```
 1
 2  ; Exercise 3.1 (*)
 3
 4  ; Define a procedure GUESS. It will have one parameter, an integer including and
 5  ; between 0 and 99. You will make the computer make successive guesses to find
 6  ; this number, where each guess will appear on the screen – use PRINT for this. You
 7  ; will need the LISP expression (random 100) to make a guess. Needless to say,
 8  ; the only acceptable way to go on making guesses as long as needed is keep calling
 9  ; yourself.
10
11
12  (defun guess (x)
13
14    (let ((y    (random 100)))
15
16      (cond ((= x y)               x)
17            ((print y)       (guess x))
18                     )
19            )
20      )
21
22  ; Exercise 3.2
23  ; Define a procedure that multiplies two integers using only addition as a primitive
24  ; arithmetic operation.
25
26
27  (defun mltp (x y)
28
29    (if  (= x 0)
30         0
31         (+ (mltp (- x 1)   y)   y)
32         )
33    )
34
35
36
37
38  ; Exercise 3.3
39  ; The factorial of a non-negative integer is defined as follows:
40
41
42  (defun factorial (x)
43
44    (if  (= x 0)
45         1
46         (* (factorial (- x 1))    x)
47         )
48    )
49
50
51  ; Exercise 3.4
52  ; Define a recursive procedure that computes the sum of the squares of the first n
53  ; non-negative integers.
54
55
56  (defun sumOfSquares (x)
57
58    (if (= x 1)
59        1
60        (+  (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 2)    (print (random 2))   )
         (toss (- n 1))
         (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)
       )
  )


```

```
121
122
123  ; Exercise 3.10
124
125  ; The Fibonacci numbers
126
127
128  (defun fib (n)
129
130    (if  (<  n  2)
131         n
132         (+  (fib (- n 1))  (fib (- n 2))  )
133         )
134    )
135
136
137  ; Exercise 3.11
138
139  ; Newton’s Method
140
141
142  (defun getnewY (x y)
143
144    (let  (   (newY       (/    (+ (/  x  y)  y)     2)  )   )
145      newY
146      )
147    )
148
149
150  (defun newton (x newY)
151
152    (print "initial guess : " )
153
154    (print newY)
155
156    (if (<=   (abs (-   x    (* newY  newY) ))     0.00001  )
157        newY
158
159        (newton x  (getnewY x  newY))
160
161
162        )
163    )
164
165
166  ; Exercise 3.12
167  ; Sum of a geometric progression.  a.r^0 + a.r^1 + a.r^2 ... + a.r^n = a (r^0 + r^1 + ...
     +r^ n)
168
169
170  (defun geo (a r n)
171    (if (= n 0)
172        a
173        (+    (* a    (expt r n)    )    (geo a r (- n 1))   )
174
175        )
176    )
177
178
179
```