

```

""#-----#""
""#-----METU Cognitive Sciences-----#""
""#-----Symbols & Programming-----#""
""#-----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-) (A B C D)

```
(cons 'A (cons 'B (cons 'C (cons 'D nil) ) ) )
```

; ( A B C D)

; if you forget to use ' then it will give error "unbound variable" since  
; it does not know these symbols

```
;;;;;;;;;;;;;
```

```
(cons (cons 'a nil) nil)
```

; this is ((A))  
; because it does come from CAR not CDR.  
; like (cons something nil) -> (something) -> ((a))

```
(cons nil (cons 'a nil))
```

; will return (NIL A)

```
;;;;;;;;;;;;;
```

```
(cons '(c d) nil)
```

; ((C D))

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

; (A (C D))

```
;;;;;;;;;;;;;
```

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

```
(cons 'A (cons (cons 'B (cons '(c d) nil) ) nil) )
```

;

```
;;;;;;;;;;;;;
```

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

```
(cons 'A (cons (cons 'B (cons (cons '(C) nil) (cons 'D nil))))
```

; to avoid dot "." you have to extent to the "nil" or

```
(cons 'A '((B (C) D))) ; this seems to be a joke. Be serious !
```

; (A (B (C) D))

```
;;;;;;;;;;;;;
```

; d-)

```
( ( (A (B (C) D) ) ) )
```

; assume '(C) = X , then we need (B X D)  
; assume (B X D) = Y , then we need (A Y)  
; assume (A Y) = Z , then we need ((Z))

```
set : (cons (cons 'Z nil) nil) == ((Z))
```

```
set : (cons 'A (cons 'Y nil)) == (A Y)
```

```
set : (cons 'B (cons 'X (cons 'D nil))) == Y
```

```
set : (cons 'C nil) == X
```

Substitute Z with (A Y)

```
now we have : (cons (cons (cons 'A (cons 'Y nil)) nil) nil)
```

Substitute Y with (B X D)

```
now we have : (cons (cons (cons 'A (cons (cons 'B (cons 'X (cons 'D nil))) nil)) nil) nil)
```

Now, Substitute X :

```
now we have : (cons (cons (cons 'A (cons (cons 'B (cons (cons 'C nil) (cons 'D nil))) nil)) nil) nil)
```

```
(cons (cons (cons 'A (cons (cons 'B (cons (cons 'C nil) (cons 'D nil))) nil)) nil) nil)
```

```
; output : (((A (B (C D))))
```

```
* (equal '(((A (B (C D)))) '(((A (B (C D)))) )
```

T

; A lot of work: but guarantee result. No pain, no gain !

```
""-----""
""-----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 lst – the first one is answered to  
; clarify the question:

```
(a x b d)      ->      (car (cdr lst))

(a b x d)      ->      (car (cdr (cdr '(a b x d))))      ->      (caddr '(a b x d))

(a (b (x d)))  ->      (cdr lst)                        =      ( (b sth) ) ,
                    (car (cdr lst))                    =      (b sth) ,
                    (cdr (car (cdr lst))) = (sth)        =      ( (X Y) )
                    (car (cdr (car (cdr lst)))) = (X Y)
                    (car (car (cdr (car (cdr lst))))) = X
                    (car (car (cdr (car (cdr '(a (b (x d))))))))

(a (b (d) x))  ->      (a (b sth x))      =      lst
                    ( (b sth x) )      =      (cdr lst)
                    (b sth x)          =      (car (cdr lst))
                    (sth x)            =      (cdr (car (cdr lst)))
                    (x)                =      (cdr (cdr (car (cdr lst))))
                    x                  =      (car (cdr (cdr (car (cdr lst)))))
                    (car (cdr (cdr (car (cdr '(a (b (d) x)))))))

(( (a (b (x) d)) )) ->      ((sth))      =      lst
                    (sth)      =      (car lst)
                    sth        =      (car (car lst))
                    ( (b (x) d) ) =      (cdr (car (car lst)))
                    (b (x) d)    =      (car (cdr (car (car lst))))
                    ((x) d)      =      (cdr (car (cdr (car (car lst)))))
                    (x)          =      (car (cdr (car (cdr (car (car lst)))))
                    x            =      (car (car (cdr (car (cdr (car (car lst)))))
                    (car (car (cdr (car (cdr (car (car '(a (b (x) d))))))))))
```

```
""-----""
""-----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)      CDR = ( (C D) (E F) )
(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 answer : (A B)")
  (print "-----")
  (print (cdr (cdr list1)) )
  (print "My answer : ((E F))")
  (print "-----")
  (print (cdr (car list1)) )
  (print "My answer : (B)")
  (print "-----")
  (print (car (cdr list1)))
  (print "My answer : (C D)")
  (print "-----")
  (print (car (cdr (cdr list1))) )
  (print "My answer : (E F)")
  (print "-----")
  (print (cdr (cdr (car (cdr list1)))))
  (print "My answer : 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?

(X Y Z)

car = X = (A B) cdr = (Y Z)

car = "A" cdr = (B) car = Y = (C D) cdr = (Z)

car = "B" cdr = nil car = (E F) cdr = nil

car = E cdr = (F)

```
car = "F" cdr = nil
```

follow paths to find the sequences of car and cdr :

```
(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 trying 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 cons 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) puts outputs/returned elements into a list
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 , there is no function that starts with a quote, I guess,
19. (+ 2 '17) -> 19 -> VERY IMPORTANT !!!
```

Because : \* (numberp '19) returns T (Best prime, ever, 19)

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

```
20. ('+ 1 4) -> ERROR GRE, no such function,
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-----##""
""#-----##""
```

Write down what the following expressions evaluate to

work them out before trying on the computer : (Roger that)

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

```
(if (listp (list 1 2))
    'it-is-not-fun-hocam
    'not-really
)
```

2.

