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:

[Signature]
Major Professor
## TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>SECTION NAME</th>
<th>CHAPTER ONE</th>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>PROJECT CONCEPTS AND DESIGN</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.1 INTRODUCTION</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>1.2 LISP LANGUAGE</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>1.3 LISP INTERPRETER SYSTEM DESIGN</td>
<td></td>
<td>2</td>
</tr>
<tr>
<td>1.3.1 LISP DRIVER</td>
<td></td>
<td>2</td>
</tr>
<tr>
<td>1.3.2 INITIALIZER DRIVER</td>
<td></td>
<td>2</td>
</tr>
<tr>
<td>1.3.3 READ MODULE</td>
<td></td>
<td>3</td>
</tr>
<tr>
<td>1.3.4 SCANNER DRIVER</td>
<td></td>
<td>4</td>
</tr>
<tr>
<td>1.3.5 INTERPRETER DRIVER</td>
<td></td>
<td>4</td>
</tr>
<tr>
<td>1.4 MODULE CONSTRUCTION AND INTERACTION</td>
<td></td>
<td>4</td>
</tr>
<tr>
<td>1.5 INTERDATA 8/32</td>
<td></td>
<td>5</td>
</tr>
<tr>
<td>1.6 SUMMARY</td>
<td></td>
<td>5</td>
</tr>
<tr>
<td><strong>CHAPTER TWO</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>READING THE USER'S PROGRAM</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2.1 INTRODUCTION</td>
<td></td>
<td>7</td>
</tr>
<tr>
<td>2.2 INREAD SPECIFICS</td>
<td></td>
<td>7</td>
</tr>
<tr>
<td>2.2.1 PARAMETERS</td>
<td></td>
<td>7</td>
</tr>
<tr>
<td>2.2.2 INPUT</td>
<td></td>
<td>8</td>
</tr>
<tr>
<td>2.2.3 OUTPUT</td>
<td></td>
<td>9</td>
</tr>
<tr>
<td>Section Name</td>
<td>Chapter Three Scanner</td>
<td>Page</td>
</tr>
<tr>
<td>--------------</td>
<td>-----------------------</td>
<td>------</td>
</tr>
<tr>
<td>3.1 INTRODUCTION</td>
<td></td>
<td>11</td>
</tr>
<tr>
<td>3.1.1 GRAMMAR</td>
<td></td>
<td>11</td>
</tr>
<tr>
<td>3.1.2 OVERVIEW</td>
<td></td>
<td>13</td>
</tr>
<tr>
<td>3.2 ISCANN</td>
<td></td>
<td>13</td>
</tr>
<tr>
<td>3.3 ISTKGN</td>
<td></td>
<td>14</td>
</tr>
<tr>
<td>3.3.1 PARAMETERS</td>
<td></td>
<td>15</td>
</tr>
<tr>
<td>3.3.2 INPUT</td>
<td></td>
<td>16</td>
</tr>
<tr>
<td>3.3.3 OUTPUT</td>
<td></td>
<td>16</td>
</tr>
<tr>
<td>3.3.4 MODULAR ORGANIZATION</td>
<td></td>
<td>16</td>
</tr>
<tr>
<td>3.3.5 OVERVIEW</td>
<td></td>
<td>17</td>
</tr>
<tr>
<td>3.4 INUMBR</td>
<td></td>
<td>19</td>
</tr>
<tr>
<td>3.4.1 INPUT</td>
<td></td>
<td>20</td>
</tr>
<tr>
<td>3.4.2 OUTPUT</td>
<td></td>
<td>21</td>
</tr>
<tr>
<td>3.5 IADDRS</td>
<td></td>
<td>21</td>
</tr>
<tr>
<td>3.5.1 INPUT</td>
<td></td>
<td>21</td>
</tr>
<tr>
<td>3.5.2 OUTPUT</td>
<td></td>
<td>21</td>
</tr>
<tr>
<td>3.6 CKSFCE</td>
<td></td>
<td>22</td>
</tr>
<tr>
<td>3.7 ISINIT</td>
<td></td>
<td>22</td>
</tr>
<tr>
<td>3.8 ISSMTE</td>
<td></td>
<td>23</td>
</tr>
<tr>
<td>3.8.1 INPUT</td>
<td></td>
<td>24</td>
</tr>
<tr>
<td>3.8.2 OUTPUT</td>
<td></td>
<td>24</td>
</tr>
<tr>
<td>3.9 ISYMCK</td>
<td></td>
<td>25</td>
</tr>
<tr>
<td>3.9.1 INPUT</td>
<td></td>
<td>25</td>
</tr>
<tr>
<td>3.9.2 OUTPUT</td>
<td></td>
<td>26</td>
</tr>
<tr>
<td>3.9.3 OVERVIEW</td>
<td></td>
<td>26</td>
</tr>
<tr>
<td>SECTION NAME</td>
<td>CHAPTER FOUR</td>
<td>PAGE</td>
</tr>
<tr>
<td>--------------</td>
<td>--------------</td>
<td>------</td>
</tr>
<tr>
<td>4.1 INTRODUCTION</td>
<td>FARSER</td>
<td>28</td>
</tr>
<tr>
<td>4.1.1 OVERVIEW</td>
<td></td>
<td>28</td>
</tr>
<tr>
<td>4.1.2 DRIVER</td>
<td></td>
<td>30</td>
</tr>
<tr>
<td>4.2 ICTREE</td>
<td></td>
<td>32</td>
</tr>
<tr>
<td>4.2.1 INPUT</td>
<td></td>
<td>32</td>
</tr>
<tr>
<td>4.2.2 OUTPUT</td>
<td></td>
<td>33</td>
</tr>
<tr>
<td>4.2.3 OVERVIEW</td>
<td></td>
<td>33</td>
</tr>
<tr>
<td>4.3 VARBLE</td>
<td></td>
<td>34</td>
</tr>
<tr>
<td>4.3.1 PARAMETERS PASSED</td>
<td></td>
<td>35</td>
</tr>
<tr>
<td>4.3.2 OVERVIEW</td>
<td></td>
<td>35</td>
</tr>
<tr>
<td>4.4 ILPARN</td>
<td></td>
<td>36</td>
</tr>
<tr>
<td>4.5 ILPARN AND IPERCD</td>
<td></td>
<td>37</td>
</tr>
<tr>
<td>4.6 LEARN AND NXTFAR</td>
<td></td>
<td>37</td>
</tr>
<tr>
<td>4.6.1 OVERVIEW</td>
<td></td>
<td>38</td>
</tr>
<tr>
<td>4.7 LEFS</td>
<td></td>
<td>38</td>
</tr>
<tr>
<td>4.7.1 INPUT</td>
<td></td>
<td>39</td>
</tr>
<tr>
<td>4.7.2 OUTPUT</td>
<td></td>
<td>39</td>
</tr>
<tr>
<td>4.7.3 OVERVIEW</td>
<td></td>
<td>39</td>
</tr>
<tr>
<td>4.8 LELTR AND IPROCL</td>
<td></td>
<td>40</td>
</tr>
</tbody>
</table>

**CHAPTER FIVE**

INTERDATA 8/32 INFORMATION

<table>
<thead>
<tr>
<th>SECTION NAME</th>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1 INTERDATA 8/32</td>
<td>42</td>
</tr>
<tr>
<td>5.2 HARDWARE</td>
<td>42</td>
</tr>
</tbody>
</table>
CHAPTER FIVE

5.3 SOFTWARE .................................. 43
  5.3.1 COMMON DATA ......................... 43
  5.3.2 DOUBLE PRECISION .................... 44
  5.3.3 JOB CONTROL LANGUAGE ............... 45
  5.3.4 DOWNTIME ............................ 46

5.4 RUNTIME JCI ................................ 46

CHAPTER SIX
TESTING

6.1 INTRODUCTION ............................. 48
6.2 LEVEL OF TESTING ....................... 48
6.3 TEST PROGRAM ............................ 48
6.4 TOKENS .................................. 49
6.5 NCODES .................................. 50

APPENDICES

APPENDIX A .................................... 51
APPENDIX B .................................... 52
APPENDIX C .................................... 54
APPENDIX D .................................... 58
APPENDIX E .................................... 81
APPENDIX F .................................... 112
APPENDIX G .................................... 113
ILLUSTRATIONS

FIGURE 1-1 LISP INTERPRETER SYSTEM DESIGN .................. 3
FIGURE 2-1 READ MODULE RELATIONSHIP .......................... 8
FIGURE 3-1 SCANNER MODULES RELATIONSHIP .................... 14
FIGURE 3-2 INITIAL TOKEN FORMAT .............................. 15
FIGURE 3-3 ISTKGN MODULAR RELATIONSHIP ...................... 17
FIGURE 3-4 SCANNER HIGH-LEVEL ALGORITHM .................... 18
FIGURE 3-5 EXPANDED TOKEN FORMAT ............................ 24
FIGURE 4-1 INTERPRETER MODULAR RELATIONSHIP ............... 30
FIGURE 4-2 GENERAL TREE REPRESENTATION ..................... 31
FIGURE 4-3 HIGH-LEVEL PARSE ALGORITHM....................... 32
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.

1.3.3 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 APPENDIX 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

LISP
INTERPRETER
DRIVER

INREAD

ISINIT

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.
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 **ISTKN** 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><ATCM PART>`
5. `<ATOMIC SYMBOI NUMERIC> ::= <ATOMIC SYMBOL INTEGER> | <ATOMIC SYMBOI FLOAT>`
6. `<ATOMIC SYMBOI INTEGER> ::= <NUMBER> | <NUMBER><ATOMIC SYMBOL INTEGER>`
(7) `<ATOMIC SYMBOL FLOAT>` ::= `<ATOMIC SYMBOL INTEGER>` |
    `<ATOMIC SYMBOL INTEGER>` . `<ATOMIC SYMBOL INTEGER>` |
    0 . `<ATOMIC SYMBOL INTEGER>`
(8) `<ATOM PART>` ::= `<NULL>` | `<NUMBER>` `<ATOM PART>` |
    `<LETTER>` `<ATOM PART>`
(9) `<S-EXPRESSION>` ::= `<ATOMIC SYMBOL NON-NUMERIC>` |
    `<ATOMIC SYMBOL NUMERIC>` |
    `<LIST>`
(10) `<LIST>` ::= `<S-EXPRESSION>` |
    `<S-EXPRESSION>` `<SEPARATORS>` `<LIST>`
(11) `<SEPARATORS>` ::= `<BLANK>` | `<BLANK .>`

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.²

² Ibid., p. 8.
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 ISSMTE 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 keyword 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 ISN module's code is located in APPENDIX D.

SCANNER MODULES RELATIONSHIP

![Diagram of 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 existence 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 encountered; and (3) if a null s-expression is encountered.

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

INITIAL TOKEN FORMAT

| BEGINNING ADDRESS | LENGTH |

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 INUMER

The INUMER 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 tokens 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 NUMCHB 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 (NUMCHB). 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) IPLOCK, 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 ISSMTE 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

ISSMIB 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.
3.9 **ISYMCK**

ISYMCK is called by ISSMTB. It takes the data passed from ISSMTB and conducts a table search of the appropriate key word table to see if a match of data occurs. The purpose of this routine is to provide to ISSMTB the table number and table location of a string. If the string is not found, zeroes are returned indicating a program unique string. The code for ISYMCK is in APPENDIX D.

3.9.1 **INPUT**

The input data, I and ILNGTH, are used to load a dummy array (ICCNVT) with the string being processed. However, if the length is greater than 12, a return to the calling routine is made. This is because none of the key words are greater than twelve. Because FORTRAN only allows eight characters to be stored in a double word, all words of length nine through twelve are truncated to eight. All key words of length eight to twelve are in the same table. This causes no great concern because the eight character key words and the truncated words are unique. The one caution to be made is to the user. He or she must be aware of this peculiarity to preclude the use of a character string which would match a truncated word. The strings in question are DIFFEREN, EVALQUOT, INTERSEC, LEFTSHIP, REMAINDE, FACTORIA, and UNSPECIA.
3.9.2 OUTPUT

The output provided by ISYMCK consists of the table in which a string is found (ITABLE), the location within the table (ILOCAT), or 0 for ITABLE and ILOCAT if the string is either too long or not found.

3.9.3 OVERVIEW

The user's program is stored one character to a full word. This was done for ease of processing in other modules. In this module the data in up to eight full words must be compacted into one double word. This is accomplished through a core-to-core read.

For convenience the string data is read into an eight element array. A process called ENCODE is then performed. ENCODE causes the string to be read and changes the format from A4 to A1. Thus, only the first byte of each word is read. However, the INTERDATA routine requires a block of 132 bytes to use the ENCODE function. The format shows the 132 bytes, but only the first eight are used in the comparison.

To compact the first eight characters into one double word the INTERDATA process called DECODE is performed. By writing the first eight bytes of the ENCODED block into a double word (ICHECK), using a format of A8, the string is then in the same format as the strings found in the keyword tables.

After this is accomplished, a normal table lookup procedure is followed. Each table is of the same length,
and as such some tables have empty elements. To preclude having to search all elements in a table, all empty elements are filled with a sentinel represented by '<<'. If the sentinel is encountered before a match is found, a return to the calling routine is made. If not, IFOUND, is set to 1 and a return is made.
Chapter 4

PARSER

4.1 INTRODUCTION

This chapter presents information concerning the implementation of the INTERPRETER modules. The INTERPRETER modules are designed to: (1) construct nodes to provide a general tree representation of the user's program; (2) execute the user's program.

This project is only concerned with the first case and will not address the execution phase at all. The general tree construction routines require the interaction of eleven modules. The modules are called depending on the string being processed. The modular relationship of the INTERPRETER is shown in Figure 4-1.

