A LISP INTERPRETER: SCANNER AND PARSER

by

DAVID CLARENCE BOSSERMAN

B.G.S., University of Nebraska at Omaha, 1969

A MASTER'S REPORT

submitted in partial fulfillment of the

requirements for the degree

MASTER OF SCIENCE

Department of Computer Science

KANSAS STATE UNIVERSITY

Manhattan, Kansas
1977

Approved by:

Linda Shapiro
Major Professor

TABLE OF CONTENTS

SECTION NAME CHAPTER THREE SCANNER PAGE

SECTION NAME CHAPTER FOUR PAGE
PARSER

CHAPTER FIVE
INTERDATA 8/32 INFORMATION

CHAPTER SIX
TESTING

APPENDICES

ILLUSTRATIONS

Chapter 1

PROJECT CONCEPTS AND DESIGN

1.1 INTRODUCTION

In this paper a portion of a LISP Interpreter is proposed for use on minicomputers. The development of the interpreter was done in the FORTRAN Language. The purpose of the project was:

(1) To design a LISP Interpreter.
(2) To code the scanner and parser in a high-level language.
(3) To exercise the program on the INTERDATA 8/32 computer.

Before the details of the program are discussed, program concepts will be addressed, and a discussion of the interaction with the INTERDATA 8/32 will be presented.

1.2 LISP LANGUAGE

The LISP language is a LISt Processing language. It is designed primarily for symbolic data processing and has been used extensively in solving mathematical problems, working with electrical circuit theory, game playing and other applications of artificial intelligence.

LISP is a formal mathematical language. As such, it is possible to give a clear, concise and complete definition of it. Due to these qualities, it is an excellent language to work with in designing, coding and implementing an interpreter.

This project deals with the implementation of the LISP

1.5 Programming Language. It is not restricted at all by the design structure of the scanner or parser. In fact, the scanner and parser modules were designed for maximum portability and adaptability.[1]

1.3 LISP INTERPRETER SYSTEM DESIGN

The interpreter's system design consists of one driver module and five major subsystem driver modules. (See Figure 1-1.) Each module will be discussed separately.

1.3.1 LISP DRIVER

The LISP DRIVER module contains two major categories of data: (1) information concerning the module itself; and (2) the common data declared for intersubroutine use. The code for this module is in APPENDIX B.

1.3.2 INITIALIZER DRIVER

The function of the INITIALIZER module is to: (1) control memory management; (2) initialize the function tables; and (3) initialize the argument tables. This driver and its functional routines were prepared by Lee R. Whitley and are discussed in his report.

---

[1] John McCarthy et al., LISP 1.5 Programmer's Manual (Cambridge: The M.I.T. Press, 1973), p.1.

