```
"""#--------------------------------------------------------------------------------------#"""
"""#----------------------         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)
```

```lisp
  (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
```

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