INFORMAL VERIFICATION OF CORRECTNESS OF THE
SCANNER MODULE OF AN INTERPRETER PROGRAM

by

JAMES NOEL JONES

B. S., Oklahoma State University, 1965

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

1975

Approved by:

[Signature]
Major Professor
# TABLE OF CONTENTS

Chapter 1 General

1.1 Purpose of the Report 6

1.2 General Background of the Interpreter Program 6

Chapter 2 Specifications and Data Formats 3

2.1 General 8

2.1.1 HEAP Storage Area Specifications 8

2.1.2 Global Assertions 12

2.2 Scanner Specifications 15

2.2.1 General 15

2.2.2 Calls to the Scanner Module 17

(2.1) Input Assertions 17

(2.2) Output Assertions 19

2.2.3 Local Data Structures 19

(3.1) Character Table 19

(3.2) Keyword Table and Command Language Table 22

(3.3) Line Stack 22

(3.4) Symbol Stack 22

(3.5) Error Stack 22

(3.6) Symbol 25

(3.7) RVALUE 25

2.2.4 Required Calls 25

2.2.5 Terminal Communication to User 25
2.2.6 Language Restrictions

2.3 Global Procedures

2.3.1 General

2.3.2 Routine Specifications and Assertions

(2.1) Routine GET

(2.2) Routine EXPAND

(2.3) Routine SYMTAB

(2.4) Routine ERRPRT

(2.5) Routine STAX

(2.6) Routine GETCHR

(2.7) Routine PUTCHR

Chapter 3 Assertion Refinement and High Level Design

3.1 General

3.1.1 Assertion Refinement

3.1.2 Notation

3.2 Global Assertion Refinement

3.3 Scanner Assertions

3.3.1 Input Assertions

3.3.2 Output Assertions

3.3.3 Table, Counter, and Flag Initialization and Variable Definition

3.4 Global Procedure Assertion Refinement

3.5 High Level Design Language with Assertions

3.5.1 SUBROUTINE SCAN

3.5.2 SUBROUTINE LNSCAN

3.5.3 SUBROUTINE FORM
<table>
<thead>
<tr>
<th>FIGURE</th>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>2-1</td>
<td>9</td>
</tr>
<tr>
<td>2-2</td>
<td>10</td>
</tr>
<tr>
<td>2-3</td>
<td>11</td>
</tr>
<tr>
<td>2-4</td>
<td>13</td>
</tr>
<tr>
<td>2-5</td>
<td>14</td>
</tr>
<tr>
<td>2-6</td>
<td>18</td>
</tr>
<tr>
<td>2-7</td>
<td>20</td>
</tr>
<tr>
<td>2-8</td>
<td>21</td>
</tr>
<tr>
<td>2-9</td>
<td>23</td>
</tr>
<tr>
<td>2-10</td>
<td>24</td>
</tr>
<tr>
<td>2-11</td>
<td>26</td>
</tr>
<tr>
<td>4-1</td>
<td>87</td>
</tr>
<tr>
<td>4-2</td>
<td>94</td>
</tr>
<tr>
<td>4-3</td>
<td>97</td>
</tr>
<tr>
<td>4-4</td>
<td>101</td>
</tr>
<tr>
<td>4-5</td>
<td>104</td>
</tr>
<tr>
<td>4-6</td>
<td>106</td>
</tr>
<tr>
<td>4-7</td>
<td>108</td>
</tr>
<tr>
<td>C1-1</td>
<td>126</td>
</tr>
</tbody>
</table>
Chapter 1
General

1.1 Purpose of the Report

The purpose of this report is to provide a documented, informal verification of correctness of the Scanner module of an interpreter program. The verification follows three levels of program development: (1) that of English specifications and assertions, (2) high level design language, and (3) FORTRAN code. Assertions are developed from the Scanner module's specifications and are refined in parallel with program development. Verification of correctness is presented for the program based on the refined assertions. The scope of the verification will include the Scanner subroutine, all subroutines developed directly from the Scanner module, and all external routines used by the Scanner.

1.2 General Background of the Interpreter Program

The interpreter program of which the Scanner module is a portion was designed and implemented by the students of CS 286-700, summer 1975 session. The program was divided into the eleven following student groups:

(1) Interpreter Driver
(2) Scanner
(3) Text Editor
(4) Top Down Parser
(5) Bottom Up Parser
(6) Command Line Interpreter
(7) Operation Functions
(8) Heap Maintenance
(9) Stack Functions
Each of the above areas were designed independently using a modular programming approach. Module interaction was coordinated based on module specifications, and in some cases including input and output assertions. The program was intended to provide a language interpreter to facilitate a new programming design language which closely resembles APL. The interpreter program was developed to use a standard FORTRAN compiler to facilitate transportability and maintenance. Further documentation and background information can be obtained from the reference "The Language and Program Documentation of a Student Designed Interpreter", included in Annex A.
Chapter 2
Specifications and Data Formats

2.1 General

The modular construction of the interpreter program is shown in figure 2-1. Conceptionally a source text being entered into the language interpreter program can be either command language instructions or a procedure to be processed. After entry of the source text into the program, the Scanner module is called to develop a token representation of the text that can be more easily processed by the other modules of the interpreter program. A common storage area called HEAP is used to facilitate passage of data between modules.

2.1.1 HEAP Storage Area Specifications

The HEAP storage area, figure 2-2, is a series of contiguous storage locations. Assignment of unique blocks of storage space within the HEAP area for specific types of data is made on a dynamic basis. The dynamic characteristic of the HEAP storage area is achieved by the use of indirect addressing and index values which represent the displacement of a block of storage locations from the start of the HEAP area. An Address Translation Table, figure 2-3, consisting of storage locations at the head of the HEAP area is used to provide indirect addresses to specific data block areas. These indirect address locations are referred to as logical addresses throughout the remainder of this report. The value stored in a logical address is an absolute address that represents the displacement of a data block from the
ILLEGIBLE DOCUMENT

THE FOLLOWING DOCUMENT(S) IS OF POOR LEGIBILITY IN THE ORIGINAL

THIS IS THE BEST COPY AVAILABLE
Figure 2-1 Interpreter Program Hierarchical Chart
HEAP(1) is the first storage word in the HEAP

Address Translation Table

- Procedure Table Block
- Text Block
- Token Block
- Symbol Table Block
- Code Block
- Temporary Block

HEAP(n) is the last storage word in the HEAP

Figure 2-2  HEAP Storage Area Contents
THIS BOOK CONTAINS NUMEROUS PAGES WITH DIAGRAMS THAT ARE CROOKED COMPARED TO THE REST OF THE INFORMATION ON THE PAGE. THIS IS AS RECEIVED FROM CUSTOMER.
ADTRAN = Address Translation Table
(sample length = 50)

Figure 2-3 Format for Address Translation Table
start of the HEAP to the location prior to the beginning of the indexed data block area. A Procedure Table data block, figure 2-4, is established and contains individual Procedure Tables for each source text that is translated by the interpreter program. The individual Procedure Tables are used to store the values of the logical addresses of the specific data blocks associated with that procedure.

The HEAP storage area and each common data structure used by more than one module of the program has a common header format and specific variable name offsets associated with key positions of the format, figure 2-5. Additional variable name offsets are associated with specific locations of like type data blocks. For assigned data blocks within the HEAP storage area, access to a unique formatted location is made by adding the absolute address to the variable name offset for the desired location. More detailed information to include formats and variable name offsets will be given in subsequent sections of this report, for each type data block area accessed by the Scanner module.

The FORTRAN initialization data for all HEAP storage area offsets by type data block is at Annex B.

2.1.2 Global Assertions

A number of assertions concerning particular data structures are common to all modules of the language interpreter program. These assertions have been designated as global and are necessary for development of specific input and output assertions for each module and submodule of the interpreter program. Further refinement of these assertions
PRCTAB = Procedure Table

HEAP(5)

location in Address Translation Table

Space_Use = header + n PRCLEN

header

PRCLEN = length of Procedure Table (8 words)

PINDEX

is relative index for any particular Procedure Table

PRCNLN

length of name in characters

PRCNAM

name of the procedure

PRCSTAT

status field

PROCOD

logical address of Code Block

PRCSYM

logical address of Symbol Table Block

PRCTOK

logical address of the Token Block

PRCTX

logical address of Text Block

eg: typical access to a specific Procedure Table by
    HEAP(Absolute_Address + PINDEX + offset)

* Offset and length constants are underlined.

Figure 2-4 Format for Procedure Table Block
HEAP (i)  

i is called the logical address location in the Address Translation Table  

j is called the absolute address  

HPTAG  

HPSPAC  

HPSIZ  

HPLAD  

these are called the header  

logical block i starts at HEAP location j+1  

HPLEN = length of header  

space used by object  

space allocated but not yet used  

first_used_index = j+HPTAG  

last_index = j+space  

last_used_index = j+size  

* Offset constants are underlined.  

Figure 2-5  Header Format for an Allocated Block in HEAP
will be presented in conjunction with those assertions which are unique to the Scanner module. The following assertions are considered global:

(2.1) That the HEAP storage area is established and conforms to prescribed formats, figures 2-2 and 2-3.

(2.2) That the fifth storage location of the HEAP contains the absolute address of the Procedure Table data block area.

(2.3) That all variable name offsets are initialized according to format.

(2.4) That the Procedure Table data block area is established and conforms to prescribed formats, figure 2-4.

(2.5) That PINDEX is an integer value that represents the displacement from the start of the Procedure Table data block to the location prior to the start of a specific Procedure Table.

2.2 Scanner Specifications

2.2.1 General

The Scanner routine is a module of a language interpreter program that is called either by the program Driver module or by the Command Language module, figure 2-1. The Scanner module evaluates characters of a source text by line of the text and constructs program symbols that consist of identifiers, key or command words, numbers, operators, separators, strings, and undefined operators. A three word token consisting of a symbol class, index or value, and a pointer to the first character of the symbol in the line of text is then associated with each recognized symbol. Tokens are then stored in a designated area by line number which
corresponds to the source text line number. Each source text line that is scanned and found to be error free is annotated as having been scanned in the appropriate text line header word. Token storage information is updated in the appropriate token storage header words after each text line is scanned and stored.

Once called, the Scanner subroutine will not terminate until each line of the source text has been scanned unless a supervisor interrupt is received or there is insufficient storage space available for the token data. A check is made for a supervisor interrupt after each line has been scanned and the tokens have been stored. If an interrupt is received, control is returned to the calling program with a distinctive error return code. A test is made for adequate token storage space after each line has been scanned and the text words and symbols identified. An error code indicating insufficient storage space available is sent to the calling program if the Scanner module is unable to store the token data.

If an error is detected in a line of text, a message corresponding to the type of error encountered is associated with the text line. Token line header information is assigned, but no tokens are generated for the symbols of the text line. When all lines of the source text have been scanned, error messages are printed for each line that an error was found, indicating the type of error and printing that portion of the line where the error was encountered.

Control is then returned to the calling module with an error code indicating either errors were or were not
encountered with the source text. For each error free line of source text that was scanned, there exists a corresponding line of token representations. If no errors were encountered in the source text, a complete token representation of the source text is stored in the token storage area for the process being interpreted.

2.2.2 Calls to the Scanner Module

Calls to the Scanner module include one input parameter, a PINDEX to a specific Procedure Table, and one output parameter, an error indicator code. The input parameter allows the Scanner routine to gain access to the desired Procedure Table, and thus to the source text for the procedure being interpreted.

(2.1) Input Assertions

For the Scanner module to function properly, certain conditions must be in effect concerning specific variables and data structures. These are considered input assertions to the Scanner module and hold for all submodules developed by the Scanner.

(2.1.1) Global Assertions

That the global assertions as stated in paragraph 2.1.2 are valid.

(2.1.2) PINDEX

That PINDEX is the displacement to the desired Procedure Table which contains a logical address of a Text data block.

(2.1.3) Text Data Block

That the Text data block, figure 2-6, is established and contains the desired source text.
HEAP(j)

location in the Address Translation Table

nlin is lines of text all with the same format

TXTTOP
TXTBOT
TXTLN

TXTLN is the line number

lnum flk blk tok nst sta len variable length

TXTFLK
TXTBLK

relative index within the block of the next (flk) or previous (blk) text line
flk ≠ forward link
blk ≠ backward link
flk = 0 for first line
blk = 0 for last line

TXTLN first word of source text

TXTLN characters in text line

TXTSTA status flag

TXTNST nesting level

TXTTOK flag indicates if line has been scanned
0 = line not scanned
1 = line has been scanned (set by scanner)

bot is relative index of last line entry in the Text Block (may not be physically the last line)
top is the relative index of the first line entry in the Text Block (may not be physically the first line in the block)

* Offset constants are underlined.

Figure 2-6 Format for Text Block
(2.1.4) Variable Offsets

That the variable offsets associated with the Procedure Table data block, Text data block, and Token data block are initialized.

(2.2) Output Assertions

The conditions which must be in effect on termination of execution of the Scanner module are considered output assertions. These assertions hold in accordance with the Scanner specifications for all program terminations of the module.

(2.2.1) Error Code

That the error parameter for all normal terminations indicates if there were syntax errors in the lines of source text. That the error parameter will indicate if an abnormal termination of the module's execution was caused by an user interrupt or for lack of HEAP storage space.

(2.2.2) Token Data Block

That a Token data block, figure 2-7, is established and contains token data as per the Scanner specifications, and that the logical address to the data block is assigned to the appropriate location of the Procedure Table referenced by the PINDEX.

2.2.3 Local Data Structures

The following data structures are initialized for use in the Scanner module and are available to all submodules of the Scanner.

(3.1) Character Table (255,3) array

This table is initialized in accordance with figure 2-8 to correspond to the character set selected for use with the
HEAP(j) location in the Address Translation Table

- **TOKNLIN** is total number of token lines
- **TOKBOT** is displacement next open word
- **TOKTOP** is displacement to first token line (may not be physically the first line)
- **TOKLNUM** is the line number
- **TOKFLK** is displacement to next line from start of block
- **TOKNUM** is number of tokens in the line

- **nfc** is the character number in the line of text where the symbol starts.
- **index** is the index into a table for classes which refer to tables or it is a value for symbols which are numbers or strings.
- **class** is code which indicates the type of symbol:
  - 1 = keyword or command language word
  - 2 = identifier
  - 3 = integer number
  - 4 = real number
  - 5 = string
  - 6 = operator
  - 7 = separator
  - 8 = undefined operator

* Offset constants are underlined.

**Figure 2-7** Format for Token Block
<table>
<thead>
<tr>
<th>Column 1</th>
<th>Column 2</th>
<th>Column 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 2 3 4</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>255</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Column 3** - Column 3 contains the external integer index values for the separators and operators based on their sequence in the machine code translation table.

**Column 2** - Column 2 contains integer values 0, 1, 2, 3, 4, 6, 7 which designate the character as:

- 0 = blank
- 1 = letter
- 2 = number
- 3 = $ 
- 4 = " 
- 6 = operator
- 7 = separator

Integer value is used as a condition code for forming specific symbols. Integer value is external class code for these characters.

**Column 1** - Column 1 contains integer values 1, 2, 3 which designate the character as:

- 1 = blank
- 2 = separator or operator
- 3 = letter or number

Value is used as a condition code.

**Row** - The total number of rows is determined by the machine code character translation table. A specific character is represented in the table by the row that corresponds to its decimal machine code number.

*See Annex B for initialization data used with FORTRAN code.*

**Figure 2-8** Character Table Format
interpreter program. Different character sets may be used with the program based on initialization of this table and if the character set does not exceed machine internal representation of 255 decimal.

(3.2) Keyword Table (10,7) array and Command Language Table (10,7) array

These arrays are initialized in accordance with figures 2-9 and 2-10 with the designated Keywords and Command Language words of the design language. Table initialization is made based on the first four characters of each word. Words are placed in table columns according to the total number of characters in each word. Table searches for comparison with a symbol is made only against the column which represents the corresponding number of characters in the symbol.

(3.3) Line Stack (64) array

This array is initialized to zero and then used to hold the characters of a line of source text for each line of text while being processed by the Scanner module.

(3.4) Symbol Stack (15,3) array

This array is initialized to zero and then used to temporarily hold the generated tokens for the symbols of a line of source text for each line of source text processed by the Scanner module prior to placement of the tokens in the Token data block.

(3.5) Error Stack (64,3) array

This array is initialized to zero and then used to hold error information associated with those lines of source text where syntax errors were encountered. Each row of the array
<table>
<thead>
<tr>
<th>Row</th>
<th>DO</th>
<th>END</th>
<th>CASE</th>
<th>BEGIN</th>
<th>ACCESS</th>
<th>ENPROC</th>
<th>ENDWRITE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>IF</td>
<td>OUT</td>
<td>ELSE</td>
<td>FALSE</td>
<td>EXPORT</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>IN</td>
<td>0</td>
<td>EXIT</td>
<td>WHILE</td>
<td>GLOBAL</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>FI</td>
<td>0</td>
<td>GOTO</td>
<td>WRITE</td>
<td>RETURN</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>0</td>
<td>PROC</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>0</td>
<td>READ</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>0</td>
<td>0</td>
<td>THEN</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>0</td>
<td>0</td>
<td>TRUE</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>9</td>
<td>0</td>
<td>0</td>
<td>QUIT</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>0</td>
<td>CALL</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Keywords were placed in columns based on the total number of characters in the word. Limitation was placed on the language in that a keyword may not exceed eight but must have a minimum of two characters.

The total number of rows of the table was dependent on the largest number of keywords with the same number of characters.

The index value of a keyword is determined by multiplying the column number by 10 and adding the row number to that total.

* See Annex B for initialization data for the FORTRAN code.

** The FORTRAN initialization stores only the first four letters of each word.

Figure 2-9 Keyword Table Format
<table>
<thead>
<tr>
<th>Row</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>ON</td>
<td>FNS</td>
<td>CHAR</td>
<td>CLEAR</td>
<td>DIGITS</td>
<td>BREAK-IT</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>NO</td>
<td>LIB</td>
<td>COPY</td>
<td>ERASE</td>
<td>RESUME</td>
<td>NOTRACE</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>OFF</td>
<td>DROP</td>
<td>HENCE</td>
<td>VALUES</td>
<td>SUSPEND</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>POP</td>
<td>EDIT</td>
<td>LINES</td>
<td>CLRSTK</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>RUN</td>
<td>HELP</td>
<td>PARSE</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>VAR</td>
<td>LIST</td>
<td>STACK</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>0</td>
<td>0</td>
<td>LOAD</td>
<td>TRACE</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>0</td>
<td>0</td>
<td>SAVE</td>
<td>WIDTH</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>9</td>
<td>0</td>
<td>0</td>
<td>VARS</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>0</td>
<td>WSID</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

**Command language words were placed in columns based on the total number of characters in the word. Limitation was placed on the language in that a command language word could not exceed eight but must have a minimum of two characters.**

**The total number of rows of the table was dependent on the largest number of command language words with the same number of characters.**

**The index value of a command language word is determined by multiplying the column number by 10 and adding the row number to that total.**

* See Annex B for initialization data for the FORTRAN code

** The FORTRAN initialization stores only the first four letters of the word.

Figure 2-10 Command Language Table Format
would contain the line number, code for the type of error, and a pointer to where the error was detected in the source line.

(3.6) Symbol (2) array

This array is initialized to zero and then is used to pack and hold characters as a symbol is being formed by the Scanner module.

(3.7) RVALUE (15) array

This array is used to hold the value of real numbers formed by the Scanner module.

2.2.4 Required Calls

This section is for reference only to provide full specification documentation of the Scanner module. Complete specifications of each of the subroutines referenced below is provided in other sections of this report.

(4.1) Calls to other Modules

(4.1.1) HEAP Module - to subroutines GET and EXPAND

(4.1.2) Driver Module - to subroutines ERRPRT, STAX, GETCHR, and PUTCHR.

(4.2) Calls to Submodules (figure 2-11)

2.2.5 Terminal Communication to User

Communication to the user through the terminal is provided by the ERRPRT Subroutine. For each line of source text that a syntax error is found, an error message is printed to the user indicating the line number of the error, type of error, and by printing that portion of the text line where the error was encountered.

2.2.6 Language Restrictions
Dash line represents calls to external subroutines of the SCANNER module.

Figure 2-11 SCANNER Module Hierarchical Chart
The implementation of the Scanner module imposed some restrictions on the syntax of the language developed for use with the interpreter program. These restrictions are considered internal assertions for the recognition of classes of symbols and error conditions within the source text. The language restrictions are as follows:

(6.1) Symbol Length

All classes of symbols except strings are limited to a length of eight characters.

