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:

Major Professor

TABLE OF CONTENTS

LIST OF FIGURES

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

(10) Symbol Tables
(11) File Maintenance

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

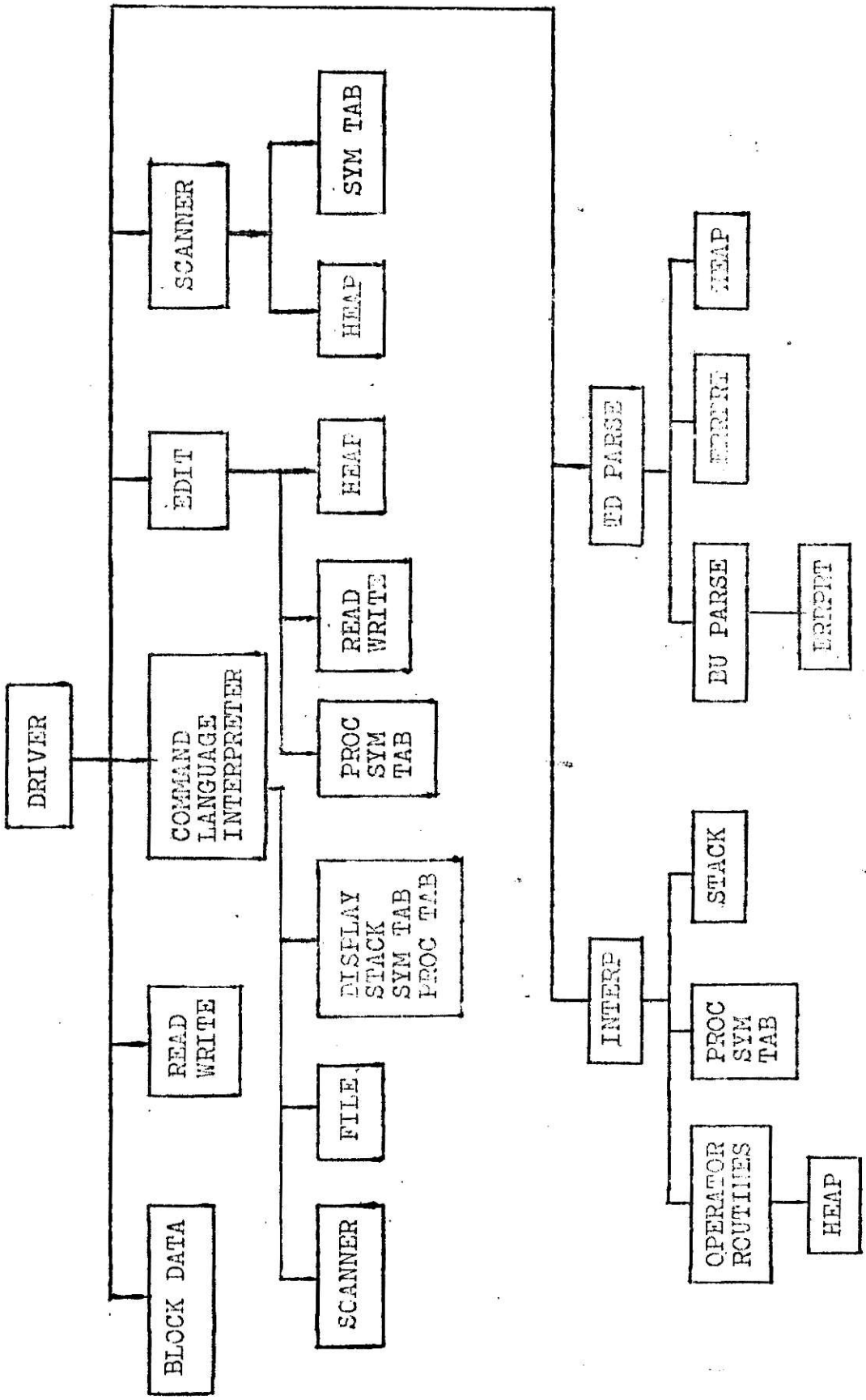


Figure 2-1 Interpreter Program Hierarchical Chart

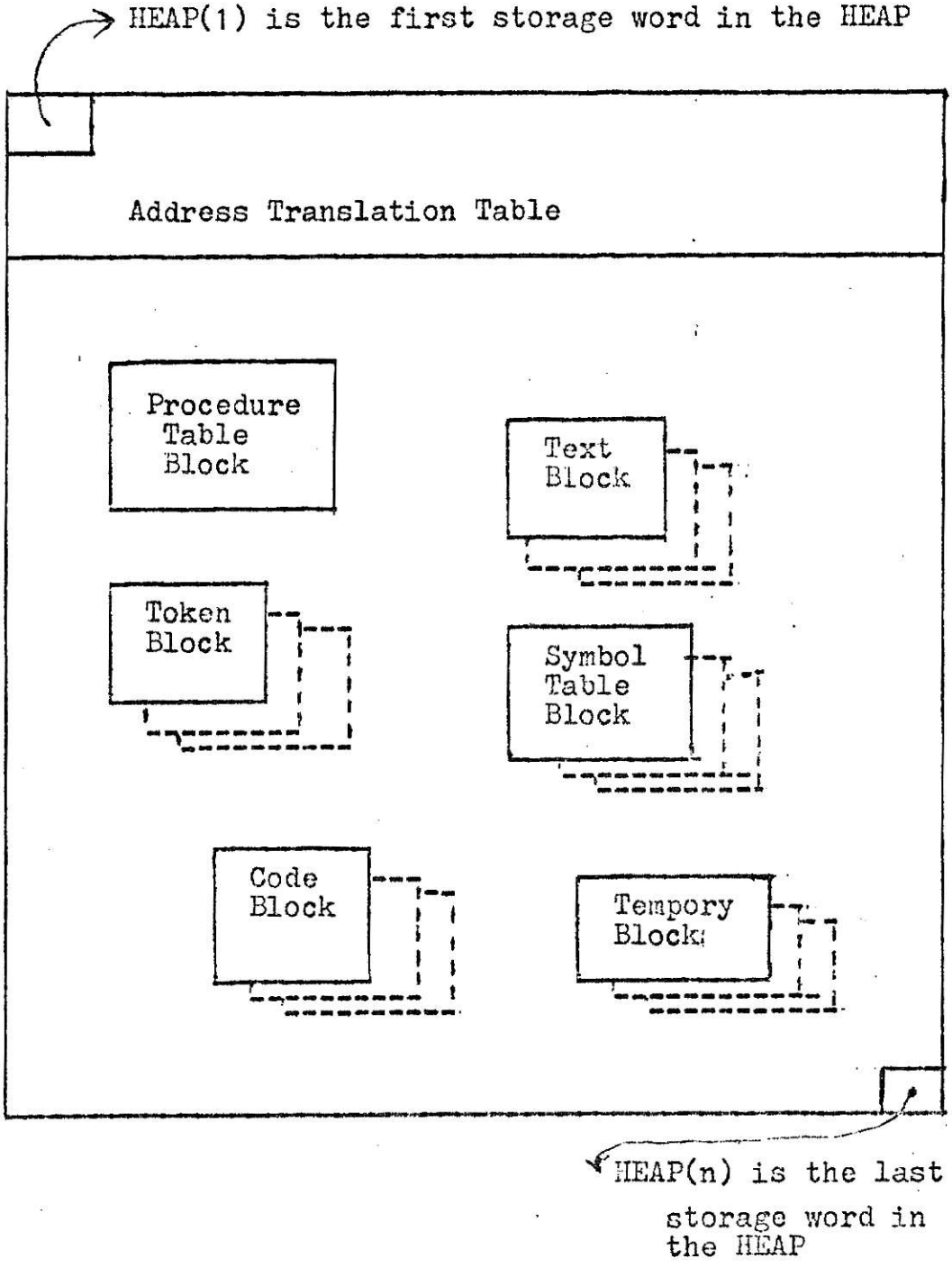


Figure 2-2 HEAP Storage Area Contents

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)

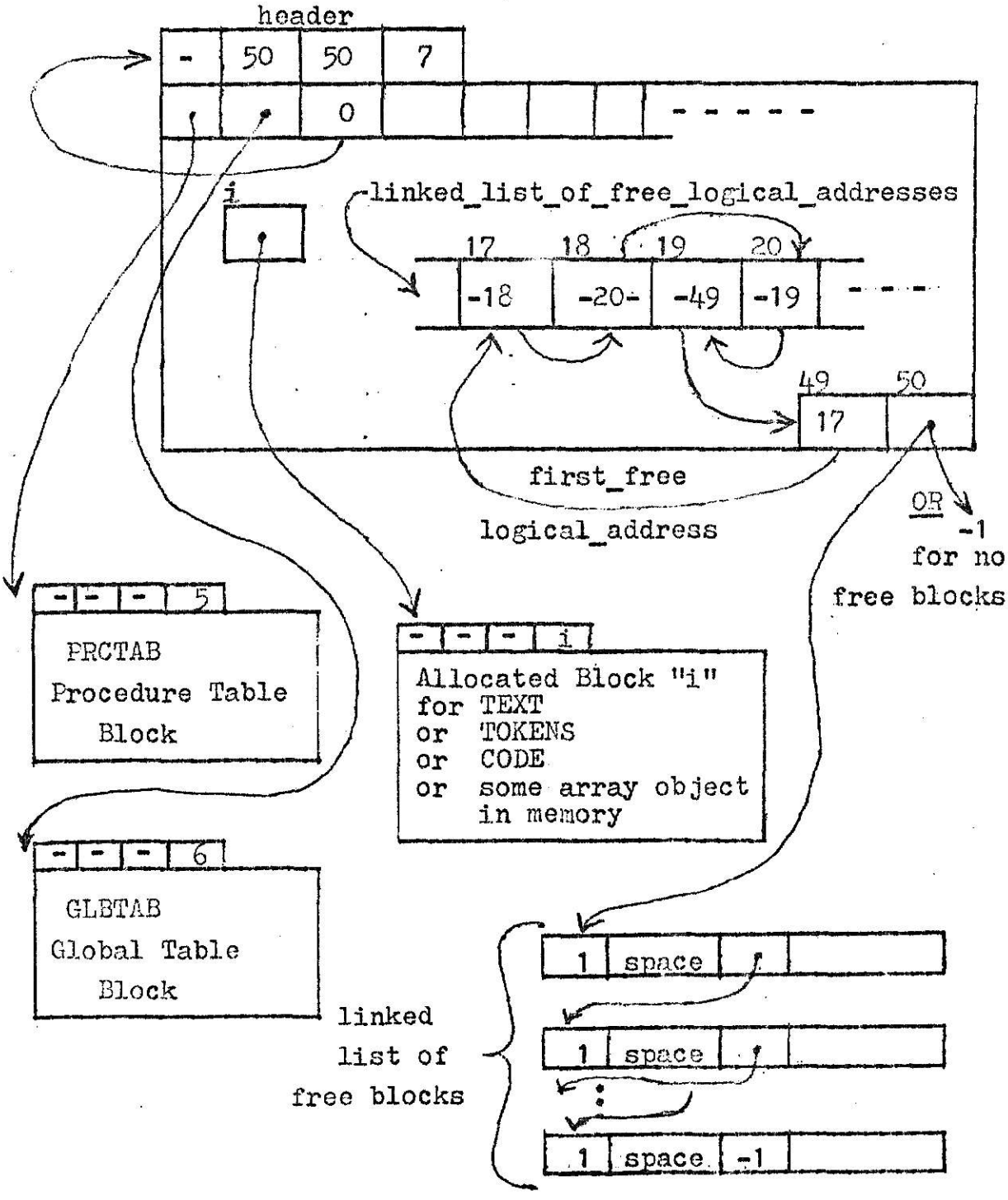


Figure 2-3 Format for Address Translation Table

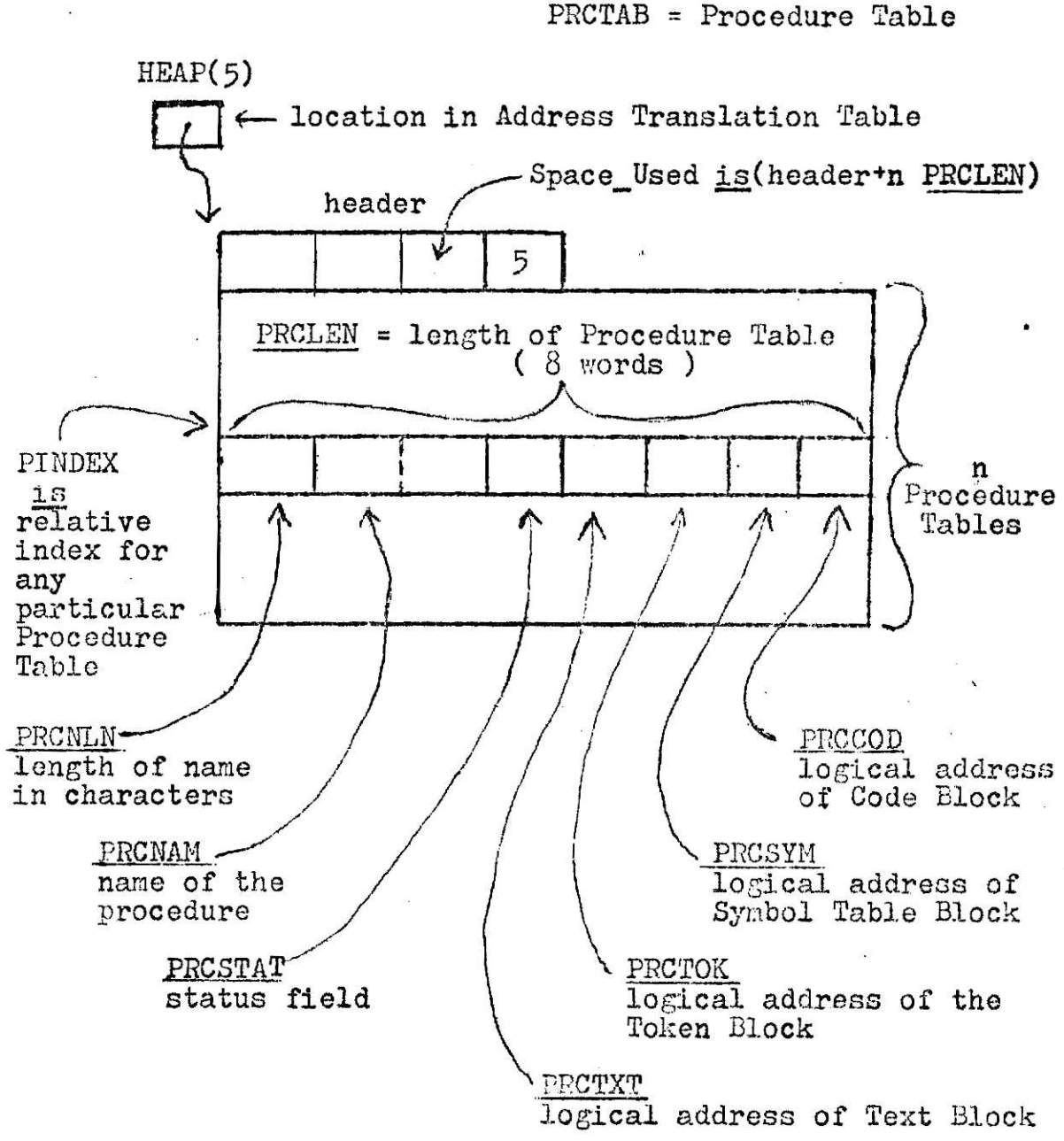
start of the HEAP to the location prior to the beginning of the indexed data block area. A Procedure Table data block, figure 2-4, is established and contains individual Procedure Tables for each source text that is translated by the interpreter program. The individual Procedure Tables are used to store 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