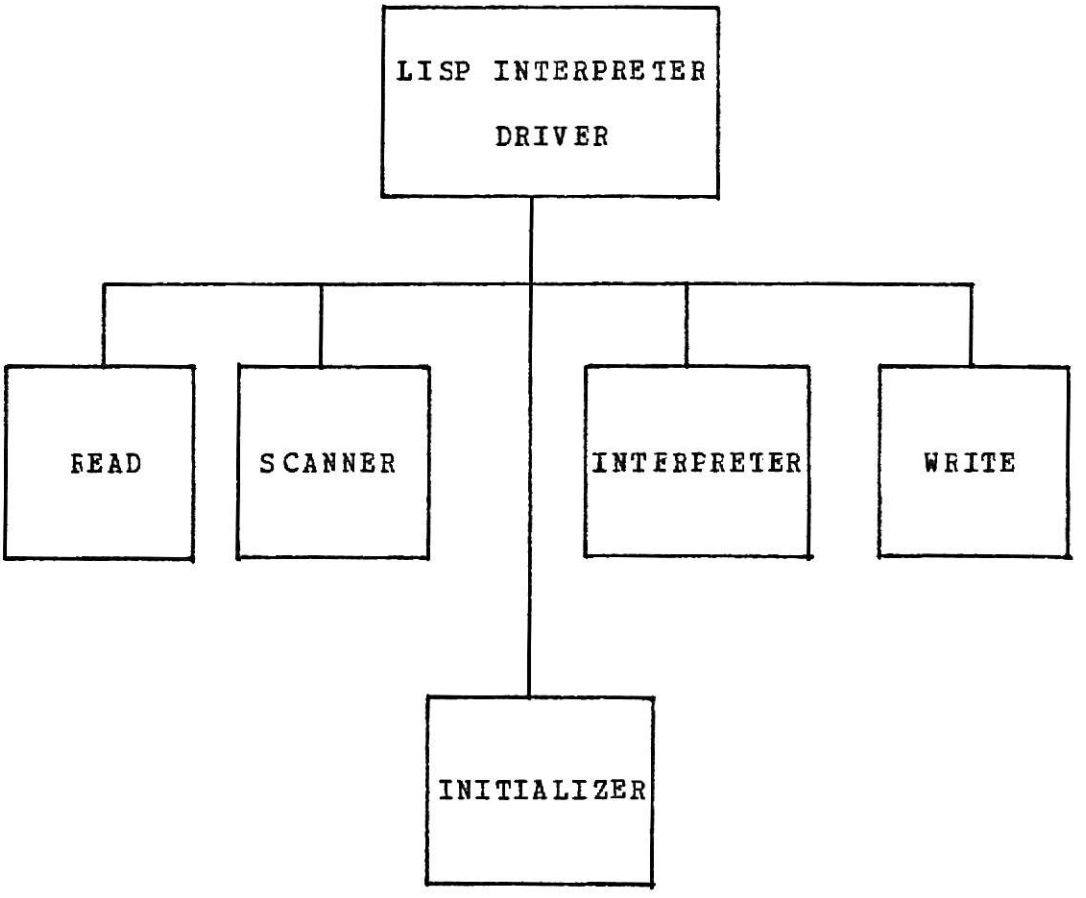

Figure 1-1

1.3.3 READ MODULE

The read routine is responsible for: (1) requesting memory space from the memory management routines; and (2) reading the user's program into addressable storage locations. The code for the read routine is found in APPENDIX C.

1.3.4 SCANNER DRIVER

The scanner routine is a driver routine. It calls the token generator routine and symbol table search routines. It causes the user's program to be scanned. From this scan an address array is generated showing: (1) the location of each symbol/string in the user's program; (2) the length of each symbol/string; (3) its keyword/function status; and (4) its location within a keyword/function table, if applicable. The scanner code is in APPENDIX D.

1.3.5 INTERPRETER DRIVER

The interpreter performs two major functions. It initially creates a general tree representation of the user's program. Once the general tree has been created, it is passed to the execution modules. The general tree construction modules are in APPENDIX E. The execution routines have not been completed and are to be the subject of a future reports. The interpreter module is in APPENEIX E.

1.4 MODULE CONSTRUCTION AND INTERACTION

The modules utilized in this interpreter were designed using a top-down structured programming approach. Each module has been restricted to one path into the module and one path out upon completion of its function. Each module has also been restricted in the number of lines of code

which it contains. No module contains more than 100 lines of executable code.

The modular concepts have also been implemented in module interaction. Except for several modules calling memory management routines, no modules interact outside the span of control of a mutual driver. Any passing of data is done through call parameters or common storage.

1.5 INTERDATA 8/32

The INTERDATA 8/32 Computer was used to test the interpreter. It provided an excellent contrast of small vs large computer procedures. The INTERDATA is a very reliable computer but, due to the newness of its hardware and software, caused many problems during project completion. The INTERDATA and its peculiarities will be discussed in greater detail in Chapter 5.

1.6 SUMMARY

The project discussed in this report is one part of a three part effort to develop a high-level language LISP interpreter. The specific parts of the interpreter covered by this report are: (1) the READ module; (2) the SCANNER modules; and (3) a portion of the INTERPRETER modules. All portions covered by this report are complete and have been tested.

