# ILLEGIBLE DOCUMENT

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

THIS IS THE BEST COPY AVAILABLE

# ILLEGIBLE

# THE FOLLOWING DOCUMENT (S) IS ILLEGIBLE DUE TO THE PRINTING ON THE ORIGINAL BEING CUT OFF

# ILLEGIBLE

# THIS BOOK CONTAINS NUMEROUS PAGE NUMBERS THAT ARE ILLEGIBLE

# THIS IS AS RECEIVED FROM THE CUSTOMER

# INTERP-700:
# DOCUMENTATION OF A STUDENT DESIGNED INTERACTIVE INTERPRETER

by

MICHAEL FRANK MITPIOME

B.B.A., St Bernadine of Siena College, 1963

---

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

## INTERP-700

# CHAPTER 1

# INTRODUCTION

1.1 GENERAL: This report describes an interpreter which will be referred to as INTERP-700. It examines and illustrates features of a high level programming language, much like APL, which is suited to a spectrum of users ranging from novice to experienced programmer. Emphasis is on documentation of INTERP-700 as it exists today, but reference is made to future enhancements which will increase the test and debug capabilities of the interpreter. Emphasis is on two areas - language documentation; and program documentation. It should be noted, however, that INTERP-700 is not a completely operational system and is still in the development stage. In spite of this, there are certain features which are operational in prototype form.

1.2 MOTIVATION: As programming systems and programming languages evolve, there is an increasing shift towards the utilization of languages by application programmers outside of computer science. This trend is illustrated by the development of programming languages such as BASIC, APL, and 'English like' COBOL. There are several problems associated with this trend. Most languages are so rigid, that it is impractical to introduce new features without major revisions. In addition, languages are sufficiently difficuly to learn that there is a reluctance to abandon one language for another. Languages are not sufficiently oriented toward ease-of-use, and there is usually an excessive 'learning overhead' barrier even between related languages. The

purpose of INTERP-700 is to provide a language system which is both flexible and portable, with modifiable core language for novice level programming, with expressive features for the experienced programmer, and with extensibility features for the expert. INTERP-700 is also designed to support a wide range of dynamic debugging aids. In addition to debugging, INTERP-700 should be able to deal with other aids for program development and verification.

1.3 REPORT ORGANIZATION: This report consists of 7 chapters, each examining a different facet of the interpreter. Chapter 2 looks at the purposes of INTERP-700, both present and future, and touches on how this interpreter differs from others available. Chapter 3 discusses the language documentation, which includes a users guide, and the three language subsets of INTERP-700 - the command language, the test editor language, and the programming language. Chapter 4 discusses the syntax rules of the language and includes a section on how parameter passing is accomplished. Chapter 5 discusses the semantic rules and gives both the data structures used, and the language restrictions. Chapter 6 looks at the current status of INTERP-700, as well as future enhancements envisioned. Chapter 7 discusses each major module of the interpreter and describes the purposes of each.

1.4 BACKGROUND OF INTERP-700: The concept for INTERP-700 was initially proposed by Hankley and Wallentine. Their contention was that there should be a programming language designed for a spectrum of users which emphasized three areas - the orientation

of language design tc user convience and availability; the efficiencies which can be achieved with information provided by the user and/or the system; and the current software engineering technology used to support modular programming and to achieve program portability. The concepts espoused were initially presented as a class project in the summer of '74 for course CS-700 (Translater Design). The results of their efforts were given to the summer '75 CS-700 course. This course was divided into student groups tc develcp INTERP-700. Each student group independently designed a part of the interpreter using a modular programming approach. Although general guidelines were provided by the instructor, informal ccordination was the responsibility of student groups whose modules were to interface. Syntax and semantic rules were also developed independently as well as various limits of language implementation. In many cases, ccordination of effort was done on a very superficial scale. Testing of modules was to be accomplished using a predefined data base provided by the instructcr. Continued efforts will be made by follow-on courses.

# CHAPTER 2

## PURPOSE OF INTERP-700

2.1.1 TRANSPORTABILITY: There is a need in today's computer technology for a conversational interpreter which is not tied to, or designed specifically for, a particular type system. INTERP-700 is intended to be a portable system. Using FORTRAN IV as its host language, it should have the capability of being easily implemented on most computer systems with very little programming effort required for compatibility.

2.1.2 MODULARITY: The trend today is toward developing software which is modular in design, making revisions and enhancements a relatively simple task, rather than a major redesign effort. INTERP-700 is completely modular in design which, unlike APL, or BASIC, allows systems programmers to tailor it to unique user requirements without altering its basic capabilities.

2.1.3 VIRTUAL MEMORY APPLICATIONS: INTERP-700 uses an address translation table to locate data within memory. In a virtual machine the various data structures addressed by the translation table can reside in on-line storage devices until needed by the user, freeing large areas of core memory. This can be done without changing its basic capabilities and would facilitate the use of mini-computers to test-prove procedures without incurring the costs associated with test and debug of programs on large main frames.

2.1.3 DYNAMIC TYPE CHECKING: INTERP-700's usage of 'tag architecture' allows complete flexibility in defining data structures. Dynamic checking of data types eliminates the probability of fatal errors since type checking is done during execution rather than at compile time. This allows the user to make immediate corrections and to continue execution rather than search program parts, make appropriate corrections, recompile, and reexecute.

2.1.5 CONVERSATIONAL: INTERP-700 is designed as an interactive programming tool, which allows the user to test and debug program logic and code at the symbolic level.

2.2 FUTURE ENHANCEMENTS:

2.2.1 EXPERIMENTAL RESEARCH: In a university type environment, or software development house, this type interpreter could provide a relatively inexpensive and effective tool for experimenting with compiler/interpreter design alternatives.

2.2.2 CODE GENERATION: The interpreter could be extended to generate output code, allowing for a conversion from a high level language to machine executable code.

Intentionally left blank

# CHAPTER 3

## LANGUAGE DOCUMENTATION

3.1 USERS GUIDE: Like APL, INTERP-700 is designed for interactive use, but with the addition of control structures which allows the operator to flexibly influence the flow of processing. The language allows extensive interaction with program execution, program variables, execution (activation) records, and debugging aids. It gives the user the capability of being able to do more than merely run a program at a private terminal. The user can also stop a program, display variables, tables, and/or data structures, identify errors, dynamically change the program, and continue execution. If an error is discovered, processing automatically stops, the exact location and type of error is noted, corrected, and processing again continues, either from the same location, or from a prior location, depending upon the error and user desires. To use INTERP-700, the operator must become familiar with the three language subsets used: the command language subset, the programming language subset itself, and the text editor language subset.

```
**************************
*                        *
*   TABLES 1-6 HERE      *
*                        *
**************************
```

3.2 COMMAND LANGUAGE: All command lines begin with the character ')' followed by an appropriate keyword described in tables 1 - 6. The keyword specifies the particular action the

user requires, or provides the user the ability to accomplish various tasks such as: place an object on disk for temporary or permanent storage; retrieve an object from a designated user's disk space, erase an object from his own disk space; list the names of all objects on his disk space; or allow the user to specify the name of his working space. Each table contains both commands which are presently available, and which should be available in the future.

3.3 TEXT EDITOR LANGUAGE: The editor programs which support INTERP-70C provides the user the capability of creating and modifying a procedure from the terminal. With editor, the user can append lines one after another in sequence, insert lines between lines already entered, delete lines, resequence the line numbers into increments of 100 after insertions and deletions have been made, and list either the new procedure or any single line. Editor is called by entering the command word )EDIT followed by the procedure name. When the edit mode is initially entered, the editor will print out either <100> if the procedure is being entered for the first time, or <::> if the procedure already exists.

3.3.1 ENTERING A NEW PROCEDURE - After receiving the return code <100>, the user types in his line of text followed by a carriage return. The editor will respond with the next line number, in this case <200>, and the user will enter another line of text. This procedure is continued until the entire procedure is entered. Each line is entered (appended) at the line number

printed out by the editor.

3.3.2 LEAVING EDITOR - To leave the editor, the user enters the edit command <Q after the last return code.

3.3.3 MODIFICATIONS - Modifying the procedure can be accomplished using the text editor commands in table 7.

```
*******************************
*                             *
*     TABLE 7  HERE           *
*                             *
*******************************

*******************************
*                             *
*        FIGURE 1 HERE        *
*                             *
*******************************
```

3.4 PROGRAMMING LANGUAGE: The programming language used by INTERP-700 consists of two parts: control grammar, and expression grammar. The expression grammar is very similar to APL. The language is designed for interactive usage and is generally constructed as illustrated in figure 1. Each procedure consists of a header and a body. Keywords PROC...ENDPROC delimit the procedure. The header is comprised of a keyword PROC and an id, followed by an optional argument(s), followed by declarations. The body of the procedure consists of one or more statements, and a keyword ENDPROC. Additional features such as scope rules, rules for parameter passing, data structures, and error messages are contained is subsequent paragraphs, chapters, or appendices.

**********
*        *
*  TABLE 8 HERE  *
*        *
**********

3.4.1 CONTROL GRAMMAR - The various statement forms of the control grammar are contained in table 8. Metasymbols are defined later.

3.4.2 EXPRESSION GRAMMAR - E expressions can be either arithmetic or logical in nature. Precedence for operators within expressions is established by the use of paranthesis. Unlike other languages such as FORTRAN, INTERP-700 expressions are evaluated from right to left with no distinction made between operations.* Specific examples of subexpressions in INTERP-700 are contained in table 9. Type and attribute checking and conversion is handled similar to APL. Expressions can be arbitrarily long except that they must be written all within a single physical line.

**********
*        *
*  FIGURE 2 HERE  *
*        *
*  FIGURE 3 HERE  *
*        *
**********

---

* For the first experimental implementation, operators are executed in a right-to-left order, but subexpression (each within their own parenthesis, including indexing expressions) are evaluated starting with the leftmost subexpression. That is, a left-to-right scan is used, but with right-to-left precedence of operators with subexpressions.

3.4.3 OPERATORS - Operators may be of type scalar, boolean, or character_string. They may also be defined by the user if preceeded by a '$' (ex: $add). Specific operators defined by the system are illustrated in figure 2. Binary operators, by type and results, are illustrated in figure 3.

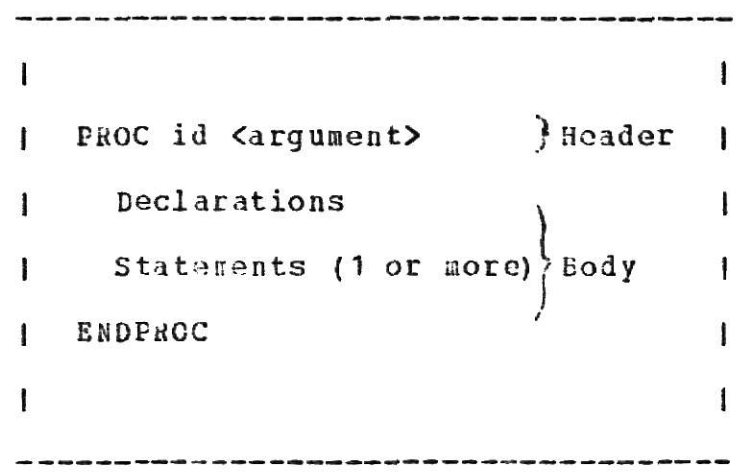



figure 1

procedure format

| Symbol | Definition | Argument type |
|---|---|---|
| <-- | Assignment | All types |
| + | Addition | Number, number array |
| - | Subtraction | Number, number array |
| = | Equal | All types |
| ≤ | Less or equal | Any types |
| ≥ | Greater or equal | Any types |
| ∧ | AND | Boolean, Bool array |
| ∨ | OR | Boolean, Bool array |
| '\|' | Concatenation | Character string |

figure 2

operator specifications

| Operators | Type operation | Result type |
|---|---|---|
| <-- | <any1> <op> <any2> | <any2> |
| +,-,<-- | <scalar> <op> <scalar> | <scalar>* |
| <,>,= | <scalar> <op> <scalar> | <boolean> |
| ∧,∨,= | <boolean> <op> <boolean> | <boolean> |
| $CAT | <string> <op> <string> | <string> |
| ⟨,⟩,= | <char> <op> <char> | <boolean> |
| +,- | <array> <op> <array> | <array>** |
| +,- | <scalar> <op> <array> | <array> |
| +,- | <array> <op> <scarar> | <array> |

* Mixed mode operations on scalars that result in scalars will always result in real scalers

** Arrays must be conformed.

figure 3

sample binary ops by type and result type

Table 1

Command Language Logging Commands

| | | |
|---|---|---|
| )CN | - | Used as the first command of the work session once communication has been established between the terminal and the interpreter. If not used as the first command, the user will be sc notified. |
| )OFF | - | Used as the last command of the work session prior to LOGOFF. If user attempts to log off the terminal without this command being used first, the LOG command will be treated as a part of the interpreter language and an error message will be returned indicating improper command. |
| )SUSPEND** | - | Allows user to end work session and store data in the active workspace in order to continue processing at a later time without starting from the beginning. |
| )RESUME** | - | Used in conjunction with )SUSPEND. Allows user to continue a work session which had been previously suspended. Work session will continue from the point processing was suspended. |

---

** future enhancement

Table 2

Command Language Execution Commands

)RUN <name> - Specities that a particular process is to be executed. <name> can be either a function or procedure which has been defined by the user.

)EDIT <name> - Used to invoke the text editor for the function defined as <name>. With this command the user can enter a new procedure/function, or affect some change to an existing function or prccedure using the language defined in the text editor language subset.

)PARSE <name> - Used to determine the syntax validity or to determine whether semantic actions are required. As abcve, <name> is a procedure or function. Causes INTERP-700 to build code for <name>. If code had previously been generated, it will be over written by new code.

Table 3

Command Language Library Commands

)LIB - Used to print the directory of all saved objects such as the names of the work spaces, globals, and procedures/functions.

)COPY <namelist> - The object defined by <namelist> will be copied into the current workspace. Command does not differentiate between procedures, functions, or objects.

)DROP <namelist> - Erases, or deletes, the object defined by <name> which has been saved. As with )COPY, objects, functions, and procedures are handled identically. (eg., )DROP Object will erase an existing file called 'Object'.

)LOAD <name> - Future applications will also allow the work space ID <WSID> to be loaded, ie., to replace the current workspace.

)SAVE <namelist> - Used to save an object which has been created.

)WSID <name> - Used to display the current work space name. <name> is an optional parameter. Future enhancements will allow the workspace id to be altered by the value defined as <name>.

Table 4

Command Language Display Commands

)FNS - Allows the user to print a list of all functions and procedures previously defined, and which exist in the workspace.

)VARS - Allows the user to print a list of all globals previously defined, and which exist in the workspace.

)HELP** - Provides the user both syntatic and semantic information about structures within the language for which the user needs assistance. May be followed by an optional parameter <arg>. If <arg> is not used, the user will be provided a listing of all language functions available. If <arg> is used, help will be provided for the argument specified.

)LIST <namelist>** - Provides a listing of the named procedures which exists in the workspace.

)VALUES <namelist>** - Provides the user with a list of global parameters with their currently assigned values.

---

** Future enhancement

Table 5

Command Language Environment Control Commands

| | |
|---|---|
| )POP | - Allows the user to delete the top activation record from the stack. Used primarily after an error has been detected and the user wishes to resume processing at a previous point to determine the validity of the correction. |
| )STACK | - Allows the user to display the names of activation records on the stack beginning with the last activation record put in the stack, sequencing through to the first activation record. |
| )CLRSTK | - Allows the user to reset the stack pointers to zero, deleting all activation records with respective arguments from the stack. |
| )CLEAR | - Allows the user to reset the stack and heap pointers to their respective initial values. |
| )DIGITS <number>** | - Sets the maximum number of digits which is printed, or changes the number of |

significant digits displayed to the value defined as <number>.