(6.2) Keywords

Keywords are restricted to those words identified and listed in the Keyword Table. Words may be added to the table, but only by modification of the initialization data of the program.

(6.3) Command Language Words

Command language words are restricted to those words identified and listed in the Command Language Table. Words may be added to the table, but only by modification of the initialization data of the program.

(6.4) Command Language Lines

The first character of all command language lines of text must be a closing parenthesis. Command lines are processed in the same manner as program lines of a source text. No special designation is made by the Scanner module to specifically identify command lines to the routine initiating the call to the Scanner.

(6.5) Operators

Operators must be one of those characters identified as
an operator in the initialization data for the program, or if previously undefined, any character or series of up to seven characters preceded by a $.

(6.6) Strings

All strings must be preceded by and terminated by a double quote mark. Strings may be of any length and may contain any combination of characters except a double quote mark that does not exceed beyond the limits of one line of source text.

(6.7) Line Continuation

There are no provisions within the Scanner module to allow for continuations of symbols or strings between lines of source text.

(6.8) Real Values

All real values must contain a decimal point and the first character must be a digit.

(6.9) Line Length

A single line of source text may not contain more than a total of 64 characters without modification of the Scanner module.

(6.10) Separations

All identifiers, keywords, command language words, values, or strings must be separated by either an operator, separator, or blank.

(6.11) Characters

Characters are limited to those of the character code table used to initialize the character table of the Scanner module. Character codes may be changed by modification of the
initialization data of the Scanner character table.

2.3 Global Procedures

2.3.1 General

A number of subroutines exist within the overall confines of the interpreter program that perform specific functions for more than one module, or are used to pass specific data information between modules. These subroutines are designated as global routines and may be called by any level of the interpreter program.

2.3.2 Routine Specifications and Assertions

(2.1) GET

Routine GET is a subroutine of the HEAP storage area module. It provides interaction between other modules of the interpreter program and the HEAP module for allocation of storage space.

(2.1.1) Specification

When called, routine GET will search the HEAP storage area for a block of contiguous locations that will meet the size requirements of the calling module. If unable to find a block of locations of the required size, the routine will generate a compaction of data in the HEAP storage area. If an area is still not available to meet the calling module's requirements, an error code will be generated. When an area has been found that will meet the required size, the first word of the block is assigned a code based on the type of data to be stored, the second word is assigned the value of the total number of words allocated in the storage block, and the forth word is assigned the value of the logical address.
location that contains the index value to the allocated storage block. The value that represents the displacement from the first location of the HEAP storage area to the requested storage block is then assigned to the logical address location.

(2.1.2) Assertions

The assertions for this routine are based on the parameters in the calling statement for the subroutine. These parameters are:

SIZE - the number of words of storage area required.
TYPE - the type of information to be stored.
LADDR - the logical address which contains the index to the data block assigned.
ERROR - error code indicating if the required area is or is not available.

(2.1.2.1) Input Assertions

That the parameter SIZE is an integer value that is greater than zero.

That the parameter TYPE is an integer value.

That the global assertions are initialized and correct.

(2.1.2.2) Output Assertions

That the HEAP storage area was scrutinized for an area to meet the required size.

That if an area was found to meet the required size that the header information was assigned according to format and the displacement pointer value was assigned to a logical address.

That the parameter LADDR is an integer and is the value
of the logical address that contains the value of the displacement index to the requested storage area.

That the parameter ERROR is an integer and of the value zero if no error was encountered or positive if the required storage area is not available.

(2.2) EXPAND

Routine EXPAND is a subroutine of the HEAP storage area module. It provides interaction between other modules of the interpreter program and the HEAP module for expanding the amount of allocated storage space to the calling module in the HEAP storage area.

(2.2.1) Specifications

When called, routine EXPAND will determine if sufficient free words exist adjacent to the previously allocated storage area to satisfy the additional request for space. If not, the routine initiates a compaction of the storage area and then checks to see if sufficient space is available. If the additional space is allocated, the return error code is zero, but if insufficient space is available, an error code is returned. The offset to the new allocated storage area is assigned to the logical address location. All data previously stored in the allocated storage space is transferred to the new area.

(2.2.2) Assertions

The assertions for this routine are based on the parameters in the calling statement for the subroutine. These parameters are:

SIZE - the number of additional words of storage space required.
LADDR - the logical address location that contains the displacement to the storage space that is to be expanded.

ERROR - error code if required space is not available.

(2.2.2.1) Input Assertions

That the parameter SIZE is an integer value that is greater than zero.

That the parameter LADDR is an integer value that is the logical address for the area to be expanded.

That the global assertions are initialized and correct.

(2.2.2.2) Output Assertions

That the HEAP storage area was scrutinized for an area to meet the additional space requirements of the calling routine.

That if an area was found to meet the new size requirements all previously stored data was transferred to the new location.

That the value in the logical address was changed to reflect the displacement to the new storage area.

That the parameter ERROR is an integer value, either zero or positive. If zero, area was available, and if positive was not available.

(2.3) SYMTAB

Routine SYMTAB provides interaction between other modules of the interpreter program and the Symbol Table block which indexes all identifiers by individual procedure.

(2.3.1) Specifications

When called, subroutine SYMTAB associates an identifier
formed by the Scanner module with a specific procedure. The subroutine then inspects the symbol table for that procedure to see if the identifier was previously indexed. If the identifier is indexed, that value is returned to the calling module. If the identifier is not in the symbol table, it is added and the new index is returned to the calling module. An error code is returned if the identifier cannot be stored.

(2.3.2) Assertions

The assertions for this routine are based on the parameters in the calling statement for the subroutine. These parameters are:

PINDEX - the displacement from the start of the Procedure Table block area to the specific Procedure Table desired.

SYML - the length of the identifier in number of characters in the identifier.

SYM - a two word array that contains the symbol with characters packed four per word.

INDEX - the value which represents the index of the identifier in the Symbol Table for the procedure.

ERROR - a value which indicates if an error occurred during the execution of the routine that prevented the identifier from being found or stored.

(2.3.2.1) Input Assertions

That the parameter PINDEX is an integer value that is the displacement index from the start of the Procedure Table data block area to the Procedure Table desired.

That the parameter SYML is an integer value greater than zero but equal to or less than eight.

That the parameter SYM is a two word array that contains
the characters of the identifier in question, with the characters packed four per word.

That the global assertions are initialized and correct.

(2.3.2.2) Output Assertions

That the parameter INDEX is an integer value that points to the identifier being processed in the symbol table for the procedure being scrutinized.

That the parameter ERROR is an integer value either zero or one. If zero, the identifier has been placed in the symbol table for the procedure identified by the PINDEX at a location represented by the value of INDEX. If one, an error was encountered during execution of the routine and the identifier was not processed.

(2.4) ERRPRT

Routine ERRPRT is a subroutine of the Driver module. It provides a common error print routine for all modules of the interpreter program.

(2.4.1) Specifications

When called, subroutine ERRPRT will print a message on the terminal indicating the procedure in which the error was found, the line number, an error message as to type of error, and identify the portion of the line with the symbol where the error was detected. The routine returns a code to the calling module indicating if the error print sequence was or was not completed.

(2.4.2) Assertions

The assertions for this routine are based on the parameter in the calling statement of the subroutine. These
parameters are:

PINDEX - the displacement from the start of the Procedure Table data block area to the desired Procedure Table.

LINE - the line number in the program source text where the error was detected.

CHAR - the number of the characters in the line of text at which the error was detected.

N - the number of characters in the error message to be printed.

STRING - a string containing the message to be printed.

RETCOD - a code indicating if the message was or was not printed.

(2.4.2.1) Input Assertions

That the parameter PINDEX is an integer value that is the displacement from the start of the Procedure Table block area to the specific Procedure Table desired.

That the parameter LINE is an integer value greater than zero and is an element of the set composed of the line numbers of the program.

That the parameter CHAR is an integer value greater than zero and is an element of the set composed of the number of characters in the line of text.

That the parameter N is an integer value representing the total characters in the error message to be printed.

That the parameter STRING is an one dimensional array of which contains the characters of the error message to be printed with characters packed four per word.

That the global assertions are initialized and correct.
(2.4.2.2) Output Assertion

That the parameter RETCOD is an integer value with zero representing no error was encountered while printing the error message, or positive indicating that the message could not be printed.

(2.5) STAX

Routine STAX is a subroutine of the Driver module. It provides interaction between the interpreter program user and all modules of the program by allowing for user interrupts. Calls to this routine are placed in modules to allow for user interrupts after logical completion of key steps of module operations.

(2.5.1) Specifications

When called, subroutine STAX allows the program user to interrupt the operation of the module which initiated the call. Interrupts are indicated by assignment of values to the two parameters in the calling statement. When a value is assigned to the parameter indicating an user interrupt, the calling module assigns a distinctive code to its error output parameter and returns control to the Driver module.

(2.5.2) Assertions

The assertions for this routine are based on the parameters in the calling statement of the subroutine. These parameters are:

SWITCH – the change in value of this parameter indicates if an interrupt was received.

DELAY – the value of this parameter is the time in milliseconds, required for the interrupt analysis.
(2.5.2.1) Input Assertions

That the value of parameter SWITCH is integer zero.
That the value of parameter DELAY is integer 150.

(2.5.2.2) Output Assertion

That if the value of parameter SWITCH is greater than zero, an user interrupt was received. If the value of parameter SWITCH remains zero, no interrupt was received.

(2.6) GETCHR

Routine GETCHR is a subroutine of the Driver module. It allows modules of the interpreter program to extract an individual character from strings which are packed with four characters per word.

(2.6.1) Specifications

When called, routine GETCHR takes one character from a string based on the number of the character as specified in the call statement. The character extracted from the packed word is then returned to the calling program as an output parameter. If the value of the character number exceeds a value of four, the routine will then select the next sequential word and extract the characters of that word.

(2.6.2) Assertions

The assertions for this routine are based on the parameters in the calling statement of the subroutine. These parameters are:

STRG - the string in which the packed characters are contained.
CHARNUM - the sequential number of the character to be extracted.
ARG3 - the character that was extracted from the packed word.
(2.6.2.1) Input Assertions
    That the parameter STRG contains packed characters.
    That the parameter CHARNUM is an integer value greater
    than zero.
    That sequential words to the storage string listed in the
    call statement contain packed characters.
(2.6.2.2) Output Assertion
    That the parameter ARG3 contains the character extracted
    from the string, right adjusted as an integer.
(2.7) PUTCHR
    Routine PUTCHR is a subroutine of the Driver module. It
    allows modules of the interpreter program to pack characters
    into a string, one character at a time.
(2.7.1) Specification
    When called, routine PUTCHR packs a character in a string
    of packed characters. The position in the string that the
    character is packed is based on the character number as speci-
    fied in the call statement.
(2.7.2) Assertions
    The assertions for this routine are based on the parma-
    meters in the calling statement of the subroutine. These
    parameters are:
    SYM - an array that is used to pack characters. The Scanner
    module uses a two word array while other modules may
    use larger arrays.
    K - a counter that indicates the sequential number of the
    character and the position it is to be packed in the
    two word array SYM.
    CHAR - the characters to be packed, right adjusted.
(2.7.2.1) Input Assertions

That the parameter K is an integer value and for the Scanner module is of the set 1 to 8.

That the parameter CHAR contains a character to be packed, and it is left adjusted in a four byte word.

(2.7.2.2.) Output Assertion

That the two word array SYM is packed with the character in the position indicated by the input parameter K.
Chapter 3
Assertion Refinement and High Level Design

3.1 General

3.1.1 Assertion Refinement

The assertions listed in this chapter represent the refinement of those assertions stated in Chapter 2 of this report. Assertion refinement is necessary in order to further restrict and qualify common data structures and parameter conditions required for program development. Refinement is made to approximately the same level of abstraction as the high level design language representation of the Scanner module. The assertion refinement also is used to define and qualify specific variables used in the high level design language representation. The assertions listed are common in all Scanner module routines and hold under all conditions.

3.1.2 Notation

The notation used throughout this chapter is defined below.

(2.1) Absolute Address

Any variable name followed by a "@" character is defined as the absolute address of that variable in the HEAP storage area.

(2.2) Variable Names

Each variable name developed in the assertion refinement is defined in terms of a predicate. The variable name predicate is illustrated by use of all capital letters and multiple
word names being joined by a "-". The variable value of the
predicate is represented by only the first letter of each
word being capitalized.

(2.3) Definitions

The symbol used to indicate the definition of a term is
a "=*
.

A period separating two predicates is used to indicate
the block area to which the predicate being defined belongs.

A variable enclosed by parentheses indicates the value
of the variable.

The word "is" is used to associate a variable with a
particular predicate.

The word "means" is used to define a predicate. The
predicate may consist of several separate conditions which
are linked together by the word "and" or the predicate may
be conditional which is designated by the word "or"

3.2 Global Assertion Refinement

The refined global assertions listed below provide the
bases for all other assertions and are used to construct and
qualify data structures, parameters, and variables used in
the Scanner module.

3.2.1 HEAP Storage Area

HEAP means HEAP is a linear array of 1 to n length,
available for dynamic assignment into block areas, as in
figure 2-2.

3.2.2 Variable Name Offsets

For the HEAP Storage Block starting at absolute
address 1:
HEAP . HPTAG ≠ HEAP(i+1)
HEAP . HPSPAC ≠ HEAP(i+2)
HEAP . HPSIZ ≠ HEAP(i+3)
HEAP . HPLAD ≠ HEAP(i+4)
HEAP . HPOBJ ≠ HEAP(i+5)

and

I is HEAP . LOG_ADD means
5 ≤ I ≤ HEAP(HPSIZ)

and

I@ is Absolute_Address means
j is Log_Add and I@ ≈ HEAP(j)
and HEAP(HPSIZ) < I@ < n

3.2.3 Procedure Table Block Address
HEAP(HPOBJ) ⇔ HEAP(5) and
PROC_TAB_BLOCK@ ≠ HEAP(HPOBJ)

3.2.4 Procedure Table Block
Proc_Tab_Block is HEAP . PROC_TAB_BLOCK means
Proc_Tab_Block is a List of procedure tables as
shown in figure 2-4.

and

PROC_TAB_BLOCK@ . HPTAG ≠ HEAP(Proc_Tab_Block@ + 1)
PROC_TAB_BLOCK@ . HPSPAC ≠ HEAP(Proc_Tab_Block@ + 2)
PROC_TAB_BLOCK@ . HPSIZ ≠ HEAP(Proc_Tab_Block@ + 3)
PROC_TAB_BLOCK@ . HPLAD ≠ HEAP(Proc_Tab_Block@ + 4)

and

HEAP(Proc_Tab_Block@ + HPSPAC) ≠ Allocated_Space
and

Allocated_Space is ALLOCATED_SPACE means the
the value of Allocated_Space is the number of storage words allocated to a block area by the HEAP module.

and

$\text{HEAP(Proc\_Tab\_Block@ + HPSIZ)} \neq \text{Space\_Used}$

and

$\text{Space\_Used is SPACE\_USED means}$

$4 \leq \text{Space\_Used} \leq \text{Allocated\_Space}$

and

$\text{PINDEX is PROC\_TAB\_BLOCK . PINDEX means}$

$\text{PINDEX } \in \{4, 12, 20, 28, \ldots, (\text{HEAP(Proc\_Tab\_Block@ + HPSIZ - 8)})\}$

and

$\exists$ a Procedure Table as shown in figure 2-4, starting at $\text{HEAP(Proc\_Tab\_Block@ + PINDEX + 1)}$

3.2.5 Procedure Table Address

$\text{Proc\_Tab\_Add@ is HEAP . PROC\_TAB\_ADD@ means}$

$\text{HEAP(Proc\_Tab\_Block@ + PINDEX) \neq Proc\_Tab\_Add@}$

3.2.6 Procedure Table

$\text{Proc\_Tab is PROC\_TAB\_BLOCK . PROC\_TAB means}$

$\text{Proc\_Tab is eight words as per figure 2-4}$

and

$\text{PROC\_TAB . PRCNLEN } \neq \text{HEAP(Proc\_Tab\_Add@ + 1)}$

$\text{PROC\_TAB . PRCNAM } \neq \text{HEAP(Proc\_Tab\_Add@ + 2)}$

$\text{PROC\_TAB . PRCSTAT } \neq \text{HEAP(Proc\_Tab\_Add@ + 4)}$

$\text{PROC\_TAB . PRCTXT } \neq \text{HEAP(Proc\_Tab\_Add@ + 5)}$

$\text{PROC\_TAB . PRCTOK } \neq \text{HEAP(Proc\_Tab\_Add@ + 6)}$

$\text{PROC\_TAB . PRCSYM } \neq \text{HEAP(Proc\_Tab\_Add@ + 7)}$

$\text{PROC\_TAB . PRCCOD } \neq \text{HEAP(Proc\_Tab\_Add@ + 8)}$
and

\[ \text{HEAP}(\text{Proc\_Tab\_Add@ + PRCNLEN}) \neq \text{not used by Scanner} \]
\[ \text{HEAP}(\text{Proc\_Tab\_Add@ + PRCNAM}) \neq \text{not used by Scanner} \]
\[ \text{HEAP}(\text{Proc\_Tab\_Add@ + PRCSTAT}) \neq \text{not used by Scanner} \]
\[ \text{HEAP}(\text{Proc\_Tab\_Add@ + PRCTXT}) \neq \text{Txt\_Log\_Add or nulltxt} \]

and

\[ \text{Txt\_Log\_Add is TXT\_LOG\_ADD means} \]
\[ \exists \text{ a Ttxt\_Log\_Add as per para. 3.3.1} \]

and

\[ \text{nulltxt means Source Text does not exist.} \]
\[ \text{HEAP}(\text{Proc\_Tab\_Add@ + PRCTOK}) \neq \text{Tok\_Log\_Add or nulltok} \]

and

\[ \text{Tok\_Log\_Add is TOK\_LOG\_ADD means} \]
\[ \exists \text{ a Tok\_Log\_Add as per para. 3.3.2.2} \]

and

\[ \text{nulltok means Token data block does not exist.} \]
\[ \text{HEAP}(\text{Proc\_Tab\_Add@ + PRCSYM}) \neq \text{not used by Scanner} \]
\[ \text{HEAP}(\text{Proc\_Tab\_Add@ + PRCCOD}) \neq \text{not used by Scanner} \]

3.3 **Scanner Assertions**

The Scanner module input assertions are developed in the same manner as the global assertions and are considered valid when the Scanner module is called either by the Driver module or the Command Language module. These assertions combined with all initialized tables, flags, and counters provide the necessary bases for development of the Scanner output assertions.