The INTERDATA 8/32 was used to test the interpreter modules. It provided an excellent vehicle for the testing

as it demonstrated the pitfalls and frustrations to be encountered on a computer which is relatively new and without complete maintenance and software packages.

Although users working on smaller computers will find that more time and effort is required to accomplish tasks, this report shows that with perseverance and hard work, comparable results to large computer output can be obtained.

Chapter 2

READING THE USER'S PROGRAM

2.1 INTRODUCTION

This chapter presents information concerning the specifics and requirements of the read module. The read module (named INREAD) does not drive any other modules. Its function is simply to read the user's program into storage so that the program may be processed by other modules. The code for INREAD is in APPENDIX C.

2.2 INREAD SPECIFICS

The INREAD module is designed to read the user's program into storage. The module is called from the LISP INTERPRETER DRIVER module and has the relationship shown in Figure 2-1. Note that INREAD only calls one module, ISINIT. ISINIT is the routine that gets space from memory management when required.

2.2.1 PARAMETERS

The INREAD parameters are: (1) IFIRST, the beginning logical address in memory of the user's program; (2) IDONE, the ending logical address in memory of the user's program; and (3) ISTART, the last logical address passed to the routine by memory management. The IFIRST and IDONE variables are used by the scanner routines to convert the

user's program into a sequence of tokens.

READ MODULE RELATIONSHIP

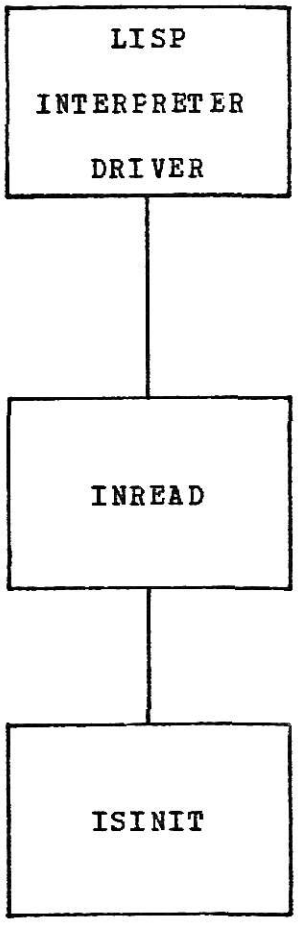

Figure 2-1

2.2.2 INPUT

The input required by the INREAD module are: (1) the user's program; and (2) a portion of an array called ISPACE which is acquired from memory management. As stated in chapter one, the user's program will be read from a file

created by the user. The file may be created by sitting at a terminal and typing it, or by allocating space for the file at a terminal and reading cards from the card reader into the file. The association of the input file with the reader logical unit in the program is done after the program tasks are loaded. In the interpreter, all reader logical units are logical unit 5. Therefore, if the user created his input file and called it INPROG, his assignment would be:

AS 5,INPROG

The other input required by this program is the logical addresses of space allocated by memory management to store the user's program. Logical addresses in this sense mean the array elements allocated. A contiguous block of space is requested so that later processing can be efficiently accomplished. Contiguous space is assured by sending a code to the memory management routine. A code of IBLOCK=0 causes contiguous space allocation whereas a code of IBLOCK=1 informs the memory routines that headers may be written for the space acquired.

2.2.3 OUTPUT

The output produced by INREAD is: (1) the user's source program stored in an array called ISPACE; (2) space allocated to enter the level of parentheses; (3) the array element which contains the start of the user's program; and (4) the array element which contains the ending of the user's program.

The user's source program with appropriately numbered parentheses is retained until execution of the program is completed. This is done for several reasons. First, it provides a list file of the program for output purposes. Second, it allows for error marking if desired in later implementations. Finally, it is required because other modules simply point to the various keywords/strings contained in the program. Therefore, there is no requirement for any rewriting of strings into any storage space.