4.1.1 OVERVIEW

The INTERPRETER modules perform a parse of the user's program using the expanded tokens generated in ISSMIB. The parse consists of looking at the token representations and creating a general tree. The nodes used for the tree are discussed in section 4.2 below. The creation of a general tree representation of a short program is shown in Figure 4-2.

The figure shows the relationship between the nodes of the tree. Each node consists of sixteen bytes. The sixteen bytes allowed are divided, from left to right as follows:
<table>
<thead>
<tr>
<th>FIELD</th>
<th>LENGTH</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>4</td>
<td>LOCATION</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>LENGTH</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>SON</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>BROTHER</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>MARK</td>
</tr>
</tbody>
</table>

The **LOCATION** field contains the logical address in storage of the expanded token being represented by the node. The **LENGTH** field contains the number of expanded tokens represented by the node, if the node LOCATION field points to a left parenthesis expanded token. If it does not, the **LENGTH** field is zero as the node represents only one expanded token. The **SON** field points to the next level subordinate node. The **BROTHER** field points to a node of equal status at the same level. The **MARK** field is designed to be used in testing and future applications. It has no current significance and is set to 2. For example, in Figure 4-2, **CAR** has the one argument (A B). The son field of the **CAR** node contains a 2, indicating that the argument is found in node 2. Node 2 represents a string starting with the expanded token array element 162 and has a length of 4 (from the **LENGTH** field). Node 2 is not terminal since it has a pointer in its **SON** field. Following the pointer, it can be seen that node 3 (A) is a son of node 2. Node 3 represents a terminal node because its **SON** field is null. Its brother field contains a pointer to B. The tree representation is then complete.

To create this type of general tree requires the use of the high-level algorithm shown in Figure 4-3. Note that the entire algorithm is based on processing parentheses or strings. The separators have been eliminated during the
scanning process.

**INTERPRETER MODULAR RELATIONSHIP**

![Diagram](Figure 4-1)

4.1.2 DRIVER

The driver module for the interpreter modules is called by the LISP INTERPRETER DRIVER. It in turn calls the module IGTREE which generates a double word array of tokens representing a general tree. Eventually, the driver will also call the necessary module to cause the execution of the
program.

The driver module receives the beginning array address and the ending array address of the tokens generated by ISCAN. It will pass on to the executor the beginning and ending addresses of the nodes representing the general tree.

GENERAL TREE REPRESENTATION

PROGRAM: CAR (A B)

EXPANDED TOKENS:

<table>
<thead>
<tr>
<th>STRING</th>
<th>TOKEN</th>
<th>ARRAY ELEMENT</th>
</tr>
</thead>
<tbody>
<tr>
<td>CAR</td>
<td>1030130</td>
<td>161</td>
</tr>
<tr>
<td>(</td>
<td>5010110</td>
<td>162</td>
</tr>
<tr>
<td>A</td>
<td>6010000</td>
<td>163</td>
</tr>
<tr>
<td>B</td>
<td>8010000</td>
<td>164</td>
</tr>
<tr>
<td>)</td>
<td>9010210</td>
<td>165</td>
</tr>
</tbody>
</table>

GENERAL TREE NODES:

<table>
<thead>
<tr>
<th>NODE</th>
<th>NODE NUMBER</th>
</tr>
</thead>
<tbody>
<tr>
<td>1610000000020002</td>
<td>1</td>
</tr>
<tr>
<td>162000400030002</td>
<td>2</td>
</tr>
<tr>
<td>1630000000000042</td>
<td>3</td>
</tr>
<tr>
<td>1640000000000002</td>
<td>4</td>
</tr>
</tbody>
</table>

TREE STRUCTURE

```
          161
           |
          162
           |
          163
           |
          164
```

Figure 4-2
HIGH-LEVEL PARSE ALGORITHM

DO WHILE ALL TOKENS NOT PROCESSED;

   GET NEXT NODE

CASE 1  INITIAL NODE
   GET FIRST TOKEN;
   PROCESS INTO NODE;
   PROCESSES ALL SAME LEVEL NODES;

CASE 2  NEXT NODE ON STACK
   IF TERMINAL
      THEN: GET NEXT NODE
   ELSE: PROCESS SON INTO NODE

Figure 4-3

4.2  IGTREE

The general tree routine is designed to produce an array of double words which represent a LISP program. Each double word contains information fields which provide the data necessary to traverse the general tree.

The IGTREE module acts as a control routine and calls the appropriate subroutine based on the character being processed. The code for IGTREE is in APPENDIX E.

4.2.1  INPUT

The input required to successfully execute this module
are: (1) IJUMP, the beginning array element of the expanded tokens being processed; (2) ISTART, the last array element containing the expanded tokens being processed; (3) the tokens stored in ISPACE; (4) the user's program stored in ISPACE; (5) the double word array to be filled by nodes; and (6) additional space in ISPACE to create a stack.

The space required in the double word array is similar to the space used to store the user's program. It must be contiguous and must remain in storage until the execution of the user's program is complete.

4.2.2 OUTPUT

IGTREE will output the beginning and ending array element containing the nodes constructed to represent the user's program. The nodes are stored in a double word array and will remain in the array until execution is complete. This allows the execution modules to access them as required.

4.2.3 OVERVIEW

IGTREE begins the processing by extracting the ISPACE array element address of the first symbol in the user's program from the first token. It then determines if the character at that address is a character, left parenthesis, right parenthesis, or a period. If it is a right parenthesis or a period, a fatal error message is returned and the program is terminated.
If the item is a character, the routine VARBLE is called to process it. If it is a left parenthesis, the routine ILPARN is called to process it. The continual look at a character and process it is a 'do while' operation. As FORTRAN does not have a 'do while' statement, a continuous loop GO TO is used with the exit parameter being the processing of all nodes.

When VARBLE or ILPARN is called, a new node is constructed. The pointer NXTPT will always point to a node that requires processing until all nodes are terminal. Thus, the node to be processed is determined by extracting the son from the next node to be processed. The son field is the only field that contains a temporary address. Until the SON field is set to point to the double word array element representing the son, it is set to point to the ISPACE element representing the token of the first character of the SON. The first character in conjunction with the length acquired from the temporary stack provides the necessary data to continue processing.

4.3 VARBLE

The VARBLE routine is designed to process strings. It accomplishes its function by: (1) placing the token address within ISPACE in the location field; (2) if the string is followed by a left parenthesis, place the token address within ISPACE in the SON field; (3) if the string is followed by a left parenthesis, call a routine to find the rightmost same level parenthesis; and (4) if the entire
program has not been processed, continue steps 1-3 until complete.

By following the above procedures, the first pass of the user's program will produce all top level strings. These top level strings will point to each other in a hierarchy of first found to second and so forth until n-1 points to n. The pointing at the same level is done by filling the brother field of a node. Each top level node will contain in its son field the token address of its next level string.

4.3.1 PARAMETERS PASSED

The parameters passed by VARBIE are: (1) I, the token storage array element being processed; (2) IFINIT, the ending address of the token array; (3) NEXPT, the pointer to the next node to be processed; (4) IDSTA, a pointer to the first node of the general tree; (5) INEDD, the pointer to the last node of the general tree; (6) ISTART, a pointer to the last available space of a temporary stack; (7) IFARPT, a pointer to the temporary stack which contains the length of the next node to be processed; and (8) ICODE, a code which precludes double processing.

4.3.2 OVERVIEW

VARBIE is called from several routines and is called to process strings. Different treatment is required if the string is the initial item processed, or if the string
occurs after a set of parentheses. Thus, early in the routine there are conditional statements which cause branching to the appropriate part of the routine.

The node is initialized by multiplying the token's address by 1000000000000.DO and placing the result in a double word array element obtained from memory management. Note that INTERDATA requires that a number as large as the node be handled as a double precision number. If the .DO is not placed at the end of the number, there will occur an overflow loss of data and the results stored will not be precise.

Once the basic node is created, tests are made to see if it has a son and/or brother. If so, the node is updated to point to the son token's and/or brother token's array address. Processing is then continued. When a son's address is placed in the parent node, its length is also stored on a temporary stack. The combination of the address and length will facilitate future processing.

4.4

ILPARN

ILPARN is a driver module. It performs the basic function of calling the appropriate routine to process a left parenthesis. If the left parenthesis is the first token being processed, LPFST is called. If the left parenthesis is encountered later in the program, LPLTR is called.
4.5 **IFFARN AND IFERCD**

IFFARN and IFERCD are error routines. If a right parenthesis is encountered for processing, an error is detected. This is due to the way in which parentheses are processed. When a node contains a left parenthesis, the son of that node is assigned the length of the node minus two. Using this approach, matched right parentheses are disposed of at the same time as the left parenthesis.

If a period is encountered for processing, an error is again apparent. This is because period separators are skipped during processing just like spaces.

Both modules print an appropriate error and return a fatal error code which causes program termination.

4.6 **LPARN AND NXTPAR**

LPARN is designed to find the furtherest same-level right parenthesis such that the token after the right parenthesis is not a same-level left parenthesis. This finds all the sub-expressions of an S-expression. It accomplishes this mission with the aid of routine NXTPAR. When a same-level right parenthesis is found NXTPAR is called. NXTPAR then looks ahead to the next character. If it is a same-level left parenthesis, processing is continued until another same-level right level parenthesis is found.

The code for LPARN and NXTPAR is found in APPENDIX E.
4.6.1 OVERVIEW

LPARN is called when a left parenthesis has been found. By the time LPARN is called a parent node for the left parenthesis has already been created. The routine then assigns the level of the left parenthesis to a variable called LEVEL. It then enters a loop to look at each character, in order, to find the right parenthesis with the same level number.

Once a right parenthesis with the same level number is found, N XPAR is called. N XPAR is designed to look ahead one character. If the look ahead causes the pointer IFINIT, the pointer to the last character, to be exceeded, a return is made immediately. If the next character is a left parenthesis, control is returned to LPARN and processing continues to find the appropriate right parenthesis. If not, a return to LPARN is made with a code of 1 meaning the next character is not a left parenthesis.

4.7 LIFST

LIFST is called if the first expanded token encountered by IGTK is a left parenthesis. Its purpose is to create double word nodes representing the top level of the user's program and then to return control to IGTK for further processing.
4.7.1 INPUT

The input to LPFST consists of: (1) I, the address in the expanded token array of the first character/string to be processed; (2) IFINIT, the address in the expanded token array of the last character/string of the user's program. The input is used to establish the bounds of a do loop to process the program and create general tree nodes.

4.7.2 OUTPUT

The output from LPFST consists of: (1) IFARET, the pointer to the next element on the length temporary stack; (2) ISETD, the last double word array element obtained from memory management; (3) IDSTA, the first double word array element obtained from memory management; (4) ISTABT, the last element of the length temporary stack; and (5) NXTPT, a pointer to the next node to be processed. The output parameters allow IGTREE to continue to process the nodes generated without double processing a node or asking for an element address that does not represent a node.

4.7.3 OVERVIEW

LPFST processes the initial level of nodes, if the first expanded token representing the user's program is a left parenthesis. It performs its functions by getting the necessary double word space from memory management and filling the space with node representations. The nodes are generated in the format discussed in IGTREE and portrayed in
Figure 4-2.

The code for LPPST is in APPENDIX E.

4.8 LFLTR AND IPROCL

LFLTR is designed to process S-expressions found between two same level parentheses. This entails creating a son node and putting a pointer to that node in the parent's SON field.

The length of the new parent node representation is retrieved from the length temporary stack. It must be remembered that LFLTR is only called when a left parenthesis is found as the first character of a SON field. Therefore, a loop is entered to look at all characters between the first left parenthesis and the last character in the length. When the same level right parenthesis is found IPROCL is called. IPROCL performs a similar function to NXTPAR. However, it is more powerful.

IPROCL first looks at the code passed. If the code is equal to 1, a brother is being processed. The brother field is set to point to the next node.

If the code is equal to 0, the parent node's son field is set to point to the next node to be acquired from memory management.

After acquisition, the creation of a parent node is accomplished for the string being processed. Then the length of the string is stacked on the length temporary stack.

After the new parent node is created a one character
look ahead is executed. If the next character exceeds the length of the string, a return is made. If not, and the next character is a left parenthesis, a return is made. If not, and the next character is a variable, the brother field is set and VARELE is called.

The code for LPLTR and IPROCL is in APPENDIX E.
Chapter 5

INTERDATA 8/32 INFORMATION

5.1 INTERDATA 8/32

The INTERDATA 8/32 computer used to test the interpreter is owned by the Computer Science Department and is located in their computer facility. It has 64K bytes of memory and all required peripheral devices to enter, process and output a LISP program.

5.2 HARDWARE

The INTERDATA 8/32 hardware proved to be very reliable during the period of writing and testing this project. This is not to say that no problems occurred. There were faulty memory problems, disk overheat problems, and console problems. However, none of the problems occurred because a high level language LISP interpreter was being run.

One hardware characteristic does affect the manner in which the user's program must be entered in the system. The card reader (Model 080416P, True Data Corporation) does not have a capability for sensing an end of file and causing that status to be relayed. Therefore, the user must read his program into a local file and assign the local file name to the reader logical unit number when he desires to execute his program. If he attempts to assign the card reader to the reader logical unit number, he will receive an error message informing him that the card reader is empty.
5.3 SOFTWARE

The software package supplied with the INTERDATA 8/32 consists of: (1) nonproduction INTERDATA software; and (2) software generated by the personnel in the Computer Science Department. Generally speaking, the software caused the greatest time delays in completing this project. The major sources for the delays were: (1) system crashes caused by the software when legal commands were entered at the terminals; (2) faulty software which caused undecodable error messages; and (3) a FORTRAN compiler that provides only bare essentials and little or no extensions to the FORTRAN language.

Some specific problems/peculiarities that occurred, and the solutions developed or implemented are discussed below.