3.3.1 **Input Assertions**

(1.1) Global assertions are valid.
(1.2) Text Logical Address

∃ i ∃ i is TXT_LOG_ADD

Txt_Log_Add is TXT_LOG_ADD means

Txt_Log_Add is a HEAP . LOG_ADD

and

HEAP(Proc_Tab_Add@ + PROTXT) ∉ Txt_Log_Add

and

HEAP(Txt_Log>Add) ∉ Txt_Add@

and

Txt_Add@ is a TXT_ADD@

(1.3) Text Absolute Address

Txt_Add@ is TXT_ADD@ means

HEAP(HPSIZ) < Txt_Add@ < n

and

∃ a text block as per figure 2-4, starting at

HEAP(Txt_Add@ + 1)

(1.4) Text Block

Txt_Block is HEAP . TXT_BLOCK means

Txt_Block is a Linked_List as shown in figures 2-4

and 2-5.

and

TXT_BLOCK . HPTAG ∉ HEAP(Txt_Add@ + 1)

TXT_BLOCK . HPSPAC ∉ HEAP(Txt_Add@ + 2)

TXT_BLOCK . HPSIZ ∉ HEAP(Txt_Add@ + 3)

TXT_BLOCK . HPLAD ∉ HEAP(Txt_Add@ + 4)

TXT_BLOCK . TXTTOP ∉ HEAP(Txt_Add@ + 5)

TXT_BLOCK . TXTBOT ∉ HEAP(Txt_Add@ + 6)

TXT_BLOCK . TXTNLIN ∉ HEAP(Txt_Add@ + 7)
and
HEAP(Txt_Add@ + HPSIZ) \triangleq Space\_Used
HEAP(Txt_Add@ + HPLAD) \triangleq Txt\_Log\_Add
HEAP(Txt_Add@ + TXTTOP) \triangleq \text{1st}\Txt\_Ln\_Disp
HEAP(Txt_Add@ + TXTBOT) \triangleq \text{Last}\Txt\_Ln\_Disp

and
Txt\_Ln\_Disp is TXT\_LN\_DISP means
\[7 \leq \Txt\_Ln\_Disp < \text{HEAP}(\text{Txt\_Add}@ + \text{HPSIZ})\]
HEAP(Txt_Add@ + TXTNLIN) \triangleq Txt\_Total\_Ln

and
Txt\_Total\_Ln is TXT\_TOTAL\_LN means
\[
\Txt\_Total\_Ln \triangleq \text{Total number of text lines in the source text.}
\]

(1.5) Text Line

Txt\_Ln is TXT\_BLOCK . TXT\_LN means
a text line and \text{HEAP}(\text{Txt\_Add}@ + \Txt\_Ln\_Disp) is word prior to the start of figure 2-7.

and

TXT\_LN . TXTLNUM \triangleq \text{HEAP}(\text{Txt\_Add}@ + \Txt\_Ln\_Disp + 1)
TXT\_LN . TXTFLK \triangleq \text{HEAP}(\text{Txt\_Add}@ + \Txt\_Ln\_Disp + 2)
TXT\_LN . TXTBLK \triangleq \text{HEAP}(\text{Txt\_Add}@ + \Txt\_Ln\_Disp + 3)
TXT\_LN . TXTTOK \triangleq \text{HEAP}(\text{Txt\_Add}@ + \Txt\_Ln\_Disp + 4)
TXT\_LN . TXTNST \triangleq \text{HEAP}(\text{Txt\_Add}@ + \Txt\_Ln\_Disp + 5)
TXT\_LN . TXTSTAT \triangleq \text{HEAP}(\text{Txt\_Add}@ + \Txt\_Ln\_Disp + 6)
TXT_LN + TXTLEN \leq HEAP(Txt_Add@ + Txt_Ln disap + 7)
TXT_LN + TXTLIN \leq HEAP(Txt_Add@ +Txt_Ln disap + 8)

and

HEAP(Txt_Add@ + Txt_Ln disap + TXTNUM) \leq Txt_Ln Num

and

Txt_Ln Num is TXT_LN NUM means

1 \leq Txt_Ln Num \leq HEAP(Txt_Add@ + TXTBOT + TXTNUM)

HEAP(Txt_Add@ + Txt_Ln disap + TXTFLK) \leq Next Txt_Ln disap

HEAP(Txt_Add@ + Txt_Ln disap + TXTBLK) \leq Last Txt_Ln disap

HEAP(Txt_Add@ + Txt_Ln disap + TXTTOK) \leq i

and

i \in \{0, 1\}

i \neq 0 means line not scanned

i \neq 1 means line scanned

HEAP(Txt_Add@ + Txt_Ln disap + TXTNST) not used by Scanner module

HEAP(Txt_Add@ + Txt_Ln disap + TXTSTAT) not used by Scanner module

HEAP(Txt_Add@ +Txt_Ln disap + TXTLEN) \leq Total_Char

and

Total_Char is TOTAL_CHAR means

64 \geq Total_Char = number of characters in source line

and

CHAR means

CHAR \in \{1, ..., 255\}

and

\{1, ..., 255\} corresponds to the machine Character Translation Table

3.3.2 Output Assertions
(2.1) ERROR

I is ERROR means

I ∈ \{0, 1, 555, 900\}.

I ≠ 0 means no errors

I ≠ 1 means syntax errors in some Text_Ln's

I ≠ 555 means insufficient Allocated_Space for Tokens
    of source text, incomplete scan

I ≠ 900 means user interrupt source text, incomplete scan

(2.2) Token Logical Address

I is TOK_LOG_ADD

Tok_Log_Add is TOK_LOG_ADD means

Tok_Log_Add is a HEAP • LOG_ADD

and

HEAP(Proc_Tab_Add + PRCTOK) ≠ Tok_Log_Add

and

HEAP(Tok_Log_Add) ≠ Tok_Add@

and

Tok_Add@ is a TOK_ADD@

(2.3) Token Absolute Address

Tok_Add@ is TOK_ADD@ means

HEAP(HPSIZ) < Tok_Add@ < n

and

∃ a token block as per figure 2-7 starting at
    HEAP(Tok_Add@ + 1)

(2.4) Token Block

Tok_Block is HEAP • TOK_BLOCK means

Tok_Block is a Linked_List as shown in figures 2-7
    and 2-8
and
TOK_BLOCK \cdot HPTAG \triangleq \text{HEAP}(\text{Tok_Add@ + 1})
TOK_BLOCK \cdot HPSPAC \triangleq \text{HEAP}(\text{Tok_Add@ + 2})
TOK_BLOCK \cdot HPSIZ \triangleq \text{HEAP}(\text{Tok_Add@ + 3})
TOK_BLOCK \cdot HPLAD \triangleq \text{HEAP}(\text{Tok_Add@ + 4})
TOK_BLOCK \cdot TOKTOP \triangleq \text{HEAP}(\text{Tok_Add@ + 5})
TOK_BLOCK \cdot TOKBOT \triangleq \text{HEAP}(\text{Tok_Add@ + 6})
TOK_BLOCK \cdot TOKNLIN \triangleq \text{HEAP}(\text{Tok_Add@ + 7})

and
\text{HEAP}(\text{Tok_Add@ + HPSPAC}) \triangleq \text{Allocated\_Space}
\text{HEAP}(\text{Tok_Add@ + HPSIZ}) \triangleq \text{Space\_Used}
\text{HEAP}(\text{Tok_Add@ + HPLAD}) \triangleq \text{Tok\_Log\_Add}
\text{HEAP}(\text{Tok_Add@ + TOKTOP}) \triangleq \text{1st\_Tok\_Ln\_Disp}

and
Tok\_Ln\_Disp \text{ is \texttt{TOK\_LN\_DISP means}}
7 \leq Tok\_Ln\_Disp \leq \text{HEAP}(\text{Tok_Add@ + HPSIZ})
\text{HEAP}(\text{Tok_Add@ + TOKBOT}) \triangleq \text{HEAP}(\text{Tok_Add@ + HPSIZ}) + 1
\text{HEAP}(\text{Tok_Add@ + TOKNLIN}) \triangleq \text{Tok\_Total\_Ln}

and
Tok\_Total\_Ln \text{ is \texttt{TOK\_TOTAL\_LN means}}
Tok\_Total\_Ln \triangleq \text{total number of token lines}

(2.5) Token Line

Tok\_Ln \text{ is \texttt{TOK\_LN means}}
\exists \text{ a token line and } \text{HEAP}(\text{Tok\_Add@ + Tok\_Ln\_Disp})
is \text{ word prior to the start of a token line as per figure 2-8.}
and

\[ \text{TOK}_\text{LN} \cdot \text{TOKLNUM} \triangleq \text{HEAP}(\text{Tok}_\text{Add}@ + \text{Tok}_\text{Ln}_\text{Disp} + 1) \]

\[ \text{TOK}_\text{LN} \cdot \text{TOKFLK} \triangleq \text{HEAP}(\text{Tok}_\text{Add}@ + \text{Tok}_\text{Ln}_\text{Disp} + 2) \]

\[ \text{TOK}_\text{LN} \cdot \text{TOKNUM} \triangleq \text{HEAP}(\text{Tok}_\text{Add}@ + \text{Tok}_\text{Ln}_\text{Disp} + 3) \]

and

\[ \text{HEAP}(\text{Tok}_\text{Add}@ + \text{Tok}_\text{Ln}_\text{Disp} + \text{TOKLNUM}) \triangleq \text{Tok}_\text{Ln}_\text{Num} \]

and

\[ \text{Tok}_\text{Ln}_\text{Num} \text{ is } \text{TOK}_\text{LN}_\text{NUM} \text{ means } \]

\[ 1 \leq \text{Tok}_\text{Ln}_\text{Num} \leq \text{HEAP}(\text{Txt}_\text{Add}@ + \text{TXTBOT} + \text{TXTLNUM}) \]

\[ \text{HEAP}(\text{Tok}_\text{Add}@ + \text{Tok}_\text{Ln}_\text{Disp} + \text{TOKFLK}) \triangleq \text{Next } \text{Tok}_\text{Ln}_\text{Disp} \]

\[ \text{HEAP}(\text{Tok}_\text{Add}@ + \text{Tok}_\text{Ln}_\text{Disp} + \text{TOKNUM}) \triangleq \text{Total } \text{Tokens} \]

and

\[ \text{Total } \text{Tokens is } \text{TOTAL } \text{TOKENS} \text{ means } \]

\[ \text{Total } \text{Tokens} \triangleq 0 \text{ or number of tokens triples in the token line as per figure 2-9.} \]

\[ \text{Total } \text{Tokens} \triangleq 0 \text{ means } \]

Syntax error in \text{Txt}_\text{Ln} and no tokens were generated.

and

\[ \text{Token} \triangleq \begin{bmatrix} \text{Class} \\ \text{Index or Value} \\ \text{Num}_\text{First}_\text{Char} \end{bmatrix} \]

3.3.3 Table, Counter, Flag Initialization and Variable Definition

The designated tables, counters, flags and variables
shown as being initialized in the Scanner module are common
to all submodules of the Scanner. Definition of state changes
are also given where appreciable.

(3.1) Table Initialization

(3.1.1) CTAB is CHARACTER TABLE means

CTAB(i, j) array

i is \( \in \{1, \ldots, 255\}\)

j is \( \in \{1, 2, 3\}\)

and

CTAB(i, j) is initialized as per figure 2-9

(3.1.2) KW TAB is KEYWORD TABLE means

KW TAB(i, j) array

i is \( \in \{1, \ldots, 10\}\)

j is \( \in \{1, \ldots, 7\}\)

and

KW TAB(i, j) is initialized as per figure 2-10

(3.1.3) CL TAB is COMMAND LANGUAGE TABLE means

CL TAB(i, j) array

i is \( \in \{1, \ldots, 10\}\)

j is \( \in \{1, \ldots, 7\}\)

and

CL TAB(i, j) is initialized as per figure 2-11

(3.1.4) LINST is LINE STACK means

LINST(i) array

i is \( \in \{1, \ldots, 64\}\)

and

LINST(i) is initialized to zero
state change

For each Txt_Ln processed that is to be scanned.
LINST(i) \& CHAR

i is \in \{1, \ldots, (Txt_Ln(TXTLLN))^2\}

(3.1.5) SYM is SYMBOL means

SYM(i) array

i is \in \{1, 2\}

and

SYM(i) is initialized to zero

state change

Array SYM is used to pack up to the maximum of
eight characters of each formed symbol that
requires additional processing to produce token
data. Classes of symbols requiring the use of
array SYM are:

identifiers
keywords
command language words
numbers

(3.1.6) SYMST is SYMBOL STACK means

SYMST(i, j) array

i is \in \{1, \ldots, 15\}

j is \in \{1, 2, 3\}

and

SYMST(i, j) is initialized to zero

state change

For all symbols formed in a Txt_Ln being scanned.

SYMST(i) \& 1 thru SYMCT
and

SYMCT is SYMBOL COUNT means
SYMCT \triangleq \text{total number of symbols formed}

and

SYMST(j) \triangleq \text{the Token Data for the symbol}

(3.1.7) ERRST is ERROR STACK means

ERRST(i,j) array

\begin{align*}
    i & \in \{1, \ldots, 64\} \\
    j & \in \{1, 2, 3\}
\end{align*}

and

ERRST(i,j) is initialized to zero

state change

Txt_Ln is scanned and an error was found

ERRST(i) \triangleq 1 thru ERRCT

and

ERRCT is ERROR_COUNT means

ERRCT \triangleq \text{total Scanned Txt_Ln with errors}

\begin{align*}
    \begin{bmatrix}
        \text{Txt_Ln_Num} \\
        \text{ERRST(j) \neq \text{ERCODE}} \\
        \text{Num_First_Char}
    \end{bmatrix}
\end{align*}

(3.2) Counter Initialization

(3.2.1) SYMCT is SYMBOL_COUNT means for each Txt_Ln

\begin{align*}
    0 \leq \text{SYMCT} \leq 15
\end{align*}

and

for each Symbol formed SYMCT is incremented by 1

(3.2.2) ERRCT is ERROR_COUNT means for each Txt_Block

\begin{align*}
    0 \leq \text{ERRCT} \leq \text{Total_Txt_Ln}
\end{align*}

and
for each Txt_Ln with syntax error,
ERRCT is incremented by one.

(3.2.3) SYML is SYMBOL_LENGTH means for each Symbol
        \[ 1 \leq SYML \leq 8 \]

(3.2.4) CHARNU is CHARACTER_NUMBER means for each CHAR in
        a Symbol
        CHARNU is incremented by one.

(3.3) Flag Initialization

(3.3.1) RESCAN = TRUE means
        Tok_Block exists prior to call of Scanner

        RESCAN = FALSE means
        Tok_Block does not exist prior to call of Scanner

(3.3.2) CLFLAG is Command Language Flag initialized FALSE

        CLFLAG = TRUE means
        first CHAR of Txt_Ln is "")"

(3.3.3) ERRFLG is ERROR_FLAG initialized FALSE

        ERRFLG = TRUE if ERRCT = 1

(3.3.4) TYPE_TOK is TYPE_TOKEN initialized "ITOK" means Token
        tag code for HEAP area.

(3.4) Variable Definitions

(3.4.1) ERCODE is ERROR_CODE means Syntax_Error,
        ERCODE \in \{1, 2, 3\}

        and
        ERCODE = 1 means Syntax_Error,
        SYML > 8 for any symbol except a string

        and
        ERCODE = 2 means Syntax_Error,
        no closing quote mark on string

        and
ERCODE \neq 3 \text{ means } \text{Syntax_Error},
\quad \text{CHAR} \notin \text{in machine character translation table or}
\quad \text{CHAR is illegal in a specific class of symbol}

(3.4.2) \text{SIZE is Requested Allocated Space for Tok_Block means}
\quad \text{SIZE \neq \text{Txt_Total_Mn} \times 15 \times 3}
\quad \text{and}
\quad 15 \text{ is maximum number of symbols per Txt_Ln}
\quad \text{and}
\quad 3 \text{ is number of storage words per token}

(3.4.3) \text{SYM_CODE is SYMBOL CODE means}
\quad \text{SYM_CODE} \in \{1, 2, 3\}
\quad \text{and}
\quad \text{SYM_CODE \neq 1 \text{ means}}
\quad \quad \text{Symbol is}\begin{cases} \text{identifier or} \\
\text{keyword or} \\
\text{command language word}
\end{cases}
\text{and}
\quad \text{SYM_CODE \neq 2 \text{ means}}
\quad \quad \text{Symbol is}\begin{cases} \text{identifier or} \\
\text{undefined operator}
\end{cases}
\text{and}
\quad \text{SYM_CODE \neq 3 \text{ means}}
\quad \quad \text{Symbol is}\begin{cases} \text{keyword or} \\
\text{command language word or} \\
\text{number or} \\
\text{string}
\end{cases}

(3.4.4) \text{CLASS is SYMBOL_CLASS means}
\quad \text{CLASS} \in \{1, 2, 3, 4, 5, 6, 7, 8\}
\quad \text{and}
CLASS \( \# 1 \) means Symbol is keyword or command language word
CLASS \( \# 2 \) means Symbol is identifier
CLASS \( \# 3 \) means Symbol is integer number
CLASS \( \# 4 \) means Symbol is real number
CLASS \( \# 5 \) means Symbol is string
CLASS \( \# 6 \) means Symbol is operator
CLASS \( \# 7 \) means Symbol is separator
CLASS \( \# 8 \) means Symbol is undefined operator

(3.4.5) INDEX is SYMBOL_INDEX means
INDEX \( \geq 1 \) or \( \# \) VALUE

and
INDEX \( \# \geq 1 \) means
\[
\begin{align*}
\text{Symbol} & \in \{ \text{keyword} \} \\
& \cup \{ \text{identifier} \} \\
& \cup \{ \text{operator} \} \\
& \cup \{ \text{separator} \} \\
& \cup \{ \text{undefined operator} \}
\end{align*}
\]
INDEX \( \# \) VALUE means
\[
\begin{align*}
\text{Symbol} & \in \{ \text{integer number} \} \\
& \cup \{ \text{real number} \} \\
& \cup \{ \text{string} \}
\end{align*}
\]

and
VALUE \( \# \) numeric value or character length of string

(3.4.6) Num_First_Char is NUM_FIRST_CHAR means

Num_First_Char \( \in \{ 1, \ldots, \text{Total_Char} \} \)

and

Num_First_Char \( \# \) number of the first character of a symbol in aTxt_Ln

3.4 Global Procedure Assertion Refinement
3.4.1 Subroutine GET(SIZE, TYPE, LADDR, ERROR)

(1.1) Input Assertions

Global Assertions are valid

I is SIZE means I > 0

I is TYPE means I is integer type code for

Proc_Tab_Block
Txt_Block
Tok_Block
Code_Block (Undefined for Scanner Proof)
Temp_Block (Undefined for Scanner Proof)

(1.2) Output Assertions

I is ERROR means

I ∈ \{0, 555\}

and

ERROR ≠ 0 means

LADDR ≠ HEAP · Log_Add

and

HEAP((HEAP(Log_Add)) + HPTAG) ≠ TYPE
HEAP((HEAP(Log_Add)) + HPSPAC) ≠ new_SIZE SIZE
HEAP((HEAP(Log_Add)) + HPLAD) ≠ LADDR

or

ERROR ≥ 555

and

LADDR ≠ Null

3.4.2 Subroutine EXPAND(SIZE, LADDR, ERROR)

(2.1) Input Assertions
Global assertions are valid

I is SIZE means I > 0

LADDR ≠ HEAP . Log_Add

(2.2) Output Assertions

I is ERROR means

I ∈ \{0, 555\}

and

ERROR = 0

means

HEAP((HEAP(Log_Add)) + HPSPAC) ≠

old_HEAP((HEAP(Log_Add)) + HPSPAC) + SIZE

or

ERROR ≠ 555

and

HEAP((HEAP(Log_Add)) + HPSPAC) ≠

old_HEAP((HEAP(Log_Add)) + HPSPAC)

3.4.3 Subroutine SYMTAB( PINDEX, SYML, SYM, INDEX, ERROR)

(3.1) Input Assertions

Global assertions are valid

I is SYML means SYML as defined in para. 3.3.3.2.3

SYM ≠ two words

and

SYML ≠ CHAR in SYM

(3.2) Output Assertions

ERROR means

ERROR ∈ \{0, 1\}

and

ERROR = 0
I is INDEX means I \geq 0

and

HEAP . SYM_TAB (INDEX) \triangleq Symbol @

*(HEAP . SYM_TAB is undefined in this report, but takes the form as a Block area).

or

ERROR \triangleq 0

I is INDEX means I \neq Null

3.4.4 Subroutine ERRPT (PINDEX, LINE, CHAR, N, STRING, RETCOD)

(4.1) Input Assertions

Global assertions and Scanner input assertions are valid

I is LINE means I \triangleq Txt_In_Num

I is CHAR means I \triangleq Num_First_Char of Syntax_Error

I is N means I \triangleq n length of STRING

STRING \triangleq Syntax_Error_Message

(4.2) Output Assertions

I is RETCOD means

I \in \{0,1\}

and

RETCOD \neq 0, message was printed

or

RETCOD \neq 1, message was not printed

3.4.5 Subroutine STAX (SWITCH, DELAY)

(5.1) Input Assertions

I is SWITCH means I \neq 0

I is DELAY means I \neq 150

(5.2) Output Assertions

I is SWITCH means
\[ I \in \{0, 1\} \]

and

\[ \text{SWITCH} = 0 \text{ means no interrupt} \]

or

\[ \text{SWITCH} = 1 \text{ means user interrupt} \]

3.4.6 Subroutine \text{GETCHR}(\text{STRG}, J, \text{ARG3})

(6.1) Input Assertions

Global assertions and Scanner input assertions are valid

\[ I \text{ is WORD means } \]

\[ I \in \text{Txt_Add} + \text{Txt_Ln Dis} \text{p} + \text{TXTLIN} \]

\[ I \text{ is J means } \]

\[ I \in \{1, \ldots, \text{Total_Char}\} \]

(6.2) Output Assertions

\[ I \text{ is ARG3 means } I \notin \text{CHAR} \]

and

\[ \text{CHAR is } (J) \]

3.4.7 Subroutine \text{PUTCHR}(\text{SYM}, K, \text{CHAR})

(7.1) Input Assertions

\[ \text{SYM} \notin \text{two words} \]

\[ I \text{ is K means } \]

\[ I \in \{1, \ldots, 8\} \]

\[ I \text{ is CHAR means } \]

\[ I \in \{1, \ldots, 255\} \]

(7.2) Output Assertions

\[ \text{SYM(2)} \notin \text{contains K number of CHAR} \]

3.5 High Level Design Language with Assertions

The High Level Design Language representation of the Scanner module is developed from the Scanner specifications
and refined assertions. Assertions are referenced by paragraph number and are enclosed in brackets at the point in the program where they are valid. Assertions are numbered sequentially in each routine.