)WIDTH <number>** - Sets the width (number of characters) on the printed page. Usually 60 for narrow paper, and 120 for wide paper.

)LINE <number>** - Sets the maximum number of lines to be listed on the printed page.

)CHAR** - Specifies the character representation to be ASCI, APL, or EBCDIC.

)HENCE** - Allows the user to change the name of standard functions.

---

**Future enhancement

Table 6

Command Language Debug Commands

| | |
|---|---|
| )TRACE <func> VARS** | - Allows the user to print the value of a global parameter each time it is used with the specified function <func>. |
| )TRACE <func> LINES** | - Allows the user to trace the progress of a function <func> at specific lines of code within a procedure. |
| )NOTRACE <func>** | - Used to turn off the traces set by the above TRACE commands. If <funct> is not specified, all tracing is turned off. |
| )BREAKPT <func> VARS** | - Similar to the TRACE except that when a parameter is referenced, control is returned to the user. Provides the user a debug capability to exercise various options such as listing values of variables. |
| )BREAKPT<func>LINES** | - Similar to the BREAKPT<func>LINES command except that control is returned to the user when specified lines are encountered. |

---

**Future enhancement

)NOBREAKPT <func>** - Used to turn off the BREAKPT options listed above. If <funct> is not specified, all breakpoints are turned off.

Table 7

Text Editor - Procedure Modifiers

| | |
|---|---|
| ⟨I line_number text_string | - Inserts a line. The line number may be any integer number less than the largest line number which was previously entered, but may not be the same as any line previously entered. Editor will return a line number with a value 100 greater than the last line in the procedure. text_string is the data to be inserted. |
| ⟨D line_number | - Used to delete the line number specified. When line is deleted, editor will respond with ⟨:⟩. |
| ⟨R | - Used to resequence the line numbers into multiples of 100. When executed, editor will respond with ⟨:⟩. |
| ⟨A | - Used to append a line when the next line number is not known. Editor will respond with the next line number in the procedure (eg.,⟨700⟩). User then enters the line as he did when entering a new procedure. |
| ⟨line_number text_string | - Used to append a line when the next line number is known. User enters the line as he did when entering a new procedure |

<L - Used to list the entire procedure. When listing is complete, editor will print <::>.

<L line_number - Used to list the line specified. When the listing is complete, editor will print <::>.

<Q - Terminate editing.

Table 8

Statement Forms

(Part of the Flow_of_Control Grammar)

| | |
|---|---|
| L: statement * | - L is a statement label delimited by and :. |
| E | - An arithmetic expression which will be evaluated by the bottom-up parser. Specific definitions of E are contained in Table 9. |
| GO TO L | - Transfers control of processing to the address specified by the value L. |
| CALL id (arglist) | - Used similar to a subroutine call in FORTRAN. (arglist) is an optional parameter and consists of one or more identifiers, integers, or expressions. If more than one argument is used they must be separated by commas., ex., CALL A(X,Y), or just CALL A. |
| If E then statelist FI | - Specifies that the logical expression E is to be evaluated. If true, THEN perform the sequence of statements defined as statelist. If false, end IF statement (FI). |
| If E THEN statelist ELSE statelist FI | - Similar to IF above except that if false ELSE statement is performed. FI delimits end of statement. |

---

* - Lower case used to represent metasymbols, except for E and L.

WHILE E DO statelist
ENDWHILE - Similar to PL/1. Specifies iterative processing as long as E is true. When E is false ENDWHILE.

RETURN - Returns control to the point of invocation. Similar to use of RETURN in FORTRAN subroutines.

Table 9

Sub-expression Examples

(Part of the Expression Grammar)

| Expression Definitions | Example |
|---|---|
| Simple identifier | A, AB |
| Value | 'string', vector, 1, 7.3, true |
| Expression in parentheses | (AB) |
| Unary operator expression | +AB, *7.3, ¬true |
| Binary operator expression | AB + 7.3, true = false |
| Function reference | AB(3,JIM) |
| Indexed expression | AB[7;3], (A+B)[3] |

# CHAPTER 4

## SYNTAX RULES

4.1 GENERAL: This section assumes that the reader is familar with BNF* notation, and is used herein to define the legal sentential formats recognized as the grammar of the language. The INTERP-700 grammar consists of both control sentences and expression statements. The general, or abstract, forms of the language composition are contained in table 10.

**********
*
*  TABLE 10 HERE
*
**********

4.2 PARAMETER PASSING: Initially discussed as <scope> in table 10. All programming languages, in one manner or another, are concerned with passing parameters to various called procedures, or subroutines. The general format for INTERP-700 contains the keyword PROC, followed by an identifier <id>, followed by the list of parameters to be passed. Parameters are declared immediately following the PROC statement (see figure 1). IN parameters are values passed by the calling program into the called procedure. OUT parameters are values passed back to the calling program by the called procedure. ACCESS parameters are used to pass values in and out of the called procedure. If no parameters are declared, the default will be for LOCAL parameters to be assumed.

4.3 LOCAL VARIABLES: LOCAL variables are those used strictly within that procedure and cannot be passed. In addition to

---

* Bacnus Naur Form

these, there are other methods for accessing variables to be used within procedures. A variable can be declared as a GLOBAL. Although not yet implemented, another method will exist for passing parameter values. If a variable is declared as EXPORT in a procedure, it can then be treated as a variable accessible to any procedure which is called (directly or indirectly) from the procedure which declared the variable as EXPORT. In order to access the variable declared as EXPORT, the procedure using the same variable must declare it as EXTERNAL. This initiates a search back through the calling procedure chain until the variable is located and value assigned.

Table 10

INTERP-700 Syntax Grammar

```
<procdef>     ::=  <header><body>
<header>      ::=  PROC<id><namelist>* ; <opt>***
<namelist>    ::=  (<id>**)
<body>        ::=  <statelist>;  ENDPROC
<opt>         ::=  <scope> <id>** ;
<scope>       ::=  IN|OUT|GLOBAL|ACCESS|EXTERNAL|EXPORT|LOCAL
<statelist>   ::=  <state> <tail>*
<tail>        ::=  ; <state>
<state>       ::=  <>L:>* <st>
<st>          ::=  <E>|Statement forms as illustrated in Table 8
<E>           ::=  Expressions as illustrated in Table 9.
<>L:>         ::=  ><id>:
```

NOTE: Separating semicolons (;) may be deleted at the end of a physical line. ie., ; eol may be replaced by eol.

---

* - Can be one or null.
** - one or more, separated by commas (ex., A,A.....,A)
*** - Zero or more occurances

# CHAPTER 5

# SEMANTIC RULES

5.1 GENERAL: This chapter looks at the generalized data structures used within INTERP-700, and discusses the various language restrictions. Each data structure discussed is illustrated in tablular form at the end of the chapter as figures 4 through 10.

5.2 GENERALIZED DATA STRUCTURES: All source programs, tokens, generated code, symbol tables, and all stored arrays and strings can be found in a data structure known as the HEAP. Execution data is contained in a data structure known as the STACK. The user can obtain relevent information pertaining to stored procedures by examining each of these structures.

5.2.1 HEAP - The HEAP is the main storage area for INTERP-700, and is accessed by all subroutines. The various components of the HEAP are listed below.

```
**********************
*                    *
*   FIGURE 4 HERE    *
*                    *
**********************
```

5.2.1.1 ADDRESS TRANSLATION TABLE (ADTRAN) - The first data structure within the HEAP (see figure 4). The ADTRAN consists of numerous pointers to the various data areas contained within the heap. The first four words of the ADTRAN contains the ADTRAN head which defines the size of the ADTRAN, and the amount of

space used. The fifth word of the ADTRAN is the index of the word immediately preceeding the procedure_table. The sixth word of the ADTRAN is the index of the word immediately preceeding the global_table. The seventh word is a pointer back to the beginning of the ADTRAN. The last word of the ADTRAN contains either the location of additional free space available, or a -1 to indicate no additional free space exists. The second last word of the ADTRAN points to locations within the ADTRAN which are free. All other words of the ADTRAN are used to point to other data structures within the HEAP.

```
**********************
*                    *
*   FIGURE 5 HERE    *
*                    *
**********************
```

5.2.1.2 PROCEDURE_TABLE(PROC_TAB) - Serves as an index to the various components of procedures defined by the user. The header for the Proc_Tab is similar to the header for all remaining data structures in the HEAP. The first word of the header is a unique tag which identifies the type data structure. The second word indicates the amount of space allocated for that data structure, and the third word indicates how much space has been used. The fourth word of the header is a backward link to the position in the ADTRAN where the particular data structure can be found. This facilitates the changing of pointer values during compaction or reallocation of space. The remainder of the Proc_Tab consists of a variable number of fixed length(PRCLEN) entries. Figure 5 contains the names (with offset codes) of the various index fields.

**********
*        *
* FIGURE 6 HERE *
*        *
**********

5.2.1.3 TOKEN TABLE - One for each procedure or function which has been scanned. Contains the tokens, by line, generated by the scanner for a procedure. Tokens are stored by line corresponding to the source text line number. The first four words of the token table is the heap header, and is similar to that described in para 5.2.1.2. The token table consists of a token table header (TCK), and a variable number of token line header (TOK.LINE) and token line (TCK.LINE.TOKEN) combinations. Figure 6 contains the names (with offset codes) of the various fields within the token table.

**********
*        *
* FIGURE 7 HERE *
*        *
**********

5.2.1.4 TEXT TABLE - One for each procedure or function which has been created using the edit process. Contains the symbolic text generated by the text editor for each procedure. The first four words of the text table is the heap header, and is similar to that described in para 5.2.1.2. The text table consists of a table header (TEXT), and a variable number of text line header (TEXT.HEADER) and text line (TEXT.LINE) combinations. Figure 7 contains the names (with offset codes) of the various fields within the text table.

5.2.1.5 SYMBOL TABLE - One for each procedure or function which

has been scanned. It is a list of all names in the module, with status information for tracing by name, for scope of variables (LOCAL, IN parameters, OUT parameters, names of procedures, global, or access), generated by the parsers. The first four words of the symbol table is the heap header, and is similar to that described in para 5.2.1.2. Figure 8 contains the names (with offset codes) of the various fields within the symbol table.

```
**********************
*                    *
*   FIGURE 8 HERE    *
*                    *
**********************
```

5.2.1.6 GLOBAL TABLE - Contains a list of all global variables which have been assigned values and which have not yet been erased. The data structure for the global table is exactly like that of the symbol table (Figure 8) except for the tag code in the heap header.

```
**********************
*                    *
*   FIGURE 9 HERE    *
*                    *
**********************
```

5.2.1.7 CODE TABLE - One for each procedure or function which has been successfully parsed. Contains the code, by line, generated by INTERP-700 for that procedure. Code is stored by line corresponding to the source text line number. The first four words of the code table is the heap header, and is similar to that described in para 5.2.1.2. The code table consists of a

variable number of code line header (CODE), and code line (CODE.LINE) combinations. Figure 9 contains the names of the various fields within the code table.

5.2.2 STACK - The stack is used principally for execution of code generated by various subroutine modules. The primary record of the stack is the activation record (AR) which consists of the AR header, and a copy of the symbol table for the procedure being executed. The AR keeps track of where execution is taking place, which facilitates identification of the precise location of execution errors. When the AR is built, the symbol table for that procedure is appended to the AR header. The results of operations performed by the operator routines is appended to the stack as temporaries. When the executing procedure calls another procedure, arguments to be used by the new procedure are appended to the stack either as values or as references to the calling AR. Next, the number of args is stacked, and then the AR for the called procedure is pushed on the stack. Figure 10 contains the names (with offset codes) of various fields in the AR. ARSYM is an offset which by-passes the AR header. Access to any field within the stacked symbol table can be achieved by referencing STACK(LAR+ARSYM+SYMNAM+Index).

**********
*        *
* FIGURE 10 HERE *
*        *
**********

5.3 LANGUAGE RESTRICTIONS: The implementation of the scanner module impose restrictions on the syntax of the language. The

language restrictions are listed below.

5.3.1 SYMBOL LENGTH - All symbols are restricted to a total length of eight characters. Longer names are truncated.

5.3.2 KEYWORDS - KEYWORDS are reserved words and they are restricted to those words identified and listed in the keyword table (see appendix D). Only the first four characters are stored in the tables. Words may be added to the table, but will necessitate modification of the tables.

5.3.3 COMMAND LANGUAGE WORDS - Command language words are restricted to those identified and listed in the command language table. Command language words are reserved words in the context of a beginning ')', eg., ') commandword'. (see appendix D). Again, only the first 4 characters are stored. Words may be added to the table, but will require modification of the tables.

5.3.4 COMMAND LANGUAGE LINES - The first character of all command language lines must be a closing parenthesis.

5.3.5 OPERATORS - Operators must be one of those characters identified as an operator in the initialization data for the program, or if previously undefined, a character or series of characters not to exceed seven characters, preceeded by a $.

5.3.6 STRINGS - All strings must be delimited by quotation marks. Input strings may be any length and contain any combination of characters that may be included on <u>one</u> physical program line.

5.3.7 LINE CONTINUATION - Expressions must be contained on a single physical line. Statements of the control grammar may be spaced over several lines. Multiple logical lines may be written on the same physical line separated by ';'.

5.3.8 REAL VALUES - All real values must contain a decimal point. Exponential numbers are of form realESDD, where S is the exponent sign, and DD represents one or two digits. Magnitude of reals is limited by the particular machine implementation of FORTRAN.

5.3.9 LINE LENGTH - A single line of source text may not contain more than a total of 64 characters.

5.3.10 SEPARATIONS - All identifiers, keywords, command language words, values, or strings must be separated by either an operator, separator, a blank, or a string of blanks.

5.3.11 CHARACTERS - Characters are limited to those of the character code table (See appendix D) used to initialize the character table of the scanner module.

5.3.12 EDIT COMMAND WORDS - All commands to the text editor must be preceeded by <.

5.3.13 TEXT EDITOR RESTRICTIONS:

5.3.13.1 No terminal entry is allowed on the same line with <Q, <A, or <R except a carriage return.

5.3.13.2 A space must be present between <I and the line number; and between the line number and text string.

5.3.13.3 A space must be present between <D and the line number.

5.3.13.4 A space must be present between <L and the line number if requesting a single line, or be entirely blank following L if the entire procedure list is requested.