Note that space is also allocated for the numbering of parentheses. This is done so that the parentheses levels, when determined can be retained. This precludes having to call a level determining routine every time the level is required. It also enhances the tree construction in the INTERPRETER modules.

Chapter 3

SCANNER

3.1 INTRODUCTION

This chapter presents information concerning the scanner modules. The scanner modules are designed to: (1) scan the user's program and record each string's position and length; and (2) update the token to provide information concerning whether a string is a keyword, and if so, in which symbol table it is found and its location within the table.

The scanner driver routine is called ISCAN. ISCAN calls the routines ISTKGN and ISSMTB to accomplish its mission. The relationship of the modules is shown in Figure 3-1.

3.1.1 GRAMMAR

The grammar used by the interpreter consists of the following reductions:

(1) <LETTER>::= A|B|C|D...|Z|SPECIAL CHARACTERS

(2) <NUMBER>::= 0|1|2|3...|9

(3) <BLANKS>::= ONE OR MORE BLANKS

(4) <ATOMIC SYMBOL NON-NUMERIC>::=<LETTER><ATOM PART>

(5) <ATOMIC SYMBOL NUMERIC>::=<ATOMIC SYMBOL INTEGER>|
<ATOMIC SYMBOL FLOAT>

(6) <ATOMIC SYMBOL INTEGER>::=<NUMBER>|
<NUMBER><ATOMIC SYMBOL INTEGER>

(7) <ATOMIC SYMBOL FLOAT>::=<ATOMIC SYMBOL INTEGER>.|
<ATOMIC SYMBCL INTEGER>.<ATOMIC SYMBOL INTEGER>|
0.<ATOMIC SYMBOL INTEGER>

(8) <ATOM PART>::=<NULL>|<NUMBER><ATOM PART>|
<LETTER><ATOM PART>

(9) <S-EXPRESSICN>::=<ATOMIC SYMBOL NON-NUMERIC>|
<ATOMIC SYMBOL NUMERIC>|
(<LIST>)

(10) <LIST>::=<S-EXPRESSION>|
<S-EXPRESSION><SEPARATERS><LIST>

(11) <SEPARATERS>::= <BLANK>|<BLANK.>[2]

The atomic symbol, or atom, is the most elementary type of S-expression. It can be numeric or non-numeric. If the string is non-numeric, it is considered a literal atom and consists of a string of capital letters and decimal digits having a letter as the first character. The atomic symbol is called atomic because it is taken as a whole and not viewed as individual characters.

S-expressions are non-atomic. They are built of atomic symbols and punctuation marks. The S-expression is always surrounded by a set of parentheses and always has two parts, a left part and a right part. Either part can be a null string. The S-expression can be either: (1) an atom; (2) a pair of atoms separated by a dot or spaces; or (3) a pair of S-expressions separated by a dot or spaces.[3]

---

[2] Ibid., p.8.
[3] Clark Weissman, LISP 1.5 Primer (Belmont, California: Dickenson Publishing Company, Inc., 1967), pp.5-6.

3.1.2 OVERVIEW

The user's program is scanned in two separate phases. The first phase, using the ISTKGN modules, causes: (1) a token to be stored for each string, left parenthesis and right parenthesis; and (2) all space delimiters and period delimiters to be ignored. Each token generated contains the logical address and length of a parenthesis or string in storage. These tokens are expanded in the second phase of the scan.

The ISSMTB modules perform the remainder of the scanner operation. They process each initial token and determine if the token represents a keyword found in the keyword tables. If so, the token is expanded to record in which table the keyword was found and where within the table it was found. If not, the token is expanded to record that the string is program unique. Once all tokens are expanded, control is returned to the LISP INTERPRETER DRIVER and a parse of the tokens is begun.

3.2 ISCAN

The ISCAN module causes tokens to be generated which provide: (1) the location of a string in memory; (2) the length of the string; (3) the key word table that the string is listed in, if applicable; and (4) the location of the string within the table, if applicable.

