[COGS 502 Section 1] Symbols and Programming

All the questions are from the course repository of Umut Özge (Informatics Institute, METU).

The link is https://github.com/umutozge/symbols-and-programming

All the answers are belong to me (Turgay Yıldız).

Note: Neither Chat-GPT nor any other AI (or Other Language Models) is used when solving answers.

All answers are completely human-made.

Best Regards.

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



"""-------------------------------------------------------------------------------------"""
"""------------------                  Exercise 1.1                  ------------------"""
"""-------------------------------------------------------------------------------------"""

; Define a procedure that takes two numbers and returns their average.

 (defun avg (x y) (/ (+ x y) 2))

"""-------------------------------------------------------------------------------------"""
"""------------------                  Exercise 1.2                  ------------------"""
"""-------------------------------------------------------------------------------------"""

; Define a procedure that takes two numbers and returns the number obtained by dividing their
; product by their average. For the inputs 3 and 4, your program must return 12/3.5. In your
; solution, use the procedure you defined for Exercise 1.1.

(defun func1 (x y) (/ (* x y) (avg x y) ) )

"""-------------------------------------------------------------------------------------"""
"""------------------                  Exercise 1.3                  ------------------"""
"""-------------------------------------------------------------------------------------"""

; Define a function that takes three arguments x, y and n, and returns the result of the following function:

; A x B  = (A1 / A2)  X  (B1  /  B2)   =  (A1  /  (a2 / a3))   X  ( (b1 + b2)  /  (B2))


(defun func2 (x y n)  (* (/ (expt x n) (- 7 (/ y 2)) )     (/ (+ (expt y (/ 2 3)) 17) 4)))

"""-------------------------------------------------------------------------------------"""
"""------------------                  Exercise 1.4                  ------------------"""
"""-------------------------------------------------------------------------------------"""


; In order to convert a temperature in Fahrenheit into Celsius, you need to subtract
; 32 from it and multiply the result by 5/9. Define a procedure that converts from
; degrees Fahrenheit to degrees Celsius.

(defun convertFtoC (f)  (* (/ 5 9) (- f 32)))
```

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



"""-----------------------------------------------------------------------------"""
"""------------------                  Exercise 2.1                 -------------------"""
"""-----------------------------------------------------------------------------"""

; Define a procedure named ASCENDINGP that takes three numbers as input and re-
; turns T if the numbers are in ascending order, and NIL otherwise. Equality means
; ascension, therefore (ASCENDINGP 3 4 4) must return T.

(defun ascendingp (x y z)

  (and (<= x y) (<= y z))
)

"""-----------------------------------------------------------------------------"""
"""------------------                  Exercise 2.2                 -------------------"""
"""-----------------------------------------------------------------------------"""

; Define a procedure that takes two numbers and returns -1 if their difference is
; negative, 0 if they are equal, and 1 if their difference is positive. You do not need
; to check for numberhood, assume that the user will always give numbers as input.
; You are allowed to compute the difference of the input numbers only once; and
; SETF and DEFVAR are forbidden.

(defun neg_eq_pos (x y)

  (if   (< x y)
        (- 0 1)
        (if (= x y)
            0
            1
         )))

"""-----------------------------------------------------------------------------"""
"""------------------                  Exercise 2.3                 -------------------"""
"""-----------------------------------------------------------------------------"""
; Solve Ex. 2.2, this time by checking for numberhood as well. Your program should
; return NIL if any (or both) of the numbers is not a number. Do NOT use AND.

(defun func1 (x y)

  (if (or (not (numberp x)) (not (numberp y))  )
      nil
      (neg_eq_pos x y)
      )
  )

; (func1 2 'a)    ->    nil
; (func1 2  a)    ->    error

"""-----------------------------------------------------------------------------"""
"""------------------                  Exercise 2.4                 -------------------"""
"""-----------------------------------------------------------------------------"""

; Define a procedure that takes three numbers and returns T if all the three are integers
; and returns NIL otherwise. Do NOT use AND.

(defun func2 (x y z)

  (if   (or  (not (integerp x))  (not (integerp y))   (not (integerp z)) )
        nil
        t
  ))


"""-----------------------------------------------------------------------------"""
"""------------------                  Exercise 2.5                 -------------------"""
"""-----------------------------------------------------------------------------"""
```

```
; Write a function HOWCOMPUTE taking 3 numbers, telling the basic arithmetic oper-
; ation that is used to compute the third number from the first two – it should say
; so if it cannot find it. Your response can be one of ADDED, MULTIPLIED, DIVIDED,
; SUBTRACTED, DONT-KNOW.9 . Use COND in your answer.

(defun howcompute (x y z)

  (cond

   ((=  (- x y) z) (print "subtracted"))
   ((=  (- y z) z) (print "subtracted"))
   ((=  (+ x y) z) (print "add"       ))
   ((=  (* x y) z) (print "multiplied"))
    (t             (print "dont know" ))
    )
  )


"""----------------------------------------------------------------------------------------"""
"""-------------------                  Exercise 2.6                  -------------------"""
"""----------------------------------------------------------------------------------------"""

; Define a function that takes two arguments and returns the greater of the two.

(defun func3 (x y)

  (if (and (numberp x)  (numberp y) )
      (if   (>= x y)
            x
            y
       )
      nil ))

"""----------------------------------------------------------------------------------------"""
"""-------------------                  Exercise 2.7                  -------------------"""
"""----------------------------------------------------------------------------------------"""

; Define a procedure that takes three arguments and returns the greatest of the three.

(defun func4 (x y z)

    (func3 (func3 x y) z)
)


"""----------------------------------------------------------------------------------------"""
"""-------------------                  Exercise 2.8                  -------------------"""
"""----------------------------------------------------------------------------------------"""

; Define a procedure that takes three numbers and gives back the second largest of
; them. Use only IF and comparison predicates like <, <=, etc.

(defun func5  (x y z)

  (if (and (<= x y) (<= x z) )

      (if (<= y z)
          y
          z
       )

      (if   (and (>= x y) (>= x z) )
            (if  (>= y z)
                 y
                 z
             )
            x)))

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

(defun func5_2 (x y z)

  (cond   ( (and (<= x y) (<= x z))               (if (<= y z)  y   z))
          ( (and (>= x y) (>= x z))               (if (>= y z)  y   z))
          ( t                                      x)
  ))
```

```
"""----------------------------------------------------------------------------------------"""
"""------------------                    Exercise 2.9                    ------------------"""
"""----------------------------------------------------------------------------------------"""

; Define a procedure that takes three numbers and gives back the sum of the squares
; of the larger two.

(defun sos (x y)  (+ (expt x 2) (expt y 2)))

(defun func9 (x y z)

  (cond ((and (<= x y) (<= x z))               (sos y z))
        ((and (<= y x) (<= y z))               (sos x z))
        (t                                     (sos x y))
  ))
"""----------------------------------------------------------------------------------------"""
"""------------------                    Exercise 2.10                   ------------------"""
"""----------------------------------------------------------------------------------------"""

; Solve Ex. 2.8 using COND.

(defun func5_2 (x y z)

  (cond   ( (and (<= x y) (<= x z))               (if (<= y z)  y   z))
          ( (and (>= x y) (>= x z))               (if (>= y z)  y   z))
          ( t                                      x)
  ))

"""----------------------------------------------------------------------------------------"""
"""------------------                    Exercise 2.11                   ------------------"""
"""----------------------------------------------------------------------------------------"""

; Define a procedure named halver that halves a given number until the result be-
; comes less than 1 and returns that result – solve the problem by making your pro-
; cedure call itself. Here is an example interaction with such a procedure:

(defun halver (x)

  (if  (<  x  1)
       x
      (halver (/ x 2))
  ))


"""----------------------------------------------------------------------------------------"""
"""------------------                    Exercise 2.12                   ------------------"""
"""----------------------------------------------------------------------------------------"""

; Rewrite (AND X Y Z W) by using cond COND.

(defun func12 (x y z w)

  (cond
  (X
   (cond
     (Y
      (cond
        (Z
         (cond
           (W t)
           (t nil)))
       (t nil)))
    (t nil)))
  (t nil))
  )


"""----------------------------------------------------------------------------------------"""
"""------------------                    Exercise 2.13                   ------------------"""
"""----------------------------------------------------------------------------------------"""

; Write COND statements equivalent to: (NOT U) and (OR X Y Z)


; (NOT U)
```

```
(defun myfunc13 (x)

  (cond (x   (not x) )  ; when x == True, if x == NIL , then this does not work
        (t      t    )  ; otherwise, if you remove this line and if x is NIL, output will be NIL
  ))

; (OR X Y Z)

(defun myfunc13-2 (x y z)

  (cond
    (X t)
    (Y t)
    (Z t))
  )         ; otherwise NIL, you dont need to add the line :   (t            nil)


(defun myfunc13-3 (x y z)

  (cond
    (x  x)
    (y  y)
    (z  z)
  )
)

"""-------------------------------------------------------------------------------------------"""
"""------------------                   Exercise 2.14                   ------------------"""
"""-------------------------------------------------------------------------------------------"""

; Write the final version of the CHANGE-COND program using only AND and OR, no IF, no COND.

( defun changer-cond ( n )

    ( cond  (( not ( numberp n ))               nil )
            (( not ( integerp n ))              ( changer-cond ( round n )))
            (( zerop ( rem n 3))                (+ (* 3 n ) 1))
            ( t                                 n )))



(defun func14 (n)

  (and (numberp n)

  (or
    (and (not ( integerp n ))               (changer-cond ( round n )))
    (and (zerop ( rem n 3))                 (+ (* 3 n ) 1))
    n)))


"""-------------------------------------------------------------------------------------------"""
"""------------------                   Exercise 2.15                   ------------------"""
"""-------------------------------------------------------------------------------------------"""

; The following definition is meant to mimic the behavior of IF using AND and OR.

( defun custom-if ( test succ fail )

        ( or ( and test succ ) fail ))


; But it is unsatisfactory in one case, what is it? Define a better procedure which
; avoids this failure.

; problem is * (custom-if T nil T)     returns   T.

( defun func15 ( test succ fail )

    (or (and test succ)  (and (not test) fail) ))
```

