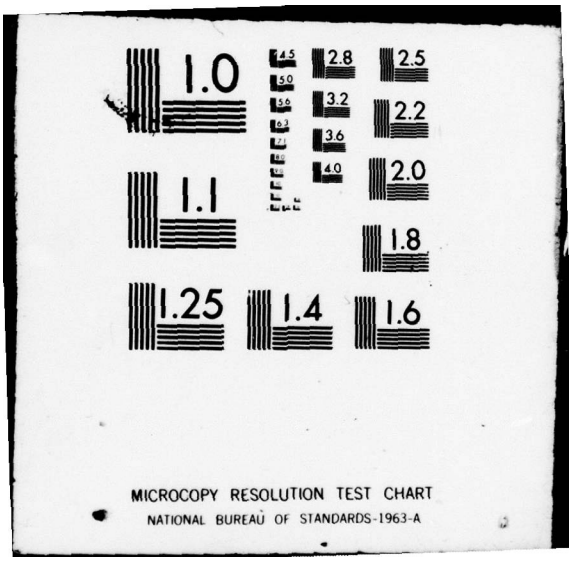
AD-A071 972 NAVAL POSTGRADUATE SCHOOL MONTEREY CA F/G 9/2
NPS-PASCAL: A PASCAL IMPLEMENTATION FOR MICROPROCESSOR-BASED CO--ETC(U)
JUN 79 J L BYRNES

UNCLASSIFIED NL

1 OF 3
AD
A071972


1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
4.5
5.0
5.6
MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

NAVAL POSTGRADUATE SCHOOL

Monterey, California

THESIS

NPS-PASCAL
A PASCAL IMPLEMENTATION FOR
MICROPROCESSOR-BASED COMPUTER SYSTEMS

by

John L. Byrnes

June 1979

Thesis Advisor: Gary A. Kildall

Approved for public release; distribution unlimited

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE	READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER	2. GOVT ACCESSION NO. 3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) NPS-PASCAL: A Pascal Implementation For Microprocessor-based Computer Systems.	5. TYPE OF REPORT & PERIOD COVERED Master's Thesis, June 1979
	6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) John L. Byrnes	8. CONTRACT OR GRANT NUMBER(s)
9. PERFORMING ORGANIZATION NAME AND ADDRESS Naval Postgraduate School Monterey, California 93940	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Naval Postgraduate School Monterey, California 93940	12. REPORT DATE June 1979
	13. NUMBER OF PAGES 282
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office) 283 p.	15. SECURITY CLASS. (of this report) Unclassified
	15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Microcomputer
Compiler
PASCAL

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

NPS-PASCAL is a Naval Postgraduate School research project whose goal is the implementation of the PASCAL programming language on a microprocessor-based system. The NPS-PASCAL compiler consists of two software subsystems, the first analyzes the source program and produces a machine-independent intermediate form, while the second produces target machine code. The system is designed to satisfy the constraints of Standard

251 450

DD FORM 1 JAN 73 1473 (Page 1) EDITION OF 1 NOV 65 IS OBSOLETE S/N 0102-014-6601

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

20. (continued)

Pascal, as defined by the British Standards Institute/ International Standards Organization Working Draft of Standard Pascal.

The analysis subsystem, defined herein, accomplishes the lexical, syntactic, and semantic analysis of a PASCAL program. It has been implemented on an Intel 8080 microcomputer, running under the CP/M operating system.

Approved for public release; distribution unlimited

NPS-PASCAL
A Pascal Implementation
For Microprocessor-based Computer Systems

by

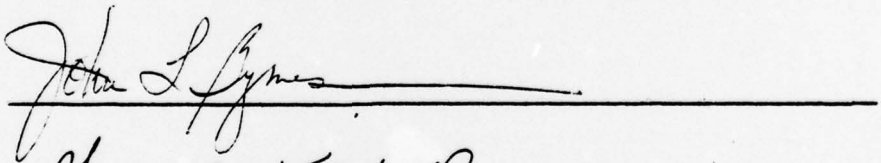
John L. Byrnes
Lieutenant, United States Navy
B.S., United States Naval Academy, 1973

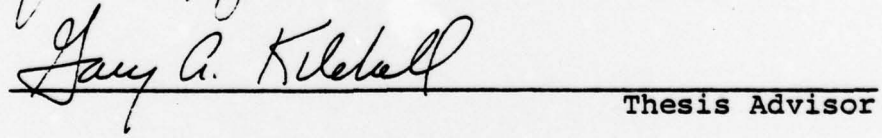
Submitted in partial fulfillment of the
requirements for the degree of

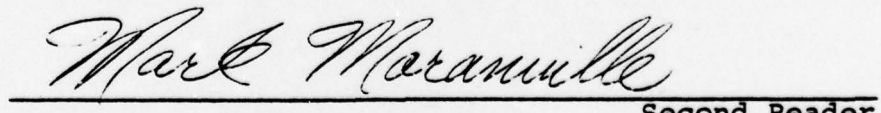
MASTER OF SCIENCE IN COMPUTER SCIENCE

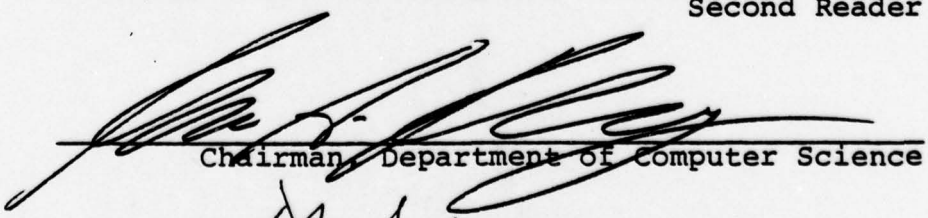
from the

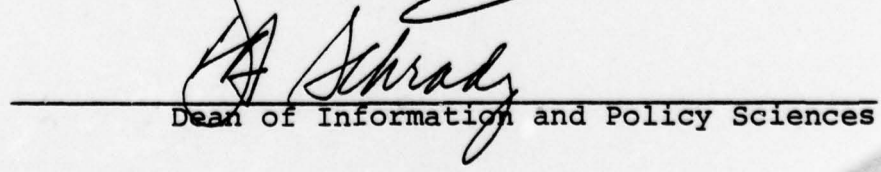
NAVAL POSTGRADUATE SCHOOL
June 1979

Author 

Approved by: 
Thesis Advisor


Second Reader


Chairman, Department of Computer Science


Dean of Information and Policy Sciences

ABSTRACT

NPS-PASCAL is a Naval Postgraduate School research project whose goal is the implementation of the PASCAL programming language on a microprocessor-based system. The NPS-PASCAL compiler consists of two software subsystems, the first analyzes the source program and produces a machine-independent intermediate form, while the second produces target machine code. The system is designed to satisfy the constraints of Standard Pascal, as defined by the British Standards Institute/ International Standards Organization Working Draft of Standard Pascal.

The analysis subsystem, defined herein, accomplishes the lexical, syntactic, and semantic analysis of a PASCAL program. It has been implemented on an Intel 8080 microcomputer, running under the CP/M operating system.

TABLE OF CONTENTS

I. INTRODUCTION---------- 8

A. BACKGROUND---------- 8

B. APPROACH---------- 8

II. NPS-PASCAL COMPILER IMPLEMENTATION----------12

A. NPS-PASCAL LANGUAGE BACKGROUND----------12

B. COMPILER ORGANIZATION----------13

C. SCANNER----------15

D. SYMBOL TABLE----------16

1. Symbol Table Construction----------16

a. Label Entries----------20

b. Constant Entries----------20

c. Type Entries----------22

(1) Scalar Types----------26

(2) Subrange Types----------26

(3) Array Types----------28

(4) Record Types----------32

(5) Set Types----------35

(6) File Types----------35

(7) Pointer Types----------35

d. Variable Entries----------39

e. Procedure and Function Entries----------39

(1) Formal Parameters----------42

f. Symbol Table Construction Procedures----45

g. Symbol Table Access----------47

2. Built-in Symbol Table Entries----------48

E. PARSER---------- 51

F. CODE GENERATION---------- 53

1. Storage Space Allocation---------- 54

a. Byte Data---------- 54

b. Integer Data---------- 54

c. Real Data---------- 56

d. String Data---------- 56

2. Arithmetic Operations---------- 58

a. Logicals---------- 58

b. Integers---------- 58

c. Reals---------- 58

3. Set Operations---------- 59

4. String Operations---------- 59

5. Procedures and Functions---------- 59

a. Invocation---------- 60

b. Storage Allocation---------- 60

c. Parameter Mapping---------- 62

d. Function Return Value---------- 65

e. Forward Declared Procedures and Functions---------- 67

f. External Procedures and Functions------ 67

g. Standard Procedures and Functions------ 68

6. Input-Output---------- 68

7. NPS-PASCAL Pseudo Operators---------- 70

a. Literal Data References---------- 72

b. Allocation Operators---------- 72

c. Arithmetic Operators---------- 73

d. Boolean Operators---------- 78

e. String Operators---------- 79
f. Stack Operators---------- 80
g. Store Operators---------- 80
h. Array Operators---------- 81
i. Set Operators---------- 82
j. File Operators---------- 83
k. Procedure and Function Operators------- 83
l. Program Control Operators---------- 87
m. Input-Output Operators---------- 88
III. CONCLUSIONS---------- 90
IV. RECOMMENDATIONS---------- 91
APPENDIX A - Compiler Error Messages---------- 92
APPENDIX B - Summary of NPS-PASCAL Operators---------- 94
APPENDIX C - Standard Tables---------- 96
APPENDIX D - NPS-PASCAL Language Structure---------- 97
APPENDIX E - Inoperative Constructs----------113
APPENDIX F - Intermediate Code DECODE Program----------115
APPENDIX G - SYMBOLTABLE Display Program----------116
NPS-PASCAL PROGRAM LISTINGS----------117
LIST OF REFERENCES----------280
INITIAL DISTRIBUTION LIST----------281

I. INTRODUCTION

A. BACKGROUND

NPS-PASCAL is an implementation of the Pascal language on an Intel 8080 microcomputer system. NPS-PASCAL is a continuing research project being developed by students in the Computer Science Curriculum at the Naval Postgraduate School, Monterey, California. The original NPS-PASCAL design and implementation was done by MAJ Joaquin C. Gracida, USMC, and LT Robert R. Stilwell (SC) USN, in their thesis submitted in June 1978. Their work is contained in Reference 1. MAJ Gracida and LT Stilwell completed work on the basic constructs of the Pascal Language by utilizing a single-pass compiler that generated intermediate code; and a code generator which then generated 8080 code from the intermediate code. With many of the Pascal constructs not implemented, thesis work was continued in October 1978 with the goal of producing a complete and debugged NPS-PASCAL compiler. Follow-on thesis work will lead to a NPS-PASCAL compiler-interpreter, and a complete NPS-PASCAL 8080 code generator. In the discussion which follows, it is assumed that the reader is familiar with the contents of Reference 1.

B. APPROACH

The first step in continuing the development of NPS-PASCAL, was to study the program listings and thesis to gain familiarity and insight into the project. A determination was then made to complete all work remaining on the compiler portion of

NPS-PASCAL, to the extent that it would meet or exceed the constructs being proposed for the standardization of the Pascal Langugage. Consequently, the BSI/ISO Working Draft 3 for Standard Pascal (BSI is the British Standards Institute; ISO is the International Standards Organization) was used as a source of Standard Pascal constructs (see reference 2).

The next step was to acquire an understanding of the PL/M cross compiler available on the CP/CMS time-sharing system on the IBM 360-67 at the Naval Postgraduate School. Tied to this was an understanding of CPM80, an expansion on Intel's INTERP/80, which provides the basic CP/M input/output facilities. The simulator contains the required facilities to test and debug PL/M programs. Reference 3 gives a detailed account of how to utilize the PLM Compiler and CPM80.

The remaining effort consisted of making additions, corrections, design changes, isolating bugs, running test programs, and developing user assistance programs. In order to implement certain constructs of the language, it was necessary to reconstruct the original grammar. Appendix E lists those features of NPS-PASCAL that were not implemented at the start of this project, and the features known to contain bugs at project completion.

Due to the non-existence of a Pascal Compiler Validation System, validation programs were taken from various textbooks on Pascal to test the compiler. Since these texts gave sample programs that demonstrated specific Pascal constructs, each NPS-PASCAL construct was tested as described below.

As a language construct was implemented in the compiler, an associated validation program was compiled to check proper operation. However, it became apparent early in project development that since there was no associated construct implementation in the code generation portion of NPS-PASCAL, checking the generated intermediate code would prove difficult. The solution was the development of the first of two user assistance programs. The NPS-PASCAL DECODE program translates the intermediate code and prints out the mneumonic form of the compiled code, along with the associated parameters for each mneumonic. A complete explanation of the DECODE program appears in Appendix F.

In compiling a validation program that revealed improper intermediate code, CPM80 was used to pinpoint errors in the compiler. Changes were then made to the source program, which was then recompiled using the PLM80 compiler. The validation program was then recompiled, and the intermediate code translated by the DECODE program, to ensure proper construct implementation in NPS-PASCAL.

As more constructs were implemented and tested, a major realization surfaced -- the need to access the symbol table during and after program compilation. The original approach taken to this problem was the addition of a PRINT$SYMBOL$TABLE subroutine to the source code. This routine printed out a symbol table entry's location, type, printname, and allocated PRT location, if any. This solution was abandoned with the decision to write the symbol table out to its own separate

file, thus providing NPS-PASCAL with the ability to access the symbol table at translation time. Consequently, the second user assistance program was developed -- the NPS-PASCAL SYMBOL-TABLE program. This program offers the NPS-PASCAL user a complete printout of the information stored in the symbol table following compilation of a Pascal program. Appendix G details the use and abilities of the SYMBOLTABLE program. The following section describes the implementation of the compiler in detail.

II. NPS-PASCAL COMPILER IMPLEMENTATION

A. NPS-PASCAL LANGUAGE BACKGROUND

NPS-PASCAL is an implementation of PASCAL based on the BSI/ISO Working Draft of Standard Pascal (2), henceforth referred to as "STANDARD PASCAL." NPS-PASCAL is in complete compliance with STANDARD PASCAL's definition of a conforming processor, with the following exceptions:

(1) Identifiers, directives, and labels can be of any length, as prescribed by STANDARD PASCAL, provided their uniqueness can be determined by the first thirty characters.

(2) Integers are limited to any value between -32,768 and +32,767. Real values can take on any negative or positive value consisting of fourteen digits multiplied by ten to the -64th power through ten to the +63rd power.

(3) "EOP" is a special symbol, or reserved word, in the NPS-PASCAL vocabulary indicating "end of program."

Consequently, any program that conforms to the rules of STANDARD-PASCAL, and meets the above three qualifications, constitutes a syntactically correct NPS-PASCAL program.

To add increased versatility to PASCAL, various features were implemented in NPS-PASCAL. These additions were designed to parallel the constructs of UCSD (Mini-Micro Computer) PASCAL (6), the current leader in PASCAL Systems for microcomputers. The implementation defined features are discussed in section II.F.

The University of Toronto's parse table generator (5) was used to specify NPS-PASCAL in LALR(1) grammar form. The program operates on the IBM 360/67 and produces the parse tables for the language, thus permitting extensions to be made in an easy and efficient manner. A complete description of the NPS-PASCAL grammar, its generation, and execution procedures are contained in the NPS-PASCAL User's Manual (8).

B. COMPILER ORGANIZATION

The compiler structure, diagrammed in Figure 1, requires a single pass through a source program to produce an intermediate language file while printing an optional source listing at the console. The one pass approach was taken to provide fast compilation and to reduce the required work and size of the compiler. The single drawback to this one pass compiler is the inability to specify the exact position where program execution is to continue after a branch. Therefore, labels are placed in the intermediate code where execution of the program is to continue. The resolution of label locations is accomplished by the code generating program as it scans the intermediate form.

The compiler builds the symbol table, converts all numbers in the source program to their internal representation, and generates the intermediate file and symbol table file on the diskette. Compiler parameters can be set to control listings of the source program, token numbers, or production numbers. Should program errors be anticipated, another compiler parameter can surpress the generation of the intermediate file.

NPS-PASCAL COMPILER STRUCTURE

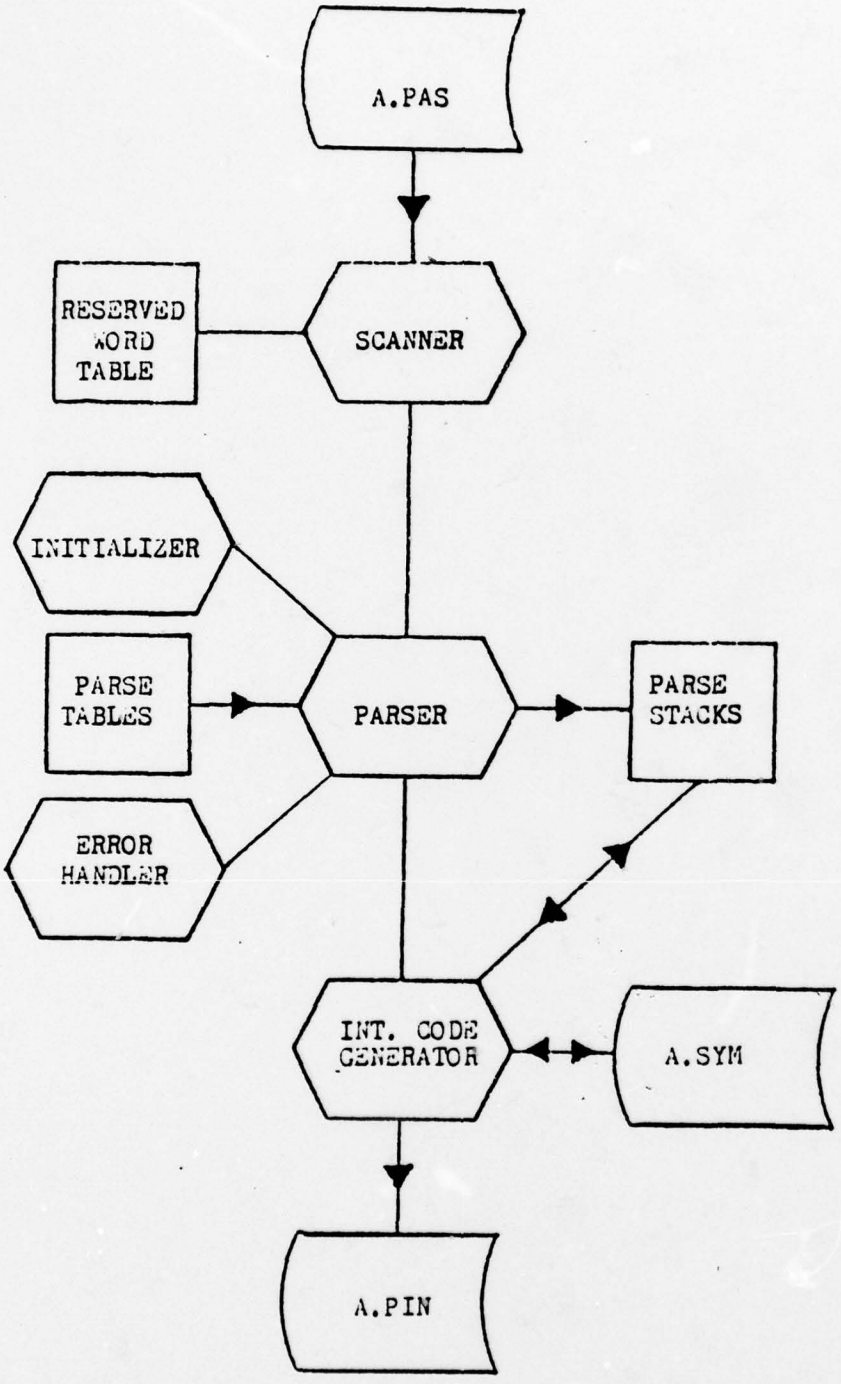


FIGURE 1

C. SCANNER

The scanner analyzes the source program character by character and sends a sequence of tokens to the parser. The scanner provides a listing of the source statements, when directed, eliminates comments, and reads the compiler parameters.

The scanner is divided into four sections which are selectively executed depending on the first non-blank character of the token. Upon determination of the scanning section, the remainder of the token is scanned and placed in the accumulator array ACCUM. The first byte of ACCUM contains the length of the token. In the case of tokens that exceed the size of ACCUM (32 bytes), a continuation flag is set to allow the scanner and parser to accept the remainder of the token.

The four sections comprising the scanner handle strings, numbers, identifiers or reserved words, and special characters. The string processing section is invoked whenever the first character of a token is a single quotation mark. The scanner then analyzes each succeeding character until a second quotation mark is scanned, indicating the end of the string. The program section that manipulates numbers determines the type of the number being scanned as it processes each character. This determination is used by subsequent routines that perform type checking and conversion to internal representation. When the scanner recognizes an identifier, it searches the vocabulary table (VOCAB) to determine if the identifier is a reserved word. If a reserved word is matched, the scanner returns the token number associated with the reserved

word's position in the VOCAB table. Special characters also found in the VOCAB table, are handled as separate tokens except in two cases. If a period is followed by numeric characters without intervening spaces, the special characters section of the scanner assumes that a real number is being scanned. This program section handles the real number in the same manner as the number section mentioned above. The second exception to special characters occurs when a pair of special characters are scanned one right after another. The scanner will pass both characters as a single token after assigning the token number from the VOCAB table.

D. SYMBOL TABLE

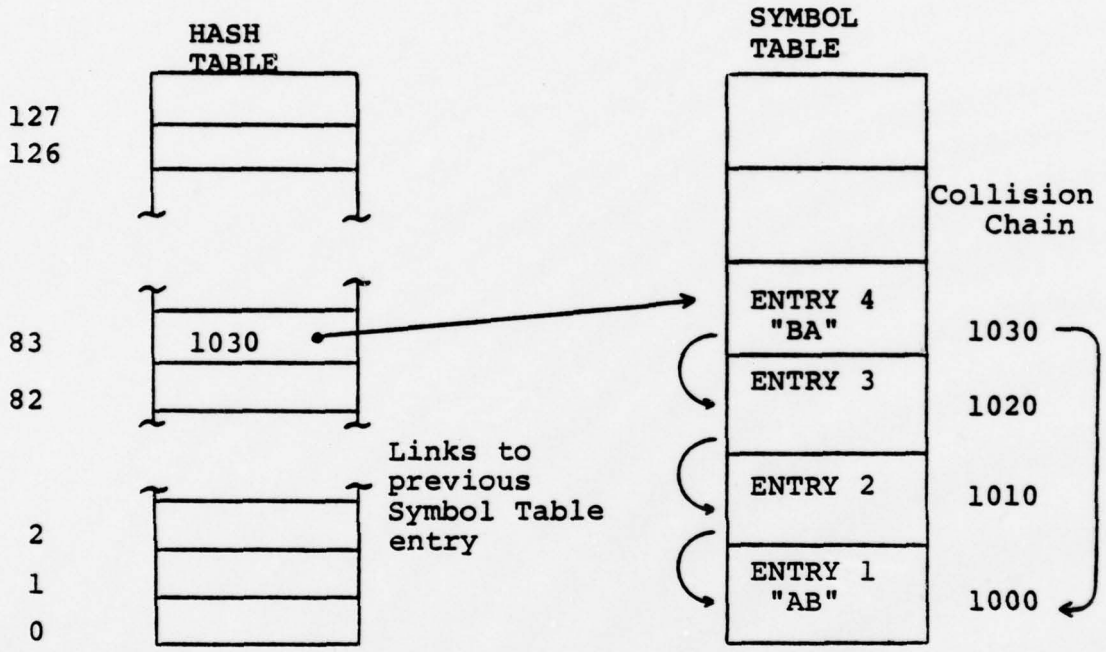
The symbol table is used to store the attributes of labels, constants, type declarations, variable identifiers, procedures, functions, and file declarations. This stored information is used by the compiler to verify that the program is semantically correct and to assist in code generation. Access to the symbol table is accomplished through various subroutines using based global variables to uniquely address the elements of each entry.

1. Symbol Table Construction

The symbol table is modelled after the Algol-M symbol table (9). It is an unordered linked list of entries which grows towards the top of memory. Individual entries are either accessed via a chained hash addressing technique as illustrated in Figure 2, or by means of address pointer fields

HASHING FUNCTION: SUM OF PRINTNAMES ASCII CHARACTERS
MODULO 128

H.F.(AB) = (41 + 42) MOD 128 = 83
H.F.(BA) = (42 + 41) MOD 128 = 83



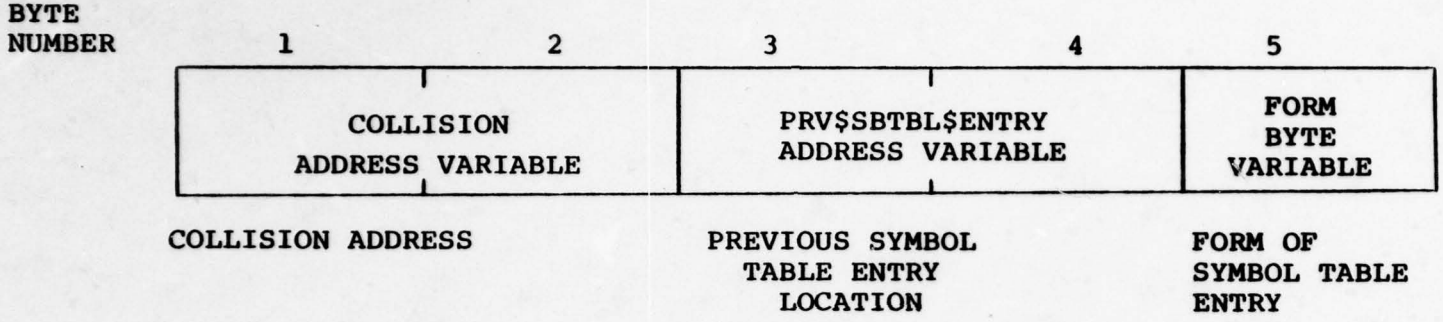
SYMBOL TABLE ACCESS

FIGURE 2

contained in other entries. This latter method of access is required since not all entries in the symbol table have an identifier, called the printname, associated with them.

Each location in the hash table heads a linked list of entries whose printname, when evaluated, results in the same hash value. A zero in any location in the hash table indicates that there are no entries whose printname produces that value. During symbol table construction or access, the global variable PRINTNAME contains the address of a vector whose first element is the length of an identifier in a single byte, followed by the identifier's characters represented in ASCII format. The variable SYMHASH contains the hashcode value which is the sum of the printname's ASCII characters, modulo 128. Entries that produce the same hash code value are linked together in the symbol table by a chain which is accessed via the individual entry's collision field. The chain is constructed in such a way as to have the latest entry constructed at the head of the chain.

Each entry in the symbol table contains a number of fields, some of which are common to all entries, and some of which apply only to particular types of entries. All entries have the same first three fields: the collision field located in the first two bytes; the previous symbol table (PRV$SBTBL$-ENTRY) entry address field located in the third and fourth byte; and the form field (FORM) located in the fifth byte, as shown in Figure 3. The remaining fields are used to uniquely describe each entry's attributes and particular identifying characteristics.



FIRST THREE FIELDS OF A SYMBOL TABLE ENTRY

FIGURE 3

There are eight different types of entries found in the NPS-PASCAL symbol table. Each of these types has a unique three bit code in its form field. The three bit code for constant entries, for example, is 001; the code for variable entries is 011. The remaining bits in the form field describe other particular characteristics of the type involved. These characteristics are described in detail as each type of NPS-PASCAL symbol table entry is presented below.

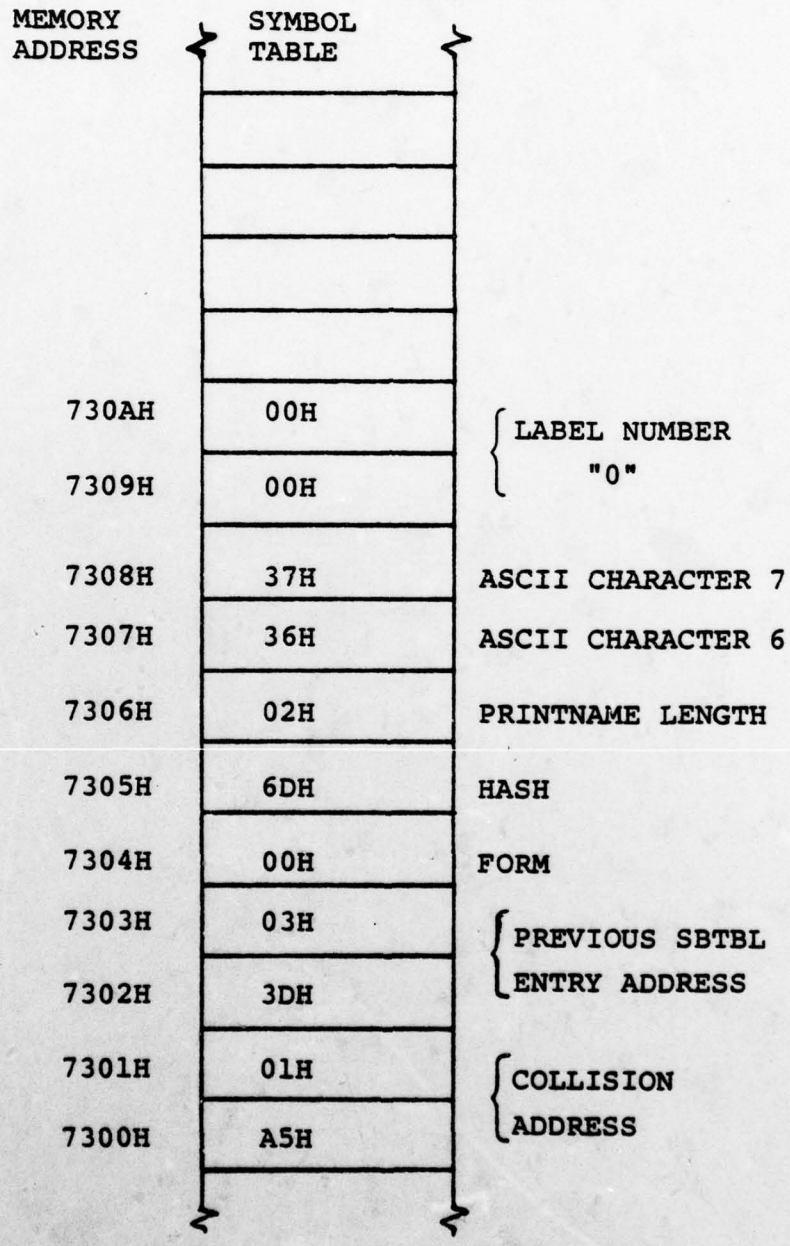
a. Label Entries

The form field of a label entry has the constant byte value of zero. A single byte follows the label's form field containing the hash value of the label's printname. The length of the label follows in the next one byte field. The individual printname characters appear after the length field. A two byte field following the printname characters contains a sequentially generated integer value which is assigned as the label's internal label number. This value is used as the target for branching in the intermediate code. An example of a label declaration with its associated symbol table entry is shown in Figure 4.

b. Constant Entries

The form field of a constant symbol table entry identifies the type of entry, and the particular type of the constant as well. There are five valid types of constants in NPS-PASCAL: an unsigned identifier where FORM = 01H; a signed identifier where FORM = 41H; an integer where FORM = 09H; a real value where FORM = 11H; and a string constant where

LABEL 67;



SYMBOL TABLE LABEL ENTRY

FIGURE 4

FORM = 19H. Following the form field of the constant entry are the printname hash field, length field, and the printname characters.

The value field may consist of another length field and the printname characters in the case of identifier and string constants, or it may contain the internal representation of a constant number (two bytes for integer values and eight bytes for real values). Figure 5 is an example of a constant entry.

c. Type Entries

NPS-PASCAL has two kinds of type entries in its symbol table: simple type entries and type declaration entries. The simple type entry can also be one of two types. It indicates that either one of NPS-PASCAL's standard types is being assigned to the entry, or that a predefined complex type declaration is to be assigned. In the latter case, a simple type entry is made in the symbol table, with a pointer to a type declaration entry. In the former case, one of the following standard types will be assigned the type entry.

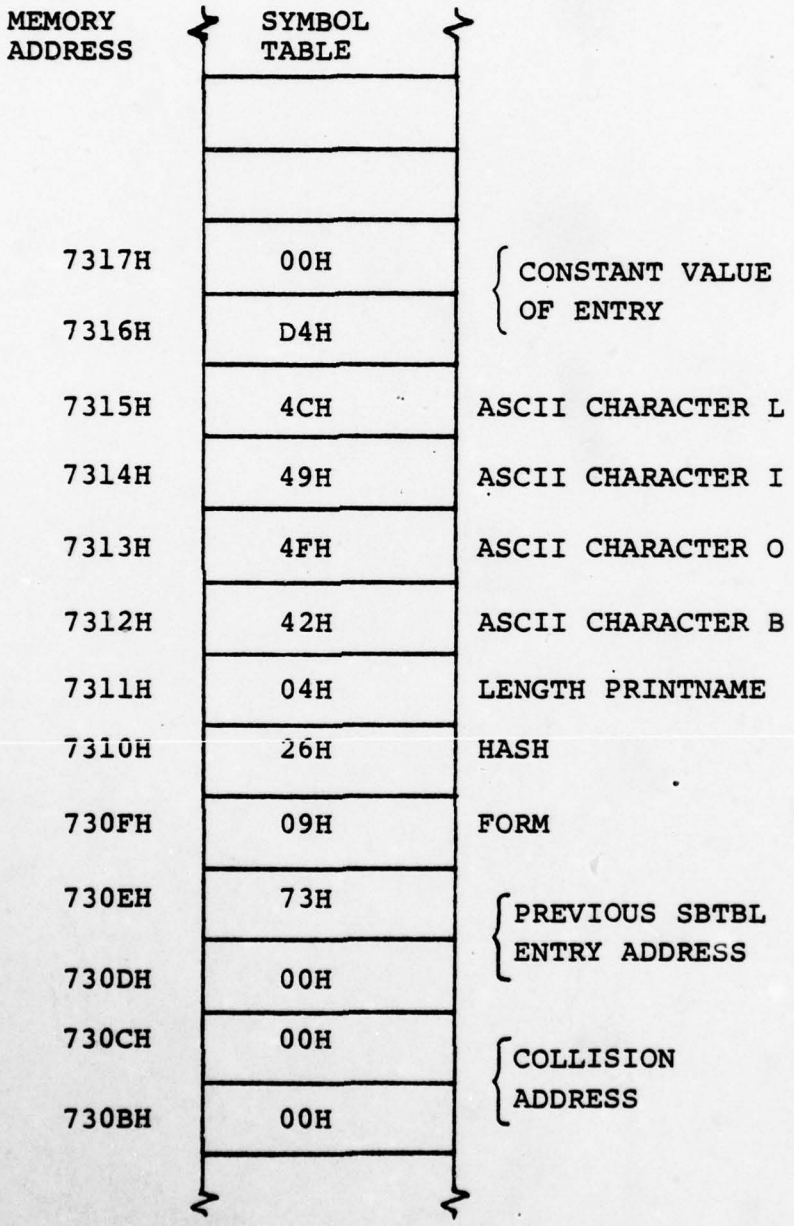
Integer - The values of this type are a subset of the whole numbers whose range is the set of values:

-maxint,-maxint+1,...,-1,0,1,...,maxint-1,maxint

where maxint = 32,767.

real - The values are a subset of the real numbers consisting of fourteen digits multiplied by ten to the -64th power through ten to the +63rd power.

```
CONST BOIL = 212;
```



SYMBOL TABLE CONSTANT ENTRY

FIGURE 5

boolean - The values are denoted by the identifiers "false" and "true," such that false is less than true.

char - The values of this type are the defined set of characters described in reference 8. The following relations hold for CHAR types:

(1) The subset of character values representing the digits 0 to 9 is ordered and contiguous.

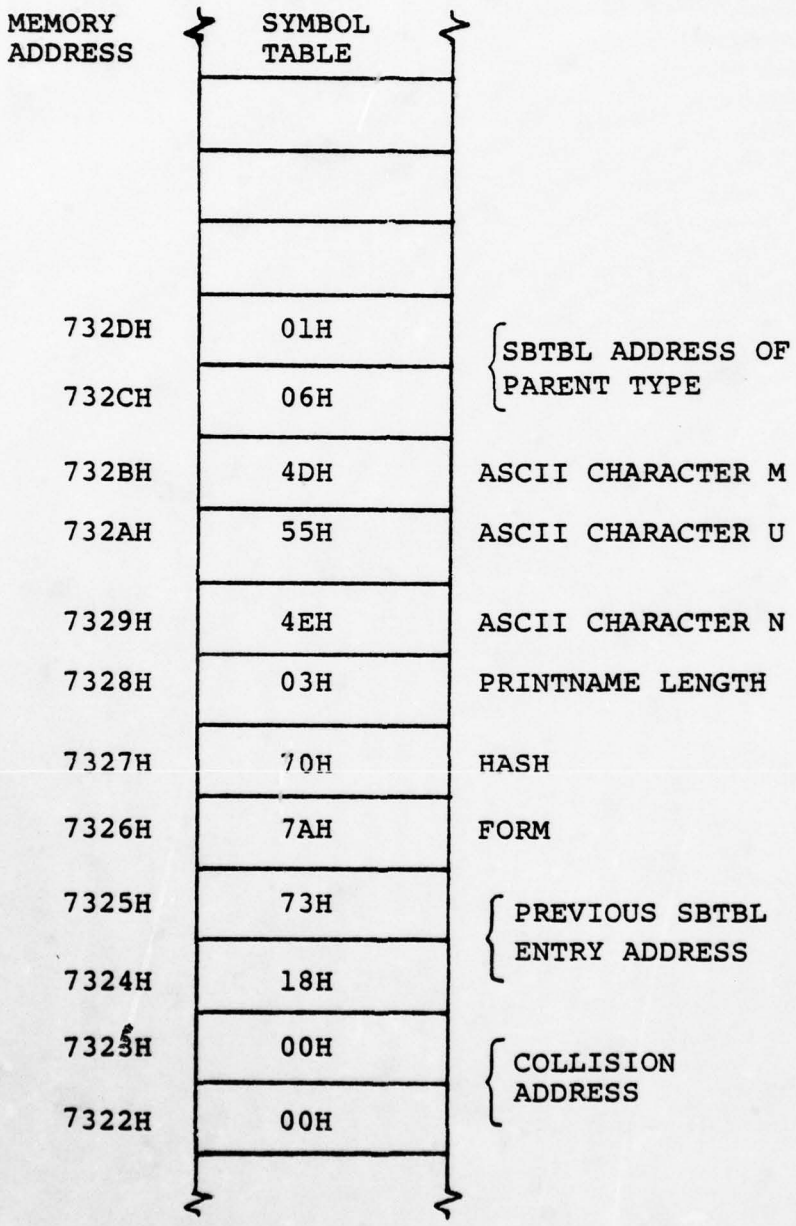
(2) The subset of character values representing the upper case letters A to Z is ordered and contiguous.

(3) The subset of character values representing the lower case letters a to z is ordered and contiguous.

Type declaraiton entries, however, are generated from user defined types found in the source program. It is possible to define a chain of type declarations. An example of this would be an array of the type array which is itself of type integer.

The symbol table entry for a type is as follows. An integer type has the FORM value of 42H, a real type has the FORM value of 4AH, a character type has a FORM value of 52H, and a boolean type has a FORM value of 5AH. A FORM value of 7AH indicates that a type declaration entry must be accessed to determine the complete type of entry. The field following the form is a one byte field containing the hashed value of the printname. The next byte contains the printname's length, which is followed by the printname characters of the type identifier. The last two bytes contain the address of the specified type. An example of a simple type entry is given in Figure 6.

```
TYPE NUM = INTEGER;
```



SYMBOL TABLE SIMPLE TYPE ENTRY

FIGURE 6

There are seven different user definable types in NPS-PASCAL. A type declaration entry is constructed whenever a scalar type, subrange type, array type, record type, set type, file type, or pointer type is encountered.

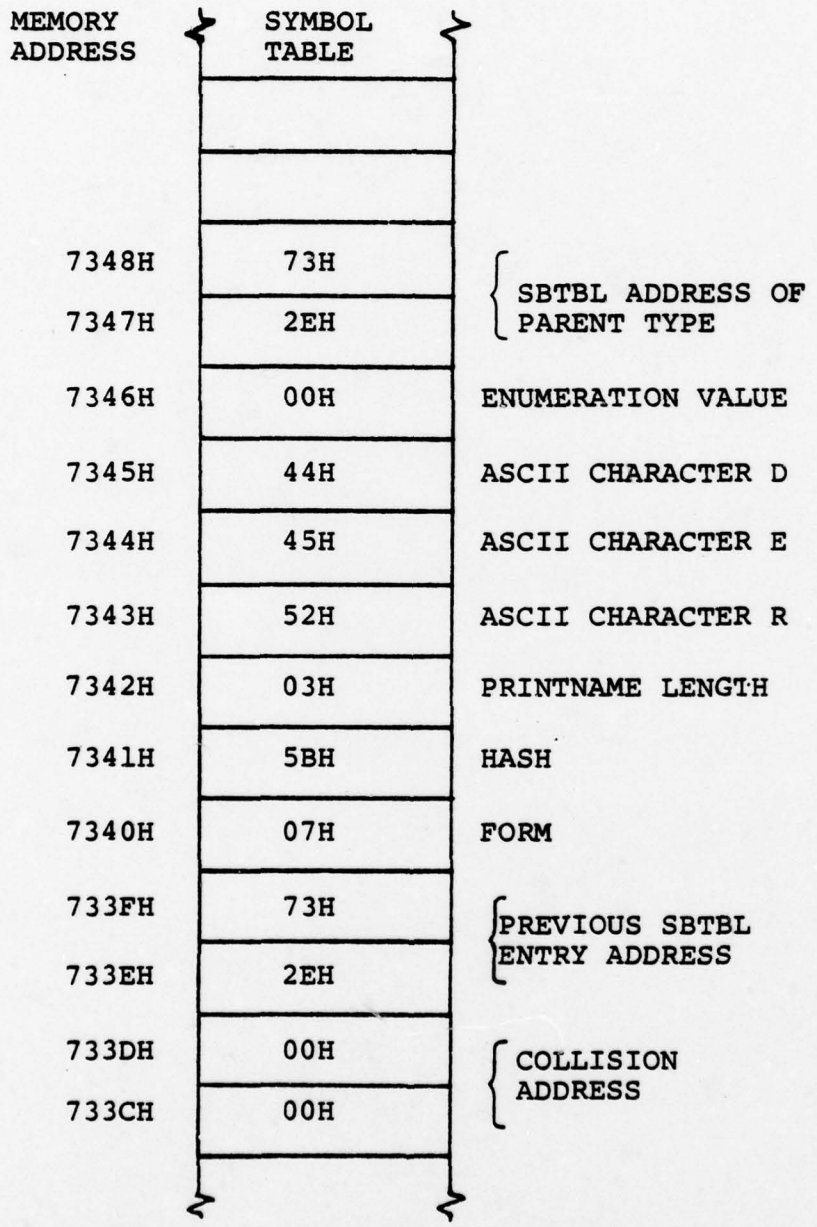
(1) Scalar Types

By their definition a scalar type is an ordered set of values whose identifiers are enumerated to denote their values. The form field entry in the symbol table has the value 07H. Scalar entries are the only type declaration entries that have an accessible printname. Consequently, the next two fields hold the printname hash value and the printname length. The printname characters follow these fields. The next field is a byte value containing the enumerated value of the scalar identifier. The final field is a two byte field storing the symbol table address of the parent type. Figure 7 displays the scalar entry format in the symbol table.

(2) Subrange Types

A subrange type is a duplicate declaration of any other predefined scalar type, integer type, or character type, but with a specified lower and upper bound on its elements. The form field of a subrange entry is OFH for enumerated elements, 4FH for integer elements, and 08FH for character elements. Bytes six and seven store the address of the subrange elements parent type. Bytes eight and nine hold the low value of the range, while the next two bytes contain the high value. The following field is two bytes in length and stores the total number of elements in the range. The final two bytes

```
TYPE COLOR = (RED, WHITE, ...
```



SYMBOL TABLE SCALAR ENTRY

FIGURE 7

hold the displacement vector value, which is utilized during array access. A typical subrange entry is shown in Figure 8.

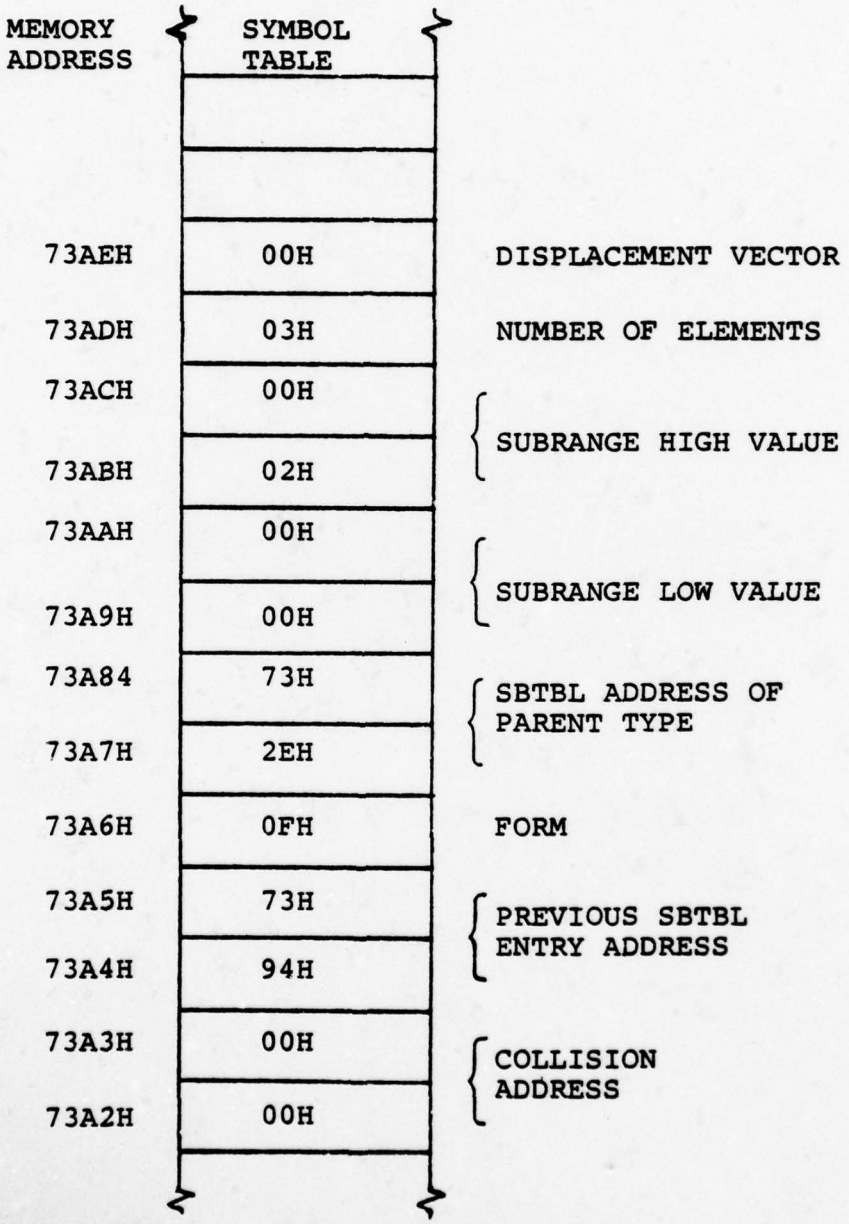
(3) Array Types

The previous type declaration entries in NPS-PASCAL are called simple type entries. They are symbol table entries utilizing a single predefined type. Structured types are compositions of types in NPS-PASCAL. In other words, one or more types are utilized to describe a symbol table entry.

The array type is a structured type consisting of a fixed number of components that are all of the same "component" type. The number of components are specified as a scalar or subrange type and are referred to as the index type (integer and real types are not allowable index types, however, the scalar or subrange type can be of type integer). The component type is of any type.

The symbol table format for an array entry has a form field value of 17H. The following byte specifies the number of indexes, or dimensions in the array. The next two fields are both two bytes long, the first containing the address of the component type; the second containing the total storage requirements in bytes for the array. The eleventh byte holds a value representing the array's component type as determined in Table 1. A two byte field follows with the symbol table address of the array's first dimension. If the array has more than one dimension, two bytes are allotted in the symbol table to store the address of each remaining dimension. Figure 9 shows an array symbol table entry.

```
TYPE PRIME = RED .. BLUE;
```



SYMBOL TABLE SUBRANGE ENTRY

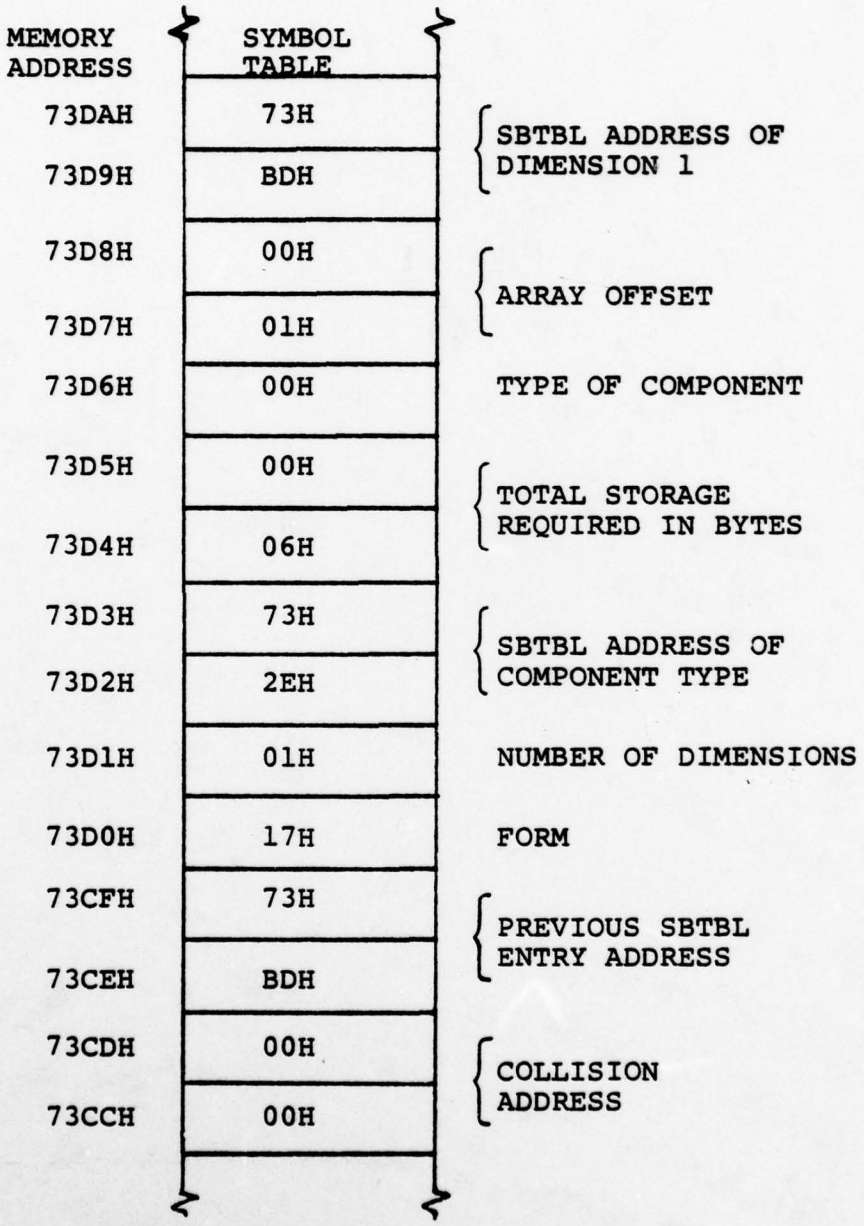
FIGURE 8

Basic Type of Components

Value	Meaning (Type of Component)
00H	Ordinate
01H	Integer
02H	Character
03H	Real
04H	Complex
05H	Boolean

TABLE 1

```
TYPE MIX = ARRAY  [1..6]  OF COLOR
```



SYMBOL TABLE ARRAY ENTRY

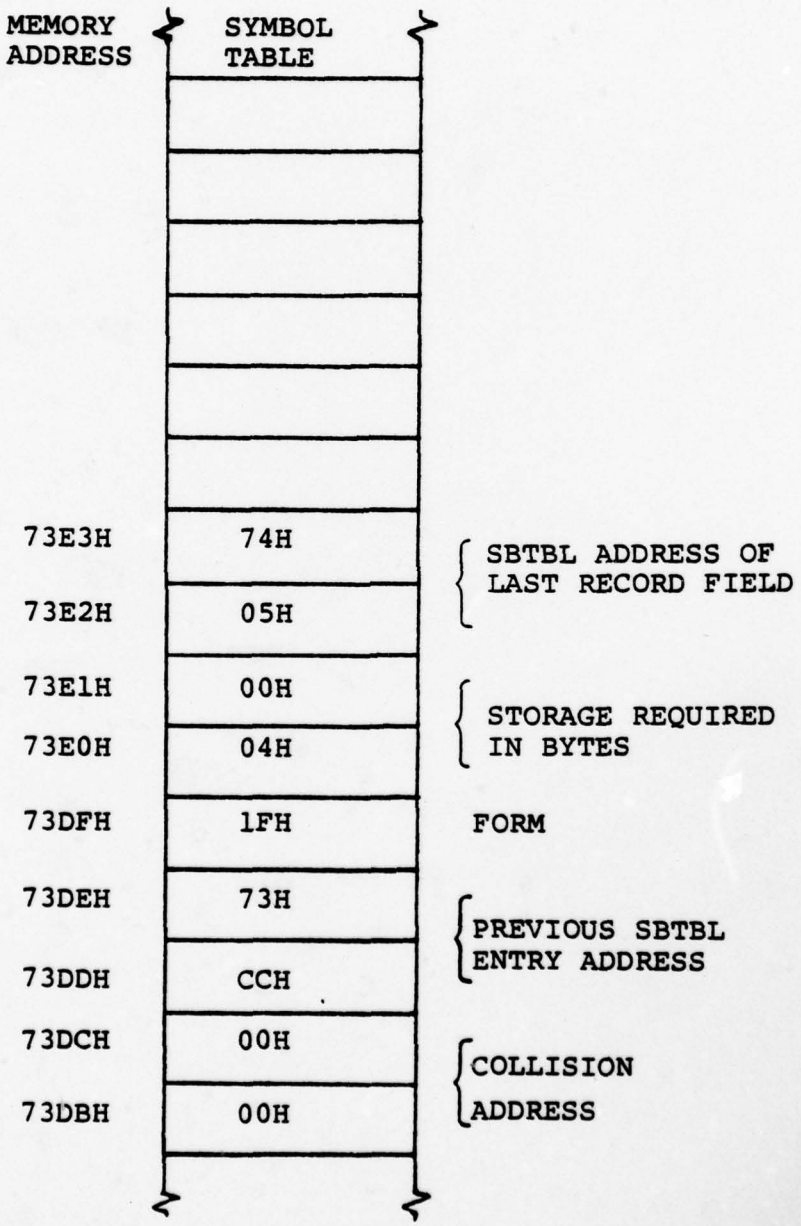
FIGURE 9

(4) Record Types

A record is another NPS-PASCAL structured type. This structure has a fixed number of components, called fields, each of which can be of any defined type. The symbol table entry for a record has a form field value of 1FH. Bytes six and seven contain the storage requirements in bytes for the record. Bytes eight and nine store the symbol table address of the last field contained in the record structure. Figure 10 formats a record entry in the symbol table.