5.3.13.5 If a specific line number is being entered, a space must be present between <line_number and text_string.

Figure 4

ADTRAN Data Structure with Free Block Structure

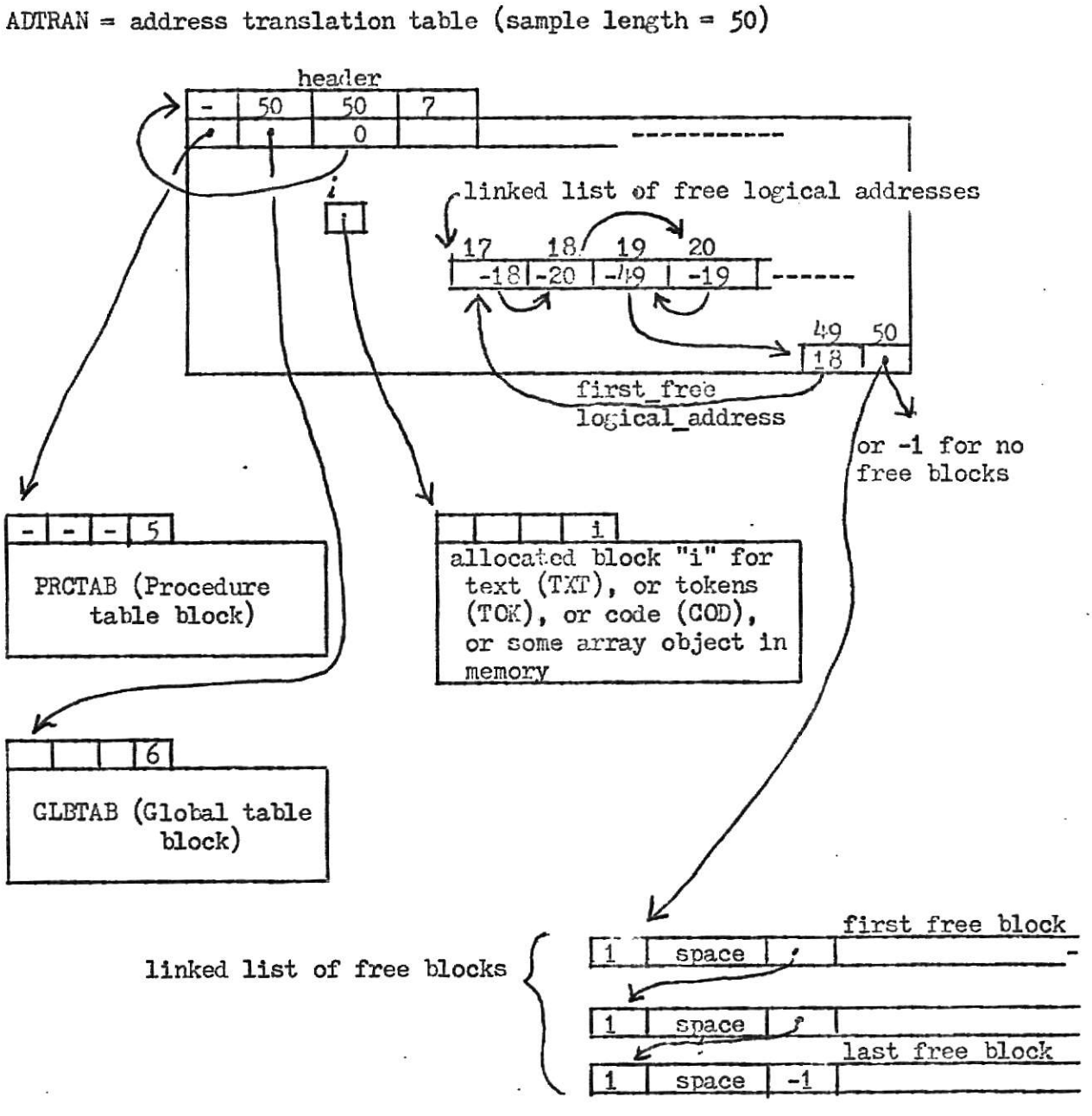
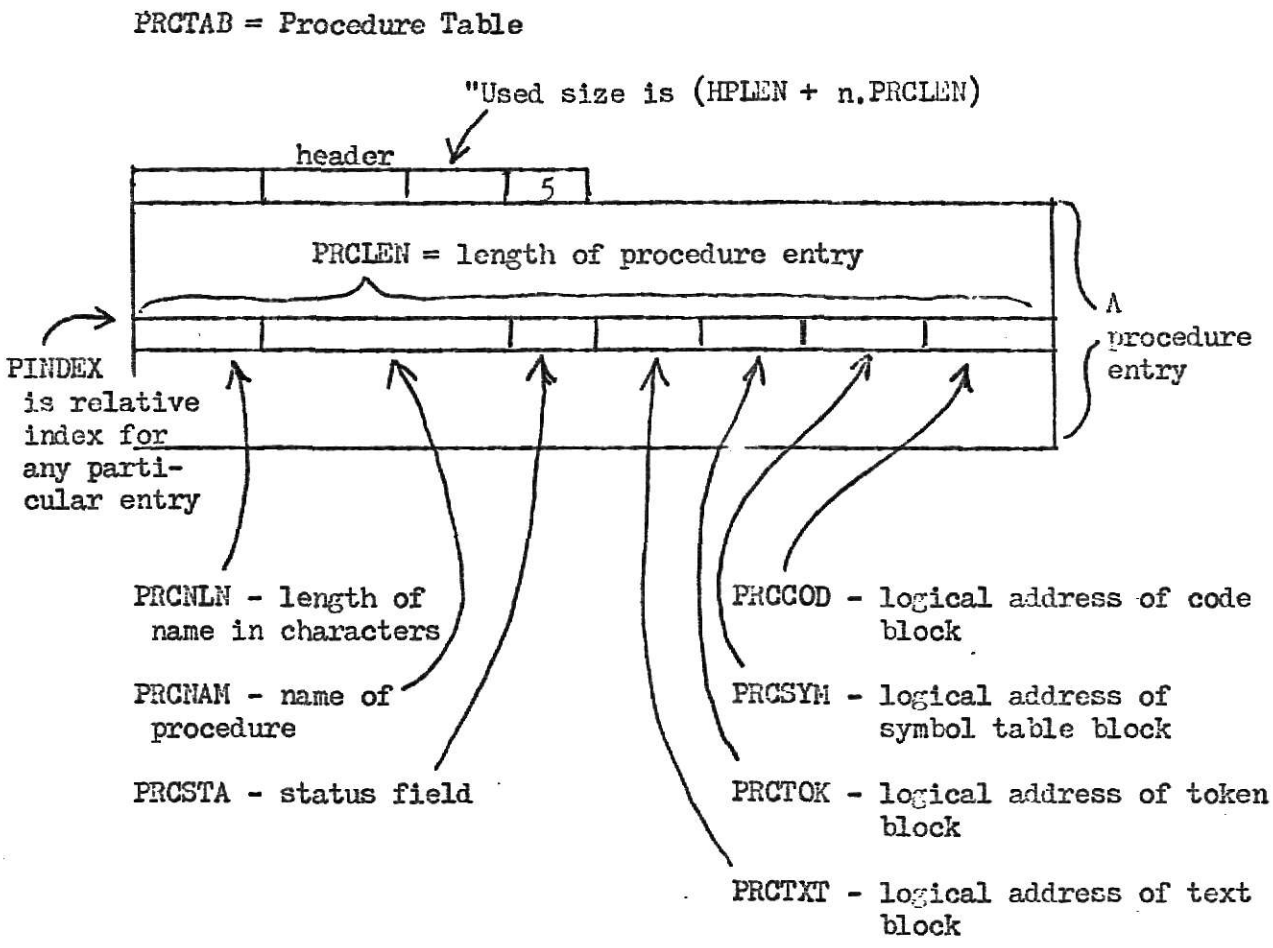


Figure 5

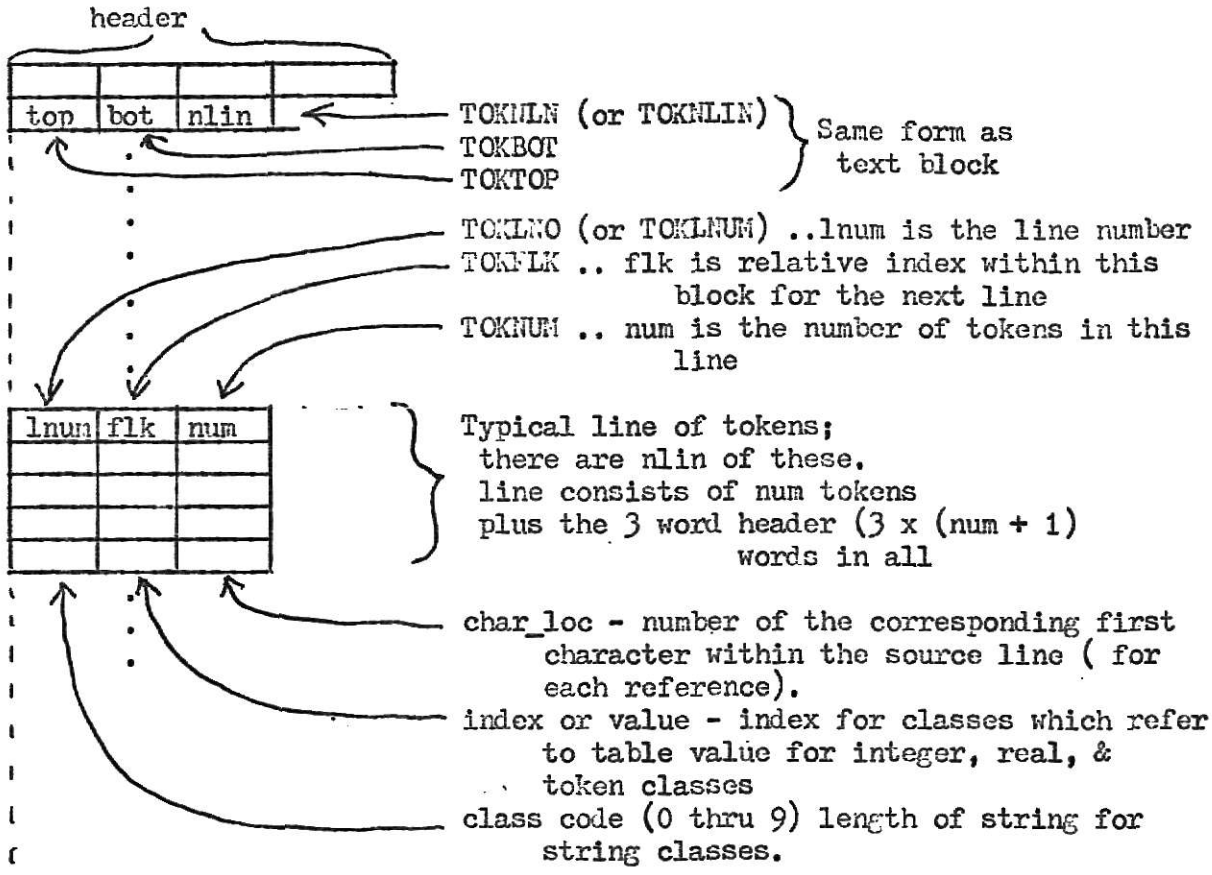
Procedure Symbol Table




eg., typical name is given as HEAP(absolute_address + pindex + PRCNAM)

Figure 6

Token Table




/TOKOFF/ TOKTOP, TOKBOT, TOKNLN, TOKLNO, TOKFLK, TOKNUM are offsets for accessing subfields within the token block.
eg., HEAP(absolute_address + line_relative_index + TOKLNO) gives a line number.

Figure 7

Text Table

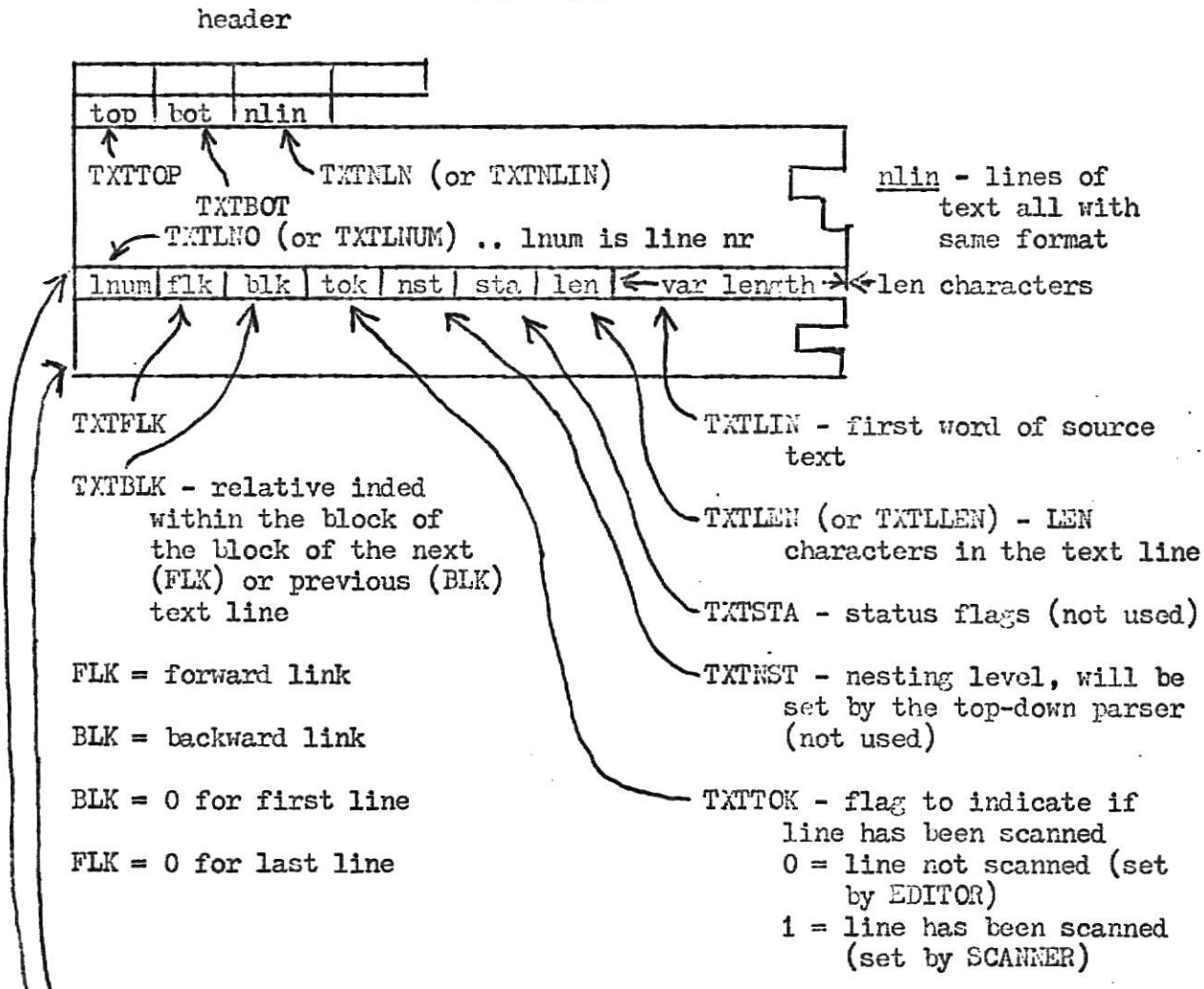
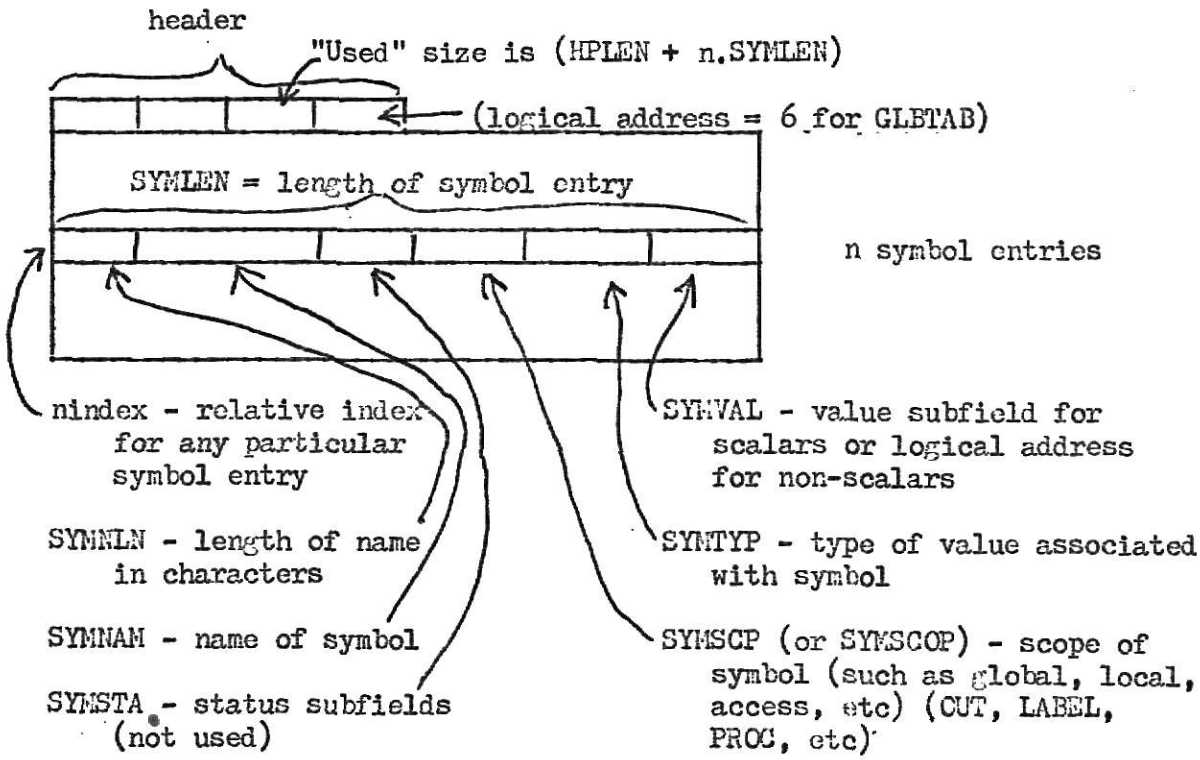