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


"""-----------------------------------------------------------------------------"""
"""------------------                   Exercise 3.1                  -------------------"""
"""-----------------------------------------------------------------------------"""

; Define a procedure GUESS. It will have one parameter, an integer including and
; between 0 and 99. You will make the computer make successive guesses to find
; this number, where each guess will appear on the screen – use PRINT for this. You
; will need the LISP expression (random 100) to make a guess. Needless to say,
; the only acceptable way to go on making guesses as long as needed is keep calling
; yourself.


(defun guess (x)

  (let ((y                  (random 100)))

    (cond ((= x y)                  x)
          ((print y)                (guess x))
           )))


""" random 100 will change if you call it again even inside let """



;   (let  (( y                      (random 100)  )   )
;
;           (and    (print  y)    (print(random 100))   )
;   )


""" returned :    39   70    70   """


"""-----------------------------------------------------------------------------"""
"""------------------                   Exercise 3.2                  -------------------"""
"""-----------------------------------------------------------------------------"""

; Define a procedure that multiplies two integers using only addition as a primitive
; arithmetic operation.


(defun mltp (x y)

  (if  (= x 0)
       0
       (+ (mltp (- x 1)   y)   y)
       )
  )

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

(defun mltp2 (x y z)

    (if  (= x 0)
         z
        (mltp2  (- x 1)  y  (+ y z) )
    )
)

(defun mltp3 (x y)

    (mltp2 x y 0)
)

"""-----------------------------------------------------------------------------"""
"""------------------                   Exercise 3.3                  -------------------"""
"""-----------------------------------------------------------------------------"""

; The factorial of a non-negative integer is defined as follows: ...
; Define a procedure that computes the factorial of a given integer.


(defun fact (x)
```

```
  (if  (= x 0)
       1
       (* (fact (- x 1))    x)
       )
  )

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


(defun fact2 (x acc)           ; tail-call optimization

  (if  (= x 0)
       acc
       (fact2 (- x 1)   (* acc x))
       )
  )


"""-------------------------------------------------------------------------------------------"""
"""------------------                  Exercise 3.4                ------------------"""
"""-------------------------------------------------------------------------------------------"""

; Define a recursive procedure that computes the sum of the squares of the first n
; non-negative integers.

; 1^2 + 2^2 + 3^2 + ... + 10^2 = 385

(defun sos (x)

  (if (= x 1)
      1
      (+  (sos (- x 1))  (* x x) )   ; lastly adds "1"
      )
  )

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

(defun sos_2 (x acc)      ;

  (if (= x 0)
      acc      ; when x = 0
      (sos_2 (- x 1)  (+ acc (* x x) ))  ; lastly    acc + 1^2
      )
  )



"""-------------------------------------------------------------------------------------------"""
"""------------------                  Exercise 3.5                ------------------"""
"""-------------------------------------------------------------------------------------------"""

; The way to toss a fair coin in LISP is to do (random 2), which would evaluate to
; 0 or 1 with a fifty-fifty chance.

; PRINT is a special form. It evaluates its first argument, returns the value com-
; puted, and, as a side-effect, prints the computed value on the screen.

; Define a recursive procedure TOSS that takes a non-negative integer n, tosses a
; coin n number of times, printing the result (0 or 1) on the screen in each toss.

; (Hint: AND is useful when you want to evaluate two forms in succession.)


(defun toss (n)

  (if   (and (> n 0) (print (random 2)) )
        (toss (- n 1))))


"""-------------------------------------------------------------------------------------------"""
"""------------------                  Exercise 3.6                ------------------"""
"""-------------------------------------------------------------------------------------------"""

; Collatz' Conjecture says that starting with any positive integer, by running the func-
; tion C defined below, you will eventually reach number 1.

(defun coll (n)

  (cond     (  (= n 1)          1)
            (  (evenp n)        (coll (/ n 2))   )
            (  (oddp  n)        (coll (+  (* 3 n) 1))  )
    ))
```

```
"""----------------------------------------------------------------------------------------"""
"""------------------                  Exercise 3.7                  --------------------"""
"""----------------------------------------------------------------------------------------"""

; Define a recursive procedure that takes two integers, say x and y, and returns the
; sum of all the integers in the range including and between x and y. Do not use a
; formula that directly computes the result.



(defun sumRange (x y)

  (if  (= x y)
       x
       (+  (sumRange   x   (- y 1) )  y)
       )
  )
"""----------------------------------------------------------------------------------------"""
"""------------------                  Exercise 3.8                  --------------------"""
"""----------------------------------------------------------------------------------------"""

; Define a tail-recursive factorial procedure.

(defun fact (x)

  (if  (= x 0)
       1
       (* (fact (- x 1))    x)   ; nesting , waiting functions, not efficient in terms of memory
       )
  )

"""----------------------------------------------------------------------------------------"""
"""------------------                  Exercise 3.9                  --------------------"""
"""----------------------------------------------------------------------------------------"""


; Define a two operand procedure that raises its first operand to the power of the
; second. You are allowed to use multiplication and subtraction. Define two versions,
; with and without an accumulator. You can check the behavior of your procedure
; by comparing it with LISP's EXPT, which does the same thing.