```

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

(nil) will give error... GRE ! (Graduate Record Examinations !)

if no quote, can not pass the exam !

(null '() ) will give TRUE, because it is null (nothing inside)

(null '(nil) ) = (null '(())) will give NIL, it has an element : nil

Difference between (null) and (endp) is (endp) gives error if it is not a list . However,

* (null 'a )

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 sth) . 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)) )

Returns "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 countE0 (x &key (even-count 0) (odd-count 0) )

  (if (= x 1)

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

      (if (evenp x)
          (countE0 (collatz x) :even-count (+ even-count 1) :odd-count odd-count)
          (countE0 (collatz x) :even-count even-count :odd-count (+ odd-count 1) )
        )
    )
  )

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

Define a procedure named INSERT-2ND, which takes a list and an object, and gives
back a list where the element is inserted after the first element of the given list. As-
sume that the input list will have at least one element. Here is a sample interaction:

( insert-2nd '( b k ) '( a c ) ) -> (A ( B K ) C )

(defun insert-2nd (x y)

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

(defun insert-2nd (x y)

  (cons (car y) (cons x (cdr y))))

(defun insert-2nd (x y)

  (append (list (car y)) (list x) (cdr y) ))

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

Define a procedure named REPLACE-2ND, which is like INSERT-2ND, but replaces
the element at the 2nd position. Assume that the input list will always have at least
two elements.

```

```

(defun replace-2nd (x y)
  (cons (car y) (cons x (cdr (cdr y))) ) )

* (replace-2nd '(x y) '(a b c) )

(A (X Y) C)

(defun replace-2nd (x y)
  (append (list (car y)) (list x) (cdr (cdr y)) ) )

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

```

Define a procedure SWAP, that takes a two element list and switches the order of the elements. You are allowed to use only CAR, CDR, CONS and NIL as built-ins.

```

(a b) -> (b a)

(defun swap (y)
  (cons (car (cdr y)) (cons (car y) nil)))

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

```

Define a procedure that takes a list and an object, and returns a list where the object is added to the end of the list.

```

'x , '(a b c) -> '(a b c x)

(defun append1 (x y)
  (append y (cons x nil) ) )

(defun append1 (x y)
  (append y (list x) ) )

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

```

Define your own procedure APPEND2 that appends two list arguments (I guess: two lists' arguments) into a third list. You are not allowed to use APPEND, LIST and REVERSE – use just CONS.

```

'(a b) '(c d) -> '(a b c d)

(defun append2 (x y)
  (cons (car x) (cons (car (cdr x)) y) ) )

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

```

Using CAR and CDR, define a procedure to return the fourth element of a list.

```

(defun list-4th (x) (caddr x) )

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

```

Define a procedure AFTER-FIRST that takes two lists and inserts all the elements in the second list after the first element of the first list.

; Given (A D E) and (B C), it should return (A B C D E).

```

(defun after-first (x y)
  (cons (car x) (append y (cdr x) )))

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

```

Define a procedure AFTER-NTH that takes two lists and an index. It inserts all the elements in the second list after the given index of the first list. Indices start with 0.

; Given (A D E), (B C), and 1, it should return (A D B C E)

```

(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)
  (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)
```

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

; Given (A D E), (B C), and 1, it should return (A D B C E)

```
(defun helper_14 (lst index storage)
  (if (= index 0)
      (append (list (reverse (cons (car lst) storage))) (list (cdr lst)))
      (helper_14 (cdr lst) (- index 1) (cons (car lst) storage))
  )
)
```

This will give ((A D) (E))

```
(defun func14 (lst1 lst2 index)
  (let ((divided (helper_14 lst1 index nil)) )
    (append (car divided) lst2 (car (cdr divided)) ))
)
```

\* (func14 '(a b c) '(d e f) 1)

(A B D E F C)

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

Assume you have data that pairs employees' last names with their monthly salaries. E.g. ((SMITH 3000) (JOHNS 2700) (CURRY 4200)) Define a procedure that takes as input employee data and a threshold salary (an integer), and returns in a list the last names of all the employees that earn above the threshold salary. Define two versions, one with, and one without an accumulator.

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

```
(defun above-TH2 (x th)
  (if (null x)
      nil
      (if (>= (cadar x) th)
          (cons (caar x) (above-TH2 (cdr x) th))
          (above-TH2 (cdr x) th)
      )
  )
)
```

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

Using MEMBER and LENGTH, write a function ORDER which gives the order of an item in a list. You can do this by combining LENGTH and MEMBER in a certain way. It should behave as follows:

```
* (order 'c '(a b c))
3
* (order 'z '(a b c))
NIL
```

```
(defun order (x y)
  (if (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-----""

Define a procedure that computes the sum of a list of numbers with and without an accumulator. Consider that there might be non-number elements in a list, which t you should ignore in your summation.

```
(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 -----""
""-----""

```

Define a procedure that returns the largest number in a list of numbers. Do not use the built-in MAX.

```

(defun find_max (x max_value) ; initially zero x = (1 a 3 5 7)

  (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) ; (1 2 3 4 5)
        (car y)
        (if (> (car y) (cadr y) )
            (find_max_2 (cons (car y) (cddr y) )) ; I dont take the small value, get rid of them.
            (find_max_2 (cons (cadr y) (cddr y))) ; Perfect code ! Neither accumulator nor nesting.
        )
    )
  )

  ; In the end, (car y) is the biggest one !

)

""-----""
""----- Exercise 4."19" -----""
""-----""