5.3.1 COMMON DATA.

A problem occurred when certain combinations of letters were used in conjunction with certain specification statements. An example of this type of error was the addition to common of an array called IALIST when IARG had been specified INTEGER*2. The program would compile, but upon execution a task error of 'WRONG PROG' would appear. After several days of debugging and tracing, it was determined that a change of name could cause the error to disappear. When an INTERDATA representative visited the department, the problem was brought to his attention. He
was not aware of any solution to the problem although he did recall that the problem had surfaced at other installations. Because this one 'glitch' was critical, a call was made to the INTERDATA software manager in Chicago. He knew immediately what our problem was and provided a patch solution. He was able to convey, over the phone, that there was erroneous data stored in two addresses of the Task Establisher Task (TET) program. The addresses along with the proper data to be read into those addresses were passed to our software people. The data changes took approximately two minutes. The lesson learned was to contact the software people when a thorough analysis at KSU provided no insight into the problem.

5.3.2 DOUBLE PRECISION

The INTERDATA FORTRAN compiler does not have the power that the IBM compiler has. There are few extensions to the basic FORTRAN requirements of a compiler. A case in point is the treatment of double precision numbers. This project required a set of sixteen consecutive digits to act as node representations of a general tree structure. An integer representation was ruled out due to the limited number of integer digits allowed. A double precision number contains exactly the sixteen digits required. The IBM compiler has an extension to its compiler that treats any numerical operation performed during assignment to a double precision variable as a double precision number. Thus, if IDSPAC were declared as double precision and the assignment
IDSPAC=961*1000000000. were made, IDSPAC would contain 961000000000.. However, INTERDATA does not contain this extension. To obtain the same results from the INTERDATA compiler all numbers in an operation that produce more than ten digits must end in `.DO`. The above example would be entered as 961*1000000000.DO to obtain the same assignment. Failure to do this causes the overflow of the least significant digits to be lost at each step in an operation.

5.3.3 JOB CONTRCL LANGUAGE

The job control language on the INTERDATA does not provide any significant problems. It is not as powerful or as complete as IBM JCL. For example, when executing a FORTRAN program, the user must: (1) assign his object file to the proper logical unit; (2) assign his data to the proper reader logical unit; (3) assign his output to the proper printer logical unit; (4) load his own program for execution; and (5) start the execution. This lack of automatic assignments plus requirements for extensive user interaction in the execution of a program, causes many typographic errors and an increased time to run a program.

A run time comparison of the general tree portion of this project was made. On the IBM computer it took 1 minute and 48 seconds from the time the start button on the card reader was pushed until the printer stopped printing. On the INTERDATA computer, the same program took 4 minutes and 12 seconds. Both runs were made in the early morning hours when little or no other activity was being processed.
5.3.4 DOWN TIME

When working with the INTERDATA 8/32 a considerable amount of frustration is generated as a result of the frequent crashes encountered. Equally as frustrating, is the amount of time the computer is not available for use due to maintenance backlog and environmentally created failures. An example of down time due to maintenance backlog is the period of approximately two weeks when there was a bad piece of memory. The bad memory had to be partitioned out of possible use. The partition size used almost 30% of available space. This caused considerable inconvenience because at the same time a PASCAL package was being tested for delivery. If, during this period, a user wanted to be sure of getting enough space to work when he wanted, he had to use the computer in the late evening and early morning hours. An example of environmental failures is the recurring loss of one disk drive due to the heat in the computer room. The climatic control devices in the computer facility are archaic and do not, during cold periods, keep the room at an appropriate operating temperature.

5.4 RUNTIME JCL

The runtime JCL for this program is in APPENDIX F. PART I shows the JCL to read a deck of cards into a file called INREAD. PART II shows the JCL required to establish the tasks for the program and PART III shows the JCL
required to run the program.

During the completion of this project, the JCL has changed three times. The data in APPENDIX P is the most current. User's will have to verify the JCL prior to use, to insure that no changes have been made.
Chapter 6

TESTING

6.1 INTRODUCTION

The scanner and parser were tested on the INTERDATA 8/32 using test programs with varying levels of difficulty. The simplest test was the input of atom 'A'. A more difficult test is the one found in APPENDIX H which is a factorial program with a trace option. The factorial program allows a good test of most of the subroutines written.

6.2 LEVEL OF TESTING

The factorial test program requires a tree construction with 19 levels. In addition, it provides several branches to brother nodes with subsequent breakdowns. Therefore, the test of this program tests each subroutine except LPFST and LPROCE. LPFST and LPROCE were tested using programs such as (A B) (C D) and (A . B) and were found to function properly.

6.3 TEST PROGRAM

PART I of APPENDIX H shows the program being tested. Note that although it is a relatively short program, it has three nodes at the initial level, DEFINE, TRACE, and FACTORIAL. It also has a parentheses level count that goes as high as 9. Although 9 is the highest level achieved in
this test program, levels are actually limited only by the
user's program. The program and the parentheses levels are
shown in PART II of APPENDIX G to illustrate how the program
is stored in memory. The subroutines used to accomplish
PART I and PART II of the test were INREAD, ISCAN, ISTKGN
and INUMBR.

6.4 TOKENS

The token generations are shown in PART III nad and
PART IV of APPENDIX H. PART III shows the incomplete token
generated by the routines called by ISTKGN. The incomplete
tokens contain only the address within the array ISFACE and
the length of the string. The last two digits contain the
length and the first four contain the address. The first
token represents the string DEFINE which is found in array
element 8 and has a length of 6.

The expanded tokens are found in PART IV. The
expansion is done in module ISSMTB and the modules it calls.
As stated previously, an expanded token contains the keyword
table in which a string is found, and its location within
the table, if applicable. Again
the first token shows that DEFINE is in table number 6
(same as length) at position number 2. The table number is
the second digit from the right and the location is the
third and fourth digit. Position number 1 is not used. It
was originally intended to be used as a mark field. It was
left as such for possible use by the EXECUTER.

Token number 9, N in the program, is a program unique
token as it was not found in a keyword table. The zeroes in position 2 through 4 indicate the program unique string.

A comparison of the extended tokens and the test program shows a one for one match of all strings/symbols. The comparison also shows a correct length applied for each string/symbol as well as a correct array element address. When dots or spaces were used in test programs, the results were also perfect.

6.5 NODES

The test program causes a total of thirty-four nodes to be generated. The node construction has been discussed in Chapter Four. The nodes generated by this program are in PART V of APPENDIX H.

The first three nodes represent strings which are at the highest level. These strings, as discussed before, are DEFINE, TRACE, and FACTORIAL. DEFINE is represented by node 1 and points to the level 1 parentheses subordinate to it, and also points to TRACE as a brother node. The TRACE node points to the level 1 parentheses subordinate to it, and also to FACTORIAL as a brother node. FACTORIAL has no additional brother so it only points to the level 1 parentheses subordinate to it. This type of continual breakdown is repeated until the entire program is processed and represented by a general tree.
APPENDIX A


APPENDIX B

***********************
THIS ROUTINE IS A DRIVER ROUTINE TO CALL THE READ ROUTINE. THE READ ROUTINE WILL READ A DECK OF CARDS REPRESENTING A LISP PROGRAM. AT THE SAME TIME IT WILL LEAVE SPACE TO NUMBER ALL PARENTHESES ENCOUNTERED.

***********************
THIS DRIVER WAS WRITTEN BY:

**********************************************************
* *
* DAVID C. BOSSELMAN *
* OCTOBER 1976 *
*
**********************************************************

INTEGER*2 IARG
DOUBLE PRECISION FUNCT, IDSPAC
COMMON/TABLE/FUNCT(8,24), IARG(6,24)
COMMON/SFACE/IPTABEL(50), ISPACE(2000)
1,IPUSED,IFREE,IPSIZE,ISIZE
COMMON/DCUBLE/IDSPAC(500)
CALL INREAD(IFIRST,IDONE,ISTRT)

*******************************
AFTER THE READ ROUTINE HAS COMPLETED ITS FUNCTION, CONTROL IS RETURNED TO THIS PROGRAM. UPON RETURN ALL AVAILABLE PROGRAM DATA HAVE BEEN STORED IN MEMORY. AT THIS TIME THE DRIVER ROUTINE CALLS ANOTHER DRIVER ROUTINE CALLED ISCAN. ISCAN WILL CALL THE NECESSARY
MODULES TO SCAN THE DATA STORED IN MEMORY.

*****************************************************************************

CALL ISCAN(IPIRST,IDONE,IJUMP,ISTART,IFINSH)
CALL INTERP(IJUMP,IFINSH,IDSTA,IDSTOP,ISTART)
STOP
END
APPENDIX C

SUBROUTINE INREAD (IFIRST, IDONE, ISTART)

*****************************************************************************
THIS ROUTINE IS DESIGNED TO READ A LISP PROGRAM TO BE 
PROCESSED. THE READING OF DATA IS ACCOMPLISHED IN THE 
FOLLOWING MANNER:
1. THE ISINIT ROUTINE IS CALLED TO OBTAIN SPACE TO 
READ 
A CARD.
2. A CARD IS READ INTO THE SPACE ACQUIRED.
3. THE ISINIT ROUTINE IS CALLED TO OBTAIN SPACE TO 
NUMBER ALL PARENTHESIS.
4. CONTINUE THE STEPS 1-3 UNTIL ALL CARDS ARE READ.
*****************************************************************************

THE INREAD ROUTINE WAS WRITTEN BY:

*****************************************************************************
*
* DAVID C. BOSSERMAN *
* OCTOBER 1976 *
*
*****************************************************************************

*****************************************************************************  *****
SPECIFIC INREAD DATA ARE AS FOLLOWS:
FUNCTION NAME - INREAD
DOMAIN:
(I) INPUT - THE USER'S LISP PROGRAM.
(II) GLOBAL DATA - PORTION OF AN ARRAY CALLED
ISPACE

FROM MEMORY MANAGEMENT.

BANGE:

1. THE USER'S SOURCE PROGRAM STORED IN ISPACE.
2. A PLACE TO ENTER PARENTHESES NUMBERS IN ISPACE.
3. THE START ARRAY SUBSCRIPT WHICH PERTAINS TO THE
   USER'S PROGRAM.
4. THE END ARRAY SUBSCRIPT WHICH PERTAINS TO THE
   USER'S PROGRAM.

RULE-OF-CORRESPONDENCE:
INREAD CAUSES THE USER'S PROGRAM TO BE PLACED IN THE
SPACE ALLOCATED BY THE MEMORY MANAGEMENT ROUTINES. IT
ALSO RESERVES A PORTION OF THE PROGRAM STORAGE AREA TO
BE USED FOR PARENTHESES NUMBERING.

*******************************************************************************

INTEGER*2 IARG
DOUELE PRECISION FUNCT
COMMON/TABLE/FUNCT(8,24),IARG(8,24)
COMMON/SPACE/IPTABL(50),ISPACE(2000)
1,IPUSED,IFREE,IPSIZE,ISIZE

*******************************************************************************

THE SPACE ROUTINE IS CALLED. NUMBDG REPRESENTS THE
NUMBER OF SPACES THAT THIS ROUTINE REQUIRES FROM
MEMORY MANAGEMENT AT A TIME. ISTART IS THE ADDRESS IN
THE MEMORY THAT DATA IS BEING READ INTO. ONLY 80
WORDS ARE PASSED AT A TIME (ONE CARD LENGTH) SO THE
ISTOP PARAMETER REPRESENTS THE STOP POINT IN MEMORY.
THE SAME WILL HOLD TRUE LATER WHEN NUMBERS ARE
ASSOCIATED WITH THE PARENTHESES. IBLOCK IS A CODE
REPRESENTATION TO MEMORY MANAGEMENT INFORMING IT THAT A CONTIGUOUS BLOCK OF MEMORY IS REQUIRED. THEREFORE, NO HEADER COUNTS WILL BE WRITTEN UNTIL AN IBLOCK=1 IS SENT.

********************************************************************************

N=0
IPRST=0
ISTART=0
ISTCP=0
NUMRQD=80
IBLOCK=0

********************************************************************************

A ROUTINE CALLED ISINIT IS CALLED TO ACQUIRE SPACE FROM MEMORY MANAGEMENT. THE ISINIT ROUTINE IS THE ONLY DIRECT LINK WITH MEMORY MANAGEMENT FROM THIS ROUTINE.

********************************************************************************

10 CALL ISINIT(NUMRQD,ISTART,IELOCK)

********************************************************************************

A VARIABLE CALLED IFIRST IS INITIALIZED WITH THE FIRST ISTART PASSED. IFIRST WILL ALLOW SUBSEQUENT ROUTINES TO BEGIN AT THE FIRST MEMORY STORAGE LOCATION. AN IF STATEMENT IS INCLUDED TO PRECLUDE REINITIALIZATION OF IFIRST.

********************************************************************************

IF(IPRST.EQ.0) IFIRST=ISTART
ISTCP=(ISTART+NUMRQD)-1

********************************************************************************

A CARD IS READ.

********************************************************************************
READ(5,500,END=505) (ISPACE(II),II=ISTART,ISTOP)

500 FORMAT(80A1)

******************************************************************************
ISINIT IS CALLED TO GET AN ADDITIONAL 80 WORDS OF SPACE. THIS SPACE WILL BE
USED EVENTUALLY TO NUMBER THE PROGRAM'S PARENTHESES. CURRENTLY A LOOP IS ENTERED
TO SET ALL VALUES TO SPACES.
******************************************************************************

CALL ISINIT(NUMRQD,ISTART,IELOCK)

ISTOP=(ISTAFT+NUMRQD)-1

DO 600 I=ISTART,ISTOP
    ISPACE(I)=''

600 CONTINUE

GO TO 10

505 ISTCP=ISTOP-80

IDONE=ISTOP
NUMRQD=-80
IBLOCK=1

CALL ISINIT(NUMRQD,ISTART,IELOCK)

ISTART=ISTART-1

RETURN

END
APPENDIX D

SUBROUTINE
ISCAN (IFIRST, IDONE, IJUMP, ISTART, IFINSH)

******************************************************************************

THIS MODULE IS A DRIVER MODULE FOR TWO SEPARATE MODULES
AS FOLLOWS:

1. ISTKGN - THE SCANNER TOKEN GENERATOR MODULE IS
   DESIGNED TO SCAN THE DATA INPUT BY THE
   INREAD ROUTINE. IT THEN GENERATES
   APPROPRIATE TOKENS REPRESENTING THE LISP
   PROGRAM.