(defun expt2 (x y)   ; (2 3) = 2 x 2 x 2

  (if  (=  y  1)
       x
       (*  (expt2 x (- y 1))  x)
       )
  )

; "expt" means "exponential"               and         "exp"  means "euler number"
; (expt 2 3) = 2^3 = 2 x 2 x 2 = 8         and         (exp 1)  =  e ^ 1 = 2.718


"""----------------------------------------------------------------------------------------"""
"""------------------                  Exercise 3.10                 --------------------"""
"""----------------------------------------------------------------------------------------"""

; The Fibonacci numbers


(defun fib (n)

  (if  (or (= n 1) (= n 0))
       1
       (+  (fib (- n 1))  (fib (- n 2))  )
       )
  )


(defun fib2 (n acc newacc)

  (if  (= n 1)
       newacc
       (fib2 (- n 1)  newacc (+ acc newacc))
   ))



  """----------------------------------------------------------------------------------------"""
  """------------------                  Exercise 3.11                 --------------------"""
  """----------------------------------------------------------------------------------------"""

; Newton's Method
```

```
(defun getnewY (x y)

  (let  (   (newY       (/    (+ (/  x  y)  y)     2)  )   )
            newY
    )
  )


(defun newton (x newY)

  (print "initial guess : " )

  (print newY)

  (if (<=   (abs (-   x    (* newY  newY) ))     0.00001  )
      newY

      (newton x  (getnewY x  newY))


      )
  )

; (float (newton 81 1))   will return  9.0   like (sqrt 81)  =  9.0

; NOTE : Original question's algorithm is WRONG ! Page : 13/42

; Because you have to use "absolute value" to prevent "difference" from being negative !
; Machine can not know which value is bigger without comparing them. And comparing is the longest way.

"""----------------------------------------------------------------------------------------"""
"""------------------                  Exercise 3.12                  ------------------"""
"""----------------------------------------------------------------------------------------"""

; Sum of a geometric progression.  a.r^0 + a.r^1 + a.r^2 ... + a.r^n


(defun geo (a r n)

  (if (= n 0)
       a
      (+    (* a    (expt r n)    )    (geo a r (- n 1))   )

      )
  )

; NOTE : Original question is WRONG (missing values) ! Page : 13/42

; Because the right sequence should be : 7, 14, 28, 56, ...

"""----------------------------------------------------------------------------------------"""
"""------------------                  Exercise 3.13                  ------------------"""
"""----------------------------------------------------------------------------------------"""

;(RANDOM N) returns a random number between and including 0 and n − 1. Define a
;procedure that takes two arguments n and r, and prints r random numbers between
;and including 0 and n. You will need to use PRINT; you can discover how it works
;by trying it at REPL.


(defun rd (n r)

    (if  (= r 1)
        (random n)
        (and   (print (random n))    (rd  n  (- r 1)  )  )
    )
)



"""----------------------------------------------------------------------------------------"""
"""------------------                  Exercise 3.14                  ------------------"""
"""----------------------------------------------------------------------------------------"""


; You can find at code/var/primep.lisp on our Github site a program that checks
; wether a given integer is prime or not. Define a procedure that takes an integer,
; changes it by Collatz' function until it reaches a prime number. Return the prime
; number that is reached.

; You can use the PRIMEP predicate in your program by loading the program it is
; defined in (the primep.lisp must be in the same folder as your own program):
```

```lisp
; remember collatz function


;(defun coll (n)
;
; (cond   (  (= n 1)       1)
;     (  (evenp n)    (coll (/ n 2))   )
;     (  (oddp  n)    (coll (+  (* 3 n) 1))  )
;     )
; )
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;; from SICP (here in clojure)
;; http://www.sicpdistilled.com/section/1.2.6/

(defun square (n)
  (* n n))


(defun dividesp (a b)
  (zerop (mod b a)))

(defun find-divisor (n test-divisor)
  (cond ((> (square test-divisor) n) n)
        ((dividesp test-divisor n) test-divisor)
        (t (find-divisor n (1+ test-divisor)))))


(defun smallest-divisor (n)
  (find-divisor n 2))


(defun primep (n)
  (= n (smallest-divisor n)))


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;



(defun if-prime (x)

    (if (primep x)

        (and (print "Prime found : "  )    (print x)  t)

        (cond   (   (= x 1)                  1)
                (   (evenp x)                (and (print x)  (if-prime (/ x 2))   ))
                (   (oddp  x)                (and (print x)  (if-prime (+  (* 3 x)  1) )  ) )
        )
    )
)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;



"""----------------------------------------------------------------------------------------"""
"""-------------------                   Exercise 3.15                 -------------------"""
"""----------------------------------------------------------------------------------------"""

;Your task is to write a program that takes a positive (n > 0) integer as an input and
;reduce it to 1 by using the Collatz’ function. While doing this, you are required
;to report any prime number you encounter along the way. Besides reporting the
;primes, your program should also report and return the sum of these primes.
;You need to write two versions: one, call it KOO, where you accumulate the sum
;as you go along and return it when you reach 1; the other, call it FOO, where you do
;not accumulate the answer as you go along.


(defun collatz (x)

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


(defun foo (x)

    (if (= x 1)
        (and (print "END" ) t)
```

```
        (if (primep x)
            (and  (print "Prime found : ")  (print x) (foo (collatz x) )   )
            (foo (collatz x) )
        )
    )
)


(defun koo_ (x sum_)

    (if (= x 1)

        (and  (print "Total sum of the primes : ") (print sum_) t)

        (if (primep x)
            (and  (print "Prime found : ")  (print x)   (koo_ (collatz x)  (+ x sum_ )   )  )
            (koo_ (collatz x) sum_ )
        )
    )
)


(defun koo (x)

    (koo_ x 0)
)



"""-------------------------------------------------------------------------------------------"""
"""------------------                   Exercise 3.16                   --------------------"""
"""-------------------------------------------------------------------------------------------"""

; Define a procedure that takes a positive integer (n > 0), reduces it to 1 by Collatz'
; algorithm, printing in each step, the difference between the current number and the
; one computed before it.

; (collatz x)   -   x   = output  -  input


(defun collatz (x)

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


(defun collatz-diff (x)

    (if (= x 1)
        (and (print "END")  t)

        (and    (print  (- (collatz x)  x)  )   (collatz-diff (collatz x))  )
    )
)



"""------------------                   END                   --------------------"""
```

```
"""#-----------------------------------------------------------------------------#"""
"""#---------------------         METU Cognitive Sciences         ---------------------#"""
"""#---------------------          Symbols & Programming          ---------------------#"""
"""#---------------------              Turgay Yildiz              ---------------------#"""
"""#---------------------        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 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           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 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               ---------------------#"""
"""#-----------------------------------------------------------------------------#"""

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)    ( cadddr 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)                ; 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)


"""-----------------          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)       ; ( (A 100) (B 150) (C 200))

    (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)       ; ( (A 100) (B 150) (C 200))

    (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)              ; (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 )  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) )
    )
)

""" -----------------       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
                )
            )
    )
)

(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))

(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 x 1   +   2^1  x 0  +   2^2  x  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 re-
turns 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 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))
        )
    )
)


"""-----------------------------------------------------------------------------------------"""
"""------------------                  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)                                            ; ****************************************
                                                            ; If you can visualize this path, you are on the right path !
    (cond   ( (null x)                          nil)        ; Note: append does not care about NIL, but values must be list
            ( (not (listp x))                   (list x))   ; Now, these Xs are both car and cdr from below

            ( (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)


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

(defun rev4  (x storage)                                        ; ****************************************

    (cond   ( (null x)                          storage)        ; If you forget the NIL below, it will print double values
            ( (not (listp x))                   x)              ; Because this is a new function. Good job !

            ( (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)
              (nt                     (- nt 1))
             )

        (dotimes    (i (length lst) (reverse storage))

            (cond   ( (and (equal (nth i lst)  x) (= counter nt))               (setf counter 0))

                    ( (equal (nth i lst)  x)                                    (and
                                                                                (setf counter   (+ counter 1))
                                                                                (setf storage   (cons (nth i lst) storage))))

                    ( t                                                         (setf storage   (cons (nth i lst) storage)))
            )
        )
    )
)
```

Everything is about "ALL" probabilities ! Consider ALL.

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

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.

(a) Define a procedure *SUBSETP* that takes two *list* arguments and decides whether the *first* is a subset of the *second*.

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


(defun subset_  ( car_x    y)                   ; this car will come from below

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


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


(defun subset-p (x y)

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

(b) Define a procedure EQUIP that takes two *list* arguments and decides whether the two are equivalent.

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

(defun equip  (x y)

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

(c) Define a procedure IDENP that takes two *list* arguments and decides whether the two have the same elements in the same order — do not directly compare the lists with EQUALP, you are required to do a element by element comparison.