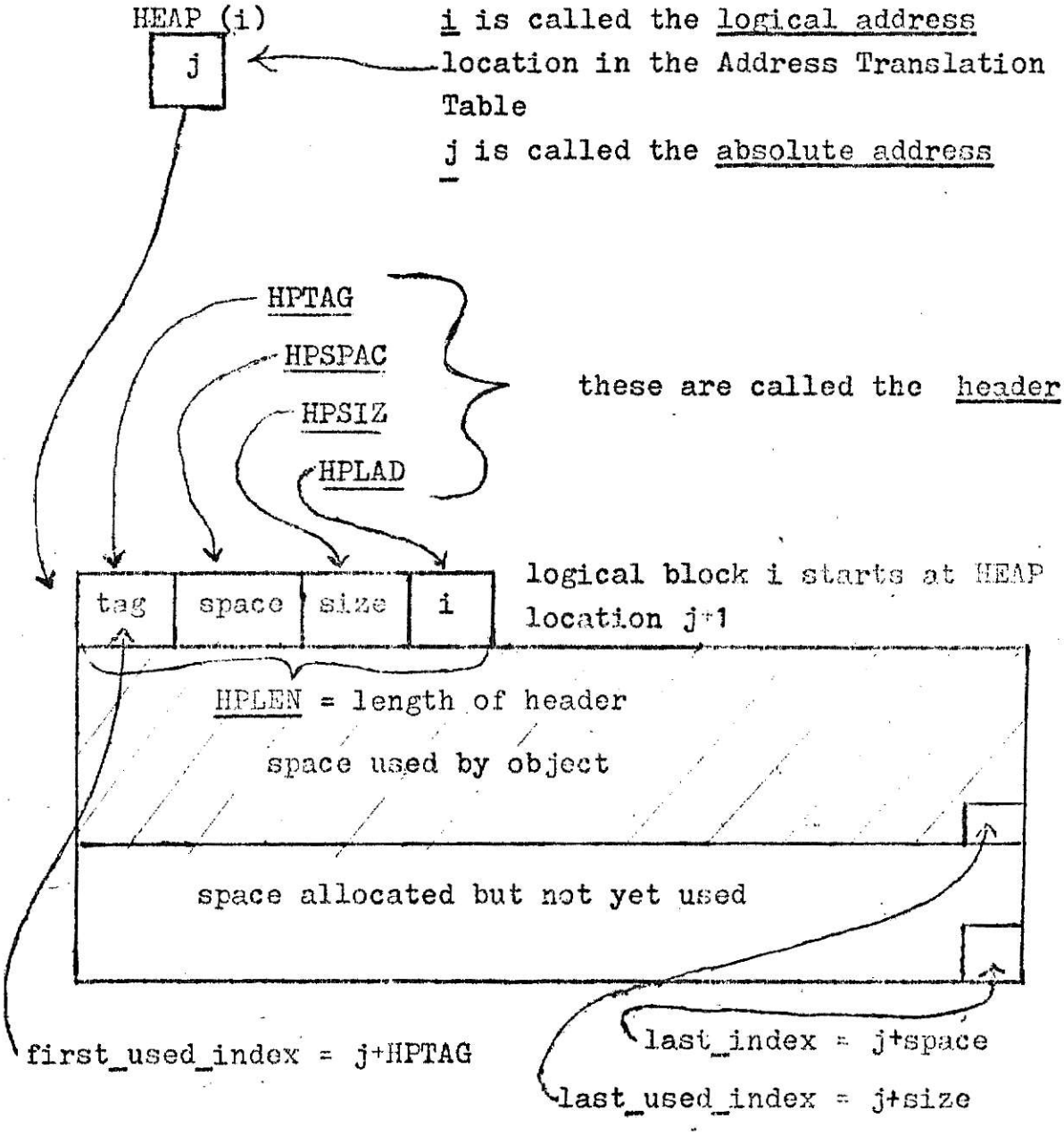
A number of assertions concerning particular data structures are common to all modules of the language interpreter program. These assertions have been designated as global and are necessary for development of specific input and output assertions for each module and submodule of the interpreter program.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



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



* 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

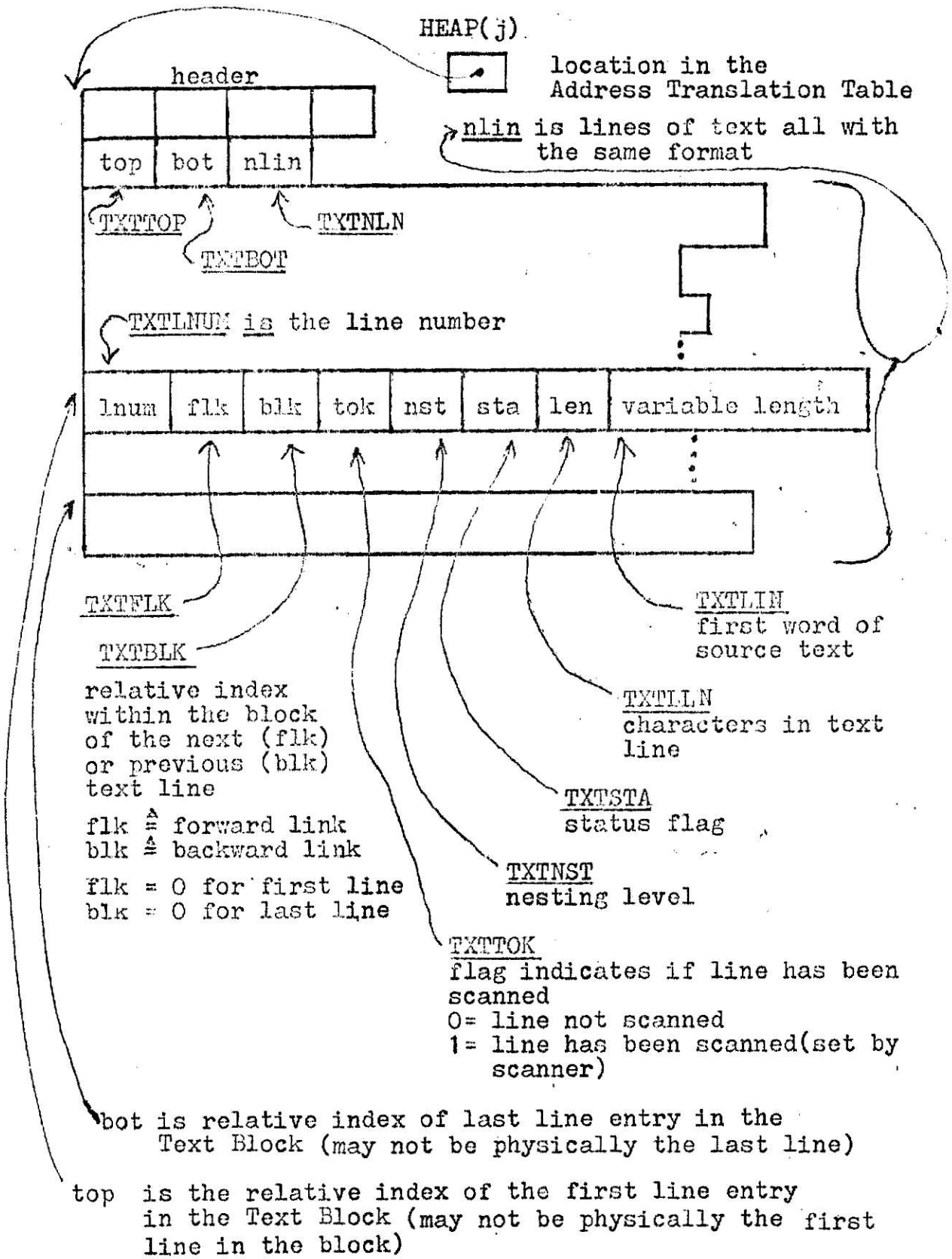
That the global assertions as stated in paragraph 2.1.2 are valid.

(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.



* 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

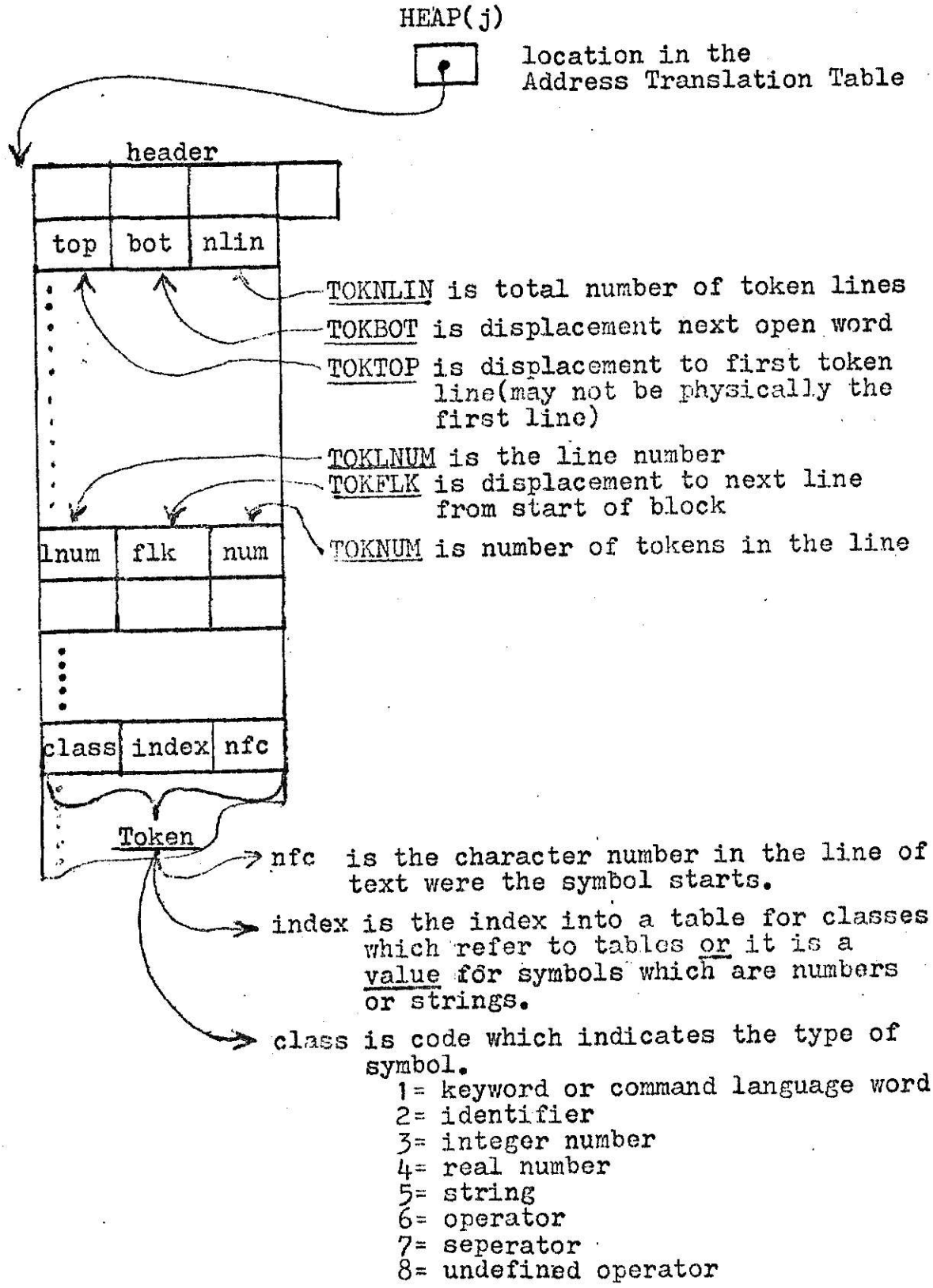
That a Token data block, figure 2-7, is established and contains token data as per the Scanner specifications, and that the logical address to the data block is assigned to the appropriate location of the Procedure Table referenced by the PINDEX.

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.

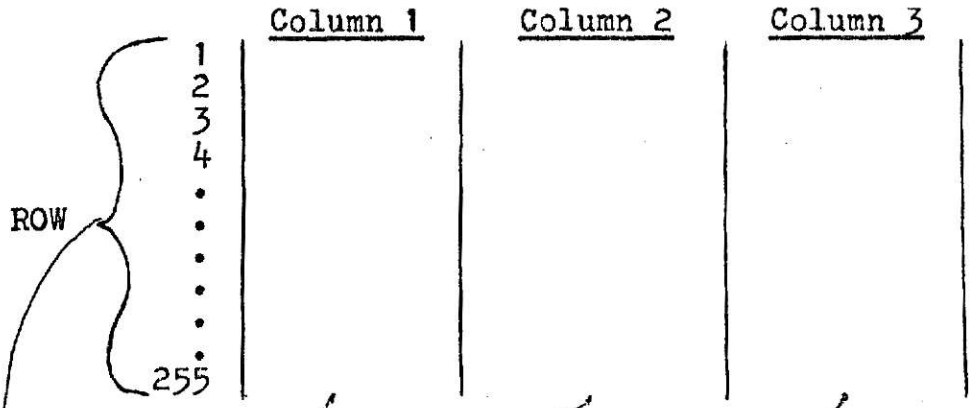
(3.1) Character Table (255,3) array

This table is initialized in accordance with figure 2-8 to correspond to the character set selected for use with the



* Offset constants are underlined.

Figure 2-7 Format for Token Block



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="

integer value is used as a condition code for forming specific symbols

6=operator
7=separator

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 initilization 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

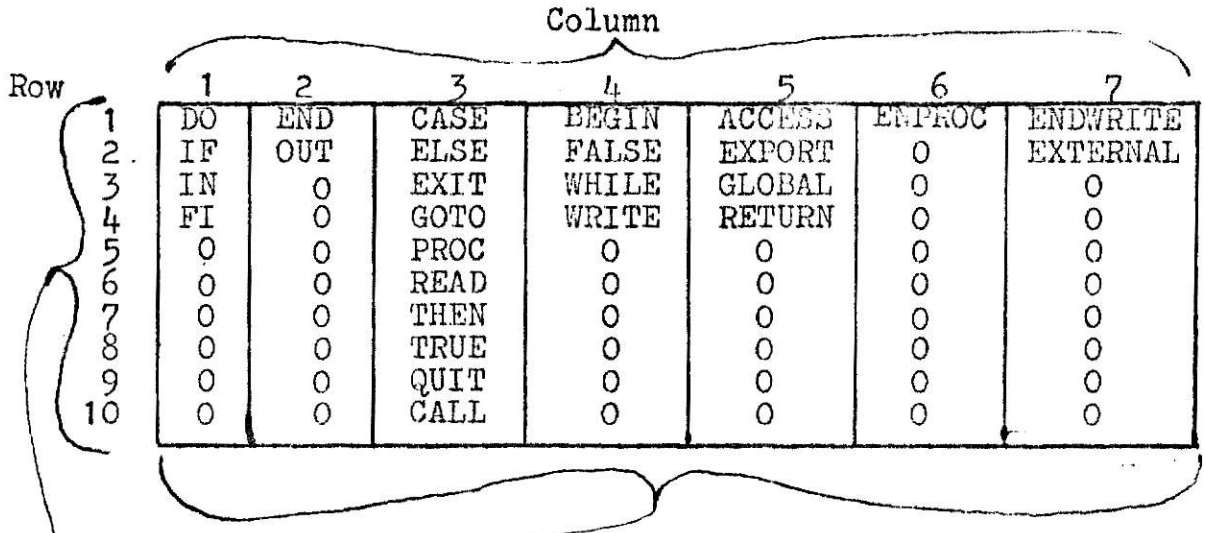
This array is initialized to zero and then used to hold the characters of a line of source text for each line of text while being processed by the Scanner module.

(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



Column- 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.

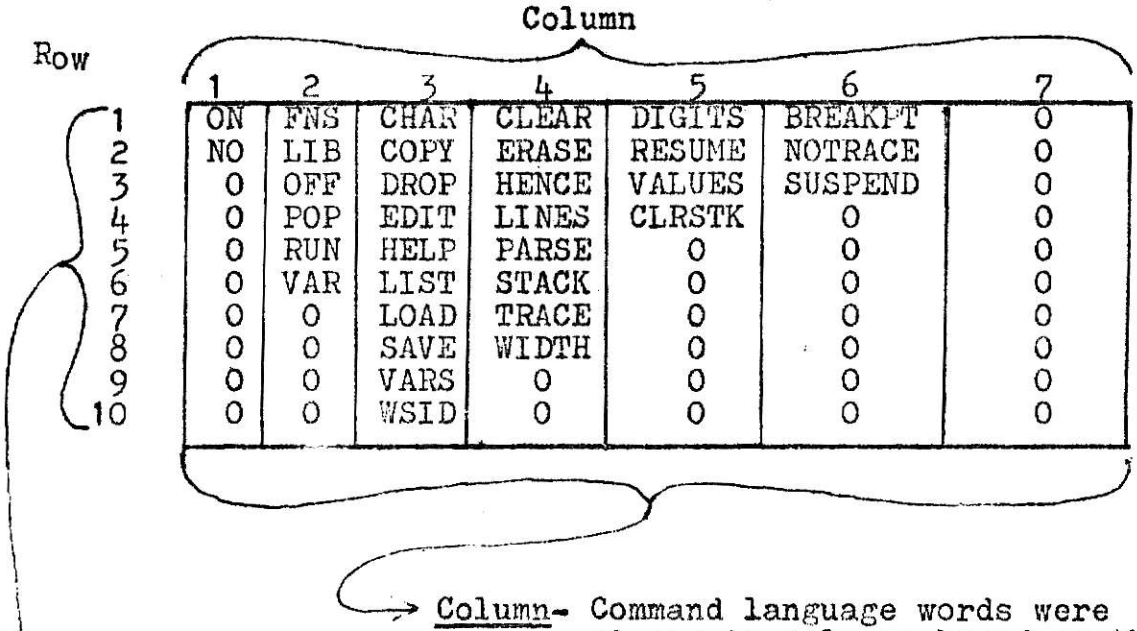
Row- The total number of rows of the table was dependent on the largest number of keywords with thw same number of characters.