2. ISSMTE - THE SCANNER SYMBOL TABLE MODULE IS DESIGNED
   TO ACCEPT INPUT FROM ISTKGN AND PROCESS
   IT. PROCESSING CONSISTS OF SEARCHING THE
   COMMON KEY WORD TABLES IN AN ATTEMPT TO
   FIND THE STRING REPRESENTED BY THE TOKEN.
   IF IT IS FOUND THE TABLE NUMBER AND
   LOCATION WITHIN THE TABLE ARE ADDED TO THE
   TOKEN. IF NOT, ZEROS ARE ADDED TO THE
   TOKEN TO SHOW THAT THE TOKEN POINTS TO A
   PROGRAM UNIQUE STRING.

******************************************************************************

******************************************************************************

THE SCANNER MODULE WAS WRITTEN AS A DRIVER ROUTINE BY:

******************************************************************************

******************************************************************************

* DAVID C. BOSSERMAN *
* OCTOBER 1976 *

***************

SPECIFIC SCANNER DATA ARE AS FOLLOWS:

FUNCTION NAME - ISCAN

DOMAIN:

(I) INPUT - THE ARRAY CALLED ISPACE FROM INREAD.
THE BEGINNING ADDRESS OF THE USER'S
PROGRAM.
THE ENDING ADDRESS OF THE USER'S
PROGRAM.

(II) GLOBAL DATA:
1. FUNCTION NAMES USED IN LISP STORED AS
COMMON DATA.
2. KEY WORDS USED IN LISP - STORED AS
COMMON DATA.

RANGE: A TOKEN ARRAY REPRESENTING THE SOURCE PROGRAM.

RULE-CF-CORRESPONDENCE:
ISCAN READS THE DATA READ INTO STORAGE BY INREAD AND
MATCHES:
1. EACH PARENTHESIS TO ITS PROPER LEVEL NUMBER.
2. EACH KEY WORD OR FUNCTION NAME TO A STORED COMMON
SYMBOL TABLE.
IT THEN PASSES THE TOKENS GENERATED TO THE INTERPRETER FOR
PARSING.

CALL ISTKGN(IFIRST,IDONE,IJUMP,ISTART)
CALL ISSMTE(IJUMP,ISTART,IFINSH)
SUBROUTINE ISTKGN (IFIRST, IDONE, IJUMP, ISTART)

*******************************************************************************

THIS MODULE PERFORMS THE FUNCTION OF READING A USER'S PROGRAM CHARACTER BY CHARACT

1. NUMBERING EACH PARENTHESIS.

2. IDENTIFYING DELIMITERS.

3. PROVIDING THE START POINT AND LENGTH OF EACH STRING IN THE USER'S PROGRAM.

*******************************************************************************

THE TOKEN GENERATION MODULE WAS WRITTEN BY:

*******************************************************************************

*   *

* DAVID C. ERSERMAN *

* OCTOBER 1976 *

*   *

*******************************************************************************

SPECIFIC TOKEN GENERATION DATA ARE AS FOLLOWS:

FUNCTION NAME = ISTKGN C DOMAX:

(I) INPUT - THE ARRAY CALLED ISPACE.

THE BEGINNING ADDRESS OF THE USER'S PROGRAM.

THE ENDING ADDRESS OF THE USER'S PROGRAM.

THE ADDRESS OBTAINED FROM ISINIT TO PLACE ADDRESSES AND LENGTHS OF STRINGS.

(II) GLOBAL DATA:

THE DELIMITERS FOR LISP PROGRAMS.

RANGE: AN ARRAY OF STRING ADDRESSES AND LENGTHS TO BE USED
BY THE ISSMTB MODULE. THE FORMAT OF THE ADDRESS ARRAY ELEMENT IS:

*******************************************************************************
*                             *
* BEGINNING ADDRESS LENGTH    *
*                             *
*******************************************************************************

RULE-OF-CORRESPONDENCE:

CONVERT THE USER'S PROGRAM INTO AN ARRAY WHICH CONTAINS THE LOCATION AND THE LENGTH OF ALL STRINGS IN THE PROGRAM.

*******************************************************************************

A DC LOOP IS ENTERED TO PROCESS EACH CHARACTER. THE PROCESSING STARTS AT ISTART AND ENDS AT IDONE. NOTE THAT THE VARIABLE IJUMP IS INITIALIZED WITH THE FIRST ARRAY SUBSCRIPT TO HELP OTHER MODULES FIND THE BEGINNING OF THE ADDRESS RETURNED FROM ISINIT. IT WILL BE PASSED AT THE END OF THE MODULE.

*******************************************************************************

INTEGER*2 IARG

DOUBLE PRECISION FUNCT

COMMON/ TABLE/FUNCT(8,24), IARG(8,24)

COMMON/ SPACE/IPTAIL(50), ISPACE(2000)

1, IPUSED, IPREE, IPSIZE, ISIZE

IPROC=0

IJUMP=0

NUMQD=1

NUMCHR=0

IBLOCK=0
ICOUNT = 0
ISKIP = 0
N = 0
DO 100 I = IFIRST, IDONE
INCR = 1
IDEC = -1
ICOUNT = ICOUNT + 1
IF (ICOUNT .GT. 80) ISKIP = ISKIP + 1
IF (ISKIP .GT. 80) ICOUNT = 1
IF (ISKIP .GT. 80) ISKIP = 0
IF (ICOUNT .GT. 80) GO TO 100

FIRST THE CHARACTER IS CHECKED TO SEE IF IT IS A PARENTHESIS. IF IT IS, A NUMBER IS ASSIGNED TO IT AND IT IS PLACED IN THE APPROPRIATE ARRAY POSITION RESERVED IN INREAD.

IF (ISPACE(I) .EQ. '(') CALL INUMBR(I, INCR, N, IPROCE, NUMCHR, ISTART)
IF (ISPACE(I) .EQ. ')') CALL INUMBR(I, IDEC, N, IPROCE, NUMCHR, ISTART)

A CHECK IS MADE AT THIS POINT TO DETERMINE IF MATCHED PARENTHESIS ARE BEING CONSTRUCTED. IF NOT, AN ERROR MESSAGE IS PRINTED.

ILINE = (I/80) + 1
IPOSIT = I
IF (N .LT. 0) WRITE(6, 10) ILINE, IPOSIT

10 FORMAT (' *' MISMATCHED PARENTHESIS FOUND AT LINE')
1, I3, ' POSITION ', I2, '.')

******************************************************************************
NEXT THE DELIMITERS '.' AND ' ' ARE CHECKED. IF A SPACE IS
ENCOUNTERED, THE COUNTER NUMCHR IS CHECKED TO SEE IF IT IS
GREATER THAN 0. IF SO, A STRING HAS BEEN IDENTIFIED.
ANOTHER CASE IS THE EMPTY EXPRESSION OR NIL. THEREFORE, IF
A SPACE IS ENCOUNTERED, A LOOKBACK AND LOOKFORWARD IS
ACCOMPLISHED TO DETERMINE IF THE SPACE REPRESENTS A NIL.
******************************************************************************

IF(ISPACE(I).EQ.'.' AND NUMCHR.GT.0) GO TO 100
IF(ISPACE(I).EQ.' ' AND NUMCHR.GT.0) CALL IADDR
1(I, NUMCHR, IFRCCE, ISTART)
IF(ISPACE(I).EQ.' ' AND NUMCHR.EQ.0) CALL CKSPCE
1(I, IPROCE, ISTART)
IF(IPROCE.NE.999) NUMCHR=NUMCHR+1
IF(ISPACE(I).EQ.' ') NUMCHR=NUMCHR-1
IF(IPROCE.NE.999) GO TO 100
IF(IJUMP.EQ.0) IJUMP=ISTART
NUMCHR=0
IPROCE=0
100 CONTINUE
******************************************************************************
ISINIT IS CALLED TO RELEASE THE ROUTINES FROM PROVIDING
CONTIGUOUS SPACE.
******************************************************************************
NUMRQD=0
IBLOCK=0
CALL ISINIT(NUMRQD, ISTART, IBLOCK)
RETURN
SUBROUTINE INUMBR(I,INB,N,IPROCE,NUMCHR,ISTART)

******************************************************************************

THIS ROUTINE WAS WRITTEN BY:

******************************************************************************

*                        *
*   DAVID C. BOSSERMAN    *
*   OCTOBER 1976         *
*                        *

******************************************************************************

THE INUMBR ROUTINE IS A UTILITY ROUTINE WHICH KEEPS TRACK OF
THE CURRENT PARENTHESIS LEVEL NUMBER AND UPDATES IT. THE
ROUTINE THEN ASSIGN THE APPROPRIATE NUMBER TO THE SPACE
ALREADY ALLOCATED IN INREAD. WHEN PRINTED, THE NUMBER WILL
APPEAR DIRECTLY BELOW THE APPLICABLE PARENTHESIS. THE FIRST
DETERMINATION TO BE MADE IS WHETHER NUMCHR IS EQUAL TO 0.
IF SO, THE PARENTHESIS STANDS ALONE AND IS PROCESSED
STARTING AT LINE 10 AS A PARENTHESIS. IF NOT, A STRING
PRECEDES THE PARENTHESIS AND THE STRING MUST BE PUT IN TOKEN
FORM BEFORE THE PARENTHESIS IS PROCESSED.

******************************************************************************

INTEGER*2 IARG
DOUBLE PRECISION FUNCT
COMMON/TABLE/FUNCT(8,24),IARG(8,24)
COMMON/SPACE/IPTABL(50),ISPACE(2000)
1,IPUSED,IPFREE,IPSIZE,ISIZE
NUMRQD=1
IBLOCK=0
IF(NUMCHR.EQ.0) GO TO 10
J represents the array element location in ISPACE where the beginning character of a string is found. TKLOC represents the location and the length.

J = (I - NUMCHR) * 100
TKLOC = J + NUMCHR
CALL ISINIT(NUMRD, ISTART, IBLOCK)
ISPACE (ISTART) = TKLOC

INNE is a counter used to keep the parenthesis number correct. Note that a right left parenthesis combination has the same number. Thus, a right parenthesis does not subtract from the number.

10 IF (INB.EQ. 1) INB = N + INE
   IF (INB.EQ. -1) JJ = -1
   IF (INB.EQ. -1) INB = N
   N = INB
   ISPACE(I+80) = INB
   IF (JJ.EQ. -1) N = N - 1
   JJ = 0
   IROCE = 999
   TKLOC = (I*100) + 1
   CALL ISINIT(NUMRD, ISTART, IBLOCK)
   ISPACE (ISTART) = TKLCC
   RETURN
END
SUBROUTINE IADDR(I,NUMCHR,IPROCE,ISTART)

******************************************************************************

THIS ROUTINE WAS WRITTEN BY:

******************************************************************************

*       *
* DAVID C. BOSSEMAN *
*       *
* OCTOBER 1976 *
*       *

******************************************************************************

******************************************************************************

THIS ROUTINE IS WRITTEN TO PROCESS THE DATA TO BE PLACED IN AN ARRAY ELEMENT. THE DATA PROCESSED PROVIDES THE LOCATION WITHIN A PROGRAM OF A STRING AND ITS LENGTH. THIS ROUTINE IS ONLY CALLED WHEN A SPACE OR PERIOD DELIMITER IS ENCOUNTERED AND NUMCHR IS GREATER THAN ZERO.

******************************************************************************

INTEGER*2 IARG

DOUBLE PRECISION FUNCT

COMMON/TABLE/FUNCT(8,24),IARG(8,24)

COMMON/SPACE/IPTABL(50),ISPACE(2000)

1,IPUSED,IFREE,IPSIZE,ISIZE

NUMRQD=1

IBLOCK=0

J=(I-NUMCHR)*100

TKLOC=J+NUMCHR

IPRCCE=999

CALL ISINIT(NUMRQD,ISTART,IBLOCK)

ISPACE(ISTART)=TKLOC
RETURN
END
SUBROUTINE CKSPCE(I,IPROCE,ISTART)

******************************************************************************

THIS ROUTINE WAS WRITTEN BY:

******************************************************************************

*             *
* DAVID C. BOSSERMAN *
* OCTOBER 1976 *
*             *

******************************************************************************

******************************************************************************

THIS ROUTINE IS WRITTEN TO PROCESS THE DATA TO BE PLACED IN AN ARRAY ELEMENT. IT IS SPECIFICALLY CHECKING THE SPACE CHARACTER TO SEE IF IT IS A NIL.

******************************************************************************

INTEGER*2 IARG

DOUBLE PRECISION FUNCT

COMMON/TABLE/FUNCT(8,24),IARG(8,24)

COMMON/SPACE/IPTABL(50),ISPACE(2000)

1,IPUSED,IPFREE,IPSIZE,ISIZE

NUMRCD=1

IBLOCK=0

IF(ISPACE(I-1).NE.'(') RETURN

IF(ISPACE(I+1).NE.')') RETURN

TKLOC=(I*100)+1

IPROCE=999

CALL ISINIT(NUMRCD,ISTART,IBLOCK)

ISPACE(ISTART)=TKLOC

RETURN

END
SUBROUTINE ISINIT(NUMREQD, ISTART, IBLOCK)

******************************************************************************

THE SCANNER INITIALIZER ROUTINE IS DESIGNED TO INTERACT WITH THE MEMORY
MANAGEMENT ROUTINE. ITS PURPOSE IS TO ACQUIRE SPACE FROM MEMORY MANAGEMENT.