```

Define a procedure that takes a list of integers and returns the second largest integer in the list.

```

(defun sec_largest (x sec_val max_val) ; initially 0 and 1 ("19" 23 17) (23 17 "19") (17 "19" 23)

  (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)
              (sec_largest (cdr x) sec_val max_val )
          )
      )
  )
)

""----- Second way -----""

(defun sec_largest_2 (lst storage)

  (let ((max_val (apply #'max lst)) ; (3 5 7 "19" 23 11 7)
        (car_ (car lst))
        (cdr_ (cdr lst))
        )
      ; *****

      (cond ((equal car_ max_val) (apply #'max (append storage cdr_)) ) ; get rid of max, then take the new max
            (t (sec_largest_2 cdr_ (cons car_ storage))) ; Great !
            )
  )
)

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

```

Define a procedure that takes a list of integers and an integer n, and returns the nth largest integer in the list.

```

'(1 2 3 4 5) 4 -> 4

```

```

(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) 9999999999999999 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) )
  )
)

""" ----- Second way ----- """
'(5 4 "3" 2 1) 3 -> 3

(defun order_ (lst storage) ; *****
  (if (null lst)
      (reverse storage)

      (let ((max_val (apply #'max lst))
            (car_ (car lst))
            (cdr_ (cdr lst))
            )
        (cond
          ((equal car_ max_val) (order_cdr_ (cons car_ storage))) ; if found, change the place of it
          (t (order_ (append cdr_ (list car_)) storage)) ; if not, send it to the back of the line
        )
        ; You are the best !
      )
  )
)

(defun nth_largest_3 (lst n)
  (nth (- n 1) (order_ lst nil) )
)

""" ----- Exercise 4.21 ----- """
Define a procedure that gives the last element of a list or gives NIL if the list is
empty. Name your procedure LASTT in order not to clash with LISP's built-in LAST.

(defun last1 (x last_element) ; (1 2 3 x 19)
  (if (null x)
      last_element
      (last1 (cdr x) (car x) )
  )
)

""" ----- Second way ----- """

```



```

(defun last2 (lst) (car (reverse lst)))
(defun last3 (lst) (nth (- (length lst) 1) lst))
; We will force it man ! Consider every possibility.
(defun last4 (lst)
  (if (null (cdr lst))
      (car lst)
      (last4 (cdr lst))))

```

-----

Exercise 4.22

-----

Define a procedure MULTI-MEMBER that checks if its first argument occurs more than once in the second.

```

'x '(a b (c x) x d e)

(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) )
            )
        )
    )
  )

""" ----- Second way ----- """

Now count them : 'x '(a b (c x) x d e)

(defun multi-member_c (x y counter)
  (if (null y)
      counter
      (if (listp (car y))
          (multi-member_c x (append (car y) (cdr y) ) counter )
          (if (equal x (car y))
              (multi-member_c x (cdr y) (+ counter 1) )
              (multi-member_c x (cdr y) counter)
            )
        )
    )
  )

""" ----- Third way ----- """

(defun multi-member_c2 (x y counter)
  (cond ( (null y) counter)
        ( (listp (car y)) (multi-member_c2 x (append (car y) (cdr y) ) counter ))
        ( (equal x (car y)) (multi-member_c2 x (cdr y) (+ counter 1) ))
        ( t (multi-member_c2 x (cdr y) counter) )
  )
)

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

```

Define a recursive member procedure that checks whether a given item is found in the given list. The item is not required to be a top-most element. Some sample interactions are as follows:

```

* (rec-mem 'a '(b (z ("a" x) k) c))
T

(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:  
 ; (count 1 '(1 2 3 1 1) ) will give 3

```

""" ----- Second way ----- """

(defun flat_it (lst storage)
  (cond ( (null lst) storage)
        ( (listp (car lst)) (flat_it (append (car lst) (cdr lst) ) storage ) )
        ( t (flat_it (cdr lst) (cons (car lst) storage)) )
  )
)

(defun rec-mem2_ (x lst)
  (cond ( (null lst) 0)
        ( (equal (car lst) x) (+ 1 (rec-mem2_ x (cdr lst))) )
        ( t (rec-mem2_ x (cdr lst)) )
  )
)

(defun rec-mem2 (x lst) (rec-mem2_ x (flat_it lst nil)))

```

```

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

```

Define a procedure LEVEL, that takes an element X and a list LST, and returns the level of depth that X is found in LST. If X is not a member, your procedure will return NIL. Top level counts as 0, every level of nesting adds 1 to the depth. Sample interaction:

```

* (level 'a '(b a c))
0

* (level 'a '(b (z (a x) k) c))
2

(defun level (x y depth)

  (cond
    ( (endp y) nil)
    ( (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) ) )
  )
)

```

```

"""-----Second way-----"""

```

```

; (level2 'a '(b (z (a x) k) c) 0) ; ***** One of the Best !

(defun level2 (x lst counter)

  (cond
    ( (null lst) nil)
    ( (equal lst x) counter)
    ( (and (not (listp lst)) (not (equal lst x))) nil)
    ( (listp lst) ; Very important ! Outputs can be either a non-nil (counter) or NIL
      (or ; OR always searches for a non-nil. Set "what you dont need" as NIL values
        (level2 x (car lst) counter)
        (level2 x (cdr lst) (+ counter 1))
      )
    )
  )
)

```

```

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

```

Define a procedure that converts a binary number (given as a list of 0s and 1s) to decimal, without checking the length of the input.

(1 0 1) ->  $2^0 \times 1 + 2^1 \times 0 + 2^2 \times 1 = 1 + 0 + 4 = 5$

```

(defun get_reverse (x storage)

  (if (endp x)
      storage
      (get_reverse (cdr x) (cons (car x) storage)))
  )
; ***** One of the Best Questions !

```

(1 0 1 0) -> (0 1 0 1)

```

(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 ( ( reverse_ (get_reverse x nil) ) )

    (binary_to_decimal reverse_ 0 0 )
  )
)

```

```

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

```

Define a procedure ENUMERATE that enumerates a list of items. Numeration starts with 0. Define two versions, one with, and one without an accumulator.