Index- 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



Column- 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.

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

Index- 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

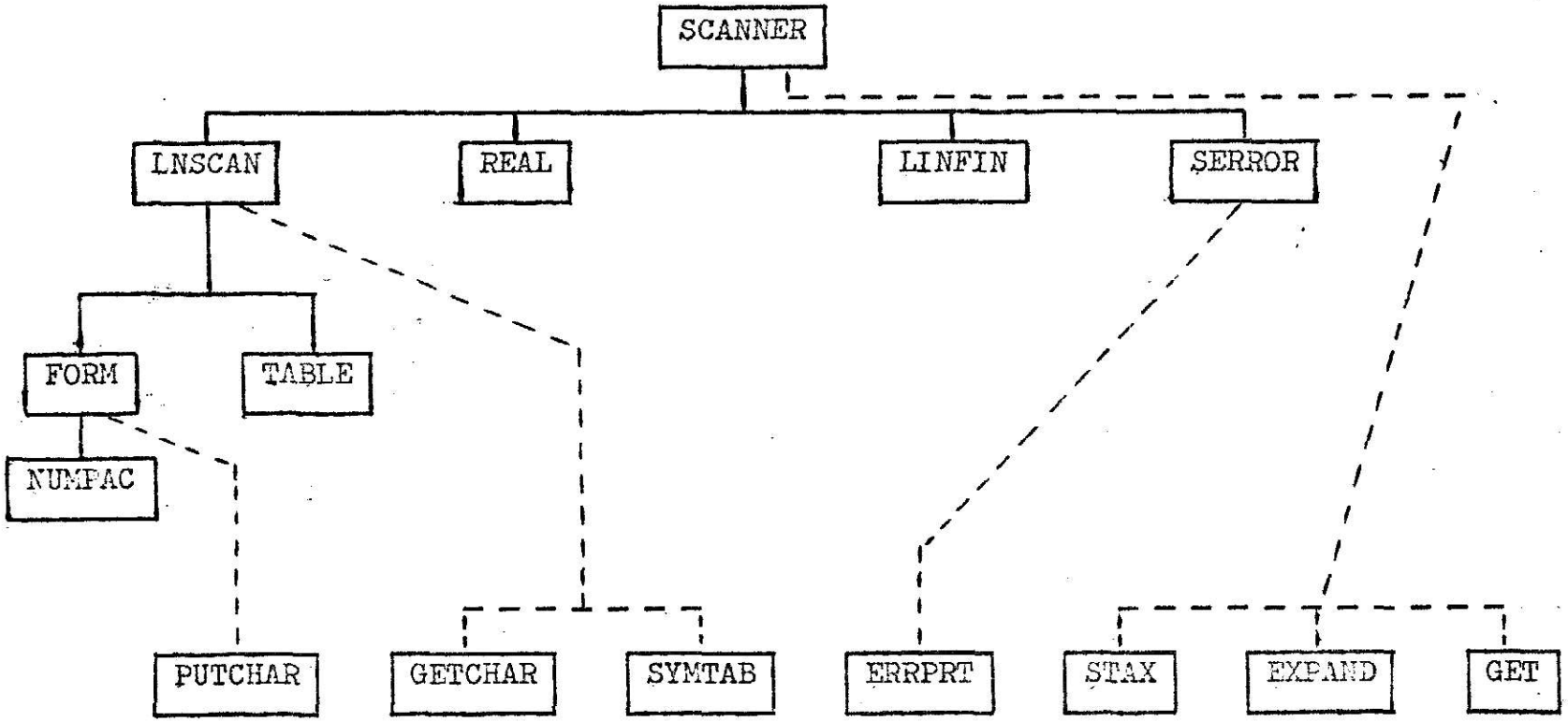
(4.1.2) Driver Module - to subroutines ERRPRT, STAX, GETCHR, and PUTCHR.

(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 preceeded by a $.

(6.6) Strings

All strings must be preceeded 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 confinds 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 specified in the call statement.

(2.7.2) Assertions

The assertions for this routine are based on the parameters 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 liniar 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 i:

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

$5 \leq I \leq$ 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_Spacd

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

HEAP(Proc_Tab_Block@ + HPSIZ) ≙ Space_Used

and

Space_Used is SPACE_USED means

$4 \leq$ Space_Used $\leq$ Allocated_Space

and

PINDEX is PROC_TAB_BLOCK . PINDEX means

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

and

$\ni$ a Procedure Table as shown in figure 2-4, starting at HEAP(Proc_Tab_Block@ + PINDEX + 1)

3.2.5 Procedure Table Address

Proc_Tab_Add@ is HEAP . PROC_TAB_ADD@ means

HEAP(Proc_Tab_Block@ + PINDEX) ≙ Proc_Tab_Add@

3.2.6 Procedure Table

Proc_Tab is PROC_TAB_BLOCK . PROC_TAB means

Proc_Tab is eight words as per figure 2-4

and

PROC_TAB . PRCNLEN ≙ HEAP(Proc_Tab_Add@ + 1)

PROC_TAB . PRCNAM ≙ HEAP(Proc_Tab_Add@ + 2)

PROC_TAB . PRCSTAT ≙ HEAP(Proc_Tab_Add@ + 4)

PROC_TAB . PRCTXT ≙ HEAP(Proc_Tab_Add@ + 5)

PROC_TAB . PRCTOK ≙ HEAP(Proc_Tab_Add@ + 6)

PROC_TAB . PRCSYM ≙ HEAP(Proc_Tab_Add@ + 7)

PROC_TAB . PRCCOD ≙ HEAP(Proc_Tab_Add@ + 8)

and

HEAP(Proc_Tab_Add@ + PRCNLEN) ≙ not used by Scanner

HEAP(Proc_Tab_Add@ + PRCNAM) ≙ not used by Scanner

HEAP(Proc_Tab_Add@ + PRCSTAT) ≙ not used by Scanner

HEAP(Proc_Tab_Add@ + PRCTXT) ≙ Txt_Log_Add or null txt

and

Txt_Log_Add is TXT_LOG_ADD means

∋ a Txt_Log_Add as per para. 3.3.1

and

Null txt means Source Text does not exist.

HEAP(Proc_Tab_Add@ + PRCTOK) ≙ Tok_Log_Add or null tok

and

Tok_Log_Add is TOK_LOG_ADD means

∋ a Tok_Log_Add as per para. 3.3.2.2

and

Null tok means Token data block does not exist.

HEAP(Proc_Tab_Add@ + PRCSYM) ≙ not used by Scanner

HEAP(Proc_Tab_Add@ + PRCCOD) ≙ 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@ + HPSPAC) ≙ Allocated_Space

HEAP(Txt_Add@ + HPSIZ) ≙ Space_Used

HEAP(Txt_Add@ + HPLAD) ≙ Txt_Log_Add

HEAP(Txt_Add@ + TXTTOP) ≙ 1st Txt_Ln_Disp

HEAP(Txt_Add@ + TXTBOT) ≙ Last Txt_Ln_Disp

and

Txt_LnDisp is TXT_LN_DISP means

$7 \leq$ Txt_Ln_Disp $<$ HEAP(Txt_Add@ + HPSIZ)

HEAP(Txt_Add@ + TXTNLIN) ≙ Txt_Total_Ln

and

Txt_Total_Ln is TXT_TOTAL_LN means

Txt_Total_Ln ≙ 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 HEAP(Txt_Add@ + Txt_Ln_Disp) is word prior to the start of figure 2-7.

and

TXT_LN . TXTLNUM ≙ HEAP(Txt_Add@ + Txt_Ln_Disp + 1)

TXT_LN . TXTFLK ≙ HEAP(Txt_Add@ + Txt_Ln_Disp + 2)

TXT_LN . TXTBLK ≙ HEAP(Txt_Add@ + Txt_Ln_Disp + 3)

TXT_LN . TXTTOK ≙ HEAP(Txt_Add@ + Txt_Ln_Disp + 4)

TXT_LN . TXTNST ≙ HEAP(Txt_Add@ + Txt_Ln_Disp + 5)

TXT_LN . TXTSTAT ≙ HEAP(Txt_Add@ + Txt_Ln_Disp + 6)

TXT_LN . TXTLLEN $\triangleq$ HEAP(Txt_Add@ + Txt_Ln_Disp + 7)

TXT_LN . TXTLIN $\triangleq$ HEAP(Txt_Add@ + Txt_Ln_Disp + 8)

and

HEAP(Txt_Add@ + Txt_Ln_Disp + TXTLNUM) $\triangleq$ Txt_Ln_Num

and

Txt_Ln_Num is TXT_LN_NUM means

$1 \leq$ Txt_Ln_Num $\leq$ HEAP(Txt_Add@ + TXTBOT + TXTLNUM)

HEAP(Txt_Add@ + Txt_Ln_Disp + TXTFIK) $\triangleq$ Next Txt_Ln_Disp

HEAP(Txt_Add@ + Txt_Ln_Disp + TXTBIK) $\triangleq$ Last Txt_Ln_Disp

HEAP(Txt_Add@ + Txt_Ln_Disp + TXTTOK) $\triangleq$ i

and

$i \in \{0,1\}$

i $\triangleq$ 0 means line not scanned

i $\triangleq$ 1 means line scanned

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

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

HEAP(Txt_Add@ + Txt_Ln_Disp + TXTLLEN) $\triangleq$ Total_Char

and

Total_Char is TOTAL_CHAR means

$64 \geq$ Total_Char = number of characters in source line

and

CHAR means

CHAR $\in \{1,\ldots,255\}$

and

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

3.3.2 Output Assertions

(2.1) ERROR

I is ERROR means

I $\in$ {0,1,555,900}

I $\triangleq$ 0 means no errors

I $\triangleq$ 1 means syntax errors in some Txt_Ln's

I $\triangleq$ 555 means insufficient Allocated_Space for Tokens of source text, incomplete scan

I $\triangleq$ 900 means user interrupt source text, incomplete scan

(2.2) Token Logical Address

$\exists$ i $\ni$ 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) $\triangleq$ Tok_Log_Add

and

HEAP(Tok_Log_Add) $\triangleq$ 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

$\ni$ 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 . HPTAG $\hat{=}$ HEAP(Tok_Add@ + 1)

TOK_BLOCK . HPSPAC $\hat{=}$ HEAP(Tok_Add@ + 2)

TOK_BLOCK . HPSIZ $\hat{=}$ HEAP(Tok_Add@ + 3)

TOK_BLOCK . HPLAD $\hat{=}$ HEAP(Tok_Add@ + 4)

TOK_BLOCK . TOKTOP $\hat{=}$ HEAP(Tok_Add@ + 5)

TOK_BLOCK . TOKBOT $\hat{=}$ HEAP(Tok_Add@ + 6)

TOK_BLOCK . TOKNLIN $\hat{=}$ HEAP(Tok_Add@ + 7)

and

HEAP(Tok_Add@ + HPSPAC) $\hat{=}$ Allocated_Space

HEAP(Tok_Add@ + HPSIZ) $\hat{=}$ Space_Used

HEAP(Tok_Add@ + HPLAD) $\hat{=}$ Tok_Log_Add

HEAP(Tok_Add@ + TOKTOP) $\hat{=}$ 1st Tok_Ln_Disp

and

Tok_Ln_Disp is TOK_LN_DISP means

$7 \leq$ Tok_Ln_Disp $<$ HEAP(Tok_Add@ + HPSIZ)

HEAP(Tok_Add@ + TOKBOT) $\hat{=}$ HEAP(Tok_Add@ + HPSIZ) + 1

HEAP(Tok_Add@ + TOKNLIN) $\hat{=}$ Tok_Total_Ln

and

Tok_Total_Ln is TOK_TOTAL_LN means

Tok_Total_Ln $\hat{=}$ total number of token lines

(2.5) Token Line

Tok_Ln is TOK_LN means

$\exists$ a token line and HEAP(Tok_Add@ + Tok_Ln_Disp)

is word prior to the start of a token line as per figure 2-8.

and

TOK_LN . TOKLNUM ≙ HEAP(Tok_Add@ + Tok_Ln_Disp + 1)

TOK_LN . TOKFLK ≙ HEAP(Tok_Add@ + Tok_Ln_Disp + 2)

TOK_LN . TOKNUM ≙ HEAP(Tok_Add@ + Tok_Ln_Disp + 3)

and

HEAP(Tok_Add@ + Tok_Ln_Disp + TOKLNUM) ≙ Tok_Ln_Num

and

Tok_Ln_Num is TOK_LN_NUM means

$1 \leq$ Tok_Ln_Num $\leq$ HEAP(Txt_Add@ + TXTBOT + TXTLNUM)

HEAP(Tok_Add@ + Tok_Ln_Disp + TOKFLK) ≙
Next Tok_Ln_Disp

HEAP(Tok_Add@ + Tok_Ln_Disp + TOKNUM) ≙ Total_Tokens

and

Total_Tokens is TOTAL_TOKENS means

Total_Tokens ≙ 0 or number of tokens triples in the token line as per figure 2-9.

Total_Tokens ≙ 0 means

Syntax error in Txt_Ln and no tokens were generated.

and

$$\text{Token} \triangleq \begin{bmatrix} \text{Class} \\ \text{Index or Value} \\ \text{Num\_First\_Char} \end{bmatrix}$$

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) KWTAB is KEYWORD TABLE means

KWTAB(i, j) array

i is $\in \{1,\ldots,10\}$

j is $\in \{1,\ldots,7\}$

and

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

(3.1.3) CLTAB is COMMAND LANGUAGE TABLE means

CLTAB(i, j) array

i is $\in \{1,\ldots,10\}$

j is $\in \{1,\ldots,7\}$

and

CLTAB(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) $\triangleq$ CHAR

i is $\in$ {1,....,(Txt_Ln(TXTLLN)}

(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,....,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) $\triangleq$ 1 thru SYMCT

and

SYMCT is SYMBOL COUNT means

SYMCT $\triangleq$ total number of symbols formed

and

SYMST(j) $\triangleq$ is the Token Data for the symbol

(3.1.7) ERRST is ERROR STACK means

ERRST(i,j) array

i is $\in \{1,\ldots,64\}$

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

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 $\triangleq$ total Scanned Txt_Ln with errors

$$\text{ERRST}(j) \triangleq \begin{cases} \text{Txt\_Ln\_Num} \\ \text{ERCODE} \\ \text{Num\_First\_Char} \end{cases}$$

(3.2) Counter Initialization

(3.2.1) SYMCT is SYMBOL_COUNT means for each Txt_Ln

$0 \leq \text{SYMCT} \leq 15$

and

for each Symbol formed SYMCT is incremented by 1

(3.2.2) ERRCT is ERROR_COUNT means for each Txt_Block

$0 \leq \text{ERRCT} \leq \text{Total\_Txt\_Ln}$

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 $\triangleq$ 3 means Syntax_Error,
CHAR $\triangleq$ in machine character translation table

or

CHAR is illegal in a specific class of symbol

(3.4.2) SIZE is Requested Allocated_Space for Tok_Block means

SIZE $\triangleq$ Txt_Total_Ln * 15 * 3

and

15 is maximum number of symbols per Txt_Ln

and

3 is number of storage words per token

(3.4.3) SYMCODE is SYMBOL CODE means

SYMCODE $\in$ {1,2,3}

and

SYMCODE $\triangleq$ 1 means

Symbol is
- identifier or
- keyword or
- command language word

and

SYMCODE $\triangleq$ 2 means

Symbol is
- identifier or
- undefined operator

and

SYMCODE $\triangleq$ 3 means

Symbol is
- keyword or
- command language word or
- number or
- string

(3.4.4) CLASS is SYMBOL_CLASS means

CLASS $\in$ {1,2,3,4,5,6,7,8}

and

CLASS $\hat{=}$ 1 means Symbol is keyword or command language word

CLASS $\hat{=}$ 2 means Symbol is identifier

CLASS $\hat{=}$ 3 means Symbol is integer number

CLASS $\hat{=}$ 4 means Symbol is real number

CLASS $\hat{=}$ 5 means Symbol is string

CLASS $\hat{=}$ 6 means Symbol is operator

CLASS $\hat{=}$ 7 means Symbol is separator

CLASS $\hat{=}$ 8 means Symbol is undefined operator

(3.4.5) INDEX is SYMBOL_INDEX means

INDEX ≥ 1 or $\hat{=}$ VALUE

and

INDEX $\hat{=} \geq 1$ means

$$\text{Symbol} \in \begin{cases} \text{keyword} \\ \text{identifier} \\ \text{operator} \\ \text{separator} \\ \text{undefined operator} \end{cases}$$

INDEX $\hat{=}$ VALUE means

$$\text{Symbol} \in \begin{cases} \text{integer number} \\ \text{real number} \\ \text{string} \end{cases}$$

and

VALUE $\hat{=}$ 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 $\hat{=}$ number of the first character of a symbol in a Txt_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