The ISCAN routine does no processing of the user's program per se. It simply acts as a driver routine and

calls the routines ISTKGN and ISSMTB in that order. ISTKGN is responsible for identifying strings and recording their location and length. ISSMTB is responsible for performing the table lookup function to complete the token. The ISCAN module's code is located in APPENDIX D.

SCANNER MODULES RELATIONSHIP

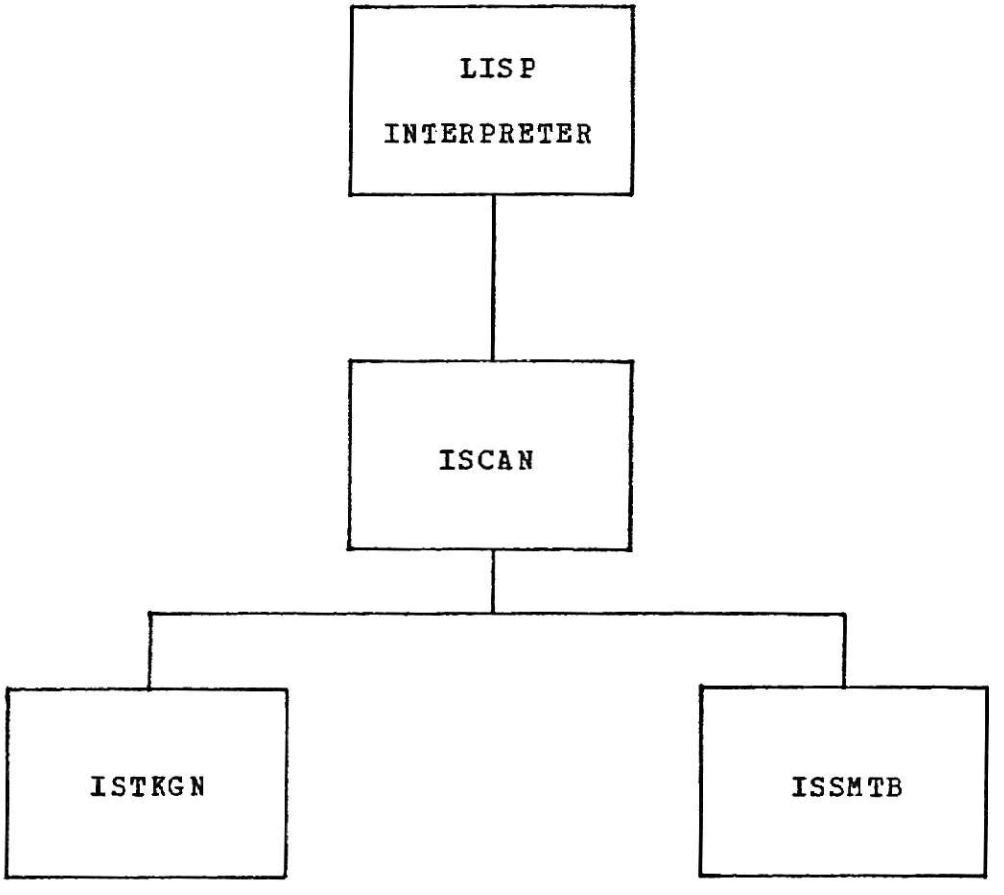

Figure 3-1

3.3 ISTKGN

The ISTKGN module causes the first two items of a

general token to be processed and placed in a token array. The array created will remain in existance until the user's program has completed execution. The space acquired from memory management will again be contiguous space to facilitate future processing.

ISTKGN is not a driver module. Its purpose is to process each array element containing the user's program and call the appropriate subroutine: (1) if a parenthesis is encountered; (2) if a string is enccuntered; and (3) if a null s-expression is encountered.

The format of the initial token generated by ISTKGN is as shcwn in Figure 3-2. In the ISSMIB routines the tokens generated by ISTKGN will be expanded to show the location of the strings within keyword tables, if applicable.