```
(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.

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 imme-
diately 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)) )
                     (lst2                          (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)


"""-------------------------------------------------------------------------------------------"""
"""------------------                  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*                 -------------------"""
"""-------------------------------------------------------------------------------------------"""

A maximal chain m in a sequence of integers I is a chain defined in the sense of
Exercise 4.45, such that there is no chain k in I such that m is a subsequence of k.
Define a procedure which takes a sequence of integers and returns the maximal
chain with the largest sum. If you detect maximal chains with equal sums, return
the one you encountered first.


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



    (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



"""#----------------------      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_
                                                            )
                    )
            )

    )
)


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

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 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 REMLAST 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)                                                        storage)
            ( (and (equal (car lst) x) (equal counter count_X))                 (remlast x (cdr lst) storage (+ counter 1) count_X))
            ( (equal (car lst) x)                                               (remlast x (cdr lst) (append storage (list (car lst))) (+ counter 1) count_X))
            ( t                                                                 (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 REMOVEX 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  removex (x lst storage)


    (cond   ( (null lst)                        (reverse storage))
            ( (equal (car lst) x)               (removex x (cddr lst) (cons (car lst) storage)))
            ( t                                 (removex 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 al-
lowed 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 re-
turns 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      (dotimes (i (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 replace-
ment 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 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) ))

    )
)

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


Define a procedure that groups the elements in a list putting consecutive occur-
rences 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 el-
ement 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)).


I need   (a 5)

(defun summarize_ (lst cr counter)


    (cond   ( (null lst)                        (cons cr (cons counter nil)))
            ( (equal cr (car lst))              (summarize_ (cdr lst) cr (+ counter 1)))
            ( t                                 (summarize_ (cdr lst) cr counter))
    )
)


Now I need '(_ b r _ c _ d _ b r _)


(defun remain (lst x storage)

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


(defun summarize (lst storage)

    (let    ( (remain                       (remain lst (car lst) nil))
              (first_                       (summarize_ lst (car lst) 0))
            )


    (cond   ( (null lst)                    storage)
            ( t                             (summarize remain (append storage (list first_)) ))
    )
    )
)



"""--------------------------------------------------------------------------------------"""
"""------------------                  Exercise 4.68                  ------------------"""
"""--------------------------------------------------------------------------------------"""


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).   (even, odd)


(defun collatz (x)

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


3   ->      3 10 5 16 8 4 2 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)  )
        )
    )
)


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

3   ->      3 10 5 16 8 4 2 1

(defun countE (x)

    (if     (= x 1)
            0
            (or
                (and (evenp x) (+ 1 (countE (collatz x))))
                (countE (collatz x))
            )
    )
)


(defun countO (x)

    (if     (= x 1)
            1
            (or
                (and (oddp x) (+ 1 (countO (collatz x))))
                (countO (collatz x))
            )
```

```
    )
)


(defun countEO2   (x)


    (append (list (counte x)) (list (counto x)))

    )



"""------------------------------------------------------------------------------------------"""
"""------------------                  Exercise 4.69                   ------------------"""
"""------------------------------------------------------------------------------------------"""

A growing difference sequence is a recursive sequence where each non-initial term
in the sequence is greater than the one before it by a difference that steadily grows
with the terms. For instance 1, 4, 8, 13, 19, 26,. . . is such a sequence where the second
term is obtained by adding 3 to the first, third term is obtained by adding 4 to the
second, fourth term is obtained by adding 5 to the third, and so on. In tabular form:

Our sequences will always start with 1. How the difference starts and grows
may change from sequence to sequence. For instance the difference in the follow-
ing sequence starts with 2 and grows as the square of the previous difference.

Define a procedure GDS that generates a growing difference sequence where the
length of the sequence, the initial value of the difference and how difference grows
will be given as parameters. An example output for the first 7 terms in the first ex-
ample above would be

((1 1) (2 4) (3 8) (4 13) (5 19) (6 26) (7 34)).


GDS - >  length = 7 ,  initial value of difference = + 3,  how grows = algorithm  here +1 ?


Our sequences will always start with 1


(defun gds (len  diff temp counter  storage)            ; len    3    1   1   nil

    (cond   ( (= counter len)                       (append '((1 1)) storage))
            ( t                                     (gds len (+ diff 1) (+ temp diff) (+ counter 1)

                                                    (append storage (list (list (+ counter 1) (+ temp diff)))) ))
    )
)

(defun main (len)

    (gds len 3 1 1  nil)
)



"""------------------------------------------------------------------------------------------"""
"""------------------                  Exercise 4.70                   ------------------"""
"""------------------------------------------------------------------------------------------"""

working of hash table :

* (defparameter *my-hash* (make-hash-table))

*MY-HASH*

* (setf (gethash 'one-entry *my-hash*) "one")

* (setf (gethash 'another-entry *my-hash*) 2/4)

* (gethash 'one-entry *my-hash*)
"one"
T
* (gethash 'another-entry *my-hash*)
1/2
T

(defun fib (n *my-hash*)


    (or     (gethash n *my-hash*)

            (setf (gethash n *my-hash*) (+ (fib (- n 1) *my-hash*)  (fib (- n 2) *my-hash*)) ) )
)



(defun main  (n)

    (defparameter *my-hash* (make-hash-table))

    (setf (gethash 0 *my-hash*) 1)
    (setf (gethash 1 *my-hash*) 1)

    (fib  n  *my-hash*)
)

"""------------------------------------------------------------------------------------------"""
"""------------------                  Exercise 4.71                   ------------------"""
"""------------------------------------------------------------------------------------------"""

Define a procedure PERMUTE that gives the permutation of a sequence — all the
sequences with the same elements in different orders. Assume all the elements in
the sequence will be distinct.

(A B C)   A , (B C)   ->  A  BC ,  A  CB
(B A C)   B,  (A C)   ->  B  AC ,  B  CA
(C A B)   C,  (A B)   ->  C  AB ,  C  BA

(B C)   B , (C)     ->

(defun permute (lst)

    (let    ((result                    nil)
            )

        (Cond   ( (not (listp lst))                 lst)
                ( (= (length lst) 2)                (list lst (reverse lst)))
                ( t

                (dolist  (i lst result)
```

```
                    (setf result (append

                    (mapcar #' (lambda (x)

                    (cons i x))
                    (permute (remove i lst))) result))
                ))
        )
    )
)


"""-------------------------------------------------------------------------------------------"""
"""------------------                  Exercise 4.72                  ------------------"""
"""-------------------------------------------------------------------------------------------"""

Define a procedure that takes a list and turns it into a binary search tree. A binary
search tree is a binary tree such that for each node in the tree, all the items in its
left sub-tree is less than or equal to the item on that node, and all the items on its
right sub-tree are greater than the item on that node. To makes things a little easier
you may assume that every node has exactly two sub-trees, which are possibly NIL.
You can represent trees in LISP as lists of three elements; where the first element
is the "parent" node, the second element is the "left child" and the third element is the
right child.