$I \in \{0, 555\}$

and

ERROR $\triangleq$ 0 means

LADDR $\triangleq$ HEAP . Log_Add

and

HEAP((HEAP(Log_Add)) + HPTAG) $\triangleq$ TYPE

HEAP((HEAP(Log_Add)) + HPSPAC) $\triangleq$ new_SIZE SIZE

HEAP((HEAP(Log_Add)) + HPLAD) $\triangleq$ LADDR

or

ERROR $\triangleq$ 555

and

LADDR $\triangleq$ 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 > 0

and

HEAP . SYM_TAB (INDEX) ≙ Symbol @

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

or

ERROR ≙ > 0

I is INDEX means I ≙ Null

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

(4.1) Input Assertions

Global assertions and Scanner input assertions are valid

I is LINE means I ≙ Txt_Ln_Num

I is CHAR means I ≙ Num_First_Char of Syntax_Error

I is N means I ≙ n length of STRING

STRING ≙ Syntax_Error_Message

(4.2) Output Assertions

I is RETCOD means

I ∈ {0, 1}

and

RETCOD ≙ 0, message was printed

or

RETCOD ≙ 1, message was not printed

3.4.5 Subroutine STAX (SWITCH, DELAY)

(5.1) Input Assertions

I is SWITCH means I ≙ 0

I is DELAY means I ≙ 150

(5.2) Output Assertions

I is SWITCH means

I ∈ {0,1

and

SWITCH = 0 means no interrupt

or

SWITCH = 1 means user interrupt

3.4.6 Subroutine GETCHR(STRG,J,ARG3)

(6.1) Input Assertions

Global assertions and Scanner input assertions are valid

I is WORD means

I ≙ Txt_Add@ + Txt_Ln_Disp + TXTLIN

I is J means

I ∈ {1,...,Total_Char}

(6.2) Output Assertions

I is ARG3 means I ≙ CHAR

and

CHAR is (J)

3.4.7 Subroutine PUTCHR(SYM,K,CHAR)

(7.1) Input Assertions

SYM ≙ two words

I is K means

I ∈ {1,...,8}

I is CHAR means

I ∈ {1,...,255}

(7.2) Output Assertions

SYM(2) ≙ 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 3.3.1)}

INITIALIZE {2. see para. 3.3.3}

```
RESCAN ← FALSE
ERRFLG ← FALSE
CLFLG ← FALSE
TYPE_TOK ← "ITOK"
SYMCT ← 0
ERRCT ← 0
CHARNU ← 0
CTAB ← to figure 2-9
KWTAB ← to figure 2-10
CLTAB ← to figure 2-11
LINST ← 0
SYM ← 0
SYMST ← 0
ERRST ← 0
```

```
Proc_Tab_Block@ ← HEAP(5)   {3. see para. 3.2.3}
Txt_Log_Add ← HEAP(Proc_Tab_Block@ + PINDEX + PRCTXT)
  {4. see para. 3.3.1.2}
Tok_Log_Add ← HEAP(Proc_Tab_Block@ + PINDEX + PRCTOK)
Txt_ADD@ ← HEAP(Txt_Log_Add)                    {5. see para. 3.2.6
Txt_Total_Lns ← HEAP(Txt_Add@ + TXTNLIN)        {6. see para. 8.3.1.3
                                                    and para. 3.3.1.4}
```

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. 3.3.3.3}
         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 + TXTFLK)
 {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_Disp
END CASE 3
```

{17. All Add@, First Txt_Ln_Disp, Next Txt_Ln_Disp, and First Tok_Ln_Disp 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_Disp + TXTTOK) = 0
```

THEN {23. Text line has not been SCANNED, see para. 3.3.1.4}

```
        CALL  LNSCAN (ERRFLG,Ln_Space_Used, Txt_Ln_Disp,
              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 8
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_Ln_Disp,RESCAN,Tok_Ln_Disp)
          {32. Tok_Ln header has been assigned following the
               Last Tok_Ln.  Tok_Ln . TOKFLK 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_Ln_Disp + TOKFLK) ←
                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_Ln_Disp + TOKNUM) ← 0
     ELSE {35. Text line was error free.}
          HEAP(Tok_Add@ + Tok_Ln_Disp + TOKNUM) ← SYMCT
          LOOP 2  iteration 1 thru SYMCT
               Token ← 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) ← Tok_Space_Used + Ln_Space_Used
HEAP(Tok_Add@ + TOKBOT) ← Tok_Space_Used + Ln_Space_Used
HEAP(Tok_Add@ + TOKNLIN) ← HEAP(Tok_Add@ + TOKNLIN) + 1
 {38. Tok_Block header has been updated, see para. 3.3.2.4}
SWITCH ← 0
```

```
DELAY ← 150
CALL  STAX (SWITCH,DELAY)  {39. see para. 3.4.5}
CASE 11  IF SEITCH > 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_Disp + TXTLNUM) =
                 HEAP(Tok_Add@ + Tok_Ln_Disp + TOKLNUM)
     THEN  Txt_Ln_Disp ← HEAP(Txt_Add@ + Txt_Ln_Disp +
                                         TXTFLK)
           Tok_Ln_Disp ← HEAP(Tok_Add@ + Tok_Ln_Disp +
                                         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. 3.3.3.3.3}
          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(Txt_Log_Add)
Txt_Ln_Total_Disp ← Txt_Add@ + Txt_Ln_Disp
Total_Char ← HEAP(Txt_Ln_Total_Disp + TXTLLEN)
{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. 3.3.3.3.2}

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, com-
               mand 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 iden-
            tifier and CLASS = 2.}
        CASE 5  IF SYMCODE = 2 OR CLASS = 2
            THEN {26. Symbol is an identifier or unde-
                    fined 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 CONDITION 3

END LOOP 2

Ln_Space_Used ← (SYMCT + 1) x 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 Ln_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 Ln_Space_Used equals three.}

END SUBROUTINE LNSCAN

3.5.3 SUBROUTINE FORM (CHARNU,SYML,Total_Char,ERCODE,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 + TXTLLEN)

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  PUTCHR(SYM,K,CHAR)
                      {12. see para. 3.4.7}
                      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
                SYMCODE ← 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  NUMPAC(Num_First_Char,SYML,
                                         VALUE,ERCODE)
                            INDEX ← VALUE
                            RETURN {18. Token data complete for
                                    number.}
                        ELSE {19. Symbol may contain more charac-
                              ters.}
                            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. 3.3.3.4.4}
    SYMCODE ← 2 {24. see para. 3.3.3.4.3}
    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.
                                    3.3.3.4.1}
                        RETURN
                    ELSE {29. Symbol is error free.}
                        LOOP 5  iteration Num_First_Char
                                thru CHARNU
                            CHAR ← LINST(iteration)
                            K ← K + 1
                            CALL  PUTCHR(SYM,K,CHAR)
                            {30. Symbol is packed four
                            characters per word
                            into array SYM; see
                            state change, para.
                            3.3.3.1.5}
                        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 $\in \{1,2\}$

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

INDEX is $\in \{0, \text{VALUE}, \text{INDEX}\}$

CHARNU is iteration of last character of symbol

SYMCODE is $\in \{1,2,3\}$

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 $\leftarrow$ 0

Column $\leftarrow$ 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 KWTAB(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
```

END LOOP 2

15. A match could not be found; symbol not a keyword.

RETURN

END CASE 2

16. Output Assertions

CLASS ∈ {0,1}

16.1 and CLASS = 0 no match was found

16.2 and CLASS = 1 match was found

and INDEX assigned; see figures 2-10 and 2-11.

END SUBROUTINE TABLE

3.5.5 SUBROUTINE NUMPAC (Num_First_Char,SYML,VALUE,ERCODE)

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_Ln 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 encoun-
                           tered.}
                          Decimal_Count ← Decimal_Count + 1
                     ELSE {8. A digit has been added to
                           the symbol.}
                          NUM ←LINST(Loop 1 iteration)
                END CASE 3
                GOTO End Loop 1
      END CASE 2
        {9. Loop until CHAR = NUMBER OR iteration = 10}
    END LOOP 2
      {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 {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)
    ERRCODE ← ERRST(iteration,3)
  {4. Variables are set to Error_Data for the line of text.}
    CONDITION 1  ERRCODE = 1
      {5. Error was too many characters in the symbol.}
        THEN   N ←21 {6. N = number of characters in
                        message}
               CALL  ERRPRT (PINDEX,Line_Num,CHAR,N,String_
                             Too_Many_Char,RETCOD)
                 {7. see para. 3.4.4}
    END CONDITION 1
    CONDITION 2  ERRCODE = 2
      {8. Error was no closing quote mark on a string.}
        THEN   N ←26 {9. N = number of characters in
                        message}
               CALL  ERRPRT (PINDEX,Line_Num,CHAR,N,String_
                             Illegal_Char,RETCOD)
                 {10. see para. 3.4.4}
    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. 3.3.3.3.1
       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_Disp ← HEAP(Tok_Add@ + TOKTOP) {6. see para. 3.3.2.4}

Txt_Ln_Num ← HEAP(Txt_Add@ + Txt_Ln_Disp + TXTLNUM)
{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 ≤ Tok_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@ + Tok_Ln_Disp + TOKFLK) ←
HEAP(Tok_Add@ + Ln_Disp + 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_Disp + TOKFLK) ←
Tok_Ln_Disp