```

(enumerate '( A B C ))

((0 A) (1 B) (2 C))

(enumerate NIL)

```

NIL

```

(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) )
  )
)

```

```
""#----- Third Way -----#""
```

```
( enumerate '( A B C ))

((0 A ) (1 B ) (2 C ))

(defun enumerate3 (lst counter storage)

  (if    (= counter (length lst))
    (reverse storage)
    (enumerate3 lst (+ counter 1) (cons (cons counter (cons (nth counter lst) nil)) storage) )

  )

)
```

```
""-----#""
""----- Exercise 4.27 -----#""
""-----#""
```

Given a possibly nested list of symbols one and only one of which will be the symbol X, compute the steps of CARs and CDRs required to get X from the list.

```
CL-USER > ( foo (( a ( z x d )) ( c s d )))

( CAR CDR CAR CDR CAR )

(defun find_x_position (x listx &optional path)      ; if you dont specify nil here, instead if you will do below inside car and cdr
                                                    ; it will go into infinite loop
  (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"

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

```
(defun find_x_position2 (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
              ( (and (not (listp listx) ) (not (eq x listx))) nil)

              ( t

                (or

                  (find_x_position2 x (car listx) (cons 'CAR path))

                  (find_x_position2 x (cdr listx) (cons 'CDR path))

                )

              )

  )

)
```

```
""-----#""
""----- Exercise 4.28 -----#""
""-----#""
```

Define a procedure NESTEDP that takes a list and returns T if at least one of its elements is a list, and returns NIL otherwise.

```
* (nestedp '( a b (c) d e) )
```

```
(defun nestedp (x)

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

  )

)
```

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

```
(defun nestedp2 (x &key (path 'cdr_))

  (cond      ( (null x)          nil)              ; do not use ENDP.
              ( (and (equal path 'car_) (listp x)) t)
              ( (listp x)

                (or

                  (nestedp2 (car x) :path 'car_)
                  (nestedp2 (cdr x) :path 'cdr_))

                )

  )

)
```

```
""-----#""
""----- Exercise 4.29 -----#""
""-----#""
```

Define a recursive function FLATTEN, which takes a possibly nested list and returns a version where all nesting is eliminated. E.g. ((1 (2) 3) 4 (((5) 6) 7)) should be returned as (1 2 3 4 5 6 7).

```
(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 -----
"""-----

```

Write a program named RANGE, that takes a non-negative integer N as argument and returns a list of non-negative integers that are less than N in increasing order. Here is a sample interaction with the first four non-negative integers, your solution must work for all non-negative integers:

```

(range 0)  ->  NIL
(range 1)  ->  (0)
(range 3)  ->  (0 1 2)

```

```

(defun range (x storage)

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

```

```

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

```

```

(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 -----
"""-----

```

Write a program that takes a sequence, a start index, an end index and returns the sub-sequence from start to (and including) end. Indices start from 0.

```

'(( a b c   d e f   g h )   3 5 nil)  ->  (d e f)

```

```

(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) )
)

```

```

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

```

```

'(( a b c   d e f   g h )   3 5 nil)  ->  (d e f)

```

```

(defun sub-seq2 (x start end storage )

  (if (= start end)
      (reverse (cons (nth start x) storage))
      (sub-seq2 x (+ start 1) end (cons (nth start x) storage) )
  )
)

```

```

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

```

Define a procedure REMOVE2 that takes an element and a list, and returns a list where all the occurrences of the element are removed from the list.

```

(defun remove-2 (x y storage) ; (a b x c x) 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) ; (a b (c d x (x v) v ) h)

  (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)) ) ) )
  )
)

```

```

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

```

```

(defun remove-4 (x y storage) ; ( (a) b (c d x (x v) v ) h)

  (cond ( (endp y)                storage)
        ( (listp (car y))         (remove-4 x (append (car y) (cdr y)) storage))
        ( (eq (car y) x)          (remove-4 x (cdr y) storage) )
        ( t                      (remove-4 x (cdr y) (append storage (list (car y)) ) ) )
  )
)

```

```

"""-----"""
"""Exercise 4.33-----"""
"""-----"""

```

Write a program that takes two parameters count and max, and returns a list of count random integers, all less than max.

```

(defun produce (count_ max_ &optional (counter 0) (storage nil))
  (cond ((< counter count_)
        (produce count_ max_ (+ counter 1) (cons (random max_) storage) ) )
    (t storage)
  )
)

```

```

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

```

```

(defun produce2 (count_ max_)
  (let ((result nil))
    (dotimes (i count_ result)
      (setf result (cons (random max_) result))
    )
  )
)

```

```

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

```

The built-in REVERSE reverses a list. Define your own version of reverse.

```

(defun rev (x &optional (storage nil) )
  (cond ((endp x) storage)
        (t (rev (cdr x) (cons (car x) storage) ) )
  )
)

```

```

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

```

```

(defun rev2 (lst)
  (let ((result nil))
    (dotimes (i (length lst) result)
      (setf result (cons (nth i lst) result))
    )
  )
)

```

We will force it man ! We will force it. There is no border for us.

```

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

```

In Ex 4.34 you defined a list reversing procedure. Now alter that definition so that it not only reverses the order of the top-level elements in the list but also reverses any members which are themselves lists.

