SCHEME INTERPRETER PSEUDO CODE

```
Procedure Main()
begin
1.  while (true)
2.     command := GetCommand()
3.     InitializeTokenizer(command)
4.     newcommand := Preprocessing()
5.     InitializeTokenizer(newcommand)
6.     root := Read()
7.     result := Eval(root)
8.     if result is not NIL
9.        PrintResult(result)
end

Procedure Preprocessing()
begin
1.  newcommand := empty string
2.  while (token := GetNextToken()) is not empty
3.     if token is "define"
4.        newcommand := Concatenate(newcommand, "define ")
5.        token := GetNextToken()
6.        if token is "("
7.           token := GetNextToken()
8.           return Concatenate(newcommand, token, "(lambda ( ", Preprocessing(), ")")
9.        else PutBack(token)
10.    else if token is ""
11.       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.             number_of_left_paren := number_of_left_paren + 1
18.          else if token is ")"
19.             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
```

```
Procedure Read()
1.  root := NIL
2.  first := true
3.  token_hash_value := GetHashValue(GetNextToken())
4.  if token_hash_value is LEFT_PAREN
5.     while (token_hash_value := GetHashValue(GetNextToken))) is not RIGHT_PAREN
6.        if first is true
7.           temp := Alloc()
8.           root := temp
9.        else
10.          Memory[temp].rchild := Alloc()
11.          temp := Memory[temp].rchild
12.          first := false
13.       if token_hash_value = LEFT_PAREN
14.          PushBack()
15.          Memory[temp].lchild := Read()
16.       else Memory[temp].lchild := token_hash_value
17.    if first is false
18.       Memory[temp].rchild := NIL
19.    return root
20. else return token_hash_value
```

```
Procedure Eval(root)
1.  if isNumber(root) is true
2.     return root
3.  if root < 0
4.     return hashTable[getIndex(root)].pointer
5.  token_index := Memory[root].lchild
6.  switch (token_index)
7.     case PLUS :                                          //+
8.        argument1 := GetValue(GetFirstArgument(root))
9.        argument2 := GetValue(GetSecondArgument(root))
10.       return GetHashValue(argument1 + argument2)
11.    case MINUS :                                         //-
12.       argument1 := GetValue(GetFirstArgument(root))
13.       argument2 := GetValue(GetSecondArgument(root))
14.       return GetHashValue(argument1 − argument2)
15.    case PRODUCT :                                       //*
16.       argument1 := GetValue(GetFirstArgument(root))
17.       argument2 := GetValue(GetSecondArgument(root))
18.       return GetHashValue(argument1 * argument2)
19.    case LESSTHAN :                                      //<
20.       argument1 := GetValue(GetFirstArgument(root))
21.       argument2 := GetValue(GetSecondArgument(root))
22.       if argument1 < argument2
23.          return TRUE
24.          else return FALSE
25.    case MORETHAN :                                      //>
26.       argument1 := GetValue(GetFirstArgument(root))
27.       argument2 := GetValue(GetSecondArgument(root))
28.       if argument1 > argument2
29.          return TRUE
30.          else return FALSE
31.    case EQUALTO :                                       //=
32.       argument1 := GetValue(GetFirstArgument(root))
33.       argument2 := GetValue(GetSecondArgument(root))
34.       if argument1 = argument2
35.          return TRUE
36.          else return FALSE
37.    case EQ :                                            //eq?
38.       argument1 := GetFirstArgument(root)
39.       argument2 := GetSecondArgument(root)
40.       if argument1 = argument2
41.          return TRUE
42.          else return FALSE
43.    case EQUAL :                                         //equal?
44.       argument1 := GetFirstArgument(root)
45.       argument2 := GetSecondArgument(root)
46.       return CheckStructure(argument1, argument2)
```

```
47.   case NUMBER :                                    //number?
48.     if isNumber(root) is true
49.       return TRUE
50.       else return FALSE
51.   case SYMBOL :                                    //symbol?
52.     if isSymbol(root) is true
53.       return TRUE
54.       else return FALSE
55.   case NULL :                                      //null?
56.     if isNull(root) is true
57.       return TRUE
58.       else return FALSE
59.   case CAR :                                       //car
60.     return Car(root)
61.   case CDR :                                       //cdr
62.     return Cdr(root)
63.   case CONS :                                      //cons
64.     newmemory := Alloc()
65.     Memory[newmemory].lchild := GetFirstArgument(root)
66.     Memory[newmemory].rchild := GetSecondArgument(root)
67.     return newmemory
68.   case COND :                                      //cond
69.     return Cond(root)
70.   case DEFINE :                                    //define
71.     Define(root)
72.     return NIL
73.   case QUOTE :                                     //quote
74.     return Memory[root].rchild
75.   case DISPLAY :                                   //display
76.     printResult(Eval(Memory[root].rchild))
77.     return NIL
78.   default :
79.       if token_index is user defined function
80.         if user defined function is not format of lambda function
81.           MakeLambdaFormat(root)
82.         PushAndSetVariables(root)
83.         Execute(root)
84.         PopAndResetVariables(root)
```

Procedure GetFirstArgument(root)
begin
1. **return** Eval(Memory[Memory[root].rchild].lchild)

Procedure GetSecondArgument(root)
begin
1. **return** Eval(Memory[Memory[Memory[root].rchild].rchild].lchild)

Procedure getIndex(index)
begin
1. **return** -1∗index

Procedure PushAndSetVariables(root)
begin
1. variableName := Memory[Memory[Memory[root].lchild].rchild].lchild
2. variableValuePosition := Memory[root].rchild
3. position : = 0
4. **whie** variableValuePosition is not NIL
5. variableValue[position] := Eval(Memory[variableValuePosition].lchild)
6. variableValuePosition := Memory[variableValuePosition].rchild
7. position := position + 1
8. position := 0
9. **while** variableName is not NIL
10. stackElement.name = Memory[variableName].lchild
11. stackElement.value = hashTable[getIndex(variableName)].pointer
12. Push(stackElement)
13. hashTable[getIndex(variableName)].pointer := variableValue[position]
14. variableName := Memory[variableName].rchild
15. position := position + 1

Procedure PopAndResetVariables(root)
begin
1. variableTrace := Memory[Memory[Memory[root].lchild].rchild].lchild
2. **while** variableTrace is not NIL
3. stackElement := Pop()
4. hashTable[getIndex(stackElement.Name)].pointer := stackElement.Value
5. variableTrace := Memory[variableTrace].rchild

Procedure MakeLambdaFormat(root)
begin
1. newroot := Alloc()
2. Memory[newroot].lchild := hashTable[getIndex(Memory[root].lchild)].pointer
3. Memory[newroot].rchild := Memory[root].rchild
4. **return** newroot