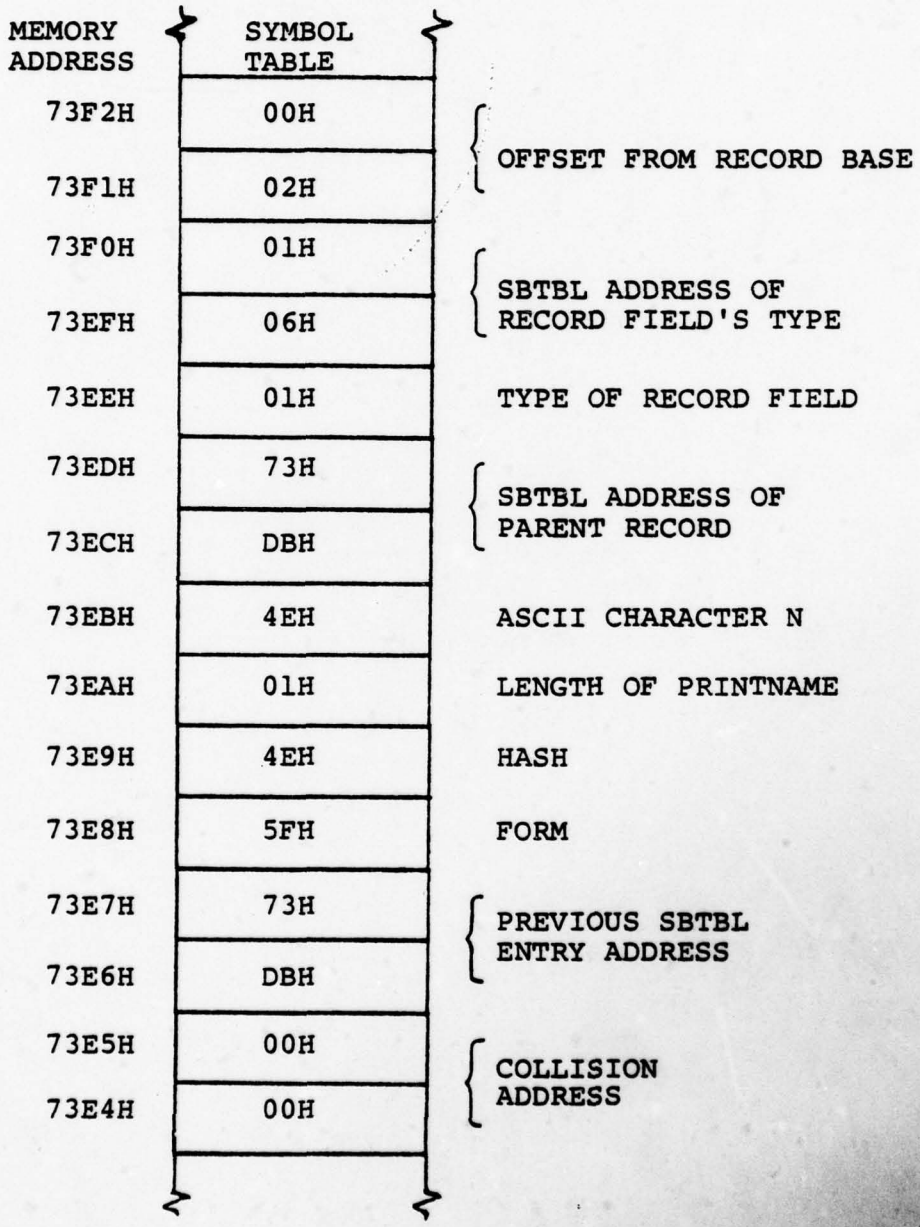
Each record field consists of an identifier and a type, and has a unique format in its symbol table entry. The form field has the constant value of 5FH. The following two fields are byte values for the hash value and the length of the field's printname. The next field holds the printname characters. The address of the parent record is stored in the next two bytes. The following field has a one byte length and is used to store the record field's type. The choice of values to be stored is the same as the list specified for an array's component type. Two more bytes are utilized to store the symbol table address of the type specified. The last field of this entry is two bytes long and holds the offset of the record field from the base of the record. The format of a record field entry is specified in Figure 11.

NPS-PASCAL supports the variant field and tag field constructs of records. These two kinds of record fields have similar symbol table entries as just described above, with the exception that the variant form field is ODFH, and the



SYMBOL TABLE RECORD ENTRY

FIGURE 10



SYMBOL TABLE RECORD FIELD ENTRY

FIGURE 11

tag form field is 9FH.

(5) Set Types

The set structure defines a set of values, which is the power set of a declared base type. The base type is required to be a scalar or subrange type. The set type symbol table entry has a form field value of 27H. The following two bytes contain the symbol table address of the set type identifier. Figure 12 shows a sample set entry in the symbol table.

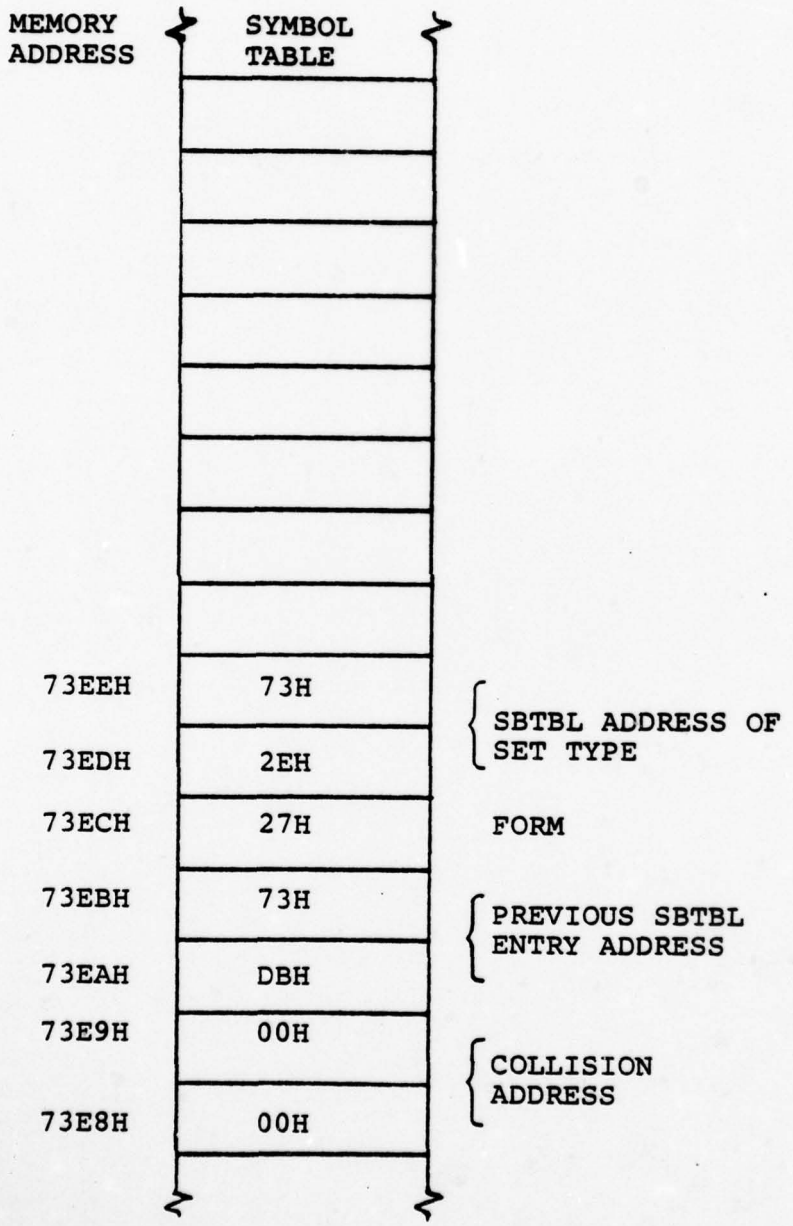
(6) File Types

A NPS-PASCAL structure consisting of a sequence of components, all of the same type, is simply called a file. A file type indicates a natural ordering of the components, whose position in the file defines the sequence. A file type declaration entry in the symbol table has a form field value of 2FH. The symbol table address of the file type's identifier is contained in the next two bytes. The file type format is displayed in Figure 13.

(7) Pointer Types

NPS-PASCAL supports dynamic variables which are generated without any correlation to the static structure of the program. These variables are assigned a special type called pointer type. The symbol table entry for this type declaration is shown in Figure 14. The form field value is set to 37H, while bytes six and seven of the entry hold the symbol table address of the pointer type identifier's entry.

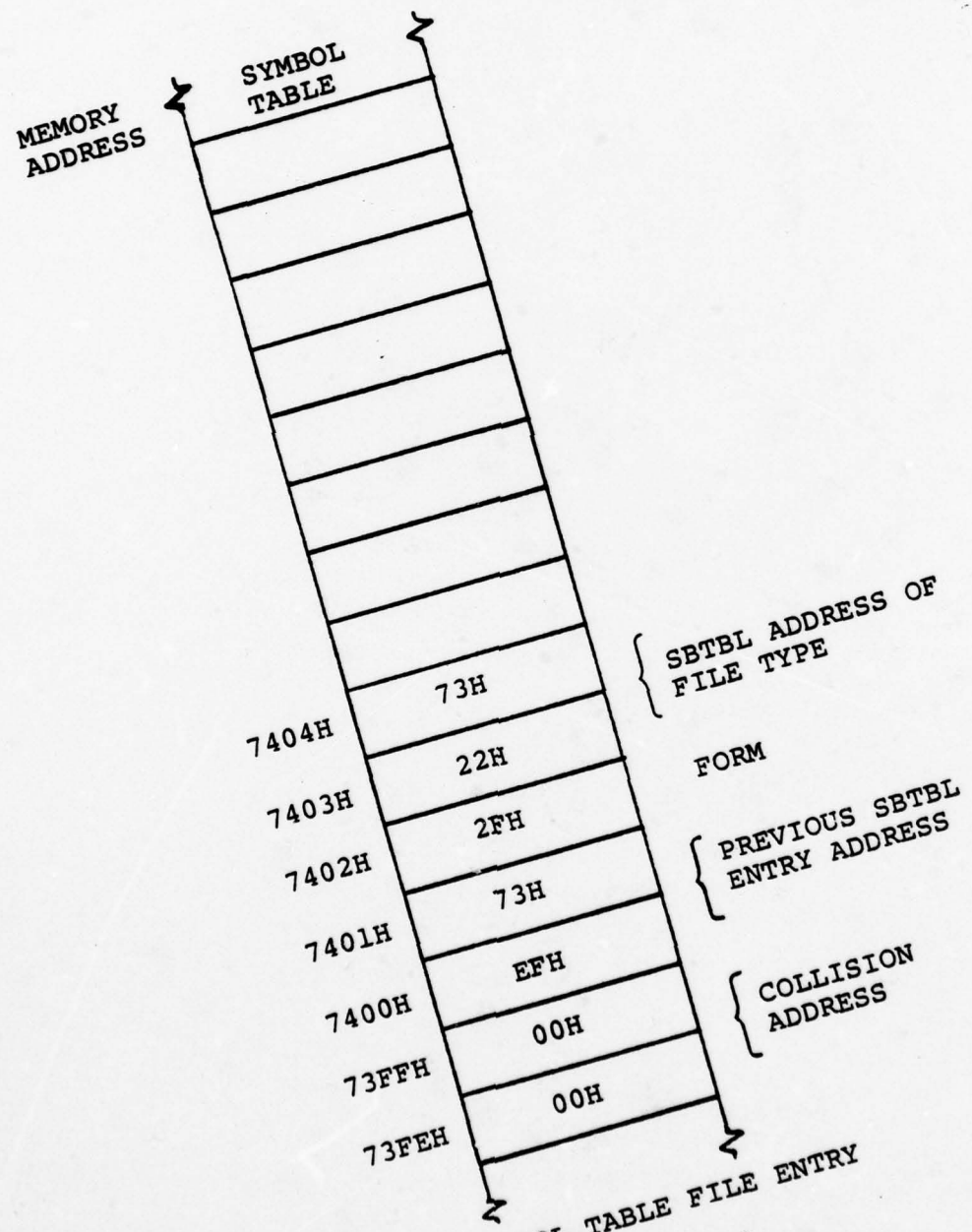
TYPE FLAG = SET OF COLOR



SYMBOL TABLE SET ENTRY

FIGURE 12

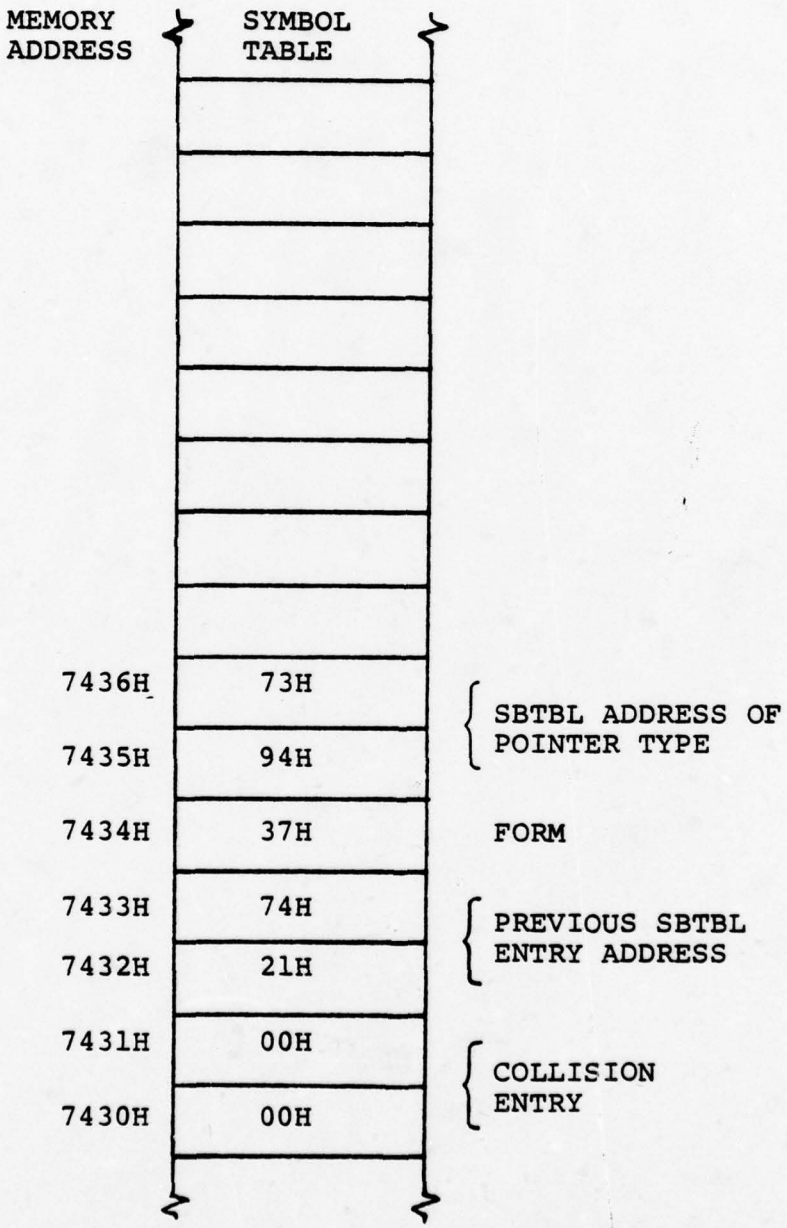
VAR DATA: FILE OF NUM



SYMBOL TABLE FILE ENTRY

FIGURE 13

```
VAR P : ↑PRIME
```



SYMBOL TABLE POINTER ENTRY

FIGURE 14

d. Variable Entries

Each variable declared in a NPS-PASCAL program is inserted in the symbol table. The form field of the variable entry contains a value which describes the type of the program variable. The values for this field and their associated types are shown in Table 2. Following the form field are the fields containing the variable identifier's printname hash value, length, and the printname characters. A two byte field which contains the variable's starting address in memory appears after the printname characters. This address is an offset from the base of the variable area, called the Program Reference Table (PRT), assigned by the NPS-PASCAL Code Generator. The variable's type determines the length, or number of bytes assigned to the variable in the PRT. The compiler keeps a count of the total amount of storage and passes this value to the Code Generator at the completion of a successful program compilation. The Code Generator subsequently converts the relative addresses in the intermediate code to absolute addresses in the final target machine code. An example of a variable entry is given in Figure 15.

e. Procedure and Function Entries

Every procedure and function in a NPS-PASCAL program has an associated entry in the symbol table. In the case of a procedure entry, the form field is assigned the value 04H. The hash value, length of printname, and the printname characters fields immediately follow the form field. A one byte field follows the printname and stores the number of parameters

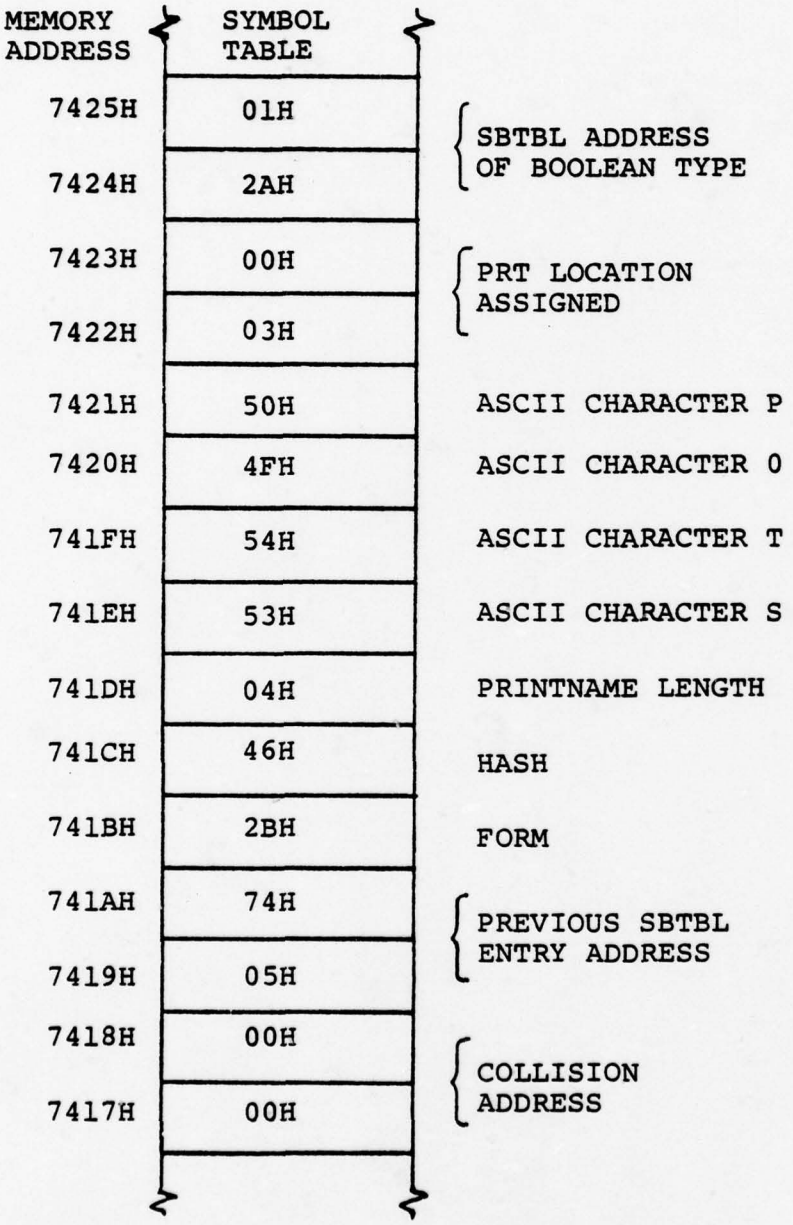
The Form Field of Variable Entries

Value	Meaning (Type of Variable)
03H	Scalar-Ordinate
0BH	Integer
13H	Character
1BH	Real
23H	Complex
2BH	Boolean

FORM FIELD OF VARIABLE ENTRIES

TABLE 2

```
VAR STOP : BOOLEAN;
```



SYMBOL TABLE VARIABLE ENTRY

FIGURE 15

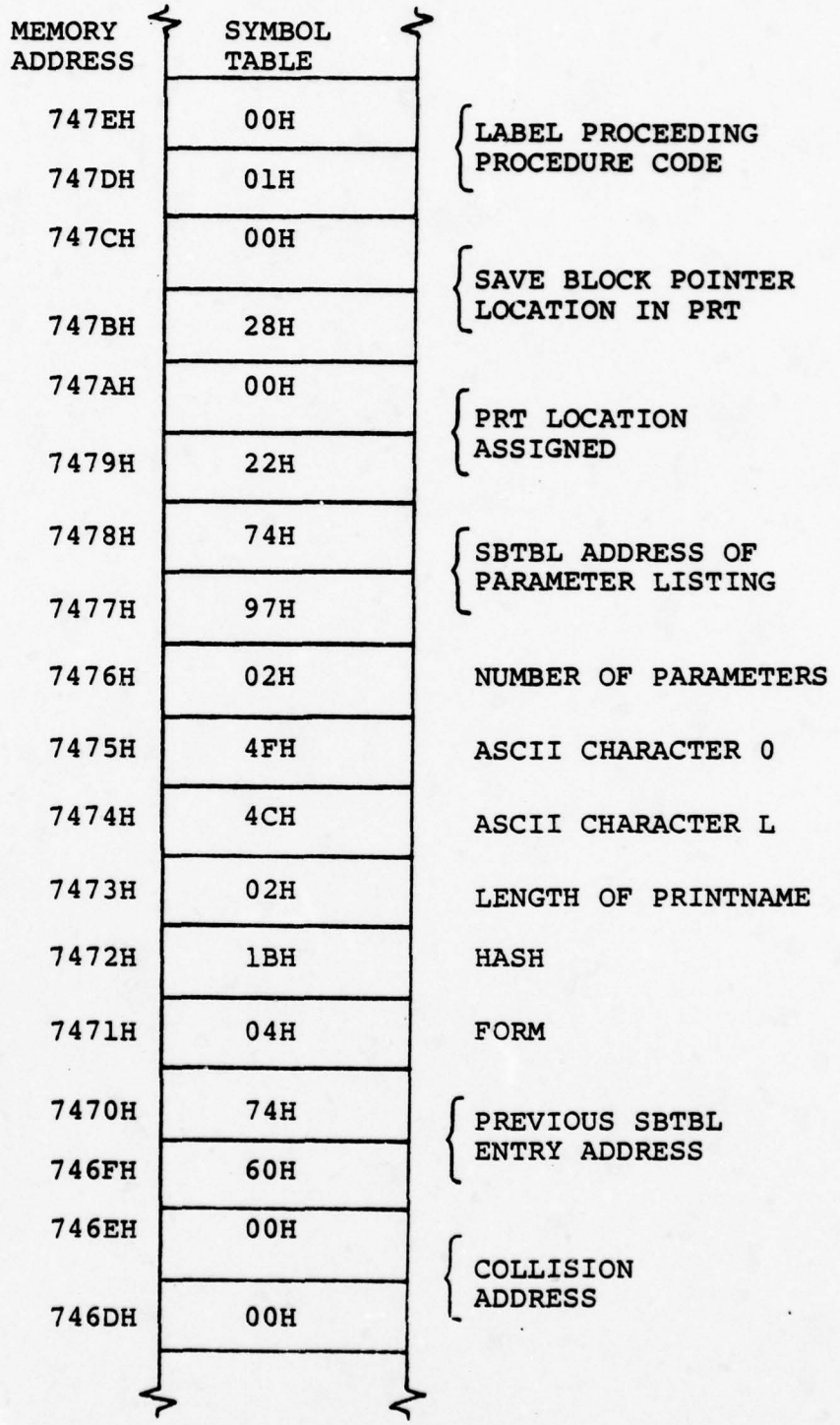
associated with the procedure. A two byte field is next, storing the symbol table location of a listing of the procedure's parameter types. This listing is referenced by the compiler to ensure proper parameter mapping, and is located immediately after the final procedure parameter entry in the symbol table. Following the parameter types address field in a procedure entry, are three more two byte fields. The first field gives the Program Reference Table (PRT) address assigned to the procedure identifier. The second field gives the PRT address assigned to the procedure save block pointer (SBP). The SBP construct is based on a similar construct in Algol-E (10) to permit recursive subroutine calls. The final field in the entry holds a label value that must be branched to when the procedure is invoked. Figure 16 illustrates the format of this entry.

A function entry in the symbol table duplicates a procedure entry with two exceptions. A function entry has a form field value of 05H; and one byte field is added at the end of the entry to designate the type of the function. Function type values are similar to the variable types specified in Table 2. A sample function entry appears in Figure 17.

(1) Formal Parameters

Formal parameters provide a mechanism that allows a procedure or function to be repeated with various values being substituted. The formal parameters are declared in the procedure or function declaration and can be of four types: value parameters, variable parameters, procedure para-

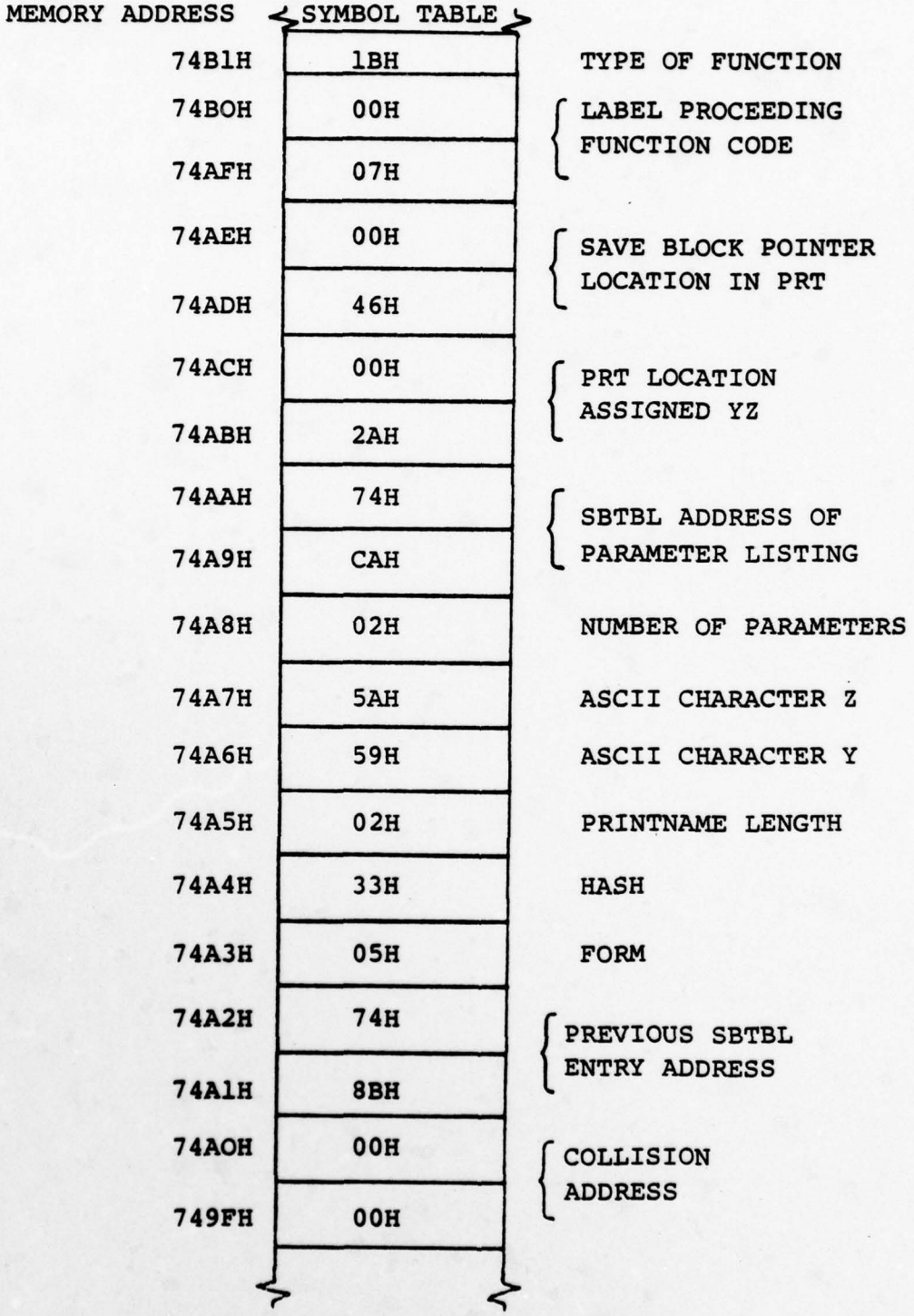
PROCEDURE LO (X:INTEGER; VAR Y : INTEGER);



SYMBOL TABLE PROCEDURE ENTRY

FIGURE 16

FUNCTION YZ(F,A:REAL):REAL;



SYMBOL TABLE FUNCTION ENTRY

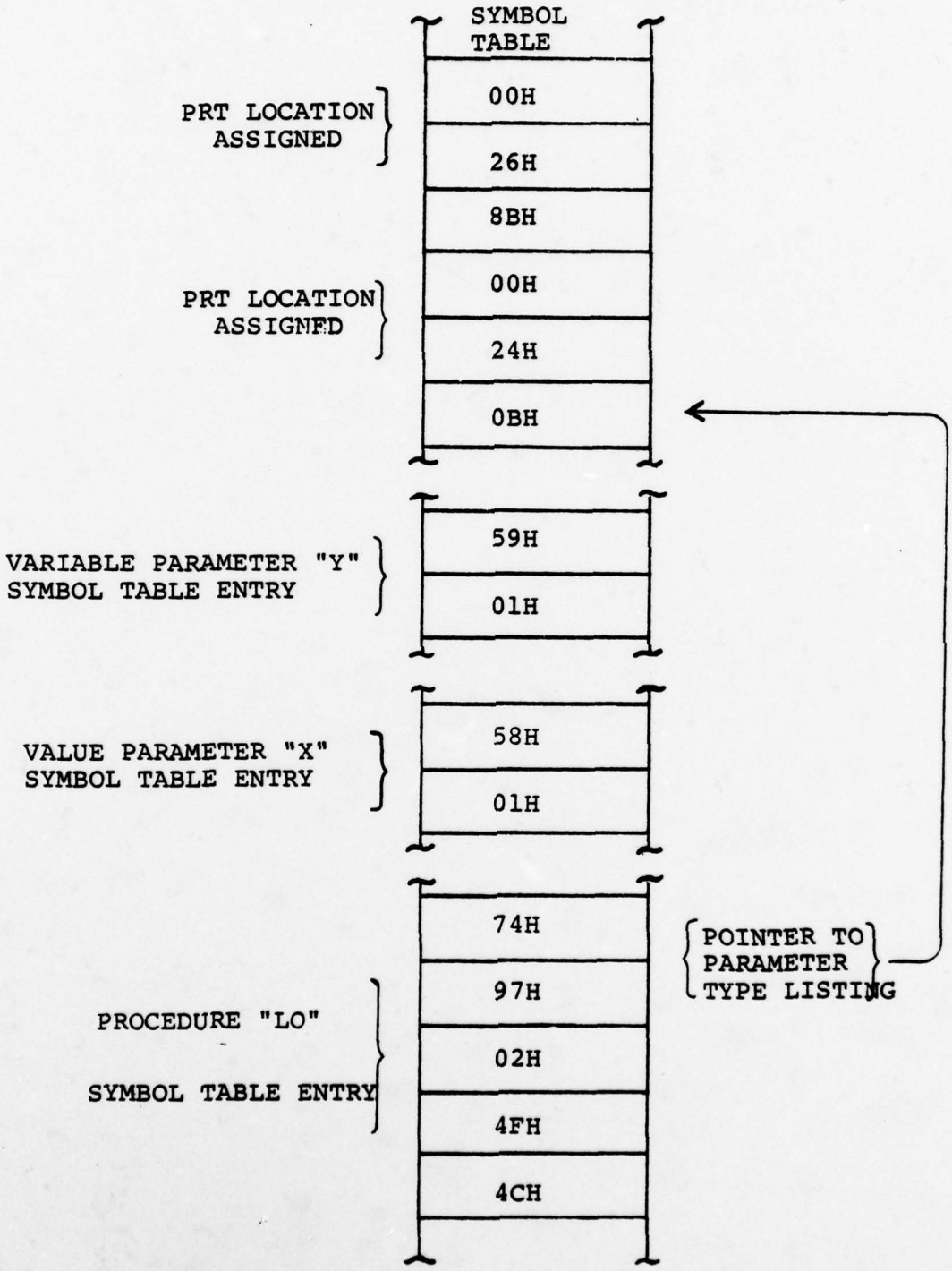
FIGURE 17

meters, and function parameters. Each declared parameter has an associated symbol table entry. A value parameter entry has exactly the same format as a variable entry. A variable parameter entry also duplicates a variable symbol table entry, with the exception of the form field. The high order bit of the form value is set to one for all variable parameters. Procedure and function parameters are entered as described above for procedure and function symbol table entries.

Figure 18 displays a sample series of symbol table entries starting with a procedure entry and followed by various formal parameter entries. Note that the final few bytes show the listing of the procedure's parameter types that will be utilized for mapping actual parameters to the formal parameters.

f. Symbol Table Construction Procedures

Several standard construction procedures were developed for the manipulation of the symbol table. The procedure ENTERVARID is used by all routines which construct a symbol table entry containing an accessible printname. This procedure calls ENTER$LINKS to assign the collision and previous symbol table entry address fields. The procedure ENTERPNID is then called to insert the printname hash value, printname length, and the printname characters. Control returns to ENTERVARID where the initial form value of the entry is set. When no printname is associated with the symbol table entry, ENTER$COMPLEX$TYPE is called to zero the collision field, enter the previous symbol table entry address, and to set the



PROCEDURE AND PARAMETERS SYMBOL TABLE ENTRY

FIGURE 18

form field specified when ENTER$COMPLEX$TYPE was invoked. Various other construction routines must be called to add additional descriptive fields for the particular entry under construction.

g. Symbol Table Access

Symbol table access in the NPS-PASCAL compiler is accomplished through the use of standard lookup procedures, and pointers contained within entries. Access is also dependent upon the current block level of the input program. Because Pascal is a block structured language, the rules of global and local variables are followed. Whenever a procedure or function declaration is recognized, the block level of the program is immediately incremented, thus permitting variable names to be reused in this new level. Any variable declared in a lower level can be accessed in a higher level provided that the variable name is not a duplicate and that the lower level variable is in fact global to the current higher level. Obviously, identical variable names cannot be declared in the same block level. When the compiler recognizes a complete procedure or function, the block level is decremented by one and the variables local to that block have their entries in the HASH$TABLE deleted.

The procedure LOOKUP$ONLY can be called with the address of a printname as a parameter. The procedure calls CHECK$PRINT$NAME to compare the symbol table entry's printname with that of the parameter. The hash table index of the parameter is used along with the symbol table collision

fields to access the correct entries in the table. The procedure LOOKUPPNID was designed to accomplish the same task as LOOKUP$ONLY with the additional features of checking the form field of the entry with a second parameter, and to check the declared block level of the printname. If either procedure succeeds in matching a symbol table entry, the variable LOOKUP$ADDR is set to the starting address in the symbol table of the matched entry, and the value TRUE is returned to the calling routine.

Upon compilation of a program, successful or not, the symbol table is copied into a separate file of type "SYM". The NPS-PASCAL user assistance program SYMBOL$TABLE may then be invoked to print out the contents of the program's symbol table.

2. Built-in Symbol Table Entries

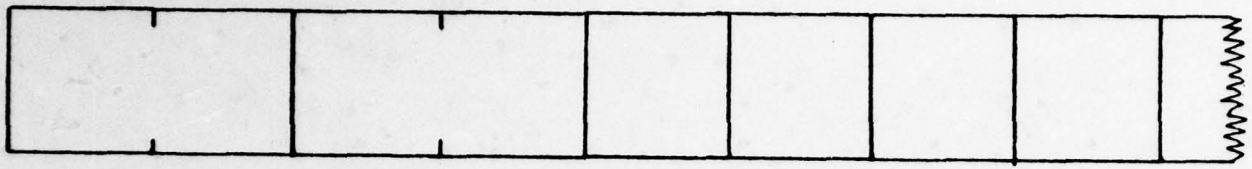
Special mention must be made of the symbol table entries that are predefined, or built into the NPS-PASCAL compiler. These entries contain type entries, file entries, procedure entries, function entries, and constant entries that make up the NPS-PASCAL standard identifiers. The standard identifiers are declared in the BUILTINTABLE.

The entries in BUILTINTABLE follow much the same construction rules as those of the compiler generated symbol table. The main difference is that the built-in symbol table has to be entered by hand. Consequently, the table was located at the first available memory location (106H) for ease of implementation and to simplify the addition of other built-in

entries. Appendix C lists a table of the standard identifiers found in BUILTINTABLE, and their usage is detailed in the NPS-PASCAL User's Manual.

All BUILTINTABLE entries have an accessible printname. Consequently, the first six fields of standard entries have the format shown in Figure 19. Built-in types and built-in files contain only these six fields. Built-in constants have an additional field that stores the constant's value. Built-in functions have a one byte field following the printname characters that holds a sequential number that uniquely identifies the function. The type of the function is stored in the following one byte field. The next byte holds the number of parameters that the built-in function has. Depending on this value, a sequence of fields follow, all of which are one byte in length, ranging from one to the number of parameters, storing the type required of the actual parameters. Standard function type fields are assigned values in accordance with the format specified in TABLE 1. Built-in procedures also have the one byte field holding the unique sequential number that identifies the procedure. However, the number of procedure parameters and their expected types are not stored in the built-in entry because most NPS-PASCAL standard procedures have a variable number of parameters. Consequently, various procedures check a built-in procedure's actual parameters to ensure proper parameter mapping.

Since BUILTINTABLE entries are entered manually, certain entries must be made in the compiler's HASH$TABLE.



FIRST SIX FIELDS OF A STANDARD ENTRY IN BUILTINTABLE

FIGURE 19

This table is used to store the starting address of either a BUILTINTABLE or SYMBOL$TABLE entry based on the entry's hash value. Procedure INITIALIZESYMTBL stores the appropriate BUILTINTABLE address at location HASHTABLE(SYMHASH), where SYMHASH is the hashed value of the printname entry stored in the fourth field of the standard entry.

Many of the standard functions, and their parameters, in NPS-PASCAL can be of type integer or real. In this case, the function type and parameter type fields were assigned to the value 13H. This value serves as a flag to the compiler to permit either type integer or type real values to be semantically acceptable. Similarly, standard functions SUCC and PRED and their single parameter can be of any type except real. Therefore, their type fields were given the value OF3H which alerts the compiler to accept any type, except real, in the evaluation of these two functions.

Further standard identifiers may be added to NPS-PASCAL, provided the above entry specifications are satisfied and that an associated entry is made in the HASH$TABLE.

E. PARSER

The parser is a table driven automation and is modelled after the ALGOL-M (9) and BASIC-E (12) parsers. The LALR(k) parser generator (5) produced the required parse tables and the vocabulary table, VOCAB, which together with the parse stacks serve as the major data structures in the parser. The parser operates by receiving tokens from the scanner, analyz-

ing them to determine if they are part of the NPS-PASCAL grammar, and then accepts or rejects the token in accordance with the syntax of NPS-PASCAL. If the token is accepted, one of two actions is taken. The parser may stack the token and continue to request tokens in the lookahead state, or it may recognize the right part of a valid production and apply the production state which results in a stack reduction. If the parser rejected the token, or it determines that the token received does not constitute a valid right part of any production in the NPS-PASCAL grammar, a syntax error will be printed at the console and the RECOVER procedure is called.

RECOVER is a procedure that permits continued program compilation in spite of the occurrence of a syntactical error. It causes the parser to back up one state and attempts to continue parsing from that state. In the event of failure, the parser continues to back up until the end of the currently pending reduction is encountered. At that point the invalid token is bypassed and an attempt is made to parse the following token. This process continues until an acceptable token is found.

The parse stacks in NPS-PASCAL consist of a state stack and eight auxiliary stacks. The auxiliary stacks are parallel to the parse stack and are used to store information needed during code generation. The parse stacks include:

BASE$LOC - stores the symbol table address of the current identifier;

FORM$FIELD - stores the form value of the current identifier as stored in the symbol table;

TYPE$STACK - stores the type value of an identifier;

PRT$ADDR - stores the assigned PRT address of an identifier;

LABEL$STACK - stores label values to be utilized with branching instructions;

PARM$NUM - stores the number of formal parameters associated with a procedure or function;

PARMNUMLOC - stores the symbol table address of the list of formal parameter types associated with a procedure or function;

EXPRESS$STK - stores the type value of an expression.

F. CODE GENERATION

The parser not only verifies the syntax of the source statements, but also controls the generation of the intermediate code by associating semantic actions with production rules. When a reduction takes place, the SYNTHESIZE procedure is called with the production number as a parameter. The SYNTHESIZE procedure contains an extensive case statement keyed by the production number to perform the appropriate semantic actions. The syntax of the language and the semantic actions for each reduction are listed in Appendix D.

A key element in understanding the compiler is a knowledge of NPS-PASCAL storage structures, the diverse operators, the employment of procedures and functions, and the communication routes between the compiler and the user. These particulars are elaborated below, along with a description of the compiler's pseudo operators, to assist in understanding

NPS-PASCAL compiler constructs and to explain the logic used in generating the intermediate code that will, in turn, be used to generate the target 8080 machine code.

1. Storage Space Allocation

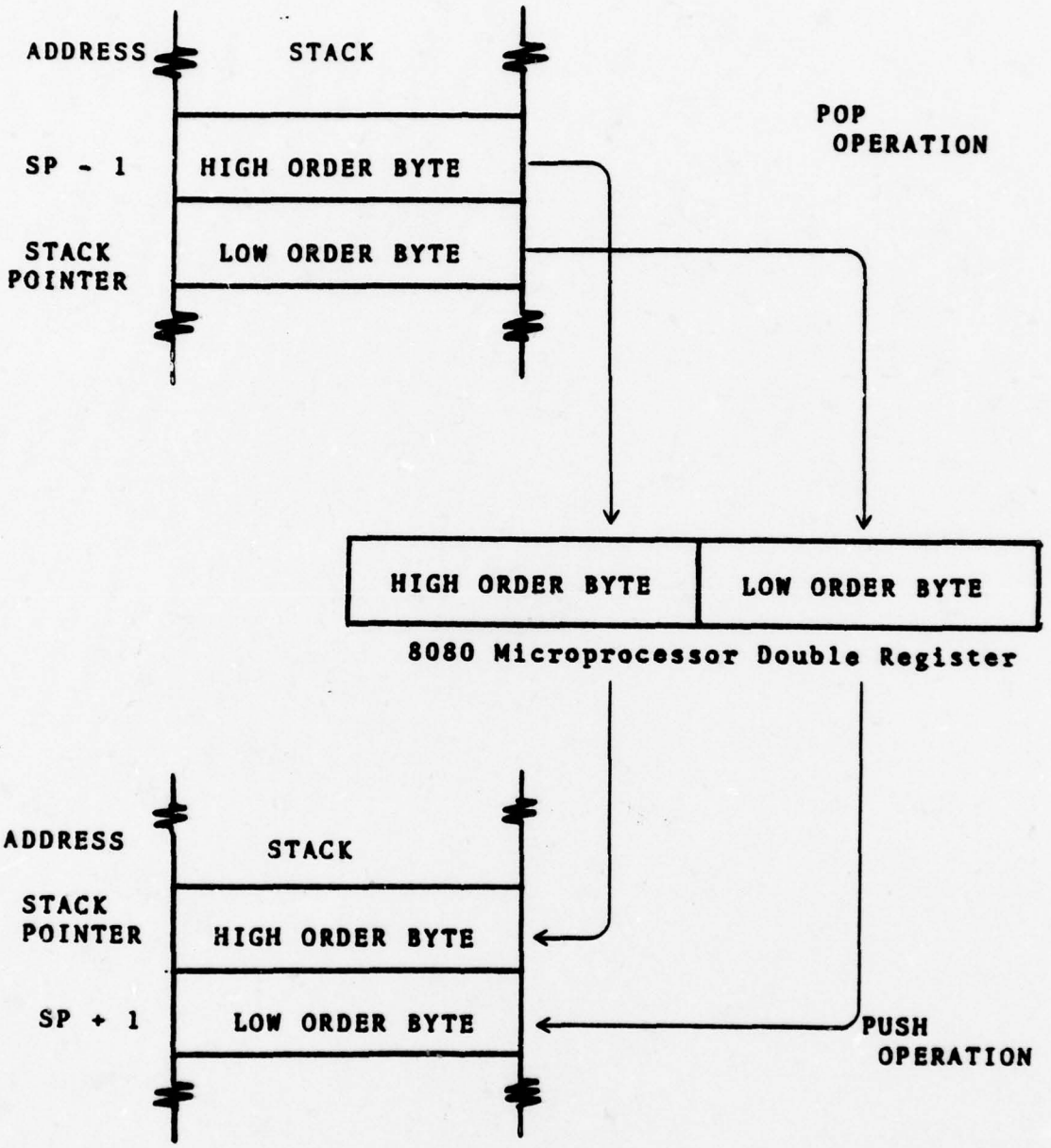
Allocation of storge space in NPS-PASCAL is dependent on the type of data encountered. For each program variable requiring storage space, the compiler specifies the number of bytes to be set aside, and keeps a count of the total storage allocated. The total count is then passed to the Code Generator for the purpose of establishing the size of the Program Reference Table.

a. Byte Data

Byte data items are stored in a single byte location in memory. These data items may represent characters, numbers, or boolean data.

b. Integer Data

Integers are represented by two byte values and are stored in memory with the high order byte preceding the low order byte of the integer number. This storage process follows the processing requirements of the 8080 Microprocessor (10) to complete moves of data from memory, or the stack, into the processor double byte registers at program execution time. An example of the execution time POP and PUSH operation is shown in Figure 20. Integers are represented in two's complement form, with the high order bit acting as the sign bit. A zero high order bit indicates a positive integer value and a one indicates a negative value.



POP AND PUSH OPERATIONS

FIGURE 20

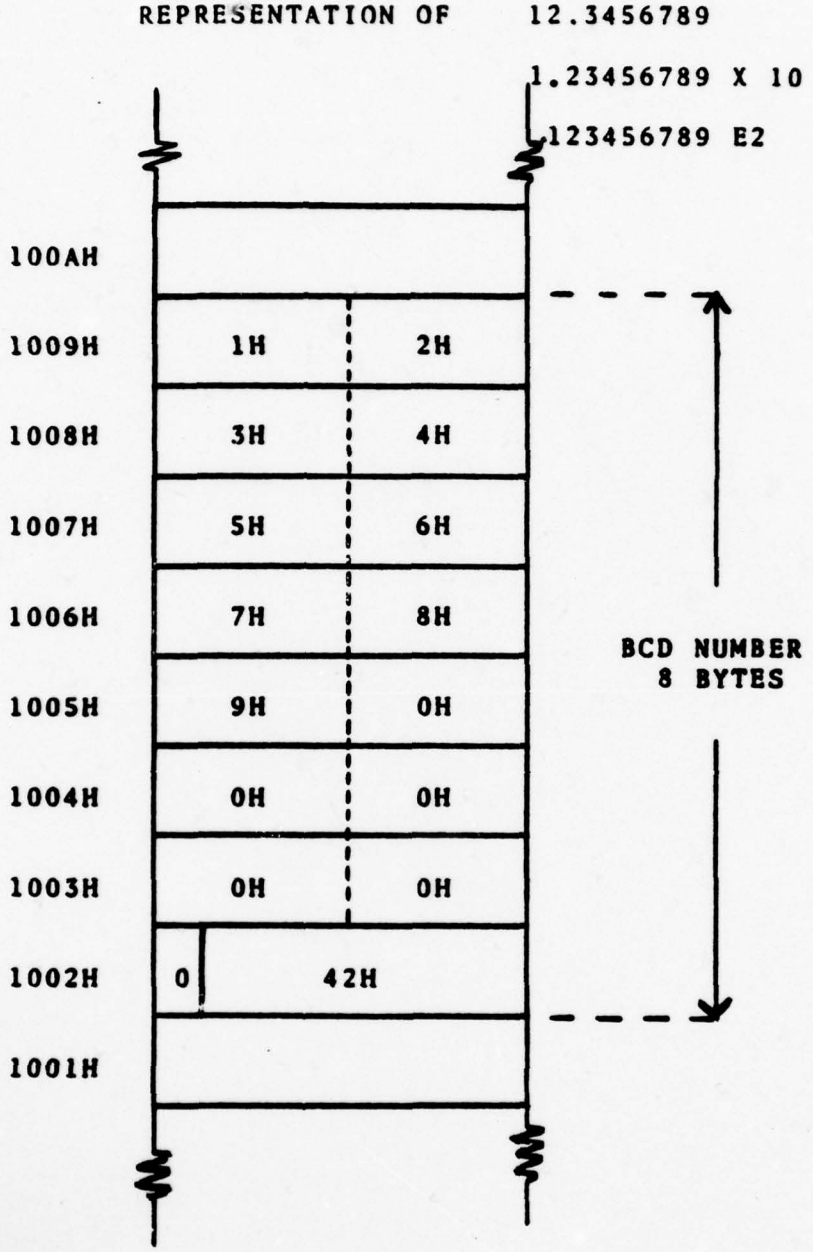
c. Real Data

Real numbers in the NPS-PASCAL compiler are represented in binary coded decimal (BCD) format. Every real number is represented by 14 digits and is stored in eight contiguous bytes. When loading a BCD value in the execution stack, the byte, located at the lowest memory address location, contains the sign of the number along with the sign and magnitude of the exponent. Succeeding bytes represent two decimal digits and are ordered in a backwards fashion. The byte that is the closest to the exponent byte represents the last two digits of the number, while the last byte of the number contains the first two digits. Figure 21 displays a BCD number stored in memory.

The exponent byte in a BCD number uses the high order bit to indicate the sign of the number -- a high order one indicating a negative number, while a zero represents a positive number. The remaining seven bits are used to represent the exponent and its sign. A bias of 64 is used for the exponent representation. Values greater than 64 depict a positive exponent; values less than 64 depict a negative exponent; and the exponent result equals the difference between 64 and the value. This reference point allows a range of exponent values from -64 to +63. The BCD number always assumes the decimal point is positioned before the first digit.

d. String Data

Strings are stored sequentially in NPS-PASCAL. The first byte of the string stores the string length, thereby



BCD NUMBER IN MEMORY

FIGURE 21

limiting strings to a maximum of 255 bytes. Immediately following are the ASCII characters that compose the string.

2. Arithmetic Operations

a. Logicals

Logical operations, or boolean operations, act on byte values of zero and one only. A zero value indicates a false condition, while a non-zero value indicates true. Logical operations requiring comparison between two elements returns the resulting value of the operation in the TRUE or FALSE form.

b. Integers

Arithmetic operations with integers are performed by taking the top two values from the execution stack, placing them in the double byte registers in the 8080 Microprocessor, and carrying out the requested operations. Integer arithmetic operations include addition, subtraction, multiplication, division with truncation, modulo division, logical comparisons, and transformations to BCD format. All computation results, except for transformations, are returned to the execution stack in the two byte integer format. Any relational operation on two integer values is carried out in NPS-PASCAL, in accordance with the rules for integer arithmetic.

c. Reals

Real arithmetic operations are more complex than those with integers due to the nature of the BCD format. The process is similar to that of integers, however, where real numbered pairs are moved into the 8080 registers. The

required operation is applied, and the resulting value is sent back to the NPS-PASCAL stack in its eight byte BCD format. NPS-PASCAL real values also follow the rules of integer arithmetic whenever two real values have a relational operator occurring between them.

3. Set Operations

NPS-PASCAL supports the three set operators: set union; set difference; and set intersection. As before, 8080 registers receive the two set operands located at the top of the stack. The set operation is performed and the resulting set value is returned to the top of the stack. The relational operators of set equality and inequality, set inclusion, and set membership were not implemented in this version of NPS-PASCAL.

4. String Operations

The relational operators of equality and inequality have been implemented for strings. The remainder of the relational operators denote lexicographic ordering according to the character set ordering, and have not been implemented in this version of NPS-PASCAL.