Yes Sir. We can also reverse the rotation of the earth, if you want !

```

'( ( a b) ((c (a b c) b) c) (a) )

```

```

(append (func cdr) + (func car)

```

```

(defun rev3 (x)
  (cond ((null x) nil)
        ((not (listp x)) (list x))
        (t (listp (car x)) (append (rev3 (cdr x)) (list (rev3 (car x))) ) )
        (t (append (rev3 (cdr x)) (rev3 (car x))) )
  )
)

```

check !

```

* (rev3 '( a b c (a b c (a b c) a b c) a b c) )

```

```

(C B A (C B A (C B A) C B A) C B A)

```

We will force it ...

```

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

```

```

(defun rev4 (x storage)
  (cond ((null x) storage)
        ((not (listp x)) x)
        (t (listp (car x)) (rev4 (cdr x) (cons (rev4 (car x) nil) storage)))
        (t (rev4 (cdr x) (cons (car x) storage)))
  )
)

```

```

* (rev4 '( a b c (a b c (a b c) a b c) a b c) nil)

```

```

(C B A (C B A (C B A) C B A) C B A)

```

```

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

```

Define a procedure HOW-MANY? that counts the top-level occurrences of an item in a list.

```
(how-many 'a '(a b r a c a d a b r a))
```

5

```
(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 ) )
  )
)
```

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

```
* (how-many2 'a '(a b r a c a d a b r a))
```

5

```
(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) ) )
  )
)
```

```
""#----- Third Way for nested -----#""
```

```
* (how-many3 'a '(a b r (a c (a) d a) b r a))
```

5

```
(defun how-many3 (x y)
  (cond ((endp y) 0)
        ((listp (car y)) (how-many3 x (append (car y) (cdr y))))
        ((equal (car y) x) (+ 1 (how-many3 x (cdr y) ) ) )
        (t (how-many3 x (cdr y) ) )
  )
)
```

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

Define a recursive procedure **D-HOW-MANY?** that counts all – **not** only top-level – occurrences of an item in a **list**.

For instance (D-HOW-MANY? 'A '((A B) (C (A X)) A)) should return 3.

```
(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.

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

```
(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 -----""
""-----""
```

Define a three argument procedure **REMOVE-NTH**, which removes **every nth** occurrence of an item from a **list**.

```
'x 2 '( a x X b b x c X d d x X ) -> '( a x _ b b x c _ d d x _ )
```

```
(defun remove- (x nt lst counter storage)
  (cond ((endp lst) storage)
        ((and (equal (car lst) x)(= counter nt) ) (remove- x nt (cdr lst) 1 storage) )
        ((equal (car lst) x) (remove- x nt (cdr lst) (+ counter 1) (cons (car lst) storage) ) )
        (t (remove- x nt (cdr lst) counter (cons (car lst) storage) ) )
  )
)
```

```
(defun remove-nth (x nt lst)
  (reverse (remove- x nt lst 1 nil) )
)
```

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

```
'x 3 '( a x x X b b x x c X d d x x X ) -> '( a x x _ b b x x c _ d d x x _ )
```

```
(defun remove-2 (x nt lst)
```

```
  (let ((counter 0)
        (storage nil))
```

Everything is about "ALL" probabilities ! Consider ALL.

A given *set* A is a subset of another *set* B if and only if all the members of A are also a *member* of B. Two sets are equivalent, if and only if they are subsets of each other. For this problem you will represent sets via lists.

$$\vdash (a \ b) \quad \vdash (x \ a \ b \ x) \quad \rightarrow \quad T$$

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

```
;      '(a b)      '( a b )  ->  T
```

```
(defun equip (x y)
  (cond
    ((and (subset-p x y) (subset-p y x)) t)
    (t nil))
)
```

```
(defun idemp (x y)
  (cond ((null x) t)
        ((equal (car x) (car y)) (idemp (cdr x) (cdr y)))
        (t nil))
)
```

```
; 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.
```

```
CL-USER > (implode '( a a b c c c d ))  
(A 2 B 1 C 3 D 1)
```

```
(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-----"""
"""-----"""

```

Define a procedure EXPLODE that realizes the inverse of the relation realized by IMplode. Assume that the input will always be a *list* where each symbol is immediately followed by a number that gives its *count* in the output.

```

CL-USER > (explode '(a 3 b 2 c 1 d 3))
(A A A B B C D D D)

(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-----"""
"""-----"""

```

Given a sequence of 0s and 1s, return the number of 0s that are preceded by a 0. Here is a sample interaction:

```

CL-USER > (zeros '(1 0 "0 0" 1 0))
2

(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 (length storage))
  )
)

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

```

```

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

```

Define a procedure REMAFTER that takes an element, a *list* and a pivot element and returns a *list* where all the occurrences of the element that are preceded by the pivot element are removed from the *list*.

```

'x 'p '(a p "x" b X c p "x" d a) -> '(a p _ b X c p _ d a)

(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) )
)

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

'x 'p '(a p "x" b X c p "x" d a) -> '(a p _ b X c p _ d a)

(defun remafter-2 (x lst pivot storage)
  (if (= (length lst) 1)
      (reverse (append lst storage))
      (let ((lst1 (list (car lst) (cadr lst)) )

```



```

        (lstd2 (list pivot x))
        (cr (car lst))
    )

    (cond ( (null lst) (reverse storage))
          ( (equal lst1 lst2) (remafter-2 x (append (list cr) (cddr lst)) pivot storage))
          ( t (remafter-2 x (cdr lst) pivot (cons cr storage) ) )
    )
)
)
)

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

```

The mean of  $n$  numbers is computed by dividing their *sum* by  $n$ . A running mean is a mean that gets updated as we encounter more numbers. Observe the following input-output sequences:

```

* (run-mean '(3 5 7 9))
  (3 4 5 6)

```

```

(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)
)

```

```

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

```

```

* (run-mean '(3 5 7 9))
  (3 4 5 6)

```

```

(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-----"""
"""-----"""

```

A chain in a sequence of numbers is such that each number in the chain is either equal to or greater than the one before it. For instance, 2 5 9 12 17 21 is a chain, but not 2 5 9 17 12 21, because the 17 12 sub-sequence breaks the chain. Define a recursive procedure that finds and returns the longest chain in a sequence of numbers. If there are more than one sequences with the highest length, return the one you encountered first. Here are some sample interactions:

