(‘LISP’)
A PASCAL IMPLEMENTATION FOR PEDAGOGICAL PURPOSES

Dissertation submitted to the Jawaharlal Nehru University
in partial fulfilment of the requirements for the
award of the Degree of
MASTER OF PHILOSOPHY

YALA KISHAN REDDY

SCHOOL OF COMPUTER AND SYSTEMS SCIENCES
JAWAHARLAL NEHRU UNIVERSITY
NEW DELHI-110067, INDIA
1983

CERTIFICATE

The research work embodied in this dissertation has been carried out in the School of Computer and Systems Sciences, Jawaharlal Nehru University, New Delhi. The work is original and has not been submitted in part or full for any other degree or diploma of any University.

Y Kishan Reddy
(Y. Kishan Reddy)
Student

R. Sadananda.
(Dr. R. Sadananda)
Acting Dean
School of Computer and
Systems Sciences
Jawaharlal Nehru University
New Delhi - 110067.
INDIA

R. Sadananda.
(Dr. R. Sadananda)
Supervisor

ACKNOWLEDGEMENTS

It is indeed a pleasure to acknowledge the guidance of Dr. R. Sadananda. His scholarly guidance, enthusiastic encouragement and constructive criticism enabled me to learn much from this experience.

For access to the computer and other facilities, I am grateful to the Dean, School of Computer and Systems Sciences, Jawaharlal Nehru University.

My thanks due to Mr. Mohan Reddy and Mr. Krishna Rao of N.I.C. for introducing me to CYBER-170 system at National Informatics Center, New Delhi. My thanks also due to all my colleagues and friends for their encouragement in carrying out this work.

Mr. S.K. Sapra deserves special mention for his prompt and efficient typing.

ykishan Reddy

(Y. Kishan Reddy)

New Delhi
October, 1983

CONTENTS

II - LISP FUNCTIONS

III - IMPLEMENTATION

IV - DISCUSSION AND CONCLUSIONS

CHAPTER - I

(INTRODUCTION AND DEFINITIONS)

1.1 INTRODUCTION :

Symbol manipulation activity is taking a central role in computing sciences. It is thus, increasingly being considered that computer is a symbol manipulator as opposed to the view of computer as a number cruncher. This view point is more general, and enables computers to handle ever increasingly complex data-structures and sophisticated descriptive schema. This is so, particularly in the areas such as : algebraic formula manipulation, information retrieval, computational linguistics, automatic decision making, Artificial Intelligence, Medical diagnostics, Robotics and other important applications. Several papers in the litrature describe the advantages and techniques of symbol processors [1-4].

1.2 WHAT IS SYMBOL MANIPULATION ?

Symbol manipulation is a branch of computing concerned with the manipulation of unpredictably structured data. Most scientific and business data processing is characterised by the manipulation of data of known length and format. In contrast, the size and format of the data involved in symbol manipulation are not known in advance and vary greately during the run of a program. These data are in the form of variable length lists. A list is a sequence of elements each of which is a data item. A multilevel list is one in which the data items may themselves be lists. The latter are called sublists of the multilevel lists. An overview of the state of the art in symbol manipulation can be found in ACM [7].

1.3 REQUIREMENTS OF SYMBOL MANIPULATING LANGUAGES :

Any Programming Language for symbol manipulation must meet two major requirements. First, there must be appropriate way of representing lists both on paper (the external representation) and in the computer memory (the internal representation). Second, there must be appropriate functions, statement types, subroutines, and other linguistic devices for specifying operations on lists.

The operations common to all symbol manipulation languages are those involving the creation and decomposition of lists. At a minimum, it must be possible to create a list by combining existing elements of lists, and to extract a portion of a list. A general exposition of symbol manipulation languages, using LISP as an example was written by Abrahams, P. [6].

1.4 LISP AND IT'S SYMBOL MANIPULATING FEATURES :

Of all existing programming languages available to-day LISP, perhaps, comes closest to being a symbol manipulating language [9]. LISP was originally designed by John McCarthy [5] a mathematician, and the purpose was to develop a mathematically complete and sound language. It is also designed to allow and infact it encourages recursive programming.

The following are some of the important features of LISP :

1. LISP has got a high-level notation for lists,
2. LISP is oriented towards programming at a terminal with rapid response. All programs and all data can be displayed or altered at will,

3. LISP functions and LISP data have the same form. One LISP function can analyse another and a set of other functions. One LISP function can synthesis a set of other LISP functions which happens to be the basis of automatic programming.
4. Over a period of last few decades most of the well known work in the area of Artificial Intelligence has been carried out in LISP, and therefore the best tools for editing and debugging are available with LISP.

1.5 SCOPE OF THE DISSERTATION :

This work presents a PASCAL implementation of LISP, as an experiment for developing software in a high-level structured language. PASCAL is a general purpose language designed by Niklaws wirth, and has come-up as a result of the movement for structured programming. PASCAL has powerful data types and encourages a Top-down design methodology. Because of these, and other reasons PASCAL is now available widely and there is an increasing number of users who are defecting from FORTRAN to PASCAL. Some of the advanced general purpose languages which are being now developed have many common features with PASCAL.

1.6 LANGUAGE PROCESSING :

Definition : The software using which the computer uses to understand the commands in an artificial language, supplied by the user is generally termed as the "LANGUAGE PROCESSOR".

Language processing can be broadly classified into two types :

1. Translation,
2. Interpretation

TRANSLATOR : A Translator is a program that translates a source language program into its equivalent object language program.

Assemblers, Compilers, and Conversion Programs are come under this category.

INTERPRETER : An Interpreter is a program that accepts a source language program, written-in-a highlevel language, and appears to execute it, as if it were in machine language form and produces the corresponding result as its out-put.

More precisely, an interpreter repeatedly executes the following sequence.

1. Get the next statement,
2. Determine the actions to be executed,
3. Perform the actions.

This sequence is very similar to the pattern of actions carried out by a traditional computer; that is,

1. Fetch the next instruction
   (i.e. the instruction whose address is specified in the PC) and increment it.
2. Decode the instruction.
3. Execute the instruction.

This similarity shows that interpretation can be viewed as a simulation, on a host computer, of a special purpose machine whose machine language is the higher level language. Pure interpretation and pure translation are two extremes. In practice, many languages are implemented by a combination of the two techniques. In a purely interpretive solution, executing a statement may require a fairly complicated decoding process to determine the operations to be executed and their operands. In most cases, this process is identical each time the statement is encountered. Consequently, if the statement appears in a frequently executed part of a program (e.g., an-inner loop), the speed of execution is strongly affected by the identical decoding process. On the other hand, pure translation generates machine code for each high level statement. In doing so, the translator decodes each high-level statement once only. Frequently used parts are then decoded many times in their machine language representation. Since this is done efficiently by hardware, pure translation can save processing time over pure interpretation. However, language processing by interpretation is deferred until data attributes have been bound. This makes interpreters particularly easy to construct, and they are therefore widely used despite execution-time inefficiencies. Virtually all processors for LISP and for APL and most of those for SNOBOL are interpreters.

1.7 REPRESENTATION OF LISTS :

We first consider the external representation of lists. For specialized lists such as character strings and algebraic expressions, there are natural written representations.

Thus a character string may be written by writing down the characters one after another enclosing the entire group in quote marks to show where it begins and ends. An algebraic expression may be written, for example, in one of the forms used for arithmatic expressions in scientific programming languages.

For more general lists, the most frequently used written representation of a list written in sequence, delimited by blanks and enclosed in perantheses. Thus,

(RAT 2 CAT)

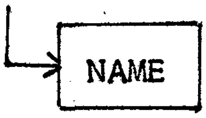
represents the list whose three elements are the character string RAT, the number 2, and the character string CAT.

((RAT 3) (CAT 5))

represents a list whose elements are two sublists. Each of these sublists in turn has two elements.

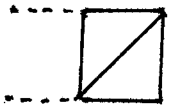
Now, we shall study how lists are represented in the computer memory. Lists are stored as structural forms built out of computer words as a parts of trees. In representing list structures in the computer memory there are two possibilities for a computer word, which may be either an atom or a list. An atomic word is a string of atmost ten characters. Whereas a list word is a rectangle devided into two sections called the "head" and the "tail". Where "head" and "Tail" are addresses that point to some other S-expressions.

Now we represent the atomic word 'NAME' in the computer as

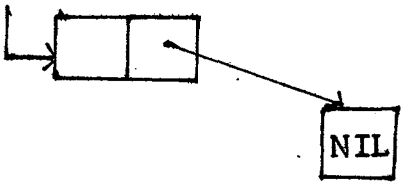


: atomic word :

It is convenient to indicate NIL by

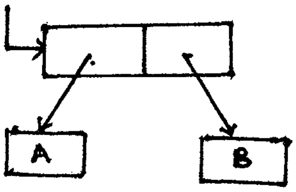


instead of

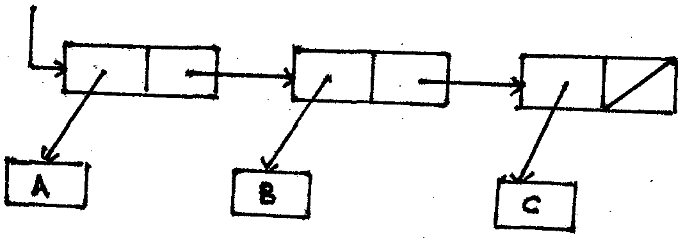


Following are some diagrammed S-expressions, shown as they would appear in the computer.

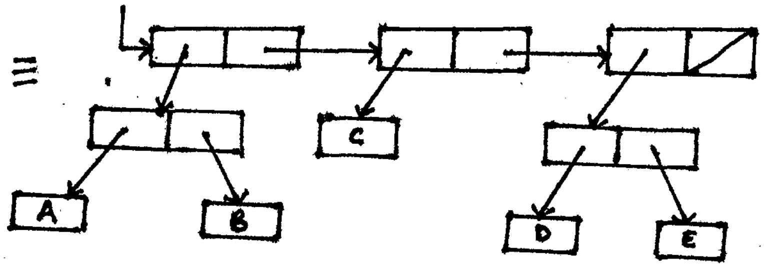
(A.B) ≡



(A B C) ≡

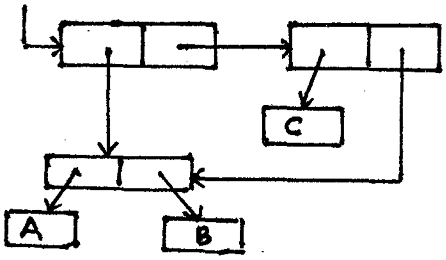


((A.B) C (D.E)) ≡

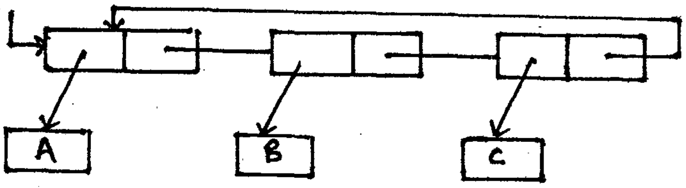


It is possible for lists to make use of common subexpressions.

((A.B) C (A.B)) could also be represented as :



Circular lists are ordinarily not permitted. They may not be read in. However, they can occur inside the computer as the result of computations involving certain functions. Their printed representation is infinite in length. For example, the structure:



will print as:

(A B C A B C . . . .)

CHAPTER - II

(LISP FUNCTIONS)

2.1 THE LISP LANGUAGE :

LISP is a formal mathematical language. It is therefore possible to give a concise, yet complete description. LISP differs from most programming languages in three important ways. The first way is in the nature of the data. In the LISP language, all data are in the form of symbolic expressions usually referred as S-expressions. S-expressions are of indefinite length and have a binary tree type structure, so that significant subexpressions can be readily isolated. In the LISP programming system, the bulk of available memory is used for storing S-expressions in the form of list structures. This type of memory organisation frees the programmer from the necessity of allocating storage for the different sections of this program.

The second important part of the LISP language is the source language itself, which specifies in what way the S-expressions are to be processed. This consists of recursive functions of S-expressions.

Third, LISP can interpret and execute programs written in the form of S-expressions. Thus like machine language, and unlike most other higher level languages, it can be used to generate programs for further execution.

2.1.1 SYMBOLIC EXPRESSIONS :

The most elementary type of S-expression is an atomic symbol. An atomic symbol is a string of no more than ten (thirty in standard LISP 1.5) characters.

The following are atoms :

A

APPLE

STRING

LONGSTRING, etc.,

<u>S-expression</u> : An S-expression is either an atomic symbol or it is composed of these elements in the following order :

a left peranthesis, an S-expression, either a dot followed by an S-expression and a right peranthesis, or a space followed by an S-expression and a right peranthesis or only a right peranthesis.

The following are S-expressions :

(ATOM)

(A.B)

(A (B C))

(A B C), etc.,

A LISP program is itself an S-expression. It is functional in that it is composed of applications of functions that produce results that may be used by other functions.

2.2 BASIC FUNCTIONS OF LISP :

There are very few primitive functions provided in pure LISP. Existing LISP systems have added to this list considerably. These new functions, however, can all be expressed in terms of the original primitive functions.

QUOTE is the identity function. It returns its (single) argument as its value. This function is needed because the atom 'A' does not represent itself but is the name of a Memory Location. The QUOTE function allows its argument to be treated as a constant. Thus, (QUOTE A) in LISP is analogus to 'A' in conventional languages.

EX: (QUOTE A) = A

(QUOTE (A B C)) = (A B C) etc.,

The most common functions are those that manipulate lists :

The function CAR has one argument. Its value is the first element of its composit argument.

EX : (CAR (QUOTE (A B C))) = A

(CAR (QUOTE ((A B) C D))) = (A B)

CAR of an atomic symbol is undefined and therefore it will give an ERROR.

The function CDR has one argument.

CDR returns all elements of its composit argument except the first.

EX : (CDR (QUOTE (A))) = NIL

(CDR (QUOTE (A B C))) = (B C)

(CDR (QUOTE ((A B) (C D E))) = ((C D E)) etc.

CDR of an atomic symbol is not defined, and gives an ERROR.

The function CONS has two arguments, and is used to build bigger S-expression from the two smaller ones.

```
EX :  (CONS (QUOTE A) (QUOTE B))  =  (A.B)
      (CONS (QUOTE A) (QUOTE (B C D)))  =  (A B C D)
      (CONS (QUOTE (A B)) (QUOTE (C D))) =  ((A B) C D)
                                               etc.,
```

In LISP, the values 'true' and 'false' are represented by the atomic symbols 'T' and 'NIL' respectively. Therefore a predicate in LISP is a function whose value is either 'T' or 'NIL'.

Let us consider some elementary predicate functions in LISP :

The predicate ATOM is true if its argument is an atomic symbol, and false otherwise.

```
EX :  (ATOM (QUOTE A))  =  T
      (ATOM (QUOTE (A)))=  NIL
```

The predicate EQ is a test for equality on atomic symbols. It checks whether its two atomic arguments are equal. It returns 'T' if they are equal, and NIL otherwise. It's value is 'NIL' for non-atomic arguments.

```
EX :  (EQ (QUOTE A) (QUOTE A))  =  T
      (EQ (QUOTE A) (QUOTE B))  =  NIL
      (EQ (QUOTE (A)) (QUOTE (A)))  =  NIL
```

In LISP programming system the conditional expression is a device for providing branches in function definitions, and is used to define a larger class of functions.

A conditional expression has the following form :

$$(\mathrm{COND}\ (P_1\ e_1)\ (P_2\ e_2)\ \ldots\ (P_n\ e_n))$$

Where each P_i is an expression, whose value may be either 'T' or 'NIL', and each e_i is any expression. The meaning of a conditional expression is the following :

It is evaluated by evaluating the P_i in turn until one is found whose value is 'T'. The value of the entire form is then obtained by evaluating the corresponding e_i. None of the other e_i's are evaluated, nor are any of the P_i following the first true one.

