```
MAIN()
1  while (true)
2      do command := GETCOMMAND()
3         INITIALIZETOKENIZER(command)
4         newcommand := PREPROCESSING()
5         INITIALIZETOKENIZER(newcommand)
6         root := READ()
7         result := EVAL(root)
8         PRINTRESULT(result, true)
```

```
PREPROCESSING()
 1  Initial newcommand to empty string
 2  while (token := GETNEXTTOKEN()) is not empty
 3      do if token is "define"
 4           then newcommand := CONCATENATE(newcommand, " define")
 5                token := GETNEXTTOKEN()
 6                if token is "("
 7                  then token := GETNEXTTOKEN()
 8                       newcommand := CONCATENATE(newcommand, token,
                                                   " (lambda (", PREPROCESSING(), ")")
 9                  else procPutback(token)

10         elseif token is "'"
11           then newcommand := CONCATENATE(newcommand, "(quote")
12                number_of_left_paren := 0
13             do
14                token := GETNEXTTOKEN()
15                newcommand := CONCATENATE(newcommand, token)
16                if token is "("
17                  then number_of_left_paren := number_of_left_paren + 1
18                elseif token is ")"
19                  then number_of_left_paren := number_of_left_paren - 1
20                while (number_of_left_paren > 0)
21           else newcommand := CONCATENATE(newcommand, token)
22  return newcommand
```

```
READ()
 1  Initialize root and first to NIL and true, respectively.
 2  token_hash_value := GETHASHVALUE(GETNEXTTOKEN())
 3  if token_hash_value is LEFT_PAREN
 4    then while (token_hash_value := GETHASHVALUE(GETNEXTTOKEN())) is not RIGHT_PAREN
 5              do if first is true
 6                   then root := temp := ALLOC()
 7                   else Array[temp].rchild := ALLOC()
 8                        temp := Array[temp].rchild
 9                        first := false
10                 if token_hash_value = LEFT_PAREN
11                   then PUTBACK()
12                        Array[temp].lchild := READ()
13                   else Array[temp].lchild := token_hash_value
14         if first is false
15           then Array[temp].rchild := NIL
16         return root
17    else return token_hash_value
```

```
EVAL(root)
 1  token_index := GETHASHVALUE(Memory[root].lchild)
 2  if (token_index = PLUS)
 3     then return GETHASHVALUE(GETVAL(EVAL(Momory[Memory[root].rchild].lchild))
                   +GETVAL(EVAL(Memory[Memory[Memory[root].rchild].rchild].lchild)))

 4  elseif (token_index = MINUS)
 5     then return GETHASHVALUE(GETVAL(EVAL(Momory[Memory[root].rchild].lchild))
                   −GETVAL(EVAL(Memory[Memory[Memory[root].rchild].rchild].lchild)))

 6  elseif (token_index = TIMES)
 7     then return GETHASHVALUE(GETVAL(EVAL(Momory[Memory[root].rchild].lchild))
                   ∗GETVAL(EVAL(Memory[Memory[Memory[root].rchild].rchild].lchild)))

 8  elseif (token_index = isEQ)                  // eq?
 9     then return EVAL(Momory[Memory[root].rchild].lchild)
                   = EVAL(Memory[Memory[Memory[root].rchild].rchild].lchild)

10  elseif (token_index = isEQUAL)               // equal?
11     then return CHECKSTRUCTURE(EVAL(Momory[Memory[root].rchild].lchild),
                   EVAL(Memory[Memory[Memory[root].rchild].rchild].lchild))

12  elseif (token_index = isNUMBER)
13     then if ISNUMBER(EVAL(Memory[Memory[root].rchild].lchild)) is true
14             then return TRUE
15             else  return FALSE

16  elseif (token_index = isSYMBOL)
17     then if result := EVAL(Memory[Memory[root].rchild].lchild) is true
                   and ISNUMBER(result) is false
18             then return TRUE
19             else  return FALSE

20  elseif (token_index = isNULL)
21     then if Memory[root].rchild is NIL
22             then return TRUE
23             else  return FALSE

24  elseif (token_index = CONS)
25     then newmemory := ALLOC()
26          Memory[newmemory].lchild := EVAL(Memory[Memory[root].rchild].lchild)
27          Memory[newmemory].rchild :=
                        EVAL(Memory[Memory[Memory].root].rchild].rchild].lchild)
28          return returnmemory

29  elseif (token_index = COND)
30     then while Memory[root].lchild is not NIL
31               do root := Memory[root].rchild
32                  if (EVAL(Memory[Memory[root].lchild].lchild) = TRUE)
33                    then return EVAL(Memory[Memory[root].lchild].rchild)
34          if Memory[Memory[Memory[root].rchild].lchild].lchild is not ELSE
35            then ERROR()
36          return EVAL(Memory[Memory[Memory[Memory[root].rchild].lchild].rchild].lchild)
```

CONTINUE($root$)

```
1  elseif (token_index = CAR)
2     then return Memory[EVAL(Memory[Memory[root].rchild].lchild)].lchild

3  elseif (token_index = CDR)
4     then return Memory[EVAL(Memory[Memory[root].rchild].lchild)].rchild

5  elseif (token_index = DEFINE)
6     then if function define
7     then hashTable[Memory[Memory[root].rchild].lchild].pointer :=
                        Memory[Memory[Memory[root].rchild].rchild].lchild
8     else  hashTable[Memory[Memory[root].rchild].lchild].pointer :=
                        EVAL(Memory[Memory[Memory[root].rchild].rchild].lchild)

9  elseif (token_index = QUOTE)
10    then return Memory[Memory[root].rchild].lchild

11 elseif token_index is user defined function
12    then push current values to stack            // refer to slide
13         set parameter by function argument
14         result:=EVAL(Memory[Memory[Memory[Memory[root].lchild].rchild].rchild].lchild)
15         pop the values from stack
16         return result
```

PRINT($index, startList$)

```
1  if root is NIL
2     then print "()"
3  elseif root < 0
4     then print hashTable[root].symbol
5  elseif root > 0
6     then if startList is true
7             then print "("
8          PRINT(Memory[root].lchild, true)
9          if Memory[root].rchild is not NIL
10           then PRINT(Memory[root].rchild, false)
11           else  print ")"
```