5. Procedures and Functions

Procedures and Functions, also called subroutines, give PASCAL the ability to display program segments as explicit subprograms. The only difference that exists between a procedure and a function is that a function returns a value to the top of the execution stack after it is invoked, while a procedure does not. This means that a function call

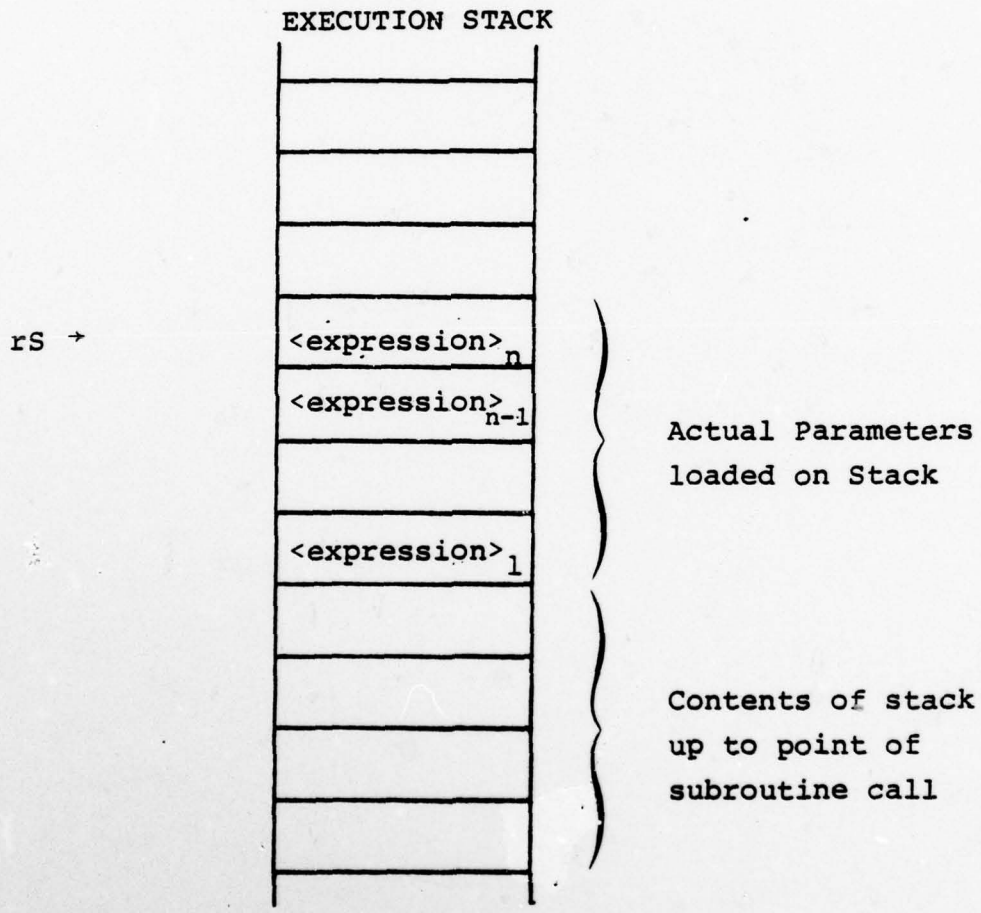
actually represents an arithmetic expression. Procedure calls, however, stand alone as a program statement in NPS-PASCAL. Due to language extensions, NPS-PASCAL allows compilation of separate functions or procedures as complete programs. These EXTERNAL programs can then be called by any other NPS-PASCAL program.

a. Invocation

Procedures and functions can be invoked with zero or more actual parameters. The list of actual parameters are substituted into the corresponding list of formal parameters declared in the procedure or function definition. If the formal parameter is a variable parameter, the actual parameter has to be a variable. Should the formal parameter be a value parameter, then the actual parameter can be an expression -- provided the expression type matches the formal parameter type. For procedure and function formal parameters, the actual parameter must be a procedure or function identifier. Actual parameter types are checked against formal parameter types, stored in the symbol table, during program compilation. The method of passing actual parameters values is via the execution stack, as shown in Figure 22. The procedure or function's code location is generated in the form PRO <label> , where PRO is a mneumonic meaning branch to subroutine, and <label> is the label value stored in the subroutine's symbol table entry.

b. Storage Allocation

All parameters and variables declared within a



STACK CONTENTS AT TIME OF SUBROUTINE CALL

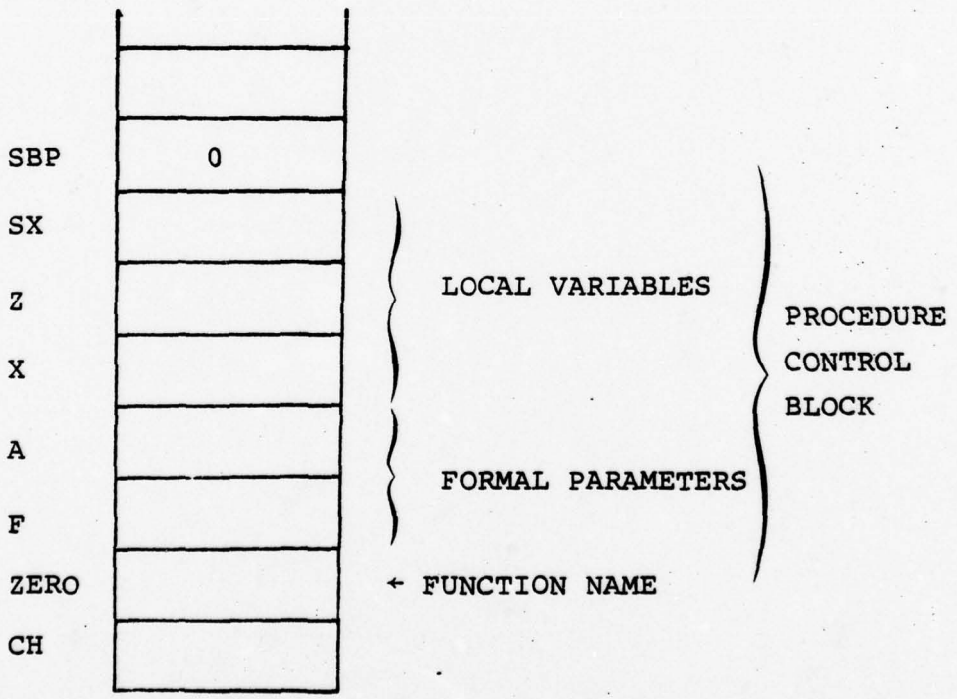
FIGURE 22

procedure or function are assigned a location in the PRT. These locations immediately follow the PRT location of the procedure or function identifier. Upon recognition of a complete subroutine, another PRT location is allocated. This location is called the Save Block Pointer (SBP) for the subroutine. The PRT locations extending from the subroutine's identifier location through the SBP make up a Procedure Control Block (PCB). The effect is that the PCB is a contiguous set of PRT cells, as seen in Figure 23. The PCB construct is based on the one used in ALGOL-E (10), and its usefulness is in recursive calls to a procedure or function.

c. Parameter Mapping

NPS-PASCAL uses a scheme similar to ALGOL-E in mapping the actual parameters of a procedure or function to its formal parameters. After recognition of NPS-PASCAL subroutine identifier, the actual parameters that are identifiers, have their intermediate code generated in the form of a "PARM" or "PARMV" mneumonic followed by the PRT location of the actual parameter. These mneumonics load the execution stack with the values of the actual parameters in the Code Generator. If the actual parameter is an expression, the expression result will be loaded automatically on top of the execution stack. Consequently, the compiler generates the mneumonic "PARMX" after the recognition of a complete expression that is acting as a value parameter. PARMX will not require any action in the Code Generator.

```
VAR CH : CHAR;
FUNCTION ZERO (F, A : REAL) : REAL;
    VAR x, z : REAL;
        sx : BOOLEAN;
```



A PROCEDURE CONTROL BLOCK

FIGURE 23

With the actual parameter in place, program control will branch to the procedure or function itself. The compiler generates code to place three items on the top of the execution stack. The first item is the number of formal parameters (f) in the subroutine, the second is the PRT location of the subroutine's identifier (IDLOC), and the third is the SBP address in the PRT (SBPLOC) of the subroutine. The compiler then generates the SAVP operator, followed by the total byte count of PRT storage (t) assigned for the subroutine's identifier and all formal parameters. This is followed by a listing of byte storage required by each formal parameter (Pi) in the PRT in descending order. The execution of the SAVP operator is expected to cause the following actions to be generated by the Code Generator.

(1) The SBP location is examined
    (a) if SBP = 0 then SBP :=1, else
    (b) SPB >0 and segment of length (SBPLOC-IDLOC + 2) is obtained from the top of available memory -- say at address x. The PCB is then copied from the PRT to the memory segment starting at x. The contents of the segment at x is then called the Save Block (SP). SBP : = x.
(2) The top two elements of the execution stack are deleted; the next element (f) is copied and deleted from the stack; pi = p_1.

(3) If f = 0 then halt. All actual parameters have been copied into the formal parameter locations in the PCB.

(4) PRT location (IDLOC + t - p_i): = top of execution stack; delete the top element of the execution stack; t := t - p_i; p_i := p_i+1.

(5) f := f - 1; go to step (3).

This process ensures that recursively calling a subroutine will not destroy the local variables and parameters of any preceding calls.

d. Function Return Value

Coupled with the SAVP operator is the UNSP (unsave) operator that reverses the actions of SAVP. Two parameters are required at the top of the stack -- the SBP location in the PRT (SBPLOC), and the PRT location of the subroutine identifier (IDLOC). The actions, then, of UNSP are:

(1) The value stored at IDLOC is copied to the top of the stack (this returns a value for function calls; this value will be deleted for procedure calls).

(2) If the value of SBPLOC is greater than 1 then the SB at location SBPLOC in the free memory area is copied back to the PCB and the memory is freed. If SBP = 1 then SBP:=0. Consequently, the UNSP operator returns a value from function calls, and restores the PCB in the event of recursive calls. Figure 24 shows

```
VAR Y : INTEGER
.
.
.
PROCEDURE LO (X : INTEGER; VARY : INTEGER);

    VAR TEMP : REAL

    Begin
         TEMP : = SQRT (X)
         y : = TRUNC (TEMP);

    End;
D : = 6;
...
Lo(49, D);
.
.
.
```

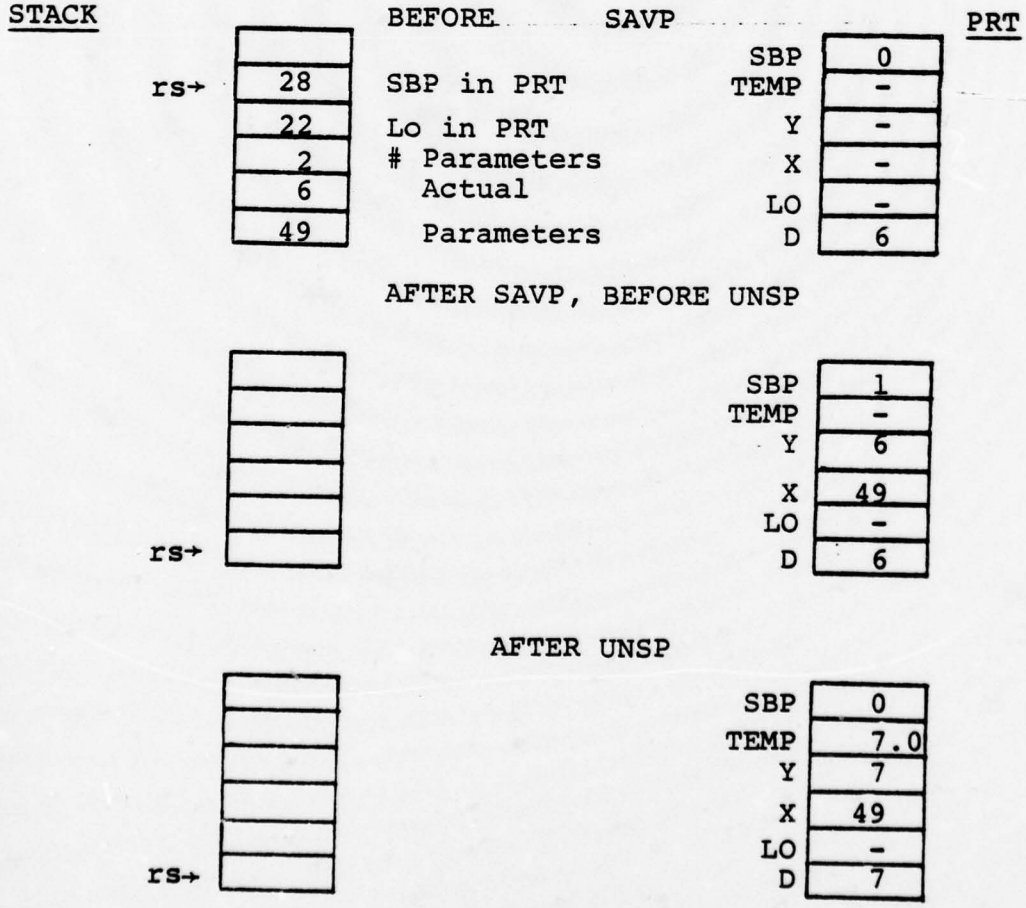


FIGURE 24

the actions of the SAVP and UNSP operators on the PRT and the execution stack.

e. Forward Declared Procedures and Functions

To permit the invocation of a procedure or function prior to its definition, NPS-PASCAL utilizes a forward reference. The forward reference consists of the procedure (function) head, followed by the word FORWARD. When the procedure (function) is defined later in the program, the parameters are not repeated. FORWARD is not a reserved word in NPS-PASCAL. It is instead referred to as a "directive." Directives are identifiers in NPS-PASCAL, that can only occur immediately after a procedure or function heading. Directives are entered in the NPS-PASCAL BUILTINTABLE.

f. External Procedures and Functions

External Procedures and Functions is a NPS-PASCAL extension to Standard Pascal. The outcome of this construct is that NPS-PASCAL functions and procedures can be separately compiled as complete programs. The NPS-PASCAL user can then gain access to one of these programs by an external reference to it. The format of this reference is a procedure (function) head, followed by the word EXTERNAL. This construct is another NPS-PASCAL directive. The intermediate code for the external file would then be read from the disk and incorporated into the program intermediate code as if it were a standard subroutine. This feature of NPS-PASCAL has not been thoroughly tested for proper implementation.

g. Standard Procedures and Functions

The built-in procedures and functions that currently exist in NPS-PASCAL correspond to the standard procedures and functions specified in STANDARD PASCAL. However, their operation is considerably different than user defined procedures and functions. The compiler first generates code for any subroutine actual parameters. A unique mneumonic for the built-in procedure or function is then generated which tells the Code Generator or Interpreter that it must remove the parameters from the execution stack, do the necessary operation, and return the result to the stack. The standard procedures for input and output (Read, Readln, Write, and Writeln) will not require special action to be taken by the Code Generator. The remaining standard procedures dealing with files and pointer variables generate mneumonics that will require action by the Code Generator. A complete listing of NPS-PASCAL standard procedures and standard functions is contained in Appendix C and their usage is outlined in the NPS-PASCAL USER'S MANUAL.

6. Input-Output

Input and output can be handled in two ways: via console, and via disk. Console I/O refers to the device the NPS-PASCAL user is utilizing to provide commands to the system -- usually a CRT terminal or teletype. Disk I/O refers to utilizing auxiliary files on the disk for data manipulation.

Input from console I/0 is achieved through READ or READLN statements. Console output is accomplished by the WRITE and WRITELN statements. Input to the console is accomplished by an operating system routine that reads one full console line into an input buffer. The Code Generator generates code to examine the buffer and convert ASCII characters contained within the buffer into appropriate NPS-PASCAL internal integer, real, or string format. The input value is associated with the appropriate read statement variable parameter and then stored in the space allocated for that variable. A write statements takes the internal representation of integer, decimal, or byte values and converts them to their ASCII character format. These values are then provided to an operating system print routine for console output. Constants and string variables are stored as ASCII strings in the NPS-PASCAL intermediate code; hence, the Code Generator will generate code to send them character by character to the system print routine.

Disk I/0 is achieved through the same read and write statements utilized for console I/0. However, to read data from a disk file requires that the file identifier be specified as the first parameter in a read statement's list of actual parameters. The file identifier has to be specified in the same location for disk write statements as well. The file identifiers used in read/write statements must be declared in a variable declaration part of a program block, or as a program parameter in the program declaration (called

an external file). The file identifier has a specific PRT entry assigned by the compiler. At program execution, space will have to be allocated on the NPS-PASCAL stack for File Control Block (FCB) information necessary to interface file operations with the operating system. Additionally, space should be provided for a 128 byte I/O buffer for every declared file.

7. NPS-PASCAL Pseudo Operators

The NPS-PASCAL compiler generates a variety of pseudo operators that were designed to permit effective conversion to 8080 code in the NPS-PASCAL code generator. The design, however, also lends itself to the development of an interpreter. Reference 1 contains an account of the current Code Generator's organization. With the added ability of symbol table access during the 8080 code generating phase, a revised organization is suggested and outlined in Figure 25. The description below of the pseudo operators is based on this suggested reorganization of the NPS-PASCAL code generator.

The top item on the NPS-PASCAL stack, or execution stack, is addressed by the variable RA. This variable is also known as the stack pointer. The next-to-the-top item on the stack is addressed by the variable RB. The values contained in the first two bytes addressed by RA and RB are referenced by the variables RA$VAL and RB$VAL. Real and string values may be represented on the stack either by storing the actual value itself on the stack, or by using an address that indicates the starting position of the actual

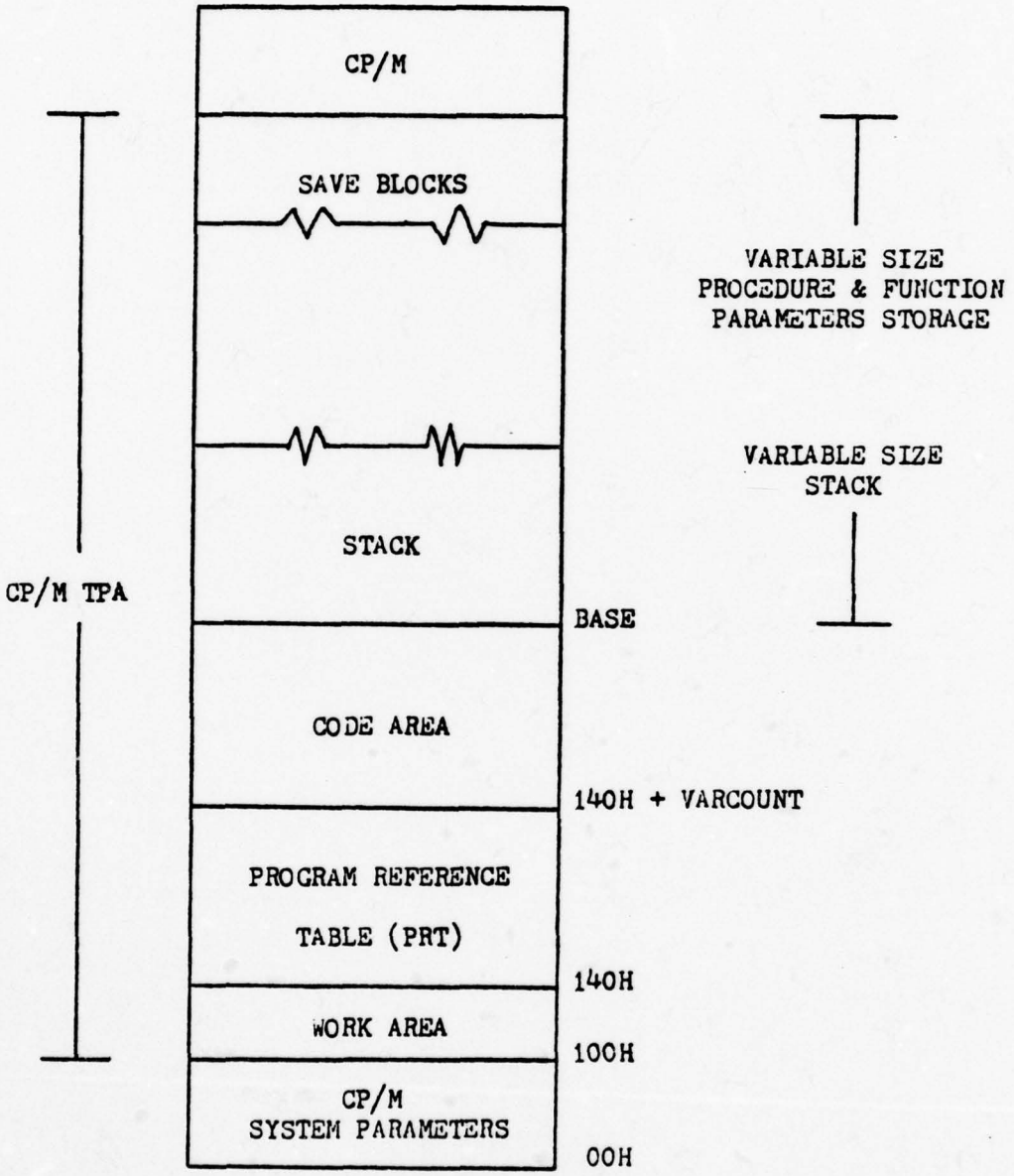

CP/M
SAVE BLOCKS
VARIABLE SIZE
PROCEDURE & FUNCTION
PARAMETERS STORAGE
VARIABLE SIZE
STACK
STACK
CP/M TPA
BASE
CODE AREA
140H + VARCOUNT
PROGRAM REFERENCE
TABLE (PRT)
140H
WORK AREA
100H
CP/M
SYSTEM PARAMETERS
00H

FIGURE 25

storage area.

a. Literal Data References

LITA: (Literal Address). This operator generates 8080 code to place the following two byte integer value on the stack.

b. Allocation Operators

ALL: (Allocate). This operator generates code that initializes the number of bytes of storage required by the PRT. The size of the PRT is contained in the following two byte integer value.

LBL: (Label). This operator is used on the Code Generator's first pass to calculate the address of the label in the code area and save it in the label table using the next two byte integer number as the label table index.

LDIB: (Load Immediate BCD). This operator generates code to place the following eight bytes on the execution stack.

LDII: (Load Immediate Integer). This operator generates code to place the following two bytes on the execution stack.

LOD: (Load Byte). This operator generates code to move RA$VAL into the 8080 HL register. The byte value stored at the location prescribed by HL register pair is then moved to the top of the stack preceded by a high order zero byte.

LODB: (Load BCD). This operator generates code to move RA$VAL into the 8080 HL register. The register is

incremented by eight, and the value stored at the location prescribed by the HL register pair, and the preceding seven bytes, are moved onto the stack in descending order.

LODI: (Load Integer). This operator generates code to move RA$VAL into the 8080 HL register. The two bytes stores at the starting location prescribed by the HL register pair are then moved onto the execution stack.

c. Arithmetic Operators

CNVB: (Convert BCD). This operator generates 8080 code to replace the BCD value of the top eight bytes in the stack by a two byte integer value. Conversion of the number takes place in the work area.

CNVI: (Convert Integer). This operator generates code to replace the two byte integer value on top of the stack by its eight byte BCD value. Conversion of the number takes place in the work area.

CNAI: (Convert Integer Preceding Address). This operator generates code to move the top two bytes from the top of the stack into a save area, then to move the following integer into the work area. Code is then generated to convert the integer into a BCD eight byte format. The resulting BCD number is then returned to the stack followed by the two bytes from the save area.

CN2I: (Convert Integer Preceding BCD). This operator generates code to move the two bytes from the top of the stack into a save area, then move the following BCD number into the work area. Code is then generated to convert

the BCD number to an integer number. The resulting integer value is then returned to the stack followed by the two bytes from the save area.

ADDB: (Add BCD). This operator generates code to move the two BCD values from the top of the stack into the work area where the sum of the two numbers is calculated and returned to the stack in BCD format.

ADDI: (Add Integer). This operator generates code to move the two integer values on the top of the stack to the 8080 registers where the sum of the two numbers is calculated and returned to the top of the stack.

SUBB: (Subtract BCD). This operator generates code to move the two BCD values from the top of the stack into the work area where the first BCD number is subtracted from the second BCD number. The resulting BCD value is returned to the top of the stack.

SUBI: (Subtraction Integer). This operator generates code to move the two integer values on top of the stack to the 8080 registers where the first integer is subtracted from the second. The resulting integer value is returned to the top of the stack.

MULB: (Multiply BCD). This operator generates code to move the two BCD values at the top of the stack into the work area where their product is calculated. The resulting BCD number is returned to the top of the stack.

MULI: (Multiply Integer). This operator generates code to move the two integer values on top of the stack

to the working area where the product is calculated. The resulting integer number is returned to the top of the stack.

DIVB: (Divide BCD). This operator generates the 8080 code to move two BCD values from the top of the stack into the work area where the second BCD number is divided by the first. The quotient is returned to the top of the stack in BCD format.

MODX: (Modulo). This operator generates code to move the two integer values at the top of the stack to the work area where the second integer is divided by the first. The remainder of the quotient is returned to the top of the stack in integer format.

DIVI: (Divide Integer). This operator generates code to move the two integer values at the top of the stack to the work area where the second integer is divided by the first. The quotient is returned to the top of the stack in integer format.

LSSB: (Less Than BCD). This operator generates code to move the two BCD values at the top of the stack to the work area where the two numbers are compared. If the second BCD number is smaller than the first BCD number, a one is returned to the stack. Otherwise, a zero is returned.

LSSI: (Less Than Integer). This operator generates code to move the two integer values at the top of the stack to the 8080 registers where the two numbers are compared. If the second integer is smaller than the first integer, a one is returned to the stack. Otherwise a zero is returned.

LEQB: (Less Than or Equal BCD). This operator generates code to move the two BCD values at the top of the stack to the work area where the two numbers are compared. If the second BCD number is smaller than, or equal to, the first, a one is returned to the stack. Otherwise, a zero is returned to the top of the stack.

LEQI: (Less Than or Equal Integer). This operator generates code to move the two integer values at the top of the stack to the 8080 registers where the two numbers are compared. If the second integer removed from the stack is smaller than, or equal to, the first integer, a one is returned to the top of the stack. Otherwise, a zero is returned.

EQLB: (Equal to BCD). This operator generates code to move the two BCD values on top of the stack to the work area where the two numbers are compared. If the two BCD numbers are equal, a one is returned to the stack. Otherwise a zero is returned.

EQLI: (Equal to Integer). This operator generates code to move the two integer values at the top of the stack to the 8080 registers where the two numbers are compared. If the two integers are equal a one is returned to the stack. Otherwise a zero is returned.

NEQB: (Not Equal to BCD). This operator generates code to move the two BCD values at the top of the stack to the work area where the two numbers are compared. If the numbers are not equal a one is returned to the stack. Otherwise a zero is returned.

NEQI: (Not Equal to Integer). This operator generates code to move the two integer values at the top of the stack to the 8080 registers where the two numbers are compared. If the numbers are not equal, a one is returned to the top of the stack. Otherwise, a zero is returned.

GEQB: (Greater Than or Equal BCD). This operator generates code to move the two BCD values at the top of the stack to the work area where the two numbers are compared. If the second number is greater than or equal to the first number, a one is returned to the stack. Otherwise, a zero is returned.

GEQI: (Greater Than or Equal Integer). This operator generates code to move the two integer values at the top of the stack to the 8080 registers where the two numbers are compared. If the second number removed from the stack is greater than, or equal to, the first, a one is returned to the stack. Otherwise, a zero is returned.

GRTB: (Greater Than BCD). This operator generates code to move the two BCD values at the top of the stack to the work area where the two numbers are compared. If the second BCD number is greater than the first BCD number, a one is returned to the stack. Otherwise a zero is returned.

GRTI: (Greater Than Integer). This operator generates code to move the two integer numbers at the top of the stack to the 8080 registers where they are compared. If the second number is greater than the first, a one is returned to the stack. Otherwise, a zero is returned.

NEGB: (Negate BCD). This operator generates code to move RA$VAL to the 8080 registers where it compliments the sign bit of the low order byte. RA$VAL is then replaced on the top of the stack.

NEGI: (Negate Integer). This operator generates code to move RA$VAL from the stack to the 8080 registers, compliments the sign bit, and returns RA$VAL to the top of the stack.

COMB: (Complement BCD). This operator generates code to move the eight byte BCD value from the top of the execution stack into the work area, finds the nine's compliment of the number and returns the compliment to the top of the stack.

COMI: (Complement Integer). This operator generates code to move RA$VAL into the 8080 registers, finds its two's compliment, and returns this value to the top of the stack.

d. Boolean Operators

NOT: (Boolean Not). This operator generates code to move RA$VAL into 8080 registers and checks the low order bit. If the bit is zero, a two byte value of one is returned to the stack. If the bit is one, a two byte value of zero is returned.

AND: (Boolean And). This operator generates code to move RA$VAL and RB$VAL into 8080 registers for a logical AND comparison of their low order bits. If the relation is true, a two byte value of one is returned to the top of the stack. If the relation is false, a two byte value of zero is returned.

BOR (Boolean Or). This operator generates code to move RA$VAL and RB$VAL into 8080 registers for a logical OR comparison of their low order bits. If the relation is true, a two byte value of one is returned to the stack. If the relation is not true, a two byte value of zero is returned to the stack.

e. String Operators

LSDI: (Load String Immediate). This operator generates code to move a variable number of bytes, depending on the value in the following byte of the intermediate code, to the top of memory. The address location of the string storage area is sent to the top of the execution stack.

EQLS: (Equal String). This operator generates code to move RA$VAL and RB$VAL into 8080 registers and conducts a byte by byte comparison of the strings located at the addresses stored in the 8080 registers. If the strings are equal, a one is returned to the stack. Otherwise a zero is returned.

NEQS: (Not Equal String). This operator takes the same actions as the EQLS operator except that a one is returned to the top of the stack when the strings are not equal. Otherwise, a zero is returned.

LEQS: (Less Than or Equal String). This operator behaves the same as EQLS except that a one may also be returned to the top of the stack of the second string is less than the first string in regards to the lexicographic ordering of the character set.

GEQS: (Greater Than or Equal String). This operator behaves the same as the LEQS operator, except a one is returned when the second string is greater than, or equal to, the first.

LSSS: (Less Than String). This operator behaves the same as the LEQS operator, except a one is returned to the stack only when the second string is less than the first string.

GRTS: (Greater Than String). This operator is the opposite of LSSS. A one is returned to the top of the stack only if the second string is greater than the first. Otherwise, a one is returned.

f. Stack Operators

XCHG: (Exchange). This operator exchanges the values of RA$VAL and RB$VAL.

INC: (Increment). This operator generates code to increment the value of RA$VAL by one.

DEC: (Decrement). This operator generates code to decrement the value of RA$VAL by one.

DEL: (Delete). This operator deletes the top two bytes from the stack. RA is set to the position of RB and RB is repositioned to the item below its current position on the stack.

g. Store Operators

STOB: (Store BCD). This operator generates code to move RA$VAL into the 8080 HL register, and then moves the next eight bytes from the execution stack into memory starting at the address specified in the HL register. The value

of the BCD number is preserved on the stack by incrementing the stack pointer by eight.

STOI: (Store Integer). This operator generates code to move RA$VAL into the HL register and then moves the value of RB$VAL into memory starting at the location specified by the HL register. The value of the integer number is preserved on the stack by incrementing the stack pointer by two.

STO: (Store Byte). This operator duplicates the actions described by the STOI operator.

STDB: (Store Destruct BCD). This operator generates code to move RA$VAL into the 8080 HL register, then moves the next eight bytes from the stack to memory at the address indicated by the HL register.

STDI: (Store Destruct Integer). This operator generates code to move RA$VAL into the 8080 HL register, and then moves RB$VAL into memory starting at the address indicated by the HL register.

STD: (Store Destruct Byte). This operator generates code to move RA$VAL into the 8080 HL register, and then moves the next byte from the stack to memory starting at the address indicated by the HL register. The stack pointer is decremented by one (to bypass high order zero left on the stack).

h. Array Operators

SUB: (Calculate Offset to a Specific Array Element). This operator generates code to move RA$VAL into the 8080 HL register. The following byte of information tells

how many indicies must be removed from the top of the stack and be used with the displacement vector information stored at the location indicated by the HL register. The resultant two byte PRT address is placed at the top of the execution stack.

i. Set Operators

UNION: (Set Union). This operator generates code to move RA$VAL to the 8080 HL register and copy the values contained at the location specified by the HL register at the top of memory. It generates code to do the same with the value stored in RB$VAL. These two sets are then merged into one at the top of memory, and their address is returned to the top of the execution stack.

STDIF: (Set Difference). This operator generates the same code as the UNION operator for the values stored in RA$VAL and RB$VAL. The two sets are merged into one discarding any set element that appears in both sets. The address of the result is returned to the top of the stack.

INSEC: (Set Intersection). This operator generates the same code as STDIF except the elements that do not appear in both sets are discarded.

EQSET: (Set Equality). This operator generates code to compare the two sets stored at the addresses specified by RA$VAL and RB$VAL. If they are equal, a one is returned to the stack, otherwise a zero is returned.

NEQST: (Set Non-equality). This operator generates the same code as the EQSET operator, but returns a

one to the stack if the two sets are not equal. Otherwise, a zero is returned to the top of the stack.

INCL1: (Set Inclusion). This operator generates code to compare the two sets stored at the addresses specified by RA$VAL and RB$VAL. If the set stored at RA$VAL is included in one at RB$VAL, then a one is returned to the top of the stack. Otherwise a zero is returned.

INCL2: (Set Inclusion). This operator generates the same code as INCL1 except the set stored at RB$VAL must be included in RA$VAL to return a one to the top of the stack. Otherwise, a zero is returned to the top of the execution stack.

j. File Operators

XTRNAL: (External File). Not implemented.

k. Procedure and Function Operators

PRO: (Subroutine Call). This operator generates code to save the present address loaded in the 8080 program counter (PC) register, and loads the PC register with the address stored in the label table and accessed by using the next two bytes of the intermediate code.

RTN: (Return from Subroutine). This operator generates code to retrieve the address stored by the previously executed PRO operator, and loads the value in the PC register to continue program execution at the point of call to the subroutine.

SAVP: (Save Parameters). This operator generates code to save the present procedure control block, if

necessary, at the top of memory in a Save Block. The series of bytes immediately following in the intermediate code are utilized to store the procedure parameters on the top of the execution stack into the appropriate formal parameter PRT locations.

UNSP: (Unsave Parameters). This operator generates code to copy a SB from the top of memory back into the PCB, if required. The area of memory used by the SB is then freed.

PARM: (Parameter). This operator loads the value stored at the following two byte address in the intermediate code on the top of the execution stack.

PARMV: (Variable Parameter). This operator loads the following two byte address in the intermediate code on the top of the execution stack.

PARMX: (Expression Parameter). This operator indicates that the value on the top of the stack is a parameter for a subroutine.

ABS: (Built in Function - Absolute Value). This operator is generated by a call to built-in function ABS. This operator takes RA$VAL and sets the sign bit to positive.

SQR: (Built in Function - Square). This operator moves the value at the top of the stack into the work area, squares it, and returns the value to the top of the stack.

SIN: (Built in Function - SINE). This operator moves the value at the top of the stack into the work area, computes the sin, and returns the result to the top of the

stack in BCD format.

COS: (Built in Function - COSINE). This operator moves the value at the top of the stack into the work area, computes the cosin, and returns the result to the top of the stack in BCD format.

ARCTN: (Built in Function - ARCTAN). This operator moves the value at the top of the stack into the work area, takes the arc-tangent of it, and returns the result to the top of the stack in BCD format.

EXP: (Built in Function - Exponential). This operator moves the top value of the stack into the work area, raises the value of the base of natural logarithms to this value, and returns the result in BCD format to the top of the stack.

LN: (Built in Function - LN). This operator moves the value at the top of the stack into the work area, takes its natural logarithm, and returns the result in BCD format to the top of the stack.

SQRT: (Built in Function - Square Root). This operator moves the value on the top of the stack into the work area, computes the positive square root of the value, and returns the result in BCD format to the top of the stack.

ODD: (Built in Function - Odd). This operator moves RA$VAL into an 8080 register and checks the low order bit. If it is set to one, then a one is returned to the top of the stack; otherwise, a zero is returned.

EOLN: (Built in Function - END OF LINE). This operator moves RA$VAL to the 8080 HL register. If the value stored at the location specified by the HL register indicates an end of line, a one is returned to the stack; otherwise a zero is returned.

EOF: (Built in Function - End of File). This operator moves RA$VAL into the 8080 HL register and checks to see if the value stored at that location indicates an end of file. If the end of file is indicated, a one is returned to the stack; otherwise a zero is returned.

TRUNC: (Built in Function - Trunc). This operator moves the BCD value from the top of the stack to the work area, truncates the decimal point and all numbers to the right of it, and returns the remaining integer value to the top of the stack.

ROUND: (Built in Function - Round). This operator moves the BCD value on the top of the stack to the work area, rounds it to the nearest integer, and returns the integer result to the top of the stack.

ORD: (Built in Function - Ord). This operator removes RA$VAL from the top of the stack, converts it to an integer value and returns it to the stack.

CHR: (Built in Function - CHR). This operator moves the integer value in RA$VAL to the work area, determines the corresponding character value, and returns the value to the top of the stack.

SUCC: (Built in Function - Successor). This operator moves the value of RA$VAL to the work area, determines the successor value of the same type, and returns this result to the top of the stack.

PRED: (Built in Function - Predecessor). This operator moves the value of RA$VAL to the work area, determines the preceding value of the same type, and returns this result to the top of the stack.

PUT: (Built in Procedure - PUT). Not implemented.

GET: (Built in Procedure - GET). Not implemented.

RESET: (Built in Procedure - RESET). Not implemented.

REWRT: (Built in Procedure - Rewrite). Not implemented.

SEEK: (Built in Procedure - Seek). Not implemented.

PAGE: (Built in Procedure - PAGE). Not implemented.

NEW: (Built in Procedure - NEW). Not implemented.

DISPZ: (Built in Procedure - DISPOSE). Not implemented.

1. Program Control Operators

ENDP: (End of Program). This operator causes the Code Generator to close the intermediate file and the object file terminating compilation.

BRL: (Branch to Label). This operator calculates the label address in the label table using the next two bytes

of intermediate code as the entering argument. The code count stored at the label table address is added to the address of the start of the code area. This value is inserted to the PC register and program control continues at that location.

BCL: (Branch Conditional Label). This operator calculates the branching address in the same manner as the BRL operator. However, this operator moves RA$VAL into an 8080 stack and checks the low order bit. If a one is found, the branch is executed. If the low order bit is zero, the program continues without branching.

KASE: (Branch to Case). This operator compares the next two bytes of intermediate code to the value of RA$VAL. If they are equal then a branch is executed using the following two bytes of intermediate code as the argument for the label table. If the value does not equal RA$VAL then program execution continues without branching. (Note: "CASE" is not used here, it is a reserved word in PL/M which would cause conflicts in the compiler source program.)

m. Input-Output Operators

RDBV: (Read Variable BCD). This operator generates code to read a BCD number from the input file, change it into its acceptable storage form, and store the eight byte internal form at the location specified on the top of the execution stack.

RDVI: (Read Variable Integer). This operator generates code to read an integer number from the input file, change it into its acceptable storage form, and store the two

byte number at the location specified on the top of the stack.

RDV: (Read Variable Byte). This operator generates code to read a byte variable from the input file, change it into its acceptable storage form, and store it at the location specified on the top of the stack.

RDVS: (Read Variable String). This operator generates code to read a string variable from the input file, and store it at the location in memory specified by the top two bytes on the stack.

WRTB: (Write BCD). This operator generates code to move the eight byte BCD number at the top of the stack into the work area, changes the number into its printable form, and sends the number to the output file.

WRTI: (Write Integer). This operator generates code to move an integer number from the top of the stack into the work area, changes it into its printable format, and sends the number to the output file.

WRT: (Write Byte). This operator generates code to move the two byte value from the top of the stack into the code area, changes it into its printable format, and sends it to the output file.

WRTS: (Write String). This operator generates code to move the string, stored at the address specified in RA$VAL, to the output file.

DUMP: (Starts New Output Line). This operator generates code to send a carriage return and line feed to the output file.

III. CONCLUSIONS

The NPS-PASCAL project is a step closer to the full implementation of STANDARD-PASCAL for Intel 8080 based microcomputers. With the major exceptions of PASCAL sets and files, the NPS-PASCAL compiler portion is complete.

The associated development of a complete 8080 code generator or interpreter is required to complete integrated program testing and timing tests. This will determine program correctness and checks the efficiency of NPS-PASCAL. Only then can firm conclusions be drawn on the usefulness of the complete system.

IV. RECOMMENDATIONS

Although NPS-PASCAL was designed to meet the criteria of STANDARD PASCAL, there are a number of extensions that could easily be made to the language to increase its usefulness. These include complete implementation of external functions and procedures, string concatenation features, and additional standard functions and procedures.

The grammar for NPS-PASCAL presently supports external subroutines. However, the means of accessing the compiled code have not been designed. Similarly, the ability to operate on strings already exists in the grammar of NPS-PASCAL, but the associated mneumonics and compilation procedures are not developed.

Prior to language extension concerns, however, comes the implementation of the Code Generator. With the added ability of symbol table access, run-time debugging should be easier to implement.

APPENDIX A - COMPILER ERROR MESSAGES

DE Disk error : Recompile.

NC Incorrect character : See User Manual.

TO Symbol table overflow : Reduce number of declarations.

EE Exponent size error : See User Manual.

IE Integer size error : See User Manual.

IP Improper parameter : The actual parameter type does not match the formal parameter type.

IS Invalid subrange error : Check type and limits of declared subrange.

IT Invalid type error : Array component type specification invalid.

IA Invalid array index : Array index types must be scalar - INTEGER or REAL types are invalid.

NS Invalid set element : Set elements must be scalar.

IC Invalid constant variable : Constant entry in symbol table is invalid - probably due to a prior error.

AT Assignment type error : Type of expression not compatible with assignment variable type.

IR Invalid read variable : Only INTEGER, REAL, or STRING values can be read.

DT Duplicate type name : Type identifiers must be unique.

PN Incorrect number of parameters : The total number of actual parameters fails to equal the total number of formal parameters.

LS Label syntax error : All labels must be integers.

DC Duplicate constant name : Constant identifiers must be unique.

TI Invalid type identifier : Type identifier not previously declared.

AN Array nest overflow : Simplify declaration.

AD Array dimension stack overflow : Simplify array declaration.

IV	Variant stack overflow : Reduce the number of variant cases.
RN	Record field stack overflow : Reduce the number of fields specified.
VN	Variable declaration stack overflow : Reduce the number of variables declared per line.
CE	Invalid expression : The variable types within the expression are not compatible.
UL	Undefined label error : Label not declared in label statement.
NE	Incorrect actual parameter : The actual parameter must be a variable and not an expression.
UO	Invalid unary operator : Variable type must be INTEGER, REAL, or subrange of INTEGER.
ET	Invalid expression type : The types of variables used in an expression are incompatible.
UP	Undeclared procedure : Procedure identifier not previously declared.
PE	Parameter error : This parameter format can only be used in a write statement.
WP	WRITE$STMT parameter error : The length parameter has to be of type INTEGER.
RT	WRITE$STMT parameter error : The parameter has to be of type REAL.
CV	Incorrect control variable : The control variable has not been declared or is of type REAL.
SO	State stack overflow : simplify program.
VO	Variable stack overflow : Reduce the length of variable printnames.
NP	No production : Syntax error in source line.

APPENDIX B - NPS-PASCAL OPERATORS

A. ARITHMETIC OPERATORS

Binary

Operator	Operation	Type of operands	Type of result
+	addition	integer or real	integer or real
-	subtraction	integer or real	integer or real
*	multiplication	integer or real	integer or real
/	division	integer or real	real
DIV	division with truncation	integer	integer
MOD	modulo	integer	integer

Unary

Operator	Operation	Type of operands	Type of result
+	identity	integer or real	integer or real
-	sign inversion	integer or real	integer or real

B. BOOLEAN OPERATORS

Operator	Operation	Type of operands	Type of result
OR	logical "or"	Boolean	Boolean
AND	logical "and"	Boolean	Boolean
NOT	logical negation	Boolean	Boolean

C. SET OPERATORS

Operator	Operation	Type of operands	Type of result
+	set union	any set type T	T
-	set difference	any set type T	T
*	set intersection	any set type T	T

D. RELATIONAL OPERATORS

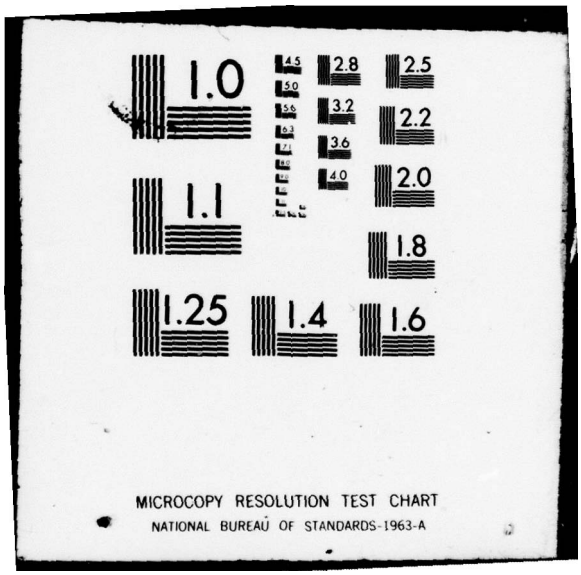
Operator	Type of operands	Type of result
=	any set, simple pointer or string type	Boolean
<>	any set, simple pointer or string type	Boolean
<	any simple or string type	Boolean
>	any simple or string type	Boolean
< =	any set, simple or string type	Boolean
> =	any set, simple or string type	Boolean
IN	left operand : any ordinal type T right operand: SET OF T	Boolean

where simple type = INTEGER, REAL or scalar type.

AD-A071 972 NAVAL POSTGRADUATE SCHOOL MONTEREY CA F/G 9/2
NPS-PASCAL: A PASCAL IMPLEMENTATION FOR MICROPROCESSOR-BASED CO--ETC(U)
JUN 79 J L BYRNES

UNCLASSIFIED NL

2 OF 3
AD
A071972


1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
4.5
5.0
5.6
MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

APPENDIX C - NPS-PASCAL STANDARD TABLES

A. Table of standard identifiers

Constants:

false, true

Types:

integer, Boolean, real, char, text

Program parameters:

input, output

Functions:

abs, arctan, chr, cos, eof, eoln, exp, ln, odd, ord,
pred, round, sin, sqr, sqrt, succ, trunc

Procedures:

get, new, page, put, read, readln, reset, rewrite,
seek, write, writeln

Directives:

forward, external

B. Table of reserved words

and	array	begin	case
const	div	do	downto
else	end	eop	file
for	function	goto	if
in	label	mod	nil
not	of	or	packed
procedure	program	record	repeat
set	then	to	type
until	var	while	with

APPENDIX D - NPS-PASCAL LANGUAGE STRUCTURE

This appendix describes the language in BNF notation followed by the semantic actions associated with the individual productions. The description, offset with asterisks, is given in terms of the compiler data structures and the intermediate code generated. Numbered productions without a production result indicate empty productions. Items enclosed in brackets and separated by slants are alternative semantic actions. N/A indicates no action is taken. This notation is based on that used in Reference 9 and Reference 10.

```
1    <program>  ::=  <program heading> <block> .
 *   <program heading> ; <block> ;
 *   ALL ;   {number of bytes allocated for variables}
 *   ENDP   {end of file indicator}

2                    <procedure heading> <block> .
 *   <procedure heading> ; <block> ;
 *   ALL ;   {number of bytes allocated for variables}
 *   ENDP   {end of file indicator}

3                    <function heading> <block> .
 *   <procedure heading> ; <block> ;
 *   ALL ;   {number of bytes allocated for variables}
 *   ENDP   {end of file indicator}

4    <program heading>  ::=  PROGRAM <program ident> (
                             <xfile ident> ) ;
```

```
 *   <program identifier> ; <xfile identifier>

5    <xfile ident>  ::=  <file ident>
 *   <file identifier>

6                          <xfile ident> , <file ident>
 *   <xfile identifier> ; <file identifier>

7    <prog ident>  ::=  <identifier>
     N/A

8    <file ident>  ::=  <identifier>
 *   <identifier>
 *   {enter file identifier in symbol table}

9    <block>  ::=  <ldp> <cdp> <tdp> <vdp> <p&fdp> <stmtp>
 *   <label declaration part>
 *   <constant definition part>
 *   <type definition part>
 *   <variable declaration part>
 *   <procedure and function declaration part>
 *   <statement part>

10   <ldp>  ::=
 *   <empty>

11                LABEL <label string> ;
 *   <label string>

12   <label string>  ::=  <label>
 *   <label>   {enter label in symbol table}

13                           <label string> , <label>
 *   <label string> ; <label>
 *   {enter label in symbol table}

14   <label>  ::=  <number>
 *   <number>   {check number type}

15   <cdp>  ::=
 *   <empty>

16                CONST <const def> ;
 *   <constant definition>

17   <const def>  ::=  <ident const def> ;
 *   <identifier constant definition>

18                       <const def> ; <ident const def>
 *   <constant definition> ;
 *   <identifier constant definition>

19   <ident const def>  ::=  <ident const> = <constant>
 *   <identifier constant> ; <constant>
```

```
 *   {enter constant value in symbol table}

20   <ident const>  ::=  <identifier>
 *   <identifier>   {enter constant entry in symbol table}

21   <constant>  ::=  <number>
 *   <number>   {assign constant attributes}

22                    <sign> <number>
 *   <sign> ; <number>   {assign constant attributes}

23                    <constant ident>
 *   <constant identifier>   {assign constant attributes}

24                    <sign> <constant ident>
 *   <sign> ; <constant identifier>
 *   {assign constant attributes}

25                    <string>
 *   <string>   {assign constant attributes}

26   <constant ident>  ::=  <identifier>
 *   <identifier>

27   <sign>  ::=  +
 *   {assign SIGNTYPE value}

28                 -
 *   {assign SIGNTYPE value}

29   <tdp>  ::=
 *   <empty>

30              TYPE <type def string>
 *   <type definition string>

31   <type def string>  ::=  <type id>
 *   <type identifier>

32                          <type def string> ; <type id>
 *   <type definition string> ; <type id>

33   <type id>  ::=  <type ids> = <type>
 *   <type identifiers> ; <type>
 *   {alter type entry}

34   <type ids>  ::=  <identifier>
 *   <identifier>   {enter type}

35   <type>  ::=  <simple type>
 *   <simple type>

36                <structured type>
 *   <structured type>
```

```
37                  <pointer type>
 *   <pointer type>

38   <simple type>  ::=  <type ident>
 *   <type identifier>

39                       ( <tident string> )
 *   <type identifier string>

40                       <constant> .. <constant>
 *   <constant> ; <constant>
 *   {enter subrange entry in symbol table}

41   <type ident>  ::=  <identifier>
 *   <identifier>   {set TYPE$LOCT}

42   <tident string>  ::=  <identifier>
 *   <identifier>
 *   {enter user defined element in symbol table}

43                         <tident string> , <identifier>
 *   <tident string> ; <identifier>
 *   {enter user defined element in symbol table}

44   <structured type>  ::=  <unpacked structured type>
 *   <unpacked structured type>

45                           PACKED
                             <unpacked structured type>
 *   <unpacked structure type>

46   <unpacked structured type>  ::=  <array type>
 *   <array type>

47                                    <record type>
 *   <record type>

48                                    <set type>
 *   <set type>

49                                    <file type>
 *   <file type>

50   <array type>  ::=  ARRAY [ <index type string> ] OF
                        <component type>
 *   <index type string> ; <component type>
 *   {enter array type, each index address in the symbol table,
     displacement vector information, and array offset in the
     symbol table}

51   <index type string>  ::=  <index type>
 *   <index type>   {set array dimensions}
```

```
52                                  <index type string> ,
                                    <index type>
 *   <index type string> ; <index type>
 *   {set array dimensions}

53   <index type>  ::=  <simple type>
 *   <simple type>

54   <component type>  ::=  <type>
 *   <type>

55   <record type>  ::=  RECORD <field list> END
 *   <field list>   {enter record type}

56   <field list>  ::=  <fixed part>
 *   <fixed part>

57                      <fixed part> ; <variant part>
 *   <fixed part> ; <variant part>

58                      <variant part>
 *   <variant part>

59   <fixed part>  ::=  <record section>
 *   <record section>

60                      <fixed part> ; <record section>
 *   <fixed part> ; <record section>

61   <record section>  ::=  <field ident string> : <type>
 *   <field identifier string> ; <type>
 *   {enter record attributes}

62
 *   <empty>

63   <field ident string>  ::=  <field ident>
 *   <field identifier>

64                              <field ident string> ,
                                <field ident>
 *   <field identifier string> ; <field identifier>

65   <field ident>  ::=  <identifier>
 *   <identifier>   {enter record field in symbol table}

66   <variant part>  ::=  CASE <tag field> <type ident> OF
                          <variant string>
 *   <tag field> ; <type identifier> ; <variant string>

67                        CASE <type ident> OF
                          <variant string>
 *   <type identifier> ; <variant string>
```

```
68   <variant string>  ::=  <variant>
 *   <variant>

69                              <variant string> ; <variant>
 *   <variant string> ; <variant>

70   <tag field>  ::=  <field ident> :
 *   <field identifier>   {set TAG$FD to true}

71   <variant>   ::=  <case prefix> ( <field list> )
 *   <case prefix> ; <field list>

72
 *   <empty>

73   <case label list>  ::=  <case label>
 *   {set LABEL$STACK(SP)} ; KASE ; <case label> ;
 *   {LABEL$STACK(SP)}

74                               <case label list> , <case label>
 *   <case label list> ; KASE ; <case label> ;
 *   {LABEL$STACK(MP)}

75   <case label>  ::=  <constant>
 *   <constant>
 *   [ {set variant attributes} / {set CASE$STMT label} ]

76   <set type>  ::=  SET OF <base type>
 *   <base type>   {enter type in symbol table}

77   <base type>  ::=  <simple type>
 *   <simple type>

78   <file type>  ::=  FILE OF <type>
 *   <type>   {enter type in symbol table}

79   <pointer type>  ::= ^ <type ident>
 *   <type identifier>   {enter type in symbol table}

80   <vdp>  ::=
 *   <empty>

81              VAR <var declar string> ;
 *   <variable declaration string>

82   <var declar string>  ::=  <var declar>
 *   <variable declaration>

83                              <var declar string> ;
                                <var declar>
 *   <variable declaration string> ;
 *   <variable declaration>

84   <var declar>  ::=  <ident var string> : <type>
```

```
 *    <identifier variable string> ; <type>
 *    {set variable attributes}

85    <ident var string>  ::=  <identifier>
 *    <identifier>   {enter variable in symbol table}

86                             <ident var string> ,
                               <identifier>
 *    <identifier variable string> ; <identifier>
 *    {enter variable in symbol table}

87    <p&fdp>  ::=
 *    <empty> ; LDII {number of parameters} ;
 *    LITA {porf PRT address} ; LITA {porf SBP} ; SAVP
 *    {only if SCOPE$NUM > 1}

88                  <porf declar>
 *    <procedure or function declaration> ;
 *    LDII {number of parameters}
 *    LITA {porf PRT address} ; LITA {porf SBP} ; SAVP
 *    {only if SCOPE$NUM > 1}

89    <porf declar>  ::=  <proc or func> ;
 *    <procedure or function>

90                        <porf declar> <proc or funct> ;
 *    <procedure or function declaration>
 *    <procedure or function>

91    <proc or funct>  ::=  <procedure heading> <block>
 *    <procedure heading> ; <block> ;
 *    LITA {SBP} ; LITA {PRT address} ; UNSP;
 *    RTN ; LBL {procedure label + 1} ; LDII 0 ;
 *    LITA {SBP} ; STDI {SBP initially 0}

92                             <procedure heading> <directive>
 *    <procedure heading> ; <directive>

93                             <function heading> <block>
 *    <function heading> ; <block> ;
 *    LITA {SBP} ; LITA {PRT address} ; UNSP;
 *    RTN ; LBL {procedure label + 1} ; LDII 0 ;
 *    LITA {SBP} ; STDI {SBP initially 0}

94                             <function heading> <directive>
 *    <function heading> ; <directive>

95    <directive>  ::=  <identifier>
 *    <identifier>   {determine if "forward" or "external"}

96    <procedure heading>  ::=  <proc id> ;
 *    <procedure identifier>

97                                <proc id> (
```

```
                                    <formal para section list> ) ;
 *    <procedure identifier> ;
 *    <formal parameter section list> ;
 *    {generate listing of formal parameter types and their
      associated PRT addresses in symbol table}

98    <proc id>  ::=  PROCEDURE <identifier>
 *    <identifier>   {enter procedure in symbol table}

99    <formal para sect list>  ::=  <formal para sect>
 *    <formal parameter section>
 *    {set formal parameter attributes}

100                                 <formal para sect list> ;
                                    <formal para sect>
 *    <formal parameter section list> ;
 *    <formal parameter section>
 *    {set formal parameter attributes}

101   <formal para sect>  ::=  <para group>
 *    <parameter group>

102                               VAR <para group>
 *    <parameter group>
 *    {modify variable parameters FORM$FIELD entry}

103                               FUNCTION <para group>
 *    <parameter group>

104                               PROCEDURE <proc ident list>
 *    <procedure identifier list>

105   <proc ident list>  ::=  <identifier>
 *    <identifier>

106                             <proc ident list> , <identifier>
 *    <procedure identifier list> ; <identifier>

107   <para group>  ::=  <para ident list> : <type ident>
 *    <parameter identifier list> ; <type identifier>

108   <para ident list>  ::=  <identifier>
 *    <identifier>   {enter formal parameter in symbol table}

109                             <para ident list> , <identifier>
 *    <parameter identifier list> ; <identifier>
 *    {enter formal parameter in symbol table}

110   <function heading>  ::=  <funct id> : <result type> ;
 *    <function identifier> ; <result type>
 *    {set the function's type field}

111                               <funct id> (
                                  <formal para sect list> ) :
```

```
                                    <result type> ;
 *    <function identifier> ;
 *    <formal parameter section list> ; <result type>
 *    {generate listing of formal parameter types and their
      associated PRT addresses in symbol table; set the
      function's type field}

112   <funct id>  ::=  FUNCTION <identifier>
 *    <identifier>   {enter function in symbol table}

113   <result type>  ::=  <type ident>
 *    <type identifier>
 *    {allocate proper length of function in the PRT}

114   <stmtp>  ::=  <compound stmt>
 *    <compound statement>

115   <stmt>  ::=  <bal stmt>
 *    <balanced statement>

116                <unbal stmt>
 *    <unbalanced statement>

117                <label def> <stmt>
 *    <label definition> ; <statement>

118   <bal stmt>  ::=  <if clause> <true part> ELSE <bal stmt>
 *    <if clause> ; <true part> ; <balanced statement> ;
 *    LBL {LABELSTACK(MP)+1}

119                    <simple stmt>
 *    <simple statement>

120   <unbal stmt>  ::=  <if clause> <stmt>
 *    <if clause> ; <statement> ;
 *    LBL {LABELSTACK(MP)}

121                      <if clause> <true part> ELSE
                         <unbal stmt>
 *    <if clause> ; <true part> ;
 *    <unbalanced statement> ;
 *    LBL {LABELSTACK(MP)+1}

122   <if clause>  ::=  IF <expression> THEN
 *    <expression> ; NOTX ; {set LABELSTACK(MP)} ;
 *    BLC {LABELSTACK(MP)}

123   <true part>  ::=  <bal stmt>
 *    <balanced statement> ;
 *    BRL {LABELSTACK(SP-1)+1} ;
 *    LBL {LABELSTACK(SP-1)}

124   <label def>  ::=  <label> :
 *    <label> ; LBL {PRT address}
```

```
125  <simple stmt>  ::=  <assignment stmt>
 *   <assignment statement>

126                      <procedure stmt>
 *   <procedure statement>

127                      <while stmt>
 *   <while statement>

128                      <repeat stmt>
 *   <repeat statement>

129                      <for stmt>
 *   <for statement>

130                      <case stmt>
 *   <case statement>

131                      <with stmt>
 *   <with statement>

132                      <goto stmt>
 *   <goto statement>

133                      <compound stmt>
 *   <compound statement>

134
 *   <empty statement>

135  <assignment stmt>  ::=  <variable> := <expression>
 *   <variable> ; <expression> ;
 *   LITA {variable PRT address}
 *   [STD / STDI / STDB / CNAI ; STDB]

136  <variable>  ::=  <variable ident>
 *   <variable identifier>

137                   <variable> ^
 *   <variable>   {NOT IMPLEMENTED}

138                   <variable> [ <expres list> ]
 *   <variable> ; <expression list> ;
 *   LDII {variable PRT address} ;
 *   SUB {number of array indicies}

139                   <variable> . <field ident>
 *   <variable> ; <field identifier>
 *   {modify PRT location and variable type}

140  <variable ident>  ::=  <identifier>
 *   <identifier>   {set variable type and PRT location}
```

```
141  <expres list>  ::=  <expression>
 *   <expression>

142                      <expres list> , <expression>
 *   <expression list> ; <expression>

143  <expression>  ::=  <simple expression>
 *   <simple expression>

144                     <simple expression>
                        <relational operator>
                        <simple expression>
 *   <simple expression> ; <relational operator> ;
 *   <simple expression> ;
 *   [EQUI / NEQI / LEQI / GEQI / LSSI / GRTI /
      EQLB / NEQB / LEQB / GEQB / LSSB / GRTB /
      EQLS / NEQS / LEQS / GEQS / LSSS / GRTS /
      EQSET / NEQST / INCL1 / INCL2 / XIN]

145  <relational operator> ::=  =
 *   {set relational operator}

146                             < >
 *   {set relational operator}

147                             < =
 *   {set relational operator}

148                             > =
 *   {set relational operator}

149                             <
 *   {set relational operator}

150                             >
 *   {set relational operator}

151                             IN
 *   {set relational operator}

152  <term>  ::=  <factor>
 *   <factor>

153              <term> <multiplying operator> <factor>
 *   <term> ; <multiplying operator> ; <factor> ;
 *   [MULI / MULB / UNION / CNVI ; CN2I ; DIVB /
      DIVB / DIVI / MODX / ANDX]

154  <multiplying operator>  ::=  *
 *   {set operator type}

155                               /
 *   {set operator type}
```