{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.}

```
                    HEAP(Tok_Add@ + Tok_Ln_Disp + TOKFLK)←
                         Ln_Disp
                    HEAP(Tok_Add@ + Last_Ln_Disp + TOKFLK)←
                         Tok_Ln_Disp
                    {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_Disp ← Ln_Disp
               Ln_Disp ← HEAP(Tok_Add@ + Last_Ln_Disp + TOKFLK)
               Tok_Last_Ln_Num ← Tok_Ln_Num
               Tok_Ln_Num ← HEAP(Tok_Add@ + Ln_Disp + 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).

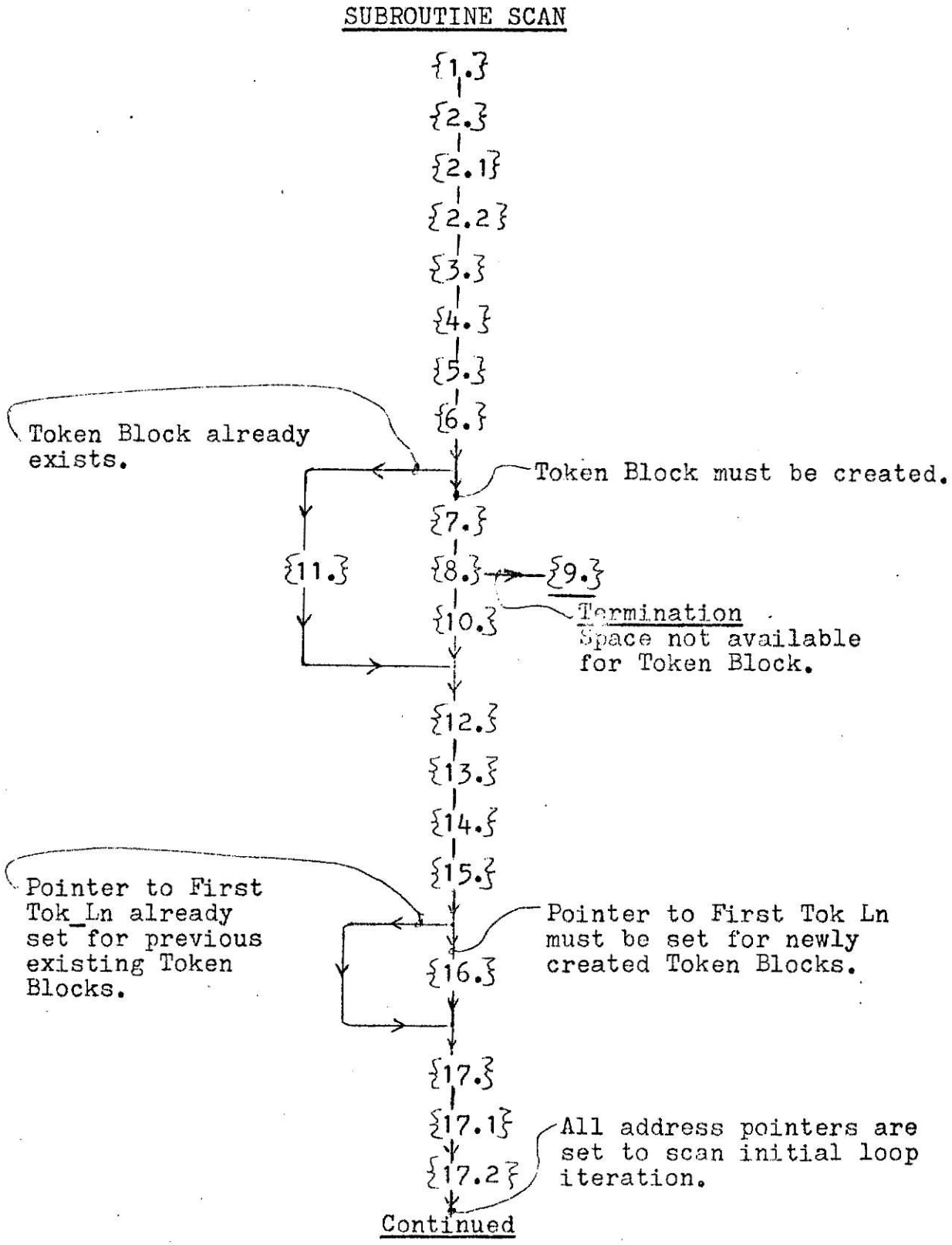


Figure 4-1 Subroutine SCAN State Transition Diagram

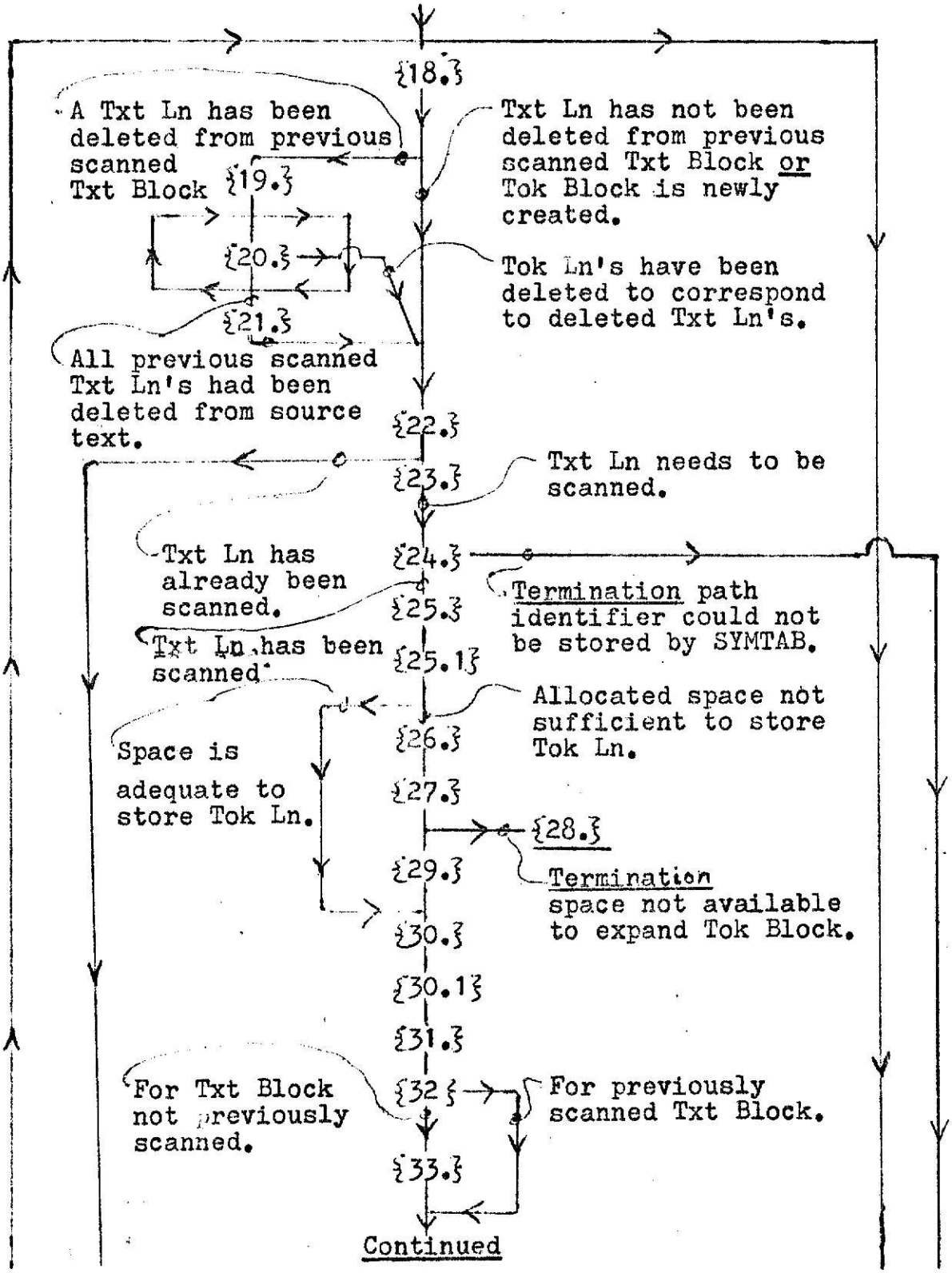


Figure 4-1 (Continued)

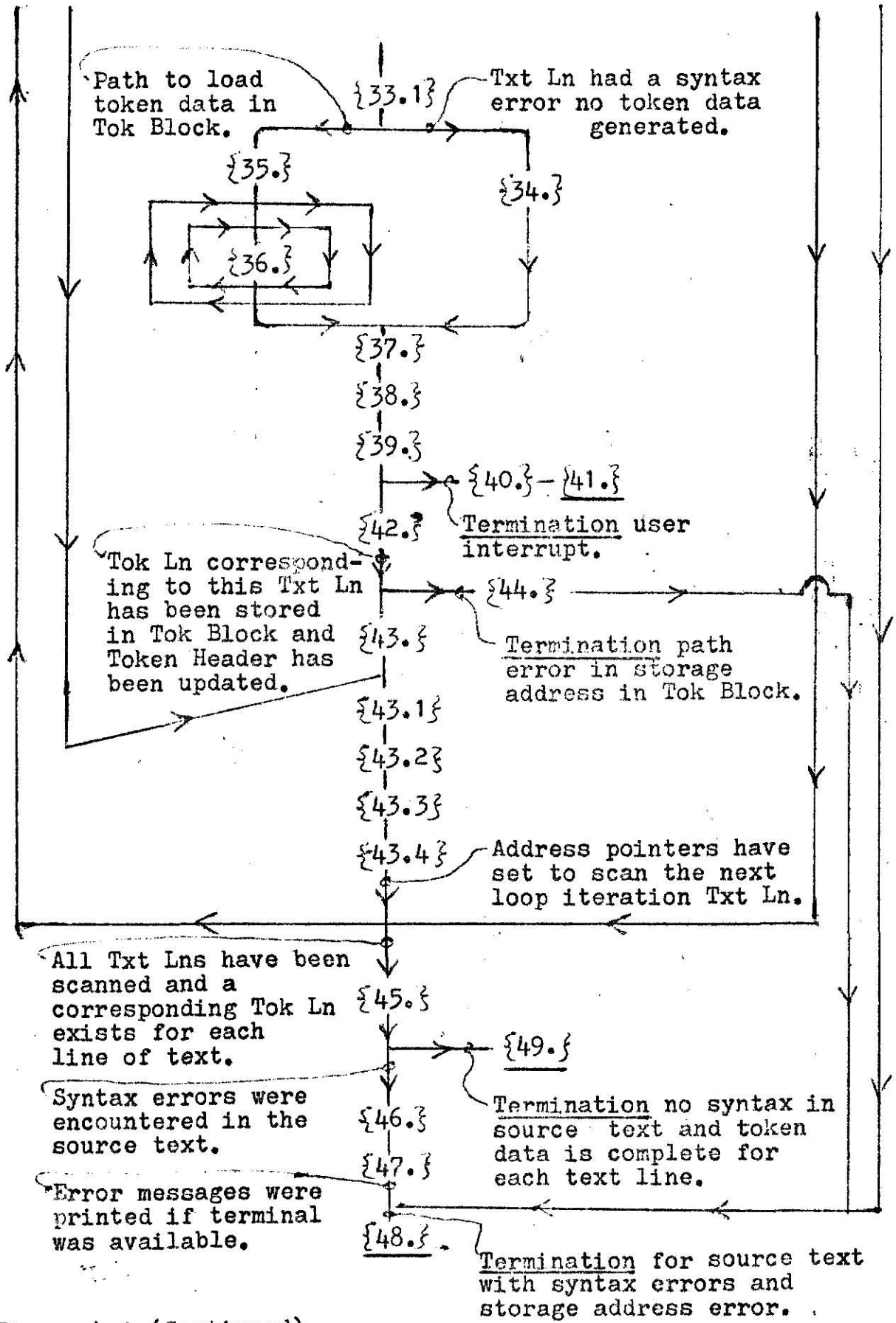


Figure 4-1 (Continued)

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 preceeding 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.

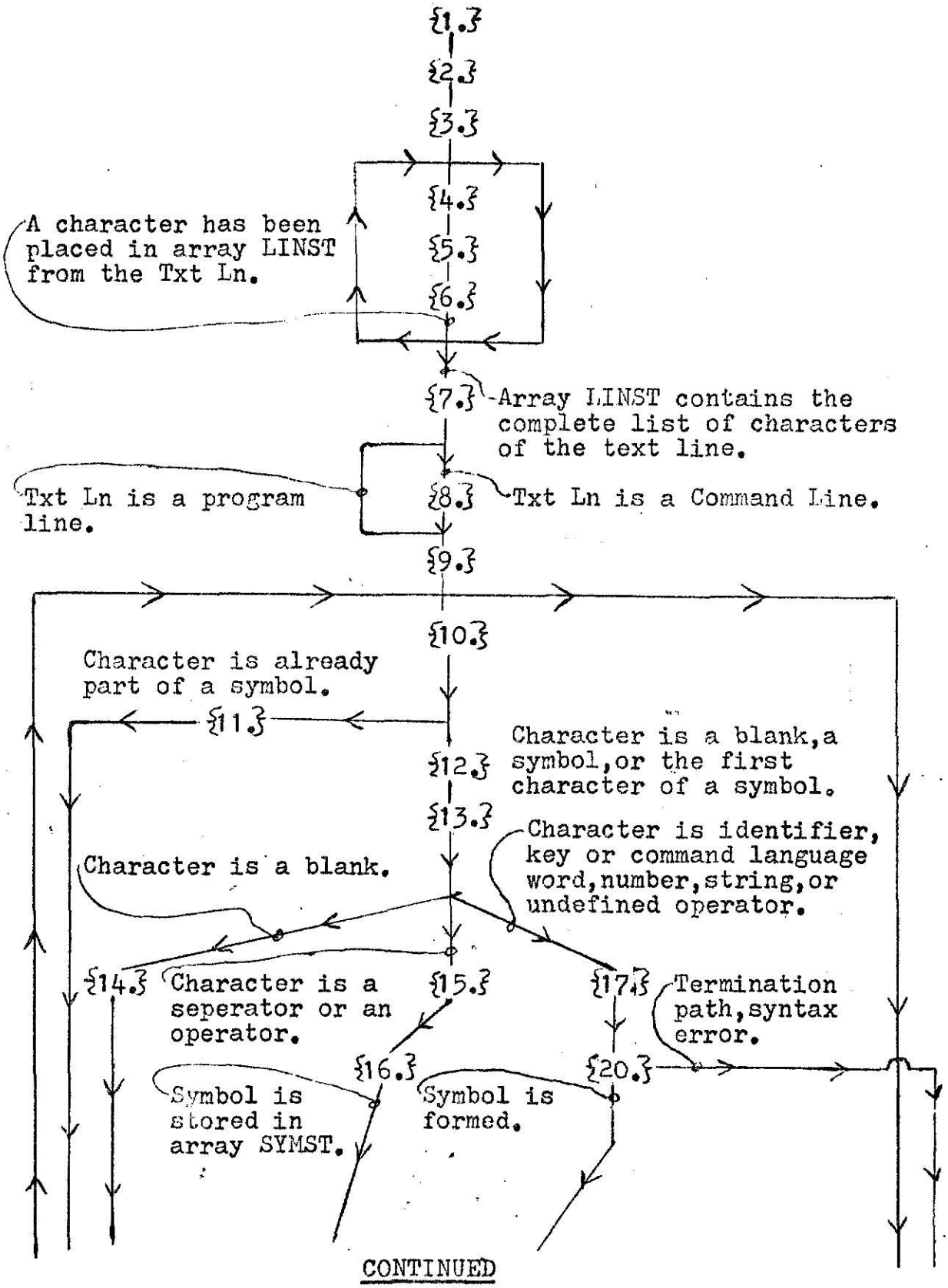
A number or string (Path 10,12,13,17,20,23,29,30,31).

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



CONTINUED

Figure 4-2 Subroutine LNSCAN State Transition Diagram

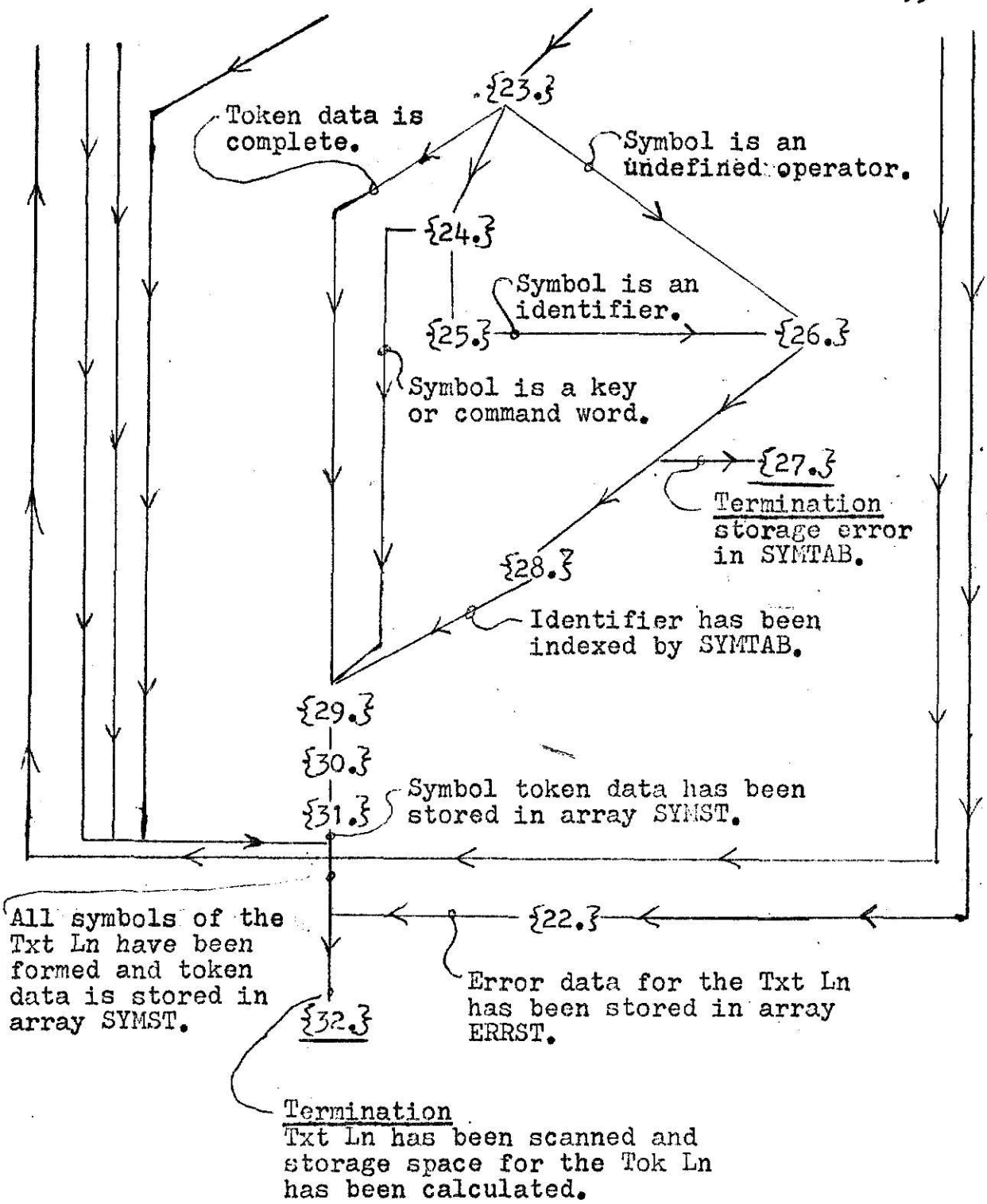


Figure 4-2 (Continued)

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

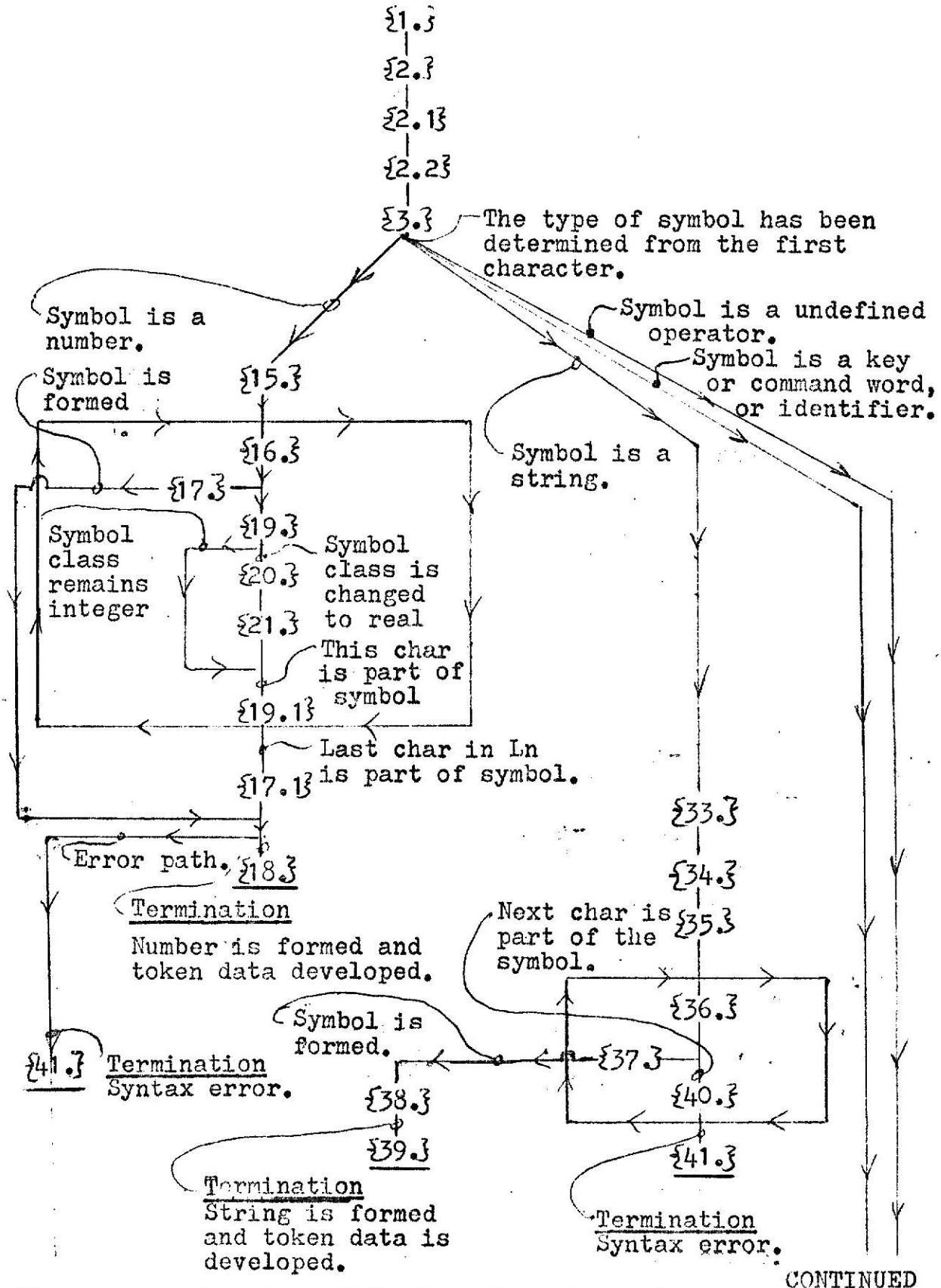


Figure 4-3 Subroutine FORM State Transition Diagram

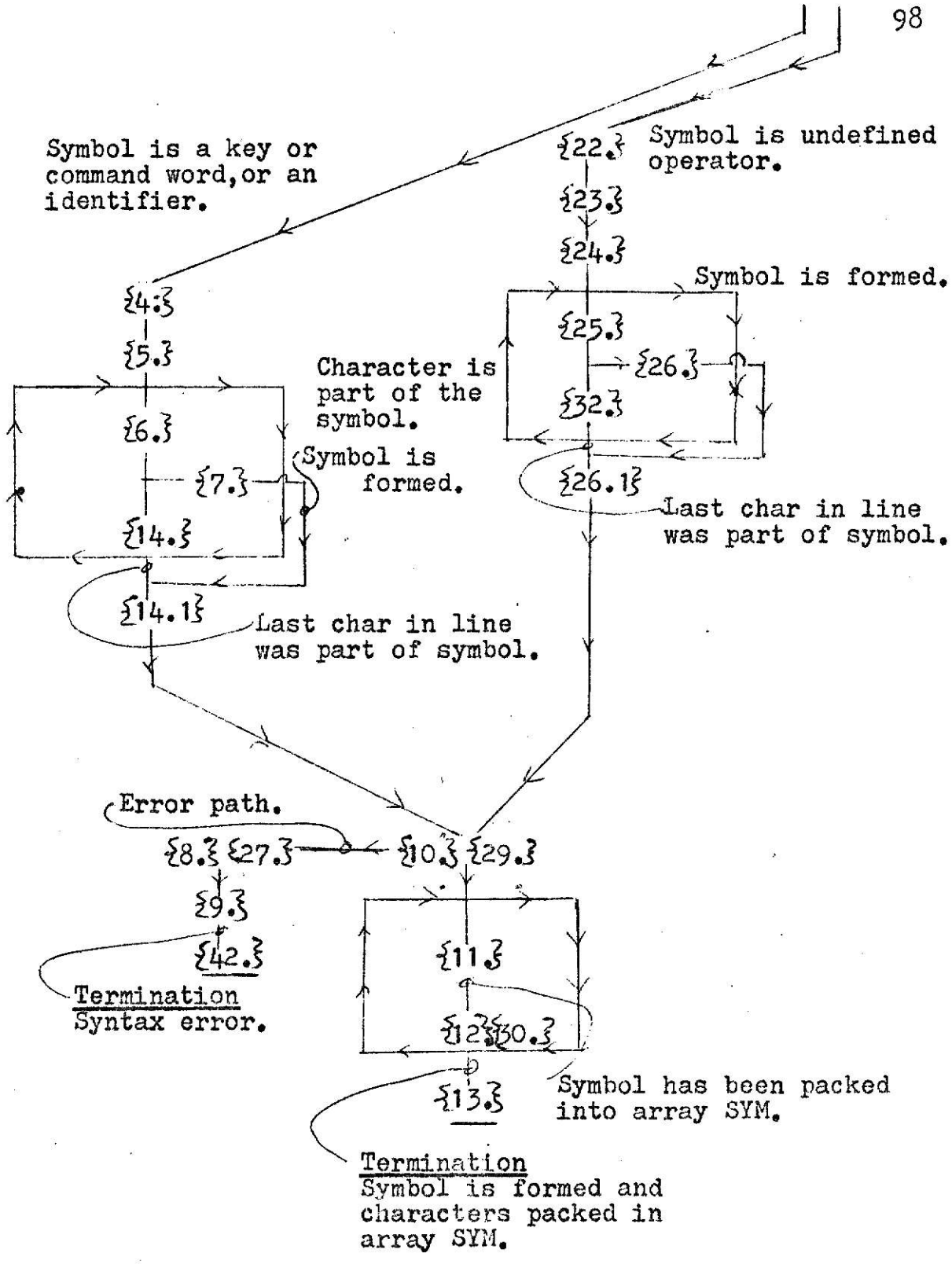


Figure 4-3 (Continued)

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).

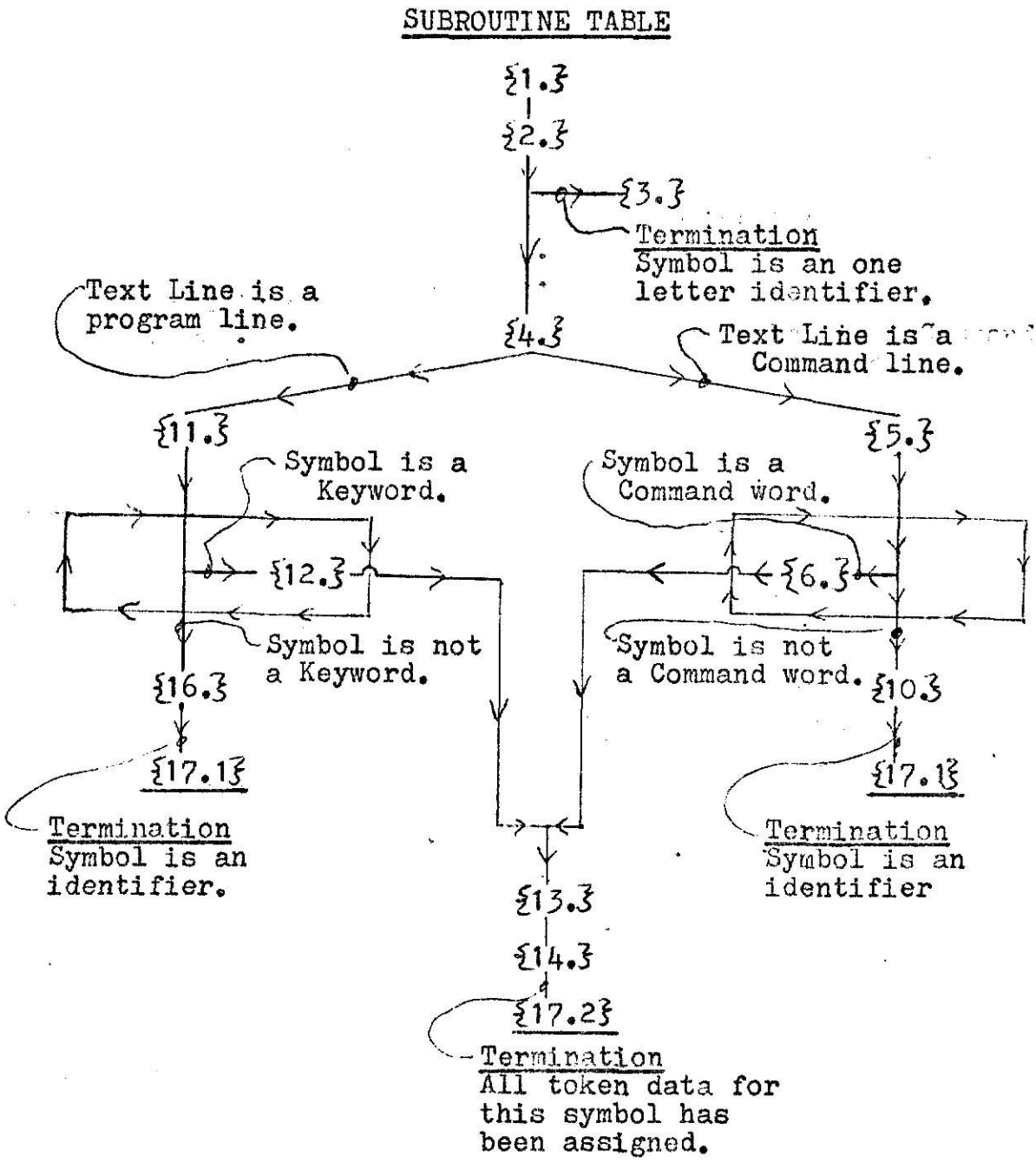


Figure 4-4 Subroutine TABLE State Transition Diagram

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 > 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:

SUBROUTINE NUMPAC

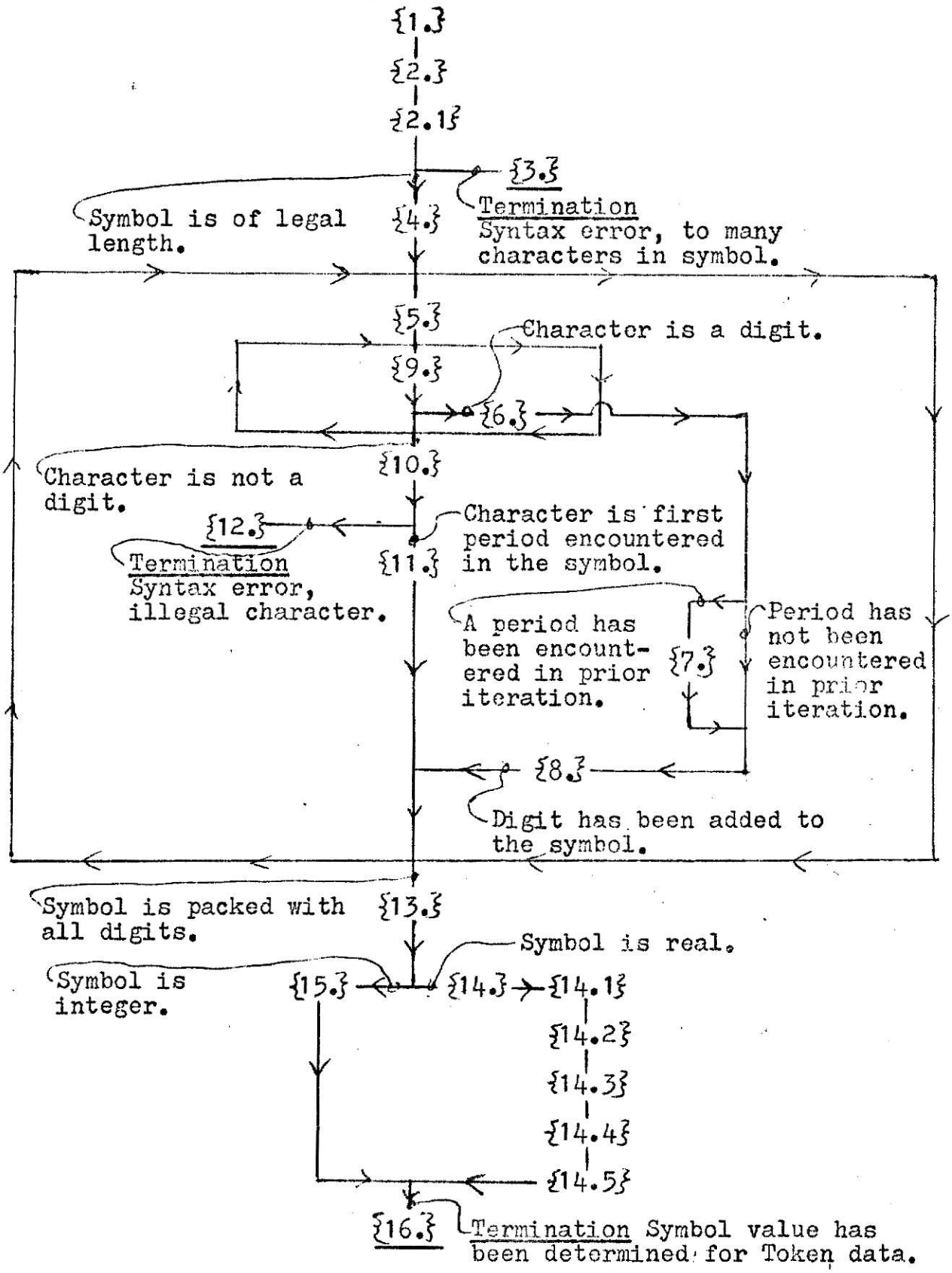


Figure 4-5 Subroutine NUMPAC State Transition Diagram

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

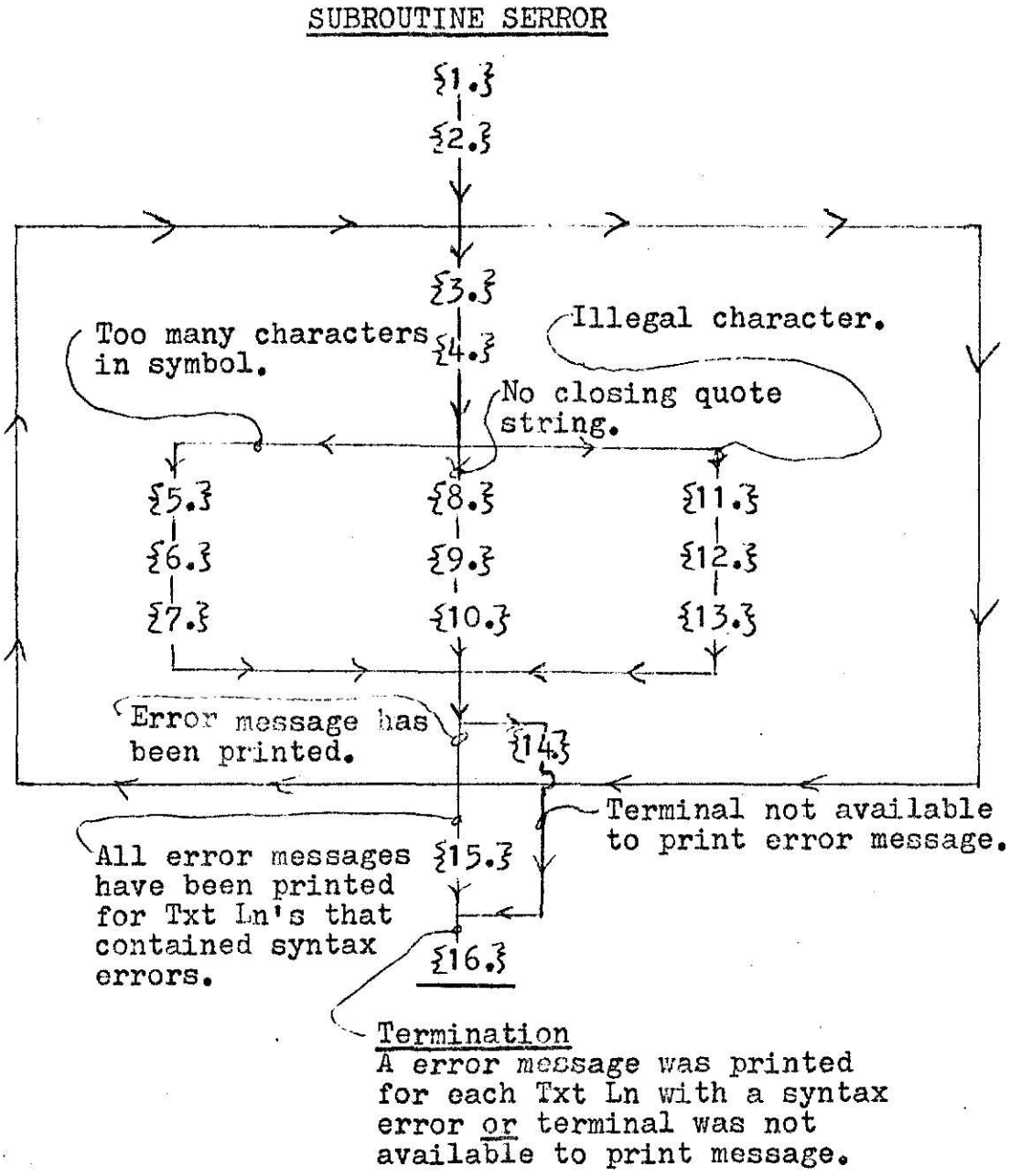


Figure 4-6 Subroutine SERROR State Transition Diagram

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

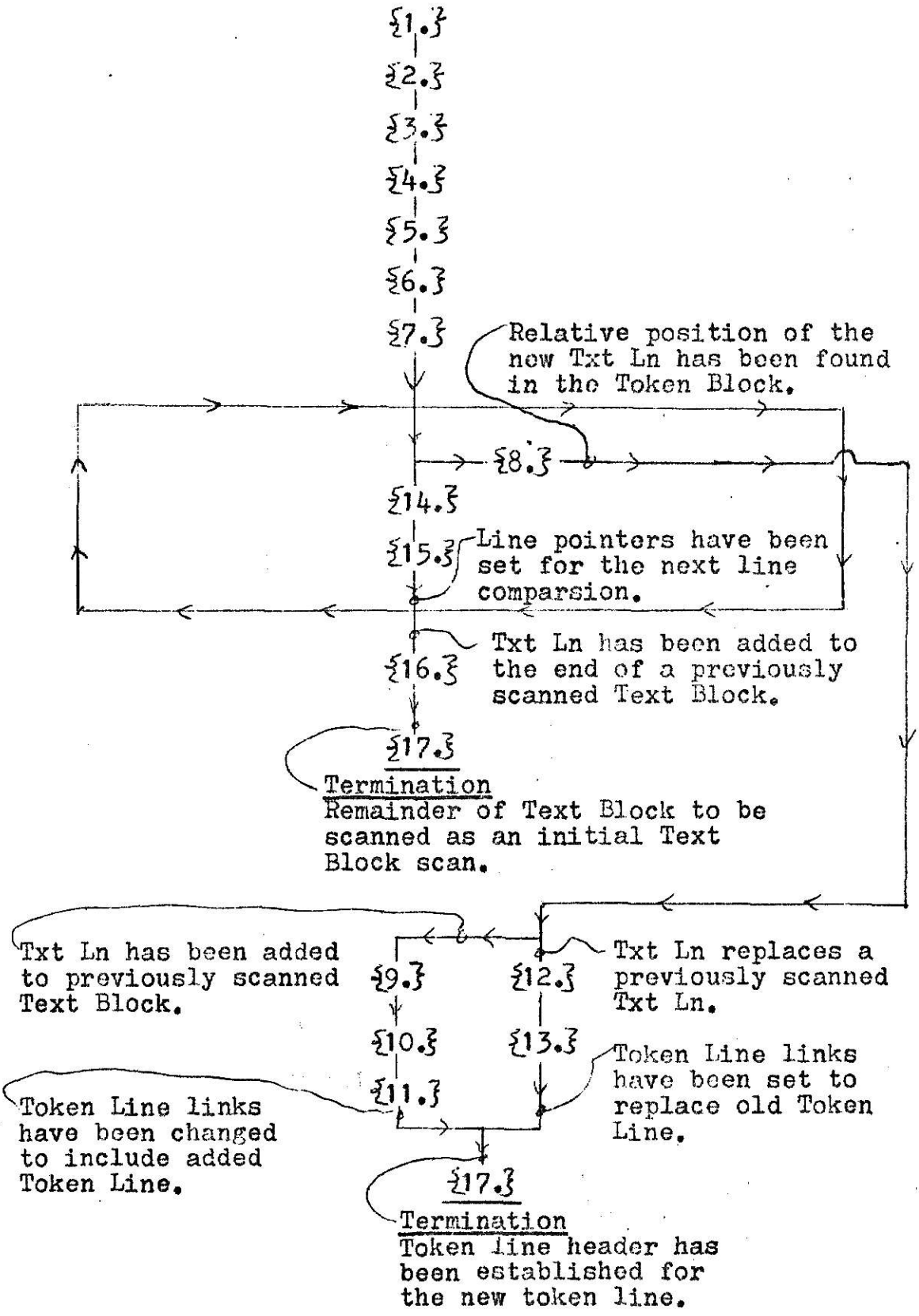


Figure 4-7 Subroutine LINFIN State Transition Diagram

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 documantation 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

Allen, C. "The Application of Formal Logic to Programs and Programming," J B M Syst Join, 1971.

Hankley, W., and P. Fisher. "Top-Down Refinement of Assertions, "3rd Texas Conf on Computing Systems, 1974.

Linden, T. "A Summary of Progress Toward Proving Program Correctness," FJCC, 1972.

Manna, Z. Mathematical Theory of Computation, 1974.

Mitrione, M. "The Language and Program Documentation of a Student Designed Interpreter," Masters Report Kansas State University, 1975.

Wirth, N. "Program Development by Stepwise Refinement," Com ACM, April, 1971.

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.

Memory Words	Type Data Stored	Offset Variable
1	Data Area Code	HPTAG = 1
2	Allocated Space	HPSPAC = 2
3	Space Used	HPSIZ = 3
4	Logical Address of Data Area	HPLAD = 4
5	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.	NONE

(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:

Memory Word	Type Data Stored	Offset Variable
1	Number of characters in Procedure name	PRCNLEN = 1
2	First four characters of Procedure name	PRCNAM = 2
3	Last four characters of Procedure name	NONE
4	Procedure Status	PRCSTAT = 4
5	Logical Address for Text Location	PRCTXT = 5
6	Logical Address for Token Location	PRCTOK = 6
7	Logical Address for Symbol Table Location	PRCSYM = 7
8	Logical Address for Code Location	PRCCOD = 8

(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.

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

(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.

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

(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.

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

(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.

Memory Word	Type Data Stored	Offset Variable
1	Line number	TOKLNUM = 1
2	Index from the first memory word of the Token storage area to the next line of tokens	TOKFLK = 2
3	Total number of tokens in line	TOKNUM = 3

(2.4.3) Token

There are three memory words associated with each token.

Memory Word	Type Data Stored
1	Class identifier (see a below)
2	Index or Value (see b below)
3	Number of the first character of the symbol in the line of source text

(2.4.3.1) Class Identifiers

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

Integer Value	Type Symbol
1	Keyword or Command Language Word
2	Identifier
3	Integer number
4	Real number
5	String
6	Operator
7	Separator
8	Undefined Operator

(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

translation table.

(2.4) Character Table initialization data