Figure 8

Symbol Table/ Global Table

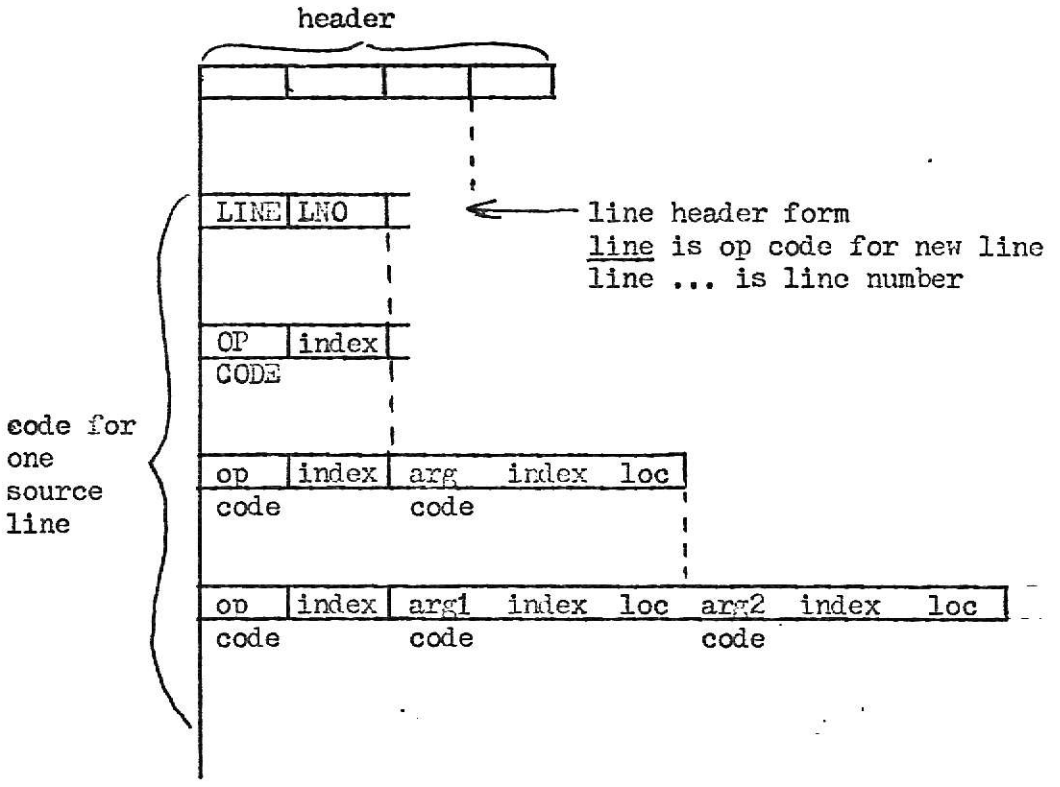



NOTE: SYMTYP and SYMVAL are not usually set at compile time.

/SYMOFF/ SYMNLN, SYMNAM, SYMSTA, STMSCP, SYMTYP, SYMVAL are offsets for accessing subfields.
eg., HEAP(absolute_address + nindex + SYMNAM) is typical symbol name

Figure 9

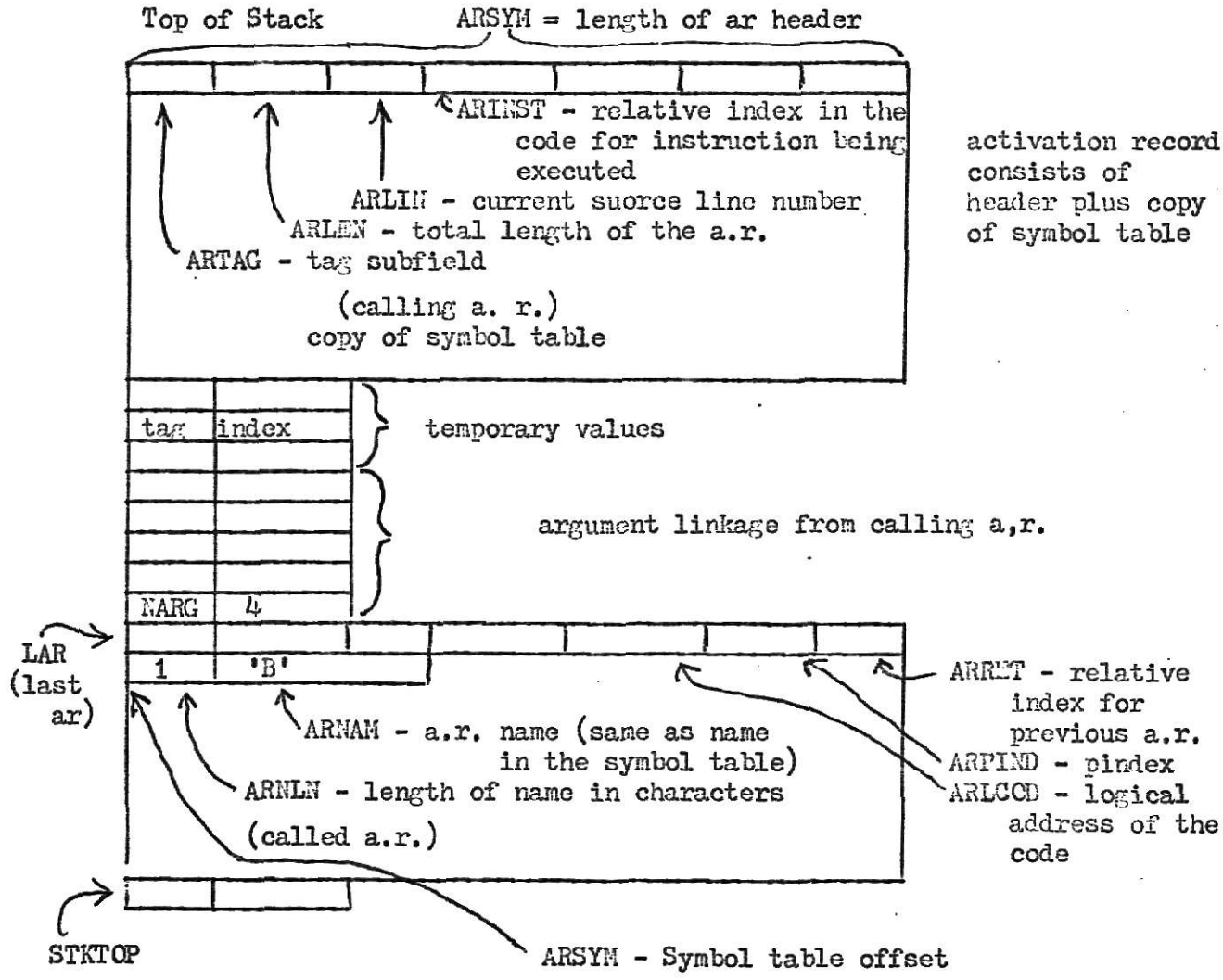
Code Table




NOTE: Code for a typical source line consists of a line header followed by a number of variable length instructions, each with an operation subfield and either zero, one, or two operand subfields.

Figure 10

Stack



# CHAPTER 6

# INTERP-700 STATUS AND EXTENSIONS

6.1 CURRENT STATUS: Of the twelve major modules designed and programmed by the summer '75 CS-700 course, only the Editor, Heap, Symbol_Table, and Errorprint routines function properly. These routines have been tested and debugged to operate in accordance with program specifications. The CLI, Scanner, and Top-Down Parser are basically operable, but small errors have been found during testing. The status of the Stack, Inter, and Operators have not yet undergone rigorous testing. Each set of module routines appeared to operate satisfactorily separately, it is yet to be determined whether they will properly interact with each other during integration testing. The Filer and Bottom-Up Parser routines do not function according to specifications. Initial indications are that the Bottom- Up Parser will have to be extensively modified, and that the Filer routines will have to be completely redone. Inconsistancies have been resolved and all naming errors have been eliminated by reducing the length of names used.

6.2 FUTURE EXTENSIONS: Enhancements to INTERP-700 fall within three areas - direct language extensions; user interface extensions; and a variety of unrelated areas titled miscellaneous.

6.2.1 DIRECT LANGUAGE EXTENSIONS:

6.2.1.1 Implement various APL array operators.

6.2.1.2 Incorporate the link to all FORTRAN library functions.

6.2.1.3 Implement 'tuples' (ie., allow dynamic 'ragged' arrays with mixed type elements).

6.2.1.4 Implement structures - allow indexing by name qualification - requires modification of symbol table.

6.2.1.5 Support composit objects, so that referance to an object also provides reference tc all sub-objects.

6.2.1.6 Implement control statements such as: 'continue loop n', 'quit loop n', 'case', 'quit case n'

6.2.1.7 Implement integer numbering of control structures.

6.2.1.8 Implement a dynamic scope rule for 'external' variables.

6.2.1.9 Extend I/O facilities to use FORTRAN's format statements.

6.2.1.10 Implement a 'pure' right-to-left expression scan to match APL.

6.2.1.11 Support a link to allow calls to user's FORTRAN (or other) subprograms.

6.2.1.12 Implement an 'assertion' facility to be used as a run. time check.

6.2.2 USER INTERFACE EXTENSIONS:

6.2.2.1 Incorporate a simple MACRO facility into the system (ie., pass tokens directly to and from the macro prccessor).

6.2.2.2 Implement reccgnition of user input numbering of control structures (as while n ..do..end loop n).

6.2.2.3 Implement a nesting level count and a 'reformated' listing (like NEATER, but not a pre processor).

6.2.2.4 Implement a 'root' name recognition (eg., use short roots of identifiers for input where not ambiguous - requires

modified symbol table search).

6.2.2.5 Support a 'graphics text editor' - editing using a CRT and 'cross hairs' for line and character selection, overtyping corrections, 'menu' selection.

6.2.2.6 Implement overstrike characters in the line recognizer.

6.2.2.7 Provide a 'HELP' feature - command cues, vocabulary explanations, etc.

6.2.2.8 Support debugging trace and breakpoint features by line, by variable, and by decision point.

6.2.2.9 Support a 'usage monitor' which records behavior of user's programs (such as types of variables, max size of arrays, frequency of execution of paths and modules).

6.2.3 MISCELLANEOUS:

6.2.3.1 (Advanced) post-compiler: allow user to 'fix' attributes in symbol table as current (or user defined) type and size; and compile an object program from the code version of the original program (eg., eliminate the dynamic type checking).

6.2.3.2 Incorporate software monitors to measure the performance of INTERP-700; report data such as: size of objects, heap fragmentation, interpreter overhead (size and speed), address translation overhead, overhead for dynamic checking, costs for scan, parse, execution, etc.

6.2.3.3 Add assertions and more detailed specifications for modules and provide an informal proof of key modules.

6.2.3.4 Add facility to create, parse, and execute code objects under program control.

# CHAPTER 7

# PROGRAM DOCUMENTATION

7.1 GENERAL: INTERP-700 programs are modular in design, expecting certain inputs and returning an expected range of outputs. No subroutine generates fatal errors. Errors noted are passed back through the calling chain to the user for correction. To maintain flexibility, the use of internally defined constants have been avoided within individual subroutines. To accomplish this flexibility, displacement indices for each of the data structures described in chapter 5 have been defined. These displacement codes and present assigned values are contained in appendix C. Access to particular locations within the various data structures is obtained by adding the displacement value to the beginning logical address of the structure being referenced. In addition, some data within INTERP-700 is passed via common data blocks. Appendix E contains the various common data blocks which will be referenced in subroutine modules. Since INTERP-700 is modular, other parts of the interpreter are referenced by subroutine calls. All subroutine calls are listed by major module for the subroutines which comprise that module.

## 7.2 SUBROUTINE MODULES:

### 7.2.1 EDITOR:

7.2.1.1 FUNCTION - The editor module controls the text editing of INTERP-700. It provides the user with the capability of creating and modifying procedures from the terminal. Editor subroutines request storage areas from the Heap. records the

logical address of the text object in the Procedure_Table; Information to the EDIT subroutines comes in 2 forms. First, the editor module is called by the Driver module, with the index of the procedure the user wishes to edit. The second input to the editor is in the form of input from the user through the input terminal. The input may be in the form of editor commands, or merely lines of program text.

7.2.1.2 CALLING FORMS: Calling forms are divided into three catagories - Calls by other modules; calls to subroutines internal to the editor routines; and calls by the editor to modules outside of the editor module.

7.2.1.2.1 CALLS FROM OTHER MODULES: CALL EDIT (PINDEX, ERROR) - Performs the functions described in para 7.2.1.1. (For detailed and high level flow charts see CS-700 class notes).

Input parameter is:

PINDEX - The index in the procedure_table of the procedure to be edited.

Output parameter is:

ERROR - Indicates whether call was successful by passing back a zero. If other than zero, error detected.

7.2.1.2.2 CALLS TO INTERNAL SUBROUTINES:

7.2.1.2.2.1 CALL SKAN (INTEXT, NCHARS, INTXC, LNNUM, KCODE, EDTERR) - This subroutine scans an intext string which has been identified as an EDIT command. The subroutine checks to insure that the command is in fact legal; that the format is correct; and when appropriate, will identify line numbers associated with the commands. It further identifies and returns a pointer to the

first character of the input string (which follows the command) as well as assigns a command code, kcode, to identify the type of command. If required, appropriate error codes may be returned.

Input parameters are:

INTEXT - Input text string from subroutine TTYIO.

NCHARS - Defined above.

Output parameters are:

LNNUM - The number of lines as appropriate.

KCODE - Identifies the type of command as described later.

EDTERR - Indicates whether internal calls are successful by returning zero.

7.2.1.2.2.2 CALL OVRWRT (INTEXT, NCHARS, TXTBAS, PTROPN, LNNUM, INTXC, NUMWDS, EDTERR) - This subroutine provides the code for effectively linking the preceeding and following lines of an old line to a newly input line of the same line number as the old. This causes the old line to be bypassed and the new input line to be put in its place.

Input parameters are:

INTEXT, NCHARS, LNNUM, INTXC - Defined above.

TXTBAS - Base address of text.

PTROPN - The real address of space just prior to the first open space.

Output parameters are:

EDTERR - Defined above.

NUMWDS - The number of words (passed through, but not used in subroutine).