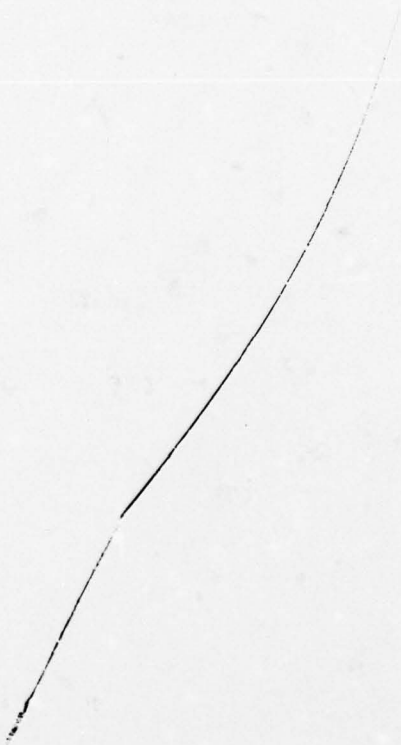
```
156                                   DIV
 *   {set operator type}

157                                   MOD
 *   {set operator type}

158                                   AND
 *   {set operator type}

159 <simple expression>  ::=  <term>
 *   <term>

160                                   <sign> <term>
 *   <sign> ; <term> ; [NEGI / NEGB]

161                                   <simple expression>
                                      <adding operator> <term>
 *   <simple expression> ; <adding operator> ;
 *   <term> ;
 *   [UNION / ADDI / ADDB / STDIF / SUBI / SUBB / BOR]

162 <adding operator>  ::=  +
 *   {set operator type}

163                                   -
 *   {set operator type}

164                                  OR
 *   {set operator type}

165 <factor>  ::=  <variable>
 *   <variable> ;
 *   [ {built-in-function identifier} / LITA {PRT location} ;
 *    [LODI / LOD / LODB] / LDII {value} / LDII {value} ; NEGI /
     LDBI {value} / LDRI {value} ; NEGB ]

166                  <variable> ( <actual para list> )
 *   <variable> ; <actual parameter list> ;
 *   {verify parameter count}
 *   [ {built-in function identifiers} / LITA {PRT location} ;
     [ LOD / LODI / LODB ] ]

167                    ( <expression> )
 *   <expression>

168                    <set>
 *   <set>   {NOT IMPLEMENTED}

169                    NOT <factor>
 *   <factor> ; NOTX

170                    <number>
 *   <number>
 *   [LDII {integer value} / LDIB {BCD real value}]
```

```
171                 NIL

172                 <string>
 *   <string> ; LDSI {ACCUM} ; NOP

173 <actual para list>  ::=  <actual para>
 *   <actual parameter>   {initialize parameter count}

174                           <actual para list> ,
                              <actual para>
 *   <actual parameter list> ; <actual parameter>
 *   {incriment parameter count}

175 <set>  ::=  [ <element list> ]
 *   <element list>   {NOT IMPLEMENTED}

176 <element list> ::=
 *   <empty>   {NOT IMPLEMENTED}

177                       <xelement list>
 *   <xelement list>   {NOT IMPLEMENTED}

178 <xelement list>  ::=  <element>
 *   <element>   {NOT IMPLEMENTED}

179                         <xelement list> , <element>
 *   <xelement list> ; <element> {NOT IMPLEMENTED}

180 <element>  ::=  <expression>
 *   <expression>

181                 <expression> .. <expression>
 *   <expression> ; <expression>

182 <goto stmt>  ::=  GOTO <label>
 *   <label> ; BRL {PRT location}

183 <compound stmt>  ::=  BEGIN <stmt lists> END
 *   <statement lists>

184 <stmt lists>  ::=  <stmt>
 *   <statement>

185                     <stmt lists> ; <stmt>
 *   <statement lists> ; <statement>

186 <procedure stmt>  ::=  <procedure ident>
 *   <procedure identifier> ;
 *   [ {built-in procedure identifier} / PRO {procedure label} ;
 *   DEL ]

187                       <procedure ident> (
                          <actual para list> )
```

```
*    <procedure identifier> ; <actual parameter lists> ;
*    {verify parameter count}
*    [ {built-in procedure identifier} / PRO {procedure label} ;
*    DEL ]

188  <procedure ident>  ::=  <identifier>
*    <identifier>
*    [ {determine which built-in procedure} /
      {store procedure attributes in stacks} ]

189  <actual para>  ::=  <expression>
*    <expression> ;
*    [ PARMX {if READPARMS = false} /
       [ WRT / WRTB / WRTI / WRTS ] {if WRITE$STMT = true} /
       [ RDV / RDVB / RDVI / RDVS ] {if READ$STMT = true} ]

190                      <expression> : <expression>
*    <expression> ; <expression> ;
*    [ WRTB 01 / WRTS 01 ]

191                      <expression> : <expression> :
                         <expression>
*    <expression> ; <expression> ; <expression> ;
*    WRTB 02

192  <case stmt>  ::=  <case express> <case list elemt list>
                       END
*    <case express> ; <case list element list> ;
*    { set LABEL$STACK(MP)}   LBL {LABEL$STACK(MP)}

193  <case express>  ::=  CASE <expression> OF
*    <expression>   {CASE$STMT = true}
*    {set LABEL$STACK(MP) , incriment CASE$COUNT}

194  <case list elemt list>  ::=  <case list element>
*    <case list element> ; BRL {LABEL$STACK(MP-1)} ;
*    LBL {LABEL$STACK(MP)+1}

195                              <case list elemt list> ;
                                 <case list element>
*    <case list element list> ; <case list element> ;
*    BRL {LABEL$STACK(MP-1)} ; LBL {LABEL$STACK(SP)+1}

196  <case list element>  ::=
*    <empty>   {CASE$STMT = false}

197                            <case prefix> <stmt>
*    <case prefix> ; <statement>

198  <case prefix>  ::=  <case label list> :
*    <case label list> ; BRL {LABEL$STACK(MP)+1} ;
*    LBL {LABEL$STACK(MP)}

199   <with stmt>  ::=  <with> <rec variable list> <do>
```

```
                                <bal stmt>
 *    <with> ; <record variable list> ; <do> ;
 *    <balanced statement>   {NOT IMPLEMENTED}

200   <with>  ::=  WITH
 *    {NOT IMPLEMENTED}

201   <rec variable list>  ::=  <variable>
 *    <variable>

202                               <rec variable list> ,
                                  <variable>
 *    <record variable list> ; <variable>

203   <do>  ::=  DO
 *    {set LABEL$STACK(SP)}
 *    BLC {LABEL$STACK(SP)}

204   <while stmt>  ::=  <while> <expression> <do> <bal stmt>
 *    <while> ; <expression> ; <do> ; <balanced statement> ;
 *    BRL {LABEL$STACK(MP)} ; LBL {LABEL$STACK(SP-1)}

205   <while>  ::=  WHILE
 *    {set LABEL$STACK(SP)}
 *    LBL {LABEL$STACK(SP)}

206   <for stmt>  ::=  FOR <control variable> := <for list>
                       DO <bal stmt>
 *    <control variable> ; <for list> ; <do> ;
 *    <balanced statement> ; BRL {LABEL$STACK(SP-2)+1} ;
 *    LBL {LABEL$STACK(SP-1)}

207   <for list>  ::=  <initial value> <to> <final value>
 *    <initial value> ; <to> ; <final value> ;
 *    GEQI

208                    <initial value> <downto> <final value>
 *    <initial value> ; <downto> ; <final value>
 *    LEQI

209   <control variable>  ::=  <identifier>
 *    <identifier>   {set variable type and PRT location}

210   <initial value>  ::=  <expression>
 *    <expression> ;
 *    LITA {control variable PRT location} ; [ STOI / STO ] ;
 *    {set LABEL$STACK(SP)} ; BRL {LABEL$STACK(SP)} ;
 *    LBL {LABEL$STACK(SP)+1} ;
 *    LITA {control variable PRT location} ; [ STDI / STD ]

211   <final value>  ::=  <expression>
 *    <expression>

212   <repeat stmt>  ::=  <repeat>  <stmt lists> UNTIL
```

```
                                    <expression>
 *    <repeat> ; <statement lists> ; <expression> ; NOTX ;
 *    BLC {LABEL$STACK(MP)}

213  <repeat>  ::=  REPEAT
 *    { set LABEL$STACK(SP)}
 *    LBL {LABEL$STACK(sp)}

214  <to>  ::=  TO
 *    INC ; LBL {LABEL$STACK(SP -1)}

215  <downto>  ::=  DOWNTO
 *    DEC ; LBL {LABEL$STACK(SP-1)}
```



APPENDIX E - INOPERATIVE CONSTRUCTS

The accompanying list shows the NPS-PASCAL constructs that had not been fully implemented at the start of this project. Those constructs requiring further work and testing at project completion are denoted with an asterisk.

Since the original work on NPS-PASCAL was not based on STANDARD PASCAL CONSTRUCTS, it was necessary to first develop a new grammar utilizing the required special characters and reserved words, and following the rules stipulated in STANDARD PASCAL. Consequently, the inoperative constructs listing is based on STANDARD PASCAL constructs, and does not include PASCAL language extensions contained in NPS-PASCAL.

- Program parameters specified in the program heading
- Procedure and Function Declaration Part
- Block structure
- The WHILE statement
- The FOR statement
- The CASE statement
- Array access
- Packed arrays *
- Record variant part
- Record tag field *
- Record component access *
- String Operators
- The WITH statement*
- Set assignments *

- Set operators
- Set relational operators
- File buffer variables *
- Standard file-handling operators
- Textfiles
- Pointer variables *
- Procedure and Function Parameters
- Recursive execution of Procedures and Functions
- Procedure statements
- Function values
- Directives
- Standard Procedures
- Standard Functions
- Program input from input file
- Program output to output file
- Strings
- Input and Output to non-standard files *

APPENDIX F - INTERMEDIATE CODE DECODE PROGRAM

Since the NPS-PASCAL compiler was developed without the parallel development of the 8080 Code Generator, an alternative means of checking the generated intermediate code was developed. The result is the PL/M program DECODE shown in the program listings.

DECODE opens the compiler generated intermediate code file (<FILENAME>. PIN) and converts the hexadecimal values into the same NPS-PASCAL mneumonics found in the compiler. The parameters associated with certain operators such as labels, branches, and load immediate mneumonics, are printed out immediately following the operator. Integer and real numbers are printed out in decimal format for ease of readability. Strings are displayed in their ASCII character format.

The execution procedure for viewing the intermediate code is as follows:

(1). Compile an NPS-PASCAL program using the command:
PASCAL <FILENAME> .

(2). Upon successful compilation, with no program errors, input the command: DECODE <FILENAME> .

The contents of the <FILENAME> . PIN file will be printed out on the console. The DECODE program leaves the intermediate code intact in its file for further use in the Code Generator program.

APPENDIX G - SYMBOL TABLE DISPLAY PROGRAM

A symbol table display program was developed as an aid to compiler development, and particularly, for use in program debugging. SYMBOLTABLE is a PL/M program that prints out the information stored in the SYM file created by the NPS-PASCAL compiler. This program is contained in the program listings section.

SYMBOLTABLE contains many of the same features as DECODE to increase readability. Integer and BCD real values stored in the symbol table are output in decimal form. PRT addresses are also displayed in decimal format. Printnames, strings, and scalars are all shown as ASCII characters. Hash collisions with other program identifiers are indicated. Identifier types are specified along with any additional data that singles out this identifier.