******************************************************************************

******************************************************************************

THE SCANNER INITIALIZER ROUTINE WAS WRITTEN AS A UTILITY ROUTINE BY:

******************************************************************************

* *
* DAVID C. BOSSELMAN *
* OCTOBER 1976 *
* *

******************************************************************************

SPECIFIC INITIALIZER DATA ARE AS FOLLOWS:

FUNCTION NAME: ISINIT

DOMAIN:

(I) INPUT: NUMBER OF WORDS REQUIRED FROM FREESPACE.
(II) GLOBAL DATA: THE DECLARED ARRAY REPRESENTING FREESPACE.

RANGE: THE LOWER AND UPPER LOGICAL ADDRESS FOR THE
THE FREESPACE ARRAY TO BE USED FOR TOKEN GENERATION.

RULE OF CORRESPONDENCE:

ISINIT REQUESTS FROM MEMORY MANAGEMENT THE LOGICAL ADDRESS
OF AVAILABLE FREESPACE TO BE USED FOR TOKEN GENERATION.

******************************************************************************
CALL GETBLK(NUMRD,ISTART,IBLOCK)

RETURN

END
SUBROUTINE ISSMTB(IJUMP, ISTARI, IFINSH)

******************************************************************************

THIS MODULE PERFORMS THE FUNCTION OF COMPARING THE ADDRESS
ARRAY TOKENS WITH THOSE FOUND IN THE KEY WORD/FUNCTION
TABLES. IF THE VALUES ARE NOT FOUND, THEY ARE MARKED AS 00.
AT THIS TIME THE ADDRESS ARRAY TOKEN IS EXPANDED TO SHOW NOT
ONLY THE LOCATION OF THE SYMBOL IN THE USER'S PROGRAM BUT
ALSO THE TABLE LOCATION IF FOUND, OR A 0 IF NOT FOUND.

******************************************************************************

******************************************************************************

THE SYMBOL TABLE IDENTIFICATION MODULE WAS WRITTEN BY:

******************************************************************************

* *
* DAVID C. BOSSERMAN *
* OCTOBER 1976 *
* *

******************************************************************************

******************************************************************************

SPECIFIC SYMBOL TABLE DATA ARE AS FOLLOWS:

FUNCTION NAME - ISSMTB

DOMAIN:

(I) INPUT - THE BEGINNING ADDRESS OF THE ADDRESS
ARRAY GENERATED IN ISTKGN.
THE ENDING ADDRESS OF THE ADDRESS ARRAY
GENERATED IN ISTKGN.

(II) GLOBAL DATA:
THE ARRAY CALLED ISPACE.
THE KEY WORD/FUNCTION TABLES.

RANGE:
AN ARRAY OF STRING/SYMBOL ADDRESSES AND LENGTHS AND 
THE TABLE LOCATION OF THOSE STRINGS/SYMBOLS. THE 
FORMAT OF THE FULL TOKEN ARRAY IS:

******************************************************
*   *   *   *   *
* BEGINNING ADDRESS * LENGTH * TABLE LOCATION * TABLE *
*   *   *   *   *
******************************************************

RULE-CF-CORRESPONDENCE:

USE THE ADDRESS ARRAY GENERATED IN ISTKGN AND ADD 
POINTERs TO THE SYMBOL TABLE LOCATION OF THE SYMBOLS 
CONTAINED IN THE USER'S PROGRAM, IF APPLICABLE.

******************************************************
INTEGER*2 IARG
DOUBLE PRECISION FUNCT
COMMON/TABLE/FUNCT(8,24),IARG(8,24)
COMMON/SPACE/IPTABL(50),ISPACE(2000)
1,IPUSED,IFREE,IPSIZE,ISIZE
IBLOCK=0
IFINSH=ISTART

******************************************************

A DO ICOP IS ENTERED TO PROCESS EACH SYMCLI FROM THE ADDRESS 
ARRAY.

******************************************************

DO 100 I=IJUMP,IFINSH

******************************************************

SUBROUTINE ISYMCK, SYMBOL CHECK, IS CALLED TO CHECK THE 
USER'S STRINGS AGAINST KEY WORD/FUNCTIONS FOUND IN THE 
GLOBAL DATA TABLES.
ALTHOUGH A DOUBLE WORD WILL ONLY CONTAIN EIGHT CHARACTERS, THE FIRST 8 CHARACTERS OF ALL KEYWORD/FUNCTIONS ARE UNIQUE. THE USER MUST BE CAUTIONED NOT TO USE THE UNIQUE 8 OR LESS LETTER COMBINATIONS IN HIS PROGRAM UNLESS HE DOES IN FACT WANT THE KEYWORD/FUNCTION TO APPLY. EVEN THOUGH THE USER'S PROGRAM WILL HAVE GREATER THAN 8 CHARACTERS, THEY WILL BE POINTED TO BY THE TOKEN GENERATED.

********************************************************************************
IAADDR=ISFACE(I)/100
ILNGTH=ISPACE(I)-(IAADDR*100)
CALL ISYMCK(I,ILNGTH,ITABLE,ILOCAT,
1ISTART)
********************************************************************************

THE ABOVE CONDITIONAL EXPRESSION HAVE SERVED TO CALL THE APPROPRIATE ROUTINE TO PROCESS THE SYMBOLS INVOLVED. NOTE THAT IF A SYMBOL IS PROCESSED BY A SUBROUTINE AND NOT FOUND, ITABLE AND ILOCAT REMAINS ZERO. THEREFORE, ANY SYMBOL NOT FOUND OR GREATER THAN 12 CHARACTERS IS AUTOMATICALLY MARKED WITH A ZERO. AFTER THE STRING IS PROCESSED, THE TABLE NUMBER AND LOCATION WITHIN THE TABLE ARE PASSED BACK TO THE CALLING ROUTINE IN THE VARIABLES ITABLE AND ILOCAT RESPECTIVELY. THIS DATA IS THEN PROCESSED INTO THE COMPLETED SYMBOL TOKEN. THE TABLES ARE NUMBERED AS follows:

0 - THE PROGRAM UNIQUE SYMBOL.
1 - ONE CHARACTER SYMBOL TABLE.
2 - TWO CHARACTER SYMBOL TABLE.
3 - THREE CHARACTER SYMBOL TABLE.
4 - FOUR CHARACTER SYMBOL TABLE.
5 - FIVE CHARACTER SYMBOL TABLE.
6 - SIX CHARACTER SYMBOL TABLE.
7 - SEVEN CHARACTER SYMBOL TABLE.
8 - EIGHT CHARACTER SYMBOL TABLE.

*****************************************************************************

ISFACE(I)=(((IAADDR*100) +ILNGTH)*10000)
+(ILCCAT*100) +ITABLE*10))

IFOUND=0

100 CONTINUE

RETURN

END
SUBROUTINE
ISYMCK (I, ILNGTH, ITABLE, ILOCAT, IFOUND, ISTART)

*******************************************************************************

THIS MODULE IS DESIGNED TO SEARCH THE GLOBAL KEY WORD/ FUNCTION TABLES AND RETURN THE TABLE NUMBER AND POSITION WITHIN THE TABLE, IF A MATCH OCCURS BETWEEN THE STRING BEING TESTED AND THE TABLE.

*******************************************************************************

*******************************************************************************

THE SYMBOL CHECK MODULE WAS WRITTEN BY:

*******************************************************************************

* *
* DAVID C. BOSSERMAN *
* OCTOBER 1976 *
* *

*******************************************************************************

*******************************************************************************

SPECIFIC SYMBOL CHECK DATA ARE AS FOLLOWS:

FUNCTION NAME - ISYMCK

DOMAIN:

(I). INPUT - THE STARTING ADDRESS OF THE STRING TO BE PROCESSED.

(II). GLOBAL DATA:

THE ARRAY CALLED ISPACE.

THE KEY WORD/FUNCTION TABLES.

RANGI: THE TABLE NUMBER AND POSITION WITHIN THE TABLE, IF A STRING MATCH IS FOUND.

RULE-CF-CORRESPONDENCE:

ACCOMPLISH A SEARCH OF KEY WORD/FUNCTION TABLES TO SEE
IF THE STRING USED BY THE USER IS A KEY WORD/FUNCTION, AND
RETURN TABLE INFORMATION IF SO.

**********************************************************************

INTEGER*2 IARG

DIMENSION ICONVT(8), IPACK(33)

DOUBLE PRECISION FUNCT, ICHECK, IDSPAC

COMMON/TABLE/FUNCT(8,24),IARG(8,24)

COMMON/SPACE/IPTABL(50),ISPACE(2000)

1,IPUSED,IFREE,IPSIZE,ISIZE

COMMON/DOUBLE/IDSPAC(500)

**********************************************************************

FIRST, THE ADDRESS OF THE STRING IN ISPACE IS DETERMINED.
NEXT, THE LENGTH OF THE STRING IS EXTRACTED. IF THE LENGTH
IS EQUAL TO 9-12, IT IS CHANGED TO 8 BECAUSE ONLY A FULL
WORD (8 BYTES) CAN BE CHECKED AT ONE TIME. IF THE LENGTH IS
GREATER THAN 12, A RETURN TO THE CALLING PROGRAM IS MADE.

**********************************************************************

ITABLE=0

ILOCAT=0

IADDR=ISPACE(I)/100

N=INOTH

IF((N.GE.9).AND.(N.LE.12)) N=8

IF(N.GT.12) RETURN

**********************************************************************

THE ARRAY USED TO CONVERT THE STRING IS INITIALIZED WITH
BLANKS. THIS IS FOLLOWED BY ASSIGNING TO EACH ELEMENT OF
THE ARRAY (UP TO 8) ONE CHARACTER OF THE STRING.

**********************************************************************

DO 5 K=1,8

5 continue
ICCNVT(K)='
DO 7 J=1,N
IPT=IADDR+ (J-1)
ICCNVT(J)=ISPACE(IPT)
7  CONTINUE

AN ENCODE ROUTINE IS USED TO READ THE CONVERSION ARRAY. THE
CONVERSION ARRAY HOLDS ONE CHARACTER PER ELEMENT IN AN A4
FORMAT. Thus, one character and three spaces are in each
ELEMENT. THE ENCODE CAUSES EACH ELEMENT TO BE READ, BUT
ONLY THE FIRST BYTE (THE CHARACTER BYTE) IS READ AND THE
SPACES SKIPPED. THE ENCODE ARRAY IS 33 ELEMENTS BECAUSE
INTERDATA REQUIRES A MINIMUM OF 132 BYTES PER ENCODE.

ENCODE(IFACK,6) ICCNVT

FORMAT(33A1)

A DECODE IS ACCOMPLISHED TO CONVERT THE ENCODED BLOCK TO A
DOUBLE WORD.

DECODE(IFACK,10) ICHECK

FORMAT(A8)

DO 100 J=1,24

THE APPROPRIATE KEY WORD/FUNCTION TABLE IS SEARCHED. THE
LAST ELEMENT IN EACH TABLE IS REPRESENTED BY A '<'. IF THE
TERMINAL SYMBOL IS FOUND BEFORE A MATCH, A RETURN IS MADE.

IF(ICHECK.EQ.FUNCT(N,J)) GO TO 200
IF(PUNCT(N,J) .EQ. '<') GO TO 150

100 CONTINUE

*******************************************************************************

THE APPROPRIATE TABLE AND LOCATION ARE SET, IF A MATCH IS MADE. IFOUND IS A SIGNAL TO THE CALLING ROUTINE AS TO WHETHER A MATCH WAS MADE.

*******************************************************************************

150 N=0
    J=0

200 ITABLE=N
    ILOCAT=J
    IFOUND=1

RETURN
END
APPENDIX E

SUBROUTINE INTERP(IJUMP,IFINSE,IDSTA,IDSTOP,ISTART)

*****************************************************************************

INTERP IS A DRIVER MODULE WHICH CURRENTLY CALLS ONLY THE ROUTINE WHICH
CONSTRUCTS A GENERAL TREE OF THE USER'S PROGRAM. THE ROUTINE IS CALLED FROM THE
LISP INTERPRETER DRIVER.

*****************************************************************************

*****************************************************************************

THIS DRIVER WAS WRITTEN BY:

*****************************************************************************

* *
* DAVID C. POSSELMAN *
* OCTOBER 1976 *
* *

*****************************************************************************

*****************************************************************************

SPECIFIC INTERPRETER DATA ARE AS FOLLOWS:
FUNCTION NAME - INTERP

DOMAIN:
(I) INPUT
   - THE BEGINNING ADDRESS OF THE TOKEN ARRAY.
   - THE ENDING ADDRESS OF THE TOKEN ARRAY.

(II) GLOBAL DATA
   - THE ARRAY CALLED ISPACE.
   - THE ARRAY CALLED IDSPAC.

RANGE: A SERIES OF NODES REPRESENTING A GENERAL TREE
OF THE USER'S PROGRAM.

RULE-OF-CORRESPONDENCE

CONSTRUCT A GENERAL TREE STRUCTURE FROM THE TOKENS GENERATED IN ISCAN.

