (= start 0)                               (sub-sequence (cdr x)   start   (- end 1)   (cons (car x) storage) ) )
            ( t                                         (sub-sequence (cdr x)   (- start 1)   (- end 1)   storage ) )
    )
)


(defun sub-seq  (x  start  end )

    (reverse (sub-sequence x   start   end   nil) )
)

"""-------------------------------------------------------------------------------------------"""
"""-------------------                  Exercise 4.32                   -------------------"""
"""-------------------------------------------------------------------------------------------"""


(defun remove-2  (x   y  storage)     ; remove all x s inside the list y

    (cond   ( (endp y)                      (reverse storage) )
            ( (eq (car y) x)                (remove-2  x  (cdr y)  storage))
            ( t                             (remove-2  x  (cdr y)  (cons (car y) storage) ) )
    )
)


(defun remove-3  (x y storage)

    (cond   ( (endp y)                                  storage)
            ( (listp (car y))                           (remove-3  x  (cdr y)  (append storage (list (remove-2  x  (car y) nil ))  ) ) )
            ( (eq (car y) x)                            (remove-3  x  (cdr y)  storage) )
            ( t                                         (remove-3  x  (cdr y)  (append storage (list (car y)) ) ) )
    )
)
```

```lisp
"""-------------------------------------------------------------------------------------------"""
"""-------------------                  Exercise 4.33                  -------------------"""
"""-------------------------------------------------------------------------------------------"""


(defun produce (count_   max_  &optional (counter 0) (storage nil))

    (cond   ( (< counter count_)            (produce count_  max_ (+ counter 1) (cons (random max_)  storage) ) )
            ( t                             storage)
    )
)


"""-------------------------------------------------------------------------------------------"""
"""-------------------                  Exercise 4.34                  -------------------"""
"""-------------------------------------------------------------------------------------------"""

(defun rev   (x   &optional (storage  nil) )

    (cond   ( (endp x)              storage)
            ( t                     (rev  (cdr x)  (cons (car x)  storage) )  )
    )
)

"""-------------------------------------------------------------------------------------------"""
"""-------------------                  Exercise 4.35                  -------------------"""
"""-------------------------------------------------------------------------------------------"""

(defun rev2  (x  &optional (storage nil) )

    (cond   ( (endp x)                      storage)
            ( (listp (car x))               (rev2  (cdr x)  (append (list (rev (car x) ))   storage) ) )
            ( t                             (rev2  (cdr x)  (cons (car x) storage) ) )
    )
)

"""-------------------------------------------------------------------------------------------"""
"""-------------------                  Exercise 4.36                  -------------------"""
"""-------------------------------------------------------------------------------------------"""


(defun how-many   (x   y   &optional  (counter 0)   )

    (cond   ( (endp y)                      counter)
            ( (equal  (car y) x)            (how-many  x  (cdr y)  (+ counter 1) ) )
            ( t                             (how-many  x  (cdr y)  counter ) )
    )
)

(defun how-many2   (x   y )

    (cond   ( (endp y)                      0)
            ( (equal (car y)  x)            (+  1  (how-many2  x  (cdr y) ) ) )
            ( t                             (how-many2 x (cdr y) ) )
    )
)


"""-------------------------------------------------------------------------------------------"""
"""-------------------                  Exercise 4.37                  -------------------"""
"""-------------------------------------------------------------------------------------------"""


(defun d-how-many  (x   y   &optional  (counter  0 ) )

    (cond   ( (endp y)                      counter )
            ( (listp  (car y) )             (d-how-many  x  (append (car y)  (cdr y) )  counter) )  ; counter
            ( (equal (car y) x)             (d-how-many  x  (cdr y)   (+ counter 1) ) )
            ( t                             (d-how-many  x  (cdr y)   counter) )
    )
)

;;; VERY IMPORTANT : if you use &optional , you will be very careful. Because every time you forgot to enter a value
; it will enter the optional value (pre-determined value) (zero here). Therefore, nesting will be meaningless.
; It will not count.