7.2.1.2.2.3 CALL SEARCH (TXTBAS, LNNUM, FLAG, PTLNFD, FLINK,

BLINK, EDTERR) - This subroutine provides for 2 types of searches through the line numbers of the procedure being edited. First it will search to match one of the existing line numbers with an input line number. Second, it will look for the first line number which is larger than the input line number. In either case it will return a pointer to the line found. If input FLAG=0, the search will be for a line number equal to LNNUM and return an error code if not found. If the input FLAG=1, the search will be for the first line number in sequence which is greater than the LNNUM. If no such line is found, or a line number is found equal to LNNUM, an error code is returned.

Input parameters are:

TXTBAS, LNNUM, FLAG - Defined above.

Output parameters are:

PTLNFD - The pointer to the identified line.

FLINK - The forward link of the line.

BLINK - The backward link of the line.

EDTERR - Defined above.

7.2.1.2.2.4 CALL LIST (TXTBAS, KCODE, LNNUM, RETCOD, EDTERR) - Subroutine LIST provides for a listing of a single line of the program text when KCODE = 10, or for a complete listing when KCODE = 4. When KCODE is 10, the line to be listed is input as LNNUM.

Input parameters are:

TXTBAS, LNNUM, KCODE - Defined above.

Output parameters are:

RETCOD - Indicates the successful completion of a command operation if zero.

EDTERR - Defined above.

7.2.1.2.2.5 CALL DELETE (IXTBAS, LNNUM, RETCOD, EDTERR) - This subroutine conducts the linking and delinking to effectively bypass a line of text which has previously been entered in the text object.

Input parameters are:

TXTBAS, LNNUM - Defined above.

Output parameters are:

RETCOD, EDTERR - Defined above.

7.2.1.2.2.6 CALL SPACE (INTEXT, NCHARS, INTXC, LTXT, NUMWDS, PTROPN, TXTBAS, ERROR) - This subroutine determines the number of words required to add the current text line and its header to the Heap. It compares the number needed with the number available. If there is sufficient space in Heap it returns to the calling subroutine. If more space is needed, it requests an additional 50 words from Heap.

Input parameters are:

INTEXT, NCHARS, INTXC - Defined above.

LTXT - The index to the real address of the text object.

Output parameters are:

NUMWDS, PTROPN, TXTBAS, ERROR - Defined above.

7.2.1.2.2.7 CALL APPEND (INTEXT, NCHARS, INTXC, LNNUM, ITYPE, KCODE, PTROPN, TXTBAS, LTXT, RETCOD, ERROR, EDTERR) - This subroutine adds lines to the text object in the Heap. It handles three different types (KCODES = 1,8,9). If the operator entered only '<A' he wishes to append a line but doesn't know the current last line number. Append returns the line number at which the

line is to be entered. Append also adds lines with commands of the form '<200 CALL E(2)' and of the form 'CALL E(2)'. The later one being used after the line number has been returned by the '<A' command or the user is entering a new procedure.

Input parameters are:

INTEXT, NCHARS, INTXC, LNNUM, and KCODE - Defined above.

Output parameters are:

PTROPN, TXTBAS, LTXT, RETCOD, ERROR, EDTERR - Defined above.

7.2.1.2.2.8 CALL INSERT (INTEXT, NCHARS, LNNUM, TXTBAS, PTROPN, RETCOD, ERROR, EDTERR, INTXC) - This subroutine inserts lines which the user wishes to add after he has begun to create a procedure. The line inserted can be before the first line or between any other two lines.

Input parameters are:

INTEXT, NCHARS, LNNUM, and INTXC - Defined above.

Input/Output parameters:

TXTBAS, PTROPN - Defined above.

Output parameters are:

ERROR, RETCOD, and EDTERR - Defined above.

7.2.1.2.2.9 CALL PACK (INTEXT, NCHARS, INTXC, NUMWDS, PTROPN, TXTBAS, EDTERR) - This subroutine packs the line of text into the text object in Heap following the line header. If the text starts at the beginning of a word in the intext array, PACK will pack the text one word at a time. If the text doesn't start at the beginning of a word, PACK uses subroutines GETCHR and PUTCHR to get and put the text in Heap one character at a time. PACK

updates the pointer to the next open space.

Input parameters are:

INTEXT, NCHARS, INTXC, and NUMWDS - Defined above.

Input/Output parameters are:

PTROPN, and TXTBAS - Defined above.

Output parameter is:

EDTERR - Defined above.

7.2.1.2.2.10 CALL RESEQ (TXTBAS, RETCOD, EDTERR) - This subroutine starts with the first line and then, following the forward links, traces through the entire program and converts the line numbers to consecutive multiples of 100 starting with 100 and continuing to the number of lines in the procedure being edited.

Input parameter is:

TXTBAS - Defined above.

Output parameters are:

RETCOD, and EDTERR - Defined above.

7.2.1.2.2.11 CALL EDERPR (EDTERR) - This subroutine prints error messages to the user and returns control to the point where the user can reenter the line which produced the error. The errors handled by EDERTR are only those which are generated within the editor module. All other errors, those passed into editor by other modules, are returned to the translator driver.

Input parameter is:

EDTERR - Defined above.

Output parameters: None.

7.2.1.2.3 CALLS TO OTHER MODULES: Editor routines call the following external modules: PRCTAB; GET; EXPAND; TTYIO.

7.2.1.3 COMMON BLOCKS: Blocks of common data used by the editor and its subroutines are - HEAP; HPOFF; PRCOFF; and TXTOFF. The contents of each common block can be found in appendix B.

7.2.2 COMMAND LINE INTERPRETER (CLI):

7.2.2.1 FUNCTION - CLI interprets input command lines. Driver will recognize the first character ')' and calls CLI. CLI will in turn call MCVTOK and GETTCK to translate the command line tokens. CLI verifies the syntax of the command line and determines what further action is required. If it is determined that the command line requires either ON, OFF, RUN, PARSE, or EDIT action, control will be returned to driver for disposition. If the command line pertains to any other implemented function of the interpreter, CLI will call the appropriate module to complete the processing. In either case notification will be provided to driver of the disposition. In the case where driver controls the action, a numeric code corresponding to the required action will be returned to driver as a key to its requirements. In the case where some other module controls the action, that module will return a status code to CLI indicating either success or failure. CLI will then return that code to driver. Should CLI detect an error during its processing, a call will be made to ERRPRT module to output an appropriate error message and a failure code returned to driver. Errors can result from syntax or reference to a nonimplemented function. CLI will also generate error messages under certain circumstances when calls to modules STACK and HEAP result in errors during their processing as well as return the error code to driver.

7.2.2.2 CALLING FORMS: Same format as used for Editor.

7.2.2.2.1 CALLS FROM OTHER MODULES: CALL CLI (PINDEX1, PINDEX2, CLASS) - This is usually a call from the Driver and performs the function described in paragraph 7.2.2.1.

Input parameter is:

PINDEX1 - An index into the procedure table which provides the location in the Heap of the command line text and tokens.

Output Parameter is:

PINDEX2 - An index into the procedure_table corresponding to a procedure to be edited, parsed, or run.

CLASS - Is the class of activity which is required to complete processing of the command line after it has been interpreted by the CLI. Class is also used to return a status indication to the Driver when some other module has been called to complete the processing. Class would also be used to return a status indicator showing an error to the driver if either CLI, or one of the called modules detects an error in the command line during processing.

7.2.2.2.2 CALLS TO INTERNAL SUBROUTINES: NONE; but development of an in line version was being rewritten at the time of this report.

7.2.2.2.3 CALLS TO EXTERNAL MODULES: The CLI calls the following external modules: FILER; PRCPRT; GLBPRT; PRCSCH; PRCDEL; GLBSCH; HPCLR; CLRSTK; POP; STACK; and ERRPRT.

7.2.2.3 COMMON BLOCKS: CLI uses the common data block HEAP.

7.2.3 SCANNER:

7.2.3.1 FUNCTION: Scans text and builds an internal record for each line. Characters of a source text are evaluated by line, and program words and symbols are constructed consisting of identifiers, key or command words, numbers, operators, and separators. A three word token table consisting of a class identifier, index or value, and a pointer to the first character of the word or symbol in the line of text is then associated with each legal word or symbol. Each source text line which is free from error is marked as having been scanned and tokens are stored in a designated area by line corresponding to the source text line number. The process of scanning the source text may be interrupted by the user by depressing the ATTN key on the terminal. When all lines have been scanned and errors are encountered, a message corresponding to the type of error, text line, and starting character of where the error was detected is printed, and control returned to user. If no errors are encountered, a token representation of the source text is available for further processing.

7.2.3.2 CALLING FORMS: Same format as for Editor.

7.2.3.2.1 CALLS FROM OTHER MODULES: SCAN (INDEXP, EXERR) - This subroutine performs the functions in paragraph 7.2.3.1. For additional information concerning this subroutine module see Masters Report done by James Jones October-November 1975.

Input parameter is:

INDEXP - Displacement index from Heap header area where the procedure symbol table is located

Output parameter is:

EXERR - Error information is set to error print for each line as error found.

7.2.3.2.2 CALLS TO INTERNAL SUBROUTINES:

7.2.3.2.2.1 CALL LNSCAN (ERRFLG, LINUSE, TXTDTL, L, *) - This subroutine gets the character in a line of text and puts them in a line stack. It then checks the character against the CTAB Table. If it is a separator or an operator, number, or string, subroutine FORM is called. If not an identifier, subroutine TABLE is called to check for keyword or command language. If an identifier of undefined operator, SYMIAB is called for an index. The subroutine will assign a class index and number of the first character of symbol in the line of text, or classify the line as an error, and place the line number, the number of the first character of the symbol that caused the error, and the error code in the error stack for output.

Input parameter is:

TXTDTL - The text displacement to this line from the text start address in the text.

Output parameters are:

ERRFLG - Indicates that an error was found in the line of text.

LINUSE - The number of words of storage in the token area of Heap the line of tokens will need.

L - The index to the open row in SYMST used to hold token data passed to the subroutine SCAN.

* - Formatted, unconditional return.

7.2.3.2.2.2 CALL TABLE (SYML, CLASS, INDEX, *) - This subroutine determines if an identifier is a keyword or Command Language word. Selection of table is based on CL flag set in subroutine SCAN. If true, CLTAB is used; if false, KWWORD is used. Table search is made based on the number of characters in the id. Table comparison is made on only the first four characters of the id.

Input parameter is:

SYML - The length of the symbol.

Output parameters are:

CLASS - The class associated with the token which defines whether token is keyword, command language word, or not found.

INDEX - Identifies the keyword or command language word in the table.

* - Defined above.

7.2.3.2.2.3 CALL LINFIN (IXTDTL, RESCAN, TOKLPT, *) - This subroutine is called when a line of text has been added to process text that already has been scanned, or when a line of text has been edited. It also compares the text line with all the token lines until it finds a match where the line is to be added, then changes the line pointers to add the new line.

Input parameters are:

TXTDTL - Defined above.

TOKLPT - The line token pointer.

Input/Output parameter is:

RESCAN - Indicates whether the text has been scanned

before, and whether this is an edit, addition, or deletion of the text line.

Output parameter is:

* - Defined above.

7.2.3.2.2.4 CALL FORM (CHARNU, SYML, TCHAR, ERCODE, INDEX, CLASS, CODE, NFC, *) - This subroutine checks the character identified by CHARNU in the line stack of program text, and determines its class and index, if possible, else it builds identifiers and calls PUTCHR to pack the identifier.

Input parameter is:

TCHAR - The total number of characters in the line of text

Input/Output parameter is:

CHARNU - The number of the character in the line of text.

Output parameters are:

SYML, INDEX, CLASS, and * - Defined above.

ERCODE - If non zero, an error has been detected.

CODE - Identifies the symbol as a string, number, id, keyword, command language word, or undefined operator.

NFC - The number of the line of the first symbol.

7.2.3.2.2.5 CALL NUMPAC (NFC, LENGTH, VALUE, ERCODE, *) - This subroutine packs the number requested by the character starting at location NFC of an array into one word. Determines whether the number is real or integer. If integer, the subroutine returns to the calling routine as a normal return with the number represented by variable VALUE. If the number is real, the subroutine returns to the calling routine at a point where

RVALUE, which is the variable holding the real value, can be passed to a real variable of the calling routine. If a character cannot be recognized, an ERCODE is given and return is made to the area of the calling routine where errors are handled.

Input parameters are:

NFC - Defined above.

LENGTH - Length of symbol passed into subroutine.

Output parameters are:

VALUE - Defines the returning number as a real or integer.

ERCODE, and * - Defined above.

7.2.3.2.2.6 CALL SERROR - This subroutine is called if the error flag was true. This subroutine takes data from each row of the error stack and puts it in correct form to call error print. This is done for each error condition by line number found in the text. All input and output parameters are passed through common blocks.

7.2.3.2.2.7 CALL REAL (X, Y, *) - This subroutine places real values in the token index storage area for a number that has been scanned as a real number. Y is input, X and * are outputs.

7.2.3.2.3 CALLS TO EXTERNAL MODULES: The scanner calls the following external modules: GET; GETCHR; SYMTAB; PUTCHR; EXPAND; STAX; and ERRPRT.

7.2.3.3 COMMON BLOCKS: Blocks of common data used by the scanner and its subroutines are: HEAP; HPOFF; TXTOFF; TOKOFF; and SCANER.

7.2.3.4 ADDITIONAL DATA: To function properly, the scanner must make use of special tables not common to other modules. these tables are: the Character Table; the Keyword Table; and the Command Language Keyword Table. The layout of these tables can be found in Appendix D. For detailed information concerning the use and meaning of the scanner tables see the Masters Report by James Jones October-November 1975.

7.2.4 TOP DOWN PARSER:

7.2.4.1 FUNCTION: The primary function of the TD Parser is to parse the tokens as they appear in the token table. These tokens were placed into the table by the scanner module. The TD Parser will parse the tokens to determine if there are semantic actions required and to determine syntax. The code table is the area where the code is stored as a result of parsing of the tokens in the token table. It is initialized by the CDGENI routine called from the driver before the TD Parse routine is called. Code will be generated as the token table is parsed. When an expression is encountered in the token record, the Bottom Up Parser will be called to parse the expression. When scope information is encountered in the parse of the token table, this information will be placed into the symbol table by the TD Parser. Scope information includes the following: procedure names, input/output arguments, names of called procedures, local/global variables, and labels.

7.2.4.2 CALLING FORMS: Same format as with Editor Module.

7.2.4.2.1 CALLS FROM OTHER MODULES: CALL TDPAR (PINDEX, PRTCOD)

- Designed to perform the functions described in paragraph 7.2.4.1. For flow charts see CS-700 class notes c/o Dr Hankley.

Input parameter is:

PINDEX - Displacement index from the Heap header area where the procedure symbol table can be found.

Output parameter is:

PRTCOD - Return code which indicates whether parse was successful. Non-zero equals error.

7.2.4.2.2 CALLS TO INTERNAL SUBROUTINES:

7.2.4.2.2.1 CALL MOVTOK (RETCOD, *, *) - This subroutine is used to move the token pointer that points to the token record, and to generate the required call to the CDGEN subroutine which will generate the line number code as a new line is encountered.

Input parameters: None.

Output parameters are:

RETCOD - Indicates whether the last token for a line has been encountered, and is used by the BU Parser to determine the end of the line.

*, * - Branch labels for cases(1) - that the last token in the token table has been encountered, and cases(2) - that the space in the Heap has been exceeded.

7.2.4.2.2.2 CALL GETTOK (OFFSET, TOKVECTOR, LINENO, RCODE, &1) - This subroutine obtains a three-word token and places it in the token vector.