Of course, program execution requires the NPS-PASCAL program in question to be compiled first. After receiving the compilation complete message on the monitor (a zero error count is not required), the user can print out the program symbol table information using the command: SYMBOLTABLE <FILENAME>. The symbol table file remains intact on the disk for use during the 8080 code generation phase.

```
100H:                    /* LOAD POINT FOR THE COMPILER */

     /*
      ***************************************************************
      *                                                             *
      *                                                             *
      *                    NPS-PASCAL COMPILER                      *
      *                                                             *
      *                                                             *
      *                U. S. NAVAL POSTGRADUATE SCHOOL              *
      *                     MONTEREY, CALIFORNIA                    *
      *                                                             *
      *                  WRITTEN AND DEVELOPED BY:                  *
      *                       JCHN L. BYRNES                        *
      *                       JACK C. GRACIDA                       *
      *                     ROBERT R. STILWELL                      *
      *                                                             *
      *                          JUNE 1979                          *
      *                                                             *
      *                                                             *
      ***************************************************************
     */

/*************************************************************************/
/*************************************************************************/
/***                    S Y S T E M   L I T E R A L S                  ***/
/*************************************************************************/
/*************************************************************************/

DECLARE LIT      LITERALLY 'LITERALLY',
        CR               LIT '13',
```

```
37:         LF              LIT '0AH',
38:         DCL             LIT 'DECLARE',
39:         POS             LIT '0',
40:         NEG             LIT '1',
41:         TAB             LIT '09H',
42:         PROC            LIT 'PROCEDURE',
43:         BDOS            LIT '5H',           /*ENTRY POINT TO DISK OP. SYS*/
44:         BOOT            LIT '0',            /* EXIT TO RETURN TO OP. SYS. */
45:         TRUE            LIT '1',
46:         ADDR            LIT 'ADDRESS',
47:         FALSE           LIT '0',
48:         COMMENT         LIT '7BH',
49:         UNCOMMENT       LIT '7DH',
50:         FILEEOF         LIT '1',
51:         FOREVER         LIT 'WHILE TRUE',
52:         STATESIZE       LIT 'ADDRESS',
53:         INDEXSIZE       LIT 'ADDRESS',
54:         BUILT$IN$PROC     LIT '0CH',
55:         BUILT$IN$FUNC LIT '0DH',
56:         QUESTIONMARK  LIT '3FH',
57:         CONS$STR$TYPE LIT '3',
58:         CONS$NUM$TYPE LIT '0',
59:         CONS$IDENT$TYPE LIT '1',
60:         CONS$SIDENT$TYPE LIT '2';
61:
62: DCL
63:         IDENTSIZE       LIT  '32',  /* MAX IDENTIFIER SIZE + 1 */
64:         VARCSIZE        LIT  '100', /* SIZE OF VARC STACK*/
65:         PSTACKSIZE      LIT  '48',  /* SIZE OF PARSE STACKS */
66:         FOLCHAR         LIT  '0DH', /* END OF SOURCE LINE CHARACTER */
67:         SOURCERECSIZE   LIT  '128', /* SIZE OF SOURCE FILE RECORD */
68:         INTRECSIZE      LIT  '128', /* INTERMEDIATE FILE RECORD SIZE */
69:         CONBUFFSIZE     LIT  '82',  /* SIZE OF CONSOLE BUFFER */
70:         HASHTBLSIZE     LIT  '128', /* SIZE OF HASHTABLE */
71:         HASHMASK        LIT  '127', /* HASHTABLE SIZE -1 */
72:         EOFFILLER       LIT  '1AH', /* CHAR FOR LAST RECORD ON FILE */
```

```
 73:          STRINGDELIM     LIT   '27H', /* CHAR USED TO DELIMIT STRINGS */
 74:          CONTCHAR        LIT   '5CH', /* CONTINUATION CHARACTER */
 75:          MAXONCOUNT      LIT   '25',  /* MAX NUMBER ON STATEMENTS */
 76:          MAXINT          LIT   '32767', /* MAX INTEGER VALUE */
 77:          BCDSIZE         LIT   '8',   /* BYTES USED FOR BCD VALUES */
 78:          MAX$NEST        LIT   '3',   /* MAX LEVEL OF NESTS FOR TYPES */
 79:          ARRY$NEST       LIT   '4',   /* MAX NESTING LEVEL FOR ARRAYS */
 80:          MAX$ARRY$DIM    LIT   '5',   /* MAX ARRY DIMENSIONS */
 81:          FORMMASK        LIT   '7',   /* USED TO DETERMINE FORM TYPE */
 82:          TYPEMASK        LIT   '38H', /* USED TO DETERMINE FORM FIELD*/
 83:
 84: /* FORM ENTRIES */
 85:
 86:          LABL$ENTRY              LIT   '0',
 87:          CONS$ENTRY              LIT   '1',
 88:          TYPE$ENTRY              LIT   '2',
 89:          VAR$ENTRY               LIT   '3',
 90:          PROC$ENTRY              LIT   '4',
 91:          FUNC$ENTRY              LIT   '5',
 92:          FILE$ENTRY              LIT   '6',
 93:          TYPE$DCLE               LIT   '7',
 94:
 95: /* NUMBER TYPES */
 96:
 97:          ORD$TYPE                LIT   '0',
 98:          INTEGER$TYPE            LIT   '1',
 99:          CHAR$TYPE               LIT   '2',
100:          UNSIGN$EXPON            LIT   '3',
101:          SIGNED$EXPON            LIT   '4',
102:          BOOLEAN$TYPE            LIT   '5',
103:          REAL$TYPE               LIT   '2',
104:          COMPLEX$TYPE            LIT   '4',
105:          STRING$TYPE             LIT   '4';
106:
107:
108:          /* MANY OF THE FOLLOWING VARIABLES CAN BE REPLACED BY MAKING
```

```
109:              USE OF THE PARALLEL PARSE STACKS                          */
110: DCL
111:          SIGNTYPE        BYTE,
112:          CONST$TYPE      BYTE,     /* TYPE OF CONSTANT */
113:          FORM            BYTE,
114:          EXPON           BYTE,
115:          VECPTR          BYTE,
116:          TYPENUM         BYTE,
117:          CONST$PTR       BYTE,
118:          STARTBDOS       ADDR INITIAL(6H), /*ADDR OF PTR TO TOP OF BDOS*/
119:          MAX             BASED STARTBDOS ADDR,
120:          TYPE$ADDR       ADDR,
121:          TYPE$LOCT       ADDR,
122:          VAR$PTR         BYTE,
123:          VAR$PARM$PTR    BYTE,
124:          ALOCBASICTYP    BYTE,
125:          ARRY$QTY(MAX$ARRY$DIM) ADDR,
126:          VAR$BASE(10)    ADDR,
127:          VAR$BASE1(10)   ADDR,
128:          TYPE$INDX       BYTE,
129:          ALLC$QTY        ADDR,
130:          TYPEFORM        BYTE,
131:          TYPE$ORD$NUM    BYTE,
132:          PARENT$TYPE     ADDR,
133:          CONST$INDX      BYTE,
134:          LOOKUP$ADDR     ADDR,
135:          CONST$VEC(4)    BYTE,
136:          CONST$VALUE(16) BYTE,
137:          CONST$PN$HASH(4) BYTE,
138:          CONST$PN$PTR    BYTE,
139:          CONST$PN$SIZE(4) BYTE,
140:          INTEGER$DIFF    ADDR,
141:          SUBR$VAL(2)     ADDR,
142:          SUBR$TYPE(2)    BYTE,
143:          SUBR$PTR        BYTE,
144:          SUB$TYP$ADDR    ADDR,
```

```
145:          SUBR$FORM          BYTE,
146:          SIGNVALU      BYTE,
147:          ARRY$BASE          ADDR,
148:          ARRY$PTR           BYTE,
149:          ARRY$DIM$PTR       BYTE,
150:          PTRPTR             BYTE,
151:          TAG$FD(MAX$NEST) BYTE,
152:          VAR$CAS$TP(MAX$NEST) ADDR,
153:          VAR$CAS$VAL(MAX$NEST) ADDR,
154:          REC$VAR$TYP(MAX$NEST) BYTE,
155:          REC$NST      BYTE  INITIAL (255),
156:          RECORD$PTR       BYTE,
157:          REC$ADDR(10)       ADDR,
158:          REC$PAR$ADR(MAX$NEST) ADDR,
159:          VARIANT$PART(MAX$NEST)     BYTE,
160:          FXD$OFST$BSE(MAX$NEST) ADDR,
161:          VAR$OFST$BSE(MAX$NEST) ADDR,
162:          CUR$OFST(MAX$NEST)       ADDR,
163:          NUM$ARRY$DIM(MAX$ARRY$DIM) BYTE,
164:          ARRY$DIMEN(25) ADDR,
165:          ARY$DM$ADR$PTR BYTE,
166:          /* CASE STATEMENT VARIABLES */
167:          CASE$STK(12)    BYTE,    /* NUMBER OF STMTS IN CURRENT CASE */
168:          CASE$COUNT    BYTE    INITIAL(255),  /* LEVEL OF CASE STMTS */
169:          CONST$NUM$TYPE(4) BYTE;
170:
171:
172: /*****************************************************************************/
173: /*****************************************************************************/
174: /***                          GLOBAL VARIABLES                             ***/
175: /*****************************************************************************/
176: /*****************************************************************************/
177:
178:
179: DCL       BCDNUM(BCDSIZE) BYTE,
180:          SCOPE(10) ADDR,
```

```
181:          SCOPE$NUM BYTE,
182:          TEMPBYTE BYTE,
183:          TEMPBYTE1 BYTE,
184:          TEMPADDR ADDR,
185:          TEMPADDR1 ADDR,
186:          PRODUCTION    BYTE,
187:          PRV$SBT$ENTRY ADDR;
188:
189: DCL
190:               /* COMPILER TOGGLES */
191:
192:           LIST$TOKEN     BYTE INITIAL(FALSE),
193:           LIST$PROD      BYTE INITIAL(FALSE),
194:           LIST$SOURCE    BYTE INITIAL(FALSE),
195:           DEBUG$LN       BYTE INITIAL(FALSE),
196:           LOWER$TO$UPPER BYTE INITIAL(TRUE),
197:           COMPILING BYTE,
198:
199:               /* COUNTERS */
200:
201:           CODESIZE       ADDR,            /* COUNTS NUMBER OF LABELS */
202:           LABLCOUNT      ADDR INITIAL(0), /* COUNTS NUMBER OF LABELS */
203:           ERRORCOUNT     ADDR INITIAL(0), /* COUNTS NUMBER OF ERRORS */
204:           ALLOC$ADDR     ADDR INITIAL(0), /* COUNTS PRT ENTRIES */
205:
206:               /* FLAGS USED DURING CODE GENERATION */
207:
208:           NOINTFILE      BYTE INITIAL(FALSE), /* NO INTERMEDIATE FILE */
209:           CASE$STMT      BYTE INITIAL(FALSE), /* IN CASE STATEMENT */
210:           WRITE$STMT     BYTE INITIAL(FALSE), /* IN WRITE STATEMENT */
211:           READ$STMT      BYTE INITIAL(FALSE), /* IN READ STATEMENT */
212:           NEW$STMT       BYTE INITIAL(FALSE), /* GETS NEW RECORD */
213:           DISPOSE$STMT   BYTE INITIAL(FALSE), /* DISPOSES OF RECORD */
214:           ALLOCATE       BYTE,     /* PRT LOCATION ASSIGNED */
215:           VARPARM        BYTE,     /* FORMAL PARAM IS VARIABLE TYPE */
216:           READPARMS      BYTE,     /* READING ACTUAL PARAMETERS */
```

```
217:          PRESENT          BYTE,     /* IDENTIFIER IS IN SYMBOL TABLE */
218:          NO$LOOK          BYTE,     /* CONTROLS CALLS TO SCANNER */
219:          SIGN$FLAG        BYTE,     /* SET WHEN SIGN PRECEDES ID */
220:
221:               /* GLOBAL VARIABLES USED BY THE SCANNER */
222:
223:          TOKEN      BYTE, /* TYPE OF TOKEN JUST SCANNED */
224:          HASHCODE   BYTE, /* HASH VALUE OF CURRENT TOKEN */
225:          NEXTCHAR   BYTE, /* CURRENT CHARACTER FROM GETCHAR */
226:          CONT       BYTE, /* INDICATES FULL ACCUM--STILL MORE */
227:          ACCUM(IDENTSIZE)  BYTE, /* HOLDS CURRENT TOKEN */
228:          TEMPCHAR1 BYTE, /* HOLDS PREVIOUSLY SCANNED TOKEN */
229:
230:               /* GLOBAL VARIABLES USED IN SYMBOL TABLE OPERATIONS */
231:
232:          BASE             ADDR, /* BASE LOCATION OF ENTRY */
233:          HASHTABLE(HASHTBLSIZE)   ADDR, /* HASHTABLE ARRAY */
234:          SBTBLTOP         ADDR, /* HIGHEST LOCATION OF SYMBOL TABLE */
235:          SBTBL            ADDR, /* CURRENT TOP OF SYMBOL TABLE */
236:          PTR BASED BASE BYTE, /* FIRST BYTE OF ENTRY */
237:          APTRADDR         ADDR, /* UTILITY VARIABLE TO ACCESS SBTBL */
238:          ADDRPTR BASED APTRADDR ADDR, /* CURRENT 2 BYTES POINTED AT */
239:          BYTEPTR BASED APTRADDR BYTE, /* CURRENT BYTE POINTED AT */
240:          PRINTNAME        ADDR, /* SET PRIOR TO LOOKUP OR ENTER */
241:          SYMHASH          BYTE, /* HASH VALUE OF AN IDENTIFIER */
242:          LAST$SBTBL$ID    ADDR, /* HOLD PREVIOUS BASE LOCATION */
243:          PARAMNUMLOC      ADDR, /* STORES POINTER TO PARAM LISTING */
244:          SBTBLSCOPE       ADDR; /* BASE OF LAST ENTRY IN PREVIOUS BLOCK
245: */
246:
247:      /*
248:       ***************************************************************
249:       *                                                             *
250:       *                  BUILT-IN SYMBOL TABLE                      *
251:       *                                                             *
252:       ***************************************************************
```

```
253:      */
254:
255:
256: DECLARE BUILT$IN$TBL DATA(0,0,0,0,42H,14,7,'I','N','T','E','G','E','R',
257:     0,0,01H,06H,4AH,36,4,'R','E','A','L',
258:     0,0,01H,14H,52H,30,4,'C','H','A','R',
259:     0,0,01H,1FH,5AH,0,7,'B','O','O','L','E','A','N',
260:     0,0,01H,2AH,62H,69,4,'T','E','X','T',
261:     0,0,01H,38H,0EH,16,5,'I','N','P','U','T',
262:     0,0,01H,43H,1EH,113,6,'O','U','T','P','U','T',
263:     0,0,01H,4FH,0DH,86,3,'A','B','S',0,13H,1,13H,
264:     0,0,01H,5CH,0DH,118,3,'S','Q','R',1,13H,1,13H,
265:     0,0,01H,6AH,0DH,106,3,'S','I','N',2,3H,1,13H,
266:     0,0,01H,78H,0DH,101,3,'C','O','S',3,3H,1,13H,
267:     0,0,01H,86H,0DH,57,6,'A','R','C','T','A','N',4,3H,1,13H,
268:     0,0,01H,94H,0DH,109,3,'E','X','P',5,3H,1,13H,
269:     0,0,01H,0A5H,0DH,26,2,'L','N',6,3H,1,13H,
270:     0,0,01H,0B3H,0DH,74,4,'S','Q','R','T',7,3H,1,13H,
271:     0,0,01H,0C0H,0DH,87,3,'O','D','D',8,5H,1H,1H,
272:     0,0,01H,0CFH,0DH,46,4,'E','O','L','N',9,5H,1,06H,
273:     0,0,01H,0DDH,0DH,90,3,'E','O','F',10,5H,1,06H,
274:     0,0,01H,0ECH,0DH,12,5,'T','R','U','N','C',11,1H,1,3H,
275:     0,0,01H,0FAH,0DH,8,5,'R','O','U','N','D',12,1H,1,3H,
276:     86H,01H,02H,0AH,0DH,101,3,'O','R','D',13,1H,1,2H,
277:     0,0,02H,1AH,0DH,93,3,'C','H','R',14,2H,1,1H,
278:     0DDH,01H,02H,28H,0DH,46,4,'S','U','C','C',15,0F3H,1,0F3H,
279:     0,0,02H,36H,0DH,43,4,'P','R','E','D',16,0F3H,1,0F3H,
280:     0,0,02H,45H,0CH,121,3,'P','U','T',17,10H,06H,
281:     0,0,02H,54H,0CH,96,3,'G','E','T',18,10H,06H,
282:     0,0,02H,61H,0CH,03,5,'R','E','S','E','T',19,10H,06H,
283:     0,0,02H,6EH,0CH,34,7,'R','E','W','R','I','T','E',20,10H,06H,
284:     0,0,02H,7DH,0CH,29,4,'P','A','G','E',21,10H,06H,
285:     79H,01H,02H,8EH,0CH,106,3,'N','E','W',22,0FFH,
286:     0,0,02H,9CH,0CH,23,7,'D','I','S','P','O','S','E',23,0FFH,
287:     0,0,02H,0A8H,09H,64,4,'T','R','U','E',0,0,0,0,
288:     0,0,02H,0B8H,09H,107,5,'F','A','L','S','E',0,0,1,0,
```

```
289:      0,0,02H,0C7H,0CH,28,4,'R','E','A','D',24,0FFH,
290:      0,0,02H,0D7H,0CH,54,6,'R','E','A','D','L','N',25,0FFH,
291:      0,0,02H,0E4H,0CH,11,5,'W','R','I','T','E',26,0FFH,
292:      0,0,02H,0F3H,0CH,37,7,'W','R','I','T','E','L','N',27,0FFH,
293:      0,0,03H,01H,0CH,40,4,'S','E','E','K',28,2,06H,01H,
294:      0,0,03H,11H,11H,21,7,'F','O','R','W','A','R','D',
295:      0,0,03H,20H,11H,99,8,'E','X','T','E','R','N','A','L',
296:      0,0,03H,2EH,11H,62,11,'I','N','T','E','R','A','C','T','I','V','E');
297:
298:
299:       /*
300:        ************************************************************
301:        *                                                          *
302:        *                  LALR(1) PARSE TABLES                    *
303:        *                      AND VARIABLES                       *
304:        *                                                          *
305:        ************************************************************
306:       */
307:
308:
309: DECLARE MAXRNO LITERALLY '185',/* MAX READ COUNT */
310:         MAXLNO LITERALLY '242',/* MAX LOOK COUNT */
311:         MAXPNO LITERALLY '269',/* MAX PUSH COUNT */
312:         MAXSNO LITERALLY '484',/* MAX STATE COUNT */
313:         STARTS LITERALLY '1',/* START STATE */
314:         EOFC   LITERALLY '25',/* EOF */
315:         NUMBERC LITERALLY '54',/* NUMBER */
316:         STRINGC LITERALLY '55',/* STRING */
317:         IDENTC LITERALLY '58';/* IDENTIFIER */
318:
319: DECLARE READ1 DATA(0,53,56,57,25,25,25,13,15,34,56,57,58,58,58,9,14,9,58
320:     ,58,58,58,58,15,58,4,10,54,55,58,3,4,6,10,33,37,42,49,50,54,55,58,22
321:     ,3,4,5,10,31,32,54,55,58,3,4,10,54,55,58,58,3,5,31,32,54,55,58,22,58
322:     ,58,58,22,58,5,20,29,35,38,41,43,47,51,54,58,58,54,33,37,42,50,35,58
323:     ,58,58,58,58,20,29,35,38,41,43,47,51,58,40,44,34,56,57,54,58,7,11,26
324:     ,27,30,58,1,1,1,14,43,3,9,17,3,14,15,1,5,6,18,1,3,5,6,9,36,36,39,22
```

```
325:      ,19,8,17,14,14,28,8,9,3,9,28,9,46,22,22,3,12,16,14,15,9,9,8,12,16,9
326:      ,12,24,48,9,9,8,12,12,9,12,14,12,14,12,3,9,8,12,8,12,18,45,58,12,14
327:      ,12,16,12,19,2,4,10,13,15,21,23,4,10,23,9,12,14,9,28,8,9,8,9,0,0,0,0
328:      );
329:
330: DECLARE LOOK1 DATA(0,13,15,0,35,58,0,16,0,58,0,58,0,58,0,35,58,0,9,28,46
331:      ,0,9,28,0,8,9,28,0,15,0,8,9,28,0,8,9,28,0,9,28,36,46,0,36,0,9,28,0
332:      ,35,58,0,17,0,1,5,6,18,0,14,0,0,0,0,40,0,44,0,34,0,43,0,7,11,26,27
333:      ,30,0,7,11,26,27,30,0,7,11,26,27,30,0,9,46,0,36,0,1,3,5,6,0,12,19,0
334:      ,12,19,0,9,28,36,46,0,36,0,9,28,46,0,17,0,14,0,14,0,9,0,9,28,0,43,0
335:      ,9,28,0,12,0,9,0,9,0,12,0,3,0,45,0,45,0,45,58,0,45,0,45,58,0,2,4,10
336:      ,13,15,21,23,0,4,10,23,0);
337:
338: DECLARE APPLY1 DATA(0,0,0,0,0,32,0,170,171,174,175,0,0,0,31,74,79,0,0,0
339:      ,23,0,0,28,29,39,51,54,61,63,150,0,96,0,15,28,29,37,39,47,48,51,52
340:      ,54,59,60,61,62,63,150,0,0,0,24,0,0,47,48,60,62,0,15,37,59,0,18,44
341:      ,45,46,69,123,0,0,0,81,0,0,0,37,0,0,0,14,0,0,26,0,0,5,36,69,0,26,0
342:      ,63,0,0,29,0,0,39,0,0,0,0,0,0,0,25,0,0,0,0,146,0,175,0,1,0,0,8,0,22
343:      ,0,0,67,85,0,0,1,0,0,46,0,0,27,120,122,141,0,71,72,88,89,90,120,121
344:      ,0,71,0,72,88,89,90,121,0,121,0,0,0,0,27,38,71,72,77,88,89,90,101
345:      ,109,120,121,122,141,0,0,0,11,16,17,40,41,49,50,55,56,57,58,70,80,91
346:      ,107,108,0,0,97,160,0,0,181,0,0,65,183,0,13,0,0,0,0,40,0,0,106,0,109
347:      ,0,0,0,42,0,0,0,0,28,0,28,150,0,0,0,0,112,128,0,0,0,0,0,0,0,108,0,0
348:      ,0,0,0);
349:
350: DECLARE READ2(219) ADDR INITIAL
351:                        (0,83,84,86,270,272,271,415,416,67,85,87,377,277,311
352:      ,366,46,273,378,375,355,312,334,417,310,296,297,290,294,295,10,296
353:      ,18,297,66,73,76,81,202,290,294,203,62,11,296,188,297,440,65,439,441
354:      ,409,10,296,297,290,294,203,478,11,188,440,65,439,441,409,59,377,354
355:      ,205,60,303,15,58,64,70,74,469,201,474,482,283,204,289,283,66,73,76
356:      ,202,69,334,276,381,367,374,58,64,70,74,469,201,474,482,204,75,78,68
357:      ,84,86,291,295,423,424,427,425,426,409,2,3,4,393,201,7,365,54,8,44
358:      ,52,5,17,406,56,5,13,17,406,189,199,200,391,462,472,436,55,49,50,324
359:      ,340,192,9,193,452,193,80,197,198,187,41,407,339,51,379,380,21,32
360:      ,444,280,31,483,484,358,359,308,35,40,195,39,467,30,45,33,12,190,456
```

```
361:      ,42,435,42,57,79,364,34,47,37,53,38,472,186,431,432,196,414,420,433
362:      ,431,432,433,191,36,48,194,461,19,22,20,22,0,0,0,0);
363:
364: DECLARE LOOK2(167) ADDR INITIAL
365:                        (0,6,6,418,14,14,243,244,16,23,285,24,299,25,350,26,26
366:      ,245,246,246,246,27,247,247,28,248,248,248,29,43,419,249,249,249,61
367:      ,250,250,250,63,251,251,251,251,71,252,72,253,253,77,82,82,254,295
368:      ,310,409,409,409,409,457,334,310,88,89,90,92,255,93,256,94,257,258
369:      ,95,98,98,98,98,98,428,99,99,99,99,99,429,100,100,100,100,100,430
370:      ,259,259,109,392,384,116,116,116,116,434,470,470,117,471,471,118,260
371:      ,260,260,260,120,261,121,262,262,262,122,130,449,131,458,132,459,135
372:      ,325,263,263,141,357,146,264,264,150,157,446,158,336,159,335,163,373
373:      ,164,455,165,265,170,266,171,171,267,174,268,175,175,269,177,177,177
374:      ,177,177,177,177,412,178,178,178,413);
375:
376: DECLARE APPLY2(248) ADDR INITIAL
377:                         (0,0,236,149,136,275,274,103,362,104,360,102,210,152
378:      ,282,451,281,105,209,119,287,286,145,344,344,344,288,309,344,344,344
379:      ,111,293,292,96,96,96,96,96,96,96,96,96,96,96,96,96,96,96,97,211
380:      ,166,301,300,114,353,330,347,323,302,322,322,346,304,348,382,376,382
381:      ,139,140,307,156,305,314,313,315,173,321,320,319,316,134,133,227,329
382:      ,328,180,408,333,144,332,326,327,233,232,123,338,337,161,343,342,317
383:      ,345,318,306,212,179,352,351,172,106,229,155,154,361,363,239,240,110
384:      ,185,184,369,368,234,371,372,370,162,237,238,113,148,147,278,454,369
385:      ,386,466,453,387,387,468,473,475,217,392,384,390,385,222,222,222,222
386:      ,222,221,125,124,223,388,394,115,220,115,115,115,115,115,115,219,115
387:      ,115,115,115,115,218,405,143,129,224,410,224,411,226,460,450,404,479
388:      ,126,127,481,480,128,480,225,181,214,215,213,183,242,241,160,438,422
389:      ,421,168,167,437,151,231,448,447,401,383,402,138,137,395,235,443,442
390:      ,399,230,182,464,463,228,228,142,400,101,176,208,207,206,396,107,398
391:      ,112,169,153,477,476,397,216,91,108);
392:
393: DECLARE INDEX1(485) ADDR INITIAL
394:                        (0,1,4,5,6,22,7,9,9,13,14,43,43,43,89,52,43,43,24,15
395:      ,16,17,9,83,71,68,89,73,25,25,18,84,13,19,20,21,22,52,116,25,43,43
396:      ,43,23,24,24,24,30,30,43,43,25,30,42,25,43,43,43,43,52,30,25,30,25
```

```
397:      ,58,59,66,67,68,69,43,95,95,70,84,71,72,73,83,84,43,85,89,91,92,67
398:      ,93,94,95,95,95,43,104,105,106,107,109,59,111,111,111,116,117,118
399:      ,119,120,121,43,43,73,122,124,141,125,127,128,132,128,128,136,73,95
400:      ,73,24,137,138,139,140,141,142,143,144,145,146,147,148,149,150,152
401:      ,154,155,73,156,157,159,160,107,161,162,163,25,165,166,168,170,171
402:      ,172,174,175,175,59,176,178,180,181,186,182,183,185,187,188,188,190
403:      ,192,188,188,194,196,203,206,207,43,209,59,211,213,1,4,7,9,11,13,15
404:      ,18,22,25,29,31,35,39,44,46,49,52,54,59,61,62,63,64,66,68,70,72,78
405:      ,84,90,93,95,100,103,106,111,113,117,119,121,123,125,128,130,133,135
406:      ,137,139,141,143,145,147,150,152,155,163,331,445,331,403,465,341,341
407:      ,341,403,403,403,331,298,284,349,356,403,403,403,403,403,465,279,279
408:      ,279,279,279,1,1,1,2,3,3,4,5,7,12,12,13,13,14,18,18,19,19,20,22,23
409:      ,23,23,23,23,32,34,34,51,51,52,52,53,55,56,56,56,61,61,61,65,72,72
410:      ,73,73,74,74,74,74,76,77,77,78,80,81,82,82,82,84,84,85,85,87,87,88
411:      ,92,92,94,94,96,97,97,99,99,100,102,103,104,105,106,106,107,107,108
412:      ,110,110,111,111,112,112,113,113,113,113,115,117,117,119,120,120,122
413:      ,122,122,122,124,124,125,128,128,129,129,131,132,134,135,135,135,140
414:      ,140,148,148,150,156,158,159,159,159,159,159,159,159,159,159,159,160
415:      ,161,161,161,161,176,177,177,178,178,195,195,195,195,195,195,195,196
416:      ,196,199,199,199,199,199,200,200,200,202,202,202,203,203,203,203,203
417:      ,203,203,203,206,206,208,209,209,210,210,211,211,213,214,216,216,218
418:      ,218,219,220,220,220,222,223,224,224,225,225,227,230,231,232,232,233
419:      ,236,237,238,239,239,240,241,242,244,245,246,247);
420:
421: DECLARE INDEX2 DATA(0,3,1,1,1,1,2,4,4,1,1,9,9,9,2,6,9,9,1,1,1,1,4,1,1,1
422:      ,2,10,5,5,1,1,1,1,1,1,1,6,1,5,9,9,9,1,1,1,1,12,12,9,9,5,12,1,5,9,9,9
423:      ,9,6,12,5,12,5,1,7,1,1,1,1,9,9,9,1,1,1,1,10,1,1,9,4,2,1,1,1,1,1,9,9
424:      ,9,9,1,1,1,2,2,7,5,5,5,1,1,1,1,1,1,9,9,10,2,1,1,2,1,4,4,3,3,1,10,9
425:      ,10,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,2,2,1,1,10,1,2,1,1,2,1,1,2,5,1,2,2,1
426:      ,1,2,1,1,1,7,2,2,1,1,1,1,2,2,1,1,2,2,2,1,2,2,7,3,1,2,9,2,7,2,2,3,3,2
427:      ,2,2,2,3,4,3,4,2,4,4,5,2,3,3,2,5,2,1,1,1,2,2,2,2,6,6,6,3,2,5,3,3,5,2
428:      ,4,2,2,2,2,3,2,3,2,2,2,2,2,2,2,3,2,3,8,4,14,16,26,27,28,29,61,63,71
429:      ,72,77,82,92,93,94,95,109,120,121,122,141,150,165,170,171,174,175,3
430:      ,3,3,5,0,2,0,0,5,0,2,0,2,0,0,2,0,2,2,0,0,1,0,1,0,0,0,0,0,2,0,2,2,0,0
431:      ,0,0,0,2,2,0,0,2,0,1,0,0,0,0,5,0,2,0,0,2,0,2,0,0,2,2,0,0,2,0,4,3,0,2
432:      ,1,3,0,0,2,0,2,0,2,1,0,2,0,2,2,0,2,0,0,1,2,1,1,1,1,0,1,4,1,0,2,0,1,1
```

```
433:      ,1,0,2,2,0,2,3,6,1,0,0,0,0,1,3,0,1,3,2,0,1,0,0,0,0,0,0,0,0,0,0,2,0,1
434:      ,3,2,0,0,2,0,2,0,1,1,1,0,0,0,0,2,0,0,0,0,0,0,1,2,0,0,0,0,3,2,0,1,0,0
435:      ,0,0,2,2,0,0,0,2,0,2,1,2,0,2,0,3,0,0,2,4,2,2,0,2,0,1,1,3,0,0,2,0,3,0
436:      ,5,2,2,0,0,0,3,0,0,0);
437:
438:
439:       /*
440:        ***************************************************************
441:        *                        SECTION 2                            *
442:        *                                                             *
443:        *            SYSTEM DEPENDENT ROUTINES AND VARIABLES          *
444:        *                                                             *
445:        *      THE FOLLOWING ROUTINES ARE USED BY THE COMPILER        *
446:        *      TO ACCESS DISK FILES AND THE CONSOLE.  THESE           *
447:        *      ROUTINES ASSUME THE USE OF THE CP/M OPERATING          *
448:        *      SYSTEM.                                                *
449:        *                                                             *
450:        *      THE FCB'S ARE USED BY THE SYSTEM TO MAINTAIN           *
451:        *      INFORMATION ON OPEN FILES.  THEY ARE ONLY USED BY      *
452:        *      PROCEDURES IN THIS SECTION.  THE BUFFERS AND POINTERS  *
453:        *      TO THE BUFFERS ARE USED BY THE REMAINDER OF THE        *
454:        *      PROGRAM, BUT THEIR SIZE MAYBE VARIED TO SUIT THE DISK  *
455:        *      OPERATING SYSTEM BEING USED.                           *
456:        *                                                             *
457:        ***************************************************************
458:       */
459:
460:
461: DCL
462:      /* NOTE: CP/M PROVIDES 5CH AS FCB AREA AND 80H AS A BUFFER FOR
463:               PROGRAM USE                                          */
464:           RFCBADDR        ADDR INITIAL(5CH),
465:           RFCB BASED RFCBADDR(33) BYTE, /* SOURCE FCB */
466:           WFCB(33)        BYTE /* INTERMEDIATE FILE FCB */
467:                           INITIAL (0,'        ','PIN',0,0,0,0),
468:           SFCB(33)        BYTE /* SYMBOL TABLE FILE FCB */
```

```
469:                        INITIAL (0,'        ','SYM',0,0,0,0),
470:          SBLOC         ADDR INITIAL(80H),
471:          SOURCEBUFF BASED SBLOC(SOURCERECSIZE) BYTE, /* SOURCE BUFFER */
472:          SOURCEPTR      BYTE INITIAL (SOURCERECSIZE), /* BUFFER INDEX */
473:          CUR$SRC$REC$SZ BYTE  INITIAL (SOURCERECSIZE),
474:          DISKOUTBUFF(INTRECSIZE)  BYTE,
475:          SYMOUTBUFF(INTRECSIZE)   BYTE,
476:          BUFFPTR        BYTE INITIAL(255), /* BUFFER INDEX */
477:          SYMBUFFPTR     BYTE INITIAL(255), /* SBTBL BUFFER INDEX */
478:          LINEBUFF(CONBUFFSIZE)    BYTE, /*CONSOLE OUT BUFFER */
479:          LINEPTR        BYTE INITIAL(0), /* BUFFER INDEX */
480:          LINENO         ADDR; /* CURRENT LINE NUMBER */
481:
482:
483: /*****************************************************************/
484: /*****************************************************************/
485: /***                  G L O B A L   P R O C E D U R E S        ***/
486: /*****************************************************************/
487: /*****************************************************************/
488:
489:
490: MON1: PROC(F,A);
491:     DCL     F BYTE,
492:     A ADDR;
493:     GO TO BDOS;
494: END MON1;
495:
496:
497: MON2: PROC (F,A) BYTE;
498:     DCL     F BYTE, A ADDR;
499:     GO TO BDOS;
500: END MON2;
501:
502:
503: MON3:PROC;                              /*USED TO RETURN TO THE SYSTEM*/
504:     GOTO BOOT;
```

```
505: END MON3;
506:
507:
508: MOVE: PROC (SOURCE,DESTIN,L); /*MOVES FM SOURCE TO DESTIN FOR L BYTES */
509:     DCL     (SOURCE,DESTIN) ADDR,                    /* L < 255 BYTES */
510:     (SCHAR BASED SOURCE, DCHAR BASED DESTIN,L) BYTE;
511:     DO WHILE (L:=L - 1) <> 255;
512:        DCHAR=SCHAR;
513:        DESTIN=DESTIN + 1;
514:        SOURCE=SOURCE + 1;
515:     END;
516: END MOVE;
517:
518:
519: FILL: PROC (A,CHAR,N);                    /* MOVE CHAR TO A N TIMES */
520:     DCL A ADDR,(CHAR,N,DEST BASED A) BYTE;
521:     DO WHILE (N := N -1) <> 255;
522:        DEST = CHAR;
523:        A = A + 1;
524:     END;
525: END FILL;
526:
527:
528: READ: PROC;
529:     DCL TOGGLE(3) BYTE;
530:     TOGGLE = 1;
531:     CALL MON1(10,.TOGGLE);
532: END READ;
533:
534:
535: PRINTCHAR: PROC(CHAR);
536:     DCL     CHAR BYTE;
537:     CALL MON1(2,CHAR);
538: END PRINTCHAR;
539:
540:
```

```
PRINT: PROC(A);
    DCL     A ADDR;
    CALL MON1(9,A);
END PRINT;


DISKERR: PROC;
    DO;
       CALL PRINT(.'DE    $');
       CALL MON3;
    END;
END DISKERR;

SETUP$INT$FIL:PROC;
   IF NOINTFILE THEN  /* ONLY MAKE FILE IF THIS TOGGLE OFF */
      RETURN;
   CALL MOVE(.RFCB,.WFCB,9);
   CALL MON1(19,.WFCB);
   IF MON2(22,.WFCB) = 255 THEN
      CALL DISKERR;
         CALL MOVE(.RFCB,.SFCB,9);
         SFCB(32) = 0;
         CALL MON1(19,.SFCB);
         IF MON2(22,.SFCB) = 255 THEN
              CALL DISKERR;
END SETUP$INT$FIL;

WRIT$INT$FILE: PROC;
   IF NOINTFILE THEN
      RETURN;
   CALL MON1(26,.DISKOUTBUFF);
   IF MON2(21,.WFCB) <> 0 THEN
      CALL DISKERR;
   CALL MON1(26,80H); /* RESET DMA ADDR */
END WRIT$INT$FILE;
```

```
577: EMIT: PROC(OBJCODE);
578:    DCL OBJCODE BYTE;
579:    IF (BUFFPTR := BUFFPTR+1) >= INTRECSIZE THEN
580:    /* WRITE TO DISK */
581:       DO;
582:         CALL WRIT$INT$FILE;
583:          BUFFPTR = 0;
584:       END;
585:    DISKOUTBUFF(BUFFPTR) = OBJCODE;
586: END EMIT;
587:
588: GENERATE: PROC(OBJCODE);
589:    DCL OBJCODE BYTE;
590:       CODESIZE = CODESIZE+1;
591:          CALL EMIT(OBJCODE);
592: END GENERATE;
593:
594: GEN$ADDR: PROC(A,B);
595:      DCL A BYTE, B ADDR;
596:      CALL GENERATE(A);
597:      CALL GENERATE(LOW(B));
598:      CALL GENERATE(HIGH(B));
599: END GEN$ADDR;
600:
601: WRIT$SYM$FILE: PROC;
602:    IF NOINTFILE THEN
603:       RETURN;
604:    CALL MON1(26,.SYMOUTBUFF);
605:    IF MON2(21,.SFCB) <> 0 THEN
606:       CALL DISKERR;
607:    CALL MON1(26,80H); /* RESET DMA ADDR */
608: END WRIT$SYM$FILE;
609:
610: EMIT$SYM: PROC(OBJCODE);
611:    DCL OBJCODE BYTE;
612:    IF (SYMBUFFPTR := SYMBUFFPTR+1) >= INTRECSIZE THEN
```

```
613:     /* WRITE TO DISK */
614:        DO;
615:          CALL WRIT$SYM$FILE;
616:           SYMBUFFPTR = 0;
617:        END;
618:     SYMOUTBUFF(SYMBUFFPTR) = OBJCODE;
619: END EMIT$SYM;
620:
621: GENSYMTBL: PROC(OBJCODE);
622:     DCL OBJCODE BYTE;
623:           CALL EMIT$SYM(OBJCODE);
624: END GENSYMTBL;
625:
626: MOVE$SBTBL: PROC;
627: DCL SYMPTR ADDRESS;
628: DCL VALUE BASED SYMPTR BYTE;
629: DO SYMPTR = .MEMORY TO (LAST$SBTBL$ID - 1);
630:     CALL GENSYMTBL(VALUE);
631: END;
632: CALL GENSYMTBL(0);
633: CALL GENSYMTBL(0);
634: CALL GENSYMTBL(0);
635: CALL GENSYMTBL(0);
636: CALL GENSYMTBL(EOFFILLER);
637: CALL GENSYMTBL(EOFFILLER);
638: CALL WRIT$SYM$FILE;
639: END MOVE$SBTBL;
640:
641:
642: CLOSE$INT$FIL: PROC;
643:   /* CLOSES A FILE */
644:   IF MON2(16,.WFCB) = 255 THEN
645:      CALL DISKERR;
646:          IF MON2(16,.SFCB) = 255 THEN
647:              CALL DISKERR;
648: END CLOSE$INT$FIL;
```

```
649:
650:
651: OPEN$SRC$FILE: PROC;
652:     CALL MOVE(.'PAS',RFCBADDR+9,3);
653:     RFCB(32),RFCB(12) = 0;
654:     IF MON2(15,RFCBADDR) = 255 THEN
655:        DO;
656:           CALL PRINT(.'NO SOURCE FILE $');
657:           CALL MON3;
658:        END;
659: END OPEN$SRC$FILE;
660:
661:
662: RWND$SRC$FILE:PROC;      /* CP/M DOES NOT REQUIRE ANY ACTION */
663:     RETURN;                    /* PRIOR TO REOPENING */
664: END RWND$SRC$FILE;
665:
666:
667: READ$SRC$FILE:PROC BYTE;
668:     DCL      DCNT BYTE;
669:     IF (DCNT:=MON2(20,RFCBADDR)) > FILEEOF THEN
670:        CALL DISKERR;
671:     RETURN DCNT;
672: END READ$SRC$FILE;
673:
674:
675: CRLF: PROC;
676:     CALL PRINTCHAR(CR);
677:     CALL PRINTCHAR(LF);
678: END CRLF;
679:
680:
681: PRINTDEC: PROC(VALUE);
682:     DCL      VALUE ADDR, I BYTE, COUNT BYTE;
683:     DCL      DECI(4) ADDR INITIAL(1000,100,10,1);
684:     DCL      FLAG BYTE;
```

```
685:      FLAG = FALSE;
686:      DO I = 0 TO 3;
687:      COUNT = 30H;
688:         DO WHILE VALUE >= DECI(I);
689:            VALUE = VALUE - DECI(I);
690:            FLAG= TRUE;
691:            COUNT = COUNT +`1;
692:         END;
693:         IF FLAG OR (I>= 3) THEN
694:            CALL PRINTCHAR(COUNT);
695:         ELSE
696:            CALL PRINTCHAR(' ');
697:      END;
698:      RETURN;
699: END PRINTDEC;
700:
701:
702: PRINT$PROD:PROC;
703:      CALL PRINT(.' PROD = $');
704:      CALL PRINT$DEC(PRODUCTION);
705:      CALL CRLF;
706: END PRINT$PROD;
707:
708:
709: PRINT$TOKEN:PROC;
710:      CALL PRINT(.' TOKEN = $');
711:      CALL PRINT$DEC(TOKEN);
712:      CALL CRLF;
713: END PRINT$TOKEN;
714:
715:
716: CLEAR$LN$BUFF:PROC;
717:      CALL FILL(.LINEBUFF,' ',CONBUFFSIZE);
718: END CLEAR$LN$BUFF;
719:
720:
```

```
721: LISTLINE: PROC(LENGTH);
722:     DCL     (LENGTH,I) BYTE;
723:     CALL PRINT$DEC(LINENO);
724:     CALL PRINT$CHAR(' ');
725:     DO I = 0 TO LENGTH;
726:        CALL PRINTCHAR(LINEBUFF(I));
727:     END;
728:     CALL CRLF;
729: END LISTLINE;
730:
731:
732: /*****************************************************************/
733: /*****************************************************************/
734: /***                S C A N N E R     P R O C E D U R E S      ***/
735: /*****************************************************************/
736: /*****************************************************************/
737:
738:
739: GETCHAR: PROC BYTE;
740:     DCL ADDEOF DATA('EOP', EOLCHAR,LF); /* ADD TO END IF LEFT OFF */
741:
742:
743:     NXT$SRC$CHAR: PROC BYTE;
744:         RETURN SOURCEBUFF(SOURCEPTR);
745:     END NXT$SRC$CHAR;
746:
747:
748:     CHECKFILE: PROC BYTE;
749:         DO FOREVER;
750:            IF (SOURCEPTR:=SOURCEPTR+1)>=CUR$SRC$REC$SZ THEN
751:               DO;
752:                  SOURCEPTR=0;
753:                  IF READ$SRC$FILE=FILEEOF THEN
754:                     RETURN TRUE;
755:               END;
756:            IF (NEXTCHAR:=NXT$SRC$CHAR)<>LF THEN
```

```
757:                RETURN FALSE;
758:            END;
759:        END CHECKFILE;
760:
761:
762:        IF CHECKFILE OR (NEXTCHAR = EOFFILLER) THEN
763:           DO;                                  /* EOF REACHED */
764:              CALL MOVE(.ADDEOF,SBLOC,5);
765:              SOURCEPTR = 0;
766:              NEXTCHAR=NXT$SRC$CHAR;
767:           END;
768:        LINEBUFF(LINEPTR:=LINEPTR + 1)=NEXTCHAR;          /*OUTPUT LINE*/
769:        IF NEXTCHAR  = EOLCHAR  THEN
770:           DO;
771:              LINENO = LINENO + 1;
772:              IF LISTSOURCE  THEN
773:                 CALL LISTLINE(LINEPTR-1);
774:              LINEPTR = 0;
775:              CALL CLEARLNBUFF;
776:           END;
777:        IF NEXTCHAR = TAB THEN
778:           NEXTCHAR = ' ';
779:        RETURN NEXTCHAR;
780: END GETCHAR;
781:
782:
783: GETNOBLANK: PROC;
784:        DO WHILE((GETCHAR = ' ') OR (NEXTCHAR = EOFFILLER));
785:        END;
786: END GETNOBLANK;
787:
788:
789:
790: TITLE:PROC;                                  /* COMPILER VERSION */
791:        CALL CRLF;
792:        CALL PRINT(.'NPS-PASCAL VERS 0.0$');
```

```
793:      CALL CRLF;
794: END TITLE;
795:
796:
797: PRINT$ERROR:PROC;
798:      CALL PRINTDEC(ERRORCOUNT);
799:      CALL PRINTCHAR(' ');
800:      CALL PRINT(.'ERROR(S) DETECTED$');
801:      CALL CRLF;
802: END PRINT$ERROR;
803:
804:
805: ERROR: PROC(ERRCODE);
806:      DCL ERRCODE ADDR,
807:      I        BYTE;
808:      ERRORCOUNT=ERRORCOUNT+1;
809:      CALL PRINT(.'***$');
810:      CALL PRINT$DEC(LINENO);
811:      CALL PRINT(.'  ERROR $');
812:      CALL PRINTCHAR(' ');
813:      CALL PRINTCHAR(HIGH(ERRCODE));
814:      CALL PRINTCHAR(LOW(ERRCODE));
815:      CALL PRINT(.' NEAR $');
816:      CALL PRINTCHAR(' ');
817:      DO I = 1 TO ACCUM;
818:         CALL PRINTCHAR(ACCUM(I));
819:      END;
820:      CALL CRLF;
821:      CALL PRINT(.'AT ERROR $');
822:      CALL PRINTCHAR(' ');
823:      CALL PRINT$TOKEN;
824:      CALL PRINT(.'AT ERROR $');
825:      CALL PRINTCHAR(' ');
826:      CALL PRINT$PROD;
827:      IF TOKEN=EOFC THEN
828:         DO;
```

```
829:            CALL PRINT$ERROR;
830:            CALL MON3;
831:         END;
832: END ERROR;
833:
834:
835: INIT$SCANNER: PROC;
836:     DCL COUNT BYTE;
837:     CALL OPEN$SRC$FILE;
838:     LINENO,LINEPTR = 0;
839:     CALL CLEAR$LN$BUFF;
840:     SOURCEPTR = 128;
841:     CALL GETNOBLANK;
842:     DO WHILE NEXTCHAR = '$';
843:        CALL GET$NO$BLANK;
844:        IF (COUNT := (NEXTCHAR AND 5FH) - 'A') <= 4 THEN
845:           DO CASE COUNT;
846:              LISTSOURCE = TRUE;
847:              LISTPROD = TRUE;
848:              NOINTFILE = TRUE;
849:              LISTTOKEN = TRUE;
850:              DEBUGLN = TRUE;
851:           END;                              /* OF CASE */
852:        CALL GETNOBLANK;
853:     END;
854: END INIT$SCANNER;
855:
856:
857: /*****************************************************************/
858: /*****************************************************************/
859: /***                    S C A N N E R                          ***/
860: /*****************************************************************/
861: /*****************************************************************/
862:
863:
864: SCANNER: PROC;
```

```
865:      DCL FLAG BYTE;
866:
867:
868:      PUTINACCUM: PROC;
869:          IF NOT CONT THEN
870:             DO;
871:                ACCUM(ACCUM := ACCUM + 1) = NEXTCHAR;
872:                HASHCODE = (HASHCODE+NEXTCHAR) AND HASHMASK;
873:                IF ACCUM = 31 THEN CONT = TRUE;
874:             END;
875:      END PUTINACCUM;
876:
877:
878:      PUTANDGET: PROC;
879:          CALL PUTINACCUM;
880:          CALL GETNOBLANK;
881:      END PUTANDGET;
882:
883:
884:      PUTANDCHAR: PROC;
885:          CALL PUTINACCUM;
886:          NEXTCHAR = GETCHAR;
887:      END PUTANDCHAR;
888:
889:
890:      NUMERIC: PROC BYTE;
891:          RETURN(NEXTCHAR - '0') <= 9;
892:      END NUMERIC;
893:
894:
895:      LOWERCASE: PROC BYTE;
896:          RETURN (NEXTCHAR >= 61H) AND (NEXTCHAR <= 7AH);
897:      END LOWER$CASE;
898:
899:
900:      DECIMALPT:PROC BYTE;
```

```
901:          RETURN NEXTCHAR='.';
902:      END DECIMALPT;
903:
904:
905:      CONV$TO$UPPER:PROC;
906:          IF LOWERCASE AND LOWER$TO$UPPER THEN
907:             NEXTCHAR=NEXTCHAR AND 5FH;
908:      END CONV$TO$UPPER;
909:
910:
911:      LETTER: PROC BYTE;
912:          CALL CONV$TO$UPPER;
913:          RETURN ((NEXTCHAR - 'A') <= 25) OR LOWERCASE;
914:      END LETTER;
915:
916:
917:      ALPHANUM: PROC BYTE;
918:          RETURN NUMERIC OR LETTER ;
919:      END ALPHANUM;
920:
921:
922:      SPOOLNUMRIC: PROC;
923:          DO WHILE NUMERIC;
924:             CALL PUTANDCHAR;
925:          END;
926:      END SPOOLNUMRIC;
927:
928:
929:      SET$NEXT$CALL: PROC;
930:          IF NEXTCHAR = ' ' THEN
931:             CALL GETNOBLANK;
932:          CONT = FALSE;
933:      END SET$NEXT$CALL;
934:
935:
936:      LOOKUP: PROC BYTE;
```

```
937:
938:         DCL MAXRWLNG LIT  '9';
939:
940: DECLARE VOCAB DATA(0,'.','<','(','+',5BH,'&','*',')',';','-','/',','
941:      ,'>',':','=',5DH,'.','<=','DO','IF','IN','OF','OR','TO','EOP'
942:      ,'AND','DIV','END','FOR','MOD','NIL','NOT','SET','VAR','CASE'
943:      ,'ELSE','FILE','GOTO','THEN','TYPE','WITH','ARRAY','BEGIN','CONST'
944:      ,'LABEL','UNTIL','WHILE','DOWNTO','PACKED','RECORD','REPEAT'
945:      ,'PROGRAM','FUNCTION','PROCEDURE');
946:
947:      DCL VLOC DATA(0,1,17,33,63,91,121,145,152,160,169);
948:
949:      DCL VNUM DATA(0,1,17,25,35,42,48,53,56,57);
950:
951:      DCL COUNT DATA(0,15,7,9,6,5,3,0,0,0);
952:
953:       DCL PTR ADDR, (FIELD BASED PTR) (9) BYTE;
954:       DCL I BYTE;
955:
956:            COMPARE: PROC BYTE;
957:                DCL I BYTE;
958:                I = 0;
959:                DO WHILE (FIELD(I) = ACCUM(I := I + 1)) AND I <= ACCUM;
960:                END;
961:                RETURN I > ACCUM;
962:            END COMPARE;
963:
964:
965:            IF ACCUM > MAXRWLNG THEN
966:               RETURN FALSE;
967:            PTR=VLOC(ACCUM)+.VOCAB;
968:            DO I=VNUM(ACCUM) TO (VNUM(ACCUM)+COUNT(ACCUM));
969:               IF COMPARE THEN
970:                  DO;
971:                     TOKEN=I;
972:                     IF I = 54 THEN
```

```
973:
974:       /*****************************************/
975:       /* THE FOLLOWING CODE SETS UP STORAGE & */
976:       /* POINTERS FOR RECORD ENTRIES IN THE   */
977:       /* SYMBOL TABLE.                        */
978:       /*****************************************/
979:
980:            DO;
981:               REC$NST=REC$NST+1;
982:               APTRADDR,REC$PAR$ADR(REC$NST)=SBTBL;
983:               ADDRPTR=0000H;
984:               APTRADDR=APTRADDR+2;
985:               ADDRPTR=PRV$SBT$ENTRY;
986:               PRV$SBT$ENTRY=SBTBL;
987:               APTRADDR=APTRADDR+2;
988:               BYTEPTR=1FH;
989:               SBTBL=SBTBL+9;
990:      /* RECORD INITIAZATIONS */
991:               VARIANT$PART(REC$NST),TAG$FD(REC$NST)=FALSE;
992:               FXD$OFST$BSE(REC$NST)=0000H;
993:               VAR$OFST$BSE(REC$NST)=0000H;
994:               CUR$OFST(REC$NST)=0000H;
995:               VAR$CAS$VAL(REC$NST)=0000H;
996:               RECORD$PTR=-1;
997:             END;
998:                   RETURN TRUE;
999:                END;
1000:             PTR=PTR+ACCUM;
1001:          END;
1002:          RETURN FALSE;
1003:       END LOOKUP;
1004:
1005:
1006:
1007: /*****************************************************************************/
1008: /*****************************************************************************/
```

```
/***                    SCANNER - MAIN   CODE                          ***/
/*****************************************************************/
/*****************************************************************/

.
    DO FOREVER;
      ACCUM, HASHCODE, TOKEN = 0;
      DO WHILE NEXTCHAR=EOLCHAR;
        CALL GETNOBLANK;
      END;
      IF (NEXTCHAR = STRINGDELIM) OR CONT THEN
      DO;                          /* FOUND STRING */
        TOKEN = STRINGC;
        CONT = FALSE;
        DO FOREVER;
          DO WHILE GETCHAR <> STRINGDELIM;
            CALL PUTINACCUM;
          END;
          CALL GETNOBLANK;
          IF NEXTCHAR <> STRINGDELIM THEN
            RETURN;
          CALL PUT$IN$ACCUM;
        END;                              /* OF DO FOREVER */
      END;                              /* OF RECOGNIZING A STRING */
      ELSE IF NUMERIC THEN
      DO;                           /* HAVE DIGIT */
        TOKEN = NUMBERC;
        TYPENUM = INTEGER$TYPE;
        DO WHILE NEXTCHAR='0'; /*ELIM LEADING ZEROS*/
           NEXTCHAR=GETCHAR;
        END;
        CALL SPOOLNUMRIC;
        IF DECIMALPT THEN
           DO;
             TEMPCHAR1 = NEXTCHAR;
             NEXTCHAR = GETCHAR;
```

```
1045:                IF DECIMALPT THEN
1046:                DO;
1047:                     NEXTCHAR = ' ';
1048:                     SOURCEPTR =SOURCEPTR - 2;
1049:                END;
1050:                ELSE
1051:                DO;
1052:                     NEXTCHAR = TEMPCHAR1;
1053:                     SOURCEPTR = SOURCEPTR - 1;
1054:                     CALL PUTANDCHAR;
1055:                     TYPENUM = REAL$TYPE;
1056:                     CALL SPOOLNUMRIC;
1057:                END;
1058:             END;
1059:          /* THIS TAKES CARE OF EXPON. FORM */
1060:          IF NEXTCHAR = 'E' THEN
1061:             DO;
1062:                TYPENUM = UNSIGN$EXPON;
1063:                CALL PUTANDCHAR;
1064:                IF NEXTCHAR = '-' OR NEXTCHAR = '+' THEN
1065:                   DO;
1066:                      CALL PUTANDCHAR;
1067:                      TYPENUM = SIGNED$EXPON;
1068:                   END;
1069:                CALL SPOOLNUMRIC;
1070:             END;
1071:          IF ACCUM = 0 THEN
1072:            HASHCODE,ACCUM(ACCUM:=1) = '0';
1073:          CALL SET$NEXT$CALL;
1074:          RETURN;
1075:       END;                    /* OF RECOGNIZING NUMERIC CONSTANT */
1076:       ELSE IF LETTER THEN
1077:      DO;                      /* HAVE A LETTER */
1078:         DO WHILE ALPHANUM;
1079:            CALL PUTANDCHAR;
1080:         END;
```

```
1081:          IF NOT LOOKUP THEN
1082:             DO;
1083:                TOKEN = IDENTC;
1084:                CALL SET$NEXT$CALL;
1085:                RETURN;
1086:             END;
1087:          ELSE             /* IS A RW BUT IF COMMENT SKIP */
1088:             DO;
1089:                CALL SET$NEXT$CALL;
1090:                RETURN;
1091:             END;
1092:       END;           /* OF RECOGNIZING RW OR IDENT */
1093:       ELSE DO;                          /* SPECIAL CHARACTER */
1094:        IF NEXTCHAR = COMMENT THEN
1095:           DO;
1096:              NEXTCHAR = GETCHAR;
1097:              DO WHILE NEXTCHAR <> UNCOMMENT;
1098:                 NEXTCHAR = GETCHAR;
1099:              END;
1100:              CALL GET$NO$BLANK;
1101:           END;
1102:        ELSE
1103:           DO;
1104:              IF NEXTCHAR = ':' THEN
1105:                 DO;
1106:                    CALL PUTANDCHAR;
1107:                    IF NEXTCHAR = '=' THEN
1108:                       CALL PUTANDGET;
1109:                 END;
1110:              ELSE
1111:                 IF NEXTCHAR = '.' THEN
1112:                    DO;
1113:                       CALL PUTANDCHAR;
1114:                       IF NEXTCHAR = '.' THEN
1115:                          CALL PUTANDGET;
1116:
```

```
1117:                     ELSE
1118:                       IF NUMERIC THEN
1119:                        DO;
1120:                           TOKEN = NUMBERC;
1121:                            TYPENUM = REAL$TYPE;
1122:                            CALL SPOOLNUMRIC ;
1123:                    /* CHECK FOR EXPONENT */
1124:                       IF NEXTCHAR = 'E' THEN
1125:                          DO;
1126:                             TYPENUM=UNSIGN$EXPON;
1127:                             CALL PUTANDCHAR;
1128:                             IF NEXTCHAR = '-' OR NEXTCHAR = '+' THEN
1129:                               DO;
1130:                                  TYPENUM=SIGNED$EXPON;
1131:                                  CALL PUTANDCHAR;
1132:                               END;
1133:                             CALL SPOOLNUMRIC;
1134:                           END;
1135:                                CALL SET$NEXT$CALL ;
1136:                                RETURN;
1137:                             END;
1138:                       END;
1139:                    ELSE
1140:                       CALL PUTANDGET;
1141:
1142:                IF NOT LOOKUP THEN
1143:                   CALL ERROR('NC');
1144:                CALL SET$NEXT$CALL;
1145:                RETURN;
1146:             END;
1147:          END;                    /* OF RECOGNIZING SPECIAL CHAR */
1148:       END;                          /* OF DO FOREVER */
1149:  END SCANNER;                      /* END OF SCANNER */
1150:
1151:
1152: /*****************************************************************************/
```

```
1153: /*****************************************************************/
1154: /***                  PROCEDURES FOR SYNTHESIZER                 ***/
1155: /*****************************************************************/
1156: /*****************************************************************/
1157:
1158:
1159: INIT$SYMTBL: PROC;
1160:
1161: DCL     SYMBASE             ADDR;
1162:
1163:         DO;
1164:           CALL FILL(.HASHTABLE,0,SHL(HASHTBLSIZE,1));
1165:           SYMBASE=.BUILT$IN$TBL;
1166:           SBTBL=.MEMORY;
1167:           HASHTABLE(14)=SYMBASE;
1168:           HASHTABLE(36)=SYMBASE+14;
1169:           HASHTABLE(30)=SYMBASE+25;
1170:           HASHTABLE(0)=SYMBASE+36;
1171:           HASHTABLE(69)=SYMBASE+50;
1172:           HASHTABLE(16)=SYMBASE+61;
1173:           HASHTABLE(113)=SYMBASE+73;
1174:           HASHTABLE(86)=SYMBASE+86;
1175:           HASHTABLE(118)=SYMBASE+100;
1176:           HASHTABLE(57)=SYMBASE+142;
1177:           HASHTABLE(109)=SYMBASE+159;
1178:           HASHTABLE(26)=SYMBASE+173;
1179:           HASHTABLE(74)=SYMBASE+186;
1180:           HASHTABLE(87)=SYMBASE+201;
1181:           HASHTABLE(90)=SYMBASE+230;
1182:           HASHTABLE(12)=SYMBASE+244;
1183:           HASHTABLE(8)=SYMBASE+260;
1184:           HASHTABLE(101)=SYMBASE+276;
1185:           HASHTABLE(93)=SYMBASE+290;
1186:           HASHTABLE(46)=SYMBASE+304;
1187:           HASHTABLE(43)=SYMBASE+319;
1188:           HASHTABLE(121)=SYMBASE+334;
```

```
1189:           HASHTABLE(96)=SYMBASE+347;
1190:           HASHTABLE(3)=SYMBASE+360;
1191:           HASHTABLE(34)=SYMBASE+375;
1192:           HASHTABLE(29)=SYMBASE+392;
1193:           HASHTABLE(106)=SYMBASE+406;
1194:           HASHTABLE(23)=SYMBASE+418;
1195:           HASHTABLE(64)=SYMBASE+434;
1196:           HASHTABLE(107)=SYMBASE+449;
1197:           HASHTABLE(28)=SYMBASE+465;
1198:           HASHTABLE(54)=SYMBASE+478;
1199:           HASHTABLE(11)=SYMBASE+493;
1200:           HASHTABLE(37)=SYMBASE+507;
1201:           HASHTABLE(40)=SYMBASE+523;
1202:           HASHTABLE(21)=SYMBASE+538;
1203:           HASHTABLE(99)=SYMBASE+552;
1204:           HASHTABLE(62)=SYMBASE+567;
1205:           PRV$SBT$ENTRY = SYMBASE+567;
1206:
1207:       END;
1208: END INIT$SYMTBL;
1209:
1210:
1211: /*****************************************************************************/
1212: /*****************************************************************************/
1213: /***                              P A R S E R                              ***/
1214: /*****************************************************************************/
1215: /*****************************************************************************/
1216:
1217:
1218: DO;                                        /* SCOPE FOR PARSER */
1219: DCL     STATE STATESIZE,
1220:         VAR(PSTACKSIZE) BYTE,
1221:         HASH(PSTACKSIZE) BYTE,
1222:         STATESTACK(PSTACKSIZE) STATESIZE,
1223:         PARMNUM(PSTACKSIZE) BYTE,     /* MAINTAINS NUMBER OF PARAMETERS
1224:                                          ASSOCIATED WITH A SUBROUTINE */
```

```
1225:          LABELSTACK(PSTACKSIZE) ADDR, /* TRACKS STATEMENT LABELS */
1226:          PARMNUMLOC(PSTACKSIZE) ADDR, /* MAINTAINS THE LOCATION IN SYMBOL
1227:                                          TBL WHERE PARAMETER INFO STORED */
1228:          BASE$LOC(PSTACKSIZE) ADDR,  /* STORES THE SYMBOL TABLE ADDRESS
1229:                                          OF THE PERTINATE ENTRY */
1230:          FORM$FIELD(PSTACKSIZE) BYTE, /* STORES THE FORM FIELD OF
1231:                                          SCANNED IDENTIFIERS */
1232:          TYPE$STACK(PSTACKSIZE)BYTE,/* HOLDS A VARIABLE'S TYPE */
1233:          EXPRESS$STK(PSTACKSIZE)BYTE, /* CONTAINS THE TYPES OF THE
1234:                                          EXPRESSION COMPONENTS */
1235:          PRT$ADDR(PSTACKSIZE) ADDR,   /* STORES AN IDENTIFIER'S PRT
1236:                                          LOCATION */
1237:          VARC(VARCSIZE) BYTE,
1238:          (VARINDEX, PARAMNUM)BYTE,
1239:          (SP,MP,MPP1,NOLOOK) BYTE;
1240:
1241:      /* NMUMONICS FOR PASCAL-SM MACHINE */
1242:
1243: DCL    NOP   LIT   '0',ENDP  LIT   '1',LBL   LIT   '2',LDIB  LIT   '3',
1244:        LDII  LIT   '4',PRO   LIT   '5',RTN   LIT   '6',SAVP  LIT   '7',
1245:        UNSP  LIT   '8',CNVB  LIT   '9',CNVI  LIT  '10',ALL   LIT  '11',
1246:        LITA  LIT  '12',ADDB  LIT  '13',ADDI  LIT  '14',SUBB  LIT  '15',
1247:        SUBI  LIT  '16',MULB  LIT  '17',MULI  LIT  '18',DIVB  LIT  '19',
1248:        DIVI  LIT  '20',MODX  LIT  '21',EQLI  LIT  '22',NEQI  LIT  '23',