(defun d-how-many-2   (x   y)

    (cond   ( (endp y)                      0)
            ( (listp (car y))               (d-how-many-2 x (append (car y)  (cdr y) ) ) )
            ( (equal (car y) x)             (+  1  (d-how-many-2 x (cdr y))  ))
            ( t                             (d-how-many-2 x (cdr y) ) )

    )
)


"""-------------------------------------------------------------------------------------------"""
"""-------------------                  Exercise 4.38                  -------------------"""
"""-------------------------------------------------------------------------------------------"""


(defun remove-   (x  z  y  counter storage)

    (cond   ( (endp y)                                          storage)
            ( (and (equal (car y) x)(= counter  z) )            (remove-  x  z  (cdr y)  1    storage)  )
            ( (equal (car y) x)                                 (remove-  x  z  (cdr y)  (+ counter 1)  (cons (car y) storage) ) )
            ( t                                                 (remove-  x  z  (cdr y)  counter      (cons (car y) storage) ) )
    )
)

(defun remove-nth  (x z y)

    (reverse (remove-  x  z  y  1  nil) )
)

"""-------------------------------------------------------------------------------------------"""
"""-------------------                  Exercise 4.39                  -------------------"""
```

```
"""-------------------------------------------------------------------------------------------"""


(defun subset_  ( car_x    y)

    (cond   ( (endp y)                           nil)
            ( (equal car_x  (car y))             t)
            ( t                                  (subset_ car_x  (cdr y)))
    )
)

(defun subset-p (x y)

    (cond   ( (endp x)                           t)
            ( (subset_ (car x) y)                (subset-p (cdr x) y) )
            ( t                                  (subset_ (car x) y))
    )
)

(defun equip  (x y)

    (cond   ( (endp x)                           nil)
            ( (subset-p (list (car x)) y)        t)
            ( t                                  (equip (cdr x) y))
    )
)

(defun idenp (x y)

    (cond   ( ( null x)                      t)
            ( (equal (car x) (car y))        (idenp (cdr x) (cdr y) ) )
            (t                               nil)
    )
)

"""-------------------------------------------------------------------------------------------"""
"""-------------------                  Exercise 4.40*                  -------------------"""
"""-------------------------------------------------------------------------------------------"""

; Define a procedure IMPLODE that takes a list of symbols and replaces the consequently
; repeating symbols with the symbol and the number of its repetitions.


(defun counter-list (x   storage   reader  counter)   ; initially  (storage = nil) and (reader = (car x) )  (counter = 0)

    (if (null x)
        (append storage (list counter))
        (if (equal  (car x)   reader)
            (counter-list   (cdr x)   storage reader (+ counter 1) )
            (counter-list   x   (append storage (list counter))   (car x)  0  )
        )
    )
)

; (counter-list '(a a b c c c d e e e )  nil  'a   0 )  will give   (2 1 3 1 3)  let's call it "list Y"


(defun unique-list  (x  storage  reader)    ; initially  (storage = nil) and (reader = (car x) )

    (if (null x)
        (append storage (list  reader) )
        (if (equal (car x)  reader )
            (unique-list  (cdr x)  storage  reader)
            (unique-list  x  (append storage (list  reader) )  (car x) )
        )
    )
)

; (unique-list '(a a b c c c d e e e ) nil 'a)  will give   (A B C D E)     let's call it "list X"


(defun merge-them  (x y storage)    ;  "list X"   and   "list Y"  ;  (A B C D E)   and   (2 1 3 1 3)

    (if (null x)
        storage
        (merge-them   (cdr x)   (cdr y)   (append storage   (cons (car x) (cons (car y) nil) ) ) )
    )
)


(defun implode  (x)

    (merge-them (unique-list x nil (car x)) (counter-list x nil (car x) 0)  nil)
)


;  (implode '(a a b c c c d e e e ) )      will give    (A 2 B 1 C 3 D 1 E 3)

"""-------------------------------------------------------------------------------------------"""
"""-------------------                  Exercise 4.41                   -------------------"""
"""-------------------------------------------------------------------------------------------"""

(defun explode_ (x  counter  storage)

    (cond   ( (endp x)                                   storage)
            ( (not (= counter (car (cdr x) )))           (explode_ x  (+ counter 1)  (append storage (list (car x)) ) ) )
            ( t                                          (explode_ (cdr (cdr x)) 0  storage) )
    )
)

(defun explode (x)

    (explode_ x  0  nil)
)