If none of the P_i are true, then the value of the entire expression is undefined. Instead an ERROR signal will come out.

Each P_i or e_i can itself be either an S-expression, a function, a composition of functions or may itself be another conditional expression.

```
EX :  (COND ((ATOM (QUOTE (A))) (QUOTE B))
      ((EQ (QUOTE A) (QUOTE A)) (QUOTE FOUND))
      (T (QUOTE NOTFOUND))) =  FOUND
```

LAMBDA :

A function is represented in the form :

$$(\text{LAMBDA } (x_1\ x_2\ .\ .\ .\ x_n)\alpha)\quad E_1\ E_2\ .\ .\ .\ E_n)$$

Where :

x_1, x_2, . . . x_n are dummy variables that appear in the expression α, and E_1, E_2 . . . E_n are values corresponding to x_1, x_2 . . . x_n respectively.

The evaluation of the expression is done by substituting E_i for the corresponding x_i.

The variables in a LAMBDA expression are dummy or bound variables because systematically changing them does not alter the meaning of the expression.

```
EX :  ((LAMBDA (X y) (CONS X Y))
            (QUOTE A) (QUOTE (B C)))
                   = (A B C)
```

2.3 A UNIVERSAL LISP FUNCTION :

A universal function is one that can compute the value of any given function applied to its arguments when given a description of that function. Such a function here is LABEL.

In order to permit recursive functions to be expressed in closed form an additional device is needed. Evaluation of the form,

(LABEL f α)

Yields the function 'α' (which must be a LAMBDA expression) and in addition associates the function name f (which must be an ATOM) with 'α' so that during the application of 'α' to arguments, any occurence of 'f' evaluates to 'α'. Thus a function may be made recursive by naming it via LABEL and then using this name within the definition, i.e., within the Lambda expression.

```
EX :  ((LABEL MEMBER
          (LAMBDA (X Y)
      (COND ((NULL Y) (QUOTE NIL))
          ((EQ X (CAR Y)) (QUOTE T))
      ((QUOTE T) (MEMBER X (CDR Y))))))
      (QUOTE A) (QUOTE (C B A)))
              =  T
```

The above defined function MEMBER checks whether the list contains the given atom.

2.4 EXTENDED LISP FUNCTIONS :

Though, higher order functions can be derived from the primitive LISP functions it is not feasible to define a function of interest each time we need it.

Here are some additional LISP functions which are frequently encountered in problem solving situations.

APPEND : The function APPEND has two arguments. It strings together the elements of lists supplied as its arguments.

```
EX :  (APPEND (QUOTE (A B)) (QUOTE (C D))
                         = (A B C D)
      (APPEND (QUOTE (A (B C))) (QUOTE (D E)))
                         = (A (B C) D E)
```

LIST : The function LIST also has two arguments. It does not run things together like APPEND does. Instead, it makes a list out of its arguments. Each argument becomes an element of the new list.

```
EX : (LIST (QUOTE (A B)) (QUOTE (C D)))
                                = ((A B) (C D))
     (LIST (QUOTE A) (QUOTE B) = (A B)
```

SUBST : SUBST is a function which makes substitution possible in LISP.

consider the form

```
(SUBST (QUOTE X) (QUOTE Y) (QUOTE Z))
```

Where X, Y and Z are S-expressions which means, SUBST replaces all the occurences of 'Y' in the list Z by the value X.

```
EX : (SUBST (QUOTE A) (QUOTE (B C)) (QUOTE (A (B C)))
                                =  (A A)
(SUBST (QUOTE (A B)) (QUOTE (B C)) (QUOTE (A (B C))))
                                =  (A (A B))
```

THE PROGRAM FEATURE :

The LISP 1.5 program feature allows the user to write an ALGOL-Like program containing LISP statements to be executed.

An example of the program feature can be seen in defining the function REV, that reverses the elements of a given list.

```
((DEFINE (REV LAMBDA (X)
(PROG (U V)
((SETQ U X)
((A) (COND ((NULL U) (RETURN V))
      ((QUOTE T) ((SETQ V (CONS (CAR U) V))
   ((SETQ U (CDR U))
(GO  ((A))))))))))))
(REV (QUOTE (A B C))))   =  (C B A)
```

The program form has the structure -
(PROG, List of program vqriables, sequence of statements and labels).

The first list after the function name, PROG, is a list of program variables. If there are none then this should be written as 'NIL' or ( ) (an empty list). Program vqriables are treated much like bound variables, but they are not bounded by LAMBDA. The value of each program variable is 'NIL' until it has been set to some thing else.

SETQ : To assign a value to the program variable, we have the form SETQ. To set a variable X to the value (A B), we write the following S-expression :

((SETQ X (QUOTE (A B))) E)

Where 'E' is another function which contains 'X' as one of its arguments. If 'E' is replaced by 'X' in the above S-expression it returns the value (A B).

The function RETURN causes a normal end of a program. The argument of RETURN is evaluated, and gives the result of the whole S-expression. No further statements are executed.

In our implementation a label symbol is of the form ((A). Where 'A' is a label symbol. Go is a form used to cause a transfer.

(GO ((A))) will cause the program to continue at the statement following ((A).

DEFINE : It is possible to associate a value with any identifier. In the case of an identifier whose value is a function, the association is created through use of the LISP function DEFINE. Normally, a LISP program consists of a sequence of applications of functions to arguments. Thus, in-order to create a complicated function using a number of subfunctions, DEFINE is used to associate the definition of each function with its name. Any of these functions may refer to any other function or to itself by name within its definition. An example (run on computer) describing the DEFINE feature is given in Appendix – B .

LIST STRUCTURE OPERATORS : LISP is made general in terms of list structure by means of the basic list operators "REPLACAR" and "REPLACDR". These operators can be used to replace the 'CAR' or 'CDR' or any word in a list. The expression,

```
(REPLACAR (QUOTE ((A B) B C)) (QUOTE A))
```

replaces the CAR part of the list ((A B) B C)) with the second argument i.e. A.

Therefore the result would be (A B C)

In terms of value, REPLACAR can be described by the expression

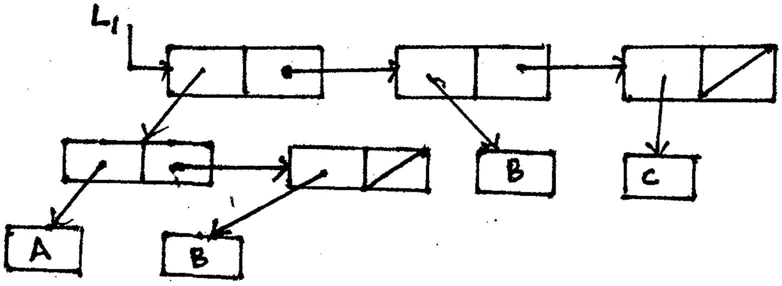
```
(REPLACAR (QUOTE X) (QUOTE Y)
      ≡ (CONS (QUOTE Y) (CDR (QUOTE X)))
```

But the effect is quite different. On operating REPLACAR, there is no CONS involved, and a new word is not created.

This can be diagramatically shown as follows :

Let l_1 = ((A B) B C)

which can be represented as:

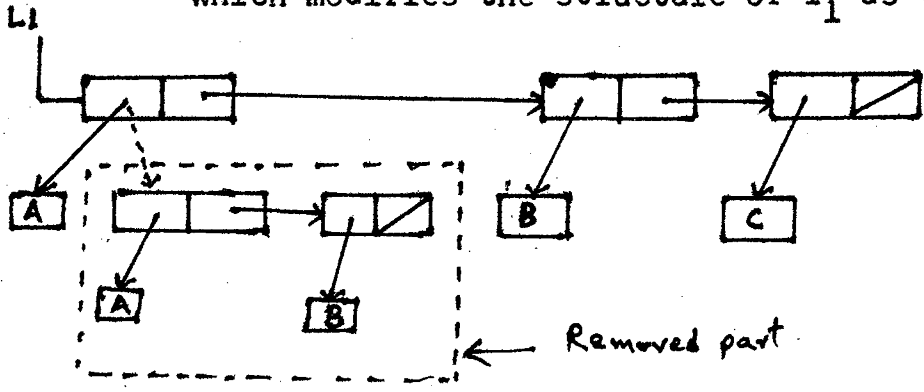


and

l_2 = A ≡

Now (REPLACAR (QUOTE l_1) (QUOTE l_2)

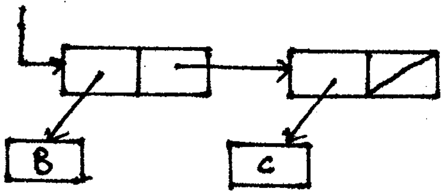
which modifies the structure of l_1 as the following :

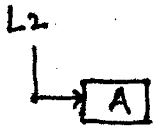


Whereas (CONS (QUOTE l_2) (CDR (QUOTE l_1)))

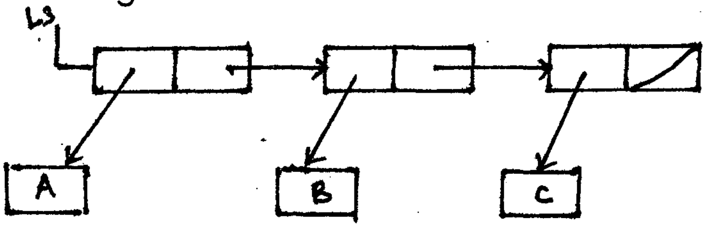
constructs another list l_3 out of the two given S-expressions i.e. the resultant list l_3 would be of the form :

(CDR (QUOTE l_1))





Now CONS of the above two S-expressions give a new list l_3 as follows :



Note that on CONS operation the original list structure of l_1 has not been changed.

In a similar way, the function

(REPLACDR (QUOTE X) (QUOTE Y))

replaces the CDR part of list X by the S-expression 'Y'.

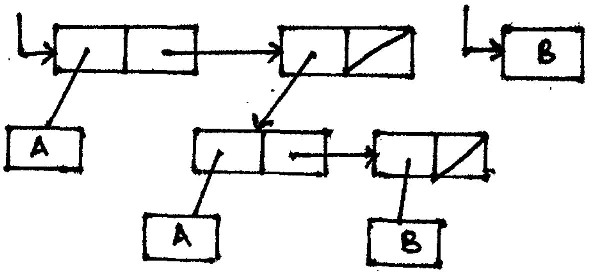
EX :

(REPLACDR (QUOTE (A (A B))) (QUOTE B))

returns the value as:

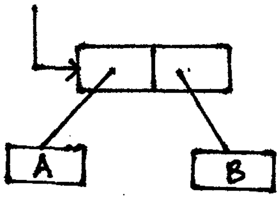
= (A B)

X = (A (A B))   Y = B



Operation on REPLACDR causes:

X = (A . B)



These operators (REPLACAR, and REPLACDR) must be used with caution. They can permanently alter existing list structures and other basic memory. They can be used to create circular lists, which can cause infinite printing, and look infinite to functions that search, such as "EQUAL" and "SUBST".

A few more predicates which are frequently encountered in LISP programs are the following :

EQUAL : The predicate EQUAL, which overrides EQ in usage, is the test for equality of its two arguments that are any S-expressions.

EX : (EQUAL (QUOTE A) (QUOTE A)) = T
(EQUAL (QUOTE (A B)) (QUOTE (A B))) = T
(EQUAL (QUOTE (A B)) (QUOTE (A C))) = NIL
etc.

The function 'EQ' is applicable only for atomic symbols.

NULL : The predicate NULL is useful in deciding whether a list is exhausted. It's value is true only if it's argument is 'NIL'.

EX : (NULL NIL) = T
(NULL (CDR (QUOTE (A)))) = T
(NULL ( )) = T
(NULL (CAR (QUOTE (A)))) = NIL

LOGICAL CONNECTIVES :

The Logical or Boolean connectives are usually considered as primitive operators. However, in LISP, they can be defined by using conditional expressions.

In the system, 'NOT' is a predicate of one argument. However, 'AND' and OR are predicates of an indefinite number of arguments, and therefore are special forms.

The value of 'AND' is 'T' only when each of its argument's value is true, 'NIL' otherwise.

```
EX :   (AND (QUOTE T) (QUOTE NIL))  =  NIL
       (AND (QUOTE T) (QUOTE T))  =  T
```

The value of OR is 'NIL' only when each of its arguments value is 'NIL', 'T' otherwise.

```
EX :   (OR (QUOTE NIL) (QUOTE T))  =  T
       (OR (QUOTE NIL) (QUOTE NIL)) = NIL
```

The value of NOT is 'T' if its arguments value is 'NIL' and viceversa.

```
EX :   (NOT (QUOTE NIL))  =  T
       (NOT (QUOTE T))  =  NIL
```

CHAPTER - III

(IMPLEMENTATION)

3.1 MEMORY ORGANIZATION :

In a list processing system it is hot feasible to create free nodes (words) each time we need to store items and to destroy (dispose) these nodes after they become no more useful. This process is crude and inefficient in both memory management and execution time.

The easiest way to keep track of available list storage is by use of a free-list, a list of all unused words. At system initialization, we chain all of available blocks together into a free list. Whenever we want to add a new item to an active-list (The concept of active-list, is the list structure, of the input S-expression, and the environment in which the values and identifiers are bounded during the run time of a program), we remove the first block from the free-list and use it to store the new item. And the words which are no more active, i.e., as soon as the execution of the S-expression is over, are automatically returned to the free-list by a technique called as "Garbage collection". We shall discuss about this technique later in this chapter.

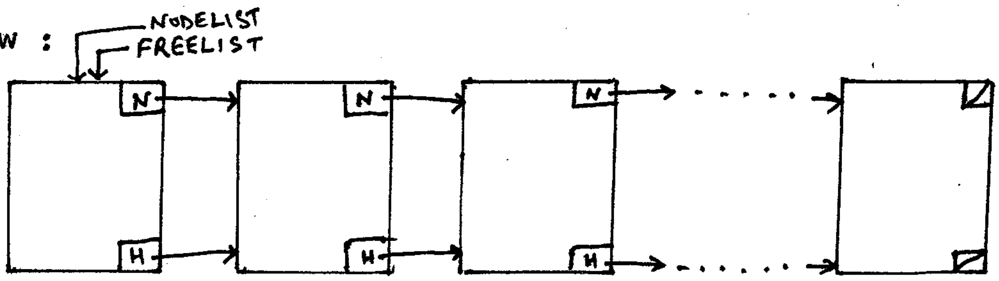
In our LISP programming system, we made use of the free-list concept. The data types POINTER, and RECORD in PASCAL provide the best mechanism to construct linked lists and other dynamic data structures. In our LISP processing system "SYMBOLIC EXPRESSION" is a "RECORD TYPE" (Lines 17 to 25 in Appendix A) which has a tag field "ANATOM" is always checked before accessing either the name field or the "HEAD" and "TAIL" fields of a word.

During the system initialization the free-list is constructed as follows :

The loop in the "PROCEDURE INITIALISE" :

```
FREELIST : = NIL ;
FOR  I :=  1 TO MAXNODES DO
    BEGIN
       NEW (NODELIST) ;
       NODELIST  NEXT := FREELIST ;
       NODELIST  HEAD := FREELIST ;
       NODELIST  STATUS := UNMARKED ;
       FREELIST := NODELIST
    END ;
```

Constructs a free-list containing the number of words equal to MAXNODES in computer memory appeared to be as shown below :



Where N stands for the pointer field NEXT,

and H stands for the pointer field HEAD.

Notice that the 'status' of all the nodes if UNMARKED

Where "MAXNODES" is any Natural number. There is a limitation in declaring the maximum number of free words. Since, the available computer memory is to be shared among input-output buffers, Interpreter program and the free-list.