(defun func_72 (lst)


    (cond   ( (null lst)                        nil)
            ( (null (cdr lst))                  lst)
            ( t


        (let*   ( (parent               (nth (random (length lst)) lst))
                  (result_l             nil)
                  (result_r             nil)
                  (left                 (dolist (i lst result_l) (if (< i parent) (setf result_l (cons i result_l) ))))
                  (right                (dolist (i lst result_r) (if (> i parent) (setf result_r (cons i result_r) ))))
                )
            (append (list parent) (list (func_72 left)) (list (func_72 right)))
        ))
    )
)

* (func_72 '(1 2 3 4 5 6) )


(5      (3 (2 (1) NIL) (4))     (6))

        (3 (2 (1) NIL) (4))

    (3  (2 (1) NIL)     (4))

        (2 (1) NIL)

    (2      (1)     NIL)


"""-------------------------------------------------------------------------------------------"""
"""------------------                  Exercise 4.73                  ------------------"""
"""-------------------------------------------------------------------------------------------"""

Write a program that computes the subsequence with the largest sum in a sequence
of integers.

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_
                                                            )
                    )
                )

        )

    )
)


(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)

"""----------------------------------------------------------------------------------------"""
"""------------------                      END                       ------------------"""
"""----------------------------------------------------------------------------------------"""
```

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



"""#----------------------------------------------------------------------------------#"""
"""#---------------------          Exercise 6.1                ---------------------#"""
"""#----------------------------------------------------------------------------------#"""

Define a procedure VALS that takes a list of one argument procedures and an argu-
ment, and returns the values obtained by applying the procedures to the argument
in the given order.

(vals '(evenp log zerop)  8)        will return     (T    2.0794415   NIL)


(defun  vals    (list1    arg)

            (if     (endp list1)
                    nil
                    (cons   (funcall    (car list1)   arg)   (vals  (cdr list1)  arg) )
            )

)

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

(defun  vals    (list1    arg   storage)

            (if     (endp list1)
                    (reverse storage)
                    (vals  (cdr list1)  arg   (cons (funcall (car list1) arg) storage) )
            )

)


"""#----------------------------------------------------------------------------------#"""
"""#---------------------          Exercise 6.2                ---------------------#"""
"""#----------------------------------------------------------------------------------#"""

Define a procedure PAIRVALS that takes a list of one argument procedures and an
argument, and returns the list of dotted pairs where each procedure is paired with
the value obtained by applying it to the argument.

(pairvals '(f g h)  8)    should give   ( (F. 64)  (G. 512)  (H. 2.079234))


(defun  pairvals_   (list1    args   func_list)

        (if     (endp list1)
                nil
                (append     (list (cons (car list1)  (funcall (car  func_list) args)))

                            (pairvals_ (cdr list1) args (cdr func_list)))
        )



)

(defun func1 (x) (* x x))
(defun func2 (x) (* x x x))

(defun  pairvals    (list1   args)

        (pairvals_  list1  args  '(func1 func2 log) )
)

"""#----------------------------------------------------------------------------------#"""
"""#---------------------          Exercise 6.3                ---------------------#"""
"""#----------------------------------------------------------------------------------#"""

Define a procedure MAXPAIR that takes a list of dotted pairs and returns the maxi-
mum pair where the comparison is done on the basis of the second components of
pairs.

* (maxpair '((A . 2) (B . 8) (C . 4)))

(B . 8)

Note that :

* (cdr '(a . 0)  )
0

* (cdr '(a 0)  )
(0)


(defun  maxpair   (y  &optional (storage '(a . 0)))

    (if     (endp y)
            storage

            (if (< (cdr (car y)) (cdr storage))

                (maxpair   (cdr y)   storage)
                (maxpair   (cdr y)   (car y) )
            )
    )
)

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


(defun maxpair2 (lst)

    (reduce #' (lambda (x y)
```

```
        (if     (> (cdr x) (cdr y))
                x
                y
        ))
        lst
    )
)

"""#-----------------------------------------------------------------------------#"""
"""#---------------------          Exercise 6.4               ----------------------#"""
"""#-----------------------------------------------------------------------------#"""

Define a procedure that takes a list of predicate symbols (e.g. CONSP, NUMBERP etc.)
and an object, and returns the list of predicates that the object satisfies. Here is a
sample interaction:

(foo '(consp listp numberp) '(a)) ==> (CONSP LISTP)


(defun  func (list_pred  object   storage_success)

    (cond   ( (endp list_pred)                          storage_success)
            ( (funcall (car list_pred)  object)         (func (cdr list_pred) object (cons (car list_pred) storage_success)))
            ( t                                         (func (cdr list_pred) object storage_success))
    )
)

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

(defun func4 (pred obj)

    (mapcar #' (lambda (x)

        (if     (funcall x obj)
                x
                nil
        ))
    pred)
)


"""#-----------------------------------------------------------------------------#"""
"""#---------------------          Exercise 6.5               ----------------------#"""
"""#-----------------------------------------------------------------------------#"""

Define a procedure that takes a list of predicate symbols (e.g. CONSP, NUMBERP etc.)
and a list of objects, collects and returns all the objects that answer yes to at least
one predicate in the predicate list.


(defun odd_p (x)

    (cond   ( (and (integerp x) (oddp x))               t)
            ( t                                         nil)
    )
)

(defun even_p (x)

    (cond   ( (and (integerp x) (evenp x))              t)
            ( t                                         nil)
    )
)

(defun  func_   (list_pred  object)                             ; (func   '(odd_p even_p)    12  )

        (cond   ( (endp list_pred)                              nil)
                ( (funcall (car list_pred)  object)             t )
                ( t                                             (func_  (cdr list_pred)  object) )
        )
)


(defun  func    (list_pred   list_object  storage_success)   ; (func   '(odd_p even_p)    '(12 23  12.4  15.3 16 12.3)  nil)

        (cond   ( (endp list_object)                        storage_success)
                ( (func_ list_pred (car list_object))       (func  list_pred   (cdr list_object)  (cons (car list_object) storage_success)))
                ( t                                         (func  list_pred   (cdr list_object)   storage_success))
        )
)


; Any object that is not suitable for any function will give error. E.g., (oddp  12.3)

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

(defun func5 (list_pred   list_object result)

    (mapcar #' (lambda (obj)

        (dolist (i list_pred result)

                (if (funcall i obj)
                    (setf result obj)
                    nil
                )
        ))
        list_object)
)

Note: (funcall with #') will not work . Because it will be detected as (funcall #'')

"""#-----------------------------------------------------------------------------#"""
"""#---------------------          Exercise 6.6               ----------------------#"""
"""#-----------------------------------------------------------------------------#"""

Define a procedure that takes a list of one argument numerical procedures (define
your own and/or use the built-ins you know) and a number, and returns the name
of the procedure that yields the maximum value when applied to the number argu-
ment.
```

```
 (square   cube   sqrt   ... )    (3)   (27)


(defun square (x) (* x x))
(defun cube   (x) (* x x x))
(defun func   (x) (* 2 (* x x)))

(defun      find_max  (pred_list  x )

        (if     (endp pred_list)
                0
                (max  (funcall (car pred_list)  x)    (find_max   (cdr pred_list)  x ))
        )
)


"""#-----------------------------------------------------------------------------------#"""
"""#---------------------          Exercise 6.7               ----------------------#"""
"""#-----------------------------------------------------------------------------------#"""

Take a list and a procedure as input and return the list of indices of all the elements
that the procedure returns a non-nil value.

(func2  '(0 1 1 0 0 1 0)   #' zerop  )   ->   (0 3 4 6)


(defun   func2   (lst    pred  index  storage)

    (cond   ( (endp lst)                          (reverse storage))
            ( (funcall pred (car lst))            (func2 (cdr lst) pred (+ index 1) (cons index storage)))
            ( t                                   (func2 (cdr lst) pred (+ index 1)             storage))
    )
)


"""#-----------------------------------------------------------------------------------#"""
"""#---------------------                END                ----------------------#"""
"""#-----------------------------------------------------------------------------------#"""
```

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



"""#----------------------------------------------------------------------------------#"""
"""#---------------------          Exercise 7.1                ---------------------#"""
"""#----------------------------------------------------------------------------------#"""

Write LAMBDA expressions that

returns the greatest of two integers.