"""-------------------------------------------------------------------------------------------"""
"""-------------------                  Exercise 4.42                   -------------------"""
"""-------------------------------------------------------------------------------------------"""
```

```
(defun  zeros_  (x storage change)

    (cond   ( (and  (not (equal (car x) 0))  (equal change 0))          (zeros_ (cdr x)  storage change)  )
            ( (equal (car x) 0)                                         (zeros_ (cdr x) (cons '0 storage)   1) )
            ( t                                                         storage)
    )
)


(defun zeros (x)

    (-  (length (zeros_ x nil 0))   1)
)


"""------------------------------------------------------------------------------------------"""
"""-------------------                  Exercise 4.43                  -------------------"""
"""------------------------------------------------------------------------------------------"""


(defun remafter_ (x   y   pivot  storage)

    (cond   ( (null y)                                  storage)
            ( (not (equal (car y) pivot))       (remafter_ x (cdr y) pivot (cons (car y) storage)))
            ( (equal (car (cdr y)) x)           (remafter_ x (cdr (cdr y)) pivot (cons (car y) storage)))
            ( t                                 (remafter_ x (cdr y)   pivot  (cons (car y) storage)))
    )
)

(defun remafter (x y pivot)

    (reverse (remafter_ x y pivot nil) )
)


"""------------------------------------------------------------------------------------------"""
"""-------------------                  Exercise 4.44                  -------------------"""
"""------------------------------------------------------------------------------------------"""


(defun run-mean_ (x   storage  mean   counter)

    (cond   ( ( null x)             (reverse storage) )
            ( t                     (run-mean_  (cdr x)  (cons   (/   (+ (car x) (* mean  (- counter 1)) )   counter)  storage)
                                     (/   (+ (car x) (* mean  (- counter 1)) )   counter)  (+ counter 1) ) )
    )
)


(defun run-mean  (x)

    (run-mean_ x nil 0 1)
)



(defun run-mean_2 (x storage mean counter)

    (dotimes    (i     (length x)    (reverse storage)  )

            (let    ( (new-mean         (/ (+ (nth i x) (* mean (- counter 1))) counter)) )

                    (setf storage       (cons new-mean storage))
                    (setf mean          new-mean)
                    (setf counter       (1+ counter)))
        )

)

(defun run-mean2 (x)
  (run-mean_2 x nil 0 1)
)


"""------------------------------------------------------------------------------------------"""
"""-------------------                  Exercise 4.45                  -------------------"""
"""------------------------------------------------------------------------------------------"""


    (defun give_first  ( x  storage)

        (cond   ( (endp x)                              (reverse (cons (car x) storage) ) )
                ( (endp (cdr x))                        (reverse (cons (car x) storage) ) )
                ( (<= (car x) (car (cdr x)) )           (give_first (cdr x) (cons (car x) storage)) )
                ( t                                     (reverse (cons (car x) storage) ) )
        )
    )
    ; (give_first '(12 13 14   1 2 3 4 5 6   9 8 7) nil)    will return     (12 13 14)


    (defun give_remain  (g_f  x )

        (cond   ( (endp g_f)            x)
                ( (equal (car g_f) (car x))             (give_remain (cdr g_f)  (cdr x) ) )
                ( t                                     (and (print "ERROR ! Two lists are different!") t))
        )
    )
    ; (give_remain '(12 13 14 ) '(12 13 14   1 2 3 4 5 6 9   8 7) )     will return     (1 2 3 4 5 6 9 8 7)

    (defun main_  (x   storage)

        (let*   (                                       ; let* works "sequential" which means that you can use assigned values later
                ( first_            (give_first  x  nil))
                ( remain_           (give_remain first_ x))
                )
                (cond   ( (endp remain)                                 storage)
                        ( (< (length storage) (length first))           (main_  remain  first) )
```

```
                    ( t                                            (main_  remain  storage) )
            )
        )
    )


    (defun    main  (x)

        (main_  x  (give_first  x  nil) )
    )


"""-------------------------------------------------------------------------------------------"""
"""-------------------                  Make it unique                  -------------------"""
"""-------------------------------------------------------------------------------------------"""

(defun  uniq    (lst)       ; (a      b c a d )