```

* (longest-chain '(14 3 8 27 25 12 19 3 1))
  (3 8 27)

```

```

* (longest-chain '(14 3 8 27 25 12 19 34 42 1))
  (12 19 34 42)

```

```

* (longest-chain '(14 3 8 27 25 12 19 34 1))
  (3 8 27)

```

```

(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) )
)

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

* (longest-chain '(14 3 8 27 25 12 19 34 1) )
  (3 8 27)

I need a list like : ( (14) (3 8 27) (25) (12 19 34) (1) )

(defun func45 (lst temp storage)
  (cond ((null lst) storage)
        ((= (length lst) 1) (append storage (list lst)))
        (<= (car lst) (cadr lst)) (func45 (cdr lst) (cons (car lst) temp) storage))
        (t (func45 (cdr lst) nil (append storage (list (reverse (cons (car lst) temp)))))))
)

(defun find_lengths (lst storage)
  ; this will give (1 3 1 3 1)
  (cond ((null lst) (reverse storage))
        (t (find_lengths (cdr lst) (cons (length (car lst)) storage))))
)

'(14 3 8 27 25 12 19 34 1)

(defun func45-2 (lst counter)
  (let* ((lst1 (func45 lst nil nil)) ; ( (14) (3 8 27) (25) (12 19 34) (1) )
         (lst2 (find_lengths lst1 nil)) ; ( 1 3 1 3 1 )
         (max_ (apply #'max lst2)) ; 3
         )
    (if (= (nth counter lst2) max_)
        (nth counter lst1)
        (func45-2 lst (+ counter 1)))
    )
)

* (func45-2 '(14 3 8 27 25 12 19 34 1) 0)
  (3 8 27)

```

```

)
; (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

```

```

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

```

```

* (longest-chain '(14 3 8 27 25 12 19 34 1) )

```

```

I need a list like : ( (14) (3 8 27) (25) (12 19 34) (1) )

```

```

(defun func45 (lst temp storage)
  (cond ((null lst) storage)
        ((= (length lst) 1) (append storage (list lst)))
        (<= (car lst) (cadr lst)) (func45 (cdr lst) (cons (car lst) temp) storage))
        (t (func45 (cdr lst) nil (append storage (list (reverse (cons (car lst) temp)))))))
)

```

```

I need a list like : ( (14) (38) (25) (65) (1) )

```

```

(defun find_sums (lst)
  (if (null lst)
      0
      (+ (car lst) (find_sums (cdr lst))))
)

```

```

(defun find_biggest (lst)
  (if (null lst)
      0
      (max (find_sums (car lst)) (find_biggest (cdr lst))))
)

```

```

(defun main (lst)
  (find_biggest (func45 lst nil nil))
)

```

```

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

```

Define a procedure which takes a sequence of integers and returns the chain – not necessarily maximal – with the largest sum. If you detect maximal chains with equal sums, return the one you encountered first.

```

(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_)
        )
    )
  )

; (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-----"""
"""-----"""

```

See the PAIRLISTS in lecture notes. Define a procedure that “pairs” an arbitrary number of lists. Here is a sample interaction:

```

pairlist '( (a b) (=) (1 2) (+ -) (3 9) ) nil nil)

(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))

```

```

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

```

```

(defun pairlist2 (x)
  (append (list (mapcar #'car x)) (list (mapcar #'cadr x)) )
  )

```

```

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

```

Define a procedure SEARCH-POS that takes a list as search item, another list as a search list and returns the list of positions that the search item is found in the search list. Positioning starts with 0. A sample interaction:

```

* (search-pos '(a b) '(a b c d a b a b))

(6 4 0)

* (search-pos '(a a) '(a a a b a b))

(2 1 0)

```

```

(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-----""
""#-----""
```

Define a procedure LAST2 that takes a list and returns the last element of the list. Of course, don't use LAST. One way could be to keep a counter, so that you can compare this to the length of the list to recognize whether you are close enough to the end of the list.

```
(a b c d)

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

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

```
(defun last3 (lst)
  (car (reverse lst))
)
```

```
""#-----Third Way-----#""
```

```
(defun last4 (lst)
  (nth (- (length lst) 1) lst)
)
```

```
""#-----""
""-----Exercise 4.51-----""
""#-----""
```

Define an iterative procedure CHOP-LAST, which removes the final element of the given list – its like CDR from the back. You are NOT allowed to make (REVERSE (CDR (REVERSE LST))). Nothing to be done for an empty list, just return it as it is; but a single element list gets "nilled".

```
(a b c d) -> (a b c)

(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-----""
""#-----""
```

Define a procedure that checks whether a given list of symbols is a palindrome. Use CAR and your solution to Ex. 4.21.

```
(ey edip ada n ada pide ye)

(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)
```

```
* (palindrome '(a b b c b b a) )
```

```
(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)
  )
)
```

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

```
(defun palindrome2 (lst)
  (cond ((endp lst) t)
        ((equal (car lst) (car (reverse lst))) (palindrome2 (reverse (cdr (reverse (cdr lst))))))
        (t nil))
  )
)
```

""#----- Third Way -----#""

```
(t t t nil t)
```

```
(defun helper (lst)
  (if (null lst)
      t
      (and (car lst) (helper (cdr lst))))
  )
)
```

```
(defun palindrome3 (lst)
```

```
  (helper
    (mapcar #'(lambda (a b) (equal a b))
             lst (reverse lst)
    )
  )
)
```

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

Define your own version of NTH (don't use *NTHCDR*).

```
(a b X d e f) 2.th X

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

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

```
(a b X d e f) 2.th X
```

```
(defun n_th_2 (lst n)
  (let ((result nil)
        (counter 0))
    (dolist (i lst result)
      (if (= counter n)
          (and (setf result i) (setf counter (+ counter 1)))
          (setf counter (+ counter 1)))
      )
    )
  )
)
```

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

Define a procedure UNIQ that takes a list and removes all the repeated elements in the list "keeping only the first" occurrence. For instance:

```
* (uniq '(a b r a c a d a b r a))
```

```
(A B R C D)
```

```
(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) )
      )
  )
)
```

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

```
(defun unique_list (lst storage)
  (cond ((null lst) storage)
        ((member (car lst) (cdr lst)) (unique_list (cdr lst) storage))
        (t (unique_list (cdr lst) (cons (car lst) storage)))
  )
)
```

```
(defun main (lst)
```

```
  (unique_list (reverse lst) nil)
)
```

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

Solve Ex 4.54 by "keeping the last" occurrence rather than the first.