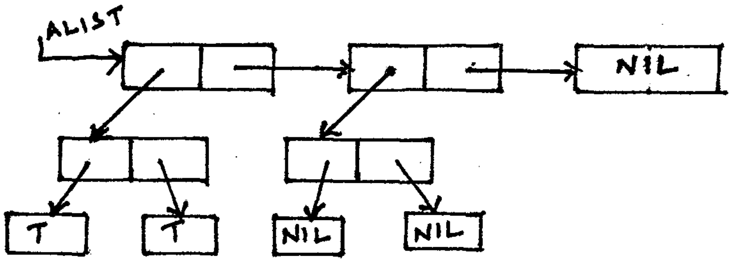
3.2 ENVIRONMENT :

As we have seen that each item in an S-expression (LISP Program) is to be evaluated unless or otherwise it is quoted by the function "QUOTE". Now, the questions arise,

Where do the values of identifiers and functional variables lie ?

How are they bounded to each other ? and lastly, how are they evaluated ?

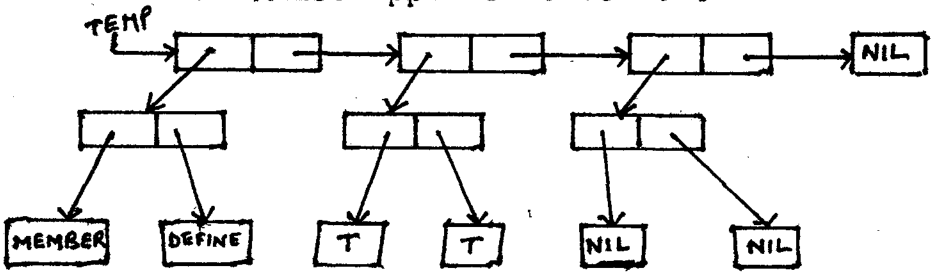
All these questions can be answered with the concept of an association-list. An association-list is (a list structure in a binary form) an environment to evaluate an S-expression, in the sense that, it contains all definitions of indentifiers (and values for the functional variables). In our LISP processing system we represent the association-list as ALIST, and henceforth it is continued to be call with this name. During the system initialization the 'ALIST' is constructed with nine nodes in the form as shown below:



Regarding the second question, the ALIST should have to have a common property that the identifiers and functional variables with their definitions (or values) should be connected in the 'ALIST' such that a single function can traverse all the existing identifiers and function variable names. The definitions (or values) to the identifiers (or variables) should be their neighbouring sublists.

Consider the initial ALIST structure, which contains the identifiers 'T' and 'NIL'. During the evaluation of these identifiers their values would be their neighbouring sublists i.e. 'T' & 'NIL' respectively.

Therefore, for example, if we want to attach one more identifier 'MEMBER' with 'DEFINE' as its value. The resultant ALIST appears to be as :



Where TEMP is the current environment, ALIST, grows dynamically during the evaluation process of LISP program. The functions LAMBDA, and PROG bind the variable to their corresponding values in the above mentioned manner and attach them to the ALIST. Similarly the forms LABEL, and DEFINE associate the definitions of identifiers (i.e., newly defined function names) with their identifiers on the ALIST. The function SETQ assigns new values to the program variables on the 'ALIST' during its run.

During the evaluation of a LISP program if any identifier is encountered the function LOOKUP (Lines 465 to 481 in Appendix A) searches for its name on the ALIST from left to right. If it is found then the function LOOKUP gives its corresponding value as the result and the process continues. If the identifier is not found on the ALIST, then the evaluation is terminated, and gives the indication that the function is not defined.

3.3 MARK/SWEEP GARBAGE COLLECTION :

Garbage collection is an effective, although (at first) apparently brutal solution to storage management. It presumes that every node in the heap is available until proven used. This is effected by a mark bit, initially cleared, in every node. Every active pointer in a register of the interpreter is taken as the root of used structure, and every such structure is traversed and marked. After the mark phase, the heap is swept sequentially; unset mark bits indicate available nodes (garbage) to be returned to available space.

The traversal of each structure requires time proportional to its size. Conventional traversal algorithms treat each structure as a tree to be traversed in preorder, where atoms, null pointers, and already marked nodes are taken as external nodes (leaves). A node is marked on its first visit. Knuth 1975 explains several algorithms, of which the lost, due to Deutsch, Schorr, and Waite [12] is the most elegant because it uses no extra stack in its traversal. Space being at a premium, the stack is maintained in reversed tree pointers that are restored as the stack is popped. A PASCAL version of this algorithm was developed by COX & TAYLOR [11] for their primitive LISP System.

3.4 LISP INPUT - OUTPUT :

Reading a list and storing it in the computer memory as a structural forms, and to print out a stored expression in the same notation are done by the procedures, READEXPR, and PRINTEXPR respectively.

3.4.1 READING AN S-EXPRESSION :

The procedure READEXPR (lines 216-261 in Appendix A), reads in a symbolic expression and stores it in the computer memory in a binary tree form. It pops the required number of free words from the free-list to store the symbolic expression, one word at a time. Procedure "READEXPR" inturn calls two other procedures namely NEXTSYMBOL, and BACKUPINPUT. Procedure NEXTSYMBOL reads the next input symbol from the input file. The type of the input symbol is defined by the global type "INPUTSYMBOL". The global variable "SYM" returns the type of the present symbol and transfers control to the procedure READEXPR. Procedure "BACKUPINPUT" puts an additional left peranthesis in the stream of input symbols to facilitate the procedure READEXPR during the read of an S-expression. BACKUPINPUT is called each time whenever the type of the next symbol, read from the S-expression, is other than a period. This additional left peranthesis would not be printed out, as it was actually not there in the input expression.

Symbolic expressions are read and stored in the appropriate structure using the following grammer for symbolic expressions :

$$\begin{aligned}\text{S-expr.} &= \langle\text{Atom}\rangle \\ &\text{of } (\langle\text{S-expr}\rangle\ .\ \langle\text{S-expr}\rangle) \\ &\text{or } (\langle\text{S-expr}\rangle\ \langle\text{S-expr}\rangle \ldots \\ &\qquad\qquad \ldots \langle\text{S-expr}\rangle)\end{aligned}$$

The third rule follows an alternative form of S-expression called the list notation.
For example, consider the following S-expression

$(l_1\ l_2\ l_3 \ldots l_n)$

This S-expression can be represented in the list notation with the same meaning as :

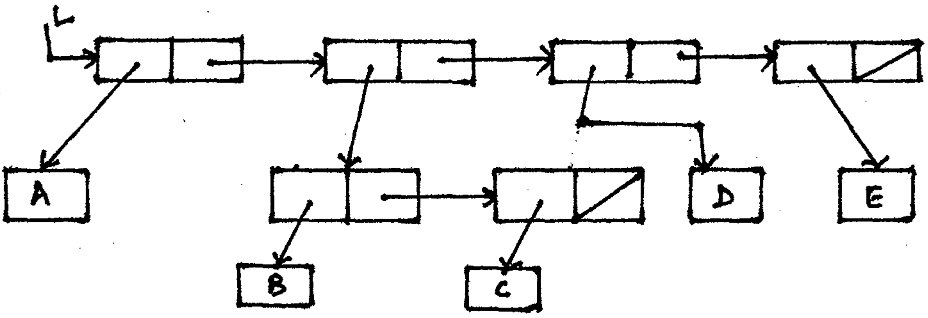
$(l_1 . (l_2 . (l_3 . (\ldots (l_n . \text{NIL}) \ldots))))$

EX : Let a list

$l = (A\ (B\ C)\ D\ E)$

on executing the instruction

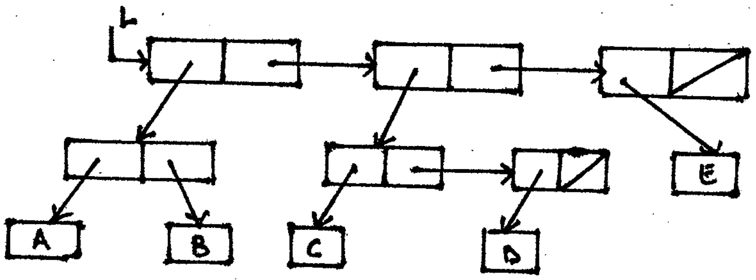
READEXPR (l), reads 'l' as input and stores in the computer memory, in a form appears to be as :



3.4.2 PRINT AN S-EXPRESSION :

Procedure "PRINTEXPR" (Lines 271-297 in Appendix A). Prints an S-expression which was stored in the computer memory through the procedure READEXPR. PRINTEXPR in turn uses another procedure called 'PRINTNAME'. Procedure PRINTNAME Prints out an atomic symbol each time it is called.

EX : the list '1' of the following structure :



On operating "PRINTEXPR" this will be printed out in the following form :

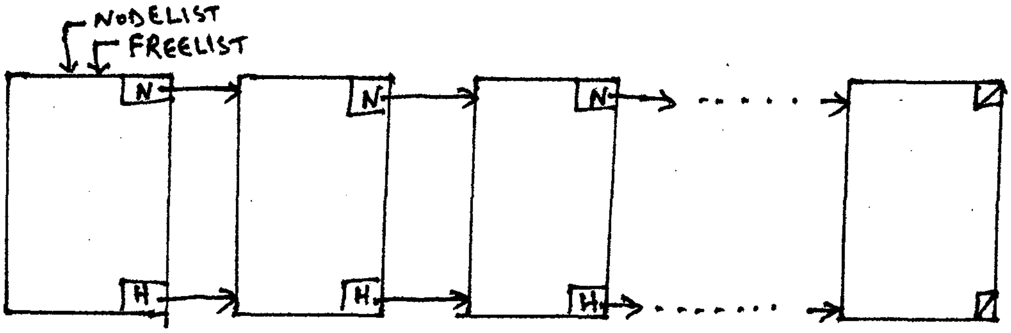
((A . B) (C D) E)

3.5 PROCEDURE POP : (Lines 129-139 in Appendix A)

The procedure 'POP' takes a word from the free-list (from one end), and stores its address at the location of its pointer argument. This word will be further used either to store an item or to link two nodes. The operations performed by the procedure 'POP' are the following :

It checks whether the free-list is completely exhausted. If it is yes, then the program is terminated and gives an indication to the user that, "NOT ENOUGH SPACE TO EVALUATE THE EXPRESSION". If the free-list is not completely exhausted, then, it removes the link between the HEAD pointer of the first word from it's next available word. Decreases the number of freenodes by 1. Saves the address of the first word in a location which is a pointer argument of the procedure 'POP'. And, the address of the free-list is changed to the address of it's next available word. The action of 'POP' operation on free-list can be diagramatically shown as :

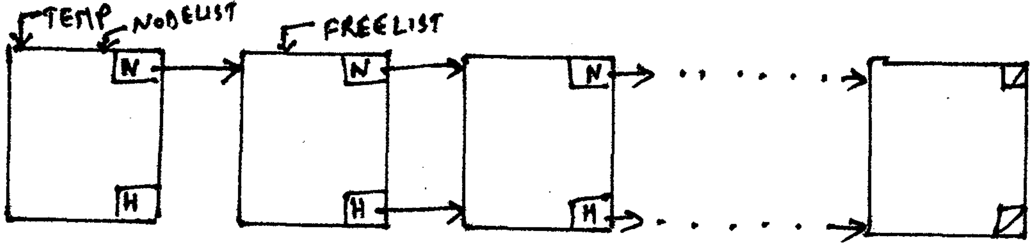
Consider the free-list of the form



Where 'N' stands for the pointer field "NEXT",

and 'H' stands for the pointer field "HEAD".

Now the operation POP (TEMP) will give the resultant free-list of the form ;



Note that the link from the left-most word's 'HEAD' pointer to it's next word has been removed.

3.6 PROCEDURE INITIALISE : (Lines 699-813 in Appendix A)

The procedure INITIALSE arranges an initial environment that is required by other procedures and functions in the interpreter program during the process of a LISP expression.

It assigns the boolean variable 'ALREADYPACKED' to 'FALSE', reads a character from the input file and writes it in the output file. It constructs a free-list, a list of available words, containing the number of words equal to the global constant 'MAXNODES' (refer sec-3.1). It assigns the global reserved words to their corresponding LISP functions (Lines 737 to 762 in Appendix A). Procedure 'INITIALISE' also constructs the initial structure of ALIST (association list) as explained in Section-3.2).

3.7 EVAL FUNCTION : (Lines 298-698 in Appendix-A)

The structure of the function EVAL is a case analysis on the syntactic type of the expression being evaluated. Function EVAL scans each word by walking the tree in a left-to-right depth first manner and classifies the words into functions, preudo functions, identifiers, and labels, and then performs their corresponding operations by calling its several local functions accordingly. This function scans the list of clauses of a case analysis, recursively evaluating the predicate part of each clause to see if it is true. If a predicate part is true, the action sequence of that part is executed. If a predicate part is not true, the scan continues. Running out of clauses to try gives an error at some point.

Now, let us study about the different local functions defined in the function EVAL, and their usage in evaluating their corresponding LISP functions.

The following functions in the PASCAL Program :

REPLACEH, REPLACET, HEAD, TAIL, CONS,
APPEND, EQQ, EQUAL, LIST, SUBST, NULL,
ATOM & NAT are called to perform the operations of their corresponding LISP functions :

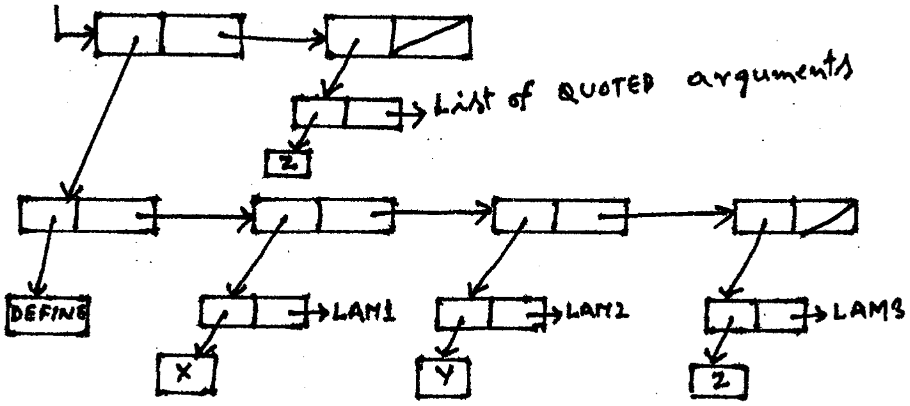
REPLACAR, REPLACDR, CAR, CDR, CONS,
APPEND, EQ, EQUAL, LIST, SUBST, NULL,
ATOM and NOT respectively.

Function LOOKUP is called in case either the function EVAL's first argument (Hereafter it is denoted as 'PTR') is an atom, or the CAR of the 'PTR' is an atom and is not a reserved LISP function. Function LOOKUP searches for the corresponding value of an identifier or variable in the 'ALIST'. An identifier may be a newly defined function using the LISP Pseudo-functions LABEL or DEFINE, or a variable bounded by the functions LAMBDA or PROG in LISP Language.
The function 'SEARCH' is called to perform the actions of the function 'DEFINE' in LISP. It attaches the 'ALIST' (association list) to the tail of the father of the last identifier in the sublist, (which contains the definitions of all the newly defined functions) and the root of the present 'ALIST' becomes the father of the first identifier of the definitions sublist.