Input parameter is:

OFFSET - Specifies how many tokens ahead to look (e.g., zero indicates that the current token is to be

obtained). A non-zero offset allows lookahead without moving the token pointer. Lookahead may be performed only within a single line.

Output parameters are:

RCODE - Set to one of the token obtained is the last token of the line. RCODE is 3 and the &1 exit is taken if the specified token does not exist within the current line. RCODE is 4 and the &1 exit is taken if the offset is negative.

LINENO - The current line number is returned for use in diagnostic messages.

TOKVECTOR - The address of the token.

7.2.4.2.2.3 CALL CDGEN (NWORDS, CODEVECTOR, RCODE, &1) - This subroutine generates a specified number of words of code in the code block.

Input parameters are:

NWORDS - Disignates the number of words to be generated.

CODEVECTOR - The vector containing the code values.

Output parameters are:

RCODE - Will be zero unless there is insufficient storage in HEAP to contain the code block in which case it will be one and the &1 exit will be taken. If NWORDS is not positive, RCODE will be 2 and the &1 exit return will not be taken.

7.2.4.2.4 CALLS TO EXTERNAL MODULES: The TD Parser and its subroutines call the following external modules: STAX; BUPAR; CDGEN; MOVTOK; and GETTOK.

7.2.4.3 COMMON BLOCKS: Blocks of common data used by the TD Parser and its subroutines are: HEAP; and HPCFF.

7.2.4.4 ADDITIONAL DATA: In some circumstances the TD Parser is required to look ahead to determine the validity of input data. To illustrate - the form (id,id*) can continue for the maximum length of the line. The parse must look ahead to determine if id is followed by a comma. If it is, a parse is made for another identifier. If not, the parser looks for a closing parenthesis. To determine code validity the module is written to check the formats illustrated in Table 10. In addition, when code is generated, the TC Parser references a table of code tags (Appendix D).

7.2.5 BOTTOM UP PARSER:

7.2.5.1 FUNCTION: The BU Parser uses a weak operator precedence algorithm, with a left to right scan, but with a right to left precedence of operators. If the parse is successful, code is stored in the code table. If the parse is not successful, an error message is generated with a pointer to the exact location in the string where the error occurred.

7.2.5.2 CALLING FORMS:

7.2.5.2.1 CALLS FROM OTHER MODULES:

7.2.5.2.2 CALLS TO INTERNAL SUBROUTINES:

7.2.5.2.3 CALLS TO EXTERNAL MODULES: BU Parser calls the

---

* - one or more

following modules: MCVTCK; GETTOK; CDGEN; and ERRPRT.

7.2.5.3 COMMON BLOCKS: HEAP.

7.2.5.4 ADDITIONAL DATA: The EU Parser uses an operator precedence matrix.

7.2.6 STACK:

7.2.6.1 FUNCTION: The STACK is a vector of n entries, where n is the maximum space allocated to the stack. The functions performed are designed to be independent, and will respond in the same manner regardless of the calling module. The stack supports the following functions: Push an item on the stack; Pop an item from the stack; Clear the stack; Print the names of all activation records (AR) in the stack; and find an argument in the stack. (See CS-700 class notes for flow charts).

7.2.6.2 CALLING FORMS:

7.2.6.2.1 CALLS FROM OTHER MODULES:

7.2.6.2.1.1 CALL CLRSTK (ICODE) - This subroutine is used to initialize the stack pointers to zero. Used principally by the Driver via CLI.

Input parameters: None

Output parameter is:

ICODE - Indicates whether operation was successful. Non zero indicates error.

7.2.6.2.1.2 CALL PUSH (NWORDS, VECT, ICODE)/CALL PUSHAR (PINDEX, ICODE) - This subroutine will push either an activation record,

on the stack.

Input parameter is:

PINDEX - Defined above.

NWORDS - The number of words to be pushed.

VECT - The vector to be pushed on the stack.

Output parameters are:

ICODE - Defined above.

7.2.6.2.1.3 CALL POP (NWORDS, VECT, ICODE)/POPAR (ICODE) - This subroutine performs the inverse of PUSH or PUSHAR type of pop to be performed will be identified by the value of the IDS.

Input parameter is:

IDS - Defined above.

Output parameters are:

ICODE - Defined above.

COMMON BLOCK VECTOR IO - Defined above.

7.2.6.2.1.4 CALL SFIND (NUM, ICODE) - This subroutine will search the argument list preceeding the current activation record for the argument defined by NUM, and return its value and tag code to the calling program via vector IO.

Input parameter is:

NUM - Defined above.

Output parameters are:

ICODE - Defined above.

VECTOR IO - Defined above.

7.2.6.2.1.5 CALL SDISP (ICODE) - This subroutine will display the name of the activation records on the stack beginning with the last activation record pushed on the stack. No input is expected and output parameter ICODE has been defined.

7.2.6.2.2 CALLS TO INTERNAL SUBROUTINES: NONE.

7.2.6.2.3 CALLS TO EXTERNAL MODULES: The only call to outside modules is performed by PCP when FREE is called.

7.2.6.3 COMMON BLOCKS: Blocks of common data used by the stack subroutines are: HEAP; CODES; STACK; PRCOFF; SYMOFF; HPOFF; ARGFF; STKPTS; and STKARG.

7.2.7 HEAP:

7.2.7.1 FUNCTION: HEAP is the main storage area for the entire INTERP-700. Each module will request storage in the heap and when finished free the space. When a module needs more area than it has already requested, a call is made to increase its allocated space. If additional space cannot be found, the Heap routines will compact all available space within the Heap and again try to fulfill the call for space.

7.2.7.2 CALLING FORMS:

7.2.7.2.1 CALLS FROM OTHER MODULES:

7.2.7.2.1.1 CALL GEI (SIZE, TYPE, LAD, RCODE) - This subroutine is used to obtain space in the Heap.

Input parameters are:

SIZE - The amount of space requested.

TYPE - The type of information to be stored.

Output parameters are:

LAD - The logical address of where the storage is located in the Heap.

RCODE - Indicates whether call was successful. Non zero indicates error.

7.2.7.2.1.2 CALL FREE (LADDR, RCODE) - This subroutine frees the space no longer needed in the Heap.

Input parameter is:

LADDR - Logical address of area to be freed.

Output parameter is:

RCODE - Defined above.

7.2.7.2.1.3 CALL HPCLR (Code) - This subroutine clears and initializes the Heap.

Input parameter: None

Output parameter is:

CODE - Same as RCODE above.

7.2.7.2.1.4 CALL EXPAND (SIZE, LADDR, CODE) - This subroutine used when a module needs more area than it has already requested.

Input parameters are:

SIZE, and LADDR - Defined above.

Output parameter is:

CODE - Defined above.

7.2.7.2.2 CALLS TO INTERNAL SUBROUTINES:

7.2.7.2.2.1 CALL COMPAC (CODE) - This subroutine moves all objects up to make one enlarged free object in the Heap area.

Input parameters: None

Output parameter is:

CODE - Defined above.

7.2.7.2.2.2 CALL GTMAIN (SMIN, SMAX, ABS, RCODE) - This subroutine obtains the space in the HEAP requested in the GET call.

Input parameters are:

SMIN - Size of the area requested plus 4.

SMAX - Size of the area requested plus 20% plus 4.

Output parameters are:

ABS - Absolute address of the block in the HEAP.

RCODE - Defined above.

7.2.7.2.2.3 CALL FRMAIN (IA) - This subroutine frees the space in the HEAP as requested by subroutine FREE.

Input parameter is:

IA - The absolute address in the Heap of the space to be freed.

7.2.7.2.3 CALLS TO EXTERNAL MODULES: NONE

7.2.7.3 COMMON BLOCKS: The Heap routines uses the following common data areas: HEAP; HPOFF; and CODES.

7.2.7.4 ADDITIONAL DATA: Data is stored in the Heap using tags to indicate the meaning of the data following. See Appendix C for the values of code tags.

7.2.8 TABLES:

7.2.8.1 FUNCTION: The Table module sets up and maintains three tables - the procedure symbol table; the symbol table; and the global table. In addition, Table subroutines will search appropriate tables for calling routines and return the relative index of the identifier sought. They will also delete table entries no longer needed.

7.2.8.2 CALLING FORMS:

7.2.8.2.1 CALLS FROM OTHER MODULES:

7.2.8.2.1.1 CALL PROTAB (LEN, NAME, PIN, RET) - Given the length and name, this subroutine searches the procedure symbol table for a procedure. If found, the location is returned through PIN. If the name is not found, it is added to the list of procedures, and a symbol table is initialized for it. The address of the new entry is then returned in PIN.

Input parameters are:

LEN - The length in characters of the variable name.

NAME - The name given to the procedure.

Output parameters are:

PIN - The location of an entry relative to the start of the table.

RET - Error return. Non zero indicates an error.

7.2.8.2.1.2 CALL SYMTAB (PIN, LEN, NAME, NIN, RET) - Given the length and name, this subroutine searches the symbol table for an identifier. If found, the location is returned through NIN. If the name is not found, it is added to the list, and the relative location is returned via NIN.

Input parameters are:

PIN, LEN, NAME - Defined above.

Output parameters are:

NIN - Location of the identifier relative to the beginning of the symbol table.

RET - Defined above.

7.2.8.2.1.3 CALL GLEDEL (GIN, RET) - Given the relative address of the entry in the table this subroutine deletes that entry. It

also relocates entries to eliminate empty spaces between entries.

Input parameter is:

GIN - Location of the global relative to the beginning of the global table.

Output parameter is:

RET - Defined above.

7.2.8.2.1.4 CALL PRCDEL ( PIN, RET) - Given the relative address of the entry in the table, this subroutine deletes that entry. It also relocates entries to eliminate empty spaces between entries.

Input parameter is:

PIN - Defined above.

Output parameter is:

RET - Defined above.

7.2.8.2.1.5 CALL PRCPRT (RET) - This subroutine prints out a list of procedure names located in the procedure table.

Input parameter: None.

Output parameter is:

RET - Defined above.

7.2.8.2.1.6 CALL GLBPRT (RET) - This subroutine prints out a list of global names located in the global table.

Input parameter: None.

Output parameter is:

RET - Defined above.

7.2.8.2.2 CALLS TO INTERNAL SUBROUTINES/FUNCTIONS:

7.2.8.2.2.1 FUNCTION STREQ (LEN, NAM, LEN1, NAM1) - Tests for equality of strings stored in NAM and NAM1 respectively.

Input parameters are:

LEN - Length in characters of the string NAM.

NAM - First string.

LEN1 - Length of the second string.

NAM1 - The second string.

Output parameter: Function is defined as logical, therefore either a .true. or .false. is returned to calling routine.

7.2.8.2.2.2 CALL STRASG (LEN, NAM, LEN1, NAM1) - Same parameter meanings as in STREQ. The values of LEN and NAM are transfered to LEN1 and NAM1. That is, LEN and NAM are input parameters, and LEN1 and NAM1 are output.

7.2.8.3 CALLS TO EXTERNAL MODULES: The table module and its subroutines call the following : EXPAND; GET; FREE; GLBSCH; and SYMSCH.

7.2.8.4 COMMON BLOCKS: The table module utilizes the following common blocks of data: HEAP; DEBUG; PRCOFF; HPOFF; CODES; and SYMOFF.

7.2.9 INTER:

7.2.9.1 FUNCTION: The INTER module is responsible for executing the code generated by the other modules. It determines the type of operator and calls either the stack or operator routines. It handles all branching code and links arguments between caller and called procedures. It places an argument list on the stack for subroutine and function calls. It causes execution (activation) records to be built and placed on the stack, and keeps track of where execution is taking place. In addition, errors detected during execution are relayed to an error print routine for

correction by the user.

7.2.9.2 CALLING FORMS:

7.2.9.2.1 CALLS FROM OTHER MODULES: CALL INTER (INDEX, IE) - This subroutine checks the opcode and branches to the appropriate subroutine. It is called by the Driver with an index into the Heap.

Input parameter is:

INDEX - An index into the Heap where the procedure symbol table can be found.

Output parameter is:

IE - Return code. If non zero, an error has been detected.

7.2.9.2.2 CALLS TO INTERNAL SUBROUTINES:

7.2.9.2.2.1 CALL ARGLSI (*, *, K, ITRIP, IE, INDEX) - This subroutine places the argument on the stack. Arguments are stacked in the order found with the name being stacked as the first argument.

Input parameters are:

*, * - The first is the normal return, the second is the error return.

K - Address of the instruction counter.

ITRIP - Absolute address in the Heap.

INDEX - Defined above.

Output parameter is:

IE - Defined above.

NB: Vector IB is used to pass the arguments to the stack, and vector IC is used by the stack to pass the top of the stack back

to the inter subroutines.

7.2.9.2.2.2 CALL BRANCH (*, *, N, ITRIP, INDEX, IE) - This subroutine handles the branching found in the code.

Input parameters are:

*, *, ITRIP, INDEX - Defined above.

N - Instruction counter.

IC - Vector used to determine location in stack where last activation record begins.

Output parameter is:

IE - Defined above.

7.2.9.2.2.3 CALL ERRINT (*, IE, I, J, INDEX) - This subroutine sets error numbers and branches to ERRPTR to print them.

Input parameters are:

* - Return location.

IE - Defined above.

Input/output parameters are:

I - Character in code which caused the error.

J - Line number on which error was detected.

INDEX - Defined above.

7.2.9.2.2.4 CALL INARG1 (*, *, L, ITRIP, INDEX, IE) - This subroutine handles the input argument linkage.

Input parameters are:

*, * - Defined above.

L - Location of instruction counter.

ITRIP - Defined above.

INDEX - Defined above.

Output parameter is:

IE - Defined above.

7.2.9.2.2.5 CALL INDIC (*, *, I, ITRIP, INDEX, IE) - This subroutine stacks indices on the stack.

Input parameters are:

*, *, ITRIP, INDEX - Defined above.

I - Location of instruction counter.

IO - Defined above.

Output parameters are:

IE - Defined above.

IB - Vector defined above.

7.2.9.2.2.6 CALL OUTARG (*, *, L, ITRIP, INDEX, IE) - This subroutine handles the output argument linkage.

Input parameters are:

*, *, ITRIP, INDEX - Defined above.

L - Location in activation record header for triples.

IO - Vector from stack with desired argument.

Output parameter is:

IE - Defined above.

7.2.9.2.2.7 CALL RETRN (*, *, INDEX, I, L, IE) - This subroutine pops the bottom activation record on the stack.

Input parameters are:

*, *, INDEX - Defined above.

I - Instruction counter.

I - Location in activation record of the triples.

IC - Vector from stack which has location of beginning of activation record.

Output parameter is:

IE - Defined above.

7.2.9.2.2.8 CALL SCAL (*, *, ITRIP, INDEX, IE) - This subroutine

pushes the next activation record on the stack for the next procedure to be executed.

Input parameters are:

*, *, ITRIP, INDEX - Defined above.

IO - Vector defined above.

Output parameters are:

IE - Defined above.

IB - Vector defined above.

7.2.9.2.3 CALLS TO EXTERNAL MODULES: Module INTER and its subroutines calls the following routines: PUSH; POP; ERRPRT; SFIND; GET; STAX; LINE; INX; READER; WRITER; OPER; PRCSCH.

7.2.9.3 COMMON BLOCKS: The following common blocks of data are used by INTER and its subroutines: PRCOFF; SYMOFF; HPOFF; AROFF; CODES; STACK; STKARG; STKPTS; and HEAP.

7.2.9.4 ADDITIONAL DATA: The INTER module is the prime user of code tags. These tags can be found in Appendix C.

7.2.10 OPERATORS: NOTE - This module presently under revision.

7.2.10.1 FUNCTION: The operator module is able to perform the following operations: Execution of triples from the stack when called by INTER; Check type of objects and arrays for compatability; Perform type conversion when appropriate; Perform arithmetic functions of addition, subtraction, multiplication, and division; Perform relational functions and stack appropriate Boolean value; Perform string functions of replacement and concatenation; Perform array functions; Performs assignment

function, temporary storage in the stack and permanent storage in the Heap; and Provides format free input and output on a limited basis.