3.5.1 SUBROUTINE SCANNER (PINDEX,ERROR)

\{1. Input Assertions (see para. 3.2 and 3.5.1)\}

**INITIALIZE** \{2. see para. 3.3.3\}

RESCELL \rightarrow FALSE
ERRFLG \rightarrow FALSE
CLFLG \rightarrow FALSE
TYPE_TOK \rightarrow "ITOK"
SYMCT \rightarrow 0
ERRCT \rightarrow 0
CHARNU \rightarrow 0

CTAB \rightarrow \text{to figure 2-9}
KW_TAB \rightarrow \text{to figure 2-10}
CL_TAB \rightarrow \text{to figure 2-11}

LINST \rightarrow 0
SYM \rightarrow 0
SYMST \rightarrow 0
ERRST \rightarrow 0

**Proc Tab Block** \rightarrow \text{HEAP(5)} \{3. see para. 3.2.3\}

\text{Txt Log Add} \rightarrow \text{HEAP(Proc Tab Block} \oplus \text{ PINDEX \oplus PRCTXT}

\{4. see para. 3.3.1.2\}

**Tok Log Add** \rightarrow \text{HEAP(Proc Tab Block} \oplus \text{ PINDEX \oplus PRCTOK)}

**Txt ADD** \rightarrow \text{HEAP(Txt Log Add)} \{5. see para. 3.2.6\}

**Txt Total Ins** \rightarrow \text{HEAP(Txt ADD} \oplus \text{ TXTNLIN} \{6. see para. 8.3.1.3\ and para. 3.3.1.4\}

\boxed{\text{CASE 1 IF Tok Log Add = 0}}
THEN \{7. see para. 3.2.6 no Token data block exists\}
SIZE ← Txt_Total_Lns * 15 * 3
CALL GET (SIZE, TYPE_TOK, Tok_Log_Add, ERROR)
\{8. see para. 3.4.1\}

CASE 2 IF ERROR > 0
THEN \{9. Token data block not available\}
RETURN

END CASE 2

HEAP(Proc_Tok_Block + PINDEX + PRCTOK) ← Tok_Log_Add
\{10. see para. 3.2.6\}
ELSE \{11. see para. 3.2.6 Token data block does exist\} and some lines of source text have been
Scanned, see para. 3.3.3
RESCAN ← TRUE

END CASE 1 \{12. A Token data block exists and header conforms to figure 2-7, see para. 3.3.2.4\}

Tok_Add@ ← HEAP(Tok_Log_Add)
Txt_Add@ ← HEAP(Txt_Log_Add)
\{13. Absolute addresses of block areas may have changed due to reorganization of HEAP and Add@ were reassigned.\}
Txt_Ln_Disp ← HEAP(Txt_Add@ + TXTTOP)

Next  Txt_Ln_Disp ← HEAP(Txt_Add@ + Txt_Ln_Disp + TXTFLX)
\{14. see para. 3.3.1.4\}
Tok_Ln_Disp ← HEAP(Tok_Add@ + TOKTOP)
\{15. see para. 3.3.2.4\}

CASE 3 IF Tok_Ln_Disp < 7
THEN \{16. Token data block contains no Token Lines\}
   Tok_Ln_Disp ← 7
   HEAP(Tok_Add@ + TOKTOP) ← Tok_Ln_Disp
HEAP(Tok_Add@ + HPSIZ) ← Tok_Ln_Dis

END CASE 3

{17. All Add@, First_txt_Ln_Dis, Next_txt_Ln_Dis, and First_tok_Ln_Dis are correct}

LOOP 1 iteration 1 thru TXT_Total_Ln

{18. Each LAST_txt_Ln, para. 3.3.1.5, has been SCANNED.}

   Tok_Ln, para. 3.3.2.5, for each TXT_Ln, para. 3.3.1.5, that has been SCANNED

   For each error-free TXT_Ln, the corresponding Tok_Ln contains tokens for each text symbol as formatted in figure 2-8, see para. 3.3.2.5.

   For each TXT_Ln that contains Syntax error, the Tok_Ln . TOKNUM equals zero, see para. 3.3.2.5, and ERRST contains error data, see para. 3.3.3.1.7.

CASE 4 IF Tok_Ln_Num < TXT_Ln_Num AND RESCAN = TRUE

   THEN {19. A line of text has been deleted.}

   DELETE Tok_Ln UNTIL Tok_Ln_Num ≥ TXT_Ln_Num

CASE 5 IF TXT_Ln_Num > LAST Tok_Ln_Num

   THEN {20. TXT_Ln is addition to end of source text that has been scanned.}

   RESCAN ← FALSE

   {21. see para. 3.3.3.3.1}

END CASE 5

END CASE 4

{22. Tok_Ln_Num ≥ TXT_Ln_Num or RESCAN = FALSE}

CASE 6 IF HEAP(Txt_Add@ + TXT_Ln_Dis + TXTTOK) = 0

   THEN {23. Text line has not been SCANNED, see}

   CALL LNSCAN (ERRFLG, Ln_Space_Used, TXT_Ln_Dis, SYMCT)
ELSE GOTO End Loop 1

END CASE 6

24. Txt_Ln has been SCANNED and state changes for SYMST, see para. 3.3.3.1.6; SYMCT as para. 3.3.3.2.1; and
   Ln_Space_Used is storage words required for Tok_Ln; or
   Syntax_Error in Txt_Ln and state change for ERRST, see para. 3.3.3.1.7; ERRCT as per para. 3.3.3.2.2; and
   ERRFLAG = TRUE.

   Tok_Space_Used ← HEAP(Tok_Add@ + HPSIZ)

   25. see para. 3.3.2.4

CASE 7  IF Tok_Space_Used + Ln_Space_Used > HEAP(Tok_Add@ + HPSPAC)

   THEN 26. Token data block is not large enough to hold all token lines.

   SIZE ← (Txt_Total_Ln - Loop 1 iteration)x15x3

   CALL EXPAND (SIZE, Tok_Log_Add, ERROR)

   27. see para. 3.4.2

CASE 8  IF ERROR > 0

   THEN 28. Additional space not available for Token data block.

   RETURN

   ELSE 29. Additional space is available.

   Tok_Add@ ← HEAP(Tok_Log_Add)

   Txt_Add@ ← HEAP(Txt_Log_Add)

   30. Absolute address may have changed

END CASE 7

END CASE 7

HEAP(Tok_Add@ + Tok_Ln Disp + TOKLNUM) ←
   HEAP(Txt_Add@ + Txt_Ln Disp + TXTLNUM)

31. see para. 3.3.2.5
CASE 9  IF RESCAN = TRUE

THEN  CALL LINFIN (Txt_LnDisp, RESCAN, Tok_LnDisp)

\{32. Tok_Ln header has been assigned following the\}
\{Last Tok_Ln. Tok_Ln in prior Tok_Ln,\}
\{and this Tok_Ln points to the correct lines,\}
\{OR RESCAN = FALSE, see para. 3.3.3.3.1.\}

ELSE  HEAP(Tok_Add@ + Tok_LnDisp + TOKFLK) \leftarrow
  HEAP(Tok_Add@ + HPSIZ) + Ln_Space_Used

\{33. Tok_Ln header as per para. 3.3.2.5 has been\}
\{assigned following the Last Tok_Ln.\}

END CASE 9

CASE 10 IF ERRFLG = TRUE

THEN \{34. An error was found in the line of text,\}
\{see para. 3.3.2.5.\}

  HEAP(Tok_Add@ + Tok_LnDisp + TOKNUM) \leftarrow 0

ELSE \{35. Text line was error free.\}

  HEAP(Tok_Add@ + Tok_LnDisp + TOKNUM) \leftarrow SYMCT

  LOOP_2  iteration 1 thru SYMCT

  Token \leftarrow SYMST

END LOOP_2

\{36. Symbol token data has been stored as per\}
\{figure 2-8, see para. 3.3.2.5.\}

END CASE 10

\{37. Tok_Ln corresponding to This Txt_Ln has been stored\}
\{in Tok_Block.\}

  HEAP(Tok_Add@ + HPSIZ) \leftarrow Tok_Space_Used + Ln_Space_Used

  HEAP(Tok_Add@ + TOKBOT) \leftarrow Tok_Space_Used + Ln_Space_Used

  HEAP(Tok_Add@ + TOKNLIN) \leftarrow HEAP(Tok_Add@ + TOKNLIN) + 1

\{38. Tok_Block header has been updated, see para. 3.3.2.4\}

SWITCH \leftarrow 0
DELAY ← 150

CALL STAX (SWITCH, DELAY) \{39. see para. 3.4.5\}

CASE 11 IF SWITCH > 0

THEN \{40. User has caused an interrupt.\}

ERROR ← 900 \{41. see para. 3.3.2.1\}

RETURN

END CASE 11

\{42. No user interrupt has occurred.\}

CASE 12 IF HEAP(Txt_Add + Txt_Ln_Dis + TXTLNUN) = 

HEAP(Tok_Add + Tok_Ln_Dis + TOLKLNUN)

THEN Txt_Ln_Dis ← HEAP(Txt_Add + Txt_Ln_Dis + 

TXTFLK)

Tok_Ln_Dis ← HEAP(Tok_Add + Tok_Ln_Dis + 

TOKFLK)

\{43. Text and Token displacement indexes now reflect 

the next lines to be processed.\}

ELSE ERROR ← 1

\{44. Error has occurred in storing token data.\}

RETURN

END CASE 12

END LOOP 1

\{45. All lines of source text have been SCANNED, and token

data stored in the Token data block as per figures 2-7

and 2-8, see para. 3.3.2.4 and 3.3.2.5.\}

CASE 13 IF ERRFLG = TRUE

THEN \{46. see para. 3.3.3.3\}

CALL SERROR

\{47. Error messages were printed for each line that

contained an error.\}

ERROR ← 1 \{48. see para. 3.3.2.1\}
RETURN
ELSE {49. All lines of source text were error free and}
{ERROR = 0, see para. 3.3.2.1}
RETURN
END CASE 13

{50. Output assertions, see para. 3.2.2}

END SUBROUTINE SCANNER

3.5.2 SUBROUTINE LNSCAN (ERRFLG, Ln_Space_Used, Txt_Ln_Disp, SYMCT)

{1. Input Assertions - Scanner Input Assertions + Scanner Tables + Scanner Flags + Scanner Counters + Scanner Variables and Txt_Ln_Disp = First unscanned Txt_Ln}

INITIALIZE

SYMCT ← 0
ERCODE ← 0
CLFLAG ← FALSE

{2. Counters and Flags reset for this line.}
Txt_Add@ ← HEAP(Txtlogan_Add)
Txt_Ln_Total_Disp ← Txt_Add@ + Txt_Ln_Disp
Total_Char ← HEAP(Txt_Ln_Total_Disp + TXTLEN)

{3. see para. 3.3.1.3 and 3.3.1.5}

LOOP 1 iteration 1 thru Total_Char

{4. After each iteration a character has been taken from a packed word in the text line and placed in a single location corresponding to the iteration number in array LINST.}

CALL GETCHR (HEAP(Txt_Ln_Total_Disp + TXTLIN), Loop 1 iteration, ARG3)

{5. see para. 3.4.6}
LINST(iteration) ← ARG3
6. see state change para. 3.3.3.1.4.

END LOOP 1

7. Character representations stored in array LINST are the numeric machine translations for the characters corresponding to table CTAB; see para. 3.3.3.1.1 and 3.3.3.1.4.

CASE 1 IF LINST(1) = ""

THEN 8. Txt_Ln is a command language line; see para.

CLFLAG <- TRUE

END CASE 1

9. Distinction has been made if Txt_Ln is or is not command language line.

LOOP 2 iteration 1 thru Total_Char

10. After each iteration a symbol has been recognized and token data placed in array SYMST, or an error has been detected in the line of text and control RETURNED to Subroutine SCAN.

CASE 2 IF CHARNU iteration

THEN 11. This character is part of a recognized symbol.

GOTO End Loop 2

END CASE 2

12. Character represented by LINST(iteration) is a blank, a symbol, or the first character of a symbol.

CHAR <- LINST(iteration)

Pointer <- CTAB(CHAR, 1)

13. Pointer identifies the character as per column 1, figure 2-10.

CONDITION 1 IF Pointer = 1

THEN 14. Character is a blank.

GOTO End Loop 2.
END CONDITION 1

CONDITION 2 IF Pointer = 2

THEN {15. Character is an operator or separator.}

SYMCT ← SYMCT + 1

SYMST(SYMCT,1) ← CTAB(CHAR,2)

SYMST(SYMCT,2) ← CTAB(CHAR,3)

SYMST(SYMCT,3) ← iteration

{16. Token data stored in array SYMST, figure 2-10, state change para. 3.3.3.1.6 and para. 3.3.3.2.1}

GOTO End Loop 2

END CONDITION 2

CONDITION 3 IF Pointer = 3

THEN {17. Character is an identifier, keyword, command language word, string, number, or undefined operator.

Num_First_Char ← iteration {18. see para. 3.3.2.5

CHARNU ← iteration {19. see para. 3.3.3.2.4

CALL FORM (CHARNU,SYML,Total_Char,ERCODE,INDEX,CLASS,SYMCODE,Num_First_Char)

{20. Symbol has been formed and is stored in array SYM or error was found in line; see para. 3.3.3.1.5

CASE 3 IF ERCODE > 0

THEN {21. Text line contained a syntax error at this symbol.

ERRCT ← ERRCT + 1

ERRST(ERRCT,1) ← HEAP(Txt_Ln_Total_Disp + TXTLNUM)

ERRST(ERRCT,2) ← Num_First_Char

ERRST(ERRCT,3) ← ERCODE
ERRFLG ← TRUE
SYMCT ← 0
RETURN

{22. If an error was in the line, error data has been stored in array ERRST; see state change para. 3.3.3.1.7}

END CASE 3

{23. No error was found in the line.}

CASE 4 IF SYMCODE = 1

THEN {24. Symbol is an identifier, keyword, or command language word; see para. 3.3.3.4.3}

CALL TABLE (SYML, CLASS, INDEX)

END CASE 4

{25. If symbol was a keyword or command language word, SYMCODE = 3; ELSE symbol is an identifier and CLASS = 2.}

CASE 5 IF SYMCODE = 2 OR CLASS = 2

THEN {26. Symbol is an identifier or undefined operator; see para. 3.3.3.4.3}

CALL SYMTAB (PINDEX, SYML, SYM, INDEX, ERROR)

CASE 6 IF ERROR > 0

THEN {27. INDEX could not be assigned.}

RETURN

END CASE 6

{28. INDEX has been assigned.}

SYMCT ← SYMCT + 1

SYMST (SYMCT, 1) ← CLASS
SYMST(SYMCT,2) ← INDEX
SYMST(SYMCT,3) ← Num_First_Char

{29. Token data stored in array SYMST; see
    state change, para. 3.3.3.1.6 and
    para. 3.3.3.2.1
}
END CASE 5

CASE 7 IF SYMCODE = 3

THEN {30. Symbol is a keyword, command
            language word, number, or
            string.
            SYMCT ← SYMCT + 1
            SYMST(SYMCT,1) ← CLASS
            SYMST(SYMCT,2) ← INDEX
            SYMST(SYMCT,3) ← Num_First_Char

            {31. Token data stored in array SYMST;
                see state change, para. 3.3.3.1.6
                and para. 3.3.3.2.1
            }
}
END CASE 7

END LOOP 2

In_Space_Used ← (SYMCT + 1) × 3

RETURN

{32. Output Assertions
    That all symbols in the Txt_Ln have been formed, Token
    data for each symbol has been placed in array SYMST,
    SYMCT equals the number of symbols in the Txt_Ln, and
    In_Space_Used equals the number of storage words required
    to store the Tok_Ln, OR an error was found in the Txt_Ln,
    error data has been stored in the array ERRST, ERRCT
    equals the total number of lines that errors have been
    found, SYMCT equals zero, and In_Space_Used equals three.
}
3.5.3 SUBROUTINE FORM (CHARNU, SYML, Total_Char, ECODE, INDEX, CLASS, SYMCODE, Num_First_Char)

1. Input Assertions - Scanner Input Assertions + Scanner Tables + Scanner Flags + Scanner Counters + Scanner Variables

   Total_Char = HEAP(Txt_Add@ + Txt_Ln_Disp + TXTLEN)
   CHARNU = Num_First_Char
   LINST = state change; see para. 3.3.3.1.4
   Num_First_Char = Number of the first CHAR of the Symbol in the Txt_Ln

INITIALIZE

   SYMCODE ← 0
   CLASS ← 0
   SYML ← 1
   INDEX ← 0
   VALUE ← 0
   K ← 0
   CHAR ← LINST(CHARNU)
   I ← Num_First_Char + 1

2. All internal variables initialized + Scanner variable reset

   Pointer ← CTAB(CHAR, 2)

3. Pointer identifies the character as a letter, number, double quote mark, or a dollar sign.

CONDITION 1  Pointer = 1

   THEN 4. Symbol is an identifier, keyword, or command language word.

      SYMCODE ← 1 5. see para. 3.3.3.4.3

   LOOP 1 iteration I thru Total_Char
{6. Last character was part of symbol.}

CHAR ← LINST(iteration)

CASE 1 IF CTAB(CHAR,2) = 1

THEN {7. Symbol is formed, and SYML = }

{ Length of Symbol. }

CASE 2 IF SYML > 8

THEN {8. Syntax error symbol is }

{ too long. }

ERCODE ← 1 {9. see para. }

{ 3.3.3.4.1 }

RETURN

ELSE {10. Symbol is error free.}

LOOP 2 iteration

Num_First_Char thru CHARNU

{11. Prior character was }

{ packed in SYM. }

CHAR ← LINST(iteration)

K ← K + 1

CALL FUTCHE(SYM,K,CHAR)

{12. see para. 3.4.7.2}

END LOOP 2

{13. Symbol is packed four }

{ characters per word }

{ into array SYM; see }

{ state change, para. }

{ 3.3.3.1.5 }

RETURN

END CASE 2

ELSE {14. Symbol may contain more characters.}

SYML ← SYML + 1
CHARNU ← CHARNU + 1

END CASE 1

END LOOP 1

END CONDITION 1

CONDITION 2 Pointer = 2

THEN {15. Symbol is a number; see para. 3.3.3.4.4 and } para. 3.3.3.4.3

CLASS ← 3

SYM CODE ← 3

LOOP 3 iteration I thru Total_Char

{16. Last CHAR was part of symbol. }

CHAR ← LINST(iteration)

CASE 3 IF CTAB(CHAR, 2) = 2 OR Period

THEN {17. Symbol is formed. }

CALL NUMFAC(Num_First_Char, SYML, VALUE, ERCODE)

INDEX ← VALUE

RETURN {18. Token data complete for number. }

ELSE {19. Symbol may contain more characters.