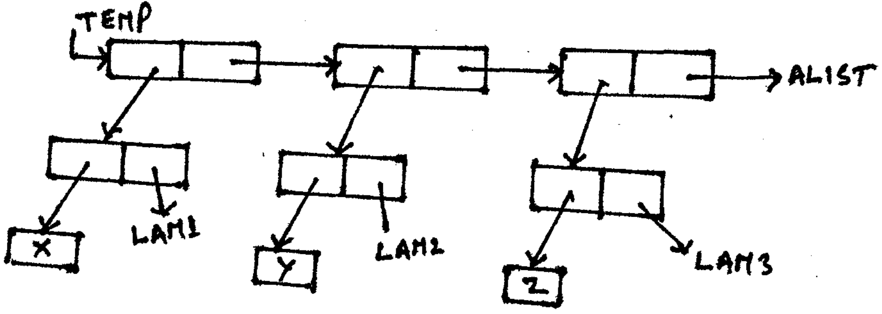
To understand more about the function search, consider the S-expression :

```
((DEFINE      (X  LAM1)
              (Y  LAM2)
              (Z  LAM3))
         (Z, list of quoted arguments)).
```

Where X, Y, and Z are identifiers, that are defined interms of LAM1s. LAMi is any Lambda expression. The list structure (in the computer memory) of the above expression appears to be as :



Now, when the function EVAL scans the LISP function 'DEFINE' in the S-expression with correct syntax, then control transfers to the function SEARCH. It attaches the 'ALIST' to the definitions sublist of the S-expression as explained above. The resultant structure of the ensironment (ALIST) would appear as :



Where "TEMP" is the address of the present "ALIST". This resultant list is used as the current environment for the function EVAL during the further evaluation of the LISP expression. The function LOCMARK is called when the CAR of CAR of 'PTR' is an atom and is not a reserved word (i.e. a LISP function). This function searches for a label mark whose name is equal to the CAR of CAR of 'PTR'. If it is found then the CAR of CDR of CAR of its grand-father node will be evaluated. And the repetitive evaluation of the statements lying between the label mark and the statement (GO ((label))), until the prespecified condition is satisfied.

The function SETARG is called to perform the operations for its corresponding LISP function SETQ. Function SETARG bounds the program variable with its corresponding value and conses this expression with the current ALIST. This value is considered as the latest one, and all the previous values which were bounded by the same variable are no more looked up. And the operations of the logical LISP functions, AND and OR are performed by calling the local function EVANDOR.

3.8 ORGANIZATION OF THE INTERPRETER :

So far, we have studied the actions of different individual functions and procedures in the interpreter program. Now let us discuss how these procedures and functions together can perform the task of interpretation of LISP-expressions.

The two-pass interpreter program scans the input symbolic expression twice during its process. In the pass-I it accepts the symbolic expression as it's input, and stores it in the computer memory in a binary tree form. And it's syntax & semantic analysis, and evaluation are all done during the pass-II.

Initially, the program calls the procedure INITIALISE, which assigns the boolean variable 'ALREADYPACKED' to 'FALSE', reads a character from the input file, constructs a free-list, assigns the reserved words to their corresponding LISP functions, and initialises the ALIST (association list) as described in Section 3.2 & 3.6 . Then, control transfers to the procedure 'NEXTSYMBOL', which decides the type of the input character which was just read at some point and reads in the next character from the input file. Further, the procedure 'READEXPR' is called to store the LISP expression in computer-memory (refer sec. 3.4.1).

Then, the interpreter program enters in a Loop, whose main function is to evaluate the LISP expression by transfering control over to the function 'EVAL', which inturn recursively executes the LISP instructions one by one and prints out the resultant list through the procedure PRINTEXPR. Since the execution of the present LISP-expression is over then the procedure 'GARBAGEMAN' is called, which collects all the used nodes, except the intial structure of 'ALIST', and attaches them to the free-list. If there are any more LISP-expressions, to be executed, in the input file, then it repeats the same process until there are no more LISP programs in the input file or a 'FIN' Card is encountered.

CHAPTER - IV

(DISCUSSION AND CONCLUSIONS)

4.1 DISCUSSION AND CONCLUSIONS :

"We must recognize the strong and undeniable influence that our language exerts on our way of thinking, and in fact defines and delimits the abstract space in which we can formulate - give form to - our thoughts" (Wirth 1974).

"Language is the vehcle by which we express our thoughts, and the relation between those thoughts and our language is a subtle and involuted one. The nature of language actually shapes and models the way we think . . . If, by providing appropriate language constructs we can improve the programs written using these structures, the entire field will benefit . . . A language design should atleast provide facilities which allow comprehensible expression of algorithms; at best a language suggests better forms of expression. But language is not a panacea. A language cannot, for example, prevent the creation of obscure programs; the ingenious programmer can always find an infinite number of paths to obfuscation." (Wolf 1977).

The relationship between software design methodologies and programming language is a most important one. This is so whether or not one views the programming language as a component of a software development facility. In trying to follow a certain design methodology, we will find that some languages are better suited than others. These are three important requirements in designing a language which are imposed by the software development process :

i) <u>Software must be reliable</u> :

i.e., users should be able to rely on the software. They should feel comfortable in using it even in the presence of infrequent or undesirable events such as hardware or software failure. Software is correct if it behaves according to its specifications, the more regorously and unambiguously the specifications are set down, the more convincingly program correctness can be proved. The reliability requirement has gained importance as software has been called upon to accomplish increasingly complicated tasks.

ii) <u>Software must be maintainable</u> :

Again, as software costs have risen and increasingly complex software systems have been developed, economic considerations have reduced to possibility of throwing away existing software and developing similar applications from scratch. So, existing software must be modified to meet new requirements.

iii) <u>Software must execute efficiently</u> :

Efficiency has always been a goal of any software system. This goal affects both the programming language and the choice of algorithms to be used.

These three requirements - reliability, maintainability, and efficiency - can be achieved by appropriate tools in the software development facility, and by certain characteristcs of the programming language.

The goal of software reliability is promoted by the following programming language qualities.

Writability, refers to the possibility of expressing a program in a way that is natural for the problem. The programmer should not be distracted by details and tricks of the language from the more important activity of problem solving. The easier it is to concentrate on problem solving activity, the less error prone is program writing.

It should be possible to follow the logic of the program, and to discover the presence of errors, by examining the program. The simpler the language is and the more naturally it allows algorithms to be expressed, then it is to understand what a program does by examining the code. For example, the GO TO statement has the potential of making programs hard to read, because it can make it impossible to read a program in one top-to-bottom pass and to understand it. Rather, one must jump around in the program in search of the targets of the GO TO statements.

The language should make it possible to trap undesired events (arithmatic overflows, invalid input, etc.) and to specify suitable responses to such events. In this way, the behaviour of the system becomes totally predicatable even in anamalous situations.

The need for maintainable programs imposes two requirements on the programming lange : Programs written in the language must be readable, and they must be modifiable. It is possible to identify features that make a program more modifiable. For example, several programming languages allow constants to be given symbolic names. Choosing an appropriate name for a constant promotes the readability of the program. Moreover, a future need to change the value would necessitate a change only in the definition of the constant, rather than in every use of constant.

Efficiency is no longer measured only by the execution speed and space. The effort required to produce a program, or system, initially and the effort required in maintenance can also be viewed as components of the efficiency measure. And, once again, the programming language can have a great impact.

A Language supports efficiency if it has qualities of writability and maintainability, and optimizability (i.e., the quality of allowing automatic program optimization).

Older languages, such as FORTRAN, were not designed to support specific design methodologies. For example, the absence of suitable high-level control structures in FORTRAN makes it difficult to systematically design algorithms in a top-down fashion. Conversely, PASCAL was designed with the explicit goal of supporting top-down design and structured programming. The developing trends in languages show that the idea that languages should support a design methodology is increasingly becoming accepted.

Now, coming to our "interpreter program" the inherent feature of PASCAL language enables us to design the problem in a top-down fashion. The recursive power of the language facilitates to define the tasks in a compact and flexible manner. The program starts from defining the global variables, and then deviding the task into separate modules namely the garbage collector, input, output routines, the evaluation procedure, and the initialization routine.

The data structures of PASCAL enables us to define the task in a natural way, and hence the reliability and maintainability. PASCAL provides us powerful "data types" to handle dynamic variables that are frequently encountered in the LISP processing system.

As the modules are well classified, if suppose one wants to introduce some more facilities to the interpreter system, then, one only needs to add their own subprograms to extend the power of the system. For example, as we did not much care to have comment statements in the LISP programs, we did not introduce this facility in our input routine. If one wants to have this facility, he can simply update the input routine in such a way that the system allows to have comments statements in LISP programs. And, the global constant "MAXNODES" whose value can be changed at once to increase (change) the number of nodes in the FREELIST. At present the number of nodes in the FREELIST is fixed at the system initialisation time. If one wants to have the dynamic expansion facility he can have this by simply writing the lines 728-735 in Appendix-A in a separate routine which can be called by the main program as many times according to the requirements of the LISP Program.

Writing software packages in a low level language is quite time consuming, and more over these packages are restricted either to one particular machine, or those family of machines. Writing a software package in a structured language yields, good readability, ease in implementation, good portability and also maintainability. Thus, the importance of building and using portable software continues to grow steadily, especially with the spreading of microprocessors.

There are clear advantages in using PASCAL, constructs for implementing LISP. We have the machine independency from the choice of higher level language and therefore portability. We had several other advantages that are inherent to PASCAL and were discussed in the preceeding sections.

If we try to draw pre-visions from the current situation, we think that portable software will use more and more high-level programming languages. Powerful microprocessors and portable low-level languages, although very successful, will probably disappear in the coming years, because writing large unstructured programs will no longer be tolerable. For the same reason, FORTRAN will no longer the only writing tool, and will be replaced by PASCAL, sometimes by languages like Ada, BCPL or C, or possible successors to these.

We have carried out this work on CYBER-170 system at National Informatics Centre, New Delhi. Our interpreter (PASCAL) program occupies 11-K words of memory. Where each word is of 60-bits size. We could not compare its execution time efficiency as there was no other LISP processing system available to us. However, we are getting quick responses with very small fraction of CPU secs, in executing even complex LISP programs.

Though in principle one can define higher order functions using the primitive LISP functions CAR, CDR, CONS, LAMBDA, COND, ATOM, DQ, and LABEL, it is not feasible on account of memory and execution time inefficiencies. Having only these functions, for example, COX, and TAYLOR's [11] system is not practically suitable for problem solving purposes. Besides these functions we have added DEFINE, PROG, SETQ, SUBST, LIST, GO, RETURN, EQUAL, NULL, and the logical connectives AND, OR and NOT functions. Having all these features in our improved system now we are in a position to use it for any symbol manipulation purposes.