7.2.10.2 CALLING FORMS:

7.2.10.2.1 CALLS FROM OTHER MODULES:

7.2.10.2.1.1 OPER (ITRIP, IE, *, I2PT) - This subroutine is called by INTER to perform binary operations. OPER handles all references to activation records and passes a modified instruction triple free of these references to the various operators via the vector CDEUF. Operators return results to OPER where error checking occurs and result is placed on the stack.

Input parameters are:

ITRIP - Absolute address of code triple to be executed.

* - Return pointer.

I2PT - Pointer to beginning of activation record.

IB - Common vector used to push and pop arguments from stack.

Output parameter is:

IE - Return code. Non-zero indicates an error.

7.2.10.2.1.2 CALL IWRITE (*, ITRIP, IE, IAPT) - This subroutine allows the outputting of a single variable. IWRITE is called by INTER. IWRITE calls the data conversion routine RFETCH to handle real valued arguments. Real matrices are limited to 100 elements.

Input parameters are:

* - Return location.

ITRIP - Absolute address in the Heap.

IAPT - An index into the Heap where the procedure symbol table can be found.

Output parameter is:

IE - Defined above.

7.2.10.2.1.3 CALL IREAD (*, ITRIP, IE, IAPT) - This subroutine will read a single variable having either scalar type or scalar vector type. IREAD calls FREAD, which returns the logical address to a string of tokens. The variable type and name is popped from the stack. IREAD will call GET to acquire space for construction of vectors.

Input parameters are:

*, ITRIP, IAPT - Defined above.

Output parameter is:

IE - Defined above.

7.2.10.2.2 CALLS TO INTERNAL SUBROUTINES:

7.2.10.2.2.1 CALL EQUAL - This subroutine is not very well documented. There are no parameters listed for the subroutine, and neither the listing nor CS-700 notes indicate its precise purpose. It apparently uses a common data item CDBUF for inputs. Type checking is done to determine whether triple is integer, integer array, real, real array, or character string. Since parameters are not declared, this subroutine violates development rules.

7.2.10.2.2.2 CALL GTHN - This subroutine is similar in structure and lack of precise documentation as EQUAL. No parameters are listed, and the CS-700 notes and program listing indicate the same type checking as for EQUAL is being determined. Also

violates development rules.

7.2.10.2.2.3 CALL LSTHAN - This subroutine similar to EQUAL and GTHN. No documentation, no parameters, and uses the same general coding structure as EQUAL and GTHN. Also violates development rules.

7.2.10.2.2.4 CALL PLUS (*) - This subroutine performs binary addition on scalars, integers and reals, and matrix addition on integer and real arrays. PLUS is called by the OPER Module and receives arguments in the vector CDBUF.

Input/Output parameters are apparently passed through common blocks of data.

Output parameter is:

* - Return location.

7.2.10.2.2.5 CALL MINUS (*) - This subroutine performs binary subtraction on scalars, integer and reals, and matrix subtraction on integer and real arrays. MINUS is called by the OPER subroutine and receives arguments via the vector CDBUF. Results are returned to OPER at the location defined as *.

7.2.10.2.2.6 CALL READD (X, Y, Z) - This subroutine adds two values together (X and Y) and returns the real result Z.

7.2.10.2.2.7 CALL RESUB (X, Y, Z) - This subroutine subtracts Y from X and returns the real result in Z.

7.2.10.2.2.8 CALL RESTOR (X, Y) - This subroutine assigns Y the real value of X.

7.2.10.2.2.9 CALL RFETCH (X, Y) - This subroutine assigns X the real value of Y.

7.2.10.2.2.10 CALL ASSG (*) - This subroutine is documented only in code, which makes it extremely difficult to determine the

purpose of the subroutine. As with other subroutines in this module, CDBUF is used to pass parameters into and out of the subroutine.

7.2.10.2.2.11 CALL AND (*) - This subroutine performs a logical .AND. function of two boolean variables. The arguments are passed in the vector CDBUF, and are returned to the calling routine though the location in *.

7.2.10.2.3 CALLS TO EXTERNAL MODULES: OPER and its subroutines call the following subroutines: GETCHR; GLBTAB; PUSH; FREE; GET; PCP; and FREAD.

7.2.10.3 COMMON BLOCKS: Blocks of common data used by this module are: HEAP; STROFF; ARYOFF; CODES; OPRCOM; LOCS; HPOFF; STACK; STKPTS; SYMOFF; ARGFF; and STKARG.

7.2.11 UTILITY ROUTINES:

7.2.11.1 SUBROUTINE PUTCHR (STRNG, NCHAR, CHAR) - This subroutine packs a one byte logical character into the byte specified.

Input parameters are:

NCHAR - The number of characters.

CHAR - The characters to be packed.

Output parameter is:

STRNG - The results of the packing operation.

7.2.11.2 SUBROUTINE GETCHR (STRNG, NCHAR, ARG3) - This subroutine gets a one byte logical character from the string and character number passed into the subroutine as input parameters.

The character is right justified in the output word ARG3. The other three bytes (characters) are set to blanks.

7.2.11.3 SUBROUTINE ITYIC (OUTSTR, LENOUT, INSTR, LENIN) - This subroutine is a BAL subroutine used for general I/O to the terminal. For detailed information on its use see the User's Guide for the Computer Science Graphics Package, running under VM/370: CMS*.

Input parameters are:

OUTSTR - The string of EBCDIC characters to be sent to the terminal.

LENOUT - The number of characters to send. If .LE. to 0 no output is sent, therefore no write.

Output parameter is:

INSTR - An area in core into which a string of characters read from the terminal is to be placed.

Input/output parameter is:

LENIN - The number of characters requested. If .LE. to 0, no read will be issued.

7.2.11.4 SUBROUTINE HPPRT - This subroutine prints the symbol table and any block from Heap. Interaction is between the subroutine and user via the terminal.

7.2.11.5 SUBROUTINE CPPRT (I) - This subroutine is called by HPPRT and prints any block from the Heap. I is the input parameter which defines where in Heap the data to be printed can

---

Gary Anderson, User's Guide for the Computer Science Graphics Package running under VM/370: CMS, Kansas State University, 1975

be found.

7.2.11.6 INTEGER FUNCTION TAG (I) - This function returns the character representation for code tag numbers. I is the input parameter, and is the tag code which is to be decoded.

7.2.11.7 SUBROUTINE ERRPRT (PIND, LIN, CHAR, MLEN, MESS, RCODE) - This subroutine does the error message printing for INTERP-700. Each subroutine or module which detects an error, sends a message line to be printed by this routine.

Input parameters are:

PIND - Displacement index from Heap header area where the procedure symbol table is located.

LIN - The line number where the error was detected.

CHAR - The character number in the line which is in error.

MLEN - The number of characters in the message.

MESS - The message text.

Output parameter is:

RCODE - Indicates whether call was successful by returning a zero. Non-zero indicates error.

# APPENDIX A

## ERROR MESSAGES

A.1 GENERAL: One of the basic features of INTERP-700 is the notification provided to the used when an error is encountered. A message is printed on the console with an indication of where the error occurred. Processing is halted and control returned to the user for error correction. Error messages currently printed are illustrated below. Parenthesis following error message indicates the module which caused the error message to be printed.

A.1.1 ARGUMENT DOES NOT EXIST (Inter) - An attempt was made by the interpreter routines to find an argument put on the stack. The argument number requested was greater than the number of arguments in the stack. System error. See system programmer.

A.1.2 ARGUMENT NOT AN IDENTIFIER (TD Parser) - Self explanatory. See figure 1 for correct procedure format. Correct and reenter.

A.1.3 ATTEMPT TO POP NONEXISTANT ACTIVATION RECORD (Inter) - An attempt was made to delete an activation record which should have been on the stack, but wasn't. System error. See system programmer.

A.1.4 ATTEMPT TO POP NONEXISTANT ELEMENT (Inter) - An attempt was made to retrieve an argument which should have been in the stack, but wasn't. System error. See system programmer.

A.1.5 BOOLEAN VALUE NOT FOUND ON TOP OF STACK (inter) - An attempt was made to retrieve a boolean value which should have been on top of the stack, but wasn't. System error. See system programmer.

A.1.6 CLI. APPARENT ERROR FROM SCAN REINPUT COMD LINE (CLI) -

Verify spelling of keyword or determine if keyword used is legal. Correct and reenter.

A.1.7 CLI. HEAP WAS ALREADY CLEAR. IF OPERATOR STILL WISHES TO CLEAR THE STACK, ENTER CLRSTK COMMAND.(CLI) - Key word 'CLEAR' used when heap was already cleared. Stack should also be empty if this condition exists, but may require separate clearing action for new processing to begin.

A.1.8 CLI. NO VARIABLE OR FUNCTION TO BE DISPLAYED (CLI) - User attempted to display a variable or function that has not been defined. Verify procedure. Must input variable or function before it can be displayed.

A.1.9 CLI. STACK EMPTY NOTHING TO DISPLAY (CLI) - Attempted to display list of activiation records on stack when none existed. Recheck code to determine if activation record should have been put on stack. Continue processing.

A.1.10 CLI. STACK EMPTY NOTHING TO POP (CLI) - Call made by the user to delete last activation record put on stack when none exists. Non-fatal error, recheck code to determine if activation record should have been put on stack. Continue processing.

A.1.11 CMD LINE ERROR. ARGUMENTS NOT ALLOWED (CLI) - User listed an argument with a keyword which does not contain arguments. Delete argument used, reenter command.

A.1.12 CMD LINE ERROR. INPUT COMD NOT IMPLEMENTED, TRY AGAIN (CLI) - User attempted to perform some action which will exist in future. Review current list of valid commands.

A.1.13 CCMD LINE ERROR. FIRST CHARACTER MUST BE ')' (CLI) - All command line keywords must be preceeded by closing parenthesis. Correct and reenter.

A.1.14 COMD LINE ERROR. ILLEGAL ARGUMENT (CLI) - User entered an argument not recognizable by interpreter. Correct and reenter.

A.1.15 CCMD LINE ERROR. IMPUT COMMD NOT IN LANGUAGE (CLI) - Self explanatory. Correct and reenter.

A.1.16 CCMD LINE ERROR. TOO MANY ARGUMENTS - ONLY ONE ALLOWED (CLI)- Self explanatory. Correct and reenter.

A.1.17 deleted.

A.1.18 through A.1.24 deleted.

A.1.25 DOMAIN ERROR (INTER) - An attempt was made to perform an illegal operation. (ex: /-1; or 1+'B' will result in domain error). Correct and reenter.

A.1.26 EDIT FORMAT ERROR -A CARRIAGE RETURN MUST IMMEDIATELY FOLLOW Q, R, OR A COMMANDS. (EDITCR) - These are standalone commands and should have nothing following on line, Retype correct command.

A.1.27 EDIT FORMAT ERROR - MUST HAVE BLANK FOLLOWING EDIT COMMAND LETTER. (EDITOR) - User failed to leave space following edit command character, therefore not recognized by editor. Correct and retype.

A.1.28 EDIT FORMAT ERROR -INCORRECT COMMAND SYMBOL; NOT A LEGAL CCMMAND IN EDIT. (EDITCR) - Self explanatory. Verify legal edit commands, correct and reenter.

A.1.29 ERROR IN LINE NUMBER - ONLY INTEGER LINE NUMBERS ARE ALLOWED. (EDITOR) Self explanatory. Correct and reenter.

A.1.30 EXPRESSION SYNTAX ERROR (PARSER VIA INTER) - User should never see this message. Error should have been detected before execution. Check expression grammar, correct and reenter.

Notify system programmers that possible problem exists within parser routines.

A.1.31 FIRST WORD OF ARGLIST NOT UNIQUE ARGLIST ID (INTER) - Before a new activation record is put on the stack, the arguments for the new activation record and a tag to define the number of arguments should have been put on the stack, but wasn't. System error. See system programmer.

A.1.32 deleted.

A.1.33 HEAP FULL (HEAP VIA INTER) - An attempt was made to obtain space from the heap, none existed. Reallocate additional space.

A.1.34 ILLEGAL CHAR (EU PARSER) - Illegal arithmetic expression. See Table 9. Correct and reenter.

A.1.35 ILLEGAL CHARACTER (SCANNER) - Characters are limited to those of the character code table. Correct and reenter.

A.1.36 ILLEGAL EDIT COMMAND (EDITOR VIA INTER) - An edit command, other than those defined for the editor, was requested. Error should have been detected by the editor routines. Correct command, continue processing, notify system programmers that possible problem exists within editor routines.

A.1.37 ILLEGAL LOGICAL ADDRESS (HEAP VIA INTER) - An attempt was made to delete data in a logical address in the heap, but address reference was empty. System error - see system programmer.

A.1.38 ILLEGAL VALUE (SCANNER VIA INTER) - User should never see this message. Error should have been detected before execution. Verify value is legal entry, correct and reenter. Notify system programmers that possible problem exists within scanner routines.

A.1.39 through A.1.41 deleted.

A.1.42 INSTRUCTION NOT IMPLEMENTED (TD PARSER) - Notifies the user that although the instruction may be syntactically correct, the necessary code has not been developed and cannot be executed.

A.1.43 INVALID EXPRESSION (TD PARSER) - Printed by top down parser when error is noted by bottom up parser. See Table 9 for correct arithmetic expressions. Correct and reenter.

A.1.44 deleted.

A.1.45 INVALID KEYWORD (TD PARSER) - Keyword used either misspelled or incorrect. See keyword table Appendix D) for valid keywords. Correct and reenter.

A.1.46 INVALID OPERATOR CODE (OPERATOR VIA INTER) - An illegal operator code was used. See figure 2 for proper op codes, correct and reenter.

A.1.47 LABEL IS NOT AN IDENTIFIER (TD PARSOR) - Self explanatory. Correct and reenter.

A.1.48 LINE NUMBER NOT FOUND . (EDITOR) - User tried to list a specific line number which does not exist. Either retype a current line number, or list entire procedure to verify line numbers.

A.1.49 LINE NUMBER ALREADY EXISTS. (EDITOR) - User tried to insert or input a duplicate line number. Retype line with an original line number.

A.1.50 MISPLACED PROC STATEMENT (TD PARSER) - Self explanatory. See figure 1 for correct procedure format. Correct an reenter.

A.1.51 MISSING COLON (ID PARSER) - Self explanatory. Correct and reenter.

A.1.52 MISSING COMMA OR PARENTHESIS (TD PARSER) - Self explanatory. Correct and reenter.

A.1.53 MISSING ELSE STATEMENT (TD PARSER) - Part of IF statement. See Table 8 for correct format. Correct and reenter.

A.1.54 MISSING END PROC (TD PARSER) - Must appear at end of procedure. See figure 1 for correct procedure format. Correct and reenter.

A.1.55 MISSING END STATEMENT (TD PARSER) - Part of either case or BEGIN statement. See Table 8 for correct format. Correct and reenter.

A.1.56 MISSING ENDWHILE STATEMENT (TD PARSER) - Part of while statement. See Table 8 for correct format. Correct and reenter.

A.1.57 MISSING FI STATEMENT (TD PARSER) - Part of IF statement. See Table 8 for correct format. Correct and reenter.

A.1.58 MISSING PROCEDURE IDENTIFIER (TD PARSER) - Identifier must follow 'PROC' to define the name of the procedure. See figure 1 for correct procedure format. Correct and reenter.