CASE 4 IF CTAB(CHAR, 2) = Period

THEN {20. Symbol is a real number. }

CLASS ← 4

{21. see para. 3.3.3.4.4}

END CASE 4

SYML ← SYML + 1

CHARNU ← CHARNU + 1
END CASE 3
END LOOP 3

END CONDITION 2

CONDITION 3 Pointer = 3

{22. Symbol is an undefined operator.}
CLASS ← 8 {23. see para. 3.3.4.4}
SYMCODE ← 2 {24. see para. 3.3.4.3}

LOOP 4 iteration I thru Total_Char
{25. Last character was part of symbol.}
CHAR ← LINST(I)

CASE 5 IF CTAB(I,2) = 1 OR 2
THEN {26. Symbol is formed and SYML = Length of Symbol}

CASE 6 IF SYML > 8
THEN {27. Symbol is syntax error.}
ERCODE ← 1 {28. see para. 3.3.4.1}
RETURN

ELSE {29. Symbol is error free.}
LOOP 5 iteration Num_First_Char thru CHARNU
CHAR ← LINST(iteration)
K ← K + 1
CALL PUTCHE(SYM,K,CHAR)
{30. Symbol is packed four characters per word into array SYM; see state change, para. 3.3.1.5

END LOOP 5
RETURN

END CASE 6

ELSE {32. Symbol may contain more characters.}
SYML ← SYML + 1
CHARNU ← CHARNU + 1

END CASE 5

END LOOP 4

END CONDITION 3

CONDITION 4  Pointer = 4

THEN {33. Symbol is a string.}
CLASS ← 5 {34. see para. 3.3.3.4.4}
SYMCODE ← 3 {35. see para. 3.3.3.4.3}
SYML ← SYML + 1
CHARNU ← CHARNU + 1

LOOP 5 iteration I + 1 thru Total_Char

{36. Last character was part of symbol.}
CHAR ← LINST(iteration)
SYML ← SYML + 1
CHARNU ← CHARNU + 1

CASE 7 IF CTAB(CHAR,2) = QUOTE

THEN {37. Symbol is formed.}
INDEX ← SYML {38. see para. 3.3.3.4.5}

RETURN

END CASE 7

{40. Symbol may contain more characters.}

END LOOP 5

ERCODE ← 2 {41. Syntax error; see para. 3.3.3.4.1}

RETURN
END CONDITION 4

\{ 42. Output Assertions \}

Symbol is FORMED OR ERCODE is  ∈ \{1, 2\}

FORMED means CLASS is  ∈ \{0, 3, 4, 5, 8\}

INDEX is  ∈ \{0, VALUE, INDEX\}

CHARNU is iteration of last character of symbol

SYMCODE is  ∈ \{1, 2, 3\}

ERCODE is 0

END SUBROUTINE FORM

3.5.4 SUBROUTINE TABLE (SYML, CLASS, INDEX)

1. Input Assertions

SYML = the number of characters in the symbol; see para. 3.3.3.2.3

Scanner Tables conform to format figures 2-10 and 2-11
see para. 3.3.3.1.2 and 3.3.3.1.3

CLFLAG see para. 3.3.3.2

SYM(2) contains the characters of the symbol packed four characters per word; see state change,
para. 3.3.3.1.5

SYMCODE = 1, see para. 3.3.3.4.3

INDEX ← 0

Column ← SYML - 1

\{ 2. Variables initialized and set to initial value. \}

CASE 1 IF Column = 0

THEN \{ 3. Symbol has only one character and is not a keyword or command language word. \}

RETURN

END CASE 1
4. Symbol has more than one character, and variable Column represents the column in both the Keyword and Command Language Tables that contain words of the same length.

CASE 2 IF CLFLAG = TRUE

THEN {5. Text is a command language line.}

LOOP 1 iteration 1 thru 10

CASE 3 IF CLTAB(iteration, Column) = SYM(1)

THEN {6. Symbol is a command language word.}

INDEX ← iteration x 10 + Column

{7. INDEX is to word where match was found; see figure 2-11.}

CLASS ← 1 {8. see para. 3.3.2.5}

SYMCODE ← 3 {9. see para. 3.3.3.4.3}

RETURN

END CASE 3

END LOOP 1

{10. A match could not be found; symbol not command language word.}

RETURN

ELSE {11. Text is not a command language line.}

LOOP 2 iteration 1 thru 10

CASE 4 IF KWTABLE(iteration, Column) = SYM(1)

THEN {12. Symbol is a keyword.}

INDEX ← iteration x 10 + Column

{13. INDEX is to word where match was found; see figure 2-11.}

CLASS ← 1 {14. see para. 3.3.2.5}

RETURN

END CASE 4
1. Input Assertions

Scanner Tables + Scanner Variables

Symbol is number

SYML see para. 3.3.3.2.3

Num_First_Char = number of the CHAR in Txt_In

where the symbol starts

INITIALIZE

NUM ← 0

VALUE ← 0

Real_Num ← 0.0

Next_Last_Char ← Num_First_Char + SYML - 1

Table NUMBER /1,2,3,4,5,6,7,8,9,0/

Num_Code ← 0

Decimal_Count ← 0

ERCODE ← 0

2. All internal variables are initialized + external variables reset.
CASE 1  IF SYML > 8
    THEN {3. Syntax error; too many characters in the symbol, see para. 3.3.3.4.1}
        ERCODE ← 3
    RETURN
END CASE 1

{4. Symbol is of legal length.}
LOOP 1 iteration Num_First_Char thru Next_Last_Char
    {5. After each iteration a CHAR has been converted to a numeric value.}
        LOOP 2 iteration 1 thru 10
            CASE 2  IF LINST(Loop 1 iteration) = NUMBER (Loop 2 iteration)
                THEN {6. Character is a digit.}
                    CASE 3  IF Num_Code = 1
                        THEN {7. A period has been encountered.}
                            Decimal_Count ← Decimal_Count + 1
                        ELSE {8. A digit has been added to the symbol.}
                            NUM ← LINST(Loop 1 iteration)
                    END CASE 3
                END CASE 2
            GOTO End Loop 1
        END LOOP 2
    {9. Loop until CHAR = NUMBER OR iteration = 10}
END LOOP 1

{10. Character is a period or error.}
CASE 4  IF LINST(Loop 1 iteration) = Period and Num_Code = 0
    THEN {11. Number is real.}
num_code ← 1
decimal_counter ← 1
else if 12. error illegal character
ercode ← 3
return
end case 4
end loop 1

{13. digits have been packed into a single word.}
case 5 if num_code = 1
then {14. number is real.}
value ← real_num
else {15. number is integer.}
value ← num
end case 5

{16. value assigned numeric value of symbol.}
return

{17. output assertions
ercode ∈ {0, 1}
and ercode = 0  no error was encountered
value = numeric value of symbol
and ercode = 1  symbol exceeded 8 characters
or ercode = 3  illegal character in symbol
value = null
end subroutine numpac

3.5.6 subroutine serror

{1. input assertions - scanner input assertions + scanner tables + scanner variable
errct ≥ 1  see para. 3.3.3.2.2
errst  see state change, para. 3.3.3.1.7}
INITIALIZE

String_Too_Many_Char/Sym has too many characters/
String_No_Close_Quote/No closing quote mark/
String_Illegal_Char/Illegal character/
RETCOD ← 0

\{2. Error messages and Error return code are initialized.\}

LOOP 1 iteration 1 thru ERRCT

\{3. After each iteration an error message has been printed to the user.\}

Line_Num ← ERRST(iteration,1)
CHAR ← ERRST(iteration,2)
RETCODE ← ERRST(iteration,3)

\{4. Variables are set to Error_Data for the line of text.\}

\begin{align*}
\text{CONDITION 1} & \quad \text{RETCODE} = 1 \\
\text{\{5. Error was too many characters in the symbol.\}} & \quad \text{THEN } N \leftarrow 21 \quad \text{\{6. } N = \text{ number of characters in message} \} \\
& \quad \text{CALL ERPPRT (PINDEX, Line_Num, CHAR, N, String_Too_Many_Char, RETCOD)} \\
& \quad \{7. see para. 3.4.4\}
\end{align*}

\text{END CONDITION 1}

\begin{align*}
\text{CONDITION 2} & \quad \text{RETCODE} = 2 \\
\text{\{8. Error was no closing quote mark on a string.\}} & \quad \text{THEN } N \leftarrow 26 \quad \text{\{9. } N = \text{ number of characters in message} \} \\
& \quad \text{CALL ERPPRT (PINDEX, Line_Num, CHAR, N, String_Illegal_Char, RETCOD)} \\
& \quad \{10. see para. 3.4.4\}
\end{align*}

\text{END CONDITION 2}
CONDITION 3  ERRCODE = 3

{11. Error was illegal character.}

THEN  N ← 17  {12. N = number of characters in message}

CALL ERRPRT (PINDEX,Line_Num,CHAR,N,String_Illegal_Char,RETCOD)

{13. see para. 3.4.4}

END CONDITION 3

CASE 1  IF RETCOD > 0

THEN {14. Error message cannot be printed.}

RETURN

END CASE 1

END LOOP 1

{15. Txt_Ln with syntax error message has been printed.}

RETURN

{16. Output Assertions

Txt_Ln with syntax error that RETCODE = 0
a message was printed to user.

END SUBROUTINE SERROR

3.5.7 SUBROUTINE LINFIN (Txt_Ln_Disp,RESCAN,Tok_Ln_Disp)

{1. Input Assertions - Scanner Input Assertions + Scanner Flags + Scanner Variables

RESCAN = TRUE  see para. 3.3.3.1

Txt_Ln_Disp = displacement to current sequential Txt_Ln

Tok_Ln_Disp = displacement to current sequential Tok_Ln

INITIALIZE

Tok_Last_Ln_Num ← 0

Last_Ln_Disp ← 3
{2. Internal variables are initialized.}
Txt_Add@ ← HEAP(Txt_Log_Add) {3. see para. 3.3.1.3}
Tok_Add@ ← HEAP(Tok_Log_Add) {4. see para. 3.3.2.3}
Tok_Total_Ln ← HEAP(Tok_Add@ + TOKNLIN) {5. see para. 3.3.2.4}
Ln_Displ ← HEAP(Tok_Add@ + TOKTOP) {6. see para. 3.3.2.4}
Txt_Ln_Num ← HEAP(Txt_Add@ + Ln_Displ + TXTLN)
{7. see para. 3.2.1.5}

LOOP 1 iteration 1 thru Tok_Total_Ln

CASE 1 IF Txt_Ln_Num > Tok_Last_Ln_Num AND Txt_Ln_Num ≤

THEN {8. Relative position of text line is found}
in token lines.

CASE 2 IF Txt_Ln_Num = Tok_Ln_Num

THEN {9. Token line is a rescan of a text}
line that has been scanned before
and a token line header for this
text line exists.

HEAP(Tok_Add@ + Ln_Displ + TOKFLK)←
HEAP(Tok_Add@ + Ln_Displ + TOKFLK)

{10. Sets forward link to the next
token line in the new token
line header for this text line.}

HEAP(Tok_Add@ + Last_Ln_Displ + TOKFLK)←
Tok_Ln_Displ

{11. Sets forward token line link
in the token prior line to the
line displacement to the new
token line for this text line.}

ELSE {12. This is a new text line that has
been added between existing lines
of text and no token line exists.}

{13. Sets forward token line link
to final line in text file.
HEAP(Tok_Add@ + Tok_Ln_Displacement + TOKFLK) ←
        Ln_Displacement
HEAP(Tok_Add@ + Last_Ln_Displacement + TOKFLK) ←
        Tok_Ln_Displacement

{13. New token line has been linked
    in its position with existing
token lines.
}
END CASE 2

RETURN

ELSE {14. Relative position of text line has not
    been found.
}
Last_Ln_Displacement ← Ln_Displacement
Ln_Displacement ← HEAP(Tok_Add@ + Last_Ln_Displacement + TOKFLK)
Tok_Last_Ln_Num ← Tok_Ln_Num
Tok_Ln_Num ← HEAP(Tok_Add@ + Ln_Displacement + TOKFNUM)

{15. Data for next line number comparison is
    set.
}
END CASE 1

END LOOP 1

{16. Text line number is greater than last token line number.
    Source text lines are an addition to those lines that
    had been scanned prior and all additional lines to be
    scanned can be treated as no prior scan of source text.
}
RESCAN ← FALSE

RETURN

{17. Output Assertions
    Tok_Ln corresponds to Txt_Ln in source text thru the
    present Txt_Ln
    and RESCAN ≠ TRUE if Txt_Ln_Num ≥ Last Tok_Ln_Num
}

END SUBROUTINE LINFIN
Chapter 4
Module Verification

4.1 **General**

The informal verification of the Scanner module will follow the sequential statements of each subroutine as it is called. Output assertions of each subroutine called will be considered valid for the subroutine being verified. Verification will consist of explanations, tracing the state transition of the module, and verifying loop invariants to insure each routine meets its output assertions and terminates. A subroutine is considered verified if the specifications and output assertions are met through the routine's state transition.

4.2 **Subroutine SCAN**

See figure 4-1 for State Transition Diagram.

4.2.1 **Assertions 1 through 6 follow sequentially.**

4.2.2 At assertion 12 either:

A new Token Block was established (Path 7,8,10,12).

A Token Block already exists (Path 11,12).

**Both Paths** at assertion 12, a Token Block exists and the Token Logical Address is stored in the Procedure Table.

4.2.3 **Assertion 13 through 15 follow sequentially.**

4.2.4 At assertion 17 either:

The **First** Token Line Displacement is set to 7 for newly established Token Blocks (Path 15,16,17).

The **First** Token Line Displacement is already set for previously existing Token Blocks (Path 15,17).
Figure 4-1 Subroutine SCAN State Transition Diagram
Figure 4-1 (Continued)

All Text lines have been scanned and a corresponding Tokn line exists for each line of text.

Syntax errors were encountered in the source text.

Error messages were printed if terminal was available.

Termination for source text and token data is complete for each text line.

Loop iteration Text line.

Address pointers have been updated.

Termination path error in Tok Block.

Token in corresponding Tokn line has been stored in Tok Block and Token Header has been updated.

Token to load from Tok Block.

Path to load token.

Text line had a syntax error, no token data generated.
Both Paths at assertion 17, the Token Block Header is correct.

4.2.5 At assertion 18; loop invariant:

Initially all address pointers are set prior to entering the loop and for each subsequent iteration, pointers are set prior to the end of the loop (Path 1 through 17 and Path 43 through 43.4).

After each iteration, a Token Line has been stored in the Token Block for the preceding Text Line (Path 33.1 to 43).

4.2.6 At assertion 22 either:

A Text Line has been deleted from a previously scanned Text Block and the corresponding Token Lines have been deleted from the Token Block (Path 18,19,20,21,22).

All remaining text lines in a Text Block that has been previously scanned have been added and no corresponding Token Lines exist. RESCAN is set to FALSE (Path 18,19,20,21,22).

The Token Block is newly established, or a Text Line has been added, or the Token Line number corresponds to the current Text Line number (Path 18,22).

All Paths at assertion 22, the Text Line number is equal to or greater than the current Token Line number or the Token Block is newly established.

4.2.7 At assertion 23, the Text Line requires scanning.

4.2.8 At assertion 24, the Text Line has been scanned and either:

Token Data is in array SYMST.

Error Data is in array ERRST.
Storage Error was encountered and control is sent to assertion 48.

4.2.9 At assertion 30 either:

Sufficient space was available in the Token Block to store the current Token Line (Path 25,1,30).

Sufficient space was not available in the Token Block to store the current Token Line and additional space was allocated (Path 25,1,26,27,29,30).

Both Paths at assertion 30, sufficient storage space exists in the Token Block to store the current Token Line.

4.2.10 Assertion 30 through 32 follow sequentially.

4.2.11 At assertion 33.1 either:

The Token Line links have been changed to include the revised Token Line of a previously scanned Text Block (Path 32,33,33.1).

The Token Block is newly established (Path 32,33.1).

Both Paths at assertion 33.1, Token Line is ready to be stored in the Token Block.

4.2.12 At assertion 38 either:

The Text Line contained a syntax error and no Token Data was generated (Path 33,1,35,36,37,38).

The Token Data for the symbols of the Text Line have been stored in the Token Block (Path 33,1,34,37,38).

Both Paths, the Token Line has been stored and the Token Block Header has been updated.

4.2.13 At assertion 42, no user interrupt was received.

4.2.14 At assertion 43, no storage error occurred.

4.2.15 At assertion 43.4, all line pointers for both the
Token and Text Blocks have been set for the next loop iteration.

4.2.16 At assertion 45, all text lines in the Text Block have been scanned and corresponding Token Lines exist in the Token Block.

4.2.17 At assertion 48 either:

The Text Block contained syntax errors and error messages were printed if the terminal was available (Path 46, 47, 48).

Storage errors occurred in the Token Block (Path 44, 48).
Storage errors occurred in SYMTAB module (Path 24, 48).

All Paths at assertion 48, termination, Token Block does not contain complete Token Data for the source text.

4.2.18 At assertion 49, termination, Token Block contains an error free token representation of the source text.

4.2.19 Assertion 50 Output

Termination at:

Assertion 9, space not available for Token Block.

Assertion 28, additional space not available to expand allocated Token Block.

Assertion 41, user interrupt occurred.

Assertion 48, syntax error in source text or storage error.

Assertion 49, error free token representation exists.

4.3 Subroutine LNSCAN

See figure 4-2 for State Transition Diagram.

4.3.1 Assertions 1 through 3 follow sequentially.

4.3.2 At assertion 6; loop invariant:
The sequential character in a Text Line has been placed in the corresponding sequential storage location of array LINST.

4.3.3 At assertion, array LINST contains the complete list of characters of the Text Line.

4.3.4 At assertion 9 either:

The Text Line is a command language line (Path 7,8,9).
The Text Line is a program line (Path 7,9).
Both Paths Command language flag is set for this line.

4.3.5 At assertion 10; loop invariant:

Initially the character is a symbol or the first character of a symbol.

After each iteration:
The character was part of a symbol already formed (Path 10,11).
The character was a blank (Path 10,12,13,14).
The character was a separator or operator and Token Data is stored in array SYMST (Path 10,12,13,15,16).
The character was the first character of one of the following symbols and Token Data is stored in array SYMST.
A number or string (Path 10,12,13,17,20,23,29,30,31).
A key or command word (Path 10,12,13,17,20,23,24,29,30,31).
An identifier (Path 10,12,13,17,20,23,24,25,26,28,29,30,31).
An undefined operator (Path 10,12,13,17,20,23,26,28,29,30,31).
SUBROUTINE LNSCAN

A character has been placed in array LINST from the Txt Ln.

Array LINST contains the complete list of characters of the text line.

Ttxt Ln is a program line.

Ttxt Ln is a Command Line.

Character is already part of a symbol.

Character is a blank, a symbol, or the first character of a symbol.

Character is a blank.

Character is a separator or an operator.

Symbol is stored in array SYMST.

Symbol is formed.

CONTINUED

Figure 4-2 Subroutine LNSCAN State Transition Diagram
Figure 4-2 (Continued)
4.3.6 At assertion 32 either:

The Text Line contained a syntax error and error data
is stored in array ERRST (Path 20,22,32).

All symbols of the Text Line have been formed and Token
Data is stored in array SYMST (Path loop completion,32).

4.3.7 Terminations either:

At assertion 32, Text Line has been scanned.

At assertion 27, storage error in Subroutine SYMTAB.

4.4 Subroutine FORM

See figure 4-3 for State Transition Diagram.

4.4.1 Assertions 1 through 3 follow sequentially.

4.4.2 At assertion 15, symbol is a number.

(2.1) At assertion 16; loop invariant:

Initially first character is part of the symbol. After
each iteration; last character was part of the symbol and
the symbol value is:

The number is integer (Path 16,19,19.1).

The number is real (Path 16,19,20,21,29,1).

(2.2) At assertion 18 either:

There are more characters in the Text Line (Path 16,17,
18).

The last character in the Text Line is part of the symbol
(Path loop completion,17.1,18).

Both Paths termination, number is formed and Token Data
developed.

4.4.3 At assertion 33, symbol is a string:

(3.1) Assertion 33 through 35 follow sequentially.

(3.2) Assertion 36; loop invariant:
SUBROUTINE FORM

Symbol is a number.
Symbol formed
Symbol is a string.
Symbol is a key or command word, or identifier.
Symbol is a undefined operator.

Symbol class remains integer
Symbol class is changed to real
This char is part of symbol

Last char in Ln is part of symbol.

Error path
Termination Number is formed and token data developed.
Termination Syntax error.

Next char is part of the symbol.
Symbol is formed.

Termination String is formed and token data is developed.
Termination Syntax error.

Figure 4-3 Subroutine FORM State Transition Diagram
Symbol is a key or command word, or an identifier.

Symbol is defined.

Character is part of the symbol.

Symbol is formed.

Last char in line was part of symbol.

Error path.

Termination Syntax error.

Symbol has been packed into array SYM.

Termination Symbol is formed and characters packed in array SYM.

Figure 4-3 (Continued)
The character corresponding to the loop iteration is part of the string.

(3.3) At assertion 39; termination, string is formed and Token Data developed.

4.4.4 At assertion 41 either:

Syntax error, illegal character in the number (Path from assertion 15 to assertion 41).

Syntax error, no closing quote mark on string (Path from assertion 33 to assertion 41).

4.4.5 At assertion 4, symbol is a keyword, command word, or an identifier.

(5.1) Assertion 4 through 5 follow sequentially.

(5.2) At assertion 6; loop invariant:

Initially character is part of symbol.

For each iteration:

Character is part of symbol (Path 6,14).

Character is a separator, operator, or blank (Path 6,7).

(5.3) At assertion 14.1 either:

Text Line contains more symbols (Path 6,7,14.1).

The last character in the Text Line is part of this symbol (Path loop completion,14.1).

Both Paths symbol is formed.

4.4.6 At assertion 22, symbol is an undefined operator.

(6.1) Assertion 22 through 24 follow sequentially.