1249:        LEQI  LIT  '24',GEQI  LIT  '25',LSSI  LIT  '26',GRTI  LIT  '27',
1250:        XIN   LIT  '28',EQLB  LIT  '29',NEQB  LIT  '30',LEQB  LIT  '31',
1251:        GEQB  LIT  '32',LSSB  LIT  '33',GRTB  LIT  '34',EQLS  LIT  '35',
1252:        NEQS  LIT  '36',LEQS  LIT  '37',GEQS  LIT  '38',LSSS  LIT  '39',
1253:        GRTS  LIT  '40',EQSET LIT  '41',NEQST LIT  '42',INCL1 LIT  '43',
1254:        INCL2 LIT  '44',NEGB  LIT  '45',
1255:        NEGI  LIT  '46',COMB  LIT  '47',COMI  LIT  '48',NOTX  LIT  '49',
1256:        ANDX  LIT  '50',BOR   LIT  '51',STOB  LIT  '52',STOI  LIT  '53',
1257:        STO   LIT  '54',STDB  LIT  '55',STDI  LIT  '56',STD   LIT  '57',
1258:        UNION LIT  '58',STDIF LIT  '59',ISEC  LIT  '60',CNAI  LIT  '61',
1259:        BRL   LIT  '62',BLC   LIT  '63',CN2I  LIT  '64',MKSET LIT  '65',
1260:        XCHG  LIT  '66',PARM  LIT  '67',PARMV LIT  '68',PARMX LIT  '69',
```

```
1261:       INC   LIT  '70',DEC   LIT  '71',DEL   LIT  '72',WRT   LIT  '73',
1262:       SUB   LIT  '74',LDSI  LIT  '75',KASE  LIT  '76',LOD   LIT  '77',
1263:       LODB  LIT  '78',LODI  LIT  '79',RDVB  LIT  '80',RDVI  LIT  '81',
1264:       RDVS  LIT  '82',WRTB  LIT  '83',WRTI  LIT  '84',WRTS  LIT  '85',
1265:       DUMP  LIT  '86',ABS   LIT  '87',SQR   LIT  '88',SIN   LIT  '89',
1266:       COS   LIT  '90',ARCTN LIT  '91',EXP   LIT  '92',LN    LIT  '93',
1267:       SQRT  LIT  '94',ODD   LIT  '95',EOLN  LIT  '96',EXF   LIT  '97',
1268:       TRUNC LIT  '98',ROUND LIT  '99',ORD   LIT '100',CHR   LIT '101',
1269:       SUCC  LIT '102',PRED  LIT '103',SEEK  LIT '104',PUT   LIT '105',
1270:       GET   LIT '106',RESET LIT '107',REWRT LIT '108',PAGE  LIT '109',
1271:       NEW   LIT '110',DISPZ LIT '111',FWD   LIT '112',XTRNL LIT '113',
1272:       RDV   LIT '114';
1273:
1274: INITIALIZE$SYNTHESIZE: PROC;
1275:    CODESIZE = 0;
1276:    SBTBLTOP=MAX-2;
1277:    VECPTR=0;     CONST$PTR=0;
1278:    CONST$INDX=0;
1279:    CONST$PN$PTR=0;
1280:    SUBR$PTR=0;
1281:    ARY$DM$ADR$PTR=-1;
1282:    ARRY$PTR=-1;
1283:    VARIANT$PART=FALSE;
1284:    ARRY$QTY=0;
1285:    ALLOC$ADDR=0;
1286: END INITIALIZE$SYNTHESIZE;
1287:
1288:
1289: /*****************************************************************/
1290: /*****************************************************************/
1291: /***                  CODE GENERATING PROCEDURES                ***/
1292: /*****************************************************************/
1293: /*****************************************************************/
1294:
1295:
1296: SYNTHESIZE: PROC;                    /* SYNTHESIZE LOCAL DECLARATIONS */
```

```
1297:
1298:
1299:    /*****************************************************
1300:     * SET$ADDRESS$PTR - THIS PROCEDURE SETS A           *
1301:     *  POINTER TO A SPECIFIC LOCATION IN THE            *
1302:     *  SYMBOL TABLE.                                    *
1303:     *****************************************************/
1304:
1305: SETADDRPTR: PROC(OFFSET);
1306:      DCL OFFSET BYTE;
1307:      APTRADDR = BASE + OFFSET;
1308: END SETADDRPTR;
1309:
1310:
1311:    /*****************************************************
1312:     * SET$PAST$PRINTNAME - THIS PROCEDURE SETS          *
1313:     *  APTRADDR TO A LOCATION IN A SYMBOL TABLE         *
1314:     *  ENTRY THAT IS PAST THE ENTRY'S PRINTNAME         *
1315:     *  (WHICH IS OF VARIABLE LENGTH).                   *
1316:     *****************************************************/
1317:
1318: SET$PAST$PN: PROC(OFFSET);
1319:      DCL OFFSET BYTE;
1320:      CALL SETADDRPTR(6);
1321:      CALL SETADDRPTR(BYTEPTR + OFFSET);
1322: END SET$PAST$PN;
1323:
1324:
1325:    /*****************************************************
1326:     * CALC$VARC - THIS PROCEDURE DETERMINES THE         *
1327:     *  LOCATION OF AN IDENTIFIER PRINTNAME.             *
1328:     *****************************************************/
1329:
1330: CALC$VARC: PROC(A) ADDR;
1331:      DCL A BYTE;
1332:      RETURN VAR(A) + .VARC;
```

```
1333: END CALC$VARC;
1334:
1335:
1336:   /*************************************************
1337:    * SET$LOOKUP - THIS PROCEDURE IS UTILIZED TO    *
1338:    *  FIND THE HASH VALUE OF AN IDENTIFIER.        *
1339:    *************************************************/
1340:
1341: SETLOOKUP: PROC(A);
1342:     DCL A BYTE;
1343:     PRINTNAME = CALC$VARC(A);
1344:     SYMHASH = HASH(A); /* HASHCODE OF PN */
1345: END SETLOOKUP;
1346:
1347:
1348:   /*************************************************/
1349:   /* ENTER$LINKS - THIS PROCEDURE ENTERS IN THE    */
1350:   /*  NEXT FOUR BYTES OF THE SYMBOL TABLE THE      */
1351:   /*  COLLISION FIELD AND THE PREVIOUS SYMBOL      */
1352:   /*  TABLE ENTRY ADDRESS FIELD FOR THE NEXT       */
1353:   /*  SYMBOL TABLE ENTRY. ( BOTH IN ADDRESS VAR )  */
1354:   /*************************************************/
1355:
1356: ENTER$LINKS: PROC;
1357:     BASE, APTRADDR, SBTBLSCOPE = SBTBL;
1358:     SCOPE(SCOPE$NUM) = SBTBL;
1359:     ADDRPTR = HASHTABLE(SYMHASH);
1360:     CALL SETADDRPTR(2);
1361:     ADDRPTR = PRV$SBT$ENTRY;
1362:     PRV$SBT$ENTRY = SBTBL;
1363:     HASHTABLE(SYMHASH) = BASE;
1364: END ENTER$LINKS;
1365:
1366:
1367:   /*************************************************
1368:    * CHECK$PRINT$NAME - THIS PROCEDURE DOES A      *
```

```
1369:     *  CHARACTER TO CHARACTER COMPARISON BETWEEN   *
1370:     *  THE CURRENTLY RECOGNIZED IDENTIFIER AND     *
1371:     *  SYMBOL TABLE ENTRIES OF THE SAME HASH VALUE.*
1372:     ***********************************************/
1373:
1374: CHK$PRT$NAME: PROC(A) BYTE;
1375:     /* A IS OFFSET FROM BASE TO PRINTNAME */
1376:     DCL N BASED PRINTNAME BYTE;
1377:     DCL (LEN,A) BYTE;
1378:     CALL SETADDRPTR(A);
1379:     IF ( LEN := BYTEPTR ) = N THEN
1380:     DO WHILE (BYTEPTR(LEN)=N(LEN));
1381:       IF ( LEN := LEN-1 ) = 0 THEN
1382:         RETURN TRUE;
1383:     END;
1384:     RETURN FALSE;
1385: END CHK$PRT$NAME;
1386:
1387:
1388:   /***********************************************/
1389:   /* LOOKUP$PRINTNAME$IDENTITY - THIS PROCEDURE  */
1390:   /*  IS PASSED THE LOCATION OF AN IDENTIFIER IN */
1391:   /*  THE PRODUCTION RULE, AND ITS TARGET ENTRY  */
1392:   /*  TYPE. IF THE IDENTIFIER IS FOUND WITH THE  */
1393:   /*  CORRECT TYPE THE PROCEDURE RETURN TRUE,    */
1394:   /*  ELSE FALSE IS RETURNED.                    */
1395:   /***********************************************/
1396:
1397: LOOKUP$PN$ID: PROC(A,ID$ENTRY) BYTE;
1398:     DCL (A,ID$ENTRY) BYTE;
1399:     CALL SETLOOKUP(A);
1400:     BASE = HASHTABLE(SYMHASH);
1401:     DO WHILE BASE <> 0;
1402:       CALL SETADDRPTR(4);
1403:       IF (( BYTEPTR AND FORMMASK ) = ID$ENTRY ) THEN
1404:         IF CHK$PRT$NAME(6) THEN
```

```
1405:            IF ((BASE < SCOPE(0)) OR (BASE >= SCOPE(SCOPE$NUM-1))
1406:            OR ((ID$ENTRY = TYPE$ENTRY) AND (BASE < SCOPE(SCOPE$NUM))))
1407:            THEN DO;
1408:              LOOKUP$ADDR=BASE;
1409:              RETURN TRUE;
1410:            END;
1411:          CALL SETADDRPTR(0);
1412:          BASE = ADDRPTR;
1413:        END;
1414:        RETURN FALSE;
1415: END LOOKUP$PN$ID;
1416:
1417:   /*****************************************************/
1418:   /*  LIMITS - THIS PROCEDURE ENSURES THAT THE         */
1419:   /*   SYMBOL TABLE ENTRY ABOUT TO BE ENTERED          */
1420:   /*   WILL NOT EXCEED THE UPPER LIMIT OF THE          */
1421:   /*   AVAILABLE SYMBOL TABLE ADDRESSES.               */
1422:   /*   THE PARAMETER IS THE BYTECOUNT OF THE           */
1423:   /*   ENTRY TO BE ENTERED.                            */
1424:   /*****************************************************/
1425:
1426: LIMITS: PROC(COUNT);
1427:     DCL COUNT BYTE;
1428:     IF SBTBLTOP <= (SBTBL + COUNT) THEN
1429:     DO;
1430:       CALL ERROR('TO');
1431:       CALL MON3;
1432:     END;
1433: END LIMITS;
1434:
1435:
1436:   /*****************************************************/
1437:   /*  ENTER$PRINTNAME$IDENTITY - THIS PROCEDURE        */
1438:   /*   LOADS THE SYMBOL TABLE WITH THE FOLLOWING:      */
1439:   /*    1. COLLISION FIELD                             */
1440:   /*
```

```
1441:    /*     2. PREVIOUS SYMBOL TABLE ENTRY ADDRESS     */
1442:    /*     3. FORM OF ENTRY ( PRESET BYTE 'FORM' )    */
1443:    /*     4. THE LENGTH OF THE PRINTNAME IN ONE BYTE*/
1444:    /*     5. THE PRINTNAME CHARACTERS                */
1445:    /*    PARAMETER: PRINTNAME IS SET PRIOR TO CALL.  */
1446:    /***************************************************/
1447:
1448: ENTER$PN$ID:PROC;
1449:      DCL (I,J,N BASED PRINTNAME) BYTE;
1450:      CALL LIMITS(I:=N+7);
1451:      CALL ENTER$LINKS;
1452:      CALL SETADDRPTR(4);
1453:      BYTEPTR = FORM;
1454:      CALL SETADDRPTR(5);
1455:      BYTEPTR = SYMHASH;
1456:      CALL SETADDRPTR(6);
1457:      BYTEPTR=N;
1458:      CALL MOVE(PRINTNAME+1,SBTBL+7,N);
1459:      LAST$SBTBL$ID = SBTBL;
1460:      SBTBL=SBTBL+I;
1461: END ENTER$PN$ID;
1462:
1463:
1464:    /***************************************************/
1465:    /*  ENTER$VARIABLE$IDENTITY - THIS PROCEDURE       */
1466:    /*    CALLS ENTER$PN$ID TO LOAD THE SYMBOL TABLE   */
1467:    /*    ENTRY CURRENTLY BEING SCANNED. IT ALSO       */
1468:    /*    GENERATES THE ENTRY'S "FORM" BY PERFORMING   */
1469:    /*    A BOOLEAN 'OR' OPERATION ON THE ID$ENTRY     */
1470:    /*    AND THE PARAMETER 'A'.                       */
1471:    /***************************************************/
1472:
1473: ENTER$VAR$ID: PROC(A,B,ID$ENTRY);
1474:      DCL (A,B,ID$ENTRY) BYTE;
1475:      IF LOOKUP$PN$ID(B,ID$ENTRY) THEN
1476:      DO;
```

```
1477:         PRESENT = TRUE;
1478:         RETURN;
1479:       END;
1480:       /* ELSE ENTER VAR NAME */
1481:       PRESENT = FALSE;
1482:       FORM = A OR ID$ENTRY;
1483:       CALL ENTER$PN$ID;
1484:       IF ID$ENTRY = VAR$ENTRY THEN
1485:       DO;
1486:         CALL LIMITS(4);
1487:         VAR$BASE1(VAR$PTR) = SBTBL;
1488:         SBTBL = SBTBL + 4;
1489:       END;
1490: END ENTER$VAR$ID;
1491:
1492:
1493:   /*****************************************************/
1494:   /*  SET$LABEL - THIS PROCEDURE ASSIGNS A LABEL  */
1495:   /*    TO THE CURRENT DECLARED LABEL AND INCREMENT*/
1496:   /*    THE LABELCOUNT ( NEXT TO ASSIGN ).          */
1497:   /*****************************************************/
1498:
1499: SET$LABEL: PROC;
1500:       ADDRPTR=LABLCOUNT;
1501:       LABLCOUNT=LABLCOUNT+1;
1502: END SET$LABEL;
1503:
1504:
1505:   /*****************************************************/
1506:   /*  ENTER$LABEL - THIS PROCEDURE LOADS A LABEL  */
1507:   /*    ENTRY INTO THE SYMBOL TABLE. SYMHASH AND    */
1508:   /*    PRINTNAME MUST BE SET PRIOR TO CALLING      */
1509:   /*****************************************************/
1510:
1511: ENTER$LABEL: PROC;
1512:       CALL LIMITS(2);
```

```
1513:      APTRADDR = SBTBL;
1514:      CALL SET$LABEL;
1515:      SBTBL = SBTBL+2;
1516:  END ENTER$LABEL;
1517:
1518:
1519:    /*************************************************
1520:     *  ALTER$PRT$LOCATIONS - THIS PROCEDURE RE-     *
1521:     *   ALLOCATES PRT LOCATIONS FOR ALL FUNCTIONS   *
1522:     *   AND FORWARD PROCEDURES AND THEIR ASSOCIATED*
1523:     *   FORMAL PARAMETERS.                          *
1524:     *************************************************/
1525:
1526:  ALTER$PRT$LOC: PROC;
1527:      DCL (I,P) BYTE;
1528:      CALL SET$PAST$PN(7);
1529:      P = BYTEPTR;
1530:      PARAMNUMLOC = APTRADDR;
1531:      DO I = 1 TO P;
1532:        CALL SET$PAST$PN(8);
1533:        APTRADDR = ADDRPTR + ((I-1)*3);
1534:        DO CASE (SHR(BYTEPTR,3) AND FORMMASK);
1535:          ALLC$QTY = 1;     /* SCALAR */
1536:          ALLC$QTY = 2;    /* INTEGER */
1537:          ALLC$QTY = 8;    /* REAL */
1538:          ALLC$QTY = 1;    /* CHAR */
1539:          ALLC$QTY = 1;    /* BOOLEAN */
1540:        END;                /* OF CASE */
1541:        APTRADDR = APTRADDR + 1;
1542:        ADDRPTR = ALLOC$ADDR;
1543:        APTRADDR = TEMPADDR1;
1544:        APTRADDR = APTRADDR + 6;
1545:        APTRADDR = APTRADDR + 1 + BYTEPTR;
1546:        ADDRPTR = ALLOC$ADDR;
1547:        ALLOC$ADDR = ALLOC$ADDR + ALLC$QTY;
1548:        TEMPADDR1 = APTRADDR + 4;
```

```
1549:      END;
1550: END ALTER$PRT$LOC;
1551:
1552:
1553:    /*************************************************
1554:     * ENTER$SUBROUTINE - THIS PROCEDURE LOADS A     *
1555:     *  SUBROUTINE ENTRY IN THE SYMBOL TABLE.  THE   *
1556:     *  PARAMETER NUMBER LOCATION IS STORED AND THE  *
1557:     *  SCOPE LEVEL IS INCRIMENTED BY ONE.           *
1558:     *************************************************/
1559:
1560: ENTER$SUBRTN: PROC(A,B,ID$ENTRY);
1561:      DCL (A,B,ID$ENTRY) BYTE;
1562:      CALL ENTER$VAR$ID(0, SP, ID$ENTRY);
1563:      IF NOT PRESENT THEN
1564:      DO;
1565:        CALL LIMITS(4);
1566:        PARAMNUMLOC = SBTBL;
1567:        SBTBL = SBTBL + 3;
1568:        CALL SET$PAST$PN(10);
1569:        ADDRPTR = ALLOC$ADDR;    ALLOC$ADDR = ALLOC$ADDR + 2;
1570:        CALL SET$PAST$PN(14);
1571:        ADDRPTR = LABLCOUNT;
1572:        LABLCOUNT = LABLCOUNT + 2;
1573:        SBTBL = SBTBL + 6;
1574:        IF ID$ENTRY = FUNC$ENTRY THEN
1575:        DO;
1576:          SBTBL = SBTBL + 1;
1577:        END;
1578:      END;
1579:      ELSE DO;   /* FORWARD FUNCTION */
1580:        CALL SET$PAST$PN(14);
1581:        IF ID$ENTRY = FUNC$ENTRY THEN TEMPADDR1 = APTRADDR + 3;
1582:        ELSE TEMPADDR1 = APTRADDR + 2;
1583:        CALL SET$PAST$PN(10);
1584:        ADDRPTR = ALLOC$ADDR;
```

```
1585:        ALLOC$ADDR = ALLOC$ADDR + 2;
1586:        CALL ALTER$PRT$LOC;
1587:      END;
1588:      PARMNUMLOC(MP) = BASE;
1589:      SCOPE(SCOPE$NUM := SCOPE$NUM+1) = SBTBL;
1590: END ENTER$SUBRTN;
1591:
1592:
1593:    /*****************************************************/
1594:    /*  LOOKUP$ONLY - THIS PROCEDURE IS PASSED THE  */
1595:    /*   POSITION OF A IDENTIFIER JUST SCANNED IN   */
1596:    /*   THE CURRENT PRODUCTION ( SP,MP,MPP1 ) AND  */
1597:    /*   RETURNS TRUE IF THE IDENTIFIER IS FOUND IN */
1598:    /*   THE SYMBOL TABLE.                          */
1599:    /*****************************************************/
1600:
1601: LOOKUP$ONLY: PROC(A) BYTE;
1602:      DCL A BYTE;
1603:      CALL SETLOOKUP(A);
1604:      BASE=HASHTABLE(SYMHASH);
1605:      DO WHILE BASE <> 0;
1606:        IF CHK$PRT$NAME(6) THEN
1607:        DO;
1608:          LOOKUP$ADDR=BASE;
1609:          RETURN TRUE;
1610:        END;
1611:        ELSE DO;
1612:          CALL SETADDRPTR(0);
1613:          BASE=ADDRPTR;
1614:          END;
1615:        END;
1616:      RETURN FALSE;
1617: END LOOKUP$ONLY;
1618:
1619:
1620:   /*************************************/
```

```
1621:  /*  THIS PROCEDURE CONVERTS A REAL   */
1622:  /*   NUMBER IN THE PROGRAM TO A BCD  */
1623:  /*   REPRESENTATION.                 */
1624:  /*************************************/
1625:
1626: CONVRTBCD: PROC(A,B); /* A=SP/MP/MPP1, B=POS/NEG */
1627:     DCL (I,J,DFLAG,EFLAG,SFLAG,A,B,N BASED PRINTNAME) BYTE;
1628:     DCL (EXPONLOOP,EXPSIGNLOOP) LABEL;
1629:     CALL SETLOOKUP(A);
1630:      /*  INITIALIZE VARIABLES */
1631:     SFLAG=FALSE; EFLAG=TRUE; DFLAG=TRUE; I=1;
1632:     DO J=0 TO 7; BCDNUM(J)=0; END;
1633:     J=0; EXPON=64; /* E+00 */
1634:     /* REMOVE LEADING ZEROS */
1635:     DO WHILE ((N(I) - '0') = 0);
1636:       I=I+1;
1637:       IF I=(N+1) THEN GOTO EXPONLOOP;
1638:     END;
1639:     /* LOAD BCDNUM WITH SIGNIFICANT DIGITS */
1640:     DO WHILE ((N(I) - '0' ) <= 9 OR N(I) = '.');
1641:       IF N(I) = '.' THEN
1642:       DO; EFLAG=FALSE;
1643:         IF I=N THEN GOTO EXPONLOOP;
1644:         I = I + 1;
1645:       END;
1646:       ELSE
1647:       DO;
1648:         DO WHILE J = 0 AND DFLAG AND (N(I) - '0') = 0;
1649:           EXPON = EXPON-1;
1650:           IF I = N THEN GOTO EXPONLOOP;
1651:           I = I + 1 ;
1652:         END;
1653:         IF J = ( BCDSIZE-1 ) THEN GOTO EXPONLOOP;
1654:         IF DFLAG THEN /* FIRST BCD PAIR */
1655:         DO;
1656:           BCDNUM(J)=ROL((N(I)-'0'),4);
```

```
1657:           DFLAG=FALSF; I= I+1;
1658:           IF EFLAG THEN EXPON=EXPON+1;
1659:         END;
1660:         ELSE
1661:         DO;
1662:           BCDNUM(J)=BCDNUM(J)+(N(I)-'0');
1663:           J = J + 1; I = I + 1;
1664:           DFLAG=TRUE; IF EFLAG THEN EXPON=EXPON+1;
1665:         END;
1666:         IF I=(N+1) THEN GOTO EXPONLOOP;
1667:       END;
1668:     END;
1669:     EXPONLOOP:
1670:     IF N(I) = 'E' THEN EFLAG = FALSE;
1671:     IF I = (N+1) THEN GOTO EXPSIGNLOOP;
1672:     IF EFLAG THEN
1673:     DO;
1674:       DO WHILE N(I) <> '.';
1675:         EXPON = EXPON + 1;
1676:         I = I + 1;
1677:       END;
1678:       I = I + 1;
1679:     END;
1680:     DO WHILE I < (N+1) AND (N(I)-'0') <= 9 ;
1681:       I = I + 1;
1682:     END;
1683:     IF TYPENUM = REALTYPE THEN GOTO EXPSIGNLOOP;
1684:     /* N(I) = E */ I = I+1;
1685:     IF TYPENUM = SIGNED$EXPON THEN
1686:     DO;
1687:       IF N(I) = 2DH THEN SFLAG = TRUE;
1688:       I = I + 1 ;
1689:     END;
1690:     IF  I = N+1 THEN
1691:     DO;
1692:       CALL ERROR('EE');
```

```
1693:       RETURN;
1694:     END;
1695:     DFLAG = 0;
1696:     DO J = I TO N;
1697:       DFLAG = (DFLAG*10)+(N(J)-'0');
1698:     END;
1699:     IF SFLAG THEN /* EXPONENT CALCULATION */
1700:        EXPON = EXPON-DFLAG;
1701:     ELSE EXPON = EXPON + DFLAG;
1702:     EXPSIGNLOOP:
1703:     BCDNUM(BCDSIZE-1)=ROL(B,7); /* SIGN OF NUMBER */
1704:     IF EXPON > 127 THEN
1705:     DO;
1706:       CALL ERROR('EE');
1707:       RETURN;
1708:     END;
1709:     ELSE BCDNUM(BCDSIZE-1)=BCDNUM(BCDSIZE-1)+EXPON;
1710: END CONVRTBCD;
1711:
1712:
1713:   /*************************************************/
1714:   /*  CONVERTI - THIS PROCEDURE IS PASSED "A", THE*/
1715:   /*    LOCATION OF A CONSTANT IN THE PRODUCTION   */
1716:   /*    AND "B" THE 'SIGN' OF THE INTEGER. THE     */
1717:   /*    FUNCTION GENERATES A SIGNED 16 BIT REPRE-  */
1718:   /*    SENTATION OF THE NUMBER AND RETURNS IT IN  */
1719:   /*    AN ADDRESS VARIABLE.                       */
1720:   /*************************************************/
1721:
1722: CONVERTI: PROC(A,B) ADDRESS;
1723:     DCL (I,A,B,N BASED PRINTNAME) BYTE;
1724:     DCL NUM ADDR;
1725:     CALL SETLOOKUP(A); NUM=0;
1726:     DO I=1 TO N;
1727:       IF (MAXINT/10) >= NUM THEN
1728:       DO;
```

```
1729:          IF (MAXINT/10) = NUM AND (N(I)-'0') > 7 THEN
1730:          DO;
1731:            CALL ERROR('IE');
1732:            RETURN NUM;
1733:          END;
1734:          NUM=(NUM*10)+(N(I)-'0');
1735:        END;
1736:        ELSE DO;
1737:          CALL ERROR('IE');
1738:          RETURN NUM;
1739:        END;
1740:      END;
1741:      IF B = POS THEN RETURN NUM;
1742:      IF NUM = MAXINT THEN
1743:      DO;
1744:        CALL ERROR('IE');
1745:        RETURN NUM;
1746:      END;
1747:      RETURN ( - NUM);
1748: END CONVERTI;
1749:
1750:
1751:    /*****************************************************/
1752:    /*  CONVERT$CONSTANT - THIS PROCEDURE IS CALLED */
1753:    /*   WITH TYPENUM SET BY THE CALLER. THE NUMBER */
1754:    /*   MUST BE POINTED TO BY "SP" IN THE PRODUC-  */
1755:    /*   TION. THE PROCEDURE RETURNS WITH "CONST$   */
1756:    /*   NUM$TYPE" AND "CONST$VALUE" SET WITH THE   */
1757:    /*   NUMBER IN ITS INTERNAL FORM.               */
1758:    /*****************************************************/
1759:
1760: CONVRT$CONST: PROC(A); /* A=POS,NEG */
1761:      DCL A BYTE,INT$ADDR ADDR;
1762:      IF TYPENUM = INTEGER$TYPE THEN
1763:      DO;
1764:        INT$ADDR=CONVERTI(SP,A);
```

```
1765:        CONST$NUM$TYPE(CONST$PTR)=INTEGER$TYPE;
1766:        CONST$PTR=CONST$PTR+1;
1767:        CALL MOVE(.INT$ADDR,.CONST$VALUE(CONST$INDX),2);
1768:        CONST$INDX=CONST$INDX+2;
1769:      END;
1770:      ELSE DO;
1771:        CALL CONVRTBCD(SP,A);
1772:        CONST$NUM$TYPE(CONST$PTR)=REAL$TYPE;
1773:        CONST$PTR=CONST$PTR+1;
1774:        CALL MOVE(.BCDNUM,.CONST$VALUE(CONST$INDX),BCDSIZE);
1775:        CONST$INDX=CONST$INDX+BCDSIZE;
1776:      END;
1777: END CONVRT$CONST;
1778:
1779:
1780:   /*************************************************/
1781:   /*  ENTER$CONSTANT$NUMBER - AFTER THE NEXT ENTRY*/
1782:   /*   HAS HAD ITS LINKS ENTERED INTO THE SYMBOL  */
1783:   /*   TABLE, THIS PROCEDURE ENTERS THE CONSTANT  */
1784:   /*   VALUE INTO THE SYMBOL TABLE AND SET THE    */
1785:   /*   ENTRY'S 'FORM' TO THE APPROPRIATE TYPE.    */
1786:   /*************************************************/
1787:
1788: ENTR$CONS$NUM: PROC;
1789:      CONST$PTR=CONST$PTR-1;
1790:      IF CONST$NUM$TYPE(CONST$PTR)= INTEGERTYPE THEN
1791:      DO;
1792:        CALL SETADDRPTR(4); BYTEPTR=8 OR CONS$ENTRY;
1793:        CALL LIMITS(2); CONST$INDX=CONST$INDX-2;
1794:        CALL MOVE(.CONST$VALUE(CONST$INDX),SBTBL,2);
1795:        SBTBL=SBTBL+2;
1796:      END;
1797:      ELSE DO;
1798:        CALL SETADDRPTR(4); BYTEPTR=10H OR CONS$ENTRY;
1799:        CALL LIMITS(BCDSIZE); CONST$INDX=CONST$INDX-BCDSIZE;
1800:        CALL MOVE(.CONST$VALUE(CONST$INDX),SBTBL,BCDSIZE);
```

```
1801:        SBTBL=SBTRL+BCDSIZE;
1802:      END;
1803: END ENTR$CONS$NUM;
1804:
1805:
1806:   /*************************************************/
1807:   /*  ENTER$STRING - AFTER THE "LINKS" AND "FORM"  */
1808:   /*    ARE ENTERED INTO THE SYMBOL TABLE, THIS    */
1809:   /*    PROCEDURE LOADS ANY IDENTIFIER ALONG WITH  */
1810:   /*    ITS LENGTH. (USED WITH CONSTANT STRINGS    */
1811:   /*    AND CONSTANT IDENTIFIERS )                 */
1812:   /*************************************************/
1813:
1814: ENTER$STRING: PROC(A);
1815:      DCL (A,N BASED PRINTNAME) BYTE;
1816:      CALL SETLOOKUP(A);
1817:      CALL LIMITS(N+1);
1818:      CALL MOVE(PRINTNAME,SBTBL,(N+1));
1819:      SBTBL=SBTRL+(N+1);
1820: END ENTER$STRING;
1821:
1822:
1823:   /*************************************************
1824:    *  ENTER$CONSTANT$ID - THIS PROCEDURE ENTERS    *
1825:    *    THE FORM FIELD OF A CONSTANT ENTRY INTO    *
1826:    *    THE SYMBOL TABLE.                          *
1827:    *************************************************/
1828:
1829: ENTR$CONS$ID: PROC(A,B); /* A=POS/NEG , B=MP/MPP1/SP */
1830:      DCL (A,B,C) BYTE;
1831:      C=ROL(A,6);
1832:      CALL SETADDRPTR(4); BYTEPTR=C OR CONS$ENTRY;
1833:      CALL ENTER$STRING(SP);
1834:      CONST$PN$PTR=CONST$PN$PTR-1;
1835:      CONST$INDX=CONST$INDX-CONST$PN$SIZE(CONST$PN$PTR);
1836: END ENTR$CONS$ID;
```

```
1837:
1838:
1839:   /*************************************************
1840:    *  ENTER$CONSTANT$ENTRY - THIS PROCEDURE        *
1841:    *   DETERMINES WHICH TYPE OF CONSTANT ENTRY IS  *
1842:    *   TO BE ENTERED IN THE SYMBOL TABLE, AND      *
1843:    *   AND CALLS THE CORRESPONDING PROCEDURE TO    *
1844:    *   MAKE THE ENTRY.                             *
1845:    *************************************************/
1846:
1847: ENTR$CONS$NTRY: PROC;
1848:     DCL IXINDEX BYTE;
1849:     VECPTR=VECPTR-1;
1850:     DO CASE EXPRESS$STK(SP);
1851:       /* CASE CONSTANT NUMBER */
1852:       CALL ENTR$CONS$NUM;
1853:       /* CASE IDENTIFIER CONSTANT */
1854:       CALL ENTR$CONS$ID(POS,SP);
1855:       /* CASE SIGNED IDENTIFIER CONSTANT */
1856:       CALL ENTR$CONS$ID(NEG,SP);
1857:       /* CASE CONSTANT STRING */
1858:       DO;
1859:         CALL SETADDRPTR(4); BYTEPTR=18H OR CONS$ENTRY;
1860:         CALL ENTER$STRING(SP);
1861:         CONST$PN$PTR=CONST$PN$PTR-1;
1862:         CONST$INDX=CONST$INDX-CONST$PN$SIZE(CONST$PN$PTR);
1863:       END;
1864:     END; /* OF CASE CONST$TYPE */
1865: END ENTR$CONS$NTRY;
1866:
1867:
1868:   /*************************************************/
1869:   /*  ENTR$CPLX$TYP - THIS PROCEDURE IS            */
1870:   /*   CALLED TO ENTER THE "LINKS" AND "FORM" FOR  */
1871:   /*   THE 'COMPLEX TYPE' SYMBOL TABLE ENTRIES.    */
1872:   /*   NOTE* THAT THIS ENTRY NEVER HAS A PRINT-    */
```

```
1873:    /*    NAME ASSIGNED.                                          */
1874:    /*************************************************************/
1875:
1876: ENTR$CPLX$TYP: PROC(A);
1877:      DCL A BYTE;
1878:      CALL LIMITS(5);
1879:      BASE,APTRADDR=SBTBL;
1880:      ADDRPTR=0000H;
1881:      CALL SETADDRPTR(2);
1882:      ADDRPTR=PRV$SBT$ENTRY;
1883:      PRV$SBT$ENTRY=BASE;
1884:      CALL SETADDRPTR(4);
1885:      BYTEPTR=A;
1886:      SBTBL=SBTBL+5;
1887: END ENTR$CPLX$TYP;
1888:
1889:
1890:    /*************************************************************/
1891:    /*  ENTR$STR$TYP - THIS PROCEDURE IS                         */
1892:    /*    CALLED BY THE 'TYPE' PRODUCTIONS:                      */
1893:    /*      1. SET TYPE                                          */
1894:    /*      2. FILE TYPE                                         */
1895:    /*      3. POINTER TYPE                                      */
1896:    /*    IT CALLS ENTR$CPLX$TYP TO SET UP ITS                   */
1897:    /*    "LINKS" AND "FORM", THEN IT SETS A POINTER             */
1898:    /*    TO THE ASSOCIATED COMPLEX TYPE.                        */
1899:    /*************************************************************/
1900:
1901: ENTR$STR$TYP: PROC(A);
1902:      DCL A BYTE;
1903:      CALL ENTR$CPLX$TYP(A);
1904:      CALL LIMITS(2);
1905:      CALL SETADDRPTR(5);
1906:      ADDRPTR=TYPE$LOCT;
1907:      SBTBL=SBTBL+2;
1908:      TYPE$LOCT=BASE;
```

```
1909: END ENTR$STR$TYP;
1910:
1911:
1912:   /**************************************************
1913:    * ENTER$PARAMETER$TYPE - THIS PROCEDURE          *
1914:    *  UTILIZES 3 BYTE OF CODE FOR EACH SUBROUT-     *
1915:    *  INE PARAMETER THAT WAS RECOGNIZED AND PUTS    *
1916:    *  THE FOLLOWING INFORMATION IN THE SYMBOL       *
1917:    *  TABLE:   1. TYPE OF PARAMETER                 *
1918:    *         2-3. RELATIVE LOCATION OF PARAMETER.   *
1919:    **************************************************/
1920:
1921: ENTR$PRM$TYP: PROC;
1922:     APTRADDR = PARAMNUMLOC + 1;
1923:     ADDRPTR = SBTBL;
1924:     SBTBL = SBTBL + 3*PARAMNUM - 3;
1925:     BASE = LAST$SBTBL$ID;
1926:     DO WHILE PARAMNUM <> 0;
1927:       CALL SETADDRPTR(4);
1928:       TEMPBYTE = BYTEPTR;
1929:       APTRADDR = SBTBL;
1930:       BYTEPTR = TEMPBYTE;
1931:       SBTBL = SBTBL + 1;
1932:       CALL SET$PAST$PN(7);
1933:       TEMPADDR = ADDRPTR;
1934:       APTRADDR = SBTBL;
1935:       ADDRPTR = TEMPADDR;
1936:       SBTBL = SBTBL - 4;
1937:       CALL SETADDRPTR(2);
1938:       BASE = ADDRPTR;
1939:       PARAMNUM = PARAMNUM - 1;
1940:     END;
1941:     APTRADDR = PARAMNUMLOC;
1942:     SBTBL = SBTBL + 3*(BYTEPTR + 1);
1943: END ENTR$PRM$TYP;
1944:
```

```
1945:
1946:    /*************************************************
1947:     * BUILT$IN$PARAMETER - THIS PROCEDURE ENSURES  *
1948:     *  A PROPER MATCH UP BETWEEN THE SUBROUTINE'S  *
1949:     *  FORMAL PARAMETERS AND THE CALLING ACTUAL    *
1950:     *  PARAMETERS.                                 *
1951:     *************************************************/
1952:
1953: BUILT$IN$PARM: PROC;
1954:      APTRADDR = PARMNUMLOC(SP);
1955:      BASE = APTRADDR;
1956:      IF BYTEPTR = 13H THEN
1957:      DO;      /* CHECK FOR INTEGER OR REAL INPUT */
1958:        IF NOT(((SHL((BYTEPTR AND FORMMASK),3) OR VAR$ENTRY)=
1959:        (FORM$FIELD(SP) AND 7FH))
1960:        OR ((ROR((BYTEPTR AND 70H),1) OR VAR$ENTRY)=
1961:        (FORM$FIELD(SP) AND 7FH))) THEN
1962:          CALL ERROR('IP');
1963:        ELSE CALL GEN$ADDR(PARM,PRT$ADDR(SP));
1964:      END;
1965:      ELSE DO;
1966:        IF BYTEPTR = 0F3H THEN
1967:        DO;
1968:          IF SHR(FORM$FIELD(SP),3) = 03H THEN  /* CAN'T BE */
1969:            CALL ERROR('IP');
1970:          ELSE CALL GEN$ADDR(PARM,PRT$ADDR(SP));
1971:        END;
1972:        ELSE DO;
1973:          IF NOT((SHL((BYTEPTR AND FORMMASK),3) OR VAR$ENTRY) =
1974:          FORM$FIELD(SP)) THEN
1975:            CALL ERROR('IP');
1976:          ELSE CALL GEN$ADDR(PARM,PRT$ADDR(SP));
1977:        END;
1978:      END;
1979:      PARMNUMLOC(SP+2) = PARMNUMLOC(SP) + 1;
1980:      IF SHR(FORM$FIELD(SP),7) THEN CALL GENERATE(LODI);
```

```
1981: END BUILT$IN$PARM;
1982:
1983:
1984:   /*************************************************
1985:    * ASSIGN$PARAMETERS - THIS PROCEDURE ENSURES    *
1986:    *  A PROPER MATCH UP BETWEEN THE SUBROUTINE'S   *
1987:    *  FORMAL PARAMETERS AND THE CALLING ACTUAL     *
1988:    *  PARAMETERS.                                  *
1989:    *************************************************/
1990:
1991: ASSIGN$PARMS: PROC;
1992:     IF SIGN$FLAG THEN
1993:     DO;
1994:       IF FORM$FIELD(MP-3) = BUILT$IN$FUNC THEN
1995:         CALL BUILT$IN$PARM;
1996:     END;
1997:     ELSE IF FORM$FIELD(MP-2) = BUILT$IN$FUNC THEN
1998:       CALL BUILT$IN$PARM;
1999:     ELSE DO;
2000:       APTRADDR = PARMNUMLOC(SP);
2001:       BASE = APTRADDR;
2002:       IF SHR(BYTEPTR,7) THEN
2003:       DO;
2004:         IF (BYTEPTR AND 7FH) = FORM$FIELD(SP) THEN
2005:           /* THIS IS A VARIABLE PARAMETER */
2006:           CALL GEN$ADDR(PARMV,PRT$ADDR(SP));
2007:         ELSE CALL ERROR('IP');
2008:       END;
2009:       ELSE DO;   /* THIS IS A VALUE PARAMETER */
2010:         IF (BYTEPTR = FORM$FIELD(SP))
2011:         OR (BYTEPTR = (FORM$FIELD(SP) AND 7FH)) THEN
2012:           CALL GEN$ADDR(PARM,PRT$ADDR(SP));
2013:         ELSE CALL ERROR('IP');
2014:       END;
2015:       PARMNUMLOC(SP+2) = PARMNUMLOC(SP) + 3;
2016:       READ$PARMS = TRUE;
```

```
2017:       END;
2018:  END ASSIGN$PARMS;
2019:
2020:
2021:     /*****************************************************/
2022:     /*  LOOKUP$IDENTIFIER - THIS PROCEDURE IS CALLED*/
2023:     /*   WITH 'SYMHASH' AND PRINTNAME SET. IT WILL  */
2024:     /*   RETURN TRUE IF THE IDENTIFIER CAN BE FOUND */
2025:     /*****************************************************/
2026:
2027:  LOOKUP$IDENT: PROC BYTE;
2028:       BASE=HASHTABLE(SYMHASH);
2029:       DO WHILE (BASE <> 0) AND (SBTBL > SCOPE(SCOPE$NUM));
2030:         IF CHK$PRT$NAME(6) THEN
2031:         DO;
2032:           LOOKUP$ADDR=BASE;
2033:           RETURN TRUE;
2034:         END;
2035:         ELSE DO;
2036:           CALL SETADDRPTR(0);
2037:           BASE=ADDRPTR;
2038:         END;
2039:       END;
2040:       RETURN FALSE;
2041:  END LOOKUP$IDENT;
2042:
2043:
2044:     /*****************************************************/
2045:     /*  LOOKUP$PRINTNAME$ONLY - THIS PROCEDURE SETS */
2046:     /*   THE "SYMHASH" AND CALLS LOOKUP$IDENT TO    */
2047:     /*   DETERMINE IF THE ENTRY IS IN THE SYMBOL    */
2048:     /*   TABLE.  THE ADDRESS OF THE PRINTNAME IS    */
2049:     /*   PASSED AS A PARAMETER. IF THE ENTRY IS     */
2050:     /*   FOUND, TRUE IS RETURNED.                   */
2051:     /*****************************************************/
2052:
```

```
2053: LOOKUP$PNAME: PROC(A) BYTE;
2054:     DCL A ADDR; /* ADDR OF PRINT-NAME */
2055:     DCL (B,N BASED A) BYTE;
2056:     HASHCODE=0;
2057:     DO B=1 TO N;
2058:       HASHCODE=(HASHCODE+N(B)) AND HASHMASK;
2059:     END;
2060:     SYMHASH=HASHCODE;
2061:     PRINTNAME=A;
2062:     RETURN LOOKUP$IDENT;
2063: END LOOKUP$PNAME;
2064:
2065:
2066:  /*************************************************/
2067:  /*  STORE$CONSTANT IDENTIFIER - THIS ROUTINE IS  */
2068:  /*    CALLED WITH PRINTNAME SET TO LOAD AN       */
2069:  /*    IDENTIFIER IN THE 'CONSTANT VALUE' VARIABLE.*/
2070:  /*************************************************/
2071:
2072: STORE$CONST: PROC;
2073:     DCL N BASED PRINTNAME BYTE;
2074:     CALL SETLOOKUP(SP);
2075:     CALL MOVE(PRINTNAME,.CONST$VALUE(CONST$INDX),(N+1));
2076:     CONST$INDX=CONST$INDX+(N+1);
2077:     CONST$PN$HASH(CONST$PN$PTR)=SYMHASH;
2078:     CONST$PN$SIZE(CONST$PN$PTR)=N+1;
2079:     CONST$PN$PTR=CONST$PN$PTR+1;
2080: END STORE$CONST;
2081:
2082:
2083:   /*************************************************
2084:    *  SUBRANGE$ERROR - THIS PROCEDURE IS CALLED   *
2085:    *    IN THE EVENT OF AN IMPROPER VALUE IN A    *
2086:    *    SUBRANGE.                                 *
2087:    *************************************************/
2088:
```

```
2089: SUBR$ERROR: PROC;
2090:     CALL ERROR('IS');
2091:     SUBR$TYPE(SUBR$PTR)=INTEGER$TYPE;
2092:     SUBR$VAL(SUBR$PTR)=0000H;
2093: END SUBR$ERROR;
2094:
2095:
2096:   /*************************************************
2097:    *  ORD$HIGH$LOW$CHECK - THIS PROCEDURE IS       *
2098:    *   CALLED TO ENSURE THE SECOND SUBRANGE VALUE  *
2099:    *   IS GREATER THAN THE FIRST.                  *
2100:    *************************************************/
2101:
2102: ORD$HI$LOW$CHK: PROC;
2103:     IF SUBR$PTR=0 THEN RETURN;
2104:     IF SUBR$TYPE=SUBR$TYPE(1) THEN
2105:       IF SUBR$VAL > SUBR$VAL(1) THEN RETURN;
2106:     CALL ERROR('IS');
2107: END ORD$HI$LOW$CHK;
2108:
2109:
2110:   /*************************************************
2111:    *  SUBRANGE$INTEGER$HI$LO$CHECK - THIS PROCE-   *
2112:    *   DURE IS CALLED TO ENSURE THAT BOTH SUB-     *
2113:    *   RANGE ELEMENTS ARE OF THE SAME TYPE, AND    *
2114:    *   THAT THEIR VALUES DO NOT EXCEED THE MAX     *
2115:    *   INTEGER VALUE.                              *
2116:    *************************************************/
2117:
2118: SUB$INT$HL$CHK: PROC;
2119:     IF SUBR$PTR=0 THEN RETURN;
2120:     IF SUBR$TYPE <> SUBR$TYPE(1) THEN
2121:     DO;
2122:       CALL SUBR$ERROR;
2123:       RETURN;
2124:     END;
```

```
2125:     IF SUBR$VAL < 32768 AND SUBR$VAL(1) >32767 THEN
2126:     DO;
2127:       INTEGER$DIFF = SUBR$VAL+( -SUBR$VAL(1))+1;
2128:       RETURN;
2129:     END;
2130:     IF SUBR$VAL > 32767 AND SUBR$VAL(1) < 32768 THEN
2131:     DO;
2132:       CALL SUBR$ERROR;
2133:       RETURN;
2134:     END;
2135:     IF SUBR$VAL < 32768 THEN /* BOTH POSITIVE */
2136:     DO;
2137:       IF(SUBR$VAL-(SUBR$VAL(1)+1)) < 32768 THEN
2138:       DO;
2139:         INTEGER$DIFF=SUBR$VAL-(SUBR$VAL(1))+1;
2140:         RETURN;
2141:       END;
2142:       CALL SUBR$ERROR;
2143:       RETURN;
2144:     END;
2145:     ELSE /* BOTH NEGATIVE */
2146:       IF ( - SUBR$VAL(1)-( - SUBR$VAL +1)) < 32768 THEN
2147:       DO;
2148:         INTEGER$DIFF=( - SUBR$VAL(1))-( - SUBR$VAL)+1;
2149:         RETURN;
2150:       END;
2151:     CALL SUBR$ERROR;
2152: END SUB$INT$HL$CHK;
2153:
2154:
2155:  /*****************************************************/
2156:  /*  SUBRANGE$IDENTIFER$PROCEDURE - THIS ROUTINE  */
2157:  /*   IS CALLED TO DETERMINE THE OFFSET ( NUMBER  */
2158:  /*   OF ENTRIES IN A SUBRANGE ) AND THE TYPE OF  */
2159:  /*   SUBRANGE, GIVEN THAT THE SUBRANGE TYPE IS   */
2160:  /*   A NAMED IDENTIFIER.                         */
```

```
2161:   /*****************************************************/
2162:
2163: SUB$ID$PROC: PROC;
2164:     CONST$PN$PTR=CONST$PN$PTR-1;
2165:     CONST$INDX=CONST$INDX-CONST$PN$SIZE(CONST$PN$PTR);
2166:     PRINTNAME=.CONST$VALUE(CONST$INDX);
2167:     SYMHASH=CONST$PN$HASH(CONST$PN$PTR);
2168:     IF NOT LOOKUP$IDENT THEN CALL SUBR$ERROR;
2169:     ELSE DO; /* FOUND CONSTANT IDENTIFIER */
2170:       BASE=LOOKUP$ADDR;
2171:       CALL SETADDRPTR(4); /* POINTS TO FORM(BYTEPTR) */
2172:       SUBR$FORM=BYTEPTR;
2173:       IF SUBR$FORM <> 07H AND (SUBR$FORM AND FORMMASK) <> CONS$ENTRY
2174:         THEN CALL SUBR$ERROR;
2175:       ELSE DO;
2176:         IF SUBR$FORM = 07H THEN
2177:         DO;
2178:           SUBR$TYPE(SUBR$PTR)=ORD$TYPE;
2179:           CALL SETADDRPTR(6);
2180:           SUBR$FORM=BYTEPTR; /* LENGTH OF P.NAME */
2181:           CALL SETADDRPTR(7+SUBR$FORM);
2182:           SUBR$VAL(SUBR$PTR)=DOUBLE(BYTEPTR);
2183:           CALL SETADDRPTR(7+SUBR$FORM);
2184:           SUB$TYP$ADDR(SUBR$PTR)=ADDRPTR;
2185:           CALL ORD$HI$LOW$CHK;
2186:         END;
2187:         ELSE DO;
2188:           DO WHILE ((SHR(SUBR$FORM,3) AND 3H)=0);
2189:             IF SHR(SUBR$FORM,5)=NEG THEN
2190:               IF SIGNVALU=POS THEN SIGNVALU=NEG;
2191:               ELSE SIGNVALU=POS;
2192:             CALL SETADDRPTR(6);
2193:             SUBR$FORM=BYTEPTR;
2194:             CALL SETADDRPTR(7+SUBR$FORM);
2195:             IF NOT LOOKUP$ONLY(APTRADDR) THEN
2196:             DO;
```

```
2197:                 CALL SUBR$ERROR;
2198:                 SUBR$PTR=SUBR$PTR+1;
2199:                 RETURN;
2200:               END;
2201:               ELSE DO;
2202:                 BASE=LOOKUP$ADDR;
2203:                 CALL SETADDRPTR(4);
2204:                 SUBR$FORM=BYTEPTR;
2205:               END;
2206:             END;
2207:             IF (SHR(SUBR$FORM,3)AND 3H) = 2 THEN
2208:             DO;
2209:               CALL SUBR$ERROR;
2210:               SUBR$PTR=SUBR$PTR+1;
2211:               RETURN;
2212:             END;
2213:             /* HERE WE HAVE EITHER AN INTEGER OR CHAR */
2214:             IF (SHR(SUBR$FORM,3) AND 3H) =  1 THEN
2215:             DO; /* INTEGER */
2216:               CALL SETADDRPTR(6);
2217:               SUBR$FORM=BYTEPTR;
2218:               CALL SETADDRPTR(7+SUBR$FORM);
2219:               IF SIGNVALU = NEG THEN
2220:                 SUBR$VAL(SUBR$PTR)=  - ADDRPTR;
2221:               ELSE SUBR$VAL(SUBR$PTR)=ADDRPTR;
2222:               SUBR$TYPE(SUBR$PTR)=INTEGER$TYPE;
2223:               CALL SUB$INT$HL$CHK;
2224:             END;
2225:             ELSE
2226:             DO;
2227:               CALL SETADDRPTR(6);
2228:               SUBR$FORM=BYTEPTR;
2229:               CALL SETADDRPTR(7+SUBR$FORM);
2230:               IF BYTEPTR <> 1 THEN
2231:               DO;
2232:                 CALL SUBR$ERROR;
```

```
2233:                 SUBR$PTR=SUBR$PTR+1;
2234:                 RETURN;
2235:               END;
2236:               CALL SETADDRPTR(8+SUBR$FORM);
2237:               IF BYTEPTR <41H OR BYTEPTR > 5AH THEN
2238:                 CALL SUBR$ERROR;
2239:               ELSE DO;
2240:                 SUBR$VAL(SUBR$PTR)=DOUBLE(BYTEPTR-41H);
2241:                 SUBR$TYPE(SUBR$PTR)=CHAR$TYPE;
2242:                 CALL ORD$HI$LOW$CHK;
2243:               END;
2244:             END;
2245:           END;
2246:         END;
2247:       END;
2248:       SUBR$PTR=SUBR$PTR+1;
2249: END SUB$ID$PROC;
2250:
2251:
2252:  /*****************************************************/
2253:  /*  SUBRANGE$CASE - THIS PROCEDURE IS USED TO    */
2254:  /*  DETERMINE THE NUMBER OF ENTRIES IN A SUBRANGE*/
2255:  /*****************************************************/
2256:
2257: SUBR$CASE: PROC;
2258:    SIGNVALU=POS;
2259:    DO CASE EXPRESS$STK(MP);
2260:     /* CASE CONST NUMBER */
2261:     DO; CONST$PTR=CONST$PTR-1;
2262:       IF CONST$NUM$TYPE(CONST$PTR)=REAL$TYPE THEN
2263:         DO;
2264:            CALL SUBR$ERROR;
2265:            CONST$INDX=CONST$INDX-BCDSIZE;
2266:         END;
2267:       ELSE
2268:         DO; /* INTEGER TYPE */
```

```
2269:            CONST$INDX=CONST$INDX-2;
2270:            CALL MOVE(.CONST$VALUE(CONST$INDX),.SUBR$VAL(SUBR$PTR),2);
2271:            SUBR$TYPE(SUBR$PTR)=INTEGER$TYPE;
2272:            CALL SUB$INT$HL$CHK;
2273:          END;
2274:        SUBR$PTR=SUBR$PTR+1; /* NEXT TO FILL */
2275:      END;
2276:      /* CASE IDENT CONSTANT */
2277:      CALL SUB$ID$PROC;
2278:      /* CASE SIGNED IDENT CONSTANT */
2279:      DO;
2280:        SIGNVALU=NEG;
2281:        CALL SUB$ID$PROC;
2282:      END;
2283:      /* CASE CONSTANT STRING */
2284:      DO;
2285:         CONST$PN$PTR=CONST$PN$PTR-1;
2286:        CONST$INDX=CONST$INDX-CONST$PN$SIZE(CONST$PN$PTR);
2287:        PRINTNAME=.CONST$VALUE(CONST$INDX);
2288:        IF CONST$PN$SIZE(CONST$PN$PTR) <> 2 THEN
2289:          CALL SUBR$ERROR;
2290:        ELSE
2291:          DO;
2292:            BASE=PRINTNAME;
2293:            CALL SETADDRPTR(1);
2294:            IF BYTEPTR < 41H OR BYTEPTR > 5AH THEN
2295:              CALL SUBR$ERROR;
2296:            ELSE
2297:              DO;
2298:                SUBR$VAL(SUBR$PTR)=DOUBLE(BYTEPTR-41H);
2299:                SUBR$TYPE(SUBR$PTR)=CHAR$TYPE;
2300:                CALL ORD$HI$LOW$CHK;
2301:              END;
2302:          END;
2303:        SUBR$PTR=SUBR$PTR+1;
2304:      END;
```

```
2305:     END; /* OF CASE EXPRESS$STK(MP) */
2306: END SUBR$CASE;
2307:
2308:
2309:  /*************************************************/
2310:  /*  ENTER$SUBRANGE$ENTRY - THIS PROCEDURE IS     */
2311:  /*    USED TO ENTER A SUBRANGE TYPE ENTRY INTO   */
2312:  /*    THE SYMBOL TABLE. THIS SYMBOL TABLE ENTRY  */
2313:  /*    HAS NO PRINTNAME ASSOCIATED WITH IT.       */
2314:  /*************************************************/
2315:
2316: ENTR$SUB$NTRY: PROC;
2317:     TYPE$LOCT=SBTBL;
2318:     CALL LIMITS(14);
2319:     VECPTR=VECPTR-1;
2320:     CALL SUBR$CASE;
2321:     VECPTR=VECPTR-1;
2322:     CALL SUBR$CASE;
2323:     CALL ENTR$CPLX$TYP(SHL(SUBR$TYPE,6)OR 0FH);
2324:     CALL SETADDRPTR(5);
2325:     IF SUBR$TYPE=INTEGER$TYPE THEN
2326:       ADDRPTR=.BUILT$IN$TBL;
2327:     IF SUBR$TYPE=CHAR$TYPE THEN ADDRPTR=(.BUILT$IN$TBL+23);
2328:     IF SUBR$TYPE=ORD$TYPE THEN ADDRPTR=SUB$TYP$ADDR;
2329:     CALL SETADDRPTR(7);
2330:     ADDRPTR=SUBR$VAL(1);
2331:     CALL SETADDRPTR(9);
2332:     ADDRPTR=SUBR$VAL;
2333:     CALL SETADDRPTR(11);
2334:     IF SUBR$TYPE=INTEGER$TYPE THEN /* RANGE 0 TO 64K */
2335:       ADDRPTR=INTEGER$DIFF; /* MAY BE GREATER THAN 32767 */
2336:     ELSE
2337:       ADDRPTR=((SUBR$VAL-SUBR$VAL(1))+1);
2338:     SUBR$PTR=0;
2339:     SBTBL=SBTBL+10;
2340: END ENTR$SUB$NTRY;
```

```
2341:
2342:
2343:    /*************************************************
2344:     *  TYPE$ERROR - THIS PROCEDURE IS CALLED IN THE*
2345:     *   EVENT OF AN INCOMPATIBLE TYPE.              *
2346:     *************************************************/
2347:
2348: TYPE$ERROR: PROC;
2349:     ALLOCATE=FALSE;
2350:     CALL ERROR('IT');
2351: END TYPE$ERROR;
2352:
2353:
2354:  /*************************************************/
2355:  /*  ALLOCATE OFFSET - THIS PROCEDURE IS CALLED TO*/
2356:  /*   DETERMINE THE NUMBER OF BYTES REQUIRED FOR  */
2357:  /*   STORAGE OF A VARIABLE OF THE TYPE GIVEN IN  */
2358:  /*   THE PARAMETER 'A'. THE VARIABLE'S ALLC$QTY  */
2359:  /*   AND ALLC$FORM ARE SET UPON RETURN.          */
2360:  /*************************************************/
2361:
2362: ALLC$OFFSET: PROC(A); /* TYPE$LOCT */
2363:     DCL A ADDR;
2364:     DCL (ALLC$FORM,B) BYTE;
2365:     BASE=A;
2366:     CALL SETADDRPTR(4); /* POINTS TO FORM OF TYPE */
2367:     ALLC$FORM= BYTEPTR AND FORMMASK;
2368:     IF ALLC$FORM <> TYPE$ENTRY AND ALLC$FORM <> TYPEDCLE THEN
2369:      DO;
2370:      CALL TYPE$ERROR;
2371:      ALLC$QTY=1;
2372:      ALOCBASICTYP=0;
2373:      RETURN;
2374:     END;
2375:     DO WHILE((SHR(BYTEPTR,3)AND FORMMASK)=7 AND ALLC$FORM=TYPE$ENTRY);
2376:      CALL SET$PAST$PN(7);
```

```
2377:       BASE=ADDRPTR; CALL SETADDRPTR(4);
2378:       ALLC$FORM=BYTEPTR AND FORMMASK;
2379:       IF ALLC$FORM <> TYPE$ENTRY AND ALLC$FORM <> TYPEDCLE THEN
2380:        DO; CALL TYPE$ERROR;
2381:          ALLC$QTY=1;
2382:          ALOCBASICTYP=0; RETURN;
2383:        END;
2384:      END;
2385:  /* HERE EXISTS EITHER A BASIC TYPE OR A TYPE DECLARATION */
2386:      IF ALLC$FORM = TYPE$ENTRY THEN
2387:       DO; /* BASIC TYPE */
2388:        DO CASE (SHR(BYTEPTR,3) AND FORMMASK);
2389:          /* INTEGER */
2390:         DO;
2391:           ALLC$QTY=2;
2392:           ALOCBASICTYP=INTEGER$TYPE;
2393:         END;
2394:          /* BCD REAL */
2395:         DO;
2396:           ALLC$QTY=8;
2397:           ALOCBASICTYP=UNSIGN$EXPON;
2398:         END;
2399:          /* CHARACTER */
2400:         DO;
2401:           ALLC$QTY=1;
2402:           ALOCBASICTYP=CHAR$TYPE;
2403:         END;
2404:          /* BOOLEAN */
2405:         DO;
2406:           ALLC$QTY=1;
2407:           ALOCBASICTYP=BOOLEAN$TYPE;
2408:         END;
2409:          /* TEXT */
2410:         DO;
2411:           ALLC$QTY = 2;
2412:           ALOCBASICTYP = STRING$TYPE;
```

```
2413:          END;
2414:        END; /* OF CASE */
2415:        ALLOCATE=TRUE;
2416:        RETURN;
2417:      END;
2418: /* HERE EXISTS A TYPE DECLARATION */
2419:      TEMPBYTE1,ALLC$FORM=(SHR(BYTEPTR,3)AND FORMMASK);
2420:      IF AILC$FORM=0 THEN
2421:       DO; /* SCALAR */
2422:        ALLOCATE=TRUE;
2423:        ALLC$QTY=DOUBLE(ALLC$FORM+1);
2424:        ALOCBASICTYP=ORD$TYPE; RETURN;
2425:       END;
2426:      IF ALLC$FORM=1 THEN
2427:       DO; /* SUBRANGE */
2428:        ALLOCATE=TRUE;
2429:        ALOCBASICTYP=COMPLEX$TYPE;
2430:        B=SHR(BYTEPTR,6);
2431:        IF B = 1 THEN ALLC$QTY=DOUBLE(ALLC$FORM+1);
2432:        ELSF ALLC$QTY=DOUBLE(ALLC$FORM); RETURN;
2433:       END;
2434:      IF ALLC$FORM=2 THEN
2435:       DO; /* ARRAY */
2436:        ALLOCATE=TRUE;
2437:        ALOCBASICTYP=COMPLEX$TYPE;
2438:        CALL SETADDRPTR(8);
2439:        ALLC$QTY=ADDRPTR; RETURN;
2440:       END;
2441:      B=2;
2442:      /* ALL OTHER CASES ALLOCATE AN ADDRESS FIELD */
2443:      ALLC$QTY=DOUBLE(B);
2444:      ALOCBASICTYP=COMPLEX$TYPE;
2445:      ALLOCATE=TRUE;
2446: END ALLC$OFFSET;
2447:
2448:
```

```
2449:  /*************************************************************/
2450:  /*  AL$NDX$OFFSET - THIS PROCEDURE IS CALLED                 */
2451:  /*   TO DETERMINE THE NUMBER OF BYTES REQUIRED               */
2452:  /*   BY AN ARRAY TO STORE THE ARRAY'S COMPONENTS             */
2453:  /*   TYPE$LOCT IS SET PRIOR TO CALLING THIS                  */
2454:  /*   ROUTINE. AN ADDRESS VARIABLE CONTAINING THE             */
2455:  /*   BYTE COUNT IS RETURNED.                                 */
2456:  /*************************************************************/
2457:
2458: AL$NDX$OFFSET: PROC ADDR;
2459:    DCL A ADDR,B BYTE;
2460:     A,BASE=TYPE$LOCT;
2461:     CALL SETADDRPTR(4);
2462:     DO WHILE (SHR(BYTEPTR,3) AND FORMMASK) = 7 AND
2463:              ( BYTEPTR AND FORMMASK ) = TYPE$ENTRY;
2464:      CALL SET$PAST$PN(7);
2465:      BASE=ADDRPTR; CALL SETADDRPTR(4);
2466:     END;
2467:  /* HERE WE HAVE EITHER A SCALAR,SUBRANGE,BOOLEAN, OR CHAR TYPE */
2468:     B= SHR(BYTEPTR,3) AND FORMMASK;
2469:     IF (BYTEPTR AND FORMMASK) = TYPE$ENTRY THEN
2470:      DO;
2471:        IF B = 0 OR B = 1 THEN
2472:          DO;
2473:            CALL ERROR('IA');
2474:            B=2;
2475:            RETURN DOUBLE(B);
2476:          END;
2477:       IF B=2 THEN /* CHARACTER SUBRANGE */
2478:         DO;
2479:          B = 26;
2480:          REC$VAR$TYP(REC$NST)=CHAR$TYPE;
2481:          RETURN DOUBLE(B);
2482:         END;
2483:       /* BOOLEAN */
2484:       REC$VAR$TYP(REC$NST)=BOOLEAN$TYPE;
```

```
2485:        B = 2; RETURN DOUBLE(F);
2486:       END;
2487:   /* COMPLEX TYPE */
2488:     IF (( BYTEPTR AND FORMMASK) <> TYPE$DCLE OR
2489:        (( B <> 0 ) AND ( B <> 1 ))) THEN
2490:       DO;
2491:         CALL ERROR('IA');
2492:         B=2; RETURN DOUBLE(B);
2493:       END;
2494:     IF B=0 THEN
2495:       DO; /* SCALAR TYPE */
2496:        REC$VAR$TYP(REC$NST)=COMPLEX$TYPE;
2497:        CALL SET$PAST$PN(7);
2498:        RETURN DOUBLE(BYTEPTR + 1);
2499:       END;
2500:   /* SUBRANGE TYPE */
2501:      REC$VAR$TYP(REC$NST)=ORD$TYPE;
2502:      CALL SETADDRPTR(11);
2503:      RETURN ADDRPTR;
2504: END AL$NDX$OFFSET;
2505:
2506:
2507:   /*************************************************
2508:    *  ALLOCATE$VARIABLES - THIS PROCEDURE IS     *
2509:    *   CALLED TO ASSIGN PRT LOCATIONS FOR EACH   *
2510:    *   OF THE PROGRAM VARIABLES.                 *
2511:    *************************************************/
2512:
2513: ALLOC$VARS: PROC;
2514:      TEMPBYTE1 = 0;
2515:      CALL ALLC$OFFSET(TYPE$LOCT);
2516:      TEMPBYTE = VAR$PTR;
2517:      DO VAR$PTR = 0 TO TEMPBYTE;
2518:        BASE=VAR$BASE(VAR$PTR);
2519:        CALL SETADDRPTR(4);
2520:        IF SHR(BYTEPTR,7) THEN
```

```
2521:         DO;
2522:           BYTEPTR = (BYTEPTR) OR (SHL(ALOCBASICTYP,3) OR VAR$ENTRY);
2523:           APTRADDR = VAR$BASE1(VAR$PTR);
2524:           ADDRPTR = ALLOC$ADDR;
2525:           ALLOC$ADDR = ALLOC$ADDR + 2;
2526:         END;
2527:         ELSE DO;
2528:           BYTEPTR=SHL(ALOCBASICTYP,3) OR VAR$ENTRY;
2529: /*        IF (BYTEPTR = 23H) AND (TEMPBYTE1 = 2) THEN
2530:           DO;
2531:             APTRADDR = TYPE$LOCT + 8;
2532:             ALLC$QTY = ADDRPTR;
2533:           END;
2534:           IF TEMPBYTE1 = 3 THEN
2535:           DO;
2536:             APTRADDR = TYPE$LOCT + 6;
2537:             APTRADDR = APTRADDR + BYTEPTR + 1;
2538:             APTRADDR = ADDRPTR + 5;
2539:             ALLC$QTY = ADDRPTR;
2540:           END;  */
2541:           APTRADDR=VAR$BASE1(VAR$PTR);
2542:           ADDRPTR=ALLOC$ADDR;
2543:           ALLOC$ADDR=ALLOC$ADDR+ALLC$QTY;
2544:         END;
2545:         APTRADDR=APTRADDR+2;
2546:         ADDRPTR=TYPE$LOCT;
2547:       END;
2548:       TEMPBYTE1 = 0;
2549: END ALLOC$VARS;
2550:
2551:
2552:   /*************************************************
2553:    *  CASE$PTRPTR - THIS PROCEDURE IS CALLED TO    *
2554:    *   SET A VARIABLE'S APPROPRIATE TYPE.          *
2555:    *************************************************/
2556:
```

```
2557: CASE$PTRPTR: PROC(A);
2558:     DCL A BYTE;
2559:     DO CASE A;
2560:      /* CASE Ø ORD VARIABLE */
2561:      DO;
2562:        PTRPTR = 10H;
2563:        CALL SET$PAST$PN(9);
2564:        BASE$LOC(SP) = ADDRPTR;    /* ADDR OF PARENT */
2565:      END;
2566:      /* CASE 1 INTEGER VARIABLE */
2567:      PTRPTR = Ø9H;
2568:      /* CASE 2 CHAR VARIABLE */
2569:      PTRPTR = ØBH;
2570:      /* CASE 3 REAL VARIABLE */
2571:      PTRPTR = ØAH;
2572:      /* CASE 4 COMPLEX VARIABLE */
2573:        DO; /* ARRAY, SUBRANGE, USER DEFINED TYPES */
2574:          TEMPADDR = BASE;  /* STORE VARIABLE SBTL LOCATION */
2575:          CALL SET$PAST$PN(9);
2576:          BASE = ADDRPTR;
2577:          CALL SETADDRPTR(4);
2578:          IF BYTEPTR = 17H THEN        /*   ARRAY   */
2579:          DO;
2580:            APTRADDR = APTRADDR + 6;
2581:            TEMPBYTE1 = BYTEPTR;
2582:          END;
2583:          ELSE IF (BYTEPTR AND ØFH) = ØFH THEN  /* SUBRANGE TYPES */
2584:            TEMPBYTE1 = SHR(BYTEPTR, 6);
2585:          ELSE IF BYTEPTR = 7AH THEN
2586:          DO;   /* USER DEFINED TYPE */
2587:            TEMPBYTE1 = Ø;
2588:            CALL SET$PAST$PN(7);
2589:            BASE = ADDRPTR;
2590:            CALL SETADDRPTR(4);
2591:            IF BYTEPTR <> 27H THEN CALL ERROR('NS');
2592:           /* THIS IS A SET TYPE */
```

```
2593:             CALL SETADDRPTR(5);
2594:             BASE$LOC(SP) = ADDRPTR;    /* ADDR OF PARENT */
2595:           END;
2596:           ELSE IF BYTEPTR = 37H THEN
2597:           DO;    /* POINTER */
2598:             CALL SETADDRPTR(5);
2599:             BASE$LOC(SP) = ADDRPTR;    /* ADDR OF PARENT */
2600:           END;
2601:           FLSE TEMPBYTE1 = 06H;
2602:           DO CASE TEMPBYTE1;
2603:             PTRPTR = 10H;
2604:             PTRPTR = 09H;
2605:             PTRPTR = 0BH;
2606:             PTRPTR = 0AH;
2607:             PTRPTR = 0CH;
2608:             PTRPTR = 08H;
2609:             PTRPTR = 0CH;
2610:           END;    /*  OF CASE  */
2611:           BASE = TEMPADDR;  /* RESTORE ORIGINAL BASE LOCATION */
2612:         END;
2613:       /* CASE 5 BOOLEAN VARIABLE */
2614:       PTRPTR = 05H;
2615:     END; /* OF VARIABLE CASE */
2616: END CASE$PTRPTR;
2617:
2618:
2619:  /*****************************************************/
2620:  /*  SET$VARIABLE$TYPE - THIS PROCEDURE IS CALLED */
2621:  /*   TO SET THE VARIABLE TYPE, VARIABLE SIGN,AND */
2622:  /*   ADDRESS OF THE BASIC TYPE GIVEN. THE ADDRESS*/
2623:  /*   VARIABLE 'LOOKUP$ADDR' IS SET PRIOR TO THE  */
2624:  /*   CALL.                                       */
2625:  /*****************************************************/
2626:
2627: SET$VAR$TYPE: PROC;
2628:
```