A.1.59 MISSING PROCEDURE STATEMENT (TD PARSER) - Self explanatory. Each procedure must begin with a procedure statement. See figure 1 for correct procedure format. Correct and reenter.

A.1.60 MISSING SEPERATER (TD PARSER) - Self explanatory. See Appendix D for legal separators. Correct and reenter.

A.1.61 MISSING THEN STATEMENT (TD PARSER) - Part of IF statement. See Table 8 for correct format. Correct and reenter.

A.1.62 NO CLOSING QUOTE ON STRING (SCANNER) - All strings must be delimited by quote marks. Correct and reenter.

A.1.63 through A.1.68 deleted.

A.1.69 PROCEDURE NAME NOT IN PROCEDURE SYMBOL TABLE (INTER) - An attempt was made to list or search for a name which did not

exist. Verify name and reenter.

A.1.70 deleted.

A.1.71 SPACE NOT AVAILABLE (TC PARSER) - An unsuccessful attempt was made to get space, allocation should be increased.

A.1.72 STACK OVERFLOW (STACK VIA INTER) - An attempt was made to put data in the stack, but no space available. Reallocate additional space.

A.1.73 STRING LENGTHS LIMITED TO 80 CHARS FOR COMPARE (TABLES) - As implemented, user should never see this message because of scanner limitations. Concatenation of strings could result in larger character strings. For comparison, character strings may not exceed 80 characters.

A.1.74 SYM TOO MANY CHAR (SCANNER) - Symbol length is restricted to eight characters. Correct and reenter.

A.1.75 SYNTAX ERROR (EU PARSER) - Illegal arithmetic expression. See Table 9 for correct formats. Correct and reenter.

A.1.76 through A.1.80 deleted.

A.1.81 VALUE ERROR (INTER) - An attempt was made to assign a variable which had not been given a value. (ex. A <- 1; C <- A + B - Assignment of B will result in value error. Assign value and reenter.

# APPENDIX B

## INTERP-700 COMMON AREAS

GENERAL STRUCTURES:

/HEAP/NHEAP, HEAP(2000)

/STACK/NSTACK, STACK(100)

OFFSETS FOR SUBFIELDS OF STRUCTURES PLUS RECORD LENGTHS:

/PRCOFF/PRCNLN, PRCNAM, PRCTXT, PRCTOK, PRCSYM, PRCCOD, PRCSTA, SRCLEN

/SYMOFF/SYMNLN, SYMNAM, SYMSCP, SYMTYP, SYMVAL, SYMSTA, SYMLEN

/HPOFF/HPTAG, HPSPAC, HPSIZ, HPLAD, HPOBJ, ADTRAN

/AROFF/ARSYM, ARLEN, ARNLEN, ARNAM, ARLIN, ARINST, ARLCOD, ARREL, ARTAG, ARPIND

/ARYOFF/ARYRNK, ARYDIM, ARYLEN, ARYSTR

/STROFF/STRLEN, STRNG

/TXTOFF/TXTTOP, TXTBOT, TXTNLN, TXTLNO, TXTFLK, TXTBLK, TXTTOK, TXINST, TXTLIN, TXTLIN, TXTSTA

/TOKOFF/TOKTOP, TOKBOT, TOKNLN, TOKLNO, TOKFLK, TOKNUM

/CODES/IPLUS, IMINUS, IMULT, IDIV, IASG, IGT, ILT, ILE, IGE, IEQ, INE, ITRUE, IFALSE, INOT, IBRT, IBRF, IBR, IBRTI, IBRFI, IBRI, ICALL, IARG, IRETN, INARG, IOTARG, NUMARG, IND, INDX, IREAD, IWRIT, INULL, IS, IRS, IBS, ISA, IRA, ICS, IBA, ITUP, ITEMP, IVAR, ISYM, ITOK, ICOD, ITXT, IPRC, IGLBTB, IADTBN, IAR, ILCC, IGLB, INAM, ILAB, IACC, IEXT, IXP

STACK ARGUMENTS:

/STKPTS/LAR, STKTOP

/STKARG/IB(2), IO(2), CODE

COMMONLY USED INDICES FOR THE $SYSLINE MODULE:

/SYSCFF/SPIND, SLTAT, SLICK, SLSYM.

COMMON FOR SCANNER:

/SCANER/CTAB(255,3), KWTAB(10,7), CLTAB(10,7), LINST(64),
SYMST(15,3), ERRST(64,3), SYM(2), TXTLAD, TOKLAD, PINDEX,
ERRCB, CLFLAG, PRCEAS, ERRCT, RVALUE(15), NEXT

# APPENDIX C

## DISPLACEMENT CODES & VALUES

### PRCOFF

PRCNLN - 1
PRCNAM - 2
PRCTXT - 5
PRCTOK - 6
PRCSYM - 7
PRCCOD - 8
PRCSTA - 4
PRCLEN - 8

### SYMOFF

SYMNLN - 1
SYMNAM - 2
SYMSCP - 5
SYMTYP - 6
SYMVAL - 7
SYMSTA - 4
SYMLEN - 7

### HPOFF

HPTAG - 1
HPSPAC - 2
HPSIZ - 3
HPLAD - 4
HPOBJ - 4
ADTRAN - 50

### ARYOFF

ARYRNK - 6
ARYDIM - 7
ARYLEN - 5
ARYSTR - 10

### STROFF

STRLEN - 5
STRNG - 6

### ARGOFF

ARSYM - 7
ARLEN - 2
ARNLEN - 12
ARNAM - 13
ARLIN - 3
ARINST - 4
ARLCOD - 5
ARRET - 7
ARTAG - 1
ARPIND - 6

### TXTOFF

TXTTOP - 5
TXTBOT - 6
TXTNLN - 7
TXTLNO - 1
TXTFLK - 2
TXTBLK - 3
TXTTOK - 4
TXTNST - 5
TXTLLN - 7
TXTLIN - 8
TXTSTA - 6

### TOKOFF

TOKTOP - 5
TOKBOT - 6
TOKNLN - 7
TOKLNO - 1
TOKFLK - 2
TOKNUM - 3

### CODES

IPLUS - 1
IMINUS - 2
IMULT - 3
IDIV - 4
IASG - 5
IGT - 50
ILT - 51
ILE - 52
IGE - 53
IEQ - 54
INE - 55
ITRUE - 56
IFALSE - 57
INOT - 58

IBRT - 100
IBRF - 101
IBR - 102
IBRTI - 103
IBRFI - 104
IBRI - 105
ICALL - 106
IARG - 107
IRETN - 108
INARG - 109
IOTARG - 110
NUMARG - 111
IND - 112
INDX - 113

IREAD - 114
IWRIT - 115
INULL - 116
IS - 200
IRS - 201
IBS - 202
ISA - 203
IRA - 204
ICS - 205
IBA - 206
ITUP - 207
ITEMP - 208
IVAR - 209
ISYM - 250

ITOK - 251
ICOD - 252
ITXT - 253
IPRC - 254
IGLBFB - 255
IADTRN - 256
IAR - 257
ILOC - 300
IGLB - 301
INAM - 302
ILAB - 303
IACC - 304
IEXT - 305
IXP - 306

# APPENDIX D

## TABLES

D.1 GENERAL - This appendix contains the local data structures used in INTERP-700 not discussed elsewhere such as those used in the scanner module. The initialization data used in these structures is dynamic and may be changed or added to as the language interpreter programs are expanded, modified, or transported to different environments.

D.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.

D.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.

D.2.2 Column two - designates the character as a letter, number, $, , operator, or separator. Zero = blank, one = letter, two = number, three = $, four = , six = operator, and seven = separator. D.2.3 Column three - designates the index number of 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.

| EBCDIC CHARACTERS | DECIMAL LOCATION | 1 | 2 | 3 |
|---|---|---|---|---|
| blank | 1 - 73 | 1 | 0 | 0 |
| ¢ | 74 | 1 | 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 |
| p | 151 | 3 | 1 | 0 |
| q | 152 | 3 | 1 | 0 |
| r | 153 | 3 | 1 | 0 |

| EBCDIC CHARACTER | DECIMAL LOCATION | 1 | 2 | 3 |
|---|---|---|---|---|
| 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 | 0 | 20 |
| blank | 237 - 239 | 1 | 0 | 0 |
| 0 | 240 | 3 | 2 | 0 |
| 1 | 241 | 3 | 2 | 0 |
| 2 | 242 | 3 | 2 | 0 |

| EBCDIC CHARACTER | DECIMAL LOCATION | 1 | 2 | 3 |
|---|---|---|---|---|
| 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 |

D.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 to be scrutinized in order to select the proper table column. The first four letters of the indentifier are then compared against all entries in the column selected in the table. Index numbers are based on column number followed by row number of matched comparisons.

KEYWORD TABLE INITIALIZATION DATA

| Coulmn | | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| Row | 1 | DC | END | CASE | BEGI | ACCE | ENDP | ENDW |
| | 2 | IF | OUT | ELSE | FALS | EXPT | 0 | EXTE |
| | 3 | IN | 0 | EXIT | WHILE | GLOB | 0 | 0 |
| | 4 | FI | 0 | GOTO | 0 | RETU | 0 | 0 |
| | 5 | C | C | PROC | 0 | 0 | 0 | 0 |
| | 6 | 0 | 0 | READ | 0 | 0 | 0 | 0 |
| | 7 | C | C | THEN | C | 0 | 0 | 0 |
| | 8 | 0 | 0 | TRUE | C | 0 | 0 | 0 |
| | 9 | C | C | QUIT | 0 | 0 | 0 | 0 |
| | 10 | 0 | 0 | CALL | C | 0 | C | 0 |

D.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 in the table. Index numbers are based on the column number followed by the row number of matched comparisons.

COMMAND LANGUAGE TABLE INITIALIZATION DATA

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

D.5 OPERATOR CODES - The following are format examples of various operations performed by INTERP-700.

D.5.1 Branches - Consist of two words. The first word is the code name for the branch, e.g., IBRI for branch indirect; the second word is the index into triples on direct branches or the index into the symbol table for indirect branches.

D.5.2 Arguments - Consist of five words. The first word is the code for the type argument; the second word is the source index;

The third word is the operand code; The fourth word is the index in the symbol table or a value; and the fifth word is the source index for the character in the current line of source code.

D.5.3 Index - Consists of five words. The first word is the code for indx; the second word is the number of indices; the third word is the operand code; the fourth word is the index in the symbol table or a value; and the fifth word is the source index.

D.5.4 Indicies - Consist of five words. The first word is the code for an indicie; the second word is the source index for the indicie code; the third word is the operand code; the fourth word is the index in the symbol table or a value; and the fifth word is the source index of the operand code.

D.5.5 Call - Consist of five words. The first word is the code for a call; the second word is the number of arguments for the call; the third word is the operand code; the fourth word is the index to the routine in the symbol table; and the fifth word is its source index.

D.5.6 Return - Consist of two words. The first word is the code for IRETN; and the second word is its source index.

D.5.7 Inarg - Consist of five words. The first word is the code for inargs; the second word is null; the third word is its operand code; the fourth word is the index into the symbol table; and the fifth word is its source index.

D.5.8 Outarg - Same as inarg except that the first word is the code for outargs.

D.5.9 Iread - Consist of two words. The first word is the code for Iread; the second word is the number of arguments.

D.5.9 Iwrite - Same as Iread except that the first word is the code for Iwrite.

D.6 The following table summarizes the code used by INTERP-700:

| NAME | NR ARGS | CODE | SOURCE INDEX |
|---|---|---|---|
| branching | 0 | IBRI | nindex |
| | | IBRF | rel index in code |
| | | IBR | rel index in code |
| return | 0 | IRETN | char_loc |
| not | 1 | INOT | char_loc |
| push argument | 1 | IARG | char_loc |
| push_ar=call | 1 | ICAL | nr of args including name of called mod |
| link_arg | 1 | INARG | null |
| b_link_arg | 1 | OUTARG | null |
| subscript | 1 | IND | char_loc |
| array_value | 1 | INDX | nr of indices |
| array_ref | 1 | none | nr of indices |
| read | 1 | IREAD | nr of args |
| write | 1 | IWRITE | nr of args |
| indexed oper | 1 | none | char_loc |
| assignment | 2 | IASG | char_loc |
| arithmetic ops | 2 | IPLUS, IMINUS, IMULT, IDIV | char_loc |
| relationals | 2 | IGT, ILT, ILE, IGE IEQ, INE | char_loc |
| logical | 2 | IAND, INOT | char_loc |
| temporary | temp | null | null |
| scalar constant | type_tag | value | char_loc |
| non_scalar const | type_tag | logical_ address | char_loc |
| variable | var | nindex | char_loc |
| string_constant | string_ constant | length | char_loc |

# APPENDIX E

## LESSONS LEARNED

E.1 It is difficult to document someone else's program when presented with only code listings. In this circumstance, it becomes necessary to generate assumptions about what is happening within the program. Should the assumptions prove to be false, the documentation becomes useless. A strong argument could be made that it might be easier and more cost effective to reprogram and document new code rather than spend the time required to research and study program code before documenting. As the complexity of existing code increases, this argument becomes more valid.

E.2 There appeared to be a direct relationship between those modules which were well documented, and those which appear to function according to specifications. The modules which are currently operational are the same modules which are well documented.

E.3 When no other documentation existed, comments within the program code made the task of documentation much easier.

E.4 Documentation is still a tiresome and boring task, but one which is so necessary if any type of maintenance is to be performed at a later time. Documenting while developing program logic would appear to be a much better way to go. The task of tying together properly developed documentation would decrease tht total time and effort required for a complete final package.

E.5 One of the most important lessons came from the developmental work on the interpreter. There has to be one

person or agency responsible for managerial direction and/or quality control. Without direction, programs, modules, or subroutines grow like topsy, sometimes without purpose. Without quality control there is no assurance that managerial directions are being followed.

# BIBLIOGRAPHY

## PRIMARY SOURCES

Class Notes. Compilition of work done by CS286-700 Computer Science course, summer session, Kansas State University, 1975.

General discussions with James Jones, graduate student, Kansas State University, 1975.

General discussions with William Hankley, Associate Professor, Computer Science Department, Kansas State University, 1975.

# INTERP-700:
# DOCUMENTATION OF A STUDENT DESIGNED INTERACTIVE INTERPRETER

by

MICHAEL FRANK MITRIONE

B.B.A., St Bernadine of Siena College, 1963

---

AN ABSTRACT OF A MASTER'S REPORT

submitted in partial fulfillment of the

requirements for the degree

MASTER OF SCIENCE

Department of Computer Science

KANSAS STATE UNIVERSITY
Manhattan, Kansas

1975

PROPOSAL: To provide a standardized document of an interactive interpreter designed and implemented by the students of CS 286-700 courses.

PURPOSE: This report describes an interpreter which will be referred to as INTERP-700. It examines and illustrates features of a high level programming language, much like APL, which is suited to a spectrum of users ranging from novice to experienced programmer. Emphasis is on documentation of INTERP-700 as it exists today, but reference is made to future enhancements which will increase the test and debug capabilities of the interpreter. Emphasis is on two areas - language documentation; and program documentation.

REPORT ORGANIZATION: This report consists of 7 chapters, each examining a different facet of the interpreter. Chapter 2 looks at the purposes of INTERP-700, both present and future, and touches on how this interpreter differs from others available. Chapter 3 discusses the language documentation, which includes a users guide, and the three language subsets of INTERP-700. Chapter 4 discusses the syntax rules of the language, and includes a section on how parameter passing is accomplished. Chapter 5 discusses the semantic rules and gives both the data structures used, and the language restrictions. Chapter 6 looks at the current status of INTERP-700, as well as future enhancements envisioned. Chapter 7 discusses each major module of

INTERP-700, describes the purposes of each, and includes input and output parameters and their meanings where feasible.