(6.2) At assertion 25; loop invariant:

Initially character is part of symbol.

For each iteration:
Character is part of symbol (Path 25,32).
Character is a separator, operator, or blank (Path 25,26).
(6.3) At assertion 26.1 either:
Text Line contains more symbols (Path 25,26,26.1).
The last character in the Text Line is part of this symbol (Path loop completion,26.1).
4.4.7 At assertion 13 either:
Symbol is a keyword, command word, or an identifier, and the characters of the symbol are packed into array SYM (Path assertion 4 path to 10,13).
Symbol is an undefined operator and the characters of the symbol are packed into array SYM (Path assertion 22 path, 29, 13).
Both Paths termination, symbol is formed and packed in array SYM.
4.4.8 At assertion 42, termination;
Syntax error, too many characters in symbol (Path assertion 4 path to 10,8,9,42 or assertion 22 path to 29,27,9,42).
4.5 Subroutine TABLE
See figure 4-4 for State Transition Diagram.
4.5.1 Assertions 1 through 2 follow sequentially.
4.5.2 At assertion 3, termination, symbol is an one letter identifier.
4.5.3 At assertion 17.2 either:
Text Line is a program line and symbol is a keyword (Path 4,11,12,13).
Text Line is a command line and symbol is a command word (Path 4,5,6,13).
Figure 4-4 Subroutine TABLE State Transition Diagram
Both Paths termination, symbol was matched in tables and
Token Data has been assigned.

4.5.4 At assertion 17.1 either:

Text Line is a program line and symbol is not a keyword
(Path 4,11,16,17.1).

Text Line is a command line and symbol is not a command
language word (Path 4,5,10,17.1).

Both Paths termination, symbol is an identifier.

4.6 Subroutine NUMPAC

See figure 4-5 for State Transition Diagram.

4.6.1 Assertions 1 through 2.1 follow sequentially.

4.6.2 At assertion 4:
The symbol is within allowable character length.

4.6.3 At assertion 5; loop invariant:
Initially first character is a digit.
After each iteration:
Character was a digit.
Character was the first period encountered.

4.6.4 At assertion 9; loop invariant:
After each iteration; character is either:
A digit and the digit \( \geq \) the last loop iteration.
Not a digit.

4.6.5 At assertion 8; The character was a digit and the
digit has been added to the symbol, and either:
A period has been encountered in a previous loop
iteration and the decimal counter has been increased (Path
9,6,7,8).
A period has not been encountered in a previous loop
iteration (Path 9,6,8).
4.6.6 At assertion 11: The character is the first period encountered in the symbol, the decimal counter is assigned the value one, and the number code is changed to indicate a real value.

4.6.7 At assertion 13: All characters of the symbol have been placed in order in a single storage word.

4.6.8 At assertion 16 either:

The value of the symbol is real and the real value of the symbol has been stored in array RVALUE and variable VALUE has been assigned the index into array RVALUE (Path 14, 14.1, 14.2, 14.3, 14.4, 14.5, 16).

The value of the symbol is integer and the integer value of the symbol has been assigned to variable VALUE (Path 15, 16).

Both Paths termination, the value of the symbol has been developed for the Token Data.

4.6.9 At assertion 17; Output Assertions;

Termination at:

Assertion 3, syntax error, too many characters in the symbol.

Assertion 12, syntax error, illegal character, either:

More than one period in the symbol or character was not a digit or a period.

Assertion 16, symbol value has been determined for Token Data.

4.7 Subroutine SERROR

See figure 4-6 for State Transition Diagram.

4.7.1 Assertions 1 through 2 follow sequentially.

4.7.2 At assertion 3; loop invariant:
Figure 4-5 Subroutine NUMPAC State Transition Diagram
After each iteration, an error message was printed to the user.

(2.1) Assertions 3 through 4 follow sequentially.

(2.2) At assertion 14 either:

Error message too many characters in symbol (Path 8, 9, 10, 14).

Error message no closing quote on a string (Path 8, 9, 10, 14).

Error message illegal character (Path 11, 12, 13, 14).

All Paths error message has been printed to user.

4.7.3 At assertion 16 either:

All error messages have been printed for text lines that contained syntax errors (Path loop completion, 15, 16).

Terminal was not available to print error message (Path 14, 16).

Both Paths termination, error messages were printed for all text lines that contained syntax errors, one line at a time, as long as a terminal was available.

4.8 Subroutine LINFIN

See figure 4-7 for State Transition Diagram.

4.8.1 Assertions 1 through 7 follow sequentially.

4.8.2 Loop Termination either:

Relative position of the Text Line is found in the Token Block (Path 8).

Text Line has been added to total lines of the Text Block (Path loop completion).

4.8.3 At assertion 17 either:

Termination, all subsequent Token Lines are added to the
SUBROUTINE SERROR

Too many characters in symbol.

Illegal character.

No closing quote string.

Error message has been printed.

Terminal not available to print error message.

All error messages have been printed for Txt In's that contained syntax errors.

Termination
A error message was printed for each Txt In with a syntax error or terminal was not available to print message.

Figure 4-6 Subroutine SERROR State Transition Diagram
end of the Token Block. RESCAN set to FALSE. (Path loop completion, 16, 17).

Text Line replaces a previously scanned Text Line (Path 8, 12, 13, 17).

Text Line has been added within previously scanned Text Block (Path 8, 9, 10, 11, 17).

All Paths termination, Token Line position has been established with respect to block line linkage.
SUBROUTINE LINFIN

\[ \text{Relative position of the new Txt Ln has been found in the Token Block.} \]

\[ \text{Line pointers have been set for the next line comparison.} \]

\[ \text{Txt Ln has been added to the end of a previously scanned Text Block.} \]

\[ \text{Termination} \]
\[ \text{Remainder of Text Block to be scanned as an initial Text Block scan.} \]

\[ \text{Txt Ln has been added to previously scanned Text Block.} \]

\[ \text{Txt Ln replaces a previously scanned Txt Ln.} \]

\[ \text{Token Line links have been set to replace old Token Line.} \]

\[ \text{Termination} \]
\[ \text{Token line header has been established for the new token line.} \]

Figure 4-7 Subroutine LINFIN State Transition Diagram
Chapter 5
Conclusions

5.1 General

The conclusions stated in this chapter are based on the revision and refinement of a previously developed Scanner program by use of assertions. Although the primary purpose of this report was the verification of the Scanner module, consideration was given to the feasibility of using this technique for program design with respect to time, amount of documentation required, and the validity of the final product. It was necessary for purposes of this report to refine global and input assertions, and global procedures to the same degree as that of the Scanner module. This documentation would be required for all modules of the interpreter program and should not be considered unique to this report.

Personal involvement with the development of the original version of the Scanner module undoubtedly was a significant asset in the verification process. However, this involvement was of little consequence to the procedure used to define global, input, and output assertions from the English specifications which were accomplished independently of the existing program. Specific procedures, such as address accessing and precise format definitions of data structures developed by assertion refinement could be used in all modules of the interpreter program to verify these particular operations independently of individual programmers.
5.2 Verification

All subroutines of the Scanner module were verified in Chapter 4 by tracing the state transition of each routine along its assertion paths. This method insured for all routines called that:

The input assertions were meet.
That all loops would terminate.
That each routine would only terminate at specific points which would meet the established output assertions.

The original version of the Scanner module did execute correctly using standard test data developed for all modules of the interpreter program. This data, however, did not verify all paths or termination points of the module. Four points were found during the verification process where the program could produce erroneous data. No significant reduction of program lines was accomplished; however, some modification of the module was made to eliminate repetitious operations or to facilitate sequential flow. The verified FORTRAN code contained in Annex C can be used with high assurance that it will function correctly.

It is feasible that a formal proof could be accomplished from the presented documentation by development of additional assertions. A formal proof, however, would not in itself necessarily improve the readability of this large of a program. The graphic portrayal of the module depicted by the state transition diagrams while only providing an informal verification of the module is less time consuming and does trace all program paths through developed assertions.
The following observations are provided concerning the verification of the module.

5.2.1 Documentation

Approximately 40 pages of documentation, which included 20 pages of verifications, was necessary for the 375 lines of FORTRAN code of the Scanner module.

5.2.2 Assertion Refinement

The majority of variables used in the Scanner module were developed and given precise definitions during the assertion refinement. The assertion refinement also provided for the modular development of specific operations that were required in the Scanner routines. Assertion refinement in both cases insured predictable results when these operations and variables were incorporated into program code.

5.2.3 Input Assertions

The use of input assertions effectively limited the scope of module and established the required data structure contents at the outset of the Scanner modular development.

5.2.4 Output Assertions

The development of precise output assertions based on the program specifications provides the required termination conditions of the program prior to coding. This basically established termination bench marks for the development of the code rather than the termination points being developed during coding. This in itself reduced the total number of code lines used in the module when compared to versions that were developed without using the assertion method. This procedure also eliminated the need for modification of
developed program code to meet input requirements of global or submodular procedures that were developed simultaneously with the Scanner module.

5.3 Application

Some areas where advantages could be derived by adoption of this method of program verification are coordination of program efforts, documentation, development of test data, and debugging.

Refined input and output assertions provide an effective method for coordination of programming efforts between individuals or teams. The precise definition of parameters and data structures can eliminate misunderstandings and insure compatibility of efforts.

By including assertions as part of the program documentation, modification and maintenance can be facilitated. Modification of a program can be accomplished by first making necessary changes to the output assertions and then either correcting existing termination points or adding additional termination paths. The effect a specific change will have on other levels of a program can also be more easily determined.

Test data that will exercise all paths of a program can be developed from refined assertions with predictable results. This has a direct application to program debugging. A great deal of program debugging can also be accomplished during the verification process, thus reducing the number of actual computer runs necessary for this process.

5.4 Recommendations

An established shorthand-type notation greatly assists
in the assertion refinement process. The notation selected should be both readable and easily converted to codable procedures and variable names.

Assertions should be refined in as much detail as possible prior to development of the program. The refined assertions can effectively define procedures and operations needed in the program and establish the exact termination conditions for the routine. The program can then be developed to specifically meet the required termination points.

Detailed specifications are necessary in order to develop the precise output assertions of a program. However, output assertions can also be very useful in validating the designer's understanding of the specifications of the program.
Annex A: References


Annex B: Initialization Data for Data Structures

B.1 Global Data Structures

B.1.1 General

The initialization data listed in this section is for the offset variable names of the global data structures of the interpreter program that are described in Chapter 2 of this report. The offsets associated with each data structure are presented in accordance with their format. The initialization of the offsets and maintenance of the global data structures in accordance with their formats are considered global assertions for the FORTRAN implementation of the Scanner.

B.1.2 Data Structure Formats

(2.1) HEAP Storage Area (figures 2-2, 2-3, 2-5)

The HEAP storage area consists of all memory space associated with the interpreter program for the purpose of manipulation of the source program. The offsets and their values presented for the HEAP data area are also used by allocated data blocks.

<table>
<thead>
<tr>
<th>Memory Words</th>
<th>Type Data Stored</th>
<th>Offset Variable</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Data Area Code</td>
<td>HPTAG = 1</td>
</tr>
<tr>
<td>2</td>
<td>Allocated Space</td>
<td>HPSPAC = 2</td>
</tr>
<tr>
<td>3</td>
<td>Space Used</td>
<td>HPSIZ = 3</td>
</tr>
<tr>
<td>4</td>
<td>Logical Address of Data Area</td>
<td>HPLAD = 4</td>
</tr>
<tr>
<td>5</td>
<td>Index to the Procedure Symbol Table data block, value represents the displacement from the start of the HEAP area to the location prior to the start of the Procedure Symbol Table data block.</td>
<td>NONE</td>
</tr>
</tbody>
</table>
(2.2) Procedure Table Block (figure 2-3)

The Procedure Table data block consists of a four word block header and the individual Procedure Tables which consist of eight contiguous storage locations. A PINDEX represents the displacement of a specific Procedure Table from the start of the Procedure Table Block area. Access to a specific process's Procedure Table is made by adding the value of HEAP location 5 and the value of PINDEX.

Procedure Table:

<table>
<thead>
<tr>
<th>Memory Word</th>
<th>Type Data Stored</th>
<th>Offset Variable</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Number of characters in Procedure name</td>
<td>PRCNLEN = 1</td>
</tr>
<tr>
<td>2</td>
<td>First four characters of Procedure name</td>
<td>PRCNAME = 2</td>
</tr>
<tr>
<td>3</td>
<td>Last four characters of Procedure name</td>
<td>NONE</td>
</tr>
<tr>
<td>4</td>
<td>Procedure Status</td>
<td>PRCSTAT = 4</td>
</tr>
<tr>
<td>5</td>
<td>Logical Address for Text Location</td>
<td>PRCXTXT = 5</td>
</tr>
<tr>
<td>6</td>
<td>Logical Address for Token Location</td>
<td>PRCSTOK = 6</td>
</tr>
<tr>
<td>7</td>
<td>Logical Address for Symbol Table Location</td>
<td>PRCSYM = 7</td>
</tr>
<tr>
<td>8</td>
<td>Logical Address for Code Location</td>
<td>PRCCOD = 8</td>
</tr>
</tbody>
</table>

(2.3) Text Block (figures 2-5, 2-6)

The text storage block associated with a procedure contains text header information followed by the lines of text. Each line of text has line header information followed by the character of the line, packed four characters per storage location.

(2.3.1) Text Header

There are seven memory words associated with the header
information of each Text Block.

<table>
<thead>
<tr>
<th>Memory Word</th>
<th>Type Data Stored</th>
<th>Offset Variable</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Text code representation value</td>
<td>HPTAG = 1</td>
</tr>
<tr>
<td>2</td>
<td>Allocated storage space for text</td>
<td>HPSFAC = 2</td>
</tr>
<tr>
<td>3</td>
<td>Used storage space</td>
<td>HPSIZ = 3</td>
</tr>
<tr>
<td>4</td>
<td>Logical Address to Text storage block</td>
<td>HPLAD = 4</td>
</tr>
<tr>
<td>5</td>
<td>Pointer to first line of text</td>
<td>TXTTOP = 5</td>
</tr>
<tr>
<td>6</td>
<td>Pointer to last line of text</td>
<td>TXTBOT = 6</td>
</tr>
<tr>
<td>7</td>
<td>Total lines of text in storage area</td>
<td>TXTNLIN = 7</td>
</tr>
</tbody>
</table>

(2.3.2) Text Line Header

There are seven memory words associated with the header information for each line of source text stored in the Text Block.

<table>
<thead>
<tr>
<th>Memory Word</th>
<th>Type Data Stored</th>
<th>Offset Variable</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Text line number</td>
<td>TXTLNUNM = 1</td>
</tr>
<tr>
<td>2</td>
<td>Index from the first memory word of the text storage area to the next line of text</td>
<td>TXTFLK = 2</td>
</tr>
<tr>
<td>3</td>
<td>Index from the first memory word of the Text storage area to the previous line of text</td>
<td>TXTBLK = 3</td>
</tr>
<tr>
<td>4</td>
<td>Value either 0 if line has not been scanned or 1 if line has been scanned</td>
<td>TXTTOK = 4</td>
</tr>
<tr>
<td>5</td>
<td>Nesting level</td>
<td>TXTNST = 5</td>
</tr>
<tr>
<td>6</td>
<td>Line status</td>
<td>TXTSTAT = 6</td>
</tr>
<tr>
<td>7</td>
<td>Value representing the number of characters in the line of text</td>
<td>TXTLLN = 7</td>
</tr>
</tbody>
</table>

(2.4) Token Block (figures 2-5, 2-7)

The token storage block associated with a procedure contains token header information followed by lines of tokens which correspond to the line numbers of the procedure text.
lines. The first three memory words of each line contain line header information. The header information is then followed by the line tokens. Each token requires three memory words. 

(2.4.1) Token Header

There are seven memory words associated with the header information of each Token Block.

<table>
<thead>
<tr>
<th>Memory Word</th>
<th>Type Data Stored</th>
<th>Offset Variable</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Token code representation</td>
<td>HPTAG = 1</td>
</tr>
<tr>
<td>2</td>
<td>Allocated storage space</td>
<td>HPSPAC = 2</td>
</tr>
<tr>
<td>3</td>
<td>Used storage space</td>
<td>HPSIZ = 3</td>
</tr>
<tr>
<td>4</td>
<td>Logical Address to token storage area</td>
<td>HPLAD = 4</td>
</tr>
<tr>
<td>5</td>
<td>Pointer to first line of tokens</td>
<td>TOKTOP = 5</td>
</tr>
<tr>
<td>6</td>
<td>Pointer to last line of tokens</td>
<td>TOKBOT = 6</td>
</tr>
<tr>
<td>7</td>
<td>Total lines of tokens in storage area</td>
<td>TOKNLIN = 7</td>
</tr>
</tbody>
</table>

(2.4.2) Token Line Header

There are three memory words associated with the header information for each line of tokens in the Token Block.

<table>
<thead>
<tr>
<th>Memory Word</th>
<th>Type Data Stored</th>
<th>Offset Variable</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Line number</td>
<td>TOKLNUM = 1</td>
</tr>
<tr>
<td>2</td>
<td>Index from the first memory word of the Token storage area to the next line of tokens</td>
<td>TOKFLK = 2</td>
</tr>
<tr>
<td>3</td>
<td>Total number of tokens in line</td>
<td>TOKNUM = 3</td>
</tr>
</tbody>
</table>

(2.4.3) Token

There are three memory words associated with each token.

<table>
<thead>
<tr>
<th>Memory Word</th>
<th>Type Data Stored</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Class identifier (see a below)</td>
</tr>
<tr>
<td>2</td>
<td>Index or Value (see b below)</td>
</tr>
<tr>
<td>3</td>
<td>Number of the first character of the symbol in the line of source text</td>
</tr>
</tbody>
</table>
(2.4.3.1) Class Identifiers

The below listed integer values are representative of the type of symbols identified by the Scanner.

<table>
<thead>
<tr>
<th>Integer Value</th>
<th>Type Symbol</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Keyword or Command Language Word</td>
</tr>
<tr>
<td>2</td>
<td>Identifier</td>
</tr>
<tr>
<td>3</td>
<td>Integer number</td>
</tr>
<tr>
<td>4</td>
<td>Real number</td>
</tr>
<tr>
<td>5</td>
<td>String</td>
</tr>
<tr>
<td>6</td>
<td>Operator</td>
</tr>
<tr>
<td>7</td>
<td>Separator</td>
</tr>
<tr>
<td>8</td>
<td>Undefined Operator</td>
</tr>
</tbody>
</table>

(2.4.3.2) Index or Value

The second memory word of a token contains either an index that can be associated with:

1) Keywords
2) Command Language Words
3) Identifiers
4) Operators
5) Separators
6) Undefined Operators

or the memory word will contain a numeric value that can be associated with:

1) Integer number
2) Real number
3) The length by number of characters of a string.

B.2 Local Data Structures

B.2.1 General

This section contains the initialization data for the local data structures used in the Scanner module. The initialization data used in these structures may be changed or added to as the language interpreter program is expanded, modified,
or transported to different environments. The initialization data represented below is that which was used with the module during the FORTRAN execution of the Scanner program for this report. EBCDIC graphics and controls were selected for program implementation.

B.2.2 Character Table (255,3) Array

This table is initialized according to the machine code character translation table selected for use with the interpreter program. The size of the array was determined by the machine decimal representation of characters as between decimal 1 and 255. Entry into the table is based on the machine decimal representation of the character being scanned.