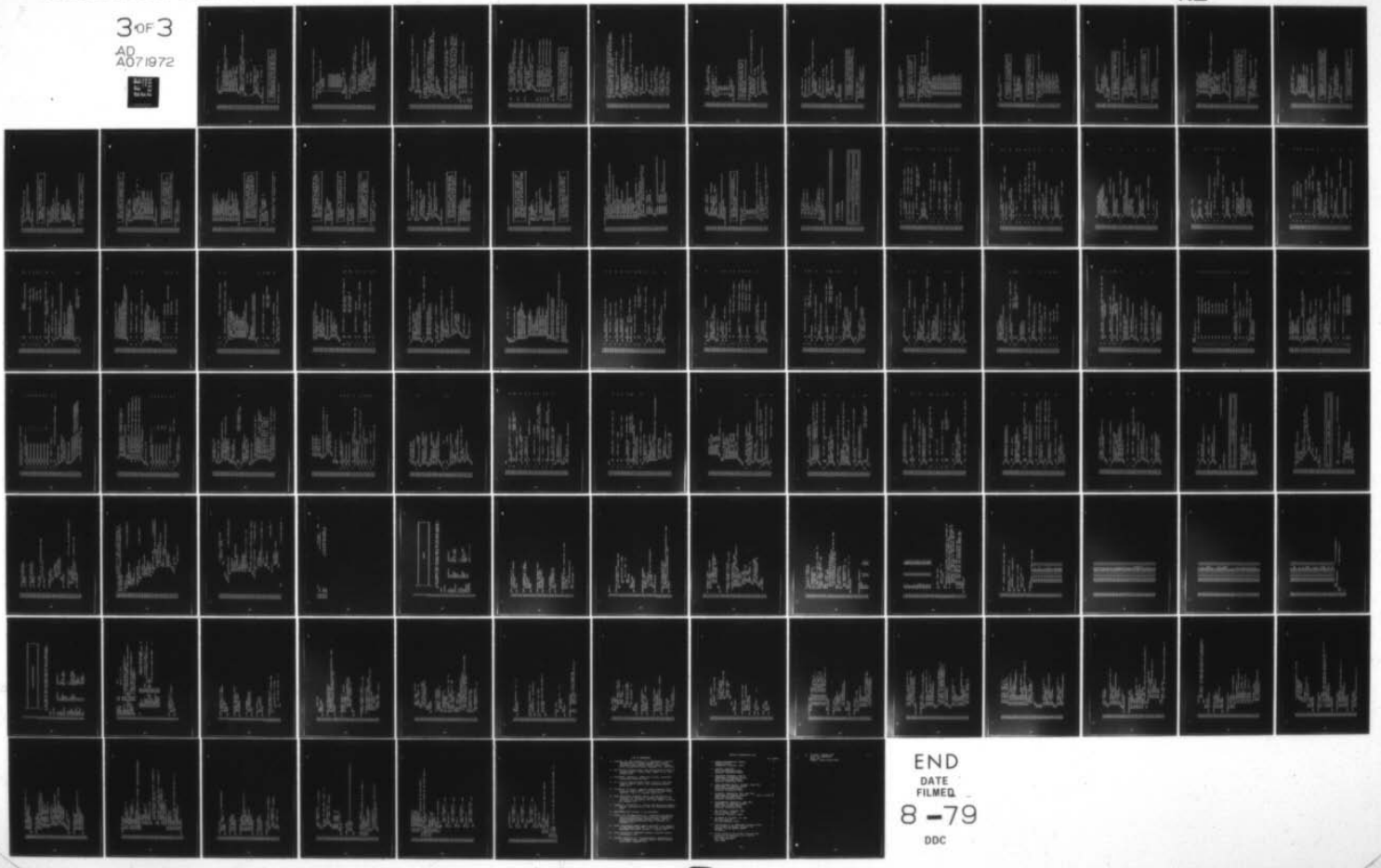
```
2629:     SET$TYP$N$LOC: PROC(A,B,C);
2630:     DCL (A, B, C) BYTE;
2631:          CALL SET$PAST$PN(A);
2632:          IF (B=04H) OR (B=05H) OR (B=06H) OR (B=11H) THEN
2633:            PRTADDR(SP) = APTRADDR;
2634:          ELSE PRTADDR(SP) = ADDRPTR;
2635:          TYPE$STACK(SP) = (B OR ROL(C, 7));
2636:     END SET$TYP$N$LOC;
2637:
2638:     BASE = LOOKUP$ADDR;
2639:     CALL SETADDRPTR(4);
2640:     FORM$FIELD(SP) = BYTEPTR;
2641:     DO CASE (FORM$FIELD(SP) AND FORMMASK);
2642:       ;
2643:       /*          CONSTANT ENTRY          */
2644:       DO;
2645:         SIGN$VALU = POS;
2646:         DO CASE (SHR(BYTEPTR,3) AND 03H);
2647:         /* FIND OUT WHAT KIND OF CONSTANT IT IS */
2648:           DO WHILE (SHR(BYTEPTR,3) AND 03H) = 0;
2649:             IF (SHR(BYTEPTR,5) AND 01H) = 01H THEN
2650:               IF SIGN$VALU THEN SIGN$VALU = NEG;
2651:               ELSE SIGN$VALU = POS;
2652:             CALL SETADDRPTR(6);
2653:             IF NOT LOOKUP$PNAME(APTRADDR) THEN
2654:             DO;
2655:               CALL ERROR('IC');
2656:               RETURN;
2657:             END;
2658:             CALL SETALDRPTR(4);
2659:             IF (BYTEPTR AND FORMMASK) <> CONS$ENTRY THEN
2660:             DO;
2661:               CALL ERROR('IC');
2662:               RETURN;
2663:             END;
2664:           END;
```

```
2665:           /*      INTEGER OR BOOLEAN CONSTANT      */
2666:           IF BASE < .MEMORY THEN          /* BOOLEAN */
2667:             CALL SET$TYP$N$LOC(9,4H,POS);
2668:           ELSE                            /* INTEGER */
2669:             CALL SET$TYP$N$LOC(7,5H,SIGN$VALU);
2670:           /*      REAL CONSTANT      */
2671:           CALL SET$TYP$N$LOC(7,6H,SIGN$VALU);
2672:            /*      STRING CONSTANT      */
2673:            CALL SET$TYP$N$LOC(7,7H,0);
2674:         END;          /*      OF CASE      */
2675:       END;
2676:       /*           TYPE ENTRY           */
2677:       ;
2678:       /*           VARIABLE ENTRY           */
2679:       DO;
2680:         IF SHR(FORM$FIELD(SP),7) THEN VARPARM = TRUE;
2681:         PTRPTR = (SHR(FORM$FIELD(SP),3) AND FORMMASK);
2682:         BASE$LOC(SP) = BASE;  /* SYMBOL TABLE LOCATION OF VARIABLE */
2683:         CALL CASEPTRPTR(PTRPTR);
2684:         CALL SET$TYP$N$LOC(7,PTRPTR,0);
2685:       END;
2686:       /*           PROCEDURE ENTRY           */
2687:       ;          /*  NO SUCH THING EXISTS IN PASCAL  */
2688:       /*           FUNCTION ENTRY           */
2689:       DO;
2690:         IF FORM$FIELD(SP) = BUILT$IN$FUNC THEN /* BUILT IN FUNCTION */
2691:         DO;
2692:           CALL SET$PAST$PN(8);
2693:           IF BYTEPTR <> 13H THEN
2694:             IF BYTEPTR <> 0F3H THEN
2695:             DO;
2696:               CALL CASEPTRPTR(BYTEPTR);
2697:               TYPE$STACK(SP) = PTRPTR;
2698:             END;
2699:           APTRADDR = APTRADDR + 1;
2700:           PARMNUM(SP) = BYTEPTR;
```

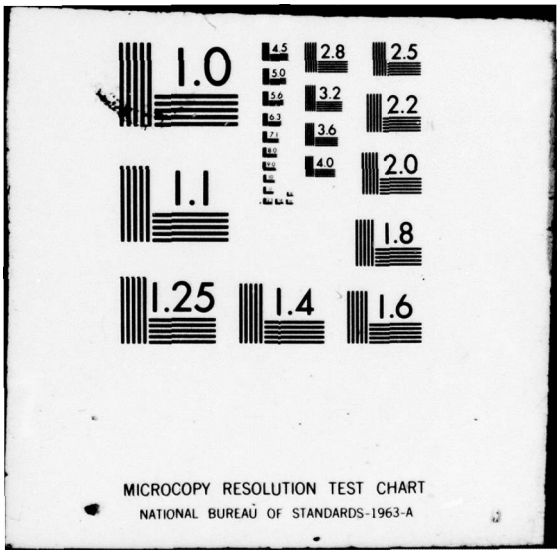
AD-A071 972 NAVAL POSTGRADUATE SCHOOL MONTEREY CA F/G 9/2
NPS-PASCAL: A PASCAL IMPLEMENTATION FOR MICROPROCESSOR-BASED CO--ETC(U)
JUN 79 J L BYRNES