APPENDIX - A

(INTERPRETER PROGRAM)

```
PROGRAM LISP(INPUT,OUTPUT);
LABEL
    1,(*USED TO RECOVER AFTER AN ERROR BY THE USER*)
    2;      (*IN CASE THE END OF THE FILE IS REACHED BEFORE A FIN CARD*)
  CONST
      MAXNODES=1200;
  TYPE
    INPUTSYMBOL=(ATOM,PERIOD,LPAREN,RPAREN);
    RESERVEDWORDS=(REPLACEHSYM,REPLACETSYM,HEADSYM,TAILSYM,EQSYM,QUOTESYM,
                   ATOMSYM,CONDSYM,LABELSYM,LAMBDASYM,COPYSYM,APPENDSYM,
                   CONCSYM,DEFINESYM,SETQSYM,NULLSYM,NOTSYM,ORSYM,ANDSYM,
                   EQUALSYM,LISTSYM,SUBSTSYM,PROGSYM,GOSYM,RETURNSYM,
                   CONSSYM);
   STATUSTYPE=(UNMARKED,LEFT,RIGHT,MARKED);
    SYMBEXPPTR=^SYMBOLICEXPRESSION;
   ALPHA=PACKED ARRAY [1..10] OF CHAR;
   SYMBOLICEXPRESSION=PACKED RECORD
                       STATUS:STATUSTYPE;
                       NEXT:SYMBEXPPTR;
                       CASE ANATOM:BOOLEAN OF
                         TRUE:(NAME:ALPHA;
                             CASE ISARESERVEDWORD:BOOLFAN OF
                                 TRUE:(RESSYM:RESERVEDWORDS));
                         FALSE:(HEAD,TAIL:SYMBEXPPTR);
                      END;
        (*THE GLOBAL  VARIABLES*)
  VAR
    LOOKAHEADSYM,
    SYM:INPUTSYMBOL;
    ID:ALPHA;
```

```
   ALREADYPACKED:BOOLEAN;
   CH:CHAR;
   PTR:SYMBEXPPTR;
   FREELIST,
   NODELIST,
   ALIST:SYMBEXPPTR;
   NILNODE,
   TNODE:SYMBOLICEXPRESSION;
   RESWORD:RESERVEDWORDS;
   RESERVED:BOOLEAN;
   RESWORDS:ARRAY [RESERVEDWORDS] OF ALPHA;
   FREENODES:INTEGER;
   NUMBEROFGCS:INTEGER;
PROCEDURE GARBAGEMAN;
PROCEDURE MARK(LIST:SYMBEXPPTR);
   VAR
     FATHER,
     SON,
     CURRENT:SYMBEXPPTR;
   BEGIN
      FATHER:=NIL;
      CURRENT:=LIST;
      SON:=CURRENT;
      WHILE CURRENT<>NIL DO
      WITH CURRENT^ DO
       CASE STATUS OF
        UNMARKED:IF ANATOM THEN STATUS:=MARKED
                 ELSE IF (HEAD^.STATUS<>UNMARKED) OR (HEAD=CURRENT)
                      THEN IF (TAIL^.STATUS<>UNMARKED) OR (TAIL=CURRENT)
                            THEN STATUS:=MARKED
```

```
                     ELSE  BEGIN
                         STATUS:=RIGHT;
                         SON:=TAIL;
                         TAIL:=FATHER;
                         FATHER:=CURRENT;
                         CURRENT:=SON
                           END
               ELSE
                 BEGIN
                  STATUS:=LEFT;
                  SON:=HEAD;
                  HEAD:=FATHER;
                  FATHER:=CURRENT;
                  CURRENT:=SON
                END;
LEFT: IF (TAIL^.STATUS<>UNMARKED) THEN
        BEGIN
          STATUS:=MARKED;
          FATHER:=HEAD;
          HEAD:=SON;
          SON:=CURRENT
       END
    ELSE
       BEGIN
         STATUS:=RIGHT;
         CURRENT:=TAIL;
         TAIL:=HEAD;
         HEAD:=SON;
         SON:=CURRENT
     END;
```

```
     RIGHT:   BEGIN
                 STATUS:=MARKED;
                 FATHER:=TAIL;
                 TAIL:=SON;
                 SON:=CURRENT
              END;
     MARKED:   CURRENT:=FATHER
   END (*CASE*)
END (*MARK*);
PROCEDURE COLLECTFREENODES;
     VAR
       TEMP:SYMBEXPPTR;
     BEGIN
       WRITELN('NUMBER OF FREE NODES BEFORE COLLECTION=',FREENODES:3,'.');
       FREELIST:=NIL;
       FREENODES:=0;
       TEMP:=NODELIST;
       WHILE TEMP<>NIL DO
        BEGIN
         IF(TEMP^.STATUS<>UNMARKED) THEN TEMP^.STATUS:=UNMARKED
           ELSE BEGIN
             FREENODES:=FREENODES+1;
             TEMP^.HEAD:=FREELIST;
             FREELIST:=TEMP
                END;
        TEMP:=TEMP^.NEXT
       END;
       WRITELN('NUMBER OF FREE NODES AFTER COLLECTION=',FREENODES:3,'.');
 END;        (*COLLECT FREENODES*)
    BEGIN   (*GARBAGEMAN*)
```

```
       NUMBEROFGCS:=NUMBEROFGCS+1;
       WRITELN;
       WRITELN('GARBAGECOLLECTION.');
       WRITELN;
     MARK(ALIST);
     IF PTR<>NIL THEN MARK(PTR);
       COLLECTFREENODES
     END (*GARBAGEMAN*);
PROCEDURE POP(VAR SPTR:SYMBEXPPTR);
    BEGIN
      IF FREELIST=NIL THEN
      BEGIN
        WRITELN('NOT ENOUGH SPACE TO EVALUATE THE EXPRESSION');
      GOTO 2
        END;
      FREENODES:=FREENODES-1;
      SPTR:=FREELIST;
      FREELIST:=FREELIST^.HEAD
    END(*POP*);
PROCEDURE ERROR(NUMBER:INTEGER);
    BEGIN
      WRITELN;
      WRITE('ERROR',NUMBER:3,'.');
      CASE NUMBER OF
        1:WRITELN('ATOM OR LPAREN EXPECTED IN THE S-EXPR.');
        2:WRITELN('ATOM,LPAREN,  OR RPAREN EXPECTED IN THE S-EXPR.');
        3:WRITELN('LABEL,LAMBDA,DEFINE AND SETQ ARE NOT NAMES OF FUNCTIN
        4:WRITELN('RPAREN EXPECTED IN THE S-EXPRESSION');
        5:WRITELN('1ST ARGUMENT OF REPLACAR IS AN ATOM.');
        6:WRITELN('1ST ARGUMENT OF REPLACDR IS AN ATOM .');
```

```
       7:WRITELN('ARGUMENT OF  CAR IS AN ATOM');
       8:WRITELN('ARGUMENT OF CDR IS AN ATOM');
       9:WRITELN('1ST ARGUMENT OF APPEND IS NOT A LIST.');
      10:WRITELN('COMA OR RPAREN EXPECTED IN CONCATENATE.');
      11:WRITELN('END OF FILE ENCOUNTERED BEFORE A FINCARD');
      12:WRITELN('EITHER OF LAMBDA,LABEL,DEFINE,SETQ IS EXPECTED.');
      13:WRITELN('VALUE OF FUNCTION COND IS NOT DEFINED.');
     14:WRITELN('FUNCTION IS NOT DEFINED.');
   15:WRITELN('ERROR IN ARGUMENTS TYPE.')
   END;
   IF NUMBER IN [11] THEN GOTO 2
      ELSE GOTO 1
 END (*ERROR*);
PROCEDURE BACKUPINPUT;
     BEGIN
        ALREADYPACKED:=TRUE;
        LOOKAHEADSYM:=SYM;
        SYM:=LPAREN
    END (*BACKUPINPUT*);
PROCEDURE NEXTSYM;
  VAR
    I:INTEGER;
  BEGIN
    IF ALREADYPACKED
    THEN BEGIN
           SYM:=LOOKAHEADSYM;
           ALREADYPACKED:=FALSE
         END
    ELSE
         BEGIN
```

```
        WHILE CH=' ' DO
        BEGIN
           IF EOLN(INPUT) THEN WRITELN;
           READ(CH);
           WRITE(CH)
        END;
 IF CH IN ['(','.',')']
    THEN BEGIN
             CASE CH OF
               '(':SYM:=LPAREN;
               '.':SYM:=PERIOD;
               ')':SYM:=RPAREN
             END(*CASE*);
 IF EOLN(INPUT) THEN WRITELN;
                 READ(CH);
                 WRITE(CH)
      END
     ELSE BEGIN
        SYM:=ATOM;
        ID:='          ';
        I:=0;
REPEAT
    I:=I+1;
  IF I<11 THEN ID[I]:=CH;
    IF EOLN(INPUT)  THEN WRITELN;
                         READ(CH);
                         WRITE(CH)
UNTIL CH IN [' ','(','.',')'];
  RESWORD:=REPLACEHSYM;
  WHILE (ID<>RESWORDS[RESWORD]) AND (RESWORD<>CONSSYM) DO
```

```
    RESWORD:=SUCC(RESWORD);
          RESERVED:=(ID=RESWORDS[RESWORD])
    END;
END
END (*NEXTSYM*);
PROCEDURE READEXPR(VAR SPTR:SYMBEXPPTR);
      VAR
        NXT,HEAD1,TAIL1:SYMBEXPPTR;
      BEGIN
         POP(SPTR);
         NXT:=SPTR^.NEXT;
         CASE SYM OF
           RPAREN,PERIOD:ERROR(1);
           ATOM:WITH SPTR^ DO
                BEGIN
                    ANATOM:=TRUE;
                    NAME:=ID;
                    ISARESERVEDWORD:=RESERVED;
                    IF RESERVED THEN RESSYM:=RESWORD
                END;
          LPAREN:WITH SPTR^ DO

                 BEGIN
                    NEXTSYM;
                    IF SYM=PERIOD THEN ERROR(2)
                       ELSE IF SYM=RPAREN THEN SPTR^:=NILNODE
             ELSE   BEGIN
                        ANATOM:=FALSE;
                        READEXPR(HEAD1);
                      HEAD:=HEAD1;
```

```
                        NEXTSYM;
              IF SYM=PERIOD
                THEN
                   BEGIN
                     NEXTSYM;
                     READEXPR(TAIL1);
                   TAIL:=TAIL1;
                     NEXTSYM;
              IF SYM<>RPAREN THEN ERROR(4);
                  END
              ELSE
               BEGIN
                   BACKUPINPUT;
                   READEXPR(TAIL1);
                 TAIL:=TAIL1
               END
            END
          END (*WITH*)
         END (*CASE*);
             SPTR^.NEXT:=NXT
     END (*READEXPR*);
PROCEDURE PRINTNAME(NAME:ALPHA);
      VAR
        I:INTEGER;
      BEGIN
          I:=1;
          REPEAT WRITE(NAME[I]);
                   I:=I+1
          UNTIL  (NAME[I]=' ') OR (I=11)
       END (*PRINTNAME*);
```

```
PROCEDURE PRINTEXPR(SPTR:SYMBEXPPTR);
      LABEL
          1;
      BEGIN
         IF SPTR^.ANATOM THEN PRINTNAME(SPTR^.NAME)
           ELSE BEGIN
             WRITE('(');
              1:WITH SPTR^ DO
             BEGIN
                PRINTEXPR(HEAD);
                IF TAIL^.ANATOM AND (TAIL^.NAME='NIL       ')
                   THEN WRITE(')')
                   ELSE
                       IF TAIL^.ANATOM THEN
                       BEGIN
                           WRITE('.');
                           PRINTEXPR(TAIL);
                           WRITE(')')
                       END
                   ELSE BEGIN
                       WRITE(' ');
                           SPTR:=TAIL;
                            GOTO 1
                     END
              END
           END
 END (*PRINTEXPR*);
FUNCTION EVAL(E,ALIST:SYMBEXPPTR):SYMBEXPPTR;
   VAR
     TEMP,NILT,STAD,TEST,CAROFE,CAAROFE:SYMBEXPPTR;
```

```
   CHECKQUO:BOOLEAN;
FUNCTION REPLACEH(SPTR1,SPTR2:SYMBEXPPTR):SYMBEXPPTR;
    BEGIN
        IF SPTR1^.ANATOM THEN ERROR(5)
      ELSE SPTR1^.HEAD:=SPTR2;
      REPLACEH:=SPTR1;
   END (*RPLACEH*);
FUNCTION REPLACET(SPTR1,SPTR2:SYMBEXPPTR):SYMBEXPPTR;
    BEGIN
      IF SPTR1^.ANATOM THEN ERROR(6)
        ELSE SPTR1^.TAIL:=SPTR2;
        REPLACET:=SPTR1
   END (*REPLACET*);
FUNCTION HEAD(SPTR:SYMBEXPPTR):SYMBEXPPTR;
    BEGIN
      IF SPTR^.ANATOM
    THEN BEGIN
           PRINTEXPR(SPTR);
           ERROR(7)
         END
    ELSE HEAD:=SPTR^.HEAD
    END (*HEAD*);
FUNCTION TAIL(SPTR:SYMBEXPPTR):SYMBEXPPTR;
    BEGIN
       IF SPTR^.ANATOM
     THEN BEGIN
            PRINTEXPR(SPTR);
            ERROR(8)
         END
     ELSE TAIL:=SPTR^.TAIL
```

```
    END (*TAIL*);
FUNCTION CONS(SPTR1,SPTR2:SYMBEXPPTR):SYMBEXPPTR;
    VAR
      TEMP:SYMBEXPPTR;
    BEGIN
      POP(TEMP);
      TEMP^.ANATOM:=FALSE;
      TEMP^.HEAD:=SPTR1;
      TEMP^.TAIL:=SPTR2;
      CONS:=TEMP
    END (*CONS*);
FUNCTION COPY(SPTR:SYMBEXPPTR):SYMBEXPPTR;
    VAR
      TEMP,NXT:SYMBEXPPTR;
    BEGIN
       IF SPTR^.ANATOM
       THEN BEGIN
              POP(TEMP);
              NXT:=TEMP^.NEXT;
              TEMP^:=SPTR^;
              TEMP^.NEXT:=NXT;
              COPY:=TEMP
            END
       ELSE  COPY:=CONS(COPY(SPTR^.HEAD),COPY(SPTR^.TAIL))
    END (*COPY*);
FUNCTION APPEND(SPTR1,SPTR2:SYMBEXPPTR):SYMBEXPPTR;
    BEGIN
       IF SPTR1^.ANATOM THEN IF SPTR1^.NAME<>'NIL       ' THEN ERROR(9)
          ELSE APPEND:=SPTR2 ELSE APPEND:=CONS(COPY(SPTR1^.HEAD),APPEND(
               SPTR1^.TAIL,SPTR2))
```

```
    END (*APPEND*);
FUNCTION CONC(SPTR1:SYMBEXPPTR):SYMBEXPPTR;
    VAR
      SPTR2,NILPTR:SYMBEXPPTR;
    BEGIN
       IF SYM<>RPAREN THEN
         BEGIN
            NEXTSYM;
            READEXPR(SPTR2);
            NEXTSYM;
            CONC:=CONS(SPTR1,CONC(SPTR2));
         END
      ELSE
          IF SYM=RPAREN THEN
         BEGIN
           NEW(NILPTR);
           WITH NILPTR^ DO
               BEGIN
                  ANATOM:=TRUE;
               NAME:='NIL       '
              END;
               CONC:=CONS(SPTR1,NILPTR);
         END
          ELSE ERROR(10)
   END (*CONC*);
FUNCTION EQQ(SPTR1,SPTR2:SYMBEXPPTR):SYMBEXPPTR;
    VAR
      TEMP,NXT:SYMBEXPPTR;
    BEGIN
      POP(TEMP);
```

```
      NXT:=TEMP^.NEXT;
      IF SPTR1^.ANATOM AND SPTR2^.ANATOM THEN
         IF SPTR1^.NAME=SPTR2^.NAME THEN TEMP^:=TNODE
         ELSE TEMP^:=NILNODE
      ELSE IF SPTR1=SPTR2 THEN TEMP^:=TNODE
           ELSE TEMP^:=NILNODE;
      TEMP^.NEXT:=NXT;
      EQQ:=TEMP
   END (*EQQ*);
 FUNCTION EQUAL(SPTR1,SPTR2:SYMBEXPPTR):SYMBEXPPTR;
    VAR
     TEMP1,NXT:SYMBEXPPTR;
  PROCEDURE EQUATE(SPTR1,SPTR2:SYMBEXPPTR);
    BEGIN
   IF SPTR1^.ANATOM AND SPTR2^.ANATOM
 THEN   IF SPTR1^.NAME=SPTR2^.NAME
        THEN TEMP1^:=TNODE ELSE TEMP1^:=NILNODE
 ELSE IF (NOT(SPTR1^.ANATOM) AND SPTR2^.ANATOM) OR (SPTR1^.ANATOM
        AND (NOT(SPTR2^.ANATOM))) THEN TEMP1^:=NILNODE
ELSE BEGIN
     EQUATE(HEAD(SPTR1),HEAD(SPTR2));
      IF TEMP1^.NAME='T         '
      THEN EQUATE(TAIL(SPTR1),TAIL(SPTR2));
      END
 END (*EQUATE*);
  BEGIN (*EQUAL*)
    POP(TEMP1);
   NXT:=TEMP1^.NEXT;
    EQUATE(SPTR1,SPTR2);
   EQUAL:=TEMP1;
```

```
  TEMP1^.NEXT:=NXT
 END (*EQUAL*);
FUNCTION LIST(SPTR1,SPTR2:SYMBEXPPTR):SYMBEXPPTR;
  VAR
    NUL,NXT:SYMBEXPPTR;
  BEGIN
    POP(NUL);
    NXT:=NUL^.NEXT;