```
(defun unique_list_2 (lst storage)
  (cond ((null lst) (reverse storage))
        ((member (car lst) (cdr lst)) (unique_list_2 (cdr lst) storage))
        (t (unique_list_2 (cdr lst) (cons (car lst) storage))))
  )
)

-----
Exercise 4.56
-----
```

Define a procedure REMPLAST which removes the last occurrence of "an item" from a list. Do not use MEMBER or REVERSE.

```
'x' (a b c x x d x X c )
```

```
(defun count_them (x lst counter)
  (cond ((null lst) counter)
        ((equal (car lst) x) (count_them x (cdr lst) (+ counter 1)))
        (t (count_them x (cdr lst) counter)))
  )
)
```

```
'x' (a b c x x d x X c )
```

```
(defun remlast (x lst storage counter count_X)
  (cond ((null lst)
        (and (equal (car lst) x) (equal counter count_X))
        (equal (car lst) x)
        t)
        (t
         (remlast x (cdr lst) storage (+ counter 1) count_X)
         (remlast x (cdr lst) (append storage (list (car lst))) (+ counter 1) count_X)
         (remlast x (cdr lst) (append storage (list (car lst))) counter count_X)))
  )
)
```

```
(defun main (x lst)
  (let ((count_X (count_them x lst 0)))
    (remlast x lst nil 0 (- count_X 1))
  )
)
```

## Exercise 4.57

Define a procedure `FINDLAST` which returns the index of the last occurrence of an item in a list. Do not use `MEMBER` or `REVERSE`.

```
'x' (a b c x x d x X c )
```

```
(defun count_them (x lst counter)
  (cond ((null lst) counter)
        ((equal (car lst) x) (count_them x (cdr lst) (+ counter 1)))
        (t (count_them x (cdr lst) counter)))
  )
)
```

```
'x' (a b c x x d x X c )
```

```
(defun findlast (x lst counterforX count_X indx)
  (cond ((null lst) indx)
        ((and (equal (car lst) x) (equal counterforX count_X)) indx)
        ((equal (car lst) x) (findlast x (cdr lst) (+ counterforX 1) count_X (+ indx 1)))
        (t (findlast x (cdr lst) counterforX count_X (+ indx 1))))
  )
)
```

```
(defun main (x lst)

  (let ((count_X (count_them x lst 0)))

    (findlast x lst 0 (- count_X 1) 0)

  )

)
```

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

```
'x' (a b c x x d x X c )
```

```
(defun findlast2 (x lst)
  (let ((result 0))
    (dotimes (i (length lst) result)
      (if (equal (nth i lst) x)
          (setf result i)
          nil)
    )
  )
)
```

```

"""-----
""" Exercise 4.58 -----
"""-----

```

Define a procedure REMOVE<sub>X</sub> that takes an element and a list; and returns a list where all the occurrences of the element that are preceded by the symbol X are removed from the list.

```

'x '(a b c X v X c X d) -> '(a b c x _ x _ x _)

```

```

(defun remove (x lst storage)

  (cond ((null lst) (reverse storage))
        ((equal (car lst) x) (remove x (cddr lst) (cons (car lst) storage)))
        (t (remove x (cdr lst) (cons (car lst) storage) ))
  )
)

```

```

"""-----
""" Exercise 4.59 -----
"""-----

```

Define a function ROTATE-LEFT that takes a list and moves the first element to the end of the list. For instance, (ROTATE-LEFT '(1 2 3)) should give (2 3 1), (ROTATE-LEFT '(1 2)) should give (2 1), etc. Apart from DEFUN, you are allowed to use LET, LIST, APPEND, CAR, DOLIST, SETF and IF. No other function is available for use.

```

(1 2 3 4 5) -> (2 3 4 5 1)

```

```

(defun ROTATE-LEFT (lst)

  (append (cdr lst) (list (car lst))))
)

```

```

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

```

```

(1 2 3 4 5) -> new_set : (1 2 3 4 5 1) -> (2 3 4 5 1)

```

```

(defun ROTATE-LEFT2 (lst)

  (let ((one 0)
        (result nil)
        (new_set (append lst (list (car lst)))))

    (dolist (i lst result)

      (if (= one 1)
          (setf result (append result (list i)))
          (setf one 1))
      )
    )
  )
)

```

```

"""-----
""" Exercise 4.60 -----
"""-----

```

Substitute : a function with 3 arguments: old, new, and exp,

```

(subs 'x 'k '(x (x y) z)) -> (k (x y) z)

```

```

(defun subs (lst old new storage)

  (cond ((null lst) (reverse storage))
        ((equal (car lst) old) (subs (cdr lst) old new (cons new storage)))
        (t (subs (cdr lst) old new (cons (car lst) storage)))
  )
)

```

```

(defun subs2 (lst old new)

  (mapcar #'(lambda (x) (if (equal x old)
                             new
                             x))
    lst)
)

```

```

"""-----
""" Exercise 4.61 -----
"""-----

```

Define a procedure MATCHES that takes two lists, a pattern and a text, and returns the count of the occurrences of the pattern in the text. You need to be careful about overlapping matches. For instance, (A C A) has 3 occurrences in (A C A C A T G C A C A T G C). You are not allowed to use procedures like SUBSEQ to take portions of the text for comparison; your solution must go through the text element by element.