EBCDIC Characters	Decimal Location	Column 1	2	3
blank	1 - 73	1	0	0
¢	74	2	6	1
.	75	2	7	1
<	76	2	6	2
(	77	2	7	2
+	78	2	6	3
|	79	2	6	4
&	80	2	6	5
blank	81 - 89	1	0	0
!	90	2	7	3
$	91	3	3	0
*	92	2	6	6
)	93	2	7	4
;	94	2	7	5
¬	95	2	6	7
-	96	2	6	8
/	97	2	6	9
blank	98 - 105	1	0	0
¦	106	2	7	6
,	107	2	6	10
%	108	2	6	11
_	109	2	6	12
>	110	2	6	13
?	111	2	7	7
blank	112 - 120	1	0	0
`	121	2	7	8
:	122	2	7	9
#	123	2	6	14
@	124	2	6	15
'	125	2	7	10
=	126	2	6	16
"	127	3	4	0
blank	128	1	0	0
a	129	3	1	0
b	130	3	1	0
c	131	3	1	0
d	132	3	1	0
e	133	3	1	0
f	134	3	1	0
g	135	3	1	0
h	136	3	1	0
i	137	3	1	0
blank	138 - 144	1	0	0
j	145	3	1	0
k	146	3	1	0
l	147	3	1	0
m	148	3	1	0
n	149	3	1	0
o	150	3	1	0

EBCDIC Characters	Decimal Location	Column 1	2	3
p	151	3	1	0
q	152	3	1	0
r	153	3	1	0
blank	154 - 160	1	0	0
~	161	2	0	0
s	162	3	1	0
t	163	3	1	0
u	164	3	1	0
v	165	3	1	0
w	166	3	1	0
x	167	3	1	0
y	168	3	1	0
z	169	3	1	0
blank	170 - 191	1	0	0
{	192	2	7	11
A	193	3	1	0
B	194	3	1	0
C	195	3	1	0
D	196	3	1	0
E	197	3	1	0
F	198	3	1	0
G	199	3	1	0
H	200	3	1	0
I	201	3	1	0
blank	202 - 203	1	0	0
⌐	204	2	6	17
blank	205	1	0	0
¥	206	2	6	18
blank	207	1	0	0
}	208	2	7	12
J	209	3	1	0
K	210	3	1	0
L	211	3	1	0
M	212	3	1	0
N	213	3	1	0
O	214	3	1	0
P	215	3	1	0
Q	216	3	1	0
R	217	3	1	0
blank	218 - 223	1	0	0
\	224	2	6	19
blank	225	1	0	0
S	226	3	1	0
T	227	3	1	0
U	228	3	1	0
V	229	3	1	0
W	230	3	1	0
X	231	3	1	0
Y	232	3	1	0
Z	233	3	1	0
blank	234 - 235	1	0	0
▪	236	2	6	20

EBCDOC Characters	Decimal Location	Column 1	2	3
blank	237 - 239	1	0	0
0	240	3	2	0
1	241	3	2	0
2	242	3	2	0
3	243	3	2	0
4	244	3	2	0
5	245	3	2	0
6	246	3	2	0
7	247	3	2	0
8	248	3	2	0
9	249	3	2	0
!	250	2	6	21
blank	251 - 255	1	0	0

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

Column		1	2	3	4	5	6	7
Row	1	DO	END	CASE	BEGI	ACCE	ENDP	ENDW
	2	IF	OUT	ELSE	FALS	EXPO	0	EXTE
	3	IN	0	EXIT	WHILE	GLOB	0	0
	4	FI	0	GOTO	WRITE	RETU	0	0
	5	0	0	PROC	0	0	0	0
	6	0	0	READ	0	0	0	0
	7	0	0	THEN	0	0	0	0
	8	0	0	TRUE	0	0	0	0

Column	1	2	3	4	5	6	7
Row 9	0	0	QUIT	0	0	0	0
10	0	0	CALL	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

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

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.

High Level Design Language	FORTRAN	Meaning
Allocated_Space	SPACE	Amount of space allocated to a block.
Column	COL	Column number used to search Keyword and Command Language Word Tables.
ERROR	EXERR	Output parameter ERROR
Decimal_Counter	IQ	Used to indicate number of decimal places in a real number.
Last_Ln_Disp	LASTLN	Displacement to previous Token Line. Used in a line by line search of the Token Block in Subroutine LNFIN.
	LOAD	Used to temporarily hold Token Data prior to placement in the Token Block storage word.
Ln_Disp	LNDISP	Line displacement to the current Token Line. Used in a line by line search of Token Block in Subroutine LNFIN.
Ln_Space_Used	LINUSE	Storage words necessary to store a line of Tokens.
	NEXT	Counter for array RVALUE.
Next_Last_Char	STOP	Next to the last character in a number symbol.
Num_Code	NT	Number code used to indicate if a period has been encountered in a number symbol.
Num_First_Char	NFC	Number of the first character of a symbol in a line of text.
PINDEX	INDEXP	Input parameter PINDEX.
Pointer	PT	Pointer used with conditional GOTO statements.

Table C 1-1 Variable Translation Table

High Level Design Language	FORTRAN	Meaning
Proc_Tab_Block@	PRCBAS	Absolute address of Procedure Table Block
	PTLABV	Points to line above in Token Block.
)	PARN	Closing parenthesis.
	RVALUE	Array used to hold real number values.
Space_Used	TKUSED	Amount of allocated space used in the Token Block.
String_Illegal_Char	STRILC	Syntax error message for illegal character.
String_No_Close_Quote	STRNCQ	Syntax error message for no closing quote mark on a string.
String_Too_Many_Char	STRTMC	Syntax error message for too many characters in a string.
SYML	LENGTH	Used for SYML in Subroutine NUMPAC.
Tok_Add@	TOKADD	Absolute address in the Token Block.
Tok_Block@	TOKSAD	Absolute address of the Token Block.
Total_Char	TCHAR	Total characters in a text line.
Tok_Last_Ln_Num	TOKLLN	Line number of previous token line used in a line by line search of the token block.
Tok_Ln_Disp	TOKDTL	Token line displacement.
Tok_Log_Add	TOKLAD	Token logical address.

Table C 1-1 (continued)

High Level Design Language	FORTRAN	Meaning
Tok_Total_Lines	TOKTLN	Total lines of Tokens in Token Block.
Txt_Add@	TXTADD	Absolute Address in Text Block.
Txt_Ln@	TXTLST	Absolute Adress of a Text Line.
Txt_Block@	TXTSAD	Absolute address of Text Block.
Txt_Ln_Disp	TXTDTL	Text line displacement.
Txt_Ln_Num	TXTLIN	Text line number.
Next Txt_Ln_Disp	TXTDNL	Next text line displacement.
Txt_Log_Add	TXTLAD	Text logical address.
Txt_Total_Lines	TOTLIN	Total text lines in Text Block.
TYPE	TOK	Token Code.

Table C 1-1 (Continued)