(2.1) Column one - designates the character as a letter or number, a separator or operator, or a blank.

one = blank
two = separator or operator
three = letter or number

(2.2) Column two - designates the character as a letter, number, $, '', operator, or separator.

zero = blank
one = letter
two = number
three = $
four = ''
six = operator
seven = separator

(2.3) Column three - designates the index number for separators and operators for use by other modules of the interpreter program. Separators and operators are numbered consecutively as they are encountered by type in row order of the character
(2.4) Character Table initialization data

<table>
<thead>
<tr>
<th>EBCDIC Characters</th>
<th>Decimal Location</th>
<th>Column 1</th>
<th>Column 2</th>
<th>Column 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>blank</td>
<td>1 - 73</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$</td>
<td>74</td>
<td>2</td>
<td>6</td>
<td>1</td>
</tr>
<tr>
<td>&lt;</td>
<td>75</td>
<td>2</td>
<td>7</td>
<td>1</td>
</tr>
<tr>
<td>(</td>
<td>76</td>
<td>2</td>
<td>6</td>
<td>2</td>
</tr>
<tr>
<td>+</td>
<td>77</td>
<td>2</td>
<td>7</td>
<td>2</td>
</tr>
<tr>
<td>:</td>
<td>78</td>
<td>2</td>
<td>6</td>
<td>3</td>
</tr>
<tr>
<td>&amp;</td>
<td>79</td>
<td>2</td>
<td>6</td>
<td>4</td>
</tr>
<tr>
<td>blank</td>
<td>80</td>
<td>2</td>
<td>6</td>
<td>5</td>
</tr>
<tr>
<td>!</td>
<td>81 - 89</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$</td>
<td>90</td>
<td>2</td>
<td>7</td>
<td>3</td>
</tr>
<tr>
<td>*</td>
<td>91</td>
<td>3</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>)</td>
<td>92</td>
<td>2</td>
<td>7</td>
<td>6</td>
</tr>
<tr>
<td>;</td>
<td>93</td>
<td>2</td>
<td>7</td>
<td>4</td>
</tr>
<tr>
<td>/</td>
<td>94</td>
<td>2</td>
<td>7</td>
<td>5</td>
</tr>
<tr>
<td>blank</td>
<td>95</td>
<td>2</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>:</td>
<td>96</td>
<td>2</td>
<td>6</td>
<td>8</td>
</tr>
<tr>
<td>%</td>
<td>97</td>
<td>2</td>
<td>6</td>
<td>9</td>
</tr>
<tr>
<td>?</td>
<td>98 - 105</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>\</td>
<td>106</td>
<td>2</td>
<td>7</td>
<td>6</td>
</tr>
<tr>
<td>.</td>
<td>107</td>
<td>2</td>
<td>6</td>
<td>10</td>
</tr>
<tr>
<td>#</td>
<td>108</td>
<td>2</td>
<td>6</td>
<td>11</td>
</tr>
<tr>
<td>@</td>
<td>109</td>
<td>2</td>
<td>6</td>
<td>12</td>
</tr>
<tr>
<td>=</td>
<td>110</td>
<td>2</td>
<td>6</td>
<td>13</td>
</tr>
<tr>
<td>blank</td>
<td>111</td>
<td>2</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>\</td>
<td>112 - 120</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>.</td>
<td>121</td>
<td>2</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>.</td>
<td>122</td>
<td>2</td>
<td>7</td>
<td>9</td>
</tr>
<tr>
<td>#</td>
<td>123</td>
<td>2</td>
<td>6</td>
<td>14</td>
</tr>
<tr>
<td>@</td>
<td>124</td>
<td>2</td>
<td>6</td>
<td>15</td>
</tr>
<tr>
<td>=</td>
<td>125</td>
<td>2</td>
<td>7</td>
<td>10</td>
</tr>
<tr>
<td>blank</td>
<td>126</td>
<td>2</td>
<td>6</td>
<td>16</td>
</tr>
<tr>
<td>a</td>
<td>127</td>
<td>3</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>b</td>
<td>128</td>
<td>3</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>c</td>
<td>129</td>
<td>3</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>d</td>
<td>130</td>
<td>3</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>e</td>
<td>131</td>
<td>3</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>f</td>
<td>132</td>
<td>3</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>g</td>
<td>133</td>
<td>3</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>h</td>
<td>134</td>
<td>3</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>i</td>
<td>135</td>
<td>3</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>j</td>
<td>136</td>
<td>3</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>k</td>
<td>137</td>
<td>3</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>blank</td>
<td>138 - 144</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>l</td>
<td>145</td>
<td>3</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>m</td>
<td>146</td>
<td>3</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>n</td>
<td>147</td>
<td>3</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>o</td>
<td>148</td>
<td>3</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>p</td>
<td>149</td>
<td>3</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>q</td>
<td>150</td>
<td>3</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>EBCDIC Characters</td>
<td>Decimal Location</td>
<td>Column 1</td>
<td>Column 2</td>
<td>Column 3</td>
</tr>
<tr>
<td>-------------------</td>
<td>------------------</td>
<td>---------</td>
<td>---------</td>
<td>---------</td>
</tr>
<tr>
<td>p</td>
<td>151</td>
<td>3</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>q</td>
<td>152</td>
<td>3</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>r</td>
<td>153</td>
<td>3</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>blank</td>
<td>154 - 160</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>~</td>
<td>161</td>
<td>2</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>s</td>
<td>162</td>
<td>3</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>t</td>
<td>163</td>
<td>3</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>u</td>
<td>164</td>
<td>3</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>v</td>
<td>165</td>
<td>3</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>w</td>
<td>166</td>
<td>3</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>x</td>
<td>167</td>
<td>3</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>y</td>
<td>168</td>
<td>3</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>z</td>
<td>169</td>
<td>3</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>blank</td>
<td>170 - 191</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>192</td>
<td>2</td>
<td>7</td>
<td>11</td>
</tr>
<tr>
<td>A</td>
<td>193</td>
<td>3</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>B</td>
<td>194</td>
<td>3</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>C</td>
<td>195</td>
<td>3</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>D</td>
<td>196</td>
<td>3</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>E</td>
<td>197</td>
<td>3</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>F</td>
<td>198</td>
<td>3</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>G</td>
<td>199</td>
<td>3</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>H</td>
<td>200</td>
<td>3</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>I</td>
<td>201</td>
<td>3</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>blank</td>
<td>202 - 203</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>,</td>
<td>204</td>
<td>2</td>
<td>6</td>
<td>17</td>
</tr>
<tr>
<td>blank</td>
<td>205</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>v</td>
<td>206</td>
<td>2</td>
<td>6</td>
<td>18</td>
</tr>
<tr>
<td>blank</td>
<td>207</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>J</td>
<td>208</td>
<td>2</td>
<td>7</td>
<td>12</td>
</tr>
<tr>
<td>K</td>
<td>209</td>
<td>3</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>L</td>
<td>210</td>
<td>3</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>M</td>
<td>211</td>
<td>3</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>N</td>
<td>212</td>
<td>3</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>O</td>
<td>213</td>
<td>3</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>P</td>
<td>214</td>
<td>3</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Q</td>
<td>215</td>
<td>3</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>R</td>
<td>216</td>
<td>3</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>blank</td>
<td>217</td>
<td>3</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>/</td>
<td>218 - 223</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>blank</td>
<td>224</td>
<td>2</td>
<td>6</td>
<td>19</td>
</tr>
<tr>
<td>S</td>
<td>225</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>T</td>
<td>226</td>
<td>3</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>U</td>
<td>227</td>
<td>3</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>V</td>
<td>228</td>
<td>3</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>W</td>
<td>229</td>
<td>3</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>X</td>
<td>230</td>
<td>3</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Y</td>
<td>231</td>
<td>3</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Z</td>
<td>232</td>
<td>3</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>blank</td>
<td>233</td>
<td>3</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>blank</td>
<td>234 - 235</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>h</td>
<td>236</td>
<td>2</td>
<td>6</td>
<td>20</td>
</tr>
<tr>
<td>EBCDIC Characters</td>
<td>Decimal Location</td>
<td>Column 1</td>
<td>Column 2</td>
<td>Column 3</td>
</tr>
<tr>
<td>-------------------</td>
<td>------------------</td>
<td>----------</td>
<td>----------</td>
<td>----------</td>
</tr>
<tr>
<td>blank</td>
<td>237 - 239</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>240</td>
<td>3</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>241</td>
<td>3</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>242</td>
<td>3</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>243</td>
<td>3</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>244</td>
<td>3</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>245</td>
<td>3</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>246</td>
<td>3</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>247</td>
<td>3</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>248</td>
<td>3</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>9</td>
<td>249</td>
<td>3</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>blank</td>
<td>250</td>
<td>2</td>
<td>6</td>
<td>21</td>
</tr>
<tr>
<td></td>
<td>251 - 255</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

B.2.3 Keyword Table (10,7) Array

This table is initialized with the designated key words of the language to be interpreted. Words are placed in table columns based on the number of letters in each word. Only the first four letters of any word are placed in the table. Comparisons against the table are made by subtracting one from the total number of letters in the identifier to be scrutinized in order to select the proper table column. The first four letters of the identifier are then compared against all entries in the column selected. Index numbers are based on column number followed by row number of matched comparison.

Keyword Table Initialization Data

<table>
<thead>
<tr>
<th>Column 1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>Row</td>
<td>1</td>
<td>DO</td>
<td>END</td>
<td>CASE</td>
<td>BEGI</td>
<td>ACCE</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>IF</td>
<td>OUT</td>
<td>ELSE</td>
<td>FALS</td>
<td>EXPO</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>IN</td>
<td>0</td>
<td>EXIT</td>
<td>WHILE</td>
<td>GLOB</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>FI</td>
<td>0</td>
<td>GOTO</td>
<td>WRITE</td>
<td>RETU</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>0</td>
<td>0</td>
<td>PROC</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>0</td>
<td>0</td>
<td>READ</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>7</td>
<td>0</td>
<td>0</td>
<td>THEN</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>0</td>
<td>0</td>
<td>TRUE</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
Column | 1 | 2 | 3 | 4 | 5 | 6 | 7
-------|---|---|---|---|---|---|---
Row 9  | 0 | 0 | QUIT 0 | 0 | 0 | 0 | 0
Row 10 | 0 | 0 | CALL 0 | 0 | 0 | 0 | 0

B.2.4 Command Language Table (10,7) Array

This table is initialized with the designated command language words of the language to be interpreted. Words are placed in table columns based on the number of letters in each word. Only the first four letters of any word are placed in the table. Comparisons against the table are made by subtracting one from the total number of letters in the identifier to be scrutinized in order to select the proper table column. The first four letters of the identifier are then compared against all entries in the column selected. Index numbers are based on the column number followed by the row number of matched comparison.

Command Language Table Initialization Data

<table>
<thead>
<tr>
<th>Column</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>Row 1</td>
<td>ON</td>
<td>FNS</td>
<td>CHAR</td>
<td>CLEA</td>
<td>DIGI</td>
<td>BREA</td>
<td>0</td>
</tr>
<tr>
<td>Row 2</td>
<td>NO</td>
<td>LIB</td>
<td>COPY</td>
<td>ERAS</td>
<td>RESV</td>
<td>NOTR</td>
<td>0</td>
</tr>
<tr>
<td>Row 3</td>
<td>0</td>
<td>OFF</td>
<td>DROP</td>
<td>HENC</td>
<td>VALU</td>
<td>SUSP</td>
<td>0</td>
</tr>
<tr>
<td>Row 4</td>
<td>0</td>
<td>POP</td>
<td>EDIT</td>
<td>LINE</td>
<td>CLRS</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Row 5</td>
<td>0</td>
<td>RUN</td>
<td>HELP</td>
<td>PARS</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Row 6</td>
<td>0</td>
<td>VAR</td>
<td>LIST</td>
<td>STAC</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Row 7</td>
<td>0</td>
<td>0</td>
<td>LOAD</td>
<td>TRAC</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Row 8</td>
<td>0</td>
<td>0</td>
<td>SAVE</td>
<td>WIDT</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Row 9</td>
<td>0</td>
<td>0</td>
<td>VARS</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Row 10</td>
<td>0</td>
<td>0</td>
<td>WSID</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

B.2.5 Other Data Structures

All other local data structures used in the Scanner module are initialized to zero or are restricted to a single submodule.
Annex C  FORTRAN Code with Assertions

C.1 General

This annex contains the FORTRAN code of the Scanner Module which consists of eight subroutines as shown in figure 2-11. Assertions, as numbered in the High Level Design Language representation in section 3.5, are also included. Only the assertion number is used where no additional refinement is necessary and the FORTRAN Code follows the same pattern as the High Level Design Language. Where further refinement was necessary or there was deviation due to FORTRAN language restriction, the High Level Design Language assertion was subdivided and more detailed qualification of the assertion was stated. A variable translation table for conversion of those names which changed from the assertion refinement and High Level Design Language to the FORTRAN Code is at Table C1-1.
<table>
<thead>
<tr>
<th>High Level Design Language</th>
<th>FORTRAN</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Allocated_Space</td>
<td>SPACE</td>
<td>Amount of space allocated to a block.</td>
</tr>
<tr>
<td>Column</td>
<td>COL</td>
<td>Column number used to search Keyword and Command Language Word Tables.</td>
</tr>
<tr>
<td>ERROR</td>
<td>EXERR</td>
<td>Output parameter ERROR</td>
</tr>
<tr>
<td>Decimal.Counter</td>
<td>IQ</td>
<td>Used to indicate number of decimal places in a real number.</td>
</tr>
<tr>
<td>Last_Ln_DISP</td>
<td>LASTLN</td>
<td>Displacement to previous Token Line. Used in a line by line search of the Token Block in Subroutine LNFIN.</td>
</tr>
<tr>
<td></td>
<td>LOAD</td>
<td>Used to temporarily hold Token Data prior to placement in the Token Block storage word.</td>
</tr>
<tr>
<td>Ln_DISP</td>
<td>LNDISP</td>
<td>Line displacement to the current Token Line. Used in a line by line search of Token Block in Subroutine LNFIN.</td>
</tr>
<tr>
<td>Ln_Space_Used</td>
<td>LINUSE</td>
<td>Storage words necessary to store a line of Tokens.</td>
</tr>
<tr>
<td></td>
<td>NEXT</td>
<td>Counter for array RVALUE.</td>
</tr>
<tr>
<td>Next_Last_Char</td>
<td>STOP</td>
<td>Next to the last character in a number symbol.</td>
</tr>
<tr>
<td>Num_Code</td>
<td>NT</td>
<td>Number code used to indicate if a period has been encountered in a number symbol.</td>
</tr>
<tr>
<td>Num_First_Char</td>
<td>NFC</td>
<td>Number of the first character of a symbol in a line of text.</td>
</tr>
<tr>
<td>PINDEX</td>
<td>INDEXP</td>
<td>Input parameter PINDEX.</td>
</tr>
<tr>
<td>Pointer</td>
<td>PT</td>
<td>Pointer used with conditional GOTO statements.</td>
</tr>
</tbody>
</table>

Table C 1-1 Variable Translation Table
<table>
<thead>
<tr>
<th>High Level Design Language</th>
<th>FORTRAN</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Proc_Tab_Block</td>
<td>PROCBS</td>
<td>Absolute address of Procedure Table Block.</td>
</tr>
<tr>
<td></td>
<td>PTLABV</td>
<td>Points to line above in Token Block.</td>
</tr>
<tr>
<td></td>
<td>PARN</td>
<td>Closing parenthesis.</td>
</tr>
<tr>
<td></td>
<td>RVALUE</td>
<td>Array used to hold real number values.</td>
</tr>
<tr>
<td>Space_Used</td>
<td>TKUSED</td>
<td>Amount of allocated space used in the Token Block.</td>
</tr>
<tr>
<td>String_Illegal_Char</td>
<td>STRILC</td>
<td>Syntax error message for illegal character.</td>
</tr>
<tr>
<td>String_No_Close_Quote</td>
<td>STRNCQ</td>
<td>Syntax error message for no closing quote mark on a string.</td>
</tr>
<tr>
<td>String_Too_Many_Char</td>
<td>STRTMC</td>
<td>Syntax error message for too many characters in a string.</td>
</tr>
<tr>
<td>SYML</td>
<td>LENGTH</td>
<td>Used for SYML in Subroutine NUMPAC.</td>
</tr>
<tr>
<td>Tok_Add</td>
<td>TOKADD</td>
<td>Absolute address in the Token Block.</td>
</tr>
<tr>
<td>Tok_Block</td>
<td>TOKSAD</td>
<td>Absolute address of the Token Block.</td>
</tr>
<tr>
<td>Total_Char</td>
<td>TCHAR</td>
<td>Total characters in a text line.</td>
</tr>
<tr>
<td>Tok_Last_Ln_Num</td>
<td>TOKLLN</td>
<td>Line number of previous token line used in a line by line search of the token block.</td>
</tr>
<tr>
<td>Tok_Ln_Displ</td>
<td>TOKDLT</td>
<td>Token line displacement.</td>
</tr>
<tr>
<td>Tok_Log_Add</td>
<td>TOKLAD</td>
<td>Token logical address.</td>
</tr>
</tbody>
</table>

Table C1-1 (continued)
<table>
<thead>
<tr>
<th>High Level Design Language</th>
<th>FORTRAN</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tok_Total_Lines</td>
<td>TOKTLN</td>
<td>Total lines of Tokens in Token Block.</td>
</tr>
<tr>
<td>Txt_Add@</td>
<td>TXTADD</td>
<td>Absolute Address in Text Block.</td>
</tr>
<tr>
<td>Txt_Ln@</td>
<td>TXTLST</td>
<td>Absolute Address of a Text Line.</td>
</tr>
<tr>
<td>Txt_Block@</td>
<td>TXTSAD</td>
<td>Absolute address of Text Block.</td>
</tr>
<tr>
<td>Txt_In Disp</td>
<td>TXTDNL</td>
<td>Text line displacement.</td>
</tr>
<tr>
<td>Txt_In Num</td>
<td>TXTLIN</td>
<td>Text line number.</td>
</tr>
<tr>
<td>Next Txt_In Disp</td>
<td>TXTDNL</td>
<td>Next text line displacement.</td>
</tr>
<tr>
<td>Txt_Log_Add</td>
<td>TXTLAD</td>
<td>Text logical address.</td>
</tr>
<tr>
<td>Txt_Total_Lines</td>
<td>TOTLIN</td>
<td>Total text lines in Text Block.</td>
</tr>
<tr>
<td>TYPE</td>
<td>TOK</td>
<td>Token Code.</td>
</tr>
</tbody>
</table>

Table C 1-1 (Continued)
SUBROUTINE SCAN(INDEXF,EXERR)

COMMON /HEAP/AHEAP,HEAP(4CC)
COMMON /PRCSTO/PRACTXT,PRCTOK,PRCSYM,PRCCOD,
  PRCSTA,PRCLEA
COMMON /HPOFF/HPTAG,HPSPAC,HPSIZE,HPLAD,HPCBJ
COMMON /TOKOFF/TOKTOP,TOKPUT,TCKNLTA,TCKLAC,TCKFLK,TCKNUM
COMMON /TXTOFF/TXTTOP,TXTPUT,TXTNLN,TXTLNO,TXTFLK,TXTFLK,
  *TXTTKC,TXTNIST,TXTNLEN,TXTLIA,TXTSTA
COMMON/SCANNER/CTAB(255,3),KHTAB(10,7),CTAB(10,7),LINST(64),
  *SYMST(15,3),ERRST(64,3),SYM(2),TXTLAD,TOKLAD,PINDEX,ERROR,
  *CLFLAG,PRCBAS,ERRGT,VALUE(15),NEXT
SYMCT = C
ERRCT = C
ERROR = 0
RE.Scan = .FALSE.
CLFLAG = .FALSE.
TOK = "ITOK"

NEXT = 1
PTLABV = 3

2.1 NEXT is a counter for array RVALUE used to hold values of real numbers. PTLABV is a back link for Token Line displacement.

PINDEX = INDEXF
EXERR = 0

2.2 Initializes PINDEX and EXERR. Substitute input parameters were used so that PINDEX and ERROR could be passed in the Common Statement.

PRCBAS = HEAP(5)

3.3

TXTLAD = HEAP(PINDEX+PRCBAS+PRCTXTI)

4.3

TOKLAD = HEAP(PINDEX+PRCBAS+PRCTOK)

5.3

TXTSAD = HEAP(TXTLAD)
TOTLIN = HEAP(TXTSAD+TXTLN)

6.3

IF(HEAP(TOKLAD) > GT) GT IC IC

7.3
SIZE = TOTLIA + 1343
CALL GET(SIZE, TCK, TCKLAC, ERRCR)

\{8.\}
IF (ERRCR .EQ. 0) GOTO 3
EXERR = ERROR
\{9.\}
RETURN:
\{10.\}
HEAP(PINDEX + PRCPS + PRCCK) = TCKLAC

\{11.\}
GC TC 2
\{12.\}
RESCAN = .TRUE.