*******************************************************************************

ICODE=0

WRITE(6,1)

1 FORMAT(' ',2X,'INTERP WAS CALLED.')

CALL IGTREE(IJUMP,IFINSH,IDSTA,IDSTCPI,ISTART,IECODE)

IF(IECODE.EQ.999) RETURN

RETURN

END
SUBROUTINE
IGTREI(IJUMP,IFINSH,IDSTA,IDSTOP,ISTART,IECODE)

*****************************************************************************

THE GENERAL TREE ROUTINE IS DESIGNED TO PRODUCE AN ARRAY OF DOUBLE WORDS WHICH REPRESENT A LISP PROGRAM. EACH DOUBLE WORD CONTAINS INFORMATION FIELDS WHICH PROVIDE THE DATA NECESSARY TO CONSTRUCT A GENERAL TREE. THE SIXTEEN BYTES ALLOWED ARE DIVIDED, FROM LEFT TO RIGHT AS FOLLOWS:

<table>
<thead>
<tr>
<th>FIELD</th>
<th>LENGTH</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>4</td>
<td>LOCATION</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>LENGTH</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>SON</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>BROTHER</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>MARK</td>
</tr>
</tbody>
</table>

THE ROUTINE ACTS AS A CONTROL ROUTINE AND CALLS THE APPROPRIATE SUBROUTINE BASED ON THE CHARACTER BEING PROCESSED.

*****************************************************************************

THIS ROUTINE WAS WRITTEN BY:

*****************************************************************************

*                           *
* DAVID C. BOSSELMAN        *
*    NOVEMBER 1976         *
*                           *
*****************************************************************************

DOUBLE PRECISION IDSPAC

COMMON/SPACE/IPTAEL(50),ISPACE(2000)
!
I,IPUSED,IFREE,IPSIZE,ISIZE

COMMON/DCUELE/IDSPAC(500)
WRITE (6, 1)
1 FORMAT (' ', 2X, 'IGTREE WAS CALLED.')
N=0
ICODE=0
NXPT=0
I=IJUMP

******************************************************************************
A POINTER IS INITIALIZED TO CONTROL A STACK WHICH WILL HOLD THE LENGTH OF SCN NODES. THE POINTER IS CALLED IPARPT AND IS SET INITIALLY TO ZERO. AT THE SAME TIME A VARIABLE IFINIT IS ASSIGNED THE VALUE OF IFINSH. IFINIT WILL BE USED THROUGHOUT THE GENERAL TREE OPERATIONS TO CONTROL THE END LIMIT OF THE STRING BEING PROCESSED.

******************************************************************************
IPARPT=0
IFINIT=IFINSH
ISETD=0
IDSTA=0

******************************************************************************
A NUMBERED STATEMENT IS USED TO PERFORM IN THE SAME MANNER AS A FL1 DO WHILE STATEMENT. THE STATEMENTS FOLLOWING THE LINE NUMBERED 10 ARE TO BE EXECUTED UNTIL ALL SCN NODES ARE PROCESSED DOWN TO THEIR TERMINAL NODE. THE EXIT FROM THE LOOP IS DEPENDENT UPON THE POINTER WHICH POINTS TO THE NEXT NODE BEING PROCESSED EXCEEDING THE NODES AVAILABLE.

******************************************************************************
******************************************************************************
THE ADDRESS ARRAY TCKEN IS PROCESSED TO YIELD THE ADDRESS OF THE STRING TO BE PROCESSED.
10 IF(I.EQ.0) GO TO 20

TERMINAL NODES HAVE NO LENGTH, OR SON. THEREFORE, I IS SET TO ZERO WHEN ONE IS PROCESSED. IF PROCESSING IS ALLOWED WITH I=0, AN ERROR CONDITION WILL RESULT. IF I=0 A TRANSFER IS MADE TO PROCESS THE NEXT NODE.

IADDR=ISPACE(I)/1000000

ONCE THE ADDRESS IS DETERMINED, FOUR POSSIBILITIES EXIST. IF THE SYMBOL/STRING LOCATED AT THE ADDRESS IS A RIGHT PARENTHESIS OR A PERIOD, AN ERROR IS APPARENT AND A TRANSFER WILL BE MADE TO PRINT AN APPROPRIATE ERROR MESSAGE. IF A LEFT PARENTHESIS IS ENCOUNTERED, THE SUBROUTINE TO PROCESS PARENTHESSES IS CALLED. IF THE FIRST THREE CONDITIONS ARE NOT MET, THE SYMBOL/TOKEN IS TREATED AS A VARIABLE AND THE VARIABLE SUBROUTINE IS CALLED.

IF((ISPACE(IADDR).NE.'(').AND.(ISPACE(IADDR).NE.'1')').AND.(ISPACE(IADDR).NE.'')) CALL VARBLE(I, ININIT, NMT, IDSTA, ISETD, ISTART, IPARFT, ICODE)

IF(ISPACE(IADDR).EQ.'(') CALL ILPARN(I, ININIT 1, NMT, IDSTA, ISETD, ISTART, IPARFT)

IF(ISPACE(IADDR).EQ.'') CALL IRPARN(IECCDE)

IF(ISPACE(IADDR).EQ.'') CALL IFEROD(IECCDE)

IF(ICODE.EQ.999) RETURN

WRITE(6,100) (IDSPAC(J), J=IDSTA, ISETD)
THE COMBINATION OF NXTPT (THE POINTER TO THE NEXT NODE TO BE PROCESSED) AND ISETD (THE POINTER TO THE LAST PARENT NODE CONSTRUCTED) ARE COMPARED. IF NXTPT EXCEEDS ISETD, THE 'DO WHILE' LOOP IS EXITED, INDICATING THAT THE GENERAL TREE IS COMPLETE.

*******************************************************************************

20 IF(NXTPT.GT.ISETD) GC TO 150

*******************************************************************************

IN THE CALLED ROUTINES, A POINTER CALLED NXTPT HAS BEEN SET TO THE NEXT NODE TO BE PROCESSED. THE NCDE PROCESSING IS DONE WITH ONLY ONE PASS. THEREFORE, WHEN NO BROTHERS ARE BEING PROCESSED, THE ROUTINE WILL CONTINUE A CYCLE OF PROCESSING THE NEXT NODE - CREATE A SON NODE, CREATE A SON NODE AND/OR A BROTHER NODE, INCREMENT THE POINTER - PROCESS THE NEXT NCDE. THE VARIABLE I REPRESENTS THE ADDRESS ARRAY LOCATION OF THE NEXT STRING/TOKEN TO BE PROCESSED.

*******************************************************************************

INTER=IDSPAC(NXTPT)/10000000.0
I=((IDSPAC(NXTPT)-(INTER*10000000.0))/10000.0)
IDSPAC(NXTPT)=IDSPAC(NXTPT)+1.
NXTPT=NXTPT+1
WRITE(6,100) IDSPAC(NXTPT-1)
100 FORMAT(' ',2X,'IDSPAC= ',F17.0)
WRITE(6,101) I,IFINIT,NXTPT,IDSTA,ISETD
GC TO 10
150 IDSTOP=ISETD
RETURN
END
SUBROUTINE
VARBLE(I,IFINIT,NXPT,IDSTA,ISETD,ISTART,IPARET,ICODE)

******************************************************************************

THIS ROUTINE IS DESIGNED TO:

1. PLACE THE VARIABLE'S LOCATION WITHIN THE ADDRESS
   ARRAY INTO THE LOCATION FIELD OF A DOUBLE WORD
   NODE.

2. IF THE VARIABLE IS FOLLOWED BY A LEFT
   PARENTHESIS, PLACE THE ADDRESS OF THE LEFT
   PARENTHESIS IN THE SON FIELD OF THE NCODE.

3. IF THE VARIABLE IS FOLLOWED BY A LEFT
   PARENTHESIS, FIND THE FURTHEREST SAME LEVEL
   RIGHT PARENTHESIS.

4. IF THE STRING TO THE RIGHT OF THE SAME
   LEVEL RIGHT PARENTHESIS IS A VARIABLE,
   COMPLETE ITS DOUBLE WORD NODE USING STEPS
   1-4.

******************************************************************************

THIS ROUTINE WAS WRITTEN BY:

******************************************************************************

           *  *  *
           *
           * DAVID C. BOSSEMAN *
           *
           * NOVEMBER 1976 *
           *

******************************************************************************

DOUBLE PRECISION IDSPAC
COMMON/SPACE/IPTABL(50),ISPACE(2000)
1,IPUSED,IPREE,IPSIZE,ISIZE
COMMON/DUOUBLE/IDSPAC(500)
WRITE (6, 1)
1 FORMAT (' ', 2X, 'VARWEB WAS CALLED. ')
IMODFY = 0
NUMRQD = 1
IBLOCK = 0
LENGTH = 0
IF (ICODE.EQ. 1) GO TO 10
IADDR = ISPACE (I) / 1000000
IF (NXPT .GT. 0) LENGTH = ISPACE (IPARPT)
IF (NXPT .GT. 0) IPARPT = IPARPT + 1
IF (NXPT .GT. 0) IFINIT = (I + LENGTH) - 1
IF (NXPT .GT. 0) NDEPA = (NXPT - 1) + IMODFY
IF (NXPT .GT. 0)
IDSPAC (NDEPA) = IDSPAC (NDEPA) - (I * 10000.)
IF (NXPT .GT. 0)
IDSPAC (NDEPA) = IDSPAC (NDEPA) + ((ISETL + 1) * 10000.)

***************************
THE DATA TO OBTAIN A DOUBLE WORD FROM MEMORY MANAGEMENT IS
INITIALIZED. AFTER THE DOUBLE WORD IS REQUESTED, THE START
POINT OF THE GENERAL TREE ARRAY IS CAPTURED BY THE VARIABLE
IDSTA. THEN THE ADDRESS ARRAY LOCATION (IADDR) IS PLACED IN
THE LOCATION FIELD OF THE DOUBLE WORD. AS ALL FIELDS OF THE
DOUBLE WORD WILL NOT BE COMPLETED IN ONE PASS, A POINTER
CALLED NXPT IS KEPT UP TO DATE. THIS POINTER POINTS TO THE
DOUBLE WORD NODE TO BE PROCESSED WHEN AN OPERATION IS
COMPLETED. INITIALLY IT WILL BE THE FIRST DOUBLE WORD. THE
LAST FIELD OF THE DOUBLE WORD IS A MARK FIELD. A ONE IN THE
MARK FIELD REPRESENTS THAT THE LOCATION FIELD AND LENGTH
FIELD OF THE NODE HAVE BEEN COMPLETED ROUTE THAT FOR A
VARIABLE THE LENGTH IS 0. FOR A SET OF PARENTHESES THE LENGTH WILL BE FROM THE LEFT TO THE RIGHT IN TERMS OF ADDRESS ARRAY ELEMENTS.

*************************************************************************
10 CALL ISINIT(NUMR(D,ISETD,IELOCK))
   IF(IDSTA.EQ.0) IDSTA=ISETD
   IF(NXTPT.EQ.0) NXTPT=ISETD
   C=I
   IDSPAC(ISETD)=C*1000000000000.D0+1.
   WRITE(6,2)IDSPAC(ISETD)
   2 FORMAT(' ',2X,'IDSPAC(ISETD)=',F17.0)

*************************************************************************
THE ROUTINE NOW LOOKS AT THE NEXT ITEM IN THE ADDRESS ARRAY TO DETERMINE ITS NATURE. NOTE THAT FIRST A CHECK IS MADE TO SEE IF THE NEXT ITEM IS OUTSIDE THE ADDRESS ARRAY. IF SO, A RETURN IS MADE.

*************************************************************************
   I=I+1
   IF(I.GT.IFINIT) RETURN
   IF(ICODE.EQ.1) RETURN
   IADDR=ISPACE(I)/1000000

*************************************************************************
THE POSITION FOLLOWING THE VARIABLE IS IDENTIFIED. IF THE POSITION IS A LEFT PARENTHESIS, ITS ARRAY LOCATION IS PLACED (AS A TEMPORARY MEASURE) IN THE SON FIELD OF THE NODE. THIS WILL BE USED LATER TO PROCESS THE REMAINDER OF THE STRINGS/SYMBOLS FOUND IN THE ADDRESS ARRAY.

*************************************************************************
   IF(ISPACE(IADDR).EQ.'(')
IDSPAC(ISETD) = IDSPAC(ISETD) + (I*10000.)
WRITE (6,2) IDSPAC(ISETD)

***********************************************************************
IF THE NEXT POSITION IS A LEFT PARENTHESIS, THE LPARN SUBROUTINE IS CALLED TO FIND THE FURTHERMOST SAME LEVEL RIGHT PARENTHESIS.

***********************************************************************
IF(ISPACE(IADDR).EQ."('") CALL LPARN(I,IFINIT,ISTART,IPARPT)

***********************************************************************
IF THE DATA RETURNED SHOWS THAT A SAME LEVEL PARENTHESIS WAS FOUND AND THERE IS A NEXT SYMBOL, THE BROTHER FIELD OF THE NODE IS COMPLETED.

***********************************************************************
IF(I.LE.IFINIT) IDSPAC(ISETD) = IDSPAC(ISETD) + ((ISETD+1.)*10.)
WRITE (6,2) IDSPAC(ISETD)
IF(I.LE.IFINIT) GO TO 10

***********************************************************************
The required address array has been searched for the same level variables. Therefore, a return is made.

**************************************************************************
RETURN
END
SUBROUTINE
ILPARN(I, IFINIT, NXTPT, IDSTA, ISETD, ISTART, IPARPT)

WRITE(6, 100)

100 FORMAT(' ', 2X, 'ILPARN WAS CALLED. ')

******************************************************************************

THIS ROUTINE MAY BE ENTERED IN ONE OF TWO CASES:

1. THE FIRST TOKEN OF A LISP PROGRAM MAY BE A LEFT PARENTHESIS AND, THEREFORE, THE SAME LEVEL CONTIGUCUS PARENTHESES BECOME A PARENT NODE, WITH OR WITHOUT BROTHERS AND OR SCNS.

2. THE SCN OF A PARENT NODE MAY BE A SET OF MATCHED PARENTHESES.

TO HANDLE THESE EVENTUALITIES, THIS ROUTINE WILL FIRST DETERMINE WHICH CASE APPLIES AND WILL CALL LPFST, IF CASE ONE IS APPLICABLE AND LPLTR, IF CASE TWO IS APPLICABLE.