```

(defun matches_ (text lst result)

  (if (equal (length lst) result)

      (setf result (cons (nth i text) result)))
  )
)

```



```
(reverse lst))
```

```
1  
0
```

```
(defun matches (text lst)
```

```
(if (null text)  
    0  
    (+ (matches_ text lst nil) (matches (cdr text) lst)))  
)
```

```
""-----""  
""-----Exercise 4.62-----""  
""-----""
```

Define a procedure SHUFFLE that takes a list and returns a random permutation of the list. A random permutation of a list is one of all the possible orderings of the elements of the list. You can follow any strategy you like – recursive or iterative. You might find two built-ins especially useful: RANDOM takes an integer and gives a random number from 0 to one less than the given integer; NTH takes an integer and a list, returning the element at the position of the given integer – remember that positions are counted starting from 0.

```
'(a b c d e)      ->      '(a c e d b)  
  
(0 1 2 3 4 5)     ->      (0 2 5 4 1 3)      (random length:6)      :      0-5  
  
I need a func like:      (0 1 2 "3" 4 5)      ->      (0 1 2 _ 4 5)
```

```
(defun shuffle_ (lst x storage)  
  (cond ((null lst) storage)  
        ((equal x (car lst)) (shuffle_ (cdr lst) x storage))  
        (t (shuffle_ (cdr lst) x (cons (car lst) storage))))  
)
```

```
(defun shuffle (lst storage)  
  (let* ((rnd (random (length lst)))  
         (x (nth rnd lst))  
         (remain (shuffle_ lst x nil)))  
    (cond ((null lst) storage)  
          ((= (length lst) 1) (cons x storage))  
          (t (shuffle remain (cons x storage))))  
  )  
)
```

```
""-----""  
""-----Exercise 4.63-----""  
""-----""
```

Modify SUBSTITUTE to D-SUBS (for “deep *substitute*”), so that it does the replacement for all occurrences of old, no matter how deeply embedded.

Substitute : a function with 3 arguments: old, new, and exp,

```
(subs 'x 'k '(x (x y) z)) -> (K (K y) z)
```

```
(defun subs3 (lst old new)
```

```
(mapcar #'(lambda (x) (if (listp x)  
                          (subs3 x old new)  
                          (if (equal x old)  
                              new  
                              x)  
                          ))  
  lst)  
)
```

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

```
(subs 'x 'k '(x (x y) z)) -> (K (K y) z)
```

```
(defun subs4 (lst old new storage)
```

```
(cond ((null lst) (reverse storage))  
      ((listp (car lst)) (subs4 (cdr lst) old new (append (list (subs4 (car lst) old new nil)) storage)))  
      ((equal (car lst) old) (subs4 (cdr lst) old new (cons new storage)))  
      (t (subs4 (cdr lst) old new (cons (car lst) storage))))  
)
```

```
(K (K Y) Z)
```

```
""-----""  
""-----Exercise 4.64-----""  
""-----""
```

Define a recursive procedure that counts the non-nil atoms in a list. For instance, an input like ((a b) c) should return 3, (a ((b (c) d))) should return 4, and so on. Remember that the built-in ATOM returns NIL for all lists except NIL; NULL returns T only for NIL; ENDP is like NULL, except that it gives an error if its input happens to be something other than a list. Your function should use a counter/accumulator – it will be a two argument function.

(a ((b (c) d))) should return 4

```
(defun counter (lst)
```

```
  (if (null lst)
      0
      (+ 1 (counter (cdr lst))))
)
```

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

```
(defun counter2 (lst counter)
```

```
  (if (null lst)
      counter
      (counter2 (cdr lst) (+ counter 1)))
)
```

""-----""  
 ""----- Exercise 4.65 -----""  
 ""-----""

Define a procedure BRING-TO-FRONT (or BFT for short), that takes an item and a list and returns a version where all the occurrences of the item in the given list are brought to the front of the list.

For instance, (bring-to-front 'a '(a b r a c a d a b r a)) would return

```
(A A A A B R C D B R);
```

and (bring-to-front 'b '(a b r a c a d a b r a)) would return

```
(B B A R A C A D A R A).
```

You are NOT allowed to count the occurrences of the item in the given list or use REMOVE.

```
(defun bft (x lst temp storage)
```

```
  (cond ((null lst) (append temp (reverse storage)))
        ((equal (car lst) x) (bft x (cdr lst) (cons x temp) storage))
        (t (bft x (cdr lst) temp (cons (car lst) storage) )))
)
```

Force it ! You are going to change the world !

This is your fate !

Remember your f\*cking past man ! Remember your past! This is determinism.

You have to change the world.

Nothing can stop you. Nothing...

""-----""  
 ""----- Exercise 4.66 -----""  
 ""-----""

Define a procedure that groups the elements in a list putting consecutive occurrences of items in lists. For instance,

(group '(a a b c c c d d e)) should give

```
((A A) (B) (C C C) (D D) (E)).
```

Note that you should NOT bring together non-consecutive repetitions; a call like

(group '(a b b c b b c)) should return

```
((A) (B B) (C) (B B) (C)).
```

```
(defun group (lst reader temp storage)
```

```
  (cond ((null lst) (append storage (list temp)))
        ((equal (car lst) reader) (group (cdr lst) reader (cons reader temp) storage))
        (t (group lst (car lst) nil (append storage (list temp)))))
)
```

""-----""  
 ""----- Exercise 4.67 -----""  
 ""-----""

Define a recursive procedure SUMMARIZE, that takes a list and returns a list of pairs whose car is an element in the list and cadr is the number of times the element occurs in the list;

(summarize '(a b r a c a d a b r a)) should give

```
((a 5) (b 2) (r 2) (c 1) (d 1)).
```