INITIAL TOKEN FORMAT

Figure 3-2

3.3.1 PARAMETERS

The ISTKGN parameters are: (1) IFIRST, which represents the first element in the array ISPACE containing the user's program; (2) IDONE, which represents the last

element in the array ISPACE containing the user's program; (3) IJUMP, which represents the first element in the array ISPACE containing the tokens generated; and (4) ISTART, which is initially equal to IDONE and is updated to represent the last element in the array ISPACE containing a token.

3.3.2 INPUT

The majority of the input to this routine is done through parameters passed. The only other input is the array elements allocated, upon request, by memory management to store the tokens generated.

3.3.3 OUTPUT

The output generated by this routine is a sequence of tokens representing the beginning address and length of each string in the user's program. These tokens are stored in the array ISPACE and their locations are passed to other routines by passing the beginning element (IJUMP) and the ending element (ISTART).

3.3.4 MODULAR ORGANIZATION

Although the routine ISTKGN is not solely a driver module, it does call three other modules to process the symbols. Those modules in turn call the module ISINIT to acquire space from memory management. The modular

relationship is shown in Figure 3-3.

ISTKGN MODULAR RELATIONSHIP

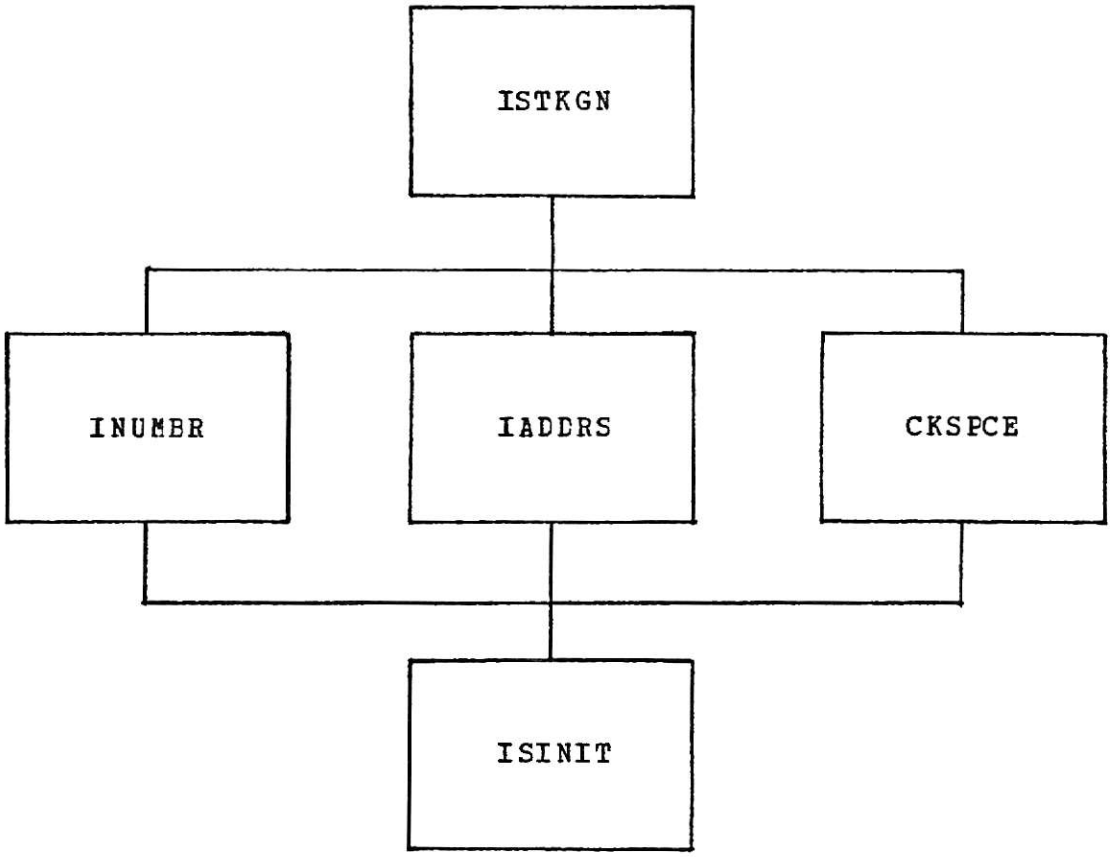

Figure 3-3

3.3.5 OVERVIEW

The high-level algorithm used by the scanner is shown in Figure 3-4. As the algorithm shows, ISTKGN accomplishes its function by looking at every character represented in the user's program. Because parentheses are used abundantly

in LISP programming to show levels, the module first looks

SCANNER HIGH-LEVEL ALGORITHM

```
DO WHILE NOT END OF FILE;
  GET NEXT CHARACTER

CASE 1   CHARACTER IS LEFT PARENTHESIS
         GET TOKEN SPACE
         PUT LOCATION AND LENGTH=1 IN SPACE

CASE 2   CHARACTER IS RIGHT PARENTHESIS; NUMCHR=0
         GET TOKEN SPACE
         PUT LOCATION AND LENGTH= 1 IN SPACE

CASE 3   CHARACTER IS RIGHT PARENTHESIS; NUMCHR GT 0
         GET TOKEN SPACE
         PUT STRING AND NUMCHR IN SPACE
         GET TOKEN SPACE
         PUT LOCATION OF PARENTHESIS AND
           LENGTH=1 IN SPACE

CASE 4   CHARACTER IS SPACE; NUMCHR=0
         IF LOCATION-1 IS LEFT PARENTHESIS AND
            LOCATION+1 IS RIGHT PARENTHESIS
          THEN: GET SPACE
                PUT LOCATION AND LENGTH=1 IN SPACE
          ELSE: NULL