IL 1153

******************************************************************************

IF(IPARPT.NE.0) GO TO 20

IF(IPARPT.EQ.0) CALL LPFST(I, IFINIT, IPARPT, ISETD, 1, IDSTA, ISTART, NXTPT)

IF(I.GT.IFINIT) RETURN

CALL VARRLE(I, IFINIT, NXTPT, IDSTA, ISETD, ISTART, 1, IPARPT, ICCDE)

RETURN

20 CALL LPLTR(I, IFINIT, IPARPT, ISETE, IDSTA, ISTART, 1, NXTPT)

RETURN

END
SUBROUTINE IRPARN(IECODE) C

******************************************************************* IF
THE INITIAL STRING PROCESSED IN THE GENERAL TREE STRUCTURE IS A RIGHT PARENTHESIS, AN ERROR CONDITION EXISTS. IF THAT IS THE CASE, AND ERROR MESSAGE IS PRINTED AND THE VARIABLE IECODE IS SET TO 999. IECODE=999 WILL CAUSE RETURN TO THE LISP DRIVER.

*******************************************************************

THIS EXECUTIVE WAS WRITTEN BY:

*******************************************************************

*   *
*  DAVID C. BOSSERMAN  *
*   NOVEMBER 1976   *
*   *

*******************************************************************

WRITE(6,100)

100 FORMAT(9,2X,'ERROR****A RIGHT PARENTHESIS PRECEDED ALL OTHER STRINGS.')

IECODE=999
RETURN
END
SUBROUTINE IFEROD (IECODE)

********************************************************************

IF THE INITIAL STRING PROCESSED IN THE GENERAL TREE
STRUCTURE IS A PERIOD, AN ERROR CONDITION EXISTS. IF THAT
IS THE CASE AN ERROR MESSAGE IS PRINTED AND THE VARIABLE
IECODE IS SET TO 999. IECODE=999 WILL CAUSE RETURN TO THE
LISP DRIVER.

********************************************************************

THIS ROUTINE WAS WRITTEN BY:

********************************************************************

*                        *
* DAVID C. BOSSERMAN     *
* NOVEMBER 1976           *
*                        *

********************************************************************

WRITE (6,100)

100 FORMAT (' ',2X,'ERROR****A PERIOD PRECEDED ALL OTHER
STRINGS.' )

IECODE=999

RETURN

END
SUBROUTINE LEFAR(N, IFINIT, ISTART, IPARPT)

*******************************************************************************

THIS ROUTINE IS DESIGNED TO FIND THE FURTHERMOST CONTIGUOUS
SAME LEVEL RIGHT PARENTHESIS AND RETURN ITS LOCATION IN THE
ADDRESS ARRAY TO THE CALLING ROUTINE.

*******************************************************************************

THIS ROUTINE WAS WRITTEN BY:

*******************************************************************************

* * *
* DAVID C. BOSSELMAN *
* NOVEMBER 1976 *
* *

*******************************************************************************

COMMON/SPACE/IPTAEL(50), ISPACE(2000)
1, IFUSE, IFREE, IPSIZE, ISIZE

*******************************************************************************

FIRST THE LEVEL NUMBER IS ASSIGNED TO A CHECK VARIABLE CALLED LEVEL.

*******************************************************************************

WRITE(6, 1)
1 FORMAT(' ', 2X, 'LEFAR WAS CALLED.')

NUMBD=1
IBLOCK=1
IPARST=I
LEVSAM=0
IADDR=ISPACE(I)/1000000
ICODE=0
N=IADDR+80
LEVEL=ISPACE(N)
A DO LOOP IS ENTERED TO CHECK ALL REMAINING ADDRESS ARRAY
ELEMENTS UNTIL A SAME LEVEL PARENTHESIS IS FOUND.

DO 100 J=I,IFINIT
IADDR=ISPACE(J)/1000000
N=IADDR+80
IF(ISPACE(IADDR).EQ.' ') LEVSAM=ISPACE(N)

A CHECK IS MADE TO SEE IF THE LEVEL IS A MATCH. IF SO A
ROUTINE IS CALLED TO CHECK THE NEXT POSITION. IF THE NEXT
POSITION IS A LEFT PARENTHESIS AND OF THE SAME LEVEL THE
SEARCH IS CONTINUED, IF NOT IT IS TERMINATED.

IF(LEVEL.EQ.LEVSAM) CALL NXTPAR(J,IFINIT,ICODE,
LEVEL,L)
LEVSAM=0
IF(ICODE.EQ.1) GO TO 200
100 CONTINUE
200 I=L

ONCE THE FURTHERMOST SAME LEVEL PARENTHESIS IS FOUND, THE
TOTAL LENGTH OF THE ELEMENTS INCLUDED BY THE PARENTHESES IS
STACKED. IPARPT WILL POINT TO THE INITIAL LENGTH ELEMENT.
EACH SAME LEVEL PARENTHESIS WILL BE REPRESENTED AT ONE TIME
OR ANOTHER IN THE STACK.

CALL ISINIT(NUMRQD,ISTART,IBLOCK)
ISPACE(ISTART)=L-IPABST
IF(IPARET.EQ.0) IPARET=ISTART
RETURN
END
SUBROUTINE NXTPAR(J,IFINIT,ICODE,LEVEL,I)

******************************************************************************

THIS ROUTINE IS CALLED AFTER A SAME LEVEL RIGHT PARENTHESIS
HAS BEEN FOUND IN SUBROUTINE LPARN. THE PURPOSE OF THIS
ROUTINE IS TO LOCK AT THE NEXT ELEMENT IN THE ADDRESS ARRAY.
IF IT IS A LEFT PARENTHESIS AND OF THE SAME LEVEL, A CODE
OF 0 IS RETURNED TO LPARN. IF NOT, A CODE OF 1 IS RETURNED.

******************************************************************************

THIS ROUTINE WAS WRITTEN BY:

******************************************************************************

* * *
* DAVID C. BOSSERMAN *
* NOVEMBER 1966 *
* *

******************************************************************************

COMMON/SPACE/IPTAEI(50),ISPACE(2000)
1,IPUSED,IFREE,IPSIZE,ISIZE
WRITE(6,1)
1 FORMAT(' ',2X,'NXTPAR WAS CALLED.'),
ICODE=1
I=J+1
IF(I.GT.IFINIT) I=I
IF(I.GT.IFINIT) RETURN
IADDR=ISPACE(I)/1000000
IF(ISPACE(IADDR) .EQ. '(') GO TO 10
L=I
RETURN
10 N=IADDR+80
LEVSAM=ISPACE(N)
IF(LEVREL.EQ.LEVEL) ICODE=0
L=I
RETURN
END
SUBROUTINE
LPFST(I,IFINIT,IFARET,ISETD,IDSTA,ISTART,NXTPT)

**********************************************************************
THIS SUBROUTINE IS CALLED IF THE FIRST TOKEN ENCOUNTERED IS A
LEFT PARENTHESIS. ITS PURPOSE IS TO CREATE DOUBLE WORD
NODES WITH THE FOLLOWING POSSIBILITIES:

1. IF THE ENTIRE PROGRAM IS INCLUDED IN THE SAME LEVEL
PARENTHESIS:
   A. CREATE A DOUBLE WORD NODE WHOSE LOCATION
      FIELD CONTAINS THE TOKEN ARRAY LOCATION OF THE
      BEGINNING LEFT PARENTHESIS.
   B. PLACE THE LENGTH IN THE LENGTH FIELD.
   C. STACK THE LENGTH MINUS TWO ON THE STRING
      LENGTH STACK.
   D. PLACE THE ADDRESS OF THE NEXT TOKEN TO BE
      PROCESSED IN THE SON FIELD.

2. IF THE PROGRAM BEGINS WITH A LEFT PARENTHESIS AND
   OUTSIDE THE SAME LEVEL RIGHT PARENTHESIS IS FOUND
   VARIABLES:
   A. PROCESS AS IN 1.
   B. CREATE BROTHER NODES AS APPLICABLE WITH
      POINTERS.

**********************************************************************

THIS SUBROUTINE WAS WRITTEN BY:

***************************************************************
*                                                          *
*                   * DAVID C. BOSSERMAN *
*       NOVEMBER 1976     *
*                                                          *
***************************************************************
DOUBLE PRECISION FUNCT, IDSPAC
COMMON/SPACE/IPTABL(50),ISPACE(2000)
1,IPUSED,IFREE,IPSIZE,ISIZE
COMMON/DDOUBLE/IDSPAC (500)
NUMROD=1
IBLOCK=1

THE FIRST STEP IS TO ACQUIRE A DOUBLE WORD NCDE AND FILL THE LOCATION FIELD.

CALL ISINIT(NUMROD,ISETD,IBLOCK)
IDSTA=ISETD
WXTPT=ISETD
IDSPAC(ISETD)=I*100000000000.DO+1.

THE NEXT STEP IS TO FIND THE FURTHERCST CONTIGOUS SAME LEVEL PARENTHESES. THIS IS DONE BY SEARCHING THE PROGRAM FOR THE NEXT SAME LEVEL PARENTHESES.

K=0
IADDR=ISPACE(I)/1000000
N=IADDR*80
LEVEL=ISPACE(N)

A DO LOOP IS ENTERED TO CHECK ALL REMAINING ADDRESS ARRAY ELEMENTS UNTIL A SAME LEVEL PARENTHESES IS FOUND.
DO 100 J=I,IPINIT
   K=J
   IADDR=ISPACE(J)/1000000
   N=IADDR+80
   IF(ISPACE(IADDR).EQ.' ') LEVSAM=ISPACE(N)
   IF(LEVEL.EQ.LEVSAM) CALL LPROCE(J,IPINIT,ISTART,I,
   (IPARPT,ISETD,IDSTIA,NXTPT,IECODE)
   IF(IECODE.EQ.1) RETURN
   LEVSAME=0
100 CONTINUE
RETURN
END
SUBROUTINE LPROCE(J,IFINIT,ISTART,I,IPARPT,ISEID, TIDSTA,NXTPT,IECODE)

*******************************************************************************

THIS ROUTINE IS DESIGNED TO FIND THE FURTHERMOST SAME LEVEL
RIGHT PARENTHESIS WHEN CALLED FROM LPPST. IN ADDITION, IT
IS RESPONSIBLE FOR LOOKING FOR ALL FIRST LEVEL BROTHER
NOLES.

*******************************************************************************

*******************************************************************************

THIS ROUTINE WAS WRITTEN BY:

*******************************************************************************

*
*
* DAVID C. BOSSELMAN *
*
* NOVEMBER 1976 *
*
*******************************************************************************

*******************************************************************************

FIRST A TEST IS MADE TO SEE IF THE NEXT CHARACTER EXCEEDS
THE PROGRAM LENGTH. IF SO, THE LENGTH IS STACKED AND A
RETURN IS MADE.

*******************************************************************************

NUMRD=1
IBLOCK=0
IECODE=0
ICODE=1
K=J+1
IF(K.GT.IFINIT) GO TO 100
IADDR=ISPACE(K)/1000000
IF(ISPACE(IADDR).EQ.' ') GO TO 200
IF THE NEXT CHARACTER IS A PERIOD OR RIGHT PARENTHESIS, AN ERROR IS APPARENT AND APPROPRIATE ERROR ROUTINES ARE CALLED.

IF(ISPACE(IADDR).EQ.'.') CALL IPEROE(ICODE)
IF(ISPACE(IADDR).EQ.'')) CALL IRPARN(ICODE)

IF THE NEXT CHARACTER IS NONE OF THE ABOVE, IF IT REPRESENTS A CHARACTER STRING SO THE NODE BEING PROCESSED IS COMPLETED AND SUBROUTINE VARBLE IS CALLED.

IF(ICODE.EQ.1) RETURN
LENGTH= (J-I)+1
CALL ISINIT(NUMRQD,ISTART,IBLOCK)
IF(IPARPT.EQ.0) IPARPT=ISTART
ISPACE(ISTART)=LENGTH-2
IDSPAC(ISETD)=IDSPAC(ISETD)+(LENGTH*10000000.DO)
IDSPAC(ISETD)=IDSPAC(ISETD)+((I+1)*10000.)
IDSPAC(ISETD)=IDSPAC(ISETD)+((ISETD+1)*10.)
I=K
50 CALL VARBLE(I,IFINIT,NXTPT,IDSTA,ISETD,ISTART,
IPARPT,ICODE)
IF(I.GT.IPINIT) RETURN
IDSPAC(ISETD)=IDSPAC(ISETD)+((ISETD+1)*10.)
IADDR=ISPACE(I)/1000000
IF(ISPACE(IADDR),EQ,'.') CALL IPEROE(ICODE)
IF(ISPACE(IADDR).EQ.'')) CALL IRPARN(ICODE)
IF(ISPACE(IADDR).NE.'(') GO TO 50
CALL ISINIT(NUMRQD,ISETD,IBLOCK)
IDSPAC (ISETD) = (I*1000000000.0 + 1).
RETURN

******************************************************************************
THE LENGTH OF THE STRING IS DETERMINED AND STORED IN THE PARENT NODE. THE SON FIELD IS SET IN THE PARENT NODE. THE SON'S LENGTH IS STORED ON A TEMPORARY STACK.
******************************************************************************

100 CALL ISINIT(NUMRQD, ISTART, IBLCK)
IF (IPARPT.EQ.0) IPARPT=ISTART
LENGTH = (J-I) + 1
ISPACE (ISTART) = LENGTH - 2
IDSPAC (ISETD) = IDSPAC (ISETD) + (LENGTH*10000000.0)
IDSPAC (ISETD) = IDSPAC (ISETD) + (I+1)*10000.
I=K
RETURN

******************************************************************************
IF THE NEXT CHARACTER TO BE PROCESSED IS ANOTHER LEFT PARENTHESIS, IT REPRESENTS THE BEGINNING OF A BROTHER STRING. TO GET READY TO PROCESS THE BROTHER THE NODE BEING PROCESSED IS COMPLETED BY SETTING ITS LENGTH, SON AND BROTHER FIELDS TO POINT TO THE APPROPRIATE DATA. THE STRING LENGTH OF THE NODE'S SON IS STACKED AND THEN THE BROTHER NODE IS CREATED. AFTER CREATION, A RETURN IS MADE TO LPFST TO CONTINUE PROCESSING.
******************************************************************************

200 LENGTH = (J-I) + 1
CALL ISINIT (NUMRQD, ISTART, IBLCK)
IF (IPARPT.EQ.0) IPARPT=ISTART
ISPACE (ISTART) = LENGTH - 2
IDSPAC(ISETD) = IDSPAC(ISETD) + (LENGTH * 10000000.0D0)
IDSPAC(ISETD) = IDSPAC(ISETD) + ((I+1) * 10000.)
IDSPAC(ISETD) = IDSPAC(ISETD) + ((ISETD+1) * 10.)
CALL ISINIT(NUMRQD, ISETD, IBLOCK)
IDSPAC(ISETD) = (K * 1000000000000.0D0) + 1.
I = K
RETURN
END
SUBROUTINE
LPLTR(I, IFINIT, IPARPT, ISETD, IDSTA, ISTART, NXTFT)

******************************************************************************

THIS ROUTINE IS DESIGNED TO PROCESS THE STRING FOUND BETWEEN TWO SAME LEVEL PARENTHESES. I IS AN IN VARIABLE AND REPRESENTS THE SCN OF A PARENT NODE. THE FIRST ACTION WILL BE TO SET THE PARENT'S SON FIELD TO ZERO. NEXT A DOUBLE WORD NODE WILL BE OBTAINED AND THE PARENT'S SON FIELD WILL BE SET TO POINT TO THE SON NODE. THEN THE SON WILL BE PROCESSED.