    NUL^:=NILNODE;
    NUL^.NEXT:=NXT;
    LIST:=CONS(SPTR1,CONS(SPTR2,NUL))
 END (*LIST*);
FUNCTION SUBST(SPTR1,SPTR2,SPTR3:SYMBEXPPTR):SYMBEXPPTR;
   VAR
     TEMP1:SYMBEXPPTR;
   BEGIN
     TEMP1:=EQUAL(SPTR2,SPTR3);
     IF TEMP1^.NAME='T         ' THEN SUBST:=SPTR1
     ELSE IF SPTR3^.ANATOM THEN SUBST:=SPTR3
   ELSE  SUBST:=CONS(SUBST(SPTR1,SPTR2,HEAD(SPTR3)),SUBST(SPTR1,SPTR2,TAIL
            (SPTR3)))
  END (*SUBST*);
 FUNCTION NULL(SPTR:SYMBEXPPTR):SYMBEXPPTR;
    VAR
     TEMP4,NXT:SYMBEXPPTR;
   BEGIN
     POP(TEMP4);
     NXT:=TEMP4^.NEXT;
     IF (SPTR^.NAME='NIL       ') THEN TEMP4^:=TNODE
     ELSE TEMP4^:=NILNODE;
```

```
      TEMP4^.NEXT:=NXT;
      NULL:=TEMP4
   END (*NULL*);
FUNCTION ATOM(SPTR:SYMBEXPPTR):SYMBEXPPTR;
    VAR
      TEMP,NXT:SYMBEXPPTR;
    BEGIN
      POP(TEMP);
      NXT:=TEMP^.NEXT;
      IF SPTR^.ANATOM THEN TEMP^:=TNODE
         ELSE TEMP^:=NILNODE;
      TEMP^.NEXT:=NXT;
      ATOM:=TEMP
   END (*ATOM*);
FUNCTION LOOKUP(KEY,ALIST:SYMBEXPPTR):SYMBEXPPTR;
    VAR
      TEMP,FUNC:SYMBEXPPTR;
    BEGIN
      TEMP:=EQQ(HEAD(HEAD(ALIST)),KEY);
      IF TEMP^.NAME='T         '
     THEN  LOOKUP:=TAIL(HEAD(ALIST))
         ELSE BEGIN
             FUNC:=TAIL(ALIST);
             IF FUNC^.NAME='NIL       '
           THEN BEGIN
                  PRINTEXPR(KEY);
                  ERROR(14)
                END
             ELSE LOOKUP:=LOOKUP(KEY,TAIL(ALIST))
           END
```

```
    END (*LOOKUP*);
FUNCTION BINDARGS(NAMES,VALUES:SYMBEXPPTR):SYMBEXPPTR;
    VAR
      TEMP,TEMP2:SYMBEXPPTR;
    BEGIN
      IF NAMES^.ANATOM AND (NAMES^.NAME='NIL       ')
         THEN BINDARGS:=ALIST
         ELSE BEGIN
               TEMP:=CONS(HEAD(NAMES),EVAL(HEAD(VALUES),ALIST));
               TEMP2:=BINDARGS(TAIL(NAMES),TAIL(VALUES));
               BINDARGS:=CONS(TEMP,TEMP2)
              END
   END (*BINDARGS*);
FUNCTION EVCON(CONDPAIRS:SYMBEXPPTR):SYMBEXPPTR;
    VAR
      TEMP,TEST:SYMBEXPPTR;
    BEGIN
     TEMP:=EVAL(HEAD(HEAD(CONDPAIRS)),ALIST);
     IF TEMP^.ANATOM AND (TEMP^.NAME='NIL       ')
     THEN BEGIN
          TEST:=TAIL(CONDPAIRS);
          IF TEST^.ANATOM AND (TEST^.NAME='NIL       ')
          THEN BEGIN
                 PRINTEXPR(CONDPAIRS);
                 ERROR(13)
               END
          ELSE EVCON:=EVCON(TAIL(CONDPAIRS))
     END
     ELSE EVCON:=EVAL(HEAD(TAIL(HEAD(CONDPAIRS))),ALIST)
    END (*EVCON*);
```

```
 FUNCTION NAT(SPTR1:SYMBEXPPTR):SYMBEXPPTR;
    VAR
      TEMP1,NXT:SYMBEXPPTR;
    BEGIN
      POP(TEMP1);
      NXT:=TEMP1^.NEXT;
      IF SPTR1^.NAME='T         '  THEN TEMP1^:=NILNODE
      ELSE IF SPTR1^.NAME='NIL       ' THEN TEMP1^:=TNODE
      ELSE BEGIN
           PRINTEXPR(SPTR1);
           ERROR(15)
         END;
      TEMP1^.NEXT:=NXT;
      NAT:=TEMP1
   END (*NAT FUNCTION*);
FUNCTION EVANDOR(PRED,SPTR1:SYMBEXPPTR):SYMBEXPPTR;
   VAR
    TEMP1,TEMP2,TEMP3:SYMBEXPPTR;
  BEGIN
    TEMP3:=EVAL(HEAD(PRED),ALIST);
    TEMP2:=TAIL(PRED);
    IF TEMP2^.NAME<>'NIL       ' THEN
    IF TEMP3^.NAME=NILT^.NAME THEN EVANDOR:=EVANDOR(TAIL(PRED),SPTR1)
    ELSE BEGIN
          TEMP1:=NAT(NILT);
          IF TEMP3^.NAME=TEMP1^.NAME THEN EVANDOR:=TEMP3
          ELSE BEGIN
              PRINTEXPR(PRED);
              ERROR(15)
             END
```

```
       END
  ELSE EVANDOR:=TEMP3
END (*EVANDOR*);
  FUNCTION SETARG(NAM,VAL:SYMBEXPPTR):SYMBEXPPTR;
     VAR
       TEMP1:SYMBEXPPTR;
        BEGIN
               TEMP1:=CONS(HEAD(NAM),FVAL(HEAD(VAL),ALIST));
               SETARG:=CONS(TEMP1,ALIST)
     END   (*SETARG*);
 FUNCTION SEARCH(FPTR:SYMBEXPPTR):SYMBEXPPTR;
    VAR
     NXT:SYMBEXPPTR;
    BEGIN
      NXT:=FPTR;
      WHILE FPTR^.TAIL^.NAME<>'NIL       ' DO
        FPTR:=FPTR^.TAIL;
     FPTR^.TAIL:=ALIST;
     SEARCH:=NXT
   END (*SEARCH*);
FUNCTION BINDVARS(SPTR,VARS:SYMBEXPPTR):SYMBEXPPTR;
   VAR
     NUL,NXT,TEMPO,TEMP1,TEMP2:SYMBEXPPTR;
FUNCTION INITVAL(VARS:SYMBEXPPTR):SYMBEXPPTR;
   BEGIN
     IF VARS^.ANATOM AND (VARS^.NAME='NIL       ')
     THEN INITVAL:=TEMPO
     ELSE BEGIN
            TEMP1:=CONS(HEAD(VARS),NUL);
            TEMP2:=INITVAL(TAIL(VARS));
```

```
            INITVAL:=CONS(TEMP1,TEMP2)
          END
   END; (*INITVAL*)
   BEGIN  (*BINDVARS*)
     POP(NUL);
     NXT:=NUL^.NEXT;
     NUL^:=NILNODE;
     NUL^.NEXT:=NXT;
     TEMP0:=SEARCH(SPTR);
     BINDVARS:=INITVAL(VARS)
   END;  (*BINDVARS*)
FUNCTION LOCMARK(KEY,ALIST:SYMBEXPPTR):SYMBEXPPTR;
   VAR
     TEMP,FUNC:SYMBEXPPTR;
   BEGIN
    FUNC:=HEAD(HEAD(ALIST));
    IF FUNC^.ANATOM
   THEN IF FUNC^.NAME='COND      '
    THEN BEGIN
         FUNC:=TAIL(HEAD(ALIST));
          WHILE NOT (FUNC^.TAIL^.NAME='NIL       ') DO
          FUNC:=FUNC^.TAIL;
          LOCMARK:=LOCMARK(KEY,TAIL(HEAD(FUNC)))
        END
    ELSE BEGIN
        FUNC:=TAIL(ALIST);
        IF FUNC^.ANATOM AND (FUNC^.NAME='NIL       ')
        THEN LOCMARK:=LOCMARK(KEY,TAIL(HEAD(ALIST)))
         ELSE LOCMARK:=LOCMARK(KEY,TAIL(ALIST))
     END
```

```
ELSE BEGIN
      TEMP:=EQQ(HEAD(HEAD(HEAD(ALIST))),KEY);
    IF TEMP^.NAME='T         ' THEN LOCMARK:=HEAD(TAIL(HEAD(ALIST)))
          ELSE LOCMARK:=LOCMARK(KEY,TAIL(HEAD(ALIST)))
         END
   END;    (*LOCATE MARKS*)
 BEGIN   (*EVAL*)
    IF E^.ANATOM THEN EVAL:=LOOKUP(E,ALIST)
       ELSE BEGIN
              CAROFE:=HEAD(E);
              IF CAROFE^.ANATOM
              THEN IF NOT CAROFE^.ISARESERVEDWORD
        THEN EVAL:=EVAL(CONS(LOOKUP(CAROFE,ALIST),TAIL(E)),ALIST)
                    ELSE CASE
                           CAROFE^.RESSYM OF
                           SETQSYM,DEFINESYM,LABELSYM,LAMBDASYM:ERROR(3);
                           QUOTESYM:EVAL:=HEAD(TAIL(E));
    NULLSYM: EVAL:=NULL(EVAL(HEAD(TAIL(E)),ALIST));
  EQUALSYM: EVAL:=EQUAL(EVAL(HEAD(TAIL(E)),ALIST),EVAL(HEAD(TAIL(TAIL(E)
             )),ALIST));
 ATOMSYM:    EVAL:=ATOM(EVAL(HEAD(TAIL(E)),ALIST));
     EQSYM:   EVAL:=EQQ(EVAL(HEAD(TAIL(E)),ALIST),EVAL(HEAD(TAIL(TAIL(E)
                     )),ALIST));
      NOTSYM: EVAL:=NAT(EVAL(HEAD(TAIL(E)),ALIST));
      ORSYM:  BEGIN
               POP(NILT);
               STAD:=NILT^.NEXT;
               NILT^:=NILNODE;
               NILT^.NEXT:=STAD;
               EVAL:=EVANDOR(TAIL(E),NILT)
```

```
           END;
    ANDSYM: BEGIN
              POP(NILT);
              STAD:=NILT^.NEXT;
              NILT^:=TNODE;
              NILT^.NEXT:=STAD;
              EVAL:=EVANDOR(TAIL(E),NILT)
           END;
     HEADSYM:  EVAL:=HEAD(EVAL(HEAD(TAIL(E)),ALIST));
     TAILSYM:  EVAL:=TAIL(EVAL(HEAD(TAIL(E)),ALIST));
  CONSSYM:  EVAL:=CONS(EVAL(HEAD(TAIL(E)),ALIST),EVAL(HEAD(TAIL(TAIL(E)))
                   ,ALIST));
        CONDSYM: EVAL:=EVCON(TAIL(E));
        CONCSYM: ;
        APPENDSYM:
             EVAL:=APPEND(EVAL(HEAD(TAIL(E)),ALIST),EVAL(HEAD(TAIL(TAIL
                    (E))),ALIST));
     LISTSYM: EVAL:=LIST(EVAL(HEAD(TAIL(E)),ALIST),EVAL(HEAD
               (TAIL(TAIL(E))),ALIST));
    SUBSTSYM: EVAL:=SUBST(EVAL(HEAD(TAIL(E)),ALIST),EVAL(HEAD(TAIL(TAIL
              (E))),ALIST),EVAL(HEAD(TAIL(TAIL(TAIL(E)))),ALIST));
        REPLACEHSYM:
                 EVAL:=REPLACEH(EVAL(HEAD(TAIL(E)),ALIST),EVAL(HEAD(
                       TAIL(TAIL(E))),ALIST));
REPLACETSYM: EVAL:=REPLACET(EVAL(HEAD(TAIL(E)),ALIST),EVAL(HEAD(TAIL(TAIL
            (E))),ALIST));
 PROGSYM: BEGIN
             TEMP:=BINDVARS(TAIL(TAIL(E)),HEAD(TAIL(E)));
             EVAL:=EVAL(HEAD(TAIL(TAIL(E))),TEMP)
          END;
```

```
GOSYM: EVAL:=EVAL(HEAD(TAIL(E)),ALIST);
RETURNSYM: EVAL:=EVAL(HEAD(TAIL(E)),ALIST);
       END (*CASE*)
          ELSE
            BEGIN
              CAAROFE:=HEAD(CAROFE);
              IF CAAROFE^.ANATOM
        THEN IF NOT.CAAROFE^.ISARESERVEDWORD
        THEN EVAL:=EVAL(LOCMARK(CAAROFE,ALIST),ALIST)
   ELSE IF NOT (CAAROFE^.RESSYM IN [SETQSYM,DEFINESYM,LABELSYM,LAMBDASYM])
                     THEN ERROR(12)
                ELSE  CASE CAAROFE^.RESSYM OF
   SETQSYM: BEGIN
              TEMP:=SETARG(TAIL(CAROFE),TAIL(TAIL(CAROFE)));
              EVAL:=EVAL(HEAD(TAIL(E)),TEMP)
            END;
     DEFINESYM:  BEGIN
                TEMP:=SEARCH(TAIL(CAROFE));
                EVAL:=EVAL(HEAD(TAIL(E)),TEMP)
              END;
                 LABELSYM:
                    BEGIN
                      TEMP:=CONS(CONS(HEAD(TAIL(CAROFE)),HEAD(TAIL(
                             TAIL(CAROFE)))),ALIST);
                     EVAL:=EVAL(CONS(HEAD(TAIL(TAIL(CAROFE))),
                             TAIL(E)),TEMP)
                    END;
     LAMBDASYM:
        BEGIN
           TEMP:=BINDARGS(HEAD(TAIL(CAROFE)),TAIL(E));
```

```
             EVAL:=EVAL(HEAD(TAIL(TAIL(CAROFE))),TEMP)
           END
                       END (*CASE*)
       ELSE
         EVAL:=EVAL(CONS(EVAL(CAROFE,ALIST),TAIL(E)),ALIST)
      END
    END
 END (*EVAL*);
PROCEDURE INITIALISE;
     VAR
       I:INTEGER;
        HEAD1,TAIL1,TEMP,NXT:SYMBEXPPTR;
     BEGIN
       ALREADYPACKED:=FALSE;
       READ(CH);
        WRITE(CH);
       NUMBEROFGCS:=0;
     FREENODES:=MAXNODES;
     WITH NILNODE DO
       BEGIN
          ANATOM:=TRUE;
          NEXT:=NIL;
          NAME:='NIL       ';
          STATUS:=UNMARKED;
          ISARESERVEDWORD:=FALSE
        END;
     WITH TNODE DO
        BEGIN
          ANATOM:=TRUE;
          NEXT:=NIL;
```

```
         NAME:='T         ';
          STATUS:=UNMARKED;
          ISARESERVEDWORD:=FALSE
       END;
    (*ALLOCATE STORAGE AND MARK IT FREE*)
     FREELIST:=NIL;
   (*$R-,W30000B*)
     FOR I:=1 TO MAXNODES DO
        BEGIN
          NEW(NODELIST);
          NODELIST^.NEXT:=FREELIST;
          NODELIST^.HEAD:=FREELIST;
          NODELIST^.STATUS:=UNMARKED;
          FREELIST:=NODELIST
       END;
  (*INITIALISE RESERVED WORD TABLE*)
   RESWORDS[REPLACEHSYM]:='REPLACAR  ';
  RESWORDS[REPLACETSYM]:='REPLACDR  ';
  RESWORDS[HEADSYM]:='CAR       ';
  RESWORDS[TAILSYM]:='CDR       ';
  RESWORDS[COPYSYM]:='COPY      ';
  RESWORDS[APPENDSYM]:='APPEND    ';
  RESWORDS[CONCSYM]:='CONC      ';
  RESWORDS[CONSSYM]:='CONS      ';
  RESWORDS[EQSYM]:='EQ        ';
  RESWORDS[QUOTESYM]:='QUOTE     ';
  RESWORDS[ATOMSYM]:='ATOM      ';
RESWORDS[NOTSYM]:='NOT       ';
RESWORDS[ORSYM]:='OR        ';
RESWORDS[ANDSYM]:='AND       ';
```

```
  RESWORDS[CONDSYM]:='COND      ';
  RESWORDS[LABELSYM]:='LABEL     ';
  RESWORDS[LAMBDASYM]:='LAMBDA    ';
 RESWORDS[SETQSYM]:='SETQ      ';
 RESWORDS[DEFINESYM]:='DEFINE    ';
RESWORDS[PROGSYM]:='PROG      ';
RESWORDS[GOSYM]:='GO        ';
RESWORDS[RETURNSYM]:='RETURN    ';
 RESWORDS[NULLSYM]:='NULL      ';
RESWORDS[EQUALSYM]:='EQUAL     ';
RESWORDS[LISTSYM]:='LIST      ';
RESWORDS[SUBSTSYM]:='SUBST     ';
    (*INITIALISE THE A-LIST WITH T AND NIL *)
  POP(ALIST);
  ALIST^.ANATOM:=FALSE;
  ALIST^.STATUS:=UNMARKED;
  POP(TAIL1);
 ALIST^.TAIL:=TAIL1;
  NXT:=ALIST^.TAIL^.NEXT;
  ALIST^.TAIL^:=NILNODE;
  ALIST^.TAIL^.NEXT:=NXT;
  POP(HEAD1);
 ALIST^.HEAD:=HEAD1;
   (*BIND NIL TO THE ATOM NIL *)
      WITH ALIST^.HEAD^ DO
  BEGIN
    ANATOM:=FALSE;
    STATUS:=UNMARKED;
    POP(HEAD1);
  HEAD:=HEAD1;
```

```
   NXT:=HEAD^.NEXT;
   HEAD^:=NILNODE;
   HEAD^.NEXT:=NXT;
   POP(TAIL1);
 TAIL:=TAIL1;
   NXT:=TAIL^.NEXT;
     TAIL^:=NILNODE;
   TAIL^.NEXT:=NXT
END;
   POP(TEMP);
   TEMP^.ANATOM:=FALSE;
   TEMP^.STATUS:=UNMARKED;
   TEMP^.TAIL:=ALIST;
   ALIST:=TEMP;
   POP(HEAD1);
 ALIST^.HEAD:=HEAD1;
      (*BIND TO THE ATOM T *)
     WITH ALIST^.HEAD^ DO
  BEGIN
    ANATOM:=FALSE;
    STATUS:=UNMARKED;
    POP(HEAD1);
  HEAD:=HEAD1;
    NXT:=HEAD^.NEXT;
    HEAD^:=TNODE;
    HEAD^.NEXT:=NXT;
    POP(TAIL1);
  TAIL:=TAIL1;
    NXT:=TAIL^.NEXT;
    TAIL^:=TNODE;
```

```
     TAIL^.NEXT:=NXT;
   END;
END  (*INITIALISE*);
BEGIN    (*LISP*)
   WRITELN('*EVAL*');
   INITIALISE;
     NEXTSYM;
   READEXPR(PTR);
   READLN;
   WRITELN;
   WHILE NOT PTR^.ANATOM OR (PTR^.NAME<>'FIN       ') DO
     BEGIN
        WRITELN;
       WRITELN('*VALUE*');
       PRINTEXPR(EVAL(PTR,ALIST));
  1: WRITELN;
     WRITELN;
      IF EOF(INPUT) THEN ERROR(11);
      PTR:=NIL;
      GARBAGEMAN;
      WRITELN;
      WRITELN;
      WRITELN('*EVAL*');
      NEXTSYM;
      READEXPR(PTR);
      READLN;
    WRITELN
    END;
2: WRITELN;
   WRITELN;
```

```
  WRITELN('TOTAL NUMBER OF GARBAGE COLLECTIONS=', NUMBEROFGCS:3,'.');
  WRITELN;
  WRITELN('FREENODES LEFT UPON EXIT=',FREENODES:3,'.');
  WRITELN;
END.  (*LISP*)
```