CASE 5   CHARACTER IS SPACE; NUMCHR GT 0
         GET SPACE
         PUT LOCATION-NUMCHR AND LENGTH IN SPACE

CASE 6   CHARACTER IS PERIOD; NUMCHR=0
         NULL

CASE 7   CHARACTER IS NOT CASE 1 - CASE 6
         INCREMENT NUMCHR

END DO WHILE;
```

Figure 3-4

to see if an element contains a parenthesis. If so, the routine INUMBR is called to process it. If it is a left parenthesis the level number assigned to it will be one greater than the last left parenthesis unless a right parenthesis has appeared since the last left parenthesis. If this is the case, it will carry the same number as the last left parenthesis as it is at the same level. If the parenthesis encountered is a right parenthesis, INUMBR is called to assign it the proper number. Whenever INUMBR is called, a token is generated.

If the element processed is not a parenthesis, it could be a delimiter (space or period) or a string. If it is a string, a counter is incremented and processing continues until a parenthesis or delimiter is encountered.

Once a delimiter is encountered, the string represented by the length counter is placed in the token array. One exception to this procedure is made. If a space is encountered, and the length counter is zero, the routine called ICKSPC is called to check the space to see if it is in fact a null string. If so, a token is generated. A null string is defined as a left parenthesis, single blank character, and a right parenthesis. If any other combinations of parentheses and blank characters occur they will not be recognized as a null string.

3.4 INUMBR

The INUMBR routine is responsible for numbering

parentheses. The parentheses' numbers are called level numbers and are stored in memory in the space allocated when the user's program was read. (See Paragraph 2.2.3.) After storage they are used by the parser to determine argument and S-expression length. They are also available to be printed by the WRITE modules for user convenience. The routine must deal with two major situations. The first is the situation of a left parenthesis. The procedure then is to simply number the parenthesis with the appropriate number. The second situation is the case of a right parenthesis which closes a string. In this case the routine is required to construct a token for the string and a token for the parenthesis. The code for INUMER is in APPENDIX D. The only other module called by INUMBR is ISINIT to get space to store the tokens generated.