(lambda (x y) (if   (> x y)  x   y ) )

given two integers, returns T if one or the other divides the other without
remainder.

(lambda (x  y) (cond    ( (zerop (rem x y))            t)
                        ( (zerop (rem y x))            t)
                        ( t                            nil)
    ))

given a list of integers, returns the mean.

(lambda (lst)  (/ (reduce #'+ lst) (length lst)) )

given a list of integers, returns the sum of their factorials – use your factorial
solution.

(lambda (lst)  (reduce #' + (mapcar #' factorial lst)) ; assume we have factorial
)

"""#----------------------------------------------------------------------------------#"""
"""#---------------------          Exercise 7.2                ---------------------#"""
"""#----------------------------------------------------------------------------------#"""

Define a procedure PAIR-PROD using MAPCAR and LAMBDA, which takes a list of two
element lists of integers and returns a list of products of these pairs. E.g. an input
like

((7 8) (1 13) (4 1)) should yield (56 13 4).

(defun pair-prod (lst)

    (mapcar #' (lambda (x) (* (car x) (car (cdr x))) )  lst)

    )

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

(defun pair-prod2 (lst)

    (mapcar #' (lambda (x)

        (reduce #' *  x))
      lst)
)


"""#----------------------------------------------------------------------------------#"""
"""#---------------------          Exercise 7.3                ---------------------#"""
"""#----------------------------------------------------------------------------------#"""

Define a procedure that takes two lists as input and returns the list of their pairwise
averages. Use only MAPCAR, LAMBDA and arithmetic operations in your definition.

(defun func (x y)

    (mapcar  #' (lambda (a b)  (/ (+ a b) 2)  )  x  y )
)


"""#----------------------------------------------------------------------------------#"""
"""#---------------------          Exercise 7.4                ---------------------#"""
"""#----------------------------------------------------------------------------------#"""

Define your own REMOVE-IF.

* (remove-if #' oddp '(1 2 3 4 5 6) )

                        (2 4 6)


(defun  my_remove_if (func1 lst)

    (if (endp lst)
        nil
        (if     (funcall func1 (car lst))
```

```
                (my_remove_if func1 (cdr lst))
                (cons (car lst) (my_remove_if func1 (cdr lst)))
        )
    )
)
"""#----------------------          Second Way              ----------------------#"""

(defun  my_remove_if2 (func1 lst result)

    (dolist (i lst result)

        (if     (funcall func1 i)
                nil
                (setf result (cons i result))
        )
    )
)

"""#-------------------------------------------------------------------------------------#"""
"""#----------------------          Exercise 7.5                ----------------------#"""
"""#-------------------------------------------------------------------------------------#"""

Define LENGTH using MAPCAR, LAMBDA, + and APPLY.

'(1 2 3 4 5) -> 5

(defun my_length (lst)

    (apply  #' + (mapcar #' (lambda (x)
                                1
                            )
                            lst
                )
    )
)

"""#-------------------------------------------------------------------------------------#"""
"""#----------------------          Exercise 7.6                ----------------------#"""
"""#-------------------------------------------------------------------------------------#"""

Define a procedure that takes an integer n and gives a list of n random single digit
numbers. Use the built-in RANDOM, MAKE-LIST, MAPCAR and LAMBDA in your solu-
tion. Check the definition of the builtins you are not familiar with from reference
books on the website or on the web.

3   ->   '(1 9 4)

(defun  make_random_list (n)

        (mapcar  #' (lambda (x)
                    (random 10)
                    )
                    (make-list n))
    )

"""#-------------------------------------------------------------------------------------#"""
"""#----------------------          Exercise 7.7                ----------------------#"""
"""#-------------------------------------------------------------------------------------#"""

Define a procedure that takes two lists: a list N of numbers and a list P of symbols
with function bindings, i.e. symbols used to define some single argument mathe-
matical procedure with DEFUN. Your procedure should return a list with the same
size as N, whose elements are lists consisting of values obtained by applying all
the procedures in P to the corresponding element in N. For example, if you pro-
vide your procedure with a list of symbols naming square, absolute value and
float functions,

e.g. (sqr abs float), and the list (1 -2 3),

it should return:
                ((1 1 1.0) (4 2 -2.0) (9 3 3.0))

You are NOT allowed to use any procedure (built-in or user-defined)
other than #', DEFUN, MAPCAR, LAMBDA and FUNCALL


(defun   my_func (lst1    lst2)

            (mapcar  #' (lambda (x)
                (mapcar  #'  (lambda  (y)
                    (funcall y x)                   ; since y will be send like 'y  otherwise unbound. no need for #'
                                                    ; mapcar sends values like '(a b c d)   ->   'a 'b 'c 'd
                            )
                        lst2)
                        )
                lst1)
)


(my_func '(1 -2 3) '(sqrt abs float)  )    output value is

((1.0 1 1.0) (#C(0.0 1.4142135) 2 -2.0) (1.7320508 3 3.0))
```

```
"""#-----------------------------------------------------------------------------#"""
"""#---------------------          Exercise 7.8              ----------------------#"""
"""#-----------------------------------------------------------------------------#"""

Define a procedure APPLIER that takes a procedure "proc", an input "input" and a
count "cnt", and gives the result of applying proc to input cnt times. For instance,

(APPLIER #'CDR '(1 2 3) 2) should give (3)


(defun  applier (proc  input_ count_ )

    (dotimes (i count_ input_)

        (setf input_ (funcall proc input_))
    )
)
"""#---------------------          Second Way              ----------------------#"""

(defun  applier2 (proc  input_ count_)

    (last (mapcar #' (lambda (x)

        (setf input_ (funcall proc input_))
        )

        (make-list count_ :initial-element proc))
    )
)

"""#-----------------------------------------------------------------------------#"""
"""#---------------------          Exercise 7.9              ----------------------#"""
"""#-----------------------------------------------------------------------------#"""

Define a procedure MOST, which takes a list and a procedure argument, and returns
the element in the list that gives the highest score when provided as an argument
to the given procedure packed in a list with its score. To get full credit, solve the
task WITHOUT using recursion or iteration, you can use MAPCAR, REDUCE
and LAMBDA besides other built-ins you would need.

* ( most '(0.3 0.5 0.2) : proc # '( lambda ( x ) (* 2 ( log x ))))

            (0.5 -1.3862944)

(defun most (lst &key (proc #'(lambda (x) (* 2 (log x)))))

    (reduce #' (lambda (x y)

        (if     (> (cadr x) (cadr y))
                x
                y
        )       )

        (mapcar #' (lambda (z)

            (list z (funcall proc z))
            )
        lst)
    )
)

* (most '(0.3 0.5 0.2))

(0.5 -1.3862944)

"""#-----------------------------------------------------------------------------#"""
"""#---------------------          Exercise 7.10             ----------------------#"""
"""#-----------------------------------------------------------------------------#"""

The built-in FIND-IF returns the first element in its second argument that returns T
for its first argument:

* ( find-if #'  (lambda ( x ) ( > x 3))     '(1 3 9 0 4))

9

Define your own version of FIND-IF, which returns the index together with
the element. Remember that indexing starts with 0. For instance your procedure
should return (2 9) for the above invocation, where 2 is the index of 9.

(defun find-if-2 (proc lst)

    (let    ( (result                   nil)
              (len                      (length lst))
              (flag                     0)
            )

        (dotimes  (i len result)

            (if     (and (= flag 0) (funcall proc (nth i lst)))  ; computation efficiency
                    (and (setf result (list i (nth i lst))) (setf flag 1))
                    nil
```

```
            )
        )
    )
)

* ( find-if-2  #'(lambda ( x ) ( > x 3))    '(1 3 9 0 4)  )

(2 9)


"""#-----------------------------------------------------------------------------#"""
"""#----------------------          Exercise 7.11               ----------------------#"""
"""#-----------------------------------------------------------------------------#"""

Define a procedure REPLACE-IF, which takes three arguments: a list LST, an item
ITEM and a function TEST, and replaces every element of LST that passes the TEST
with ITEM. You may find using keyword arguments useful (see the lecture notes).
Make use of MAPCAR, LAMBDA and FUNCALL in your solution.