APPENDIX - B

(RESULTS)

```
 *EVAL*
  (CAR (QUOTE ((A B) (A B C))))

*VALUE*
(A B)


GARBAGECOLLECTION.

NUMBER OF FREE NODES BEFORE COLLECTION=568.
NUMBER OF FREE NODES AFTER COLLECTION=591.


*EVAL*

   (CDR (QUOTE ((A B) (A B C))))

*VALUE*
((A B C))


GARBAGECOLLECTION.

NUMBER OF FREE NODES BEFORE COLLECTION=568.
NUMBER OF FREE NODES AFTER COLLECTION=591.


*EVAL*

   (CONS (QUOTE A) (QUOTE (B C)))


*VALUE*
(A B C)


GARBAGECOLLECTION.

NUMBER OF FREE NODES BEFORE COLLECTION=571.
NUMBER OF FREE NODES AFTER COLLECTION=591.
```

```
*EVAL*
  (APPEND (QUOTE (A B)) (QUOTE (C D)))


*VALUE*
(A B C D)


GARBAGECOLLECTION.

NUMBER OF FREE NODES BEFORE COLLECTION=564.
NUMBER OF FREE NODES AFTER COLLECTION=591.


*EVAL*
  (LIST (QUOTE (A B)) (QUOTE (C D)))


*VALUE*
((A B) (C D))


GARBAGECOLLECTION.

NUMBER OF FREE NODES BEFORE COLLECTION=565.
NUMBER OF FREE NODES AFTER COLLECTION=591.


*EVAL*
  (REPLACAR (QUOTE ((A B) B C)) (QUOTE A))


*VALUE*
(A B C)


GARBAGECOLLECTION.

NUMBER OF FREE NODES BEFORE COLLECTION=566.
NUMBER OF FREE NODES AFTER COLLECTION=591.
```

```
*EVAL*
  (REPLACDR (QUOTE (A (A B))) (QUOTE B))


*VALUE*
(A.B)


GARBAGECOLLECTION.

NUMBER OF FREE NODES BEFORE COLLECTION=568.
NUMBER OF FREE NODES AFTER COLLECTION=591.
*EVAL*
   ((LABEL MEMBER (LAMBDA (X Y)
           (COND ((NULL Y) (QUOTE NIL))
                 ((EQUAL X (CAR Y)) (QUOTE T))
                 ((QUOTE T) (MEMBER X (CDR Y))))))
     (QUOTE (A B))
     (QUOTE (C D A B (A B))))


*VALUE*
T


GARBAGECOLLECTION.

NUMBER OF FREE NODES BEFORE COLLECTION=388.
NUMBER OF FREE NODES AFTER COLLECTION=591.
```

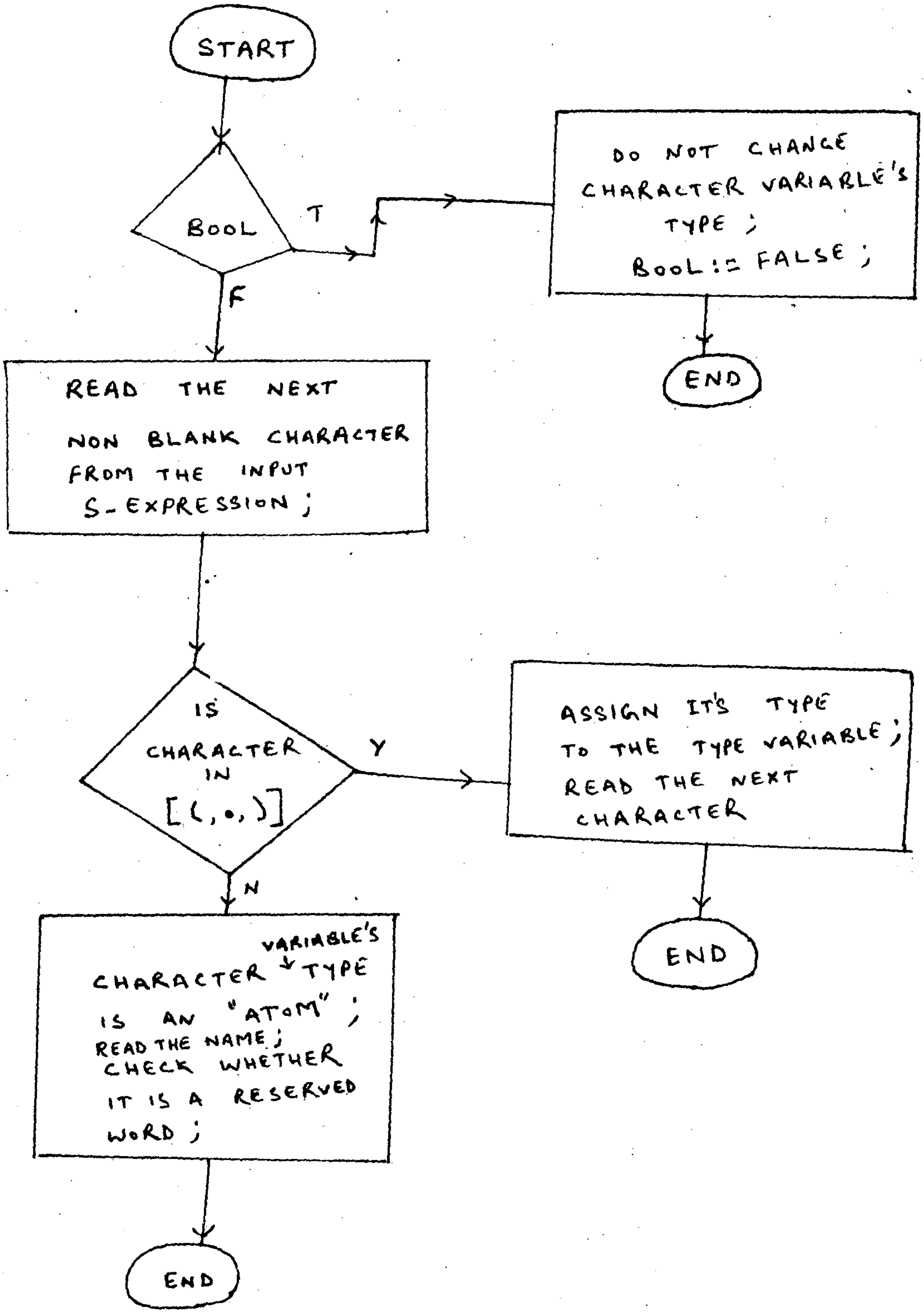
```
*EVAL*
  ((DEFINE (MEMBER LAMBDA (X Y)
                   (COND ((NULL Y) (QUOTE NIL))
                          ((EQUAL X (CAR Y)) (QUOTE T))
                          ((QUOTE T) (MEMBER X (CDR Y)))))
            (UNION LAMBDA (X Y)
                   (COND ((NULL X) Y)
                         ((MEMBER (CAR X) Y) (UNION (CDR X) Y))
                         ((QUOTE T) (CONS (CAR X) (UNION (CDR X) Y))))))
        (UNION (QUOTE (A B C D)) (QUOTE (A E F B G H))))


*VALUE*
(C D A E F B G H)


GARBAGECOLLECTION.

NUMBER OF FREE NODES BEFORE COLLECTION=407.
NUMBER OF FREE NODES AFTER COLLECTION=1191.
```

```
*EVAL*
  ((DEFINE (REVERSE LAMBDA (X)
                       (PROG (U V)
                       ((SETQ U X)
                 ((A)  (COND ((NULL U) (RETURN V))
                             ((QUOTE T) ((SETQ V (CONS (CAR U) V))
                                        ((SETQ U (CDR U))
                                        (GO ((A))))))))))))
           (REVERSE (QUOTE (A B D E F G H))))


*VALUE*
(H G F E D B A)


GARBAGECOLLECTION.

NUMBER OF FREE NODES BEFORE COLLECTION=937.
NUMBER OF FREE NODES AFTER COLLECTION=1191.
*EVAL*
   (AND (QUOTE T) (QUOTE T) (QUOTE NIL))


*VALUE*
NIL


GARBAGECOLLECTION.

NUMBER OF FREE NODES BEFORE COLLECTION=1169.
NUMBER OF FREE NODES AFTER COLLECTION=1191.
```

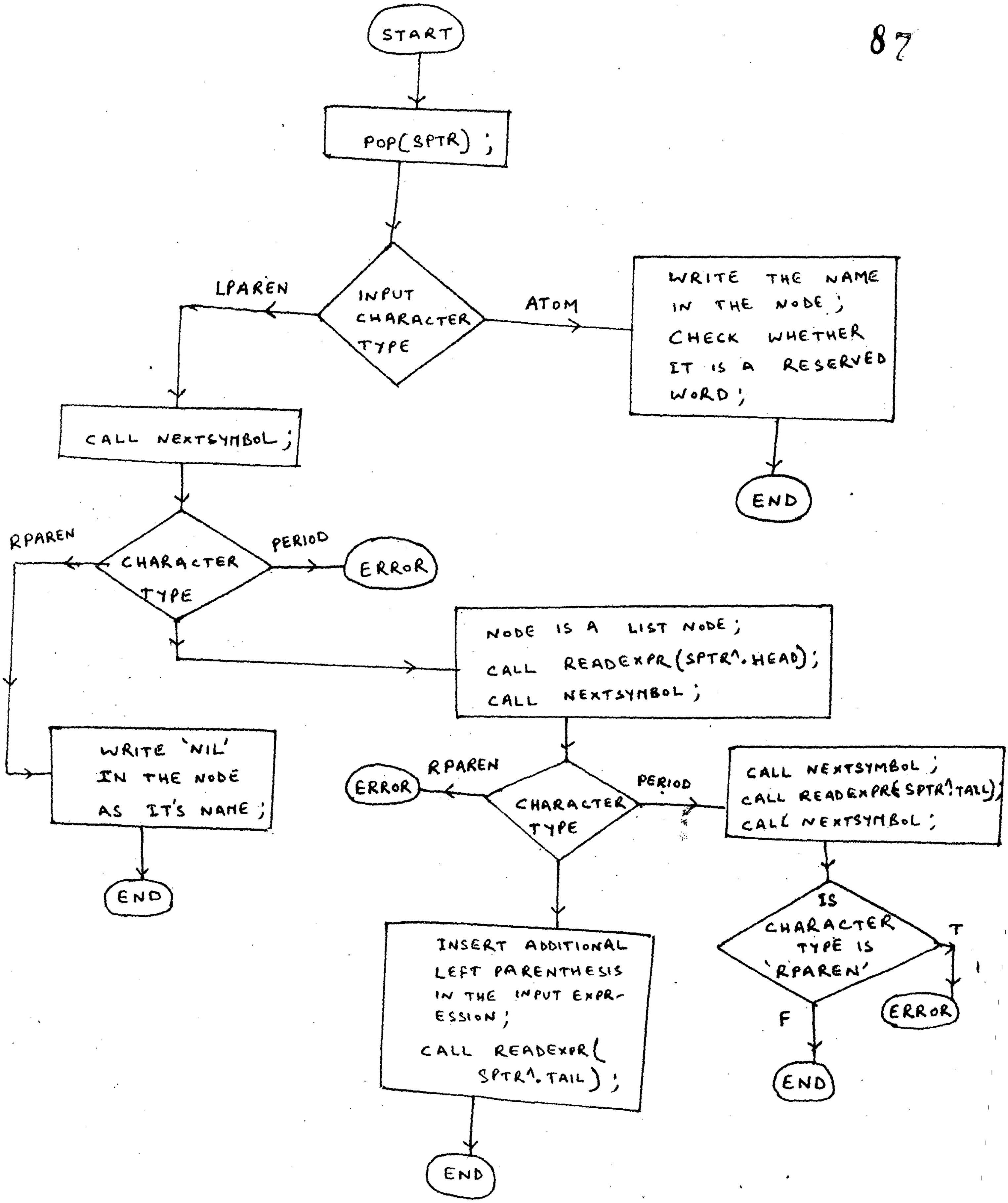
```
*EVAL*
  (OR (QUOTE NIL) (QUOTE NIL) (QUOTE T))

*VALUE*
T


GARBAGECOLLECTION.

NUMBER OF FREE NODES BEFORE COLLECTION=1169.
NUMBER OF FREE NODES AFTER COLLECTION=1191.


*EVAL*
  (NOT (OR (QUOTE NIL) (QUOTE T)))

*VALUE*
NIL
```

APPENDIX - C

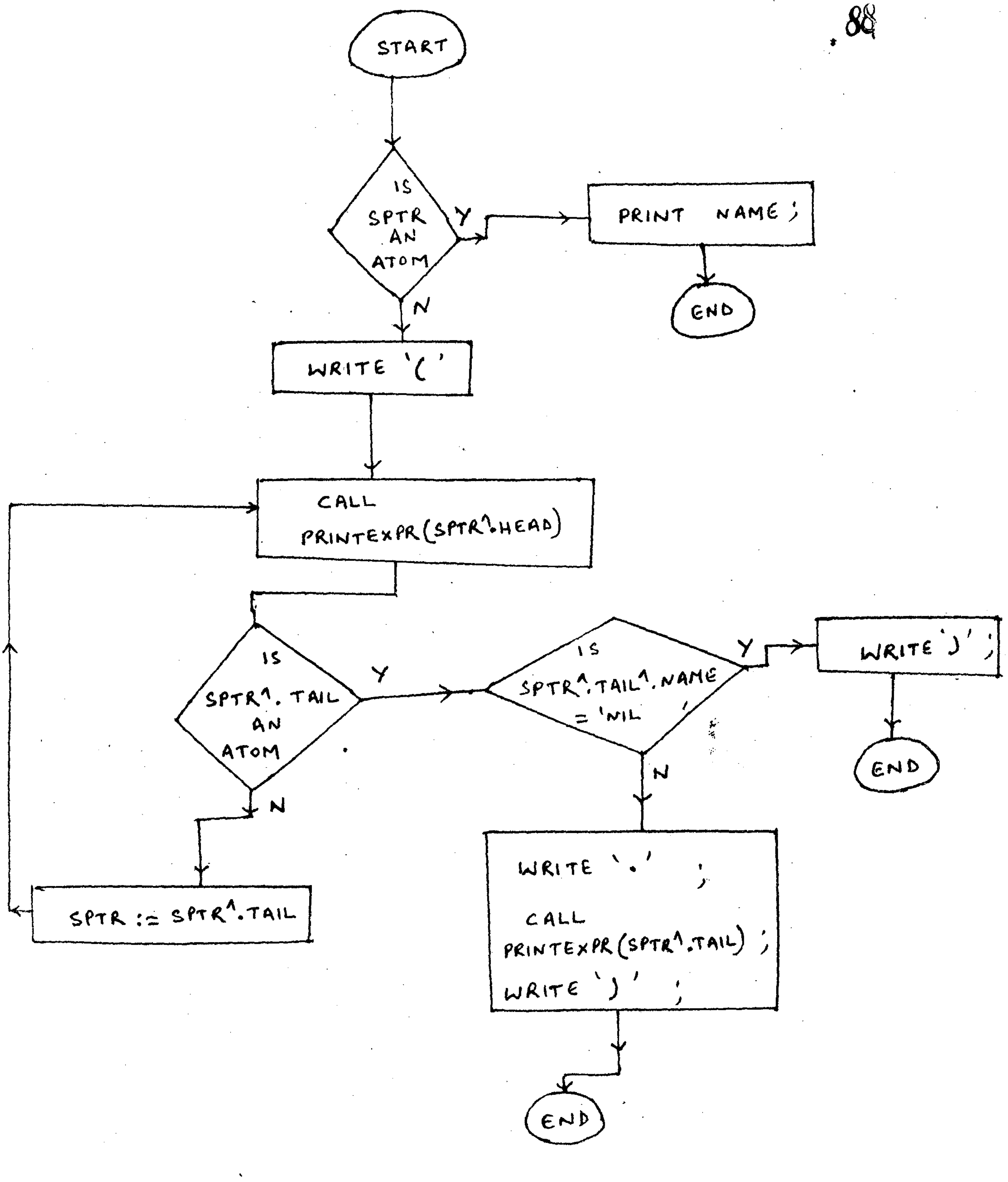
(FLOW DIAGRAMS)