\{13.\}
TXTSAD = HEAP(TXTLAD)
TOKSAL = HEAP(TOKLAD)

\{14.\}
TXTCDL = HEAP(TXTSAD + TXTTPC)
TXTCTL = HEAP(TXTSAD + TXTCTL + TXTPLK)

\{15.\}
TOKCTL = HEAP(TOKSAD + TOKTPC)

\{16.\}
IF (TOKCTL .GE. 7) GOTO 5

\{17.\}
TOKDTL = 7
HEAP(TCKSAD + TCKTCP) = TOKDTL
HEAP(TOKSAD + HPSIZ) = TOKDTL

\{17.1 see para. 4.3.2.4\}
TXTADD = TXTSAD + HEAP(TXTSAD + TXTTCP)
TOKADD = TOKSAD + HEAP(TOKSAD + TCKTCP)

\{17.2 see para. 4.3.1.4 & 4.3.2.4\}
DO 1C I = 1, TC1LN
   IF (HEAP (TOKADD + TOKLN) .GE. HEAP (TXTADD + TXTLN)) OR .
      .NOT. RESCAN) GOTO 11

   DO 2C K = 1, TC1LN
      IF (HEAP (TOKADD + TOKLN) .GE. HEAP (TXTADD + TXTLN)) GCT
      TCKACC = TCKSAC + HEAP (TCKSAD + PTLABV + TCKFLK)
      HEAP (TCKSAD + PTLABV + TCKFLK) = HEAP (TCKACC + TCKFLK)
      HEAP (TOKSAD + TOKNLN) = HEAP (TCKSAD + TCKNLN) + 1
   2C CONTINUE

   RESCAN = .FALSE.

   11 IF (HEAP (TXTADD + TXTTCK) .GE. 1) GOTO 12

   CALL LSCAN (ERRFLG, LINUSE, TXTCLN, SYMCT, 6900)

   TKLSED = HEAP (TCKSAD + HPSIZE)

   SPACE = HEAP (TCKSAD + HPSPACE)

   25.1 Space = Allocated Space is para. 4.3.2.4
   IF (TKUSEC + LINUSE .LE. SPACE) GOTO 14

   26.3
   SIZE = (TOL1LN - 1) * 45
   CALL EXPANSIZE (SIZE, TCKLAC, ERRCR)

   27.3
   IF (ERRCR .EQ. 0) GOTO 14

   28.3 EXERR = ERROR
RETURN
\{29.2\}

14
TOKSAD = HEAP(TOKLAD)
TXTSAC = HEAP(TXILAD)
\{30.3\}
TOKDTL = HEAP(TOKSAD+HPSIZ)
\{30.1 Displacement is to the first open storage word in Token Block, see para. 4.3.2.4.\}
HEAP(TOKSAD+TOKDTL+TCKLAC) = HEAP(TXTSAC+TXTCTL+TXIL40)
\{31.3\}
IF(RESCAN) CALL LINFIN(TXTDTL,RESCAN,TOKDTL,& 15)
\{32.2\}
HEAP(TOKSAC+TOKDTL+TOKFLK) = HEAP(TOKSAD+HPSIZ) + LINUSE
\{33.3\}

15
TOKLAC = TOKSAD+TOKDTL+10KNUM
\{33.1 see para. 4.3.2.5.\}
IF(.NOT. ERRFLG) GO TO 16
\{34.3\}
HEAP(TOKADD) = 0
GO TO 17
\{35.3\}

16
HEAP(TCKACC) = SYMCT
DO 3C M = 1,SYMCT
LD 40 N = 1,3
LOC = SYMST(M,N)
TCKADD = TCKADD + 1
IF(SYMST(N,1).EQ.5.AND. N.EQ.2) CALL REAL(HEAP(TOKADD),RVALUE(LOAD),& 4C)
HEAP(TOKACC) = LOC
3C CONTINUE
\{36.3\}
3C CONTINUE
\{37.3\}
HEAP(TOKSAD+HPSIZE) = TKNUSED+LINUSE
HEAP(TOKSAD+TKBOFOT) = TCKDTL
HEAP(TOKSAD+TKNLIN) = HEAP(TOKSAD+TCKNLN) + 1

$38.3$

SWITCH = C
DELAY = 150
CALL STAX(SWITCH, DELAY)

$39.3$

IF(SWITCH .EQ. C) GO TO 12

$40.3$

EXERR = 9CC

$41.3$

RETURN

$42.3$

IF(HEAP(TOKSAD+TCKDTL+TCKLAC) .NE. HEAP(TXTSAD+TYDTDL+TXTNLN)) GOTO 9CC

$43.3$

TXTACE = TXTSAC+TXTCNL

$43.1$ see para. 4.3.1.4

TXTCTL = TXTCNL
TXTCNL = HEAP(TXTSAC+TXTCTL+TXTFLK)

$43.2$ see para. 4.3.1.4

PTLAEV = TCKDTL

$43.3$ Used as back link for last Token Line.

TCKACE = HEAP(TOKSAD+TCKCTL+TCKFLK)

$43.4$ see para. 4.3.2.4
10 CONTINUE
   {45.3}
   IF (.NOT. ERRLG) RETURN  {49.3}
   {46.3}
   CALL SERRCR
   {47.3}
   5CC  EXERR = 1
   {48.3}
   RETURN
   {50.3}
END
SUBROUTINE LSCAN(ERRLG, LINUSE, TXTDL, SYMCT, *)

{1.3}

IMPLICIT INTEGER(A-Z)
REAL RVVALUE
LOGICAL CLFLAG, ERRLG, RSCANN, FLAG
COMMON/SCANNER/CTAB(255, 1), KHTAB(LC, 7), CTAB(10, 7), LINST(64),
  *SYMST(15, 3), ERST(16, 3), SYM(2), TXTLAD, TOKLAD, PINDW, ERRL,
  *CLFLAG, PRCBAS, ERRCT, RVVALUE(15), NEXT
COMMON /HEAP/NHEAP, HEAP(4CC)
COMMON/PSCFF/PSCAG, HPSPAC, HPSIZ, HPLAD, HPOBJ
COMMON/PRSCFF/PRCLNLN, PRACMP, PRTXT, PRCTOK, PRCSYM, PRCCOD,
  1 PRSTA, PRCLEN
COMMON/TOKOFF/TOKTOP, TOKBOF, TOKLNC, TCKFLK, TCKNLN
COMMON/TXTOFF/TXTTOP, TXXXXT, TXTLND, TXTFLK, TXTLNLN,
  *TXTTOK, TXTST, TXTNLLEN, TXTLNLN, TXTSTA
DATA PARN /*")*/
CHARNU=0
ERCODE = 0
SYMCT = C
CLFLAG = .FALSE.

{2.3}

TXTSAE = HEAP(TXTLNLN)
TXTST= TXTSAE + TXTTOK
TCHAR=HEAP(TXTLST+TXTMLN)

{3.3}

DC 10 I = 1, TCHAR

{4.3}

CALL GETCHR (HEAP(TXTLST+TXTLNLN), I, ARG3)

{5.3}

LINST(I)= ARG3

{6.3}

10 CONTINUE

{7.3}

IF(LINST(I) .EQ. PARN) CLFLAG = .TRUE.

{8.3}

{9.3}
DC 2C J = 1, ICHAR

{10.3}

IF (J .LE. CHARNU) GC TC 20 {11.3}

{12.3}

CHAR = LINST (J)
PT = GTAB (CHAR, 1)

{13.3}

GC TC (2C, 3C, 4C), PT

{14.3} PT = 1, CHAR is a blank, GO TO 20

2C
SYMCT = SYMCT + 1
SYMST (SYMCT, 1) = GTAB (CHAR, 2)
SYMST (SYMCT, 2) = GTAB (CHAR, 3)
SYMST (SYMCT, 3) = J

{15.3}

GC TC 20

{16.3}

40
NFC = J {17.3}

{18.3}

CHARNU = J

{19.3}

CALL FURM (CHARNU, SYML, ICHAR, ERCCDE, INDEX, CLASS, SYM CODE, *
AFC, £ 55)

{20.3}

GOTO (70, 80, 90), SYM CODE

{23.3}
CALL TABLE(SYM, CLASS, INDEX, &90)

CLASS=2

CALL SYMTAB(PINDEX, SYM, SYM, INDEX, ERRCR)

IF(ERRCR .EQ. 1) RETURN 1

SYMCT = SYMCT+1
SYMST(SYMCT,1) = CLASS
SYMST(SYMCT,2) = INDEX
SYMST(SYMCT,3) = NFC

CONTINUE

GO TO 100

ERRCT= ERRCT+1  & RETURN FROM SUBROUTINE FORM.

TXTSAE = HEAP(TXTLAC)
ERRST(ERRCT,1)=HEAP(TXTSAE+TXTDTL+TXTLC)
ERRST(ERRCT,2)= NFC
ERRST(ERRCT,3)= ERCODE
EERRFGL=$TRUE.
SYMCT = C

LINUSE = (SYMCT+1)*3
RETURN

END
SUBROUTINE FORM (CHARNU, SYML, TCHR, ERCODE, INDEX, CLASS,
CODE, NFC, *)

{1.}

IMPLICIT INTEGER(A-Z)
REAL RVALSE
LOGICAL CLFLAG, ERRFLG, RESCAN, FLAG
COMMON/SCANP/CTAE(255, 3), KWTAB(10, 7), CLTAB(10, 7), LINST(64),
*SYMST(15, 3), ERRST(64, 3), SYM(2), TXTLAD, TOLKAD, PINDEX, ERROR,
*CLFLAG, PRCBAS, ERRCT, RVALSE(15), NEXT
COMMON /FEAP/NFEAP, HEAP(4CC)
CODE=C
CLASS = C
SYML=1
INDEX=0
VALUE=0
K=C
SYM(1)=C
SYM(2)=0
CHAR=LINST(CHARNU)
J=CHARNU+1

{2.}

DATA PERIOD/"./, BLANK/ /"/

{2.1 Initialization of variables to indicate Period and Blank.}
IF (J.GT. TCHR) J=TCHR

{2.2 Text for last character is LINST for this line.}
PT= CTAB(CHAR, 2)

{3.}

GE TE (20, 30, 40, 50), PT

20 CODE = 1

{4.}

{5.}

DO 25 I=J, TCHR

{6.}
CHAR=LINST(I)
IF(CTAB(CHAR,1).LT.3 .OR. CTAB(CHAR,2).GT.2) GOTO 10

14.
SYM1=SYM1+1
CHARNC=CHARNC+1

CONTINUE
14.1 Last character in line was part of Symbol.
GO TO 1C

CLASS=3
CODE=3

EC 35 I=J, TCHAR

16.
CHAR=LINST(I)
IF(CTAB(CHAR,2).NE.2 .OR. CHAR.NE. PERIOD) GOTO 60

19.
IF(CHAR.EQ. PERIOD) CLASS=4

20.
SYM1=SYM1+1
CHARNC=CHARNC+1

19.1 Counters are increased.

CONTINUE
17.1 Last character in line was part of Symbol.

CALL NUMPAC(NFC,SYM1,VALUE,ERCODE,90)
INDEX=VALUE

18.
RETURN
CLASS=5

45 CONTINUE

55 CONTINUE
ERCCDF = 2

{41.}

50 RETURN 1
70 INDEX = SYML

{38.}
RETURN {39.}

10 IF (SYML .GT. E) GO TO 80 {8. & 27.}

{10.} {29.}

DC 1 L = NFG, CHARNU

{11.}
CHAR = LINST (L)
K = K + 1
CALL PUTCHAR (SYM(I), K, CHAR)

{12.} {30.}

1 CONTINUE

{13.}
RETURN

60 ERCCDE = 1 {9.}
RETURN 1

{42.}
END
SUBROUTINE TABLE(SYML, CLASS, INDEX, *)

{1.3

IMPLICIT INTEGER(A-Z)

REAL RVALUE

LOGICAL CLFLAG, ERRFLG, RESCAN, FLAG

COMMON/SCANNER/CTAB(255, 3), KTAB(1C, 7), CLTAB(1C, 7), LINST(64),
  *SYMN(15, 3), ERNST(64, 3), SYM(2), TXTLAD, TOKLAD, PINDEX, ERRUR,
  *CLFLAG, PRCHRAS, ERRCT, RVVALUE(15), NEXT

COMMON /HEAP/ JHEAP, HEAP(4CC)
INDEX = 0
CCL = SYML - 1

{2.3

IF(CCL .EQ. 0) RETURN

{3.3

{4.3

IF(CLFLAG) GO TO 10

{5.3

{11.3

DO 20 I = 1, 1C
  IF(KTAB(I, CCL).EQ.SYM(I)) GO TO 40

20 CONTINUE

{16.3

RETURN

{17.1 Symbol is an identifier.

1C DC 30 I = 1, 10

{5.3

IF(CLTAB(I, CCL).EQ.SYM(I)) GO TO 40

20 CONTINUE

{10.3
RETURN {17.1 Symbol is an identifier.}

INDEX = (1 + 10) + COL

CLASS = 1

RETURN 1

{17.2 Control returned to label 90 in LNSCAN.}

END
SUBROUTINE NUMPAC (NFC, LENGTH, VALUE, ERCODE, *)

1.3

IMPLICIT INTEGER(A-Z)
REAL RNUM, RVALUE
LOGICAL CLFLAG, ERRFLG, RESCAN, FLAG
COMMON/SCANNER/CTAB(255,3), KWTAB(10,7), CLTAB(10,7), LINST(64),
* SYMST(15,3), ERRST(64,3), SYM(2), TXTLAD, TOKLAD, PINDEX, ERRCR,
* CLFLAG, PCHRAS, ERRCT, RVALUE(15), NEXT
COMMON /HEAP/NHEAP, HEAP(400)
DIMENSION NUMBER(10)
NUM=0
VALUE=0
RNUM = 0.0
STOP= NFC+LENGTH-1
DATA NUMBER/240, 241, 242, 243, 244, 245, 246, 247, 248, 249/
NT= 0
ERCODE = 0

2.3
IQ= 0
DATA PERIOD/* */

2.13
IF (LENGTH .LE. 8) GOTO 5

3.3
ERCODE = 1
RETURN 1

4.3
5 DO 10 I = NFC, STOP

5.3
DO 20 J = 1,10
   IF (LINST(I) .EQ. NUMBER(J)) GO TO 1

9.3
CONTINUE

10.3
   IF (LINST(I) .NE. PERIOD OR. IQ .GT. 0) GOTO 40

11.3
NT = 1
IQ = 1
GOTO 10

1      IF (NT .EQ. 1) IQ = IQ+1  \(\approx 7.3\)

INUM = J-1
NUM = NUM*10+INUM  \(\approx 8.3\)

10 CONTINUE  \(\approx 13.3\)
IF (NT .EQ. 1) GOTO 30  \(\approx 14.3\)

VALUE = NUM  \(\approx 15.3\)
RETURN  \(\approx 16.3\)

30 RNUM = NUM  \(\approx 14.13\)
RNUM = RNUM*.1**IQ  \(\approx 14.23\)
RVALUE(NEXT) = RNUM  \(\approx 14.33\)
VALUE = NEXT  \(\approx 14.43\)
NEXT = NEXT+1  \(\approx 14.53\)
RETURN  \(\approx 16.3\)

40 ERCODE = 3  \(\approx 12.3\)
RETURN 1

END
SUBROUTINE ERROR

\{1.\}

IMPLICIT INTEGER(A-Z)
REAL RVALUE
LOGICAL CLFLAG, ERRFLG, RESCAN, FLAG
COMMON /HEAP/HEAP, HEAP(400)
COMMON/SCANNER/CTAB(255,3), KXTAB(10,7), CLTAB(10,7), LINST(64),
  *SYMST(15,3), ERRST(64,3), SYM(2), TXTLD, TOKLD, PINDEX, ERRCR,
  *CLFLAG, PRCBAS, ERRCT, RVALUE(15), NEXT

DIMENSION STRMC(17), STRNGC(26), STRILC(17)
DATA STRMC/"S","Y","N","T","L","O","M","A","N","Y","I","A","H","C","U","O","T","E","G","I","N","G","S","T","R","L","N","G","/"
  *STRILC/"T","L","E","C","A","L","C","H","A","P","A","C","T"
  *"E","R","/"

\{2.\}

DO 10 I = 1, ERRCT

\{3.\}

LINE = ERRST(I,1)
CHAR = ERRST(I,2)
PT = ERRST(I,3)

\{4.\}

GO TO(20,30,40,C), PT

20 N = 21

\{5.\}

\{6.\}

CALL ERRPRINT (PINDEX, LINE, CHAR, N, STRMC, RETCOD)

\{7.\}

GO TO 99

30 N = 26

\{8.\}

\{9.\}
CALL ERRPRINT (PINDEX, LINE, CHAR, N, STRNCO, RETCOD)

10.

GO TO 99

40 N = 17  11.

12.

CALL ERRPRINT (PINDEX, LINE, CHAR, N, STRILC, RETCOD)

13.

99 IF (RETCOD .GT. 0) RETURN

14.

10 CONTINUE

15.

RETURN

16.

END
SUBROUTINE LINF(XTDL, RESCAN, TKCTL, *)

{1.3}

IMPLICIT INTEGER(A-Z)
REAL RVALUE
LOGICAL CLFLAG, ERRFLG, RESCAN, FLAG
COMMON /HEAP/AHEAP, HEAP(400)
COMMON/SCANNER/CTAB(255, 3), KHTAB(10, 7), CLTAB(10, 7), LINST(64),
*SYMST(15, 3), ERRST(64, 3), SYM(2), TXTLD, TUKLD, PINDEX, ERRCR,
*CLFLAG, PRCPAS, ERRCT, RVALUE(15), NEXT
COMMON /HPDFH/HPTAG, HPSPC, HPSIZ, EPLAD, HPBCJ
COMMON/TKOFF/TKTOP, TKISLE, TCKLN, TCKLNR, TCKLFK, TCKNUM
COMMON/TXTOFF/TXTTOP, TXTDET, TXTLNR, TXTLNO, TXTFLK, TXTFLK,
*TXTCK, TXTNST, TXTLLEN, TXTLIN, TXTSTA
COMMON/PRCOFF/PRCLEN, PRCPAS, PRCTXT, PRCTCK, PRCSYN, PRCSSD,
PRCSTA, PRCLEN

TOKLN = 0
LASTLN = 3

{2.3}

TXTSAD=HEAP(TXTLD)

{3.3}

TOKSAD=HEAP(TKLD)

{4.3}

TOKTLN=HEAP(TOKSAD+TOKLN)

{5.3}

LNDISP=HEAP(TCKSAD+TOKTOP)

{6.3}

TXTLIN=HEAP(TXTSAD+TXTL+TXTLNO)

{7.3}

DO 10 I=1, TOKLN

IF(TXTLIN .GT. TCKLN .AND. TXTLIN .LE. HEAP(TOKSAD+LNDISP+TOKLNO)) GO TO 20

{8.3}

{14.3}
TOKLN = HEAP(TOKSAD+LADISP+TCNLNC)
LASTLN = LADISP
LADISP = HEAP(TOKSAD+LASTLN+TCFLK)

\[ \text{15.3} \]

1C CONTINUE

\[ \text{16.3} \]

RESCAN = .FALSE.

RETURN

\[ \text{17.3} \]

2C IF(TXTLN.EQ. HEAP(TOKSAD+LADISP+TCNLNC)) GOTO 30

\[ \text{12.3} \]

HEAP(TCKSA+TCGLTL+TCFLK) = HEAP(TOKSAD+LASTLN+TCFLK)
HEAP(TOKSAD+TCALT+TCFLK) = HEAP(TOKSAD+LASTLN+TCFLK)
HEAP(TOKSAD+LASTLN+TCFLK) = TCCTL
HEAP(TOKSAD+TKNLN) = HEAP(TOKSAD+TKNLN) + 1

\[ \text{13.3} \]

RETURN 1

\[ \text{17.3} \]

3C HEAP(TOKSAD+TCGLTL+TCFLK) = HEAP(TCKSA+LADISP+TCFLK)

\[ \text{10.3} \]

HEAP(TOKSAD+LASTLN+TCFLK) = TCCTL

\[ \text{11.3} \]

RETURN 1

\[ \text{17.3} \]

END
SUBROUTINE REAL (X, Y, *)

REAL X, Y
X = Y
RETURN 1
END
INFORMAL VERIFICATION OF CORRECTNESS OF THE
SCANNER MODULE OF AN INTERPRETER PROGRAM

by

JAMES NOEL JONES

B. S., Oklahoma State University, 1965

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
1975
Abstract

This report provides a documented informal verification of the correctness of the Scanner module of an interpreter program designed and implemented by the students of CS 286-700, summer 1975 session. The purpose of the report was to show that a large multi-program can be verified by the refinement of assertions developed from the program specification.

Verification of the Scanner module was accomplished by parallel refinement of assertions with three levels of program development: (1) that of English specifications, (2) high level design language, and (3) FORTRAN code. Scope of the module proof included the Scanner routine, all subroutines developed directly from the Scanner module, and all external routines used by the Scanner. Verification of the correctness is based on the refined assertions and tracing the state transition of each routine of the Scanner module. A routine is considered verified if all termination points comply with the output assertions developed from the specifications for the routine.