        (if lst
            (if (member (car lst) (cdr lst))
                (uniq (cdr lst))
                (cons (car lst)  (uniq (cdr lst)))
            )
            nil

        )
)


(defun  uniq2   (lst    &optional (acc nil))  ; (a      b c a d )     acc = nil


    (if     lst
            (uniq2 (cdr lst)    (if (member (car lst) acc)
                                    acc
                                    (append acc (list (car lst )))
                                )
            )
            acc
    )
    )


"""-------------------------------------------------------------------------------------------"""
"""-------------------                  Exercise 4.46                   -------------------"""
"""-------------------------------------------------------------------------------------------"""



    (defun give_first  ( x  storage)

        (cond   ( (endp x)                              (reverse (cons (car x) storage) ) )
                ( (endp (cdr x))                        (reverse (cons (car x) storage) ) )
                ( (<= (car x) (car (cdr x)) )           (give_first (cdr x) (cons (car x) storage)) )
                ( t                                     (reverse (cons (car x) storage) ) )
        )
    )
    ; (give_first '(12 13 14   1 2 3 4 5 6   9 8 7) nil)    will return     (12 13 14)



    (defun  give_first_sum (x)

        (if     (endp x)
                0
                (+ (car x)  (give_first_sum (cdr x)))

        )
    )
    ; (give_first_sum (give_first '(12 13 14 1 2 3 4 5 6) nil) )    will return     39



    (defun give_remain  (g_f  x )

        (cond   ( (endp g_f)            x)
                ( (equal (car g_f) (car x))             (give_remain (cdr g_f)  (cdr x) ) )
                ( t                                     (and (print "ERROR ! Two lists are different!") t))
        )
    )

; (give_remain '(12 13 14 ) '(12 13 14   1 2 3 4 5 6 9   8 7) )     will return     (1 2 3 4 5 6 9 8 7)



(defun main  (x)

    (let*   (( first_                   (give_first   x    nil))             ; (12 13 14)
            ( first_sum                 (give_first_sum   first_))          ; 39
            ( remain_                   (give_remain      first_  x))       ; (1 2 3 4 5 6 9 8 7)
            )

            (if     (endp remain_)
                    first_sum
                    (max    first_sum   (main remain_))
            )
    )
)




; (main  '(12 13 14   1 2 3 4 5 6 9 10 11   0 177    50 50 50 50   ))   will return       200


"""-------------------------------------------------------------------------------------------"""
"""-------------------                  Exercise 4.47                   -------------------"""
"""-------------------------------------------------------------------------------------------"""
```

```lisp
    (defun give_first  ( x  storage)

        (cond   ( (endp x)                              (reverse (cons (car x) storage) ) )
                ( (endp (cdr x))                        (reverse (cons (car x) storage) ) )
                ( (<= (car x) (car (cdr x)) )           (give_first (cdr x) (cons (car x) storage)) )
                ( t                                     (reverse (cons (car x) storage) ) )
        )
    )
    ; (give_first '(12 13 14   1 2 3 4 5 6   9 8 7) nil)    will return     (12 13 14)


    (defun  give_first_sum (x)

        (if     (endp x)
                0
                (+ (car x)  (give_first_sum (cdr x)))

        )
    )
    ; (give_first_sum (give_first '(12 13 14 1 2 3 4 5 6) nil) )    will return     39


    (defun give_remain  (g_f  x )

        (cond   ( (endp g_f)            x)
                ( (equal (car g_f) (car x))             (give_remain (cdr g_f)  (cdr x) ) )
                ( t                                     (and (print "ERROR ! Two lists are different!") t))
        )
    )

; (give_remain '(12 13 14 ) '(12 13 14   1 2 3 4 5 6 9   8 7) )     will return     (1 2 3 4 5 6 9 8 7)


(defun main_  (x   sum_   storage)

    (let*   (( first_                   (give_first   x   nil))             ; (12 13 14)
            ( first_sum                 (give_first_sum   first_))          ; 39
            ( remain_                   (give_remain      first_  x))       ; (1 2 3 4 5 6 9 8 7)
            )

            (if     (endp remain_)
                    (if     (< first_sum   sum_)
                            storage
                            first_
                    )
                    (main_  remain_  (max sum_ first_sum)  (if  (< first_sum   sum_)
                                                                storage
                                                                first_
                                                                    )
                    )
            )