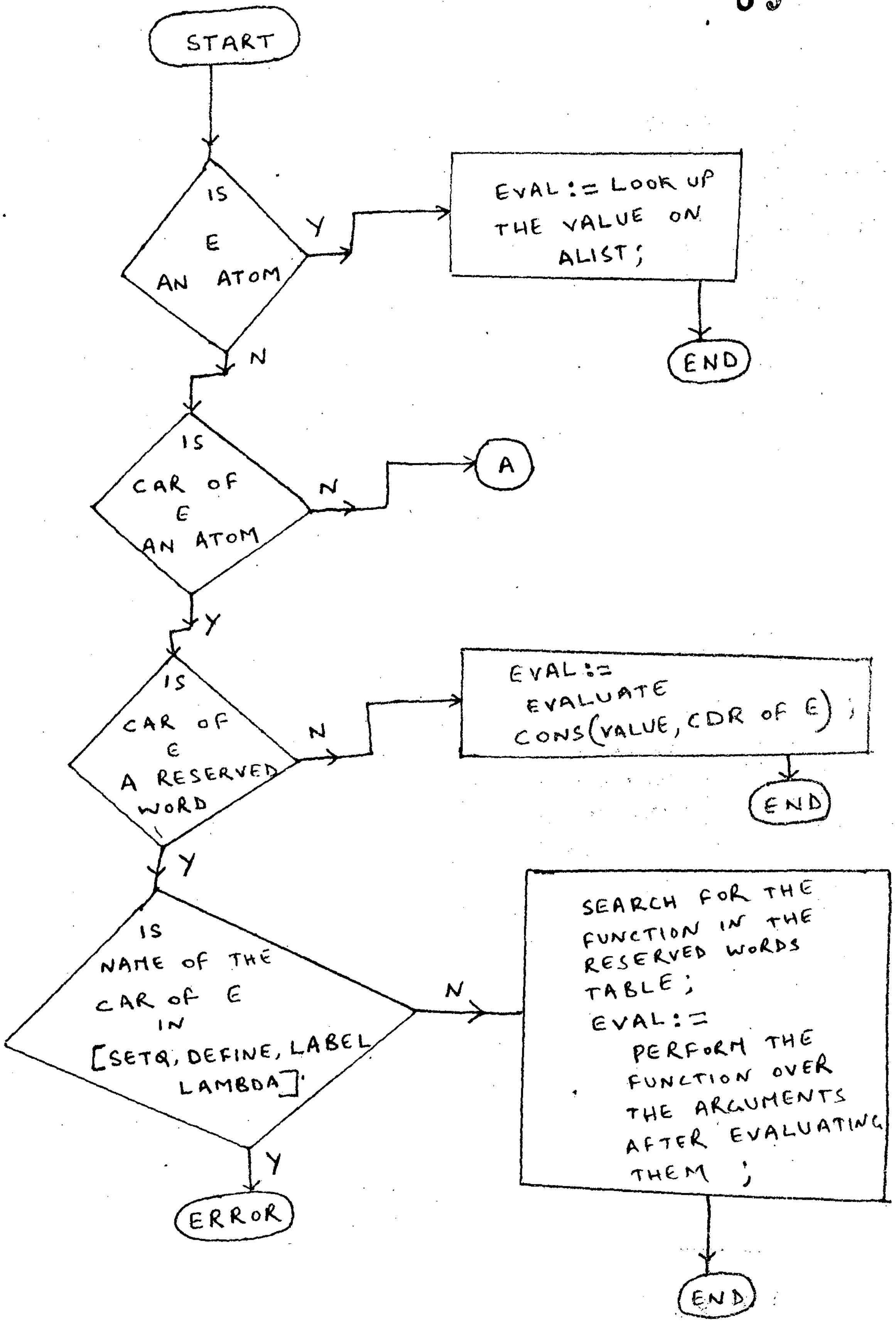
PROCEDURE NEXTSYMBOL



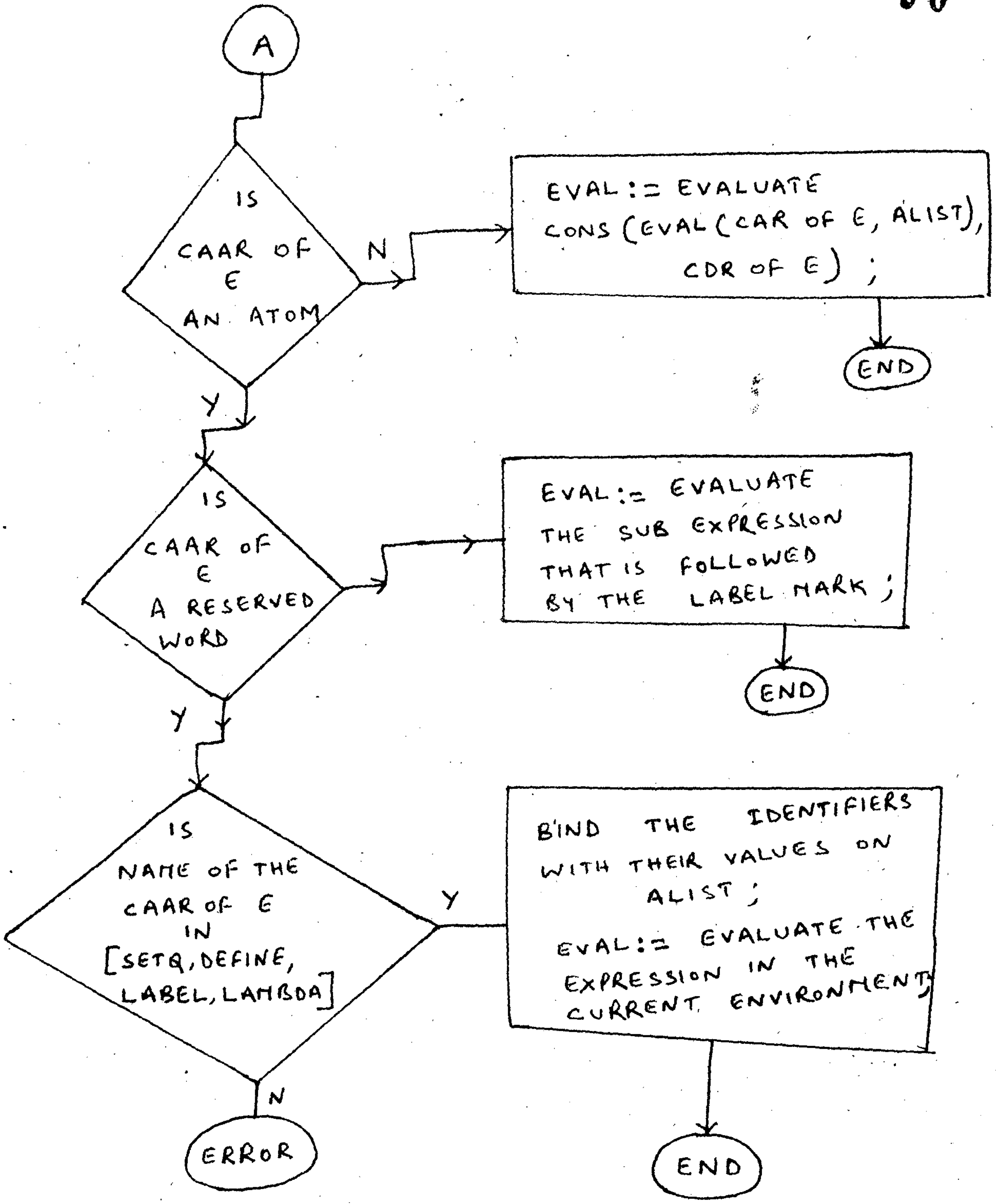
: PROCEDURE READEXPR(SPTR) :



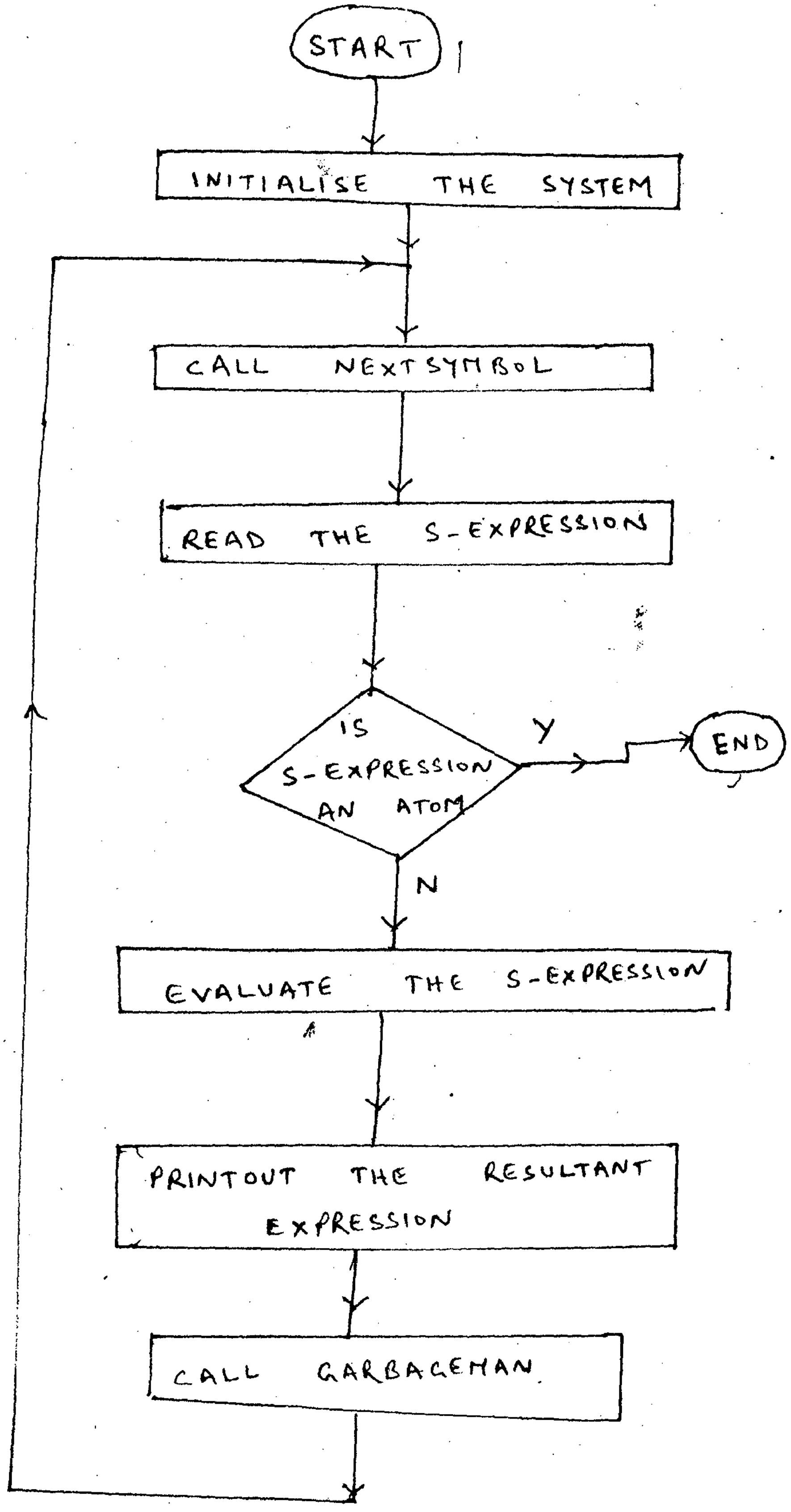
PROCEDURE PRINTEXPR (SPTR) :



: FUNCTION EVAL(E, ALIST) :



FUNCTION EVAL(E, ALIST) (CONTINUED) :



: "LISP" INTERPRETER :

APPENDIX - D

(REFERENCES)

1. B.F. Green, "Computer Languages for Symbol manipulation", IRE Trans., HFE2 (March 1961).
2. B. Raphael, "Aspects and applications of symbol manipulation", Proc. 21st Natl. Conf., ACM (Aug 1966).
3. M.V. Wilkes, "Lists and why they are useful", Proc. 19th Natl. Conf. ACM (Aug 1964).
4. J.E. Sammet, "Formula manipulation by Computer", TROO.1363, IBM Systems Development Division, Poughkeepsic, N.Y., (Nov. 1965).
5. John McCarthy, "Recursive Functions of Symbolic expressions and their Computation by Machine", Comm. of ACM (April 1960).
6. Abrahams, P., "Digital Computer User's Handbook (McGraw Hill 1967).
7. "An over-view of the state-of-the-Art in symbol manipulation", Comm. ACM (Aug. 1966).
8. John McCarthy et al, "LISP 1.5 Programmers Manual", (MIT Press, 1962).
9. Winston, P.K., Horn, B., K., P., "LISP" (Addison-Wesley, 1981).
10. K. Jensen and N.Wirth, "Pascal User Manual and report", (Springer Verlog, 1978).
11. W. Taylor, and L. Cox, "The Essence of LISP interpreter", Pascal News, PUG, (Sept., 1980).
12. H. Schorr, and W.M. Waite, "An Efficient Machine-Independent Procedure for Garbage Collection in various List Structures", Comm. ACM, (Aug., 1967).
13. Y. Kishan Reddy, and R. Sadananda, "A Structured Implementation of LISP for Pedegogical Purposes", proceedings of A Natl. seminar on COMPUTER AND THE SOCIETY, College of Engg., Anna University, Madras, India, Feb. 23-24 (1983).
14. A Darlington, P. Henderson, and D.A. Turner, "Functional Programming and its applications" (Cambridge).
15. J.P. Fitch, and A.C. Norman, "Implementation of LISP in a high level language", Software Practice and Experience (1977).

16. Carlo Ghezzi, and Mehd Jazayeri, "Programming Language Concepts" (John Wiley & Sons, Inc.)

17. Paul W. Abrahams, "Symbol manipulation Languages", Advances in Computers, Vol. 9, 1968, pp. 51-110.

18. Wirth N., "On the design of Programming Languages" in IFIP Congress 74, Vol.2: Software, 1974, 386-393 .

19. Wirth N., "An assessment of the Programming Language PASCAL", IEEE trans., Software Engg., SE-1,2, pp. 192-198 (June 1975).