```
      SUBROUTINE SCAN(INDEXP,EXERR)
```

{1.}

```
      COMMON /HEAP/NHEAP,HEAP(4CC)
      COMMCN/PRCOFF/PRCNLN,PRCNAM,PRCTXT,PRCTOK,PRCSYM,PRCCOD,
     1              PRCSTA ,PRCLEN
      COMMON /HPOFF/HPTAG,HPSPAC,HPSIZ,HPLAD,HPCBJ
      COMMCN/TOKOFF/TOKTOP,TOKBOT,TCKNLN,TCKLNC,TCKFLK,TCKNUM
      CCMMCN/TXTOFF/TXTTOP,TXTBOT,TXTNLN,TXTLNO,TXTFLK,TXTBLK,
     *TXTTCK,TXTNST,TXTLLEN,TXTLIN,TXTSTA
      COMMON/SCANER/CTAB(255,3),KWTAB(1C,7),CLTAB(LO,7),LINST(64),
     *SYMST(15,3),ERRST(64,3),SYM(2),TXTLAD,TOKLAD,PINDEX,ERROR,
     *CLFLAG,PRCBAS,ERRCT,RVALUE(15),NEXT
      SYMCT = C
      ERRCT=C
      ERROR=0
      RESCAN= .FALSE.
      CLFLAG= .FALSE.
      TOK= "ITOK"
```

{2.}

```
      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= INDEXP
      EXERR= C
```

{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.}

```
      TXTLAD= HEAP(PINDEX+PRCBAS+PRCTXT)
```

{4.}

```
      TOKLAD= HEAP(PINDEX+PRCBAS+PRCTCK)
```

{5.}

```
      TXTSAD = HEAP(TXTLAD)
      TOTLIN = HEAP(TXTSAD+TXTNLN)
```

{6.}

```
      IF(HEAP(TOKLAD) .GT. C) GC TC 1
```

{7.}

```
      SIZE= TOTLIN*15*3
      CALL GET(SIZE,TCK,TCKLAD,ERRCR)
```

{8.}

```
      IF(ERRCR .EQ. 0)  GOTO  3
      EXERR = ERROR
```

{9.}

```
      RETURN
3     HEAP(PINDEX+PRCPAS+PRCTCK)= TCKLAD
```

{10.}

```
      GO TO 2
```

{11.}

```
1     RESCAN= .TRUE.
```

{12.}

```
2     TXTSAD= HEAP(TXTLAD)
      TOKSAD= HEAP(TOKLAD)
```

{13.}

```
      TXTCTL=HEAP(TXTSAD+TXTTOP)
      TXTCNL=HEAP(TXTSAD+TXTCTL+TXTFLK)
```

{14.}

```
      TOKCTL=HEAP(TOKSAD+TOKTOP)
```

{15.}

```
      IF(TOKCTL .GE. 7)  GOTO 5
```

{16.}

```
      TOKCTL = 7
      HEAP(TCKSAD+TCKTOP)=TCKCTL
      HEAP(TOKSAD+HPSIZ)= TOKCTL
```

{17.}

```
5     TOKTLN = HEAP(TCKSAD+TOKNLN)
```

{17.1 see para. 4.3.2.4}

```
      TXTADD = TXTSAD+HEAP(TXTSAD+TXTTOP)
      TOKADD=TOKSAD+HEAP(TOKSAD+TCKTOP)
```

{17.2 see para. 4.3.1.4 & 4.3.2.4}

```
      DO 1C I= 1,TC1LIN
```

{18.}

```
        IF(HEAP(TCKADD+TCKLNG)  .GE. HEAP(TXTADD+TXTLNO)  .OR.
     *    .NOT. RESCAN) GOTO 11
```

{19.}

```
        DO 2C K= I,TCKTLN
           IF(HEAP(TOKADD+TOKLNG) .GE. HEAP(TXTADD+TXTLNC))  GOT
             TCKADC=TCKSAC+HEAP(TCKSAD+PTLABV+TOKFLK)
             HEAP(TCKSAD+PTLABV+TCKFLK )=HEAP(TCKADC+TCKFLK)
             HEAP(TOKSAD+TOKNLN)  = HEAP(TCKSAD+TCKNLN)  - 1
```

{20.}

```
2C      CONTINUE
```

{21.}

```
        RESCAN= .FALSE.
```

{22.}

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

{23.}

```
        CALL LNSCAN(ERRFLG,LINUSE,TXTDTL,SYMCT,&900)
```

{24.}

```
         TKUSED=HEAP(TCKSAD+HPSIZ)
```

{25.}

```
      SPACE=HEAP(TCKSAD+HPSPAC)
```

{25.1 Space = Allocated_Space is para. 4.3.2.4}

```
         IF(TKUSED+LINUSE .LE. SPACE) GO TO 14
```

{26.}

```
         SIZE= (TOTLIN-I) * 45
         CALL EXPAND(SIZE,TCKLAC,ERRCR)
```

{27.}

```
         IF(ERRCR .EQ. 0)  GOTO  14
```

{28.}

```
         EXERR = ERROR
```

```
      RETURN
```
{29.}

```
14    TOKSAD = HEAP(TOKLAD)
      TXTSAD = HEAP(TXTLAD)
```
{30.}

```
      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+TOKLAC) = HEAP(TXTSAD+TXTDTL+TXTLNO)
```
{31.}

```
      IF(RESCAN) CALL LINFIN(TXTDTL,RESCAN,TOKDTL,& 15)
```
{32.}

```
      HEAP(TOKSAD+TOKDTL+TOKFLK) = HEAP(TOKSAD+HPSIZ) + LINUSE
```
{33.}

```
15    TOKADD = TOKSAD+TOKDTL+TOKNUM
```
{33.1 see para. 4.3.2.5}

```
      IF(.NOT. ERRFLG) GO TO 16
```
{34.}

```
      HEAP(TOKADD)= 0
      GO TO 17
```
{35.}

```
16    HEAP(TOKADD)= SYMCT
      DO 3C M= 1,SYMCT
         DO 40 N= 1,3
            LOAD= SYMST(M,N)
            TOKADD= TOKADD+ 1
            IF(SYMST(M,1) .EQ. 5 .AND. N .EQ. 2) CALL REAL(HEAP(
     *      TOKADD),RVALUE(LOAD),& 4C)
            HEAP(TOKADD)= LOAD
4C          CONTINUE
```
{36}