(replace-if '(1 0 1 0 1 0 1 1 1 0 0 1)  '1    'zerop)


(defun REPLACE-IF (lst arg test)

    (mapcar #' (lambda (x)

        (if     (funcall test x)
                arg
                x
        )
                )

    lst)
)

* (replace-if '(1 0 1 0 1 0 1 1 1 0 0 1)  '1    'zerop)

(1 1 1 1 1 1 1 1 1 1 1 1)

"""#-----------------------------------------------------------------------------#"""
"""#----------------------          Exercise 7.12               ----------------------#"""
"""#-----------------------------------------------------------------------------#"""

MAPCAR can work on any number of lists; you only need to be careful to provide a
function with the correct number of arguments. For instance

( mapcar # '( lambda ( x y ) (+ x y )) '(1 2 3) '(4 5 6))

gives (5 7 9). Don’t worry if lists are not of equal length, MAPCAR goes as far as
the shortest list.

Define procedures that use MAPCAR and LAMBDA and

• zip two lists together – (zip '(a b) '(1 2)) should give ((A 1) (B 2)).

(defun zip (lst1 lst2)

    (mapcar #' (lambda (x y)

        (list x y)
                )
    lst1 lst2)
)

(zip '(a b c) '(1 2 3) )

((A 1) (B 2) (C 3))

• take three lists: first two will be lists of integers, and the third is a list of
functions. Apply the corresponding function to corresponding arguments.

(func12 '(1 2 3 4) '(1 2 3 4)   '(+ * / - )  )     ->    (2  4  1  0)

(defun func12 (lst1 lst2 lst3)

    (mapcar #' (lambda (x y z)

        (apply z (list x y))
                )
    lst1 lst2 lst3)

)

(func12 '(1 2 3 4) '(1 2 3 4)   '(+ * / - )  )

                                (2 4 1 0)


"""#-----------------------------------------------------------------------------#"""
"""#----------------------      Differences Between Them        ----------------------#"""
"""#-----------------------------------------------------------------------------#"""
Funcall sees the elements one-by-one , even lists ! No loop !
```

```
* (funcall 'list  '(a b c) )                    ->          (list '(a b c))            ->  (A B C)
* (funcall 'list  '(a b c)  '(1 2 3) )          ->          (list '(a b c) '(1 2 3))   ->  (  (A B C) (1 2 3)  )

* (funcall 'car  '(a b c) )                     ->          A
* (funcall 'car  '(a b c) '(1 2 3) )            ->          ERROR !     Car takes one element.

Apply sees all the elements in the LAST list at once : like parallel working
If there is only one list. It sees everything in it together
Apply is stingy/miser. It wants everything. But if the outputs is too much.
It knows it can not eat all of that. It just wants the last aggregated data.
Loop for the last element if list !

* (apply 'list  '(a b c) )                      ->          (list 'a 'b 'c)            ->      (A B C)
* (apply 'list  '(a b c)  '(1 2 3) )            ->          (list '(a b c) '1 '2 '3)   ->      ((A B C) 1 2 3)


* (apply 'car  '(a b c) )                       ->          ERROR !     Car takes one element.

Mapcar is different. It takes and gives back. It is very generous/bountiful

* (mapcar 'list  '(a b c) )                     ->              ((A) (B) (C))
* (mapcar 'list  '(a b c)  '(1 2 3) )           ->              ((A 1) (B 2) (C 3))


* (funcall '+   1 2 3 )                             ->      6
* (apply   '+   1 2 3 '(4 5 6)  )                   ->      21   funcall would give error for this input.

* (mapcar  '+   '(1 2 3 )   )                       ->      (1 2 3)
* (mapcar  '+   '(1 2 3 )  '(4 5 6 )   )            ->      (5 7 9)
* (mapcar  '+   '(1 2 3 )  '(4 5 6 ) '(8 9 19))     ->      (13 16 28)

mapcar takes only lists !

"""#-------------------------------------------------------------------------------------#"""
"""#---------------------          Exercise 7.13              ----------------------#"""
"""#-------------------------------------------------------------------------------------#"""

                WHEN REAL MATHEMATICS COMES INTO THE GAME !

The way to toss a fair coin in LISP is to do (random 2), which would evaluate to
0 or 1 with a fifty-fifty chance.
Study the following procedure and indicate what the parameters n, f, c and s
stand for. In other words, describe what this procedure computes.

(lets assume n=3 and f=1 , and n=number of heads f=1 for head , then this function computes
many random coin tosses. If it reached the number of succeses = 3 . Then it returns th
number of trials
    )

(defun h (n f &optional (c 0) (s 0))

    (if     (= c n )
            s
            (h n f (+ c (if (= (random 2) f) 1 0)) (+ s 1))))



(defun func (number_success head_or_tail &optional (counter 0) (number_trials 0))

    (if     (= counter number_success)
            number_trials
            (func number_success head_or_tail (+ counter (if (= (random 2) head_or_tail) 1 0)) (+ number_trials 1))
    )
)
Lets try to get 10 Heads in x tosses !

* (func 10 1 )
19              ; found in 19 tosses.
* (func 10 1 )
28
* (func 10 1 )
17
* (func 10 1 )
19
* (func 10 1 )
17
* (func 10 1 )
16
* (func 10 1 )
19
* (func 10 1 )
22
* (func 10 1 )
18
* (func 10 1 )
19
* (func 10 1 )
22
* (func 10 1 )
23
* (func 10 1 )
17
```

Lets prove the function !

```
(defun prove (n counter storage)

    (if (= counter n)
        (float (/ (apply #'+  storage) (length storage)))
        (prove n (+ counter 1) (cons (func 10 1) storage))
    )
)

(prove 100 0 nil )
20.83

* (prove 1000 0 nil )
20.156

* (prove 10000 0 nil )
19.9234
```

We proved that to get 10 Heads we need to toss the coin nearly 20 times.
Because the probability is 0.5 .

```
"""#-----------------------------------------------------------------------------------------#"""
"""#---------------------          Exercise 7.14              ----------------------#"""
"""#-----------------------------------------------------------------------------------------#"""
```

Find the numbers in a given range that have the same Collatz length using the
techniques of this section.

```
(range = "[1 6)"    -> assume ( (1 1)  (2 5) (3 8) (4 99) (5 8) (6 99))    ->   (3 and 5) (6 and 4))

( defun collatz_generate ( n )

        ( if     (= n 1)
                '(1)
                ( cons n ( collatz_generate ( if    ( evenp n )
                                                    (/ n 2)
                                                    (+ (* n 3) 1))))
        )
)

* (collatz_generate 5)          ->      (5 16 8 4 2 1)


(defun collatz_length ( n )

    (- (length (collatz_generate n )) 1))

(collatz_length 5)              ->      5


I need a func like (range 2 6)    ->   (2 3 4 5)

(defun range (lower upper  storage)

    (if     (= lower upper)
            (reverse storage)
            (range (+ lower 1) upper (cons lower storage))
    )
)

(defun main (lower upper counter)


    (if     (= counter (- upper lower))
            nil

            (let*   ( (rnge                     (range lower upper nil))
                      (result                   nil)
                      (nt                       (nth counter rnge))
                    )

                (cons

                    (dolist (i rnge result)
                        (if     (= (collatz_length i) (collatz_length nt))
                                (setf result (cons i result))
                                nil
                        )
                    )
                    (main lower upper (+ counter 1))

                )
            )
    )
)

* (main 1 20 0)

((1) (2) (3) (4) (5) (6) (7) (8) (9) (10) (11) (13 12)
(13 12) (15 14) (15 14) (16) (17) (19 18) (19 18))
```

