```
"""#-------------------------------------------------------------------------------------#"""
"""#----------------------      METU Cognitive Sciences        ----------------------#"""
"""#----------------------         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)
       )
  )


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

```lisp
; The factorial of a non-negative integer is defined as follows:


(defun factorial (x)

  (if  (= x 0)
       1
       (* (factorial (- x 1))    x)
       )
  )


  """-----------------------------------------------------------------------------------------"""
  """-------------------                   Exercise 3.4                   -------------------"""
  """-----------------------------------------------------------------------------------------"""
; Define a recursive procedure that computes the sum of the squares of the first n
; non-negative integers.


(defun sumOfSquares (x)

  (if (= x 1)
      1
      (+  (sumOfSquares (- x 1))  (* x x) )
      )
  )


  """-----------------------------------------------------------------------------------------"""
  """-------------------                   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.


(defun toss (n)

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


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


(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 3.9                -------------------"""
"""-----------------------------------------------------------------------------------------"""

(defun exponential (x y)

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




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

; The Fibonacci numbers


(defun fib (n)

  (if  (<  n  2)
       n
       (+  (fib (- n 1))  (fib (- n 2))  )
       )
  )


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


      )
  )


"""-----------------------------------------------------------------------------------------"""
"""-------------------                    Exercise 3.12                -------------------"""
"""-----------------------------------------------------------------------------------------"""
; Sum of a geometric progression.  a.r^0 + a.r^1 + a.r^2 ... + a.r^n = a (r^0 + r^1 + ... +r^ n)


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

      )
  )
```

```
  """----------------------------------------------------------------------------------------"""
  """-------------------                   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                  -------------------"""
  """----------------------------------------------------------------------------------------"""
; 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 zo (x sum)

  (if (= x 1)

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

    (if (primep x)
      (and  (print "Prime found : ")  (print x)   (zo (collatz x)  (+ x sum )   )  )
      (zo (collatz x) sum )
    )
  )
)


(defun zoo (x)

  (zo 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.


(defun collatz (x)

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

```lisp
(defun collatz-diff (x)

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

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