```
3C    CONTINUE
```
{37.}

```
17      HEAP(TCKSAD+HPSIZ)=TKUSED+LINUSE
        HEAP(TOKSAD+TOKBOT)=TCKDTL
        HEAP(TOKSAD+TOKNLIN)=HEAP(TOKSAD+TCKNLN) +1
```

{38.}

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

{39.}

```
        IF(SWITCH .EQ. C ) GO TO 12
```

{40.}

```
        EXERR = SCC
```

{41.}

```
        RETURN
```

{42.}

```
12      IF(HEAP(TCKSAD+TCKDTL+TCKLAC) .NE. HEAP(TXTSAD+TXTDTL+
     *     TXTLNO)) GOTO SCC
```

{44.}

{43.}

```
        TXTACC = TXTSAD+TXTDNL
```

{43.1 see para. 4.3.1.4}

```
        TXTDTL = TXTDNL
        TXTDNL = HEAP(TXTSAD+TXTDTL+TXTFLK)
```

{43.2 see para. 4.3.1.4}

```
        PTLABV = TCKDTL
```

{43.3 Used as back link for last Token Line.}

```
        TCKACC = HEAP(TOKSAD+TCKDTL+TCKFLK)
```

{43.4 see para. 4.3.2.4}

```
10      CONTINUE
        {45.}
        IF(.NOT. ERRFLG) RETURN    {49.}
        {46.}
        CALL SERRCR
        {47.}
9CC     EXERR= 1
        {48.}
        RETURN
        {50}
        END
```

```
      SUBROUTINE LNSCAN(ERRFLG,LINUSE,TXTDTL,SYMCT,*)
```

{1.}

```
      IMPLICIT INTEGER(A-Z)
      REAL RVALUE
      LOGICAL CLFLAG,ERRFLG,RESCAN,FLAG
      COMMON/SCANER/CTAB(255,3),KWTAB(10,7),CLTAB(10,7),LINST(64),
     *SYMST(15,3),ERRST(64,3),SYM(2),TXTLAD,TOKLAD,PINDEX,ERROR,
     *CLFLAG,PRCBAS,ERRCT,RVALUE(15),NEXT
      COMMON /HEAP/NHEAP,HEAP(4CC)
      COMMON/HPCOFF/HPTAG,HPSPAC,HPSIZ,HPLAD,HPOBJ
      COMMON/PRCOFF/PRCNLN,PRCNAM,PRCTXT,PRCTOK,PRCSYM,PRCCOD,
     1              PRCSTA ,PRCLEN
      COMMON/TOKOFF/TOKTOP,TOKBOT,TOKNLN,TOKLNO,TOKFLK,TOKNLM
      COMMON/TXTOFF/TXTTOP,TXTBOT,TXTNLN,TXTLNO,TXTFLK,TXTBLK,
     *TXTTOK,TXTNST,TXTLLEN,TXTLIN,TXTSTA
      DATA PARN /")"/
      CHARNU=0
      ERCODE = 0
      SYMCT = C
      CLFLAG = .FALSE.
```

{2.}

```
      TXTSAD= HEAP(TXTLAD)
      TXTLST= TXTSAD + TXTDTL
      TCHAR=HEAP(TXTLST+TXTNLN)
```

{3.}

```
      DO 10 I= 1,TCHAR
```

{4.}

```
        CALL GETCHR (HEAP(TXTLST+TXTLIN),I,ARG3)
```

{5.}

```
       LINST(I)= ARG3
```

{6.}

```
10    CONTINUE
```

{7.}

```
      IF (LINST(1) .EQ. PARN) CLFLAG = .TRUE.
```

{8.}

{9.}

```
      DO 2C J= 1,ICHAR
```

{10.}

```
      IF(J .LE. CHARNU) GO TO 20
```

{11.}

{12.}

```
       CHAR= LINST (J)
       PT= CTAB(CHAR,1)
```

{13.}

```
       GO TO (2C,3C,40),PT
```

{*14. PT = 1, CHAR is a blank, GO TO 20}

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

{15.}

```
       GO TO 20
```

{16.}

```
40     NFC= J
```

{17.}

{18.}

```
       CHARNU=J
```

{19.}

```
      CALL FORM(CHARNU,SYML,ICHAR,ERCODE,INDEX,CLASS,SYMCODE,
     *  NFC,& 55)
```

{20.}

```
      GOTO (70,80,90), SYMCODE
```

{23.}

```
7C      CALL TABLE(SYML,CLASS,INDEX,&90)
```

{24.}

```
        CLASS=2
```

{25.}

```
8C        CALL SYMTAB (PINDEX,SYML,SYM,INDEX,ERRCR)
```

{26.}

```
          IF(ERRCR .EQ. 1) RETURN 1
```

{27.}

{28.}

```
9C        SYMCT = SYMCT+1
          SYMST(SYMCT,1) = CLASS
          SYMST(SYMCT,2) = INDEX
          SYMST(SYMCT,3) = NFC
```

{29. & 30.}

{31.}

```
2C      CONTINUE

        GC TC 100

55      ERRCT= ERRCT+ 1
```

{21. & RETURN From SUBROUTINE FORM.}

```
        TXTSAC= HEAP(TXTLAC)
        ERRST(ERRCT,1)=HEAP(TXTSAC+TXTCTL+TXTLNC)
        ERRST(ERRCT,2)= NFC
        ERRST(ERRCT,3)= ERCODE
        ERRFLG= .TRUE.
        SYMCT = C
```

{22.}

```
1CC     LINUSE = (SYMCT+1)*3
        RETURN
```

{32.}

```
        END
```

```
      SUBROUTINE FORM (CHARNU,SYML,TCHAR,ERCODE,INDEX,CLASS,
     *                 CODE,NFC,*)
```

{1.}

```
       IMPLICIT INTEGER(A-Z)
      REAL RVALUE
      LOGICAL CLFLAG,ERRFLG,RESCAN,FLAG
      COMMON/SCANER/CTAB(255,3),KWTAB(10,7),CLTAB(10,7),LINST(64),
     *SYMST(15,3),ERRST(64,3),SYM(2),TXTLAD,TOKLAD,PINDEX,ERROR,
     *CLFLAG,PRGBAS,ERRCT,RVALUE(15),NEXT
      COMMON /HEAP/NHEAP,HEAP(400)
      CODE=0
      CLASS = 0
      SYML=1
      INDEX=0
      VALUE=0
         K=0
           SYM(1)=0
           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.TCHAR) J=TCHAR
```

{2.2 Text for last character is LINST for this line.}

```
      PT= CTAB(CHAR,2)
```

{3.}

```
      GO TO (20,30,40,50) ,PT

20    CODE = 1
```

{4.}

{5.}

```
      DO 25 I=J,TCHAR
```

{6.}

```
      CHAR=LINST(I)
      IF(CTAB(CHAR,1).LT.3 .OR. CTAB(CHAR,2).GT.2) GOTO 10 {7.}
        {14.}
      SYML=SYML+1
      CHARNU=CHARNU+1
25    CONTINUE
        {14.1 Last character in line was part of Symbol.}
      GO TO 10
30    CLASS=3   {15.}
      CODE=3
      DO 35 I=J,TCHAR
        {16.}
      CHAR=LINST(I)
      IF(CTAB(CHAR,2) .NE. 2 .OR. CHAR .NE. PERIOD) GOTO 60 {17.}
        {19.}
      IF(CHAR .EQ. PERIOD) CLASS= 4   {21.}
        {20.}
      SYML=SYML+1
      CHARNU=CHARNU+1
        {19.1 Counters are increased.}
35    CONTINUE
        {17.1 Last character in Line was part of Symbol.}
60    CALL NUMPAC(NFC,SYML,VALUE,ERCODE,&90)
      INDEX=VALUE
        {18.}
      RETURN
```

```
4C    CLASS=8   {22.}
        {23.}
      CODE = 2
        {24.}
      DO 45 I=J,TCHAR
        {25.}
        CHAR=LINST(I)
        IF(CTAB(CHAR,1) .LT. 3 .OR. CTAB(CHAR,2) .GT. 2) GOTO 10
                                                          {26.}
          {32.}
        SYML=SYML+1
        CHARNL=CHARNU+1

45    CONTINUE
        {26.1 Last character in Line was part of Symbol.}
      GO TO 1C
5C    CLASS=5   {33.}
        {34.}

      CODE=3
        {35.}
      SYML=SYML+1
      CHARNL= CHARNL+ 1
      J=J+1
      DC 55 I=J,TCHAR
        {36.}
        CHAR=LINST(I)
        SYML=SYML+1
        CHARNL=CHARNL+1
        IF (CTAB(CHAR,2) .EQ.4) GO TO 7C    {37.}
          {40.}
55    CONTINUE
```

```
      ERRCODE=2
```
{41.}
```
90    RETURN 1
70    INDEX=SYML
```
{38.}
```
      RETURN
```
{39.}
```
10    IF(SYML .GT.8) GO TO 80
```
{8. & 27.}
{10.} {29.}
```
      DO 1 L=NEC,CHARNU
```
{11.}
```
        CHAR=LINST(L)
        K=K+1
        CALL PUTCHAR (SYM(1),K,CHAR)
```
{12.} {30.}
```
1     CONTINUE
```
{13.}
```
      RETURN
80    ERRCODE=1
```
{9.}
```
      RETURN 1
```
{42.}
```
      END
```

```
      SUBROUTINE TABLE(SYML,CLASS,INDEX,*)
```
{1.}

```
      IMPLICIT INTEGER(A-Z)
      REAL RVALUE
      LOGICAL CLFLAG,ERRFLG,RESCAN,FLAG
      COMMON/SCANER/CTAB(255,3),KWTAB(10,7),CLTAB(10,7),LINST(64),
     *SYMST(15,3),ERRST(64,3),SYM(2),TXTLAD,TOKLAD,PINDEX,IRROR,
     *CLFLAG,PRGBAS,ERRCT,RVALUE(15),NEXT
      COMMON /HEAP/NHEAP,HEAP(400)
      INDEX=0
      COL=SYML-1
```
{2.}

```
      IF(CCL.EQ.0) RETURN
```
{3.}

{4.}

```
      IF(CLFLAG) GO TO 10
```
{5.}

{11.}

```
      DO 20 I=1,10
         IF(KWTAB(I,CCL).EQ.SYM(1)) GO TO 40
```
{12.}

```
20    CONTINUE
```
{16.}

```
      RETURN
```
{17.1 Symbol is an identifier.}

```
10    DO 30 I=1,10
```
{5.}

```
      IF(CLTAB(I,CCL).EQ.SYM(1)) GO TO 40
```
{6.}

```
30    CONTINUE
```
{10.}

```
      RETURN     {17.1  Symbol is an identifier.}
40    INDEX=(I*10) + COL
        {13.}
      CLASS=1
        {14.}

      RETURN 1
        {17.2  Control returned to lable 90 in LNSCAN.}

      END
```

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

{1.}

```
       IMPLICIT INTEGER(A-Z)
      REAL RNUM,RVALUE
      LOGICAL CLFLAG,ERRFLG,RESCAN,FLAG
      COMMON/SCANER/CTAB(255,3),KWTAB(10,7),CLTAB(10,7),LINST(64),
     *SYMST(15,3),ERRST(64,3),SYM(2),TXTLAD,TOKLAD,PINDEX,ERROR,
     *CLFLAG,PRCBAS,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.}

```
      IQ= 0
      DATA PERIOD/'.'/
```

{2.1}

```
      IF (LENGTH .LE. 8)  GOTO 5
```

{3.}

```
      ERCODE = 1
      RETURN 1
```

{4.}

```
5     DO 10 I = NFC,STOP
```

{5.}

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

6.

{9.}

```
20       CONTINUE
```

{10.}

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

{11.}

```
            NT = 1
            IQ = 1
            GOTO  10

1           IF(NT .EQ. 1) IQ = IQ+1        {7.}

            INUM= J-1
            NUM= NUM*10+INUM
            {8.}
10      CONTINUE
            {13.}
        IF(NT.EQ.1) GOTO 30    {14.}
            {15.}
        VALUE = NUM

        RETURN
            {16.}
30      RNUM = NUM
            {14.1}
        RNUM = RNUM*.1**IQ
            {14.2}
        RVALUE(NEXT)= RNUM
            {14.3}
        VALUE= NEXT
            {14.4}
        NEXT= NEXT+1
            {14.5}
        RETURN
            {16.}
40      ERCODE = 3
            {12.}
        RETURN 1

        END
```

```
      SUBROUTINE SERROR
```
{1.}
```
      IMPLICIT INTEGER(A-Z)
      REAL RVALUE
      LOGICAL CLFLAG,ERRFLG,RESCAN,FLAG
      COMMON /HEAP/NHEAP,HEAP(400)
      COMMON/SCANER/CTAB(255,3),KWTAB(10,7),CLTAB(10,7),LINST(64),
     *SYMST(15,3),ERRST(64,3),SYM(2),TXTLAD,TOKLAD,PINDEX,ERRCR,
     *CLFLAG,PRCBAS,ERRCT,RVALUE(15),NEXT


      DIMENSION STRTMC(17), STRNCQ(26), STRILC(17)
      DATA STRTMC/'S','Y','N',' ','T','L','O',' ','M','A','N','Y',' ',
     *'C','H','A','R'/ ,STRNCQ/'N','C',' ','C','L','O','S','I','N','G',
     *' ','Q','U','O','T','E',' ','O','N',' ','S','T','R','I','N','G'/,
     *STRILC/'I','L','L','E','G','A','L',' ','C','H','A','R','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),PT

20    N= 21
```
{5.}
{6.}
```
        CALL ERRPRINT (PINDEX,LINE,CHAR,N,STRTMC,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 LINFIN(TXTDTL,RESCAN,TOKDTL,*)
```

{1.}

```
      IMPLICIT INTEGER(A-Z)
      REAL RVALUE
      LOGICAL CLFLAG,ERRFLG,RESCAN,FLAG
      COMMON /HEAP/NHEAP,HEAP(400)
      COMMON/SCANER/CTAB(255,3),KWTAB(10,7),CLTAB(10,7),LINST(64),
     *SYMST(15,3),ERRST(64,3),SYM(2),TXTLAD,TOKLAD,PINDEX,ERROR,
     *CLFLAG,PROBAS,ERRCT,RVALUE(15),NEXT
      COMMON /HPOFF/HPTAG,HPSPAC,HPSIZ,HPLAD,HPOBJ
      COMMON/TOKOFF/TOKTOP,TOKBOT,TOKNLN,TOKLNO,TOKFLK,TOKNUM
      COMMON/TXTOFF/TXTTOP,TXTBOT,TXTNLN,TXTLNO,TXTFLK,TXTPLK,
     *TXTTOK,TXTNST,TXTLLEN,TXTLIN,TXTSTA
      COMMON/PROOFF/PRONLN,PRONAM,PROTXT,PROTOK,PROSYM,PROCOD,
     1              PROSTA ,PROLEN
```

```
      TOKLLN = 0
      LASTLN = 3
```

{2.}

```
      TXTSAD=HEAP(TXTLAD)
```

{3.}

```
      TOKSAD= HEAP(TOKLAD)
```

{4.}

```
      TOKTLN=HEAP(TOKSAD+TOKNLN)
```

{5.}

```
      LNDISP=HEAP(TOKSAD+TOKTOP)
```

{6.}

```
      TXTLIN=HEAP(TXTSAD+TXTDTL+TXTLNO)
```

{7.}

```
      DO 10 I=1,TOKTLN
         IF(TXTLIN .GT.TOKLLN .AND.TXTLIN .LE. HEAP(TOKSAD+LNDISP+
     *    TOKLNO )) GOTO 20
```

{8.}

{14.}

```
      TOKLLN = HEAP(TOKSAD+LNDISP+TCKLNC)
      LASTLN = LNCISP
      LNCISP = HEAP(TOKSAC+LASTLN+TOKFLK)
```

{15.}

```
1C    CONTINUE
```

{16.}

```
      RESCAN= .FALSE.

      RETURN
```

{17.}

```
2C    IF(TXTLIN .EQ. HEAP(TOKSAD+LNDISP+TCKLNC)) GOTO 30
```

{9.}

{12.}

```
      HEAP(TCKSAD+TCKCTL+TOKFLK )=HEAP(TCKSAC+LASTLN+TCKFLK)
      HEAP(TOKSAD+TOKCTL+TOKFLK) = HEAP(TOKSAD+LASTLN+TOKFLK)
      HEAP(TCKSAD+LASTLN+TOKFLK) = TCKCTL
      HEAP(TOKSAD+TOKNLN) =HEAP(TOKSAD+TOKNLN) +1
```

{13.}

```
      RETURN 1
```

{17.}

```
3C    HEAP(TOKSAD+TOKCTL+TOKFLK) =HEAP(TCKSAC+LNDISP+TCKFLK)
```

{10.}

```
      HEAP(TCKSAD+LASTLN+TCKFLK) = TCKCTL
```

{11.}

```
      RETURN 1
```

{17.}

```
      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 fulfillmant 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.