```
(defun temp (lst) (if (= (length lst) 1) t nil))


(defun main2 (lower upper)

    (remove-if #' temp (main lower upper 0))
)



"""#-----------------------------------------------------------------------------------#"""
"""#----------------------          Exercise 7.15               ----------------------#"""
"""#-----------------------------------------------------------------------------------#"""

* (sort '( (1 2) (1 3) (1 1) (1 6)) #'< :key #'cadr)

(sort   input     func_for_comparison       func_for_transform_input)

((1 1) (1 2) (1 3) (1 6))


( defun collatz_generate ( n )

        ( if     (= n 1)
                '(1)
                ( cons n ( collatz_generate ( if    ( evenp n )
                                                    (/ n 2)
                                                    (+ (* n 3) 1))))
        )
)

(defun collatz_length ( n )

    (- (length (collatz_generate n )) 1))


(defun sorted (lst)  (sort lst #'> :key #'collatz_length))

* (sorted '(1 5 4 3 2 6 7 8 9 ) )

(9 7 6 3 5 8 4 2 1)

"""#-----------------------------------------------------------------------------------#"""
"""#----------------------                END                ----------------------#"""
"""#-----------------------------------------------------------------------------------#"""
```

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

"""-----------------------------------------------------------------------------"""
"""------------------                  Exercise 8.1                  ------------------"""
"""-----------------------------------------------------------------------------"""

Define a procedure APPEND2 that appends two lists.

(a b c) (1 2 3)      ->     (a b c 1 2 3)

(defun append2 (lst1 lst2)

    (dolist (i (reverse lst1) lst2)

        (setf lst2 (cons i lst2))
    )
)

* (append2 '(a b c) '(1 2 3) )

(A B C 1 2 3)

"""-----------------------------------------------------------------------------"""
"""------------------                  Exercise 8.2                  ------------------"""
"""-----------------------------------------------------------------------------"""

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)))
        )
)

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

(defun chop_last2 (lst)

    (mapcar #' (lambda (x y)

        x)

    lst (make-list (- (length lst) 1)))

)

"""-----------------------------------------------------------------------------"""
"""------------------                  Exercise 8.3                  ------------------"""
"""-----------------------------------------------------------------------------"""


Define an iterative procedure UNIQ that takes a list and removes all the repeated
elements in the list keeping only the first occurrence. This is the expected behavior:

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

    (A B R C D)


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

    (cond   ( (null lst)                        nil)
            ( (member (car lst) (cdr lst))      (unique-list (cdr lst)))
            ( t                                 (cons (car lst) (unique-list (cdr lst))))
    )
)


"""-----------------------------------------------------------------------------"""
"""------------------                  Exercise 8.4                  ------------------"""
"""-----------------------------------------------------------------------------"""

Define a procedure that reverses the top-level elements of a list.

( a b c (d e f) g k l)   ->    (l k g (d e f) c b a)


(defun my_reverse (lst storage)

    (cond   ( (null lst)                        storage)
            ( t                                 (my_reverse (cdr lst) (cons (car lst) storage)))
    )
)


"""-----------------------------------------------------------------------------"""
"""------------------                  Exercise 8.5                  ------------------"""
"""-----------------------------------------------------------------------------"""

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)

The first element 3 is the mean of the list (3), the second element 4 is the mean
of (3 5), and so on. Implement RUN-MEAN by using DOTIMES and NTH.


(defun run_mean (lst)

    (let    ( (result                           nil)
              (cum_total                        0)
              (mean                             0)
            )
        (dotimes    (i (length lst) (reverse result))

            (setf cum_total   (+ (nth i lst) cum_total))
            (setf mean  (float (/ cum_total (+ i 1))))
            (setf result (cons mean result))
        )
    )
)

"""----------------------------------------------------------------------------------------"""
"""------------------                  Exercise 8.6                  ------------------"""
"""----------------------------------------------------------------------------------------"""


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. As usual, positioning starts with 0. Use DOTIMES. 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 a b a b))
                        '(2 1 0)

(defun search_pos (lst1 lst2 counter storage)

    (let    ( (result                   t)
              (flag                     0)
              (len_1                    (length lst1))
              (len_2                    (length lst2))
            )

        (cond   ( (> len_1 len_2)                       (reverse storage))
                (t

                  (if

                      (dotimes    (i len_1 result)

                          (if     (and (equal (nth i lst1) (nth i lst2)) (= flag 0))
                                  (setf result t)
                                  (and (setf result nil) (setf flag 1))
                          )
                      )
                      (search_pos lst1 (cdr lst2) (+ counter 1) (cons counter storage))
                      (search_pos lst1 (cdr lst2) (+ counter 1) storage)
                  )
                )
        )
    )
)


"""----------------------------------------------------------------------------------------"""
"""------------------                  Exercise 8.7                  ------------------"""
"""----------------------------------------------------------------------------------------"""

Define a procedure that reverses the elements in a list including its sublists as well.


( a b c (d e f) g k l)   ->    (l k g (f e g) c b a)


(defun my_reverse2 (lst storage)

    (cond   ( (null lst)                            storage)
            ( (listp (car lst))                     (append (my_reverse2 (cdr lst) nil)  (list (my_reverse2 (car lst) nil)) storage))
              (t                                    (my_reverse2 (cdr lst) (cons (car lst) storage)))
    )
)


"""----------------------------------------------------------------------------------------"""
"""------------------                  Exercise 8.8                  ------------------"""
"""----------------------------------------------------------------------------------------"""

Write a procedure LAST-NTH that returns the nth element from the end of a given
list. Do NOT use NTH or ELT; use DOLIST.

2   '(6 5 4 3 2 1 0)        ->          2   (last 2 = first 4)    (length - 1) - n


(defun last-nth (n lst)

    (let    ((result            nil)
             (len               (- (- (length lst) 1) n))
             (counter           0)
            )
```

```
        (dolist (i lst result)

            (if     (= counter len)
                    (and (setf result (nth counter lst)) (setf counter (+ counter 1)))
                    (setf counter (+ counter 1))
            )
        )
    )
)

"""-------------------------------------------------------------------------------------------"""
"""------------------                  Exercise 8.9                  ------------------"""
"""-------------------------------------------------------------------------------------------"""

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

* (pairlists '((a b) (= =) (1 2) (+ -) (3 9)))

               ((A = 1 + 3) (B = 2 - 9))


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 8.10                 ------------------"""
"""-------------------------------------------------------------------------------------------"""

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

CL-USER > ( enumerate '( A B C ))

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




(defun range (x   storage)

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

(range 5 nil)

(0 1 2 3 4)


(defun enum_ (lst)

    (mapcar #' (lambda (x y)

        (list x y)
        )
    lst (range (length lst) nil))
)

"""-------------------------------------------------------------------------------------------"""
"""------------------                  Exercise 8.11                 ------------------"""
"""-------------------------------------------------------------------------------------------"""


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 )          ->      (d e f)


(defun sub-seq (lst start end )

    (let    ( (result                       nil)
            )
        (dolist  (i lst (reverse result))

            (if  (and (= start 0) (>= end 0))
                 (and (setf result (cons i result)) (setf end (- end 1)) )
                 (and (setf start (- start 1)) (setf end (- end 1)))
            )
```

```
        )
    )
)

"""-------------------------------------------------------------------------------------------"""
"""------------------                  Exercise 8.12                  -------------------"""
"""-------------------------------------------------------------------------------------------"""

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 (lst)

    (if     (not (= (car lst) 0) )
            (zeros (cdr lst))

            (let    ( (result              0)
                      (flag                0)
                    )

                (dolist  (i lst (- result 1))

                    (if   (and (= i 0) (= flag 0))
                            (setf result (+ result 1))
                            (setf flag 1)
                    )
                )
            )
    )
)

"""-------------------------------------------------------------------------------------------"""
"""------------------                      END                  -------------------"""
"""-------------------------------------------------------------------------------------------"""
```