    )
)


(defun main     (x)

    (main_  x  0   nil)

    )


; (main  '(12 13 14   1 2 3 4 5 6 9 10 11   0 177    50 50 50 50   ))    will return   (50 50 50 50)


"""-------------------------------------------------------------------------------------------"""
"""-------------------                 Exercise 4.48                 -------------------"""
"""-------------------------------------------------------------------------------------------"""


(defun  pairlist  (x   list1  list2)


    (cond   ( (endp x)                  (cons (reverse list1)  (list (reverse list2))))
            ( t                         (pairlist  (cdr x)  (cons (caar x) list1)   (cons (cadar x) list2)))

    )
)

; (pairlist '( (a b) (= =) (1 2) (+ -) (3 9) )  nil nil)    will return     ((A = 1 + 3) (B = 2 - 9))


"""-------------------------------------------------------------------------------------------"""
"""-------------------                 Exercise 4.49                 -------------------"""
"""-------------------------------------------------------------------------------------------"""


(defun get_first_n_elm   (x   n  storage)                 ; n   =   index + 1

        (if     (= n 0)
                (reverse storage)
                (get_first_n_elm  (cdr x)  (- n 1 )  (cons (car x) storage))
        )
)
    ; (get_first_n_elm '(a b c d e f g h )  5  nil)   will give    (A B C D E)


(defun  search_pos_    (x  y  index_storage  index)

        (cond   ( (endp y)                                                   index_storage)
                ( (equal x  (get_first_n_elm y  (length x)  nil))            (search_pos_ x  (cdr y) (cons index index_storage) (+ index 1)))
                ( t                                                          (search_pos_ x  (cdr y)  index_storage  (+ index 1)))
```

```
        )
)


(defun   search_pos   (x  y)

    (search_pos_ x y nil 0)
)


"""-------------------------------------------------------------------------------------------"""
"""-------------------                  Exercise 4.50                   -------------------"""
"""-------------------------------------------------------------------------------------------"""


(defun      last2   (x)

    (cond   ( (endp (cdr x))                    (car x))
            ( t                                 (last2 (cdr x)))
    )
)


"""-------------------------------------------------------------------------------------------"""
"""-------------------                  Exercise 4.51                   -------------------"""
"""-------------------------------------------------------------------------------------------"""


(defun  chop_last   (x   storage)

    (cond   ( (endp x)                              nil)
            ( (endp (cdr x))                        (reverse storage))
            ( t                                     (chop_last (cdr x)  (cons (car x) storage)))
        )
)

"""-------------------------------------------------------------------------------------------"""
"""-------------------                  Exercise 4.52                   -------------------"""
"""-------------------------------------------------------------------------------------------"""

(defun last1  (x    last_element)

    (if (endp x)
        last_element
        (last1  (cdr x)   (car x) )
    )
)
; (last1 '(a b c d ) nil)       will give   D


(defun  chop_last   (x   storage)

    (cond   ( (endp x)                              nil)
            ( (endp (cdr x))                        (reverse storage))
            ( t                                     (chop_last (cdr x)  (cons (car x) storage)))
        )
)
; (chop_last '(a b c d x) nil)      will give       (A B C D)


(defun  palindrome  (x)         ; (a b a)

    (cond   ( (endp x)                                      t)
            ( (equal (car x)  (last1 x nil))                (palindrome (chop_last (cdr x) nil))) ; send without car
            ( t                                             nil)

        )
)


"""-------------------------------------------------------------------------------------------"""
"""-------------------                  Exercise 4.53                   -------------------"""
"""-------------------------------------------------------------------------------------------"""


(defun n_th  ( x  n index)   ;  (a b X d e f )   2.th  X

    (cond   ( (endp x)                      nil)
            ( (= n index)                   (car x))
            ( t                             (n_th (cdr x) n (+ index 1)))
    )
)


"""-------------------------------------------------------------------------------------------"""
"""-------------------                  Exercise 4.54                   -------------------"""
"""-------------------------------------------------------------------------------------------"""





"""-------------------------------------------------------------------------------------------"""
"""-------------------                  Exercise 4.55                   -------------------"""
"""-------------------------------------------------------------------------------------------"""


...
```