3.4.1 INPUT

The input data to the INUMBR routine are: (1) whether the element is a left or a right parenthesis (INB); (2) the location of the array element pointer (I); (3) the number of characters found in the string; and (4) the current parenthesis level (N).

Of course, there is a continuing global input from memory management. This input provides the space to store the tckens generated.

3.4.2 OUTPUT

The output data are provided to allow modules to properly continue processing. IPROCE is a code which keeps ISTKGN from double processing an element. ISTART is passed so that the end address of the token array can be set.

3.5 IADDRS

The IADDRS routine is called when a string exists and a space delimiter is encountered. The existence of a string is known because NUMCHR is greater than zero. Note that the period delimiter does not enter into this routine because period delimiters are preceded by a space and succeeded by a space. Thus, a period is essentially read and skipped. The code for IADDRS is in APPENDIX D.

3.5.1 INPUT

The input required for the IADDRS routine consists of those variables necessary to identify the end of the string (I), and the length (NUMCHR). With this data the necessary token can be generated

In this module, an additional input is required from memory management. Memory management must provide the array element to store the token generated.

3.5.2 OUTPUT

The data output by this module are; (1) the token

representing the beginning address and the length; and (2) the pointer to the array element which contains the token.

The variable IPROCE is assigned a code value and passed back to the calling routine. This code prevents double processing of an element.

3.6 CKSPCE

The CKSPCE routine performs the function of testing blank characters, when NUMCHR is equal to zero, to see if the blank character represents a null string. To accomplish this task, the routine requires one look back and one look forward. The look back and look forward are done to see if the space is surrounded by parentheses.

CKSPCE only requires the pointer to the blank character in the source program. Using that pointer a look back and look forward can be accomplished. IPROCE is used as stated above, as is ISTART.

3.7 ISINIT

Although the ISINIT routine is not used only by the ISCAN routines, it will be discussed at this time. The ISINIT routine is a utility routine used to get space from memory management.

Three parameters are passed to ISINIT. The two input parameters are: (1) NUMRQD, the number of array elements required from ISPACE; and (2) IBLOCK, a code which informs memory management when contiguous storage is needed and when

to cancel it. If IBLOCK equals 1, contiguous storage is required. If IBLOCK equals 0, contiguous storage is either no longer required or not required at all. The output parameter ISTART contains the beginning array element allocated by memory management.

As a utility routine ISINIT is called by any routine requiring space in memory. The code for ISINIT is in APPENDIX D.

3.8 ISSMTB

ISSMTB processes the tokens generated by ISTKGN and expands them. In the expanded token, a table location and table are added to the already existing beginning address and length. The code for ISSMTB is found in APPENDIX D. The format of the expanded token is shown in Figure 3-5.

The digit representation for the token is: (1) Beginning Address=(4 digits); (2) Length=(2 digits); (3) Table Location=(2 digits); (4) Table=(1 digit); and (5) Mark=(1 digit). The beginning address and length are the same as determined in ISTKGN. The number placed in the table field is the number of the keyword table in which the string was found. The table location field contains the element within the table which represents the string. The keyword tables are initialized by the INITIALIZER modules and remain in storage until execution is complete. The mark field is not used by the scanner or parser. It was implemented to be used by the execution modules. For example, in Figure 4-2, CAR is represented in token form as

1030310. This shows that it is stored in position one of the user's program and has a length of 3. It is also contained in keyword table 3 in position 1.

EXPANDED TOKEN FORMAT

BEGINNING ADDRESS	LENGTH	TABLE LOCATION	TABLE	MARK

Figure 3-5

3.8.1 INPUT

The input data to ISSMTB are the array containing the tokens generated in ISTKGN along with the beginning (IJUMP) and ending (ISTART) pointers. No other input is required.

3.8.2 OUTPUT

ISSMTB outputs an expanded token array. It requires no additional space from memory management. IFINSH is output as a pointer to the last element in the token array. As the expanded tokens occupy the same array elements as the tokens generated in ISTKGN, no other pointers are required.