******************************************************************************

THIS ROUTINE WAS WRITTEN BY:

******************************************************************************

                  *                        *
                  *   DAVID C. BOSSERMAN   *
                  *      NOVEMBER 1976    *
                  *                        *

******************************************************************************

DOUBLE PRECISION FUNCT, IDSPAC
COMMON/SPACE/IPTABL(50), ISPACI(2000)
I, IPUSED, IPFREE, IPSIZE, ISIZE
COMMON/DCUELE/IDSPAC(500)
NUMRD=1
IBLOCK=1
LEVEL=0
LEVSAF=0
LEVNUM=0
WRITE (6, 1)
1 FORMAT(' ', 2X, 'LPLTR WAS CALLED.')
NODEPA = (NXTFT - 1)
IDSPAC (NODEPA) = IDSPAC (NODEPA) - (I * 10000.)

******************************************************************************
THE LENGTH OF THE STRING IS RETRIEVED FROM THE STRING LENGTH STACK.
******************************************************************************
LENGTH = ISPACE (IPARPT)
IPARPT = IPARPT + 1

******************************************************************************
THE LENGTH REPRESENTS THE LENGTH OF THE SAME LEVEL STRING. HOWEVER, THE
LENGTH COULD BE DIVIDED INTO SEVERAL BROTHER NODES WHICH SHARE THE TOTAL
LENGTH. THEREFORE, A LOOP IS ENTERED TO PROCESS THE STRING.
******************************************************************************
ICODE = 0
IFINIT = (I + LENGTH) - 1
IADDR = ISPACE (I) / 1000000
N = IADDR + 80
LEVEL = ISFACE (N)
JJ = I
DO 10 J = JJ, IFINIT
IADDR = ISPACE (J) / 1000000
N = IADDR + 80
IF (ISPACE (IADDR) .EQ. '.') LEVSAM = ISPACE (N)
IF (LEVEL .EQ. LEVSAM) CALL IPROCI (I, J, ICODE, ISETD, IFICCK, ISTAR,
1 NODEPA, IFINIT, NXIPT, IPARPT, IDSTA)
LEVSAM = 0
10 CONTINUE
RETURN
END
SUBROUTINE
IPROCL(I,J,ICODE,ISETD,IBLOCK,ISTART,NODEPA,IFINIT,
******************************************************************************
SUBROUTINE IPROCL IS DESIGNED TO PROCESS A SECOND SET OF
SAME LEVEL PARENTHESES AT THE BROTHER LEVEL. IT IS CALLED
BY ILIFTR.
******************************************************************************
THIS ECUTINE WAS WRITTEN BY:
******************************************************************************
                        *                      
                        *  DAVID C. BOSSEMAN  *
                        *                      
                        *  NOVEMBER 1976  *
                        *                      
******************************************************************************
1NXPT,1PARPT,IDSTA)
DOUELE PRECISION FUNCT, IDSPAC
COMMON/SPACE/IPTABL(50),ISPACE(2000)
1,IPUSED,IPRED,IPSIZE,ISIZE
COMMON/DDOUBLE/IDSPAC(500)
WRITE(6,1)
1 FORMAT( ' ',2X,'IPROCL WAS CALLED.' )
NUMRD=1
IBLOCK=1
******************************************************************************
IF THE CODE PASSED IS NOT EQUAL TO ZERO, THEN ONE PASS
THROUGH IPROCL HAS BEEN MADE. THEREFORE, A SECOND SET OF
SAME LEVEL PARENTHESES IS ABOUT TO BE PROCESSED. THE FIRST
ACTION IS TO SET THE BROTHER FIELD TO POINT TO THE NEXT SET.
IF (ICODE.NE.0)
IDSPAC (ISETD) = IDSPAC (ISETD) + ((ISETD+1.)*10.)

SPACE IS ACQUIRED TO CREATE A NEW NODE. THE NEW NODE WILL
CONSIST OF THE STARTING POINT OF A STRING INCLOSED BY
PARENTHESES.

CALL ISINIT (NUMRCD, ISETD, IBLOCK)
IDSPAC (ISETD) = I*10000000000.0 + 1.

IF THE CODE EQUALS ZERO, THE PARENT NODE'S SON FIELD IS SET
TO POINT TO THE NODE JUST CREATED.

CALL ISINIT (NUMRCD, ISTART, IBLOCK)
ISPACE (ISTART) = LENGTH - 2

THE LENGTH OF THE NEW NODE IS PLACED IN THE LENGTH FIELD.

IDSPAC (ISETD) = (LENGTH*10000000.0 + IDSPAC (ISETD)

THE NEW SON'S ADDRESS IS PLACED IN THE SON FIELD. THIS IS A
TEMPORARY POINTER AND WILL BE REPLACED LATER BY THE ACTUAL
SON'S NODE ADDRESS.

***************************************************************************
IDSFAC(ISETD) = IDSPAC(ISETD) + ((I+1) * 10000.)
ICODE = 1
I = J + 1
IF (I .GT. IFINIT) RETURN
IADD = ISPACE(I) / 1000000
IF (ISPACE(IADD) .EQ. ' ') I = I + 1

***************************************************************************
IF THERE ARE STILL TOKENS TO BE PROCESSED IN THE STRING AND
THE NEXT ONE IS NOT A LEFT PARENTHESIS, IT MUST REPRESENT A
VARIABLE. THEREFORE, THE VARIABLE ROUTINE IS CALLED. NOTE
THAT THE BROTHER FIELD IS UPDATED FIRST.

***************************************************************************
IF (ISPACE(IADD) .NE. '(')
IDSPAC(ISETD) = IDSFAC(ISETD) + ((ISETD+1) * 110.)
IF (ISPACE(IADD) .NE. ')' CALL VARBLE(I,IFINIT
1,NXTPT,IDSTA,ISETD,ISTART,IPARPI,ICODE)
RETURN
END
APPENDIX F

PART I

CARD DECK INPUT JCL

1. ALLOCATE INDEXED FILE SPACE OF 80 CHARACTER LENGTH RECORDS:
   AL INREAD,IN,80
2. READ THE CARDS FROM THE CARD READER (CR:) INTO THE FILE:
   &COPYA CR:,INREAD

PART II

ESTABLISH PROGRAM TASKS

COMPILE THE PROGRAM (LISP):
MPORT LISP
   TASK PAUSED—RETURNED BY COMPUTER
   CLOSE 1
   AS 1, SYS1: FVRTL.OEJ,SRO,1
   CO
   END OF TASK 0—RETURNED BY COMPUTER

PART III

JCL TO RUN PROGRAM

1. LOAD TASKS:
   LO LISP
2. ASSIGN LOGICAL UNITS:
   AS 5, INREAD
   AS 6, FR: -- PR: = PRINTER
3. START THE PROGRAM EXECUTION:
   SI
APPENDIX G

PART I
PROGRAM

DEFINE ((
(FACTORIAL (LAMBDA (N)) (COND ((ZEROP N) 1)
(T (TIMES N (FACTORIAL (SUB1 N))))))))
))
TRACE (T)
FACTORIAL (6)

PART II
PROGRAM WITH PARENTHESIS LEVELS

DEFINE ((
12
(FACTORIAL (LAMBDA (N)) (COND ((ZEROP N) 1)
3 4 5 5 5 67 7 6
(T (TIMES N (FACTORIAL (SUB1 N))))))
6 7 8 9 9876543
))
21
TRACE (T)
1 1
FACTORIAL (6)
1 1
PART III
TOKEN ARRAY

806
1501
1601
16201
16309
17301
17406
18101
18201
18301
18501
18604
19101
19201
19305
19901
20001
20201
20301
32201
32401
32601
32705
33301
33501
33609
34601
34704
35201
35301
35401
35501
35601
35701
35801
35901
48101
48201
64305
64901
65001
65101
80109
81001
81101
81201
PART IV
EXPANDED TOKENS

8060260
15010110
16010110
162010110
163090980
173010110
174062260
181010110
182010000
183010210
185010110
186041840
191010110
192010110
193051150
199010000
200010210
202010000
203010210
322010110
324010000
326010110
327051650
333010000
335010110
336090980
346010110
347040840
352010000
353010210
354010210
355010210
356010210
357010210
358010210
359010210
481010210
482010210
643051050
649010110
650010000
651010210
801090980
810010110
811010000
812010210
<table>
<thead>
<tr>
<th>Numbers</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>9610000000040022</td>
<td>Node 1</td>
</tr>
<tr>
<td>999000000050032</td>
<td>Node 2</td>
</tr>
<tr>
<td>1003000000060002</td>
<td>Node 3</td>
</tr>
<tr>
<td>9620003700070002</td>
<td>Node 4</td>
</tr>
<tr>
<td>1000000300080002</td>
<td>Node 5</td>
</tr>
<tr>
<td>1004000300090002</td>
<td>Node 6</td>
</tr>
<tr>
<td>963003500100002</td>
<td>Node 7</td>
</tr>
<tr>
<td>1001000000000002</td>
<td>Node 8</td>
</tr>
<tr>
<td>1005000000000002</td>
<td>Node 9</td>
</tr>
<tr>
<td>964003300110002</td>
<td>Node 10</td>
</tr>
<tr>
<td>965000000120002</td>
<td>Node 11</td>
</tr>
<tr>
<td>966003000130002</td>
<td>Node 12</td>
</tr>
<tr>
<td>967000000140002</td>
<td>Node 13</td>
</tr>
<tr>
<td>968000300160152</td>
<td>Node 14</td>
</tr>
<tr>
<td>971002400170002</td>
<td>Node 15</td>
</tr>
<tr>
<td>969000000000002</td>
<td>Node 16</td>
</tr>
<tr>
<td>972000000180002</td>
<td>Node 17</td>
</tr>
<tr>
<td>973000700200192</td>
<td>Node 18</td>
</tr>
<tr>
<td>980001400220002</td>
<td>Node 19</td>
</tr>
<tr>
<td>974000400230212</td>
<td>Node 20</td>
</tr>
<tr>
<td>978000000000002</td>
<td>Node 21</td>
</tr>
<tr>
<td>981000000250002</td>
<td>Node 22</td>
</tr>
<tr>
<td>9750000000000242</td>
<td>Node 23</td>
</tr>
<tr>
<td>976000000000002</td>
<td>Node 24</td>
</tr>
<tr>
<td>982001100260002</td>
<td>Node 25</td>
</tr>
<tr>
<td>9830000000000272</td>
<td>Node 26</td>
</tr>
<tr>
<td>984000000280002</td>
<td>Node 27</td>
</tr>
<tr>
<td>985000700290002</td>
<td>Node 28</td>
</tr>
<tr>
<td>986000000030002</td>
<td>Node 29</td>
</tr>
<tr>
<td>987000400310002</td>
<td>Node 30</td>
</tr>
<tr>
<td>9880000000000322</td>
<td>Node 31</td>
</tr>
<tr>
<td>989000000000002</td>
<td>Node 32</td>
</tr>
</tbody>
</table>
A LISP INTERPRETER: SCANNER AND PARSER

by

DAVID CLARENCE BOSSERMAN

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

AN ABSTRACT OF 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
A LISP Interpreter is a program which interprets strings. This paper describes the design and format of three parts of a LISP Interpreter. The parts included are the reader, scanner and parser.

All parts are written in the FORTRAN language and have been tested on the INTERDATA 8/32 computer. The reader module reads the user's program into storage. The scanner modules convert the user's program into numeric tokens which are used as pointers by the other modules. The interpreter modules: (1) construct a general tree representation of the user's program; and (2) execute the user's program. The execution modules have not been completed and are to be the subject of a future report. The code for all modules completed and discussed as a part of this report are included in Appendices to the report.

The modules completed are to be integrated with memory management modules and execution modules to form an efficient high level language interpreter for use by minicomputers. As such, the read, scan and interpreter modules are designed for maximum portability with minimum adaptation.