UNCLASSIFIED NL

3 OF 3
AD
A071972



END
DATE
FILMED
8-79
DDC


1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
4.5
5.0
5.6
6.3
MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

```
2701:             PARMNUMLOC(SP) = APTRADDR + 1;
2702:           END;
2703:           ELSE DO;
2704:             CALL SET$PAST$PN(16);
2705:             CALL CASEPTRPTR(SHR(BYTEPTR,3) AND FORMMASK);
2706:             CALL SET$TYP$N$LOC(10,PTRPTR,0);
2707:             CALL SET$PAST$PN(7);
2708:             PARMNUM(SP) = BYTEPTR;
2709:             CALL SET$PAST$PN(8);
2710:             PARMNUMLOC(SP) = ADDRPTR;
2711:             CALL SET$PAST$PN(14);
2712:             LABELSTACK(SP) = ADDRPTR;
2713:           END;
2714:           IF TOKEN <> 18 THEN READPARMS = TRUE;
2715:         /*  OTHERWISE, THIS WILL BE A FUNCTION ASSIGNMENT STATEMENT  */
2716:           PARMNUMLOC(SP+2) = PARMNUMLOC(SP);
2717:         END;
2718:         /*          FILE ENTRY          */
2719:         ;
2720:         /*          SCALAR ENTRY          */
2721:         DO;
2722:           CALL SET$TYP$N$LOC(7,11H,0);
2723:           APTRADDR = APTRADDR + 1;
2724:           BASE$LOC(SP) = ADDRPTR;
2725:         END;
2726:       END;          /*   OF CASE   */
2727: END SET$VAR$TYPE;
2728:
2729:
2730:  /*****************************************************/
2731:  /*  LOAD$VARI - THIS PROCEDURE GENERATES THE     */
2732:  /*   INTERMEDIATE CODE TO LOAD THE NEXT VARIABLE */
2733:  /*   ON THE EXECUTION STACK OF THE OBJECT FILE   */
2734:  /*****************************************************/
2735:
2736: LOAD$VARI: PROC(PT);
```

```
2737:     DCL PT BYTE;  /* PT REPRESENTS A STACK POINTER */
2738:
2739:     EXP$STACK: PROC;
2740:          DCL A BYTE;
2741:          DO CASE (TYPE$STACK(PT) AND 0FH);
2742:            A = ORD$TYPE;
2743:            A = ORD$TYPE;
2744:            ;
2745:            ;
2746:            A = BOOLEAN$TYPE;
2747:            A = INTEGER$TYPE;
2748:            A = UNSIGN$EXPON;
2749:            A = STRING$TYPE;
2750:            A = BOOLEAN$TYPE;
2751:            A = INTEGER$TYPE;
2752:            A = UNSIGN$EXPON;
2753:            A = CHAR$TYPE;
2754:          END;        /* OF CASE */
2755:          EXPRESS$STK(PT) = A;
2756:     END EXP$STACK;
2757:
2758:     LOAD: PROC(A, B, C);
2759:          DCL (A, B, C) BYTE;
2760:          /* CHECK IF LOADING A FUNCTION VALUE */
2761:          IF (FORM$FIELD(PT) AND 7FH) <> FUNC$ENTRY THEN
2762:          DO;
2763:            CALL GENERATE(A);
2764:            CALL GENERATE(B);
2765:            IF SHR(TYPE$STACK(PT), 6) THEN /* ACCESSING ARRAY */
2766:              CALL GENERATE(SUB);
2767:            ELSE CALL GENERATE(C);
2768:            IF A = LDIB THEN     /* LOAD REST OF BCD NUMBER */
2769:            DO PTRPTR = 2 TO (BCDNUM/2);
2770:                 APTRADDR = APTRADDR + 2;
2771:                 CALL GENERATE(BYTEPTR);
2772:                 CALL GENERATE(HIGH(ADDRPTR));
```

```
2773:              END;
2774:              IF SHR(FORM$FIELD(PT),7) THEN /* VARIABLE PARAMETER */
2775:                CALL GENERATE(LODI);
2776:            END;
2777:            ELSE CALL GEN$ADDR(PRO,LABELSTACK(MP));
2778:            CALL EXP$STACK;
2779:      END LOAD;
2780:
2781:      IF READSTMT THEN RETURN; /* GOING TO READ THIS VALUE */
2782:      IF READPARMS THEN
2783:      DO;   /* READING A SUBROUTINE'S PARAMETERS */
2784:        IF (TOKEN <> 12) AND (TOKEN <> 8) THEN READPARMS = FALSE;
2785:        /* THIS MEANS THIS PARAMETER IS AN EXPRESSION THAT MUST BE
2786:           EVALUATED.  AFTER EVALUATION, READPARMS WILL BE SET TO TRUE. */
2787:        ELSE DO;
2788:          CALL ASSIGN$PARMS;
2789:          CALL EXP$STACK;
2790:          RETURN;
2791:        END;
2792:      END;
2793:      /* IF LOADING A FUNCTION VALUE, GO TO THE CASE STATEMENT */
2794:      IF (FORM$FIELD(MP) AND 7FH) <> FUNC$ENTRY THEN
2795:      DO;
2796:        IF ((TYPE$STACK(PT) > 08H) AND (TYPE$STACK(PT) < 11H)) OR
2797:        ((TYPE$STACK(PT) AND 40H) = 40H) THEN /* IN CASE OF ARRAYS */
2798:          CALL GENERATE(LITA);   /* GOING TO LOAD A PRT ADDR */
2799:        ELSE APTRADDR = PRTADDR(PT); /* GOING TO LOAD A CONSTANT */
2800:      END;
2801:      DO CASE (TYPE$STACK(PT) AND 0FH);
2802: /*0*/     /*   ORD VARIABLE   */
2803:           CALL LOAD(LOW(PRTADDR(PT)),HIGH(PRTADDR(PT)),LOD);
2804: /*1*/     /*   ORD CONSTANT   */
2805:           CALL LOAD(LDII,BYTEPTR,NOP);
2806: /*2*/     ;
2807: /*3*/     ;
2808: /*4*/     /*   BOOLEAN CONSTANT   */
```

```
2809:             CALL LOAD(LDII,BYTEPTR,NOP);
2810: /*5*/       DO;    /* INTEGER CONSTANT */
2811:               CALL LOAD(LDII,BYTEPTR,HIGH(ADDRPTR));
2812:               IF TYPE$STACK(PT) = 85H THEN CALL GENERATE(NEGI);
2813:             END;
2814: /*6*/       DO;    /* BCD CONSTANT */
2815:               CALL LOAD(LDIB,BYTEPTR,HIGH(ADDRPTR));
2816:               IF TYPE$STACK(PT) = 86H THEN CALL GENERATE(NEGB);
2817:             END;
2818: /*7*/       DO;    /* STRING CONSTANT */
2819:               CALL GENERATE(LDSI);
2820:               TEMPBYTE = BYTEPTR;    /* LENGTH OF STRING */
2821:               DO PTRPTR = 0 TO TEMPBYTE;
2822:                 CALL GENERATE(APTRADDR + PTRPTR);
2823:               END;
2824:             END;
2825: /*8*/       /*    BOOLEAN VARIABLE    */
2826:             CALL LOAD(LOW(PRTADDR(PT)),HIGH(PRTADDR(PT)),LOD);
2827: /*9*/       /*    INTEGER VARIABLE    */
2828:             CALL LOAD(LOW(PRTADDR(PT)),HIGH(PRTADDR(PT)),LODI);
2829: /*A*/       /*    REAL VARIABLE    */
2830:             CALL LOAD(LOW(PRTADDR(PT)),HIGH(PRTADDR(PT)),LODB);
2831: /*B*/       /*    CHARACTER VARIABLE    */
2832:             CALL LOAD(LOW(PRTADDR(PT)),HIGH(PRTADDR(PT)),LOD);
2833:         END;       /*    OF CASE    */
2834: END LOAD$VARI;
2835:
2836:
2837:  /*****************************************************/
2838:  /*  ASSIGN$VARI - THIS PROCEDURE GENERATES THE    */
2839:  /*    INTERMEDIATE CODE TO LOAD THE LEFT SIDE OF  */
2840:  /*    AN ASSIGNMENT STATEMENT ON THE EXECUTION    */
2841:  /*    STACK AND STORES A RESULT AT THAT LOCATION. */
2842:  /*****************************************************/
2843:
2844: ASSIGN$VARI: PROC(LS, STORE$TYPE);
```

```
2845:     DCL LS BYTE;  /* LS IS THE LEFT SIDE OF ASSMT STMT */
2846:     DCL (A, B, STORE$TYPE) BYTE;  /* STORE$TYPE INDICATES WHETHER
2847:     TO DELETE OR LEAVE THE CURRENT VALUE AT THE TOP OF THE STACK */
2848:     IF (TYPE$STACK(LS) AND 40H) = 40H THEN
2849:     DO;
2850:       TYPE$STACK(LS) = (TYPE$STACK(LS) AND 0BFH);
2851:       CALL GENERATE(XCHG);
2852:     END;
2853:     ELSE CALL GEN$ADDR(LITA,PRT$ADDR(LS));
2854:     IF SHR(FORM$FIELD(LS),7) THEN  /* CHECK FOR VAR PARAMETER */
2855:       CALL GENERATE(LODI);
2856:     DO CASE EXPRESS$STK(SP);
2857:       /* CASE 0 - ORD TYPE */
2858:       IF (TYPE$STACK(LS) <> 11H) AND (TYPE$STACK(LS) <> 10H) THEN
2859:         CALL ERROR('AT');
2860:       ELSE  A = 2;
2861:       /* CASE 1 - INTEGER TYPE */
2862:       IF TYPE$STACK(LS) = 09H THEN
2863:          A = 1;
2864:       ELSE DO;
2865:         IF TYPE$STACK(LS) = 0AH THEN
2866:         DO;
2867:           CALL GENERATE(CNAI);
2868:            A = 0;
2869:         END;
2870:         ELSE CALL ERROR('AT');
2871:       END;
2872:       /* CASE 2 - CHAR$TYPE */
2873:       IF TYPE$STACK(LS) = 0BH THEN
2874:          A = 2;
2875:       ELSE CALL ERROR('AT');
2876:       /* CASE 3 - REAL TYPE */
2877:       IF TYPE$STACK(LS) = 0AH THEN
2878:          A = 0;
2879:       ELSE CALL ERROR('AT');
2880:       /* CASE 4 - STRING TYPE */
```

```
2
 881:       A = 2;
2882:       /* CASE 5 - BOOLEAN TYPE */
2883:       IF TYPE$STACK(LS) = 08H THEN
2884:          A = 2;
2885:       ELSE CALL ERROR('AT');
2886:     END; /* OF CASE */
2887:     IF STORE$TYPE THEN A = A + 3;
2888:     DO CASE A;
2889:       B = STDB;
2890:       B = STDI;
2891:       B = STD;
2892:       B = STOB;
2893:       B = STOI;
2894:       B = STO;
2895:     END;   /* OF CASE */
2896:     CALL GENERATE(B);
2897: END ASSIGN$VARI;
2898:
2899:
2900:  /*************************************/
2901:  /* THIS PROCEDURE CHECKS THE TOP TWO */
2902:  /* VARIABLES ON THE EXECUTION STACK  */
2903:  /* FOR PROPER TYPE.                  */
2904:  /*************************************/
2905:
2906: CHK$EXPR$TYPE: PROC BYTE;
2907:     IF (EXPRESS$STK(SP) = EXPRESS$STK(MP)) AND EXPRESS$STK(SP) <> 0H
2908:       THEN RETURN TRUE;
2909:     IF EXPRESS$STK(SP) = 1H THEN
2910:     DO;
2911:       IF EXPRESS$STK(MP) = 3H THEN
2912:       DO;
2913:         CALL GENERATE(CNVI); /* CONVERT INT TO BCD */
2914:         EXPRESS$STK(SP)  =3H;
2915:         RETURN TRUE;
2916:       END;
```

```
2917:       ELSE RETURN FALSE;
2918:     END;
2919:     IF EXPRESS$STK(SP)  =3H THEN
2920:     DO;
2921:       IF EXPRESS$STK(MP) = 1H THEN
2922:       DO;
2923:         CALL GENERATE(CN2I); /* CONVERT SECOND INT TO BCD */
2924:         EXPRESS$STK(MP) = 3H;
2925:         RETURN TRUE;
2926:       END;
2927:       ELSE RETURN FALSE;
2928:     END;
2929:     IF EXPRESS$STK(SP) = 0H THEN
2930:     DO;
2931:       IF EXPRESS$STK(MP) <> 0H THEN
2932:         RETURN FALSE;
2933:       ELSE DO;
2934:         IF BASE$LOC(SP)=BASE$LOC(MP) THEN
2935:           RETURN TRUE;
2936:       END;
2937:     END;
2938:     RETURN FALSE;
2939: END CHK$EXPR$TYPE;
2940:
2941:
2942:   /**********************************************
2943:    *  COPY$STACKS - THIS PROCEDURE DUPLICATES THE *
2944:    *   STACK VALUES STORED AT ONE POINTER LOCATION*
2945:    *   AT ANOTHER SPECIFIED POINTER LOCATION.     *
2946:    **********************************************/
2947:
2948: COPY$STACKS: PROC(A, B);
2949:     DCL (A, B) BYTE;
2950:     TYPE$STACK(A) = TYPE$STACK(B);
2951:     PRT$ADDR(A) = PRT$ADDR(B);
2952:     EXPRESS$STK(A) = EXPRESS$STK(B);
```

```
2953:      FORM$FIELD(A) = FORM$FIELD(B);
2954:      BASE$LOC(A) = BASE$LOC(B);
2955: END COPY$STACKS;
2956:
2957:
2958:    /*****************************************
2959:     *  GENERATE$BUILT$IN - THIS PROCEDURE   *
2960:     *   GENERATES CODE FOR THE BUILT-IN     *
2961:     *   FUNCTION.                           *
2962:     *****************************************/
2963:
2964: GEN$BUILT$IN: PROC;
2965:      APTRADDR = PARMNUMLOC(MP) - 2;
2966:      IF (BYTEPTR = 13H) OR (BYTEPTR = 0F3H) THEN
2967:        CALL COPY$STACKS(MP, SP-1);
2968:      ELSE EXPRESS$STK(MP) = BYTEPTR;
2969:      /*  GENERATE THE NUMONIC CODE FOR THE BUILT IN FUNCTION  */
2970:      APTRADDR = APTRADDR - 1;
2971:      DO CASE BYTEPTR;
2972:        CALL GENERATE(ABS);
2973:        CALL GENERATE(SQR);
2974:        CALL GENERATE(SIN);
2975:        CALL GENERATE(COS);
2976:        CALL GENERATE(ARCTN);
2977:        CALL GENERATE(EXP);
2978:        CALL GENERATE(LN);
2979:        CALL GENERATE(SQRT);
2980:        CALL GENERATE(ODD);
2981:        CALL GENERATE(EOLN);
2982:        CALL GENERATE(EXF);
2983:        CALL GENERATE(TRUNC);
2984:        CALL GENERATE(ROUND);
2985:        CALL GENERATE(ORD);
2986:        CALL GENERATE(CHR);
2987:        CALL GENERATE(SUCC);
2988:        CALL GENERATE(PRED);
```

```
2989:     END;     /* OF CASE */
2990: END GEN$BUILT$IN;
2991:
2992:
2993:   /*****************************************/
2994:   /*  WRITE$STRING - THIS PROCEDURE WRITES */
2995:   /*    A STRING TO THE INTERMED. CODE     */
2996:   /*****************************************/
2997:
2998: WRITE$STRING: PROC(NUMB);
2999:     DCL NUMB BYTE;
3000:     CALL GENERATE(WRTS);
3001:     CALL GENERATE(NUMB);
3002: END WRITE$STRING;
3003:
3004:
3005:   /*****************************************/
3006:   /*  WRITE$VARIABLE - THIS PROCEDURE WILL */
3007:   /*    WRITE A VARIABLE TO THE CONSOLE VIA */
3008:   /*    THE INTERMED. CODE.                */
3009:   /*****************************************/
3010:
3011: WRITE$VAR: PROC(NUMB);
3012:     DCL NUMB BYTE;  /* NUMBER OF WRITE PARAMS */
3013:     IF NOT READPARMS THEN
3014:       DO CASE EXPRESS$STK(MP);
3015:         /* ORD TYPE */
3016:         CALL GENERATE(WRT);
3017:         /* INTEGER TYPE */
3018:         CALL GENERATE(WRTI);
3019:         /* CHAR TYPE */
3020:         CALL GENERATE(WRTI);
3021:         /* REAL TYPE */
3022:         CALL GENERATE(WRTB);
3023:         /* STRING TYPE */
3024:         DO;
```

```
3025:            CALL WRITE$STRING(NUMB);
3026:            RETURN;
3027:          END;
3028:          /* BOOLEAN TYPE */
3029:          CALL GENERATE(WRTI);
3030:      END;    /* CASE EXPRESS$STK(MP) */
3031:      CALL GENERATE(NUMB);
3032: END WRITE$VAR;
3033:
3034:    /*************************************************/
3035:    /*  READ$VARIABLE - THIS PROCEDURE GENERATES */
3036:    /*   THE INTERMEDIATE CODE TO READ A VARIABLE*/
3037:    /*   FROM THE CONSOLE.                       */
3038:    /*************************************************/
3039:
3040: READ$VAR: PROC;
3041:      IF (TYPE$STACK(SP) < 09H) OR (TYPE$STACK(SP) > 0BH) THEN
3042:        CALL ERROR('IR');
3043:      ELSE DO CASE (TYPE$STACK(SP) - 9);
3044:        CALL GENERATE(RDVI);
3045:        CALL GENERATE(RDVB);
3046:        CALL GENERATE(RDV);
3047:      END;    /* CASE (TYPE$STACK(SP) - 9) */
3048: END READ$VAR;
3049:
3050:
3051:    /*************************************************
3052:     *  B$I$PROCEDURE - THIS PROCEDURE IS CALLED    *
3053:     *   UPON RECOGNITION OF A BUILT-IN PROCEDURE   *
3054:     *   STATEMENT.                                 *
3055:     *************************************************/
3056:
3057: B$I$PROCEDURE: PROC;
3058:      BASE = BASELOC(MP);
3059:      CALL SET$PAST$PN(7);
3060:      IF BYTEPTR < 22 THEN    /* FILE HANDLING PROCEDURE */
```

```
3061:        DO CASE (BYTEPTR - 17);
3062:          CALL GENERATE(PUT);
3063:          CALL GENERATE(GET);
3064:          CALL GENERATE(RESET);
3065:          CALL GENERATE(REWRT);
3066:          CALL GENERATE(PAGE);
3067:          CALL GENERATE(NEW);
3068:        END;    /* OF CASE (BYTEPTR - 17) */
3069:      ELSE DO CASE (BYTEPTR - 22); /* VARIABLE NUMBER OF PARAMETERS */
3070:          NEWSTMT = FALSE;
3071:          DISPOSE$STMT = FALSE;
3072:          READ$STMT = FALSE;
3073:          READ$STMT = FALSE;
3074:          WRITE$STMT = FALSE;
3075:          DO;
3076:            WRITE$STMT = FALSE;
3077:            CALL GENERATE(DUMP);
3078:          END;
3079:        END;      /* OF CASE (BYTEPTR - 22) */
3080: END B$I$PROCEDURE;
3081:
3082:
3083:    /*****************************************************
3084:     * BREAK$LINKS - THIS PROCEDURE REMOVES THE          *
3085:     *  SYMBOL TABLE LOCATIONS FROM THE HASH TABLE       *
3086:     *  FOR THOSE IDENTIFIERS THAT WERE LOCAL TO         *
3087:     *  THE CURRENT SCOPE; AND THE SCOPE POINTER IS      *
3088:     *  DECRIMENTED BY ONE.                              *
3089:     *****************************************************/
3090:
3091: BREAK$LINKS: PROC;
3092:      DO WHILE SBTBLSCOPE > SCOPE(SCOPE$NUM - 1);
3093:        BASE = SBTBLSCOPE;
3094:        CALL SETADDRPTR(4);
3095:        IF ((BYTEPTR AND FORMMASK) = 7H) THEN
3096:        DO;
```

```
3097:          CALL SETADDRPTR(2);
3098:          SBTBLSCOPE, BASE = ADDRPTR;
3099:        END;
3100:        ELSE DO;
3101:          CALL SETADDRPTR(5);
3102:          SYMHASH = BYTEPTR;
3103:          CALL SETADDRPTR(0);
3104:          HASHTABLE(SYMHASH) = ADDRPTR;
3105:          CALL SETADDRPTR(2);
3106:          SBTBLSCOPE, BASE = ADDRPTR;
3107:        END;
3108:      END;
3109:      SBTBLSCOPE = SCOPE(SCOPE$NUM - 1);
3110: END BREAK$LINKS;
3111:
3112:
3113:    /*************************************************
3114:     *  SCOPE$BRANCH - THIS PROCEDURE GENERATES THE *
3115:     *   INTERMEDIATE CODE THAT PERMITS BRANCHING   *
3116:     *   AROUND ANY CODE GENERATED FOR SUBROUTINES. *
3117:     *************************************************/
3118:
3119: SCOPE$BRANCH: PROC;
3120:      IF SCOPE$NUM > 1 THEN
3121:      DO;
3122:        APTRADDR = PARAMNUMLOC + 7;
3123:        CALL GEN$ADDR(BRL,(ADDRPTR+1));
3124:        CALL GEN$ADDR(LBL,ADDRPTR);
3125:      END;
3126: END SCOPE$BRANCH;
3127:
3128:
3129:    /*************************************************
3130:     *  LABEL$MAKER - THIS PROCEDURE ENTERS ALL     *
3131:     *   LABELS IN THE SYMBOL$TABLE.                *
3132:     *************************************************/
```

```
3133: LABEL$MAKER: PROC;
3134:     IF TYPENUM = INTEGER$TYPE THEN
3135:     DO;
3136:       CALL ENTER$VAR$ID(0,SP,LABL$ENTRY);
3137:       CALL ENTER$LABEL;
3138:     END;
3139: END LABEL$MAKER;
3140:
3141:
3142:   /**************************************************
3143:    *  USER$TYPE - THIS PROCEDURE PERMITS THE       *
3144:    *   PLACEMENT OF USER DEFINED TYPES IN THE      *
3145:    *   SYMBOL TABLE.                               *
3146:    **************************************************/
3147:
3148: USER$TYP: PROC(A);
3149:     DCL A BYTE;
3150:     TYPE$LOCT=SBTBL;
3151:     IF LOOKUP$ONLY(SP) THEN
3152:       CALL ERROR('DT');
3153:     CALL ENTER$VAR$ID(0,SP,TYPE$DCLE);
3154:     IF NOT PRESENT THEN
3155:     DO;
3156:       CALL LIMITS(3);
3157:       APTRADDR=SBTBL;
3158:       BYTEPTR=A;
3159:       APTRADDR=APTRADDR+1;
3160:       ADDRPTR=PARENT$TYPE;
3161:       SBTBL=SBTBL+3;
3162:     END;
3163: END USER$TYP;
3164:
3165:
3166:   /**************************************************
3167:    *  SETSAVE$BLOCK - THIS     PROCEDURE IS       *
3168:    *   CALLED UPON DETERMINATION OF A SUBROUTINE  *
```

```
3169:     *    BLOCK.  IT INCRIMENTS THE PRT BY ONE LOCA- *
3170:     *    TION TO PERMIT THE INSERTION OF THE SBP==  *
3171:     *    AND THIS ALLOWS FOR RECURSIVE CALLS.       *
3172:     ************************************************/
3173:
3174: SETSAVE$BLOCK: PROC;
3175:       LAST$SBTBL$ID = SBTBL;
3176:       IF SCOPE$NUM > 1 THEN
3177:       DO;
3178:         BASE = PARMNUMLOC(SP - 5);
3179:         CALL SET$PAST$PN(12);
3180:         ADDRPTR = ALLOC$ADDR;      /*   SBP   */
3181:         ALLOC$ADDR = ALLOC$ADDR + 2;
3182:         CALL SET$PAST$PN(7);
3183:         TEMPADDR = BYTEPTR AND 0FFH;
3184:         CALL GEN$ADDR(LDII,TEMPADDR);
3185:         CALL SET$PAST$PN(10);
3186:         CALL GEN$ADDR(LITA,ADDRPTR);
3187:         CALL SET$PAST$PN(12);
3188:         CALL GEN$ADDR(LITA,ADDRPTR);
3189:         CALL GENERATE(SAVP);
3190:       END;
3191: END SETSAVE$BLOCK;
3192:
3193:
3194:   /************************************************
3195:     *  HEAD$AND$BLOCK - UPON RECOGNITION OF A      *
3196:     *    SUBROUTINE'S HEADING AND BLOCK, THIS      *
3197:     *    PROCEDURE IS CALLED TO GENERATE REQUIRED  *
3198:     *    CODE FOR UNSAVING THE SUBROUTINE'S        *
3199:     *    PARAMETERS IN THE EVENT OF RECURSIVE CALLS.*
3200:     ************************************************/
3201:
3202: HEAD$N$BLK: PROC;
3203:       BASE = PARMNUMLOC(MP);
3204:       CALL SET$PAST$PN(12);
```

```
3205:      CALL GEN$ADDR(LITA,ADDRPTR);
3206:      CALL SET$PAST$PN(10);
3207:      CALL GEN$ADDR(LITA,ADDRPTR);
3208:      CALL GENERATE(UNSP);
3209:      CALL BREAK$LINKS;
3210:      BASE = PARMNUMLOC(MP);
3211:      SCOPF$NUM=SCOPE$NUM-1;
3212:      CALL GENERATE(RTN);
3213:      CALL SET$PAST$PN(14);
3214:      CALL GEN$ADDR(LBL,(ADDRPTR+1));
3215:      TEMPADDR = 00H;
3216:      CALL GEN$ADDR(LDII,TEMPADDR);
3217:      CALL SET$PAST$PN(12);
3218:      CALL GEN$ADDR(LITA,ADDRPTR);
3219:      CALL GENERATE(STDI);
3220: END HEAD$N$BLK;
3221:
3222:
3223:    /*************************************************
3224:     *    FORWARD$SUBROUTINE - IN THE EVENT OF A     *
3225:     *    FORWARD DEFINED SUBROUTINE, THE ALLOCATED  *
3226:     *    SPACES IN THE PRT FOR THE ROUTINE AND ITS  *
3227:     *    ASSOCIATED PARAMETERS ARE DE-ALLOCATED AND *
3228:     *    WILL BE REALLOCATED AT THE POINT OF THE    *
3229:     *    SUBROUTINE'S DEFINITION.                   *
3230:     *************************************************/
3231:
3232: FWD$SUBRTN: PROC;
3233:      SCOPE$NUM = SCOPE$NUM - 1;
3234:      APTRADDR = PARAMNUMLOC + 3;
3235:      ALLOC$ADDR = ADDRPTR;
3236: END FWD$SUBRTN;
3237:
3238:
3239:    /*************************************************
3240:     *  GOT$PARAMETERS - THIS PROCEDURE IS CALLED    *
```

```
3241:     *   ONCE ALL A SUBROUTINE'S PARAMETERS HAVE    *
3242:     *   BEEN RECOGNIZED AND ENTERED IN THE SYMBOL  *
3243:     *   TABLE.  THE NUMBER OF PARAMETERS AND THEIR *
3244:     *   ASSOCIATED TYPE ARE THEN STORED IN THE     *
3245:     *   SYMBOL TABLE.                              *
3246:     **********************************************/
3247:
3248: GOT$PARAMS: PROC;
3249:     APTRADDR = PARAMNUMLOC;
3250:     BYTEPTR = PARAMNUM;
3251:     CALL ENTR$PRM$TYP;
3252: END GOT$PARAMS;
3253:
3254:
3255:   /**********************************************
3256:     *  SET$OP$TYPE - THIS PROCEDURE IS CALLED TO  *
3257:     *   LOAD THE TYPE OF OPERATOR USED IN AN EX-  *
3258:     *   PRESSION.                                 *
3259:     **********************************************/
3260:
3261: SET$OP$TYPE: PROC(A);
3262:     DCL A BYTE;
3263:     TYPE$STACK(MP)=A;
3264: END SET$OP$TYPE;
3265:
3266:
3267:   /**********************************************
3268:     *  CALL$A$PROCEDURE - THIS PROCEDURE IS CALLED *
3269:     *   TO GENERATE INTERMEDIATE CODE UPON        *
3270:     *   INVOKING A SUBROUTINE.  THE NUMBER OF     *
3271:     *   PARAMETERS REQUIRED IS ALSO CHECKED.      *
3272:     **********************************************/
3273:
3274: CALL$A$PROC: PROC(A);
3275:     DCL A BYTE;   /* TRUE OR FALSE */
3276:     READPARMS = FALSE;
```

```
3
 277:      IF A THEN           /* THE SUBROUTINE HAS PARAMETERS */
3278:      DO;
3279:        IF PARMNUM(MP) <> PARMNUM(SP-1) THEN
3280:          CALL ERROR('PN');
3281:      END;
3282:      IF SHR(FORM$FIELD(MP),3) THEN
3283:      DO;
3284:        IF FORM$FIELD(MP) = ØDH THEN
3285:          CALL GEN$BUILT$IN;
3286:        ELSE CALL B$I$PROCEDURE;
3287:      END;
3288:      ELSE DO;
3289:        IF FORM$FIELD(MP) = FUNC$ENTRY THEN
3290:          CALL LOAD$VARI(MP);
3291:        ELSE DO;
3292:          CALL GEN$ADDR(PRO,LABELSTACK(MP));
3293:          CALL GENERATE(DEL);
3294:        END;
3295:      END;
3296: END CALL$A$PROC;
3297:
3298:
3299:    /*************************************************
3300:     *  GOT$FUNCTION$TYPE - THIS PROCEDURE ENTERS    *
3301:     *   THE TYPE OF THE FUNCTION INTO THE SYMBOL    *
3302:     *   TABLE AND ALLOCATES A POSITION IN THE PRT   *
3303:     *   FOR THE FUNCTION VALUE TO BE STORED IN.     *
3304:     *************************************************/
3305:
3306: GOT$FUNC$TYPE: PROC;
3307:      BASE=PARMNUMLOC(MP);
3308:      CALL SET$PAST$PN(16);
3309:      BYTEPTR=SHL(ALOCBASICTYP,3) OR VAR$ENTRY;
3310:      CALL SET$PAST$PN(10);
3311:      ALLOC$ADDR = ADDRPTR;
3312:      ALLOC$ADDR = ALLOC$ADDR + ALLC$QTY;
```

```
3313: END GOT$FUNC$TYPE;
3314:
3315:
3316:   /*************************************************
3317:    *  END$PROGRAM - THIS PROCEDRE IS CALLED UPON   *
3318:    *   RECOGNITION OF THE END OF A PROGRAM.  IT    *
3319:    *   PRINTS OUT THE ERROR COUNT, CLOSES THE      *
3320:    *   INTERMEDIATE FILE, WRITES THE SYMBOL TABLE  *
3321:    *   FILE, AND INFORMS THE PROGRAMMER OF PROGRAM *
3322:    *   COMPILATION.                                *
3323:    *************************************************/
3324:
3325: END$PROGRAM: PROC;
3326:     CALL PRINT$ERROR;
3327:     CALL PRINTCHAR(' ');
3328:     CALL CRLF;
3329:     IF NOT (ERRORCOUNT > 0) THEN
3330:     DO;
3331:       CALL GEN$ADDR(ALL,ALLOC$ADDR);
3332:       CALL GENERATE(ENDP);
3333:     END;
3334:     CALL WRIT$INT$FILE;
3335:     CALL MOVE$SBTBL;
3336:     CALL CLOSE$INT$FIL;
3337:     CALL PRINT(.' COMPILATION COMPLETE.$');
3338:     CALL MON3;
3339: END END$PROGRAM;
3340:
3341:
3342:   /*************************************************
3343:    *  ARRAY$DECLARE - THIS PROCEDURE DETERMINES    *
3344:    *   AND STORES SYMBOLTABLE INFO ON ARRAYS.      *
3345:    *   THIS PROCEDURE FAILS TO MAKE USE OF THE     *
3346:    *   PARALLEL PARSE STACKS.                      *
3347:    *************************************************/
3348:
```

```
3349: ARAY$DECLARE: PROC;
3350:       IF ARRY$PTR = -1 THEN ARRY$PTR=0;
3351:       CALL ENTR$CPLX$TYP(17H);
3352:       ARY$DM$ADR$PTR=ARY$DM$ADR$PTR-NUM$ARRY$DIM(ARRY$PTR);
3353:       ARRY$BASE=BASE;
3354:       CALL LIMITS((NUM$ARRY$DIM(ARRY$PTR)*2)+3);
3355:       CALL SETADDRPTR(5);
3356:       BYTEPTR=NUM$ARRY$DIM(ARRY$PTR);
3357:       CALL SETADDRPTR(6);
3358:       ADDR$PTR=TYPE$LOCT;
3359:       CALL ALLC$OFFSET(TYPE$LOCT);
3360:       BASE=ARRY$BASE;
3361:       CALL SETADDRPTR(8);
3362:       ADDRPTR=ARRY$QTY(ARRY$PTR)*ALLC$QTY;
3363:       CALL SETADDRPTR(10);
3364:       BYTEPTR=ALOCBASICTYP;
3365:       CALL SETADDRPTR(11);
3366:       ADDRPTR = 00H;
3367:       SUBR$FORM = NUM$ARRY$DIM(ARRY$PTR) - 1;
3368:       DO WHILE SUBR$FORM < 255;
3369:         CALL SETADDRPTR(13+(2*SUBR$FORM));
3370:         ADDRPTR = ARRY$DIMEN(ARY$DM$ADR$PTR+SUBR$FORM+1);
3371:         IF SUBR$FORM = (NUM$ARRY$DIM(ARRY$PTR) - 1) THEN
3372:         DO;
3373:           APTRADDR = ADDRPTR + 7;
3374:           TEMPADDR1 = ADDRPTR;
3375:           APTRADDR = APTRADDR + 6;
3376:           ADDRPTR = 1;
3377:         END;
3378:         ELSE DO;
3379:           APTRADDR = (ARRY$DIMEN(ARY$DM$ADR$PTR+SUBR$FORM+2)+11);
3380:           TEMPADDR = ADDRPTR;
3381:           APTRADDR = APTRADDR + 2;
3382:           TEMPADDR = TEMPADDR * ADDRPTR;
3383:           APTRADDR = ARRY$DIMEN(ARY$DM$ADR$PTR+SUBR$FORM+1)+7;
3384:           TEMPADDR1 = ADDRPTR;
```

```
3385:           APTRADDR = APTRADDR + 6;
3386:           ADDRPTR = TEMPADDR;
3387:         END;
3388:         TEMPADDR1 = ADDRPTR * TEMPADDR1;
3389:         CALL SETADDRPTR(11);
3390:         ADDRPTR = TEMPADDR1 + ADDRPTR;
3391:         SUBR$FORM = SUBR$FORM - 1;
3392:       END;
3393:       TYPE$LOCT=BASE;
3394:       SBTBL=SBTBL+((NUM$ARRY$DIM(ARRY$PTR)*2)+8);
3395:     ARRY$PTR=ARRY$PTR-1;
3396: END ARAY$DECLARE;
3397:
3398:
3399:   /*************************************************
3400:    *  FIND$RELOP - THIS PROCEDURE DETERMINES       *
3401:    *   WHAT MNEUMONIC SHOULD BE GENERATED FOR ANY  *
3402:    *   RELATIONAL OPERATOR.                        *
3403:    *************************************************/
3404:
3405: FIND$RELOP: PROC;
3406:     DCL A BYTE;
3407:     DO CASE (TYPE$STACK(MPP1)-8);
3408:       A = EQLI;
3409:       A = NEQI;
3410:       A = LEQI;
3411:       A = GEQI;
3412:       A = LSSI;
3413:       A = GRTI;
3414:       IF EXPRESS$STK(SP) <> ORD$TYPE THEN CALL ERROR('CE');
3415:       ELSE A = XIN;
3416:     END; /* CASE (TYPE$STACK(MPP1)-8) */
3417:     DO CASE EXPRESS$STK(SP);
3418:       /* ORD TYPE */
3419:       IF (A = LSSI) OR (A = GRTI) THEN CALL ERROR('CE');
3420:       ELSE IF A <> XIN THEN A = A + 19;
```

```
3421:         /* INTEGER TYPE */
3422:         ;    /* NO OFFSET REQUIRED */
3423:         /* CHAR TYPE */
3424:         ;    /* NO OFFSET REQUIRED */
3425:         /* REAL TYPE */
3426:         A = A + 7;
3427:         /* STRING TYPE */
3428:         A = A + 13;
3429:         /* BOOLEAN TYPE */
3430:         ;    /* NO OFFSET REQUIRED */
3431:       END; /* OF CASE EXPRESS$STK(SP) */
3432:       CALL GENERATE(A);
3433:       EXPRESS$STK(MP) = BOOLEAN$TYPE;
3434: END FIND$RELOP;
3435:
3436:
3437: /*****************************************************************************/
3438: /*****************************************************************************/
3439:
3440:
3441:       IF LISTPROD THEN
3442:         CALL PRINT$PROD;
3443:
3444:       DO CASE PRODUCTION;
3445:
3446:
3447: /*****************************************************************************/
3448: /*****************************************************************************/
3449: /***                    P R O D U C T I O N S                              ***/
3450: /*****************************************************************************/
3451: /*****************************************************************************/
3452: /***                THE FOLLOWING IS THE INPUT GRAMMAR                     ***/
3453: /*****************************************************************************/
3454: /*****************************************************************************/
3455:
3456:
```

```
3457: /*      CASE 0 NOT USED    */  ;
3458:
3459:  /*      1   <PROGRAM> ::= <PROGRAM HEADING> <BLOCK> . _^_          */
3460:     CALL END$PROGRAM;
3461:  /*      2                 ^ <PROCEDURE HEADING> <BLOCK> . _^_     */
3462:     CALL END$PROGRAM;
3463:  /*      3                 ^ <FUNCTION HEADING> <BLOCK> . _^_      */
3464:     CALL END$PROGRAM;
3465:
3466:  /*      4   <PROGRAM HEADING> ::= PROGRAM <PROG IDENT> (          */
3467:  /*      4                         <XFILE IDENT> ) ;               */
3468:     DO;
3469:       SCOPE$NUM = 0;
3470:       SCOPE(SCOPE$NUM) = SBTBL;
3471:       SCOPE$NUM = 1;
3472:     END;
3473:
3474:  /*      5   <XFILE IDENT> ::= <FILE IDENT>                        */
3475:      ;
3476:  /*      6                   ^ <XFILE IDENT> , <FILE IDENT>        */
3477:      ;
3478:
3479:  /*      7   <PROG IDENT> ::= <IDENTIFIER>                         */
3480:      ;
3481:
3482:  /*      8   <FILE IDENT> ::= <IDENTIFIER>                         */
3483:     CALL ENTER$VAR$ID(16,SP,FILE$ENTRY);
3484:
3485:  /*      9   <BLOCK> ::= <LDP> <CDP> <TDP> <VDP> <P&FDP> <STMTP>   */
3486:      ;
3487:
3488:  /*     10   <LDP> ::=                                             */
3489:     CALL SCOPE$BRANCH;
3490:  /*     11             ^ LABEL <LABEL STRING> ;                   */
3491:     CALL SCOPE$BRANCH;
3492:
```

```
3493:  /*     12   <LABEL STRING> ::= <LABEL>                                  */
3494:     CALL LABEL$MAKER;
3495:  /*     13                    ^ <LABEL STRING> , <LABEL>                 */
3496:     CALL LABEL$MAKER;
3497:
3498:  /*     14   <LABEL> ::= <NUMBER>                                        */
3499:     IF TYPENUM <> INTEGER$TYPE THEN
3500:       CALL ERROR('LS');
3501:
3502:  /*     15   <CDP> ::=                                                   */
3503:     ;
3504:  /*     16           ^ CONST <CONST DEF> ;                               */
3505:     ;
3506:
3507:  /*     17   <CONST DEF> ::= <IDENT CONST DEF>                           */
3508:     ;
3509:  /*     18                 ^ <CONST DEF> ; <IDENT CONST DEF>             */
3510:     ;
3511:
3512:  /*     19   <IDENT CONST DEF> ::= <IDENT CONST> = <CONSTANT>            */
3513:     CALL ENTR$CONS$NTRY;
3514:
3515:  /*     20   <IDENT CONST> ::= <IDENTIFIER>                              */
3516:     DO;
3517:       IF LOOKUP$ONLY(SP) THEN
3518:         CALL ERROR('DC');
3519:       CALL ENTER$VAR$ID(0,SP,CONS$ENTRY);
3520:     END;
3521:
3522:  /*     21   <CONSTANT> ::= <NUMBER>                                     */
3523:     DO;
3524:       CALL CONVRT$CONST(POS);
3525:       EXPRESS$STK(MP)=CONS$NUM$TYPE;
3526:       VECPTR=VECPTR+1;
3527:     END;
3528:  /*     22                ^ <SIGN> <NUMBER>                             */
```

```
3529:       DO;
3530:         IF SIGNTYPE=NEG THEN
3531:           CALL CONVRT$CONST(NEG);
3532:         ELSE CALL CONVRT$CONST(POS);
3533:         EXPRESS$STK(MP)=CONS$NUM$TYPE;
3534:         VECPTR=VECPTR+1;
3535:         SIGN$FLAG = FALSE;
3536:       END;
3537:  /*     23                    ^ <CONSTANT IDENT>                          */
3538:       DO;
3539:         EXPRESS$STK(MP)=CONS$IDENT$TYPE;
3540:         VECPTR=VECPTR+1;
3541:         CALL STORE$CONST;
3542:       END;
3543:  /*     24                    ^ <SIGN> <CONSTANT IDENT>                   */
3544:       DO;
3545:         IF SIGNTYPE=NEG THEN
3546:           EXPRESS$STK(MP)=CONS$SIDENT$TYPE;
3547:         ELSE EXPRESS$STK(MP)=CONS$IDENT$TYPE;
3548:         VECPTR=VECPTR+1;
3549:         CALL STORE$CONST;
3550:         SIGN$FLAG = FALSE;
3551:       END;
3552:  /*     25                    ^ <STRING>                                  */
3553:       DO;
3554:         EXPRESS$STK(MP)=CONS$STR$TYPE;
3555:         VECPTR=VECPTR+1;
3556:         CALL STORE$CONST;
3557:       END;
3558:
3559:  /*     26   <CONSTANT IDENT> ::= <IDENTIFIER>                            */
3560:       ;
3561:
3562:  /*     27   <SIGN> ::= +                                                 */
3563:      DO;
3564:        SIGN$TYPE = POS;
```

```
3565:       SIGN$FLAG = TRUE;
3566:     END;
3567:  /*     28              ^ -                                          */
3568:     DO;
3569:       SIGN$TYPE = NEG;
3570:       SIGN$FLAG = TRUE;
3571:     END;
3572:
3573:  /*     29   <TDP> ::=                                               */
3574:      CASE$STMT=FALSE;
3575:  /*     30             ^ TYPE <TYPE DEF STRING> ;                    */
3576:      CASE$STMT=FALSE;
3577:
3578:  /*     31   <TYPE DEF STRING> ::= <TYPE ID>                         */
3579:      ;
3580:  /*     32                        ^ <TYPE DEF STRING> ; <TYPE ID>    */
3581:      ;
3582:
3583:  /*     33   <TYPE ID> ::= <TYPE IDS> = <TYPE>                       */
3584:      DO;
3585:        APTRADDR=TYPE$ADDR;
3586:        ADDRPTR=TYPE$LOCT;
3587:      END;
3588:
3589:  /*     34   <TYPE IDS> ::= <IDENTIFIER>                             */
3590:      DO;
3591:        IF LOOKUP$ONLY(SP) THEN
3592:          CALL ERROR('DT');
3593:        PARENT$TYPE=SBTBL;
3594:        CALL ENTER$VAR$ID(78H,SP,TYPE$ENTRY);
3595:        IF NOT PRESENT THEN
3596:        DO;
3597:          CALL LIMITS(2);
3598:          TYPE$ADDR=SBTBL;
3599:          SBTBL=SBTBL+2;
3600:        END;
```

```
3601:       END;
3602:
3603:   /*      35   <TYPE> ::= <SIMPLE TYPE>                                  */
3604:       ;
3605:   /*      36            ^ <STRUCTURED TYPE>                              */
3606:       ;
3607:   /*      37            ^ <POINTER TYPE>                                 */
3608:       ;
3609:
3610:   /*      38   <SIMPLE TYPE> ::= <TYPE IDENT>                            */
3611:       ;
3612:   /*      39                   ^ ( <TIDENT STRING> )                     */
3613:       ;
3614:   /*      40                   ^ <CONSTANT> .. <CONSTANT>                */
3615:       CALL ENTR$SUB$NTRY;
3616:
3617:   /*      41   <TYPE IDENT> ::= <IDENTIFIER>                             */
3618:       IF LOOKUP$PN$ID(SP,TYPE$ENTRY) THEN
3619:         TYPE$LOCT=LOOKUP$ADDR;
3620:       ELSE DO;
3621:         CALL ERROR('TI');
3622:         TYPE$LOCT=.BUILT$IN$TBL;  /* INTEGER DEFAULT */
3623:       END;
3624:
3625:   /*      42   <TIDENT STRING> ::= <IDENTIFIER>                          */
3626:       DO;
3627:         TYPE$ORD$NUM=0;
3628:         CALL USER$TYP(TYPE$ORD$NUM);
3629:       END;
3630:   /*      43                   ^ <TIDENT STRING> , <IDENTIFIER>          */
3631:       DO;
3632:         TYPE$ORD$NUM=TYPE$ORD$NUM+1;
3633:         CALL USER$TYP(TYPE$ORD$NUM);
3634:       END;
3635:
3636:   /*      44   <STRUCTURED TYPE> ::= <UNPACKED STRUCTURED TYPE>          */
```

```
3637:       ;
3638:
3639:   /*      45                          ^ PACKED                          */
3640:   /*      45                            <UNPACKED STRUCTURED TYPE>      */
3641:       ;
3642:
3643:   /*      46    <UNPACKED STRUCTURED TYPE> ::= <ARRAY TYPE>             */
3644:       ;
3645:   /*      47                                   ^ <RECORD TYPE>          */
3646:       ;
3647:   /*      48                                   ^ <SET TYPE>             */
3648:       ;
3649:   /*      49                                   ^ <FILE TYPE>            */
3650:       ;
3651:
3652:   /*      50    <ARRAY TYPE> ::= ARRAY [ <INDEX TYPE STRING> ] OF       */
3653:   /*      50                     <COMPONENT TYPE>                       */
3654:      CALL ARAY$DECLARE;
3655:
3656:   /*      51    <INDEX TYPE STRING> ::= <INDEX TYPE>                    */
3657:       DO;
3658:         IF ARRY$PTR=ARRY$NEST-1 THEN
3659:         DO;
3660:           CALL ERROR('AN');
3661:           ARY$DM$ADR$PTR=ARY$DM$ADR$PTR-NUM$ARRY$DIM(ARRY$PTR);
3662:         END;
3663:         ELSE ARRY$PTR= ARRY$PTR+1;
3664:         ARRY$DIM$PTR=0;
3665:         ARY$DM$ADR$PTR=ARY$DM$ADR$PTR+1;
3666:         ARRY$DIMEN(ARY$DM$ADR$PTR)=TYPE$LOCT;
3667:         ARRY$QTY(ARRY$PTR)=AL$NDX$OFFSET;
3668:         NUM$ARRY$DIM(ARRY$PTR)=1;
3669:       END;
3670:   /*      52                          ^ <INDEX TYPE STRING> ,           */
3671:   /*      52                            <INDEX TYPE>                    */
3672:       DO;
```

```
3673:          IF ARRY$DIM$PTR=MAX$ARRY$DIM-1 THEN
3674:            CALL ERROR('AD');
3675:          ELSE ARRY$DIM$PTR=ARRY$DIM$PTR+1;
3676:          ARY$DM$ADR$PTR=ARY$DM$ADR$PTR+1;
3677:          ARRY$DIMEN(ARY$DM$ADR$PTR)=TYPE$LOCT;
3678:          ARRY$QTY(ARRY$PTR)=ARRY$QTY(ARRY$PTR)*AL$NDX$OFFSET;
3679:          NUM$ARRY$DIM(ARRY$PTR)=NUM$ARRY$DIM(ARRY$PTR)+1;
3680:        END;
3681:
3682:  /*      53   <INDEX TYPE> ::= <SIMPLE TYPE>                          */
3683:       ;
3684:
3685:  /*      54   <COMPONENT TYPE> ::= <TYPE>                             */
3686:      ;
3687:
3688:  /*      55   <RECORD TYPE> ::= RECORD <FIELD LIST> END               */
3689:      DO;
3690:        VARIANT$PART(REC$NST)=FALSE;
3691:        BASE,TYPE$LOCT=REC$PAR$ADR(REC$NST);
3692:        IF VAR$CAS$VAL(REC$NST) <> 0 THEN
3693:          CALL ERROR('IV');
3694:        CALL SETADDRPTR(5);
3695:        ADDRPTR=FXD$OFST$BSE(REC$NST);
3696:        CALL SETADDRPTR(7);
3697:        ADDRPTR= PRV$SBT$ENTRY;
3698:        REC$NST=REC$NST-1;
3699:      END;
3700:
3701:  /*      56   <FIELD LIST> ::= <FIXED PART>                           */
3702:       ;
3703:  /*      57                  | <FIXED PART> ; <VARIANT PART>        */
3704:       ;
3705:  /*      58                  | <VARIANT PART>                       */
3706:       ;
3707:
3708:  /*      59   <FIXED PART> ::= <RECORD SECTION>                       */
```

```
3709:     ;
3710:  /*     60                    ^ <FIXED PART> ; <RECORD SECTION>          */
3711:     ;
3712:
3713:  /*     61   <RECORD SECTION> ::= <FIELD IDENT STRING> : <TYPE>          */
3714:     DO;
3715:        CALL ALLC$OFFSET(TYPE$LOCT);
3716:        /* ALOCBASICTYP AND ALLC$QTY ARE SET */
3717:        DO PTRPTR = 0 TO RECORD$PTR;
3718:          BASE = REC$ADDR(PTRPTR);
3719:          CALL SET$PAST$PN(9);
3720:          BYTEPTR = ALOCBASICTYP;
3721:          APTRADDR=APTRADDR+1;
3722:          ADDRPTR=TYPE$LOCT;
3723:          APTRADDR=APTRADDR+2;
3724:          ADDRPTR=CUR$OFST(REC$NST);
3725:          CUR$OFST(REC$NST)=CUR$OFST(REC$NST)
3726:             + ALLC$QTY;
3727:        END;
3728:        RECORD$PTR=0;
3729:        IF FXD$OFST$BSE(REC$NST) < CUR$OFST(REC$NST)
3730:        THEN FXD$OFST$BSE(REC$NST)=CUR$OFST(REC$NST);
3731:      END;
3732:  /*     62                          ^                                   */
3733:     ;
3734:
3735:  /*     63   <FIELD IDENT STRING> ::= <FIELD IDENT>                     */
3736:     ;
3737:  /*     64                              ^ <FIELD IDENT STRING> ,        */
3738:  /*     64                                <FIELD IDENT>                 */
3739:     ;
3740:
3741:  /*     65   <FIELD IDENT> ::= <IDENTIFIER>                             */
3742:     DO;
3743:       IF RECORD$PTR <> 10 THEN RECORD$PTR=RECORD$PTR+1;
3744:       ELSE CALL ERROR('RN');
```

```
3745:         REC$ADDR(RECORD$PTR)=SBTBL;
3746:         CALL ENTER$VAR$ID(58H,SP,TYPF$DCLE);
3747:         IF NOT PRESENT THEN DO;
3748:           CALL LIMITS(7);
3749:           APTRADDR=SBTBL;
3750:           ADDRPTR=REC$PAR$ADR(REC$NST);
3751:           SBTBL=SBTBL+7;
3752:         END;
3753:         IF VARIANT$PART(REC$NST) THEN
3754:         DO;
3755:           BASE=REC$ADDR(RECORD$PTR);
3756:           CALL LIMITS(2);
3757:           CALL SETADDRPTR(4);
3758:           BYTEPTR=0DFH;
3759:         END;
3760:       END;
3761:
3762:   /*      66   <VARIANT PART> ::= CASE <TAG FIELD> <TYPE IDENT> OF      */
3763:   /*      66                          <VARIANT STRING>                  */
3764:       ;
3765:   /*      67                        ^ CASE <TYPE IDENT> OF              */
3766:   /*      67                          <VARIANT STRING>                  */
3767:       ;
3768:
3769:   /*      68   <VARIANT STRING> ::= <VARIANT>                           */
3770:       ;
3771:   /*      69                          ^ <VARIANT STRING> ; <VARIANT>    */
3772:       ;
3773:
3774:   /*      70   <TAG FIELD> ::= <FIELD IDENT> :                          */
3775:       TAG$FD(REC$NST)=TRUE;
3776:
3777:   /*      71   <VARIANT> ::= <CASE LABEL LIST> : ( <FIELD LIST> )       */
3778:       ;
3779:   /*      72                       ^                                    */
3780:       ;
```

```
3781:
3782:  /*     73   <CASE LABEL LIST> ::= <CASE LABEL>                          */
3783:     DO;
3784:       LABELSTACK(SP) = LABLCOUNT;
3785:       LABLCOUNT = LABLCOUNT + 2;
3786:       CALL GEN$ADDR(KASE,ALLC$QTY);
3787:       CALL GENERATE(LOW(LABELSTACK(SP)));
3788:       CALL GENERATE(HIGH(LABELSTACK(SP)));
3789:     END;
3790:  /*     74                        ^ <CASE LABEL LIST> , <CASE LABEL>    */
3791:     DO;
3792:       CALL GEN$ADDR(KASE,ALLC$QTY);
3793:       CALL GENERATE(LOW(LABELSTACK(MP)));
3794:       CALL GENERATE(HIGH(LABELSTACK(MP)));
3795:     END;
3796:
3797:  /*     75   <CASE LABEL> ::= <CONSTANT>                                */
3798:     IF CASE$STMT THEN
3799:     DO;
3800:       CASE$STK(CASE$COUNT) = CASE$STK(CASE$COUNT) + 1;
3801:       DO CASE EXPRESS$STK(SP)  ;
3802:          /* NUMBER */
3803:          ALLC$QTY = CONVERTI(SP,POS);
3804:          /* IDENTIFIER */
3805:          DO;
3806:            IF NOT LOOK$UP$ONLY(SP) THEN CALL ERROR('DT');
3807:            ELSE DO;
3808:              BASE = LOOKUP$ADDR;
3809:              CALL SET$PAST$PN(7);
3810:              ALLC$QTY = ADDRPTR;
3811:            END;
3812:          END;
3813:          /* SIGNED IDENTIFIER */
3814:         DO;
3815:         END;
3816:          /* STRING TYPE */
```

```
3817:          ;
3818:        END;            /*   OF CASE   */
3819:      END;
3820:      ELSE
3821:      DO;
3822:       IF NOT VARIANT$PART(REC$NST) THEN
3823:        DO;
3824:          VARIANT$PART(REC$NST)=TRUE;
3825:          VAR$CAS$TP(REC$NST)=TYPE$LOCT;
3826:          VAR$CAS$VAL(REC$NST)=AL$NDX$OFFSET;
3827:          CALL ALLC$OFFSET(TYPE$LOCT);
3828:          IF TAG$FD(REC$NST) THEN
3829:           DO;
3830:             TAG$FD(REC$NST)=FALSE;
3831:             BASE=REC$ADDR(RECORD$PTR);
3832:             CALL SETADDRPTR(4);
3833:             BYTEPTR=9FH;
3834:             CALL SETADDRPTR(5);
3835:             CALL SETADDRPTR(8+BYTEPTR);
3836:             ADDRPTR=VAR$CAS$VAL(REC$NST);
3837:             APTRADDR=APTRADDR+2;
3838:             ADDRPTR=VAR$CAS$TP(REC$NST);
3839:             APTRADDR=APTRADDR+2;
3840:             ADDRPTR=CUR$OFST(REC$NST);
3841:          CUR$OFST(REC$NST)=CUR$OFST(REC$NST)+ALLC$QTY;
3842:           END;
3843:          VAR$OFST$BSE(REC$NST)=CUR$OFST(REC$NST);
3844:          FXD$OFST$BSE(REC$NST)=CUR$OFST(REC$NST);
3845:        END;
3846:       /* CALL COMPARE$CONST$VARIANT; */
3847:       /* THE ROUTINE ABOVE CHECKS THE CASE LABLE WITH THE VARIANT TYPE */
3848:
3849:       CUR$OFST(REC$NST)=VAR$OFST$BSE(REC$NST);
3850:       VECPTR=VECPTR-1;
3851:       CONST$PTR,CONST$INDX,CONST$PN$PTR=0;
3852:      END;
```

```
3853:
3854:  /*     76   <SET TYPE> ::= SET OF <BASE TYPE>                          */
3855:     CALL ENTR$STR$TYP(27H);
3856:
3857:  /*     77   <BASE TYPE> ::= <SIMPLE TYPE>                             */
3858:     ;
3859:
3860:  /*     78   <FILE TYPE> ::= FILE OF <TYPE>                            */
3861:     CALL ENTR$STR$TYP(2FH);
3862:
3863:  /*     79   <POINTER TYPE> ::= ^ <TYPE IDENT>                         */
3864:     CALL ENTR$STR$TYP(37H);
3865:
3866:  /*     80   <VDP> ::=                                                 */
3867:     SCOPE(SCOPE$NUM) = SBTBL;
3868:  /*     81             ^ VAR <VAR DECLAR STRING> ;                     */
3869:     SCOPE(SCOPE$NUM) = SBTBL;
3870:
3871:  /*     82   <VAR DECLAR STRING> ::= <VAR DECLAR>                      */
3872:     ;
3873:  /*     83                          ^ <VAR DECLAR STRING> ;            */
3874:  /*     83                            <VAR DECLAR>                     */
3875:     ;
3876:
3877:  /*     84   <VAR DECLAR> ::= <IDENT VAR STRING> : <TYPE>              */
3878:     DO;
3879:       CALL ALLOC$VARS;
3880:     END;
3881:
3882:  /*     85   <IDENT VAR STRING> ::= <IDENTIFIER>                       */
3883:     DO;
3884:       VAR$PTR = 0;
3885:       VAR$BASE(VARPTR) = SBTBL;
3886:       CALL ENTER$VAR$ID(0,SP,VAR$ENTRY);
3887:     END;
3888:  /*     86                       ^ <IDENT VAR STRING> ,                */
```

```
3889:  /*     86                                <IDENTIFIER>                    */
3890:     IF VARPTR <> 10 THEN
3891:     DO;
3892:       VAR$PTR = VAR$PTR + 1;
3893:       VAR$BASE(VAR$PTR) = SBTBL;
3894:       CALL ENTER$VAR$ID(0,SP,VAR$ENTRY);
3895:     END;
3896:     ELSE CALL ERROR('VN');
3897:
3898:  /*     87   <PROFDP> ::=                                               */
3899:     CALL SETSAVE$BLOCK;
3900:  /*     88                ^ <PORF DECLAR>                               */
3901:     CALL SETSAVE$BLOCK;
3902:
3903:  /*     89   <PORF DECLAR> ::= <PROC OR FUNCT> ;                        */
3904:    ;
3905:  /*     90                   ^ <PORF DECLAR> <PROC OR FUNCT> ;          */
3906:    ;
3907:
3908:  /*     91   <PROC OR FUNCT> ::= <PROCEDURE HEADING> <BLOCK>            */
3909:     CALL HEAD$N$BLK;
3910:  /*     92                       ^ <PROCEDURE HEADING> <DIRECTIVE>      */
3911:      ;
3912:  /*     93                       ^ <FUNCTION HEADING> <BLOCK>           */
3913:     CALL HEAD$N$BLK;
3914:  /*     94                       ^ <FUNCTION HEADING> <DIRECTIVE>       */
3915:      ;
3916:
3917:
3918:  /*     95   <DIRECTIVE> ::= <IDENTIFIER>                               */
3919:     IF NOT LOOKUP$ONLY(SP) THEN CALL ERROR('DT');
3920:     ELSE DO;
3921:       BASE = LOOKUP$ADDR;
3922:       CALL SETADDRPTR(5);
3923:       IF BYTEPTR = 21 THEN CALL FWD$SUBRTN;
3924:     END;
```

```
3925:
3926:  /*     96   <PROCEDURE HEADING> ::= <PROC ID> ;                          */
3927:       ;
3928:  /*     97                          ^ <PROC ID> (                        */
3929:  /*     97                            <FORMAL PARA SECT LIST> ) ;        */
3930:     CALL GOT$PARAMS;
3931:
3932:  /*     98   <PROC ID> ::= PROCEDURE <IDENTIFIER>                        */
3933:      DO;
3934:        PARAMNUM = 0;
3935:        CALL ENTER$SUBRTN(0,SP,PROC$ENTRY);
3936:      END;
3937:
3938:  /*     99   <FORMAL PARA SECT LIST> ::= <FORMAL PARA SECT>              */
3939:     CALL ALLOC$VARS;
3940:  /*    100                            ^ <FORMAL PARA SECT LIST> ;       */
3941:  /*    100                              <FORMAL PARA SECT>              */
3942:     CALL ALLOC$VARS;
3943:
3944:  /*    101   <FORMAL PARA SECT> ::= <PARA GROUP>                        */
3945:     ;
3946:  /*    102                         ^ VAR <PARA GROUP>                   */
3947:     DO;
3948:       TEMPBYTE = VAR$PTR;
3949:       DO VAR$PARM$PTR = 0 TO TEMPBYTE;
3950:         BASE = VARBASE (VAR$PARM$PTR);
3951:         CALL SETADDRPTR(4);
3952:         BYTEPTR = BYTEPTR OR 80H;
3953:       END;
3954:     END;
3955:  /*    103                         ^ FUNCTION <PARA GROUP>              */
3956:     DO;
3957:       TEMPBYTE = VAR$PTR;
3958:       DO VAR$PARM$PTR = 0 TO TEMPBYTE;
3959:         BASE = VARBASE (VAR$PARM$PTR);
3960:         CALL SETADDRPTR(4);
```

```
3961:          BYTEPTR = FUNC$ENTRY OR 80H;
3962:        END;
3963:     END;
3964:  /*    104                          ^ PROCEDURE <PROC IDENT LIST>        */
3965:     ;
3966:
3967:  /*    105   <PROC IDENT LIST> ::= <IDENTIFIER>                          */
3968:     DO;
3969:       VAR$PTR=0;
3970:       PARAMNUM = PARAMNUM + 1;
3971:       VAR$BASE=SBTBL;
3972:       CALL ENTER$SUBRTN(0,SP,PROC$ENTRY);
3973:     END;
3974:  /*    106                          ^ <PROC IDENT LIST> , <IDENTIFIER>   */
3975:     IF VAR$PTR <> 10 THEN
3976:     DO;
3977:       VAR$PTR=VAR$PTR+1;
3978:       PARAMNUM = PARAMNUM + 1;
3979:       VAR$BASE(VAR$PTR)=SBTBL;
3980:       CALL ENTER$SUBRTN(0,SP,PROC$ENTRY);
3981:     END;
3982:     ELSE CALL ERROR('VN');
3983:
3984:  /*    107   <PARA GROUP> ::= <PARA IDENT LIST> : <TYPE IDENT>           */
3985:     ;
3986:
3987:  /*    108   <PARA IDENT LIST> ::= <IDENTIFIER>                          */
3988:     DO;
3989:       VAR$PTR=0;
3990:       PARAMNUM = PARAMNUM + 1;
3991:       VAR$BASE=SBTBL;
3992:       CALL ENTER$VAR$ID(0,SP,VAR$ENTRY);
3993:     END;
3994:  /*    109                          ^ <PARA IDENT LIST> , <IDENTIFIER>   */
3995:     IF VAR$PTR <> 10 THEN
3996:     DO;
```

```
3997:       VAR$PTR=VAR$PTR+1;
3998:       PARAMNUM = PARAMNUM + 1;
3999:       VAR$BASE(VAR$PTR)=SBTBL;
4000:       CALL ENTER$VAR$ID(0,SP,VAR$ENTRY);
4001:     END;
4002:     ELSE CALL ERROR('VN');
4003:
4004:  /*    110   <FUNCTION HEADING> ::= <FUNCT ID> : <RESULT TYPE> ;      */
4005:     CALL GOT$FUNC$TYPE;
4006:  /*    111                          ^ <FUNCT ID> (                    */
4007:  /*    111                            <FORMAL PARA SECT LIST> ) :     */
4008:  /*    111                            <RESULT TYPE> ;                 */
4009:     DO;
4010:       CALL GOT$PARAMS;
4011:       CALL GOT$FUNC$TYPE;
4012:       CALL ALTER$PRT$LOC;
4013:     END;
4014:
4015:  /*    112   <FUNCT ID> ::= FUNCTION <IDENTIFIER>                      */
4016:      DO;
4017:       PARAMNUM = 0;
4018:       CALL ENTER$SUBRTN(0,SP,FUNC$ENTRY);
4019:      END;
4020:
4021:  /*    113   <RESULT TYPE> ::= <TYPE IDENT>                           */
4022:     CALL ALLC$OFFSET(TYPELOCT);
4023:
4024:  /*    114   <STMTP> ::= <COMPOUND STMT>                              */
4025:     ;
4026:
4027:  /*    115   <STMT> ::= <BAL STMT>                                    */
4028:     ;
4029:  /*    116            ^ <UNBAL STMT>                                  */
4030:     ;
4031:  /*    117            ^ <LABEL DEF> <STMT>                            */
4032:     ;
```

```
4033:
4034:  /*    118   <BAL STMT> ::= <IF CLAUSE> <TRUE PART> ELSE <BAL STMT>   */
4035:       CALL GEN$ADDR(LBL,(LABELSTACK(MP)+1));
4036:  /*    119                  <SIMPLE STMT>                             */
4037:     ;
4038:
4039:  /*    120   <UNBAL STMT> ::= <IF CLAUSE> <STMT>                      */
4040:     CALL GEN$ADDR(LBL,LABELSTACK(MP));
4041:  /*    121                    <IF CLAUSE> <TRUE PART> ELSE            */
4042:  /*    121                    <UNBAL STMT>                            */
4043:     CALL GEN$ADDR(LBL,(LABELSTACK(MP)+1));
4044:
4045:  /*    122   <IF CLAUSE> ::= IF <EXPRESSION> THEN                     */
4046:     DO;
4047:       LABELSTACK(MP)=LABLCOUNT;
4048:       LABLCOUNT=LABLCOUNT+2;
4049:       IF EXPRESS$STK(MPP1) = BOOLEAN$TYPE THEN
4050:       DO;
4051:         CALL GENERATE(NOTX);
4052:         CALL GEN$ADDR(BLC,LABELSTACK(MP));
4053:       END;
4054:       ELSE CALL ERROR('CE');
4055:     END;
4056:
4057:  /*    123   <TRUE PART> ::= <BAL STMT>                               */
4058:     DO;
4059:       CALL GEN$ADDR(BRL,(LABELSTACK(SP-1)+1));
4060:       CALL GEN$ADDR(LBL,LABELSTACK(SP-1));
4061:     END;
4062:
4063:  /*    124   <LABEL DEF> ::= <LABEL> :                                */
4064:     IF LOOKUP$PN$ID(MP,LABL$ENTRY) THEN
4065:     DO;
4066:       CALL SETADDRPTR(5);
4067:       CALL SETADDRPTR(6+BYTEPTR);
4068:       CALL GEN$ADDR(LBL,ADDRPTR);
```

```
4069:     END;
4070:     ELSE CALL ERROR('UL');
4071:
4072:  /*    125   <SIMPLE STMT> ::= <ASSIGNMENT STMT>                    */
4073:     ;
4074:  /*    126                    ^ <PROCEDURE STMT>                   */
4075:     ;
4076:  /*    127                    ^ <WHILE STMT>                       */
4077:     ;
4078:  /*    128                    ^ <REPEAT STMT>                      */
4079:     ;
4080:  /*    129                    ^ <FOR STMT>                         */
4081:     ;
4082:  /*    130                    ^ <CASE STMT>                        */
4083:     ;
4084:  /*    131                    ^ <WITH STMT>                        */
4085:     ;
4086:  /*    132                    ^ <GOTO STMT>                        */
4087:     ;
4088:  /*    133                    ^ <COMPOUND STMT>                    */
4089:      ;
4090:  /*    134                    ^                                    */
4091:     ;
4092:
4093:  /*    135   <ASSIGNMENT STMT> ::= <VARIABLE> := <EXPRESSION>       */
4094:        CALL ASSIGN$VARI(MP, FALSE);
4095:
4096:  /*    136   <VARIABLE> ::= <VARIABLE IDENT>                       */
4097:     ;
4098:  /*    137                  ^ <VARIABLE> ^                         */
4099:     ;
4100:  /*    138                  ^ <VARIABLE> [ <EXPRES LIST> ]         */
4101:     DO;
4102:        TYPE$STACK(MP) = (TYPE$STACK(MP) OR 40H);
4103:        BASE = BASE$LOC(MP);
4104:        CALL SET$PAST$PN(7);
```

```
4105:       CALL GEN$ADDR(LDII, ADDRPTR);
4106:       CALL GENERATE(SUB);
4107:       CALL SETADDRPTR(9);
4108:       BASE = ADDRPTR;
4109:       CALL SETADDRPTR(5);
4110:       CALL GENERATE(BYTEPTR);
4111:     END;
4112:  /*    139                ^ <VARIABLE> . <FIELD IDENT>                */
4113:     IF NOT LOOKUP$ONLY(SP) THEN CALL ERROR('DT');
4114:     ELSE DO;
4115:       BASE = LOOKUP$ADDR;
4116:       CALL SET$PAST$PN(12);
4117:       PRT$ADDR(MP) = ADDRPTR + PRT$ADDR(MP);
4118:       CALL SET$PAST$PN(9);
4119:       CALL CASEPTRPTR(BYTEPTR);
4120:       TYPE$STACK(MP), TYPE$STACK(SP) = PTRPTR;
4121:     END;
4122:
4123:  /*    140   <VARIABLE IDENT> ::= <IDENTIFIER>                         */
4124:     DO;
4125:       VARPARM = FALSE;
4126:       IF NOT LOOKUP$ONLY(SP) THEN
4127:         CALL ERROR('DT');
4128:       ELSE CALL SET$VAR$TYPE; /* LOOKUP$ADDR SET HERE */
4129:     END;
4130:
4131:  /*    141   <EXPRES LIST> ::= <EXPRESSION>                            */
4132:     ;
4133:  /*    142                   ^ <EXPRES LIST> , <EXPRESSION>            */
4134:     ;
4135:
4136:  /*    143   <EXPRESSION> ::= <SIMPLE EXPRESSION>                      */
4137:     ;
4138:  /*    144                   ^ <SIMPLE EXPRESSION>                     */
4139:  /*    144                     <RELATIONAL OPERATOR>                   */
4140:  /*    144                     <SIMPLE EXPRESSION>                     */
```

```
4141:     IF CHK$EXPR$TYPE THEN CALL FIND$RELOP;
4142:     ELSE CALL ERROR('CE');
4143:
4144:  /*    145   <RELATIONAL OPERATOR> ::= =                              */
4145:     CALL SET$OP$TYPE(08H);
4146:  /*    146                          ^ < >                             */
4147:     CALL SET$OP$TYPE(09H);
4148:  /*    147                          ^ < =                             */
4149:     CALL SET$OP$TYPE(0AH);
4150:  /*    148                          ^ > =                             */
4151:     CALL SET$OP$TYPE(0BH);
4152:  /*    149                          ^ <                               */
4153:     CALL SET$OP$TYPE(0CH);
4154:  /*    150                          ^ >                               */
4155:     CALL SET$OP$TYPE(0DH);
4156:  /*    151                          ^ IN                              */
4157:     CALL SET$OP$TYPE(0EH);
4158:
4159:  /*    152   <TERM> ::= <FACTOR>                                      */
4160:     ;
4161:  /*    153            ^ <TERM> <MULTIPLYING OPERATOR> <FACTOR>        */
4162:     DO;
4163:       IF READPARMS THEN
4164:       DO;
4165:         APTRADDR = PARMNUMLOC(MP);
4166:         IF SHR(BYTEPTR,7) THEN
4167:           CALL ERROR('NE');
4168:       END;
4169:       IF CHK$EXPR$TYPE THEN
4170:       DO;
4171:         DO CASE TYPE$STACK(MPP1);
4172:         /*0*/ IF EXPRESS$STK(SP) = 1H THEN CALL GENERATE(MULI);
4173:             ELSE IF EXPRESS$STK(SP) = 3H THEN CALL GENERATE(MULB);
4174:               ELSE IF EXPRESS$STK(SP) = 0H THEN CALL GENERATE(ISEC);
4175:                 ELSE CALL ERROR('CE');
4176:         /*1*/ IF EXPRESS$STK(SP) = 1H THEN
```

```
4177:             DO;
4178:               CALL GENERATE(CNVI); /* CONVERT 1ST INTEGER */
4179:               CALL GENERATE(CN2I); /* CONVERT 2ND INTEGER */
4180:               CALL GENERATE(DIVB);
4181:               EXPRESS$STK(MP) = UNSIGN$EXPON;
4182:             END;
4183:             ELSE IF EXPRESS$STK(SP) = 3H THEN CALL GENERATE(DIVB);
4184:             ELSE CALL ERROR('CE');
4185:           /*2*/ IF EXPRESS$STK(SP)=INTEGER$TYPE THEN CALL GENERATE(DIVI);
4186:             ELSE CALL ERROR('CE');
4187:           /*3*/ IF EXPRESS$STK(SP)=INTEGER$TYPE THEN CALL GENERATE(MODX);
4188:             ELSE CALL ERROR('CE');
4189:           /*4*/ IF EXPRESS$STK(SP)=BOOLEAN$TYPE THEN CALL GENERATE(ANDX);
4190:             ELSE CALL ERROR('CE');
4191:            END;  /*  OF CASE VAR$TYPE$STK */
4192:         END;
4193:         ELSE CALL ERROR('CE');
4194:       END;
4195:
4196:   /*    154   <MULTIPLYING OPERATOR> ::= *                              */
4197:       CALL SET$OP$TYPE(00H);
4198:   /*    155                            ^ /                              */
4199:       CALL SET$OP$TYPE(01H);
4200:   /*    156                            ^ DIV                            */
4201:       CALL SET$OP$TYPE(02H);
4202:   /*    157                            ^ MOD                            */
4203:       CALL SET$OP$TYPE(03H);
4204:   /*    158                            ^ AND                            */
4205:       CALL SET$OP$TYPE(04H);
4206:
4207:   /*    159   <SIMPLE EXPRESSION> ::= <TERM>                            */
4208:       ;
4209:   /*    160                          ^ <SIGN> <TERM>                    */
4210:       DO;
4211:         IF READPARMS THEN DO;
4212:           APTRADDR = PARMNUMLOC(SP);
```

```
4213:          IF SHR(BYTEPTR,7) THEN
4214:            CALL ERROR('NE');
4215:        END;
4216:        IF SIGNTYPE = NEG THEN
4217:        DO;
4218:          IF EXPRESS$STK(SP) = UNSIGN$EXPON THEN
4219:            CALL GENERATE(NEGE);
4220:          ELSE IF EXPRESS$STK(SP) = INTEGER$TYPE THEN
4221:            CALL GENERATE(NEGI);
4222:          ELSE CALL ERROR('UO');
4223:        END;
4224:        SIGN$FLAG = FALSE;
4225:        CALL COPY$STACKS(MP,SP);
4226:      END;
4227:  /*     161                          ^ <SIMPLE EXPRESSION>                */
4228:  /*     161                            <ADDING OPERATOR> <TERM>           */
4229:      DO;
4230:        IF READPARMS THEN DO;
4231:          APTRADDR = PARMNUMLOC(MP);
4232:          IF SHR(BYTEPTR,7) THEN
4233:            CALL ERROR('NE');
4234:        END;
4235:        IF CHK$EXPR$TYPE THEN
4236:          DO;
4237:            IF TYPE$STACK(MPP1)=5H THEN /* ARITH ADD */
4238:            DO CASE EXPRESS$STK(SP);
4239:              CALL GENERATE(UNION);   /* CASE 0 - ORD TYPE */
4240:              CALL GENERATE(ADDI);   /* CASE 1 - INTEGER */
4241:              CALL ERROR('CE'); /* CASE 2 - CHAR */
4242:              CALL GENERATE(ADDB); /* CASE 3 - REAL */
4243:              CALL ERROR('CE'); /* CASE 4 - STRING */
4244:              CALL ERROR('CE'); /* CASE 5 - BOOLEAN */
4245:            END; /* CASE */
4246:            ELSE IF TYPE$STACK(MPP1)= 6H THEN /* ARITH SUBTRC */
4247:              DO CASE EXPRESS$STK(SP);
4248:                CALL GENERATE(STDIF); /* CASE 0 - ORD TYPE */
```

```
4249:                CALL GENERATE(SUBI);
4250:                CALL ERROR('CE');
4251:                CALL GENERATE(SUBB);
4252:                CALL ERROR('CE');
4253:                CALL ERROR('CE');
4254:              END;
4255:              ELSE IF TYPE$STACK(MPP1)=7H THEN /* BOOLEAN OR */
4256:                DO;
4257:                  IF EXPRESS$STK(SP) = BOOLEAN$TYPE THEN
4258:                    CALL GENERATE(BOR);
4259:                  ELSE CALL ERROR('CE');
4260:                END;
4261:          END;
4262:          ELSE CALL ERROR('CE');
4263:        END;
4264:
4265:  /*    162   <ADDING OPERATOR> ::= +                                  */
4266:      CALL SET$OP$TYPE(05H);
4267:  /*    163                         ^ -                                */
4268:      CALL SET$OP$TYPE(06H);
4269:  /*    164                         ^ OR                               */
4270:      CALL SET$OP$TYPE(07H);
4271:
4272:  /*    165   <FACTOR> ::= <VARIABLE>                                  */
4273:      IF (FORM$FIELD(MP) = 05H) OR (FORM$FIELD(MP) = 0DH) THEN
4274:        CALL CALL$A$PROC(FALSE);
4275:      ELSE CALL LOAD$VARI(SP);
4276:  /*    166                   ^ <VARIABLE> ( <ACTUAL PARA LIST> )      */
4277:      CALL CALL$A$PROC(TRUE);
4278:  /*    167                   ^ ( <EXPRESSION> )                       */
4279:      CALL COPY$STACKS(MP, MPP1);
4280:  /*    168                   ^ <SET>                                  */
4281:      ;
4282:  /*    169                   ^ NOT <FACTOR>                           */
4283:      DO;
4284:        IF EXPRESS$STK(SP) = BOOLEAN$TYPE THEN
```

```
4285:          CALL GENERATE(NOTX);
4286:        ELSE CALL ERROR('CE');
4287:        CALL COPY$STACKS(MP,SP);
4288:      END;
4289:  /*    170                 ^ <NUMBER>                                  */
4290:      IF TYPENUM=INTEGER$TYPE THEN
4291:      DO;
4292:        EXPRESS$STK(SP)  =INTEGER$TYPE;
4293:        ALLC$QTY=CONVERTI(SP,POS);
4294:        CALL GEN$ADDR(LDII,ALLC$QTY);
4295:      END;
4296:      ELSE DO;
4297:        EXPRESS$STK(SP)  =UNSIGN$EXPON;
4298:        CALL CONVRTBCD(SP,POS);
4299:        CALL GENERATE(LDIB);
4300:        DO PTRPTR=0 TO BCDSIZE-1;
4301:          CALL GENERATE(BCDNUM(PTRPTR));
4302:        END;
4303:      END;
4304:  /*    171                 ^ NIL                                       */
4305:      ;
4306:  /*    172                 ^ <STRING>                                  */
4307:      DO;
4308:        EXPRESS$STK(SP) = STRING$TYPE;
4309:        CALL GENERATE(LDSI);
4310:        DO FOREVER;
4311:          DO PTRPTR = 1 TO ACCUM;
4312:            CALL GENERATE(ACCUM(PTRPTR));
4313:          END;
4314:          IF CONT THEN /* STRING > 32 CHARS */
4315:            CALL SCANNER;
4316:          ELSE DO;
4317:            CALL GENERATE(NOP);
4318:            RETURN;
4319:          END;
4320:        END;
```

```
4321:     END;
4322:
4323:  /*    173   <ACTUAL PARA LIST> ::= <ACTUAL PARA>                    */
4324:        PARMNUM(SP) = 1;
4325:  /*    174                          ^ <ACTUAL PARA LIST> ,          */
4326:  /*    174                            <ACTUAL PARA>                 */
4327:        PARMNUM(MP) = PARMNUM(MP) + 1;
4328:
4329:  /*    175   <SET> ::= [ <ELEMENT LIST> ]                           */
4330:     CALL COPY$STACKS(MP, MPP1);
4331:
4332:  /*    176   <ELEMENT LIST> ::=                                     */
4333:     CALL COPY$STACKS(SP, SP-3);
4334:  /*    177                        ^ <XELEMENT LIST>                 */
4335:     ;
4336:
4337:  /*    178   <XELEMENT LIST> ::= <ELEMENT>                          */
4338:     ;
4339:  /*    179                          ^ <XELEMENT LIST> , <ELEMENT>   */
4340:     IF EXPRESS$STK(MP) <> EXPRESS$STK(SP) THEN CALL ERROR('ET');
4341:
4342:  /*    180   <ELEMENT> ::= <EXPRESSION>                             */
4343:     ;
4344:  /*    181                       ^ <EXPRESSION> .. <EXPRESSION>     */
4345:     IF EXPRESS$STK(MP) <> EXPRESS$STK(SP) THEN CALL ERROR('ET');
4346:
4347:  /*    182   <GOTO STMT> ::= GOTO <LABEL>                           */
4348:     IF LOOKUP$PN$ID(SP,LABL$ENTRY) THEN
4349:     DO;
4350:       CALL SETADDRPTR(5);
4351:       CALL SETADDRPTR(6+BYTEPTR);
4352:       CALL GEN$ADDR(BRL,ADDRPTR);
4353:     END;
4354:     ELSE DO;
4355:       CALL ERROR('UL');
4356:       CALL GENERATE(NOP); CALL GENERATE(NOP);
```

```
4357:     END;
4358:
4359:  /*    183   <COMPOUND STMT> ::= BEGIN <STMT LISTS> END                  */
4360:     ;
4361:
4362:  /*    184   <STMT LISTS> ::= <STMT>                                     */
4363:     ;
4364:  /*    185                  ^ <STMT LISTS> ; <STMT>                     */
4365:     ;
4366:
4367:  /*    186   <PROCEDURE STMT> ::= <PROCEDURE IDENT>                      */
4368:     CALL CALL$A$PROC(FALSE);
4369:  /*    187                        ^ <PROCEDURE IDENT> (                  */
4370:  /*    187                          <ACTUAL PARA LIST> )                 */
4371:     IF FORM$FIELD(MP) = BUILT$IN$PROC THEN
4372:       CALL CALL$A$PROC(FALSE);
4373:     ELSE CALL CALL$A$PROC(TRUE);
4374:
4375:  /*    188   <PROCEDURE IDENT> ::= <IDENTIFIER>                          */
4376:     DO;
4377:       IF NOT LOOKUP$ONLY(SP) THEN
4378:         CALL ERROR('UP');
4379:       ELSE DO;
4380:         BASELOC(SP) = LOOKUP$ADDR;
4381:         CALL SETADDRPTR(4);
4382:         FORM$FIELD(SP) = BYTEPTR;
4383:         IF FORM$FIELD(SP) = BUILT$IN$PROC THEN /* BUILT$IN$PROCEDURE */
4384:         DO;
4385:           CALL SET$PAST$PN(7);
4386:           IF BYTEPTR = 28 THEN
4387:           DO;
4388:             PARMNUM(SP) = 2;
4389:             PARMNUMLOC(SP) = APTRADDR + 1;
4390:           END;
4391:           ELSE IF BYTEPTR > 21 THEN
4392:             DO CASE (BYTEPTR - 22);
```

```
4393:                NEW$STMT = TRUE;
4394:                DISPOSE$STMT = TRUE;
4395:                READ$STMT = TRUE;
4396:                READ$STMT = TRUE;
4397:                WRITE$STMT = TRUE;
4398:                WRITE$STMT = TRUE;
4399:              END;      /* OF CASE (BYTEPTR - 22) */
4400:          END;
4401:          ELSE DO; /* NOT BUILT IN */
4402:            CALL SET$PAST$PN(7);
4403:            PARMNUM(SP) = BYTEPTR;
4404:            CALL SET$PAST$PN(8);
4405:            PARMNUMLOC(SP) = ADDRPTR;
4406:            APTRADDR = APTRADDR + 6;
4407:            LABELSTACK(SP) = ADDRPTR;
4408:            READPARMS = TRUE;
4409:            PARMNUMLOC(SP+2) = PARMNUMLOC(SP);
4410:          END;
4411:        END;
4412:      END;
4413:
4414:  /*     189   <ACTUAL PARA> ::= <EXPRESSION>                          */
4415:      IF READ$STMT THEN CALL READ$VAR;
4416:      ELSE IF WRITE$STMT THEN CALL WRITE$VAR(0);
4417:        ELSE IF NOT(READPARMS) THEN
4418:        DO;
4419:          READPARMS = TRUE;
4420:          CALL GENERATE(PARMX);  /* PARAMETER IS AN EXPRESSION VALUE */
4421:        END;
4422:  /*     190                    ^ <EXPRESSION> : <EXPRESSION>          */
4423:      IF NOT WRITE$STMT THEN CALL ERROR('PE');
4424:      ELSE DO;
4425:        IF EXPRESS$STK(SP) <> INTEGER$TYPE THEN CALL ERROR('WP');
4426:        CALL WRITE$VAR(1);
4427:      END;
4428:  /*     191                    ^ <EXPRESSION> : <EXPRESSION> :        */
```

```
4429:  /*    191                          <EXPRESSION>                          */
4430:     IF NOT WRITE$STMT THEN CALL ERROR('PE');
4431:     ELSE DO;
4432:       IF EXPRESS$STK(MP) <> UNSIGN$EXPON THEN CALL ERROR('RT');
4433:       IF (EXPRESS$STK(SP) <> INTEGER$TYPE) AND
4434:         (EXPRESS$STK(SP-2) <> INTEGER$TYPE) THEN CALL ERROR('WP');
4435:       CALL WRITE$VAR(2);
4436:     END;
4437:
4438:  /*    192   <CASE STMT> ::= <CASE EXPRESS> <CASE LIST ELEMT LIST>     */
4439:  /*    192                   END                                       */
4440:     DO;
4441:       LABLCOUNT = LABLCOUNT + 1;
4442:       CALL GEN$ADDR(LBL,LABELSTACK(MP));
4443:       CASE$COUNT = CASE$COUNT - 1;
4444:     END;
4445:
4446:  /*    193   <CASE EXPRESS> ::= CASE <EXPRESSION> OF                   */
4447:     DO;
4448:      CASE$STMT=TRUE;
4449:      IF (EXPRESS$STK(MPP1) = UNSIGN$EXPON) THEN CALL ERROR('RT');
4450:      LABELSTACK(MP) = LABLCOUNT;
4451:      LABLCOUNT = LABLCOUNT + 1;
4452:      CASE$STK(CASE$COUNT := CASE$COUNT + 1) = 0;
4453:     END;
4454:
4455:  /*    194   <CASE LIST ELEMT LIST> ::= <CASE LIST ELEMENT>            */
4456:     IF CASE$STMT THEN
4457:     DO;
4458:       CALL GEN$ADDR(BRL,LABELSTACK(MP-1));
4459:       CALL GEN$ADDR(LBL,(LABELSTACK(MP)+1));
4460:     END;
4461:  /*    195                              ^ <CASE LIST ELEMT LIST> ;     */
4462:  /*    195                                <CASE LIST ELEMENT>          */
4463:     IF CASE$STMT THEN
4464:     DO;
```

```
4465:       CALL GEN$ADDR(FRL,LABELSTACK(MP-1));
4466:       CALL GEN$ADDR(LBL,(LABELSTACK(SP)+1));
4467:     END;
4468:
4469:  /*    196   <CASE LIST ELEMENT> ::=                                  */
4470:     CASE$STMT = FALSE;
4471:  /*    197                          ^ <CASE PREFIX> <STMT>            */
4472:     ;
4473:
4474:  /*    198   <CASE PREFIX> ::= <CASE LABEL LIST> :                    */
4475:     DO;
4476:       CALL GEN$ADDR(FRL,(LABELSTACK(MP)+1));
4477:       CALL GEN$ADDR(LBL,LABELSTACK(MP));
4478:     END;
4479:
4480:  /*    199   <WITH STMT> ::= <WITH> <REC VARIABLE LIST> <DO>          */
4481:  /*    199                   <BAL STMT>                               */
4482:     ;
4483:
4484:  /*    200   <WITH> ::= WITH                                          */
4485:     ;
4486:
4487:  /*    201   <REC VARIABLE LIST> ::= <VARIABLE>                       */
4488:      ;
4489:  /*    202                          ^ <REC VARIABLE LIST> ,           */
4490:  /*    202                            <VARIABLE>                      */
4491:     ;
4492:
4493:  /*    203   <DO> ::= DO                                              */
4494:     DO;
4495:       LABELSTACK(SP) = LABLCOUNT;
4496:       CALL GEN$ADDR(BLC,LABELSTACK(SP));
4497:       LABLCOUNT = LABLCOUNT + 1;
4498:     END;
4499:
4500:  /*    204   <WHILE STMT> ::= <WHILE> <EXPRESSION> <DO> <BAL STMT>    */
```

```
4501:     DO;
4502:       CALL GEN$ADDR(BRL,LABELSTACK(MP));
4503:       CALL GEN$ADDR(LBL,LABELSTACK(SP-1));
4504:     END;
4505:
4506:  /*    205   <WHILE> ::= WHILE                                          */
4507:     DO;
4508:       LABELSTACK(SP) = LABLCOUNT;
4509:       CALL GEN$ADDR(LBL,LABELSTACK(SP));
4510:       LABLCOUNT = LABLCOUNT + 1;
4511:     END;
4512:
4513:  /*    206   <FOR STMT> ::= FOR <CONTROL VARIABLE> := <FOR LIST>        */
4514:  /*    206                  <DO> <BAL STMT>                             */
4515:     DO;
4516:       CALL GEN$ADDR(BRL,(LABELSTACK(SP-2)+1));
4517:       CALL GEN$ADDR(LBL,LABELSTACK(SP-1));
4518:     END;
4519:
4520:  /*    207   <FOR LIST> ::= <INITIAL VALUE> <TO> <FINAL VALUE>          */
4521:     DO;
4522:       IF EXPRESS$STK(MP) <> EXPRESS$STK(SP) THEN CALL ERROR('ET');
4523:       CALL GENERATE(GEQI);
4524:     END;
4525:  /*    208                ^ <INITIAL VALUE> <DOWNTO> <FINAL VALUE>     */
4526:     DO;
4527:       IF EXPRESS$STK(MP) <> EXPRESS$STK(SP) THEN CALL ERROR('ET');
4528:       CALL GENERATE(LEQI);
4529:     END;
4530:
4531:  /*    209   <CONTROL VARIABLE> ::= <IDENTIFIER>                        */
4532:     DO;
4533:       VARPARM = FALSE;
4534:       IF NOT LOOKUP$ONLY(SP) THEN
4535:         CALL ERROR('CV');
4536:       ELSE DO;
```

```
4537:          APTRADDR = LOOKUP$ADDR + 4;
4538:          IF BYTEPTR = 1BH THEN CALL ERROR('CV');
4539:          ELSE CALL SET$VAR$TYPE;
4540:        END;
4541:      END;
4542:
4543:  /*    210   <INITIAL VALUE> ::= <EXPRESSION>                    */
4544:      DO;
4545:        CALL ASSIGN$VARI(SP-2, TRUE);
4546:        LABELSTACK(SP) = LABLCOUNT;
4547:        LABLCOUNT = LABLCOUNT + 2;
4548:        CALL GEN$ADDR(BRL,LABELSTACK(SP));
4549:        CALL GEN$ADDR(LBL,(LABELSTACK(SP)+1));
4550:        CALL LOAD$VARI(SP-2);
4551:      END;
4552:
4553:  /*    211   <FINAL VALUE> ::= <EXPRESSION>                      */
4554:      ;
4555:
4556:  /*    212   <REPEAT STMT> ::= <REPEAT> <STMT LISTS> UNTIL       */
4557:  /*    212                      <EXPRESSION>                     */
4558:      DO;
4559:        IF EXPRESS$STK(SP) = BOOLEAN$TYPE THEN
4560:        DO;
4561:          CALL GENERATE(NOTX);
4562:          CALL GEN$ADDR(BLC,LABELSTACK(MP));
4563:        END;
4564:        ELSE CALL ERROR('CE');
4565:      END;
4566:
4567:  /*    213   <REPEAT> ::= REPEAT                                 */
4568:      DO;
4569:        CALL GEN$ADDR(LBL,LABLCOUNT);
4570:        LABELSTACK(SP)=LABLCOUNT;
4571:        LABLCOUNT = LABLCOUNT + 1;
4572:      END;
```

```
4573:
4574:  /*     214    <TO> ::= TO                                              */
4575:      DO;
4576:        CALL GENERATE(INC);
4577:        CALL GEN$ADDR(LBL,LABELSTACK(SP-1));
4578:      END;
4579:
4580:  /*     215    <DOWNTO> ::= DOWNTO                                      */
4581:      DO;
4582:        CALL GENERATE(DEC);
4583:        CALL GEN$ADDR(LBL,LABELSTACK(SP-1));
4584:      END;
4585:
4586:
4587: END;                                          /* OF CASE STATEMENT */
4588:
4589: END SYNTHESIZE;
4590:
4591:
4592: /*****************************************************************************/
4593: /*****************************************************************************/
4594: /***                          ERROR RECOVERY ROUTINES                      ***/
4595: /*****************************************************************************/
4596: /*****************************************************************************/
4597:
4598:
4599: NOCONFLICT: PROC (CSTATE) BYTE;
4600:      DCL CSTATE STATESIZE, (I,J,K) INDEXSIZE;
4601:      J= INDEX1(CSTATE);
4602:      K= J + INDEX2(CSTATE) - 1;
4603:      DO I = J TO K;
4604:         IF READ1(I) = TOKEN THEN RETURN TRUE;
4605:      END;
4606:      RETURN FALSE;
4607: END NOCONFLICT;
4608:
```

```
4609:
4610: RECOVER: PROC STATESIZE;
4611:     DCL TSP BYTE, RSTATE STATESIZE;
4612:     DO FOREVER;
4613:        TSP = SP;
4614:        DO WHILE TSP <> 255;
4615:           IF NOCONFLICT(RSTATE:=STATESTACK(TSP)) THEN
4616:              DO;                    /* STATE WILL READ TOKEN */
4617:                 IF SP <> TSP THEN SP = TSP - 1;
4618:                 RETURN RSTATE;
4619:              END;
4620:           TSP = TSP - 1;
4621:        END;
4622:        CALL SCANNER;
4623:     END;
4624: END RECOVER;
4625:
4626:
4627: /*****************************************************************/
4628: /*****************************************************************/
4629: /***                  LALR  PARSER  ROUTINES                   ***/
4630: /*****************************************************************/
4631: /*****************************************************************/
4632:
4633:
4634:     DO;                                  /*BLOCK FOR DECLARATIONS*/
4635:     DCL    (I,J,K) INDEXSIZE, INDEX BYTE;
4636:
4637:
4638:     INITIALIZE: PROC;
4639:         CALL INIT$SCANNER;
4640:         CALL INIT$SYMTBL;
4641:         CALL INITIALIZE$SYNTHESIZE;
4642:         CALL TITLE;
4643:     END INITIALIZE;
4644:
```

```
4645:
4646:      GETIN1: PROC INDEXSIZE;
4647:          RETURN INDEX1(STATE);
4648:      END GETIN1;
4649:
4650:
4651:      GETIN2: PROC INDEXSIZE;
4652:          RETURN INDEX2(STATE);
4653:      END GETIN2;
4654:
4655:
4656:      INCSP: PROC;
4657:          IF (SP := SP + 1) = LENGTH(STATESTACK) THEN
4658:              CALL ERROR('SO');
4659:      END INCSP;
4660:
4661:
4662:      LOOKAHEAD: PROC;
4663:          IF NOLOOK THEN
4664:             DO;
4665:                CALL SCANNER;
4666:                NOLOOK = FALSE;
4667:                IF LISTTOKEN THEN
4668:                   CALL PRINT$TOKEN;
4669:             END;
4670:      END LOOKAHEAD;
4671:
4672:
4673:      SET$VARC$I: PROC(I);          /* SET VARC, AND INCRMNT VARINDEX */
4674:          DCL   I BYTE;
4675:          VARC(VARINDEX)=I;
4676:          IF (VARINDEX:=VARINDEX+1) > LENGTH(VARC) THEN
4677:             CALL ERROR('VO');
4678:      END SET$VARC$I;
4679:
4680:
```

```
4681:                 /* INITIALIZE FOR INPUT - OUTPUT OPERATIONS  */
4682: CALL MOVF(.RFCB,.WFCB,9);      /* PUT FILENAME IN WRITE FCB */
4683: CALL SETUP$INT$FIL; /* CREATES OUTPUT FILE FOR GENERATED CODE */
4684: CALL INITIALIZE;
4685:
4686: DO FOREVER;
4687:    DO WHILE TRUE;                          /* INITIALIZE VARIABLES */
4688:       COMPILING,NOLOOK=TRUE;
4689:       STATE=STARTS;
4690:       SP=255;
4691:       VARINDEX,VAR = 0;
4692:
4693:       DO WHILE COMPILING;
4694:          IF STATE<=MAXRNO THEN                /* READ STATE */
4695:             DO;
4696:                CALL INCSP;
4697:                STATESTACK(SP)=STATE;
4698:                I=GETIN1;
4699:                CALL LOOKAHEAD;
4700:                J=I+GETIN2-1;
4701:                DO I=I TO J;
4702:                   IF READ1(I)=TOKEN THEN          /* SAVE TOKEN */
4703:                      DO;          /* COPY ACCUM TO PROPER POSITION */
4704:                         VAR(SP)=VARINDEX;
4705:                         DO INDEX = 0 TO ACCUM;
4706:                            CALL SET$VARC$I(ACCUM(INDEX));
4707:                         END;
4708:                         HASH(SP) = HASHCODE;
4709:                                  /* SAVE RELATIVE TABLE LOCATION */
4710:                         STATE=READ2(I);
4711:                         NOLOOK=TRUE;
4712:                         I=J;
4713:                      END;
4714:                   ELSE IF I=J THEN
4715:                           DO;
4716:                              CALL ERROR('NP');
```

```
4717:                                IF (STATE := RECOVER)=Ø THEN
4718:                                   COMPILING = FALSE;
4719:                             END;
4720:                   END;
4721:                END;
4722:             ELSE IF STATE>MAXPNO THEN          /* APPLY PRODUCTION STATE */
4723:                      DO;
4724:                         MP=SP-GETIN2;
4725:                         MPP1=MP+1;
4726:                         PRODUCTION = STATE-MAXPNO;
4727:                         CALL SYNTHESIZE;
4728:                         SP=MP;
4729:                         I=GETIN1;
4730:                         VARINDEX=VAR(SP);
4731:                         J=STATESTACK(SP);
4732:                         DO WHILE (K:=APPLY1(I)) <> Ø AND J <> K;
4733:                            I=I+1;
4734:                         END;
4735:                         IF (STATE:= APPLY2(I))=Ø THEN COMPILING = FALSE;
4736:                      END;
4737:                   ELSE IF STATE<= MAXLNO THEN      /* LOOKAHEAD STATE */
4738:                            DO;
4739:                               I=GETIN1;
4740:                               CALL LOOKAHEAD;
4741:                               DO WHILE (K:=LOOK1(I)) <> Ø AND TOKEN <> K;
4742:                                  I=I+1;
4743:                               END;
4744:                               STATE=LOOK2(I);
4745:                            END;
4746:                                                    /* PUSH STATE  */
4747:                         ELSE DO;
4748:                                 CALL INCSP;
4749:                                 STATESTACK(SP)= GETIN2;
4750:                                 STATE=GETIN1;
4751:                              END;
4752:       END;                                        /* OF WHILE COMPILING  */
```

```
4753:     END;                                  /*OF WHILE      TRUE      */
4754: END;                                    /*OF DO FOREVER*/
4755:
4756: END;                                  /* OF BLOCK FOR PARSER  */
4757: END;                                /*OF BLOCK FOR DECLARATIONS*/
4758: EOF
```

```
 1:
 2:
 3:      /*
 4:       ***************************************************************
 5:       *                                                             *
 6:       *                                                             *
 7:       *                          DECODE                             *
 8:       *                                                             *
 9:       *                                                             *
10:       ***************************************************************
11:      */
12:
13:
14:     /* THIS PROGRAM TAKES THE CODE OUTPUT FROM THE PASCAL COMPILER
15:     AND CONVERTS IT INTO A READABLE OUTPUT TO FACILITATE DEBUGGING */
16:
17: 100H:
18:
19: DECLARE
20:
21: LIT             LITERALLY       'LITERALLY',
22: BOOT            LIT             '0',
23: BDOS            LIT             '5H',
24: FCB             ADDRESS         INITIAL (5CH),
25: FCB$BYTE        BASED           FCB (1) BYTE,
26: I               BYTE,
27: TRUE            LIT             '1',
28: FALSE           LIT             '0',
29: ADDR            ADDRESS         INITIAL (100H),
30: CHAR            BASED           ADDR BYTE,
31: BUFF$END        LIT             '0FFH',
32: BCDNUM(8)       BYTE,
33: FILE$TYPE                       DATA('P','I','N');
34:
35:
36:
```

```
37: MON1: PROCEDURE (F,A);
38:     DECLARE F BYTE, A ADDRESS;
39:     GO TO BDOS;
40: END MON1;
41:
42:
43:
44: MON2: PROCEDURE (F,A) BYTE;
45:     DECLARE F BYTE, A ADDRESS;
46:     GO TO BDOS;
47:     RETURN 0;
48: END MON2;
49:
50:
51:
52: PRINT$CHAR: PROCEDURE (CHAR);
53:     DECLARE CHAR BYTE;
54:     CALL MON1(2,CHAR);
55: END PRINT$CHAR;
56:
57:
58:
59: CRLF: PROCEDURE;
60:     CALL PRINT$CHAR(13);
61:     CALL PRINT$CHAR(10);
62: END CRLF;
63:
64:
65:
66: P: PROCEDURE(ADD1);
67:     DECLARE ADD1 ADDRESS, C BASED ADD1 (1) BYTE;
68:     CALL CRLF;
69:     DO I=0 TO 4;
70:         CALL PRINT$CHAR(C(I));
71:     END;
72:     CALL PRINT$CHAR(' ');
```

```
 73: END P;
 74:
 75:
 76:
 77: GET$CHAR: PROCEDURE BYTE;
 78:     IF (ADDR:=ADDR+1) > BUFF$END THEN
 79:     DO;
 80:          IF MON2(20,FCB) <> 0 THEN
 81:          DO;
 82:               CALL P(.('END  '));
 83:          END;
 84:          ADDR=80H;
 85:     END;
 86:     RETURN CHAR;
 87: END GET$CHAR;
 88:
 89:
 90:
 91:
 92: WRITE$STRING: PROCEDURE;
 93:     DECLARE J BYTE;
 94:     DO WHILE 1;
 95:       J = GET$CHAR;
 96:       IF J <> 00H THEN CALL PRINT$CHAR(J);
 97:       ELSE RETURN;
 98:     END;
 99: END WRITE$STRING;
100:
101:
102:
103: D$CHAR: PROCEDURE(OUTPUT$BYTE);
104:     DECLARE OUTPUT$BYTE BYTE;
105:     IF OUTPUT$BYTE < 10 THEN CALL PRINT$CHAR(OUTPUT$BYTE + 30H);
106:     ELSE CALL PRINT$CHAR(OUTPUT$BYTE + 37H);
107: END D$CHAR;
108:
```

```
109:
110:
111: D: PROCEDURE (COUNT);
112:      DECLARE (COUNT, J) ADDRESS;
113:      DO J=1 TO COUNT;
114:           CALL D$CHAR(SHR(GET$CHAR,4));
115:           CALL D$CHAR(CHAR AND 0FH);
116:           CALL PRINT$CHAR(' ');
117:      END;
118: END D;
119:
120:
121:
122: PRINT$BCD: PROCEDURE (COUNT);
123:      DECLARE (COUNT, J, K, K1, K2) BYTE;
124:
125:     P$EXPON: PROCEDURE(VALU);
126:        DECLARE (VALU, X, COUNT1) BYTE;
127:        DECLARE DECI(3) BYTE INITIAL (100,10,1);
128:        DECLARE FLAG BYTE;
129:        FLAG = FALSE;
130:        DO X = 0 TO 2;
131:          COUNT1 = 30H;
132:          DO WHILE VALU >= X;
133:            VALU = VALU - X;
134:            FLAG = TRUE;
135:            COUNT1 = COUNT1 + 1;
136:          END;
137:          IF FLAG OR (X >= 1) THEN
138:            CALL PRINT$CHAR(COUNT1);
139:          ELSE CALL PRINT$CHAR(' ');
140:        END;
141:        RETURN;
142:     END P$EXPON;
143:
144:
```

```
145:      DO J = 0 TO (COUNT-1);
146:        BCDNUM(J) = GET$CHAR;
147:      END;
148:      K = BCDNUM(COUNT-1);
149:      CALL PRINT$CHAR(' ');
150:      K = (K AND 7FH);
151:      IF (SHR(K,4) < 4) THEN K1 = DEC(40H - K);
152:      ELSE K1 = DEC(K - 40H-6H);
153:      DO WHILE BCDNUM(COUNT-2) = 00H;
154:        COUNT = COUNT - 1;
155:      END;
156:      COUNT = COUNT - 2;
157:      DO K2 = 0 TO COUNT;
158:        CALL D$CHAR(SHR(BCDNUM(K2),4));
159:        IF K2 = 0 THEN CALL PRINT$CHAR(2EH);
160:        IF NOT((K2=COUNT) AND ((BCDNUM(K2) AND 0FH)=0)) THEN
161:          CALL D$CHAR(BCDNUM(K2) AND 0FH);
162:      END;
163:      CALL PRINT$CHAR(45H);
164:      IF (SHR(K,4) < 4) THEN
165:      DO;
166:        CALL PRINT$CHAR(2DH);
167:        CALL P$EXPON(K1 := K1 + 1);
168:      END;
169:      ELSE CALL P$EXPON(K1 := K1 - 1);
170: END PRINT$BCD;
171:
172:
173:
174:
175: PRINT$REST: PROCEDURE;
176:      DECLARE
177:      ENDP          LIT         '1H',
178:      EQLS          LIT         '23H',
179:      NEQS          LIT         '24H',
180:      LBL           LIT         '2H',
```

```
181:      LDII          LIT         '4H',
182:      ALL           LIT         '0BH',
183:      LITA          LIT         '0CH',
184:      BRL           LIT         '3EH',
185:      BLC           LIT         '3FH',
186:      PRO           LIT         '5H',
187:      ANDX          LIT         '32H',
188:      BOR           LIT         '33H',
189:      PARM          LIT         '43H',
190:      PARMV         LIT         '44H',
191:      LDIB          LIT         '3H',
192:      WRTB          LIT         '53H',
193:      WRTI          LIT         '54H',
194:      WRTS          LIT         '55H',
195:      LDSI          LIT         '4BH',
196:      KASE          LIT         '4CH';
197:
198:
199:
200:      IF CHAR = ENDP THEN
201:      DO;
202:           CALL P(.('END  '));
203:           GO TO BOOT;
204:      END;
205:      IF (CHAR=EQLS) OR (CHAR=NEQS) OR (CHAR=WRTB) OR (CHAR=WRTI) OR
206:           (CHAR=WRTS) THEN DO; CALL D(1); RETURN; END;
207:      IF (CHAR=LBL) OR (CHAR=LDII) OR (CHAR=ALL) OR (CHAR=LITA) OR
208:           (CHAR=BRL) OR (CHAR=BLC) OR (CHAR=PRO) OR (CHAR=ANDX) OR
209:           (CHAR=BOR) OR (CHAR=PARM) OR (CHAR=PARMV) THEN DO;
210:           CALL D(2); RETURN; END;
211:      IF CHAR = KASE THEN DO; CALL D(4); RETURN; END;
212:      IF CHAR = LDIB THEN DO; CALL PRINT$BCD(8); RETURN; END;
213:      IF CHAR = LDSI THEN DO; CALL WRITE$STRING; RETURN; END;
214:      RETURN;
215: END PRINT$REST;
216:
```

```
217:
218:
219: /****    PROGRAM EXECUTION STARTS HERE    ****/
220:
221:
222: FCB$BYTE(32), FCB$BYTE(0) = 0;
223: DO I = 0 TO 2;
224:     FCB$BYTE(I+9) = FILE$TYPE(I);
225: END;
226:
227: IF MON2(15,FCB) = 255 THEN
228: DO;
229:     CALL P(.('ZZZZ '));
230:     GO TO BOOT;
231: END;
232:
233:
234: DO WHILE 1;
235:     IF GET$CHAR <= 72H THEN DO CASE CHAR;
236:         CALL P(.('NOP  '));
237:         CALL P(.('ENDP '));
238:         CALL P(.('LBL  '));
239:         CALL P(.('LDIB '));
240:         CALL P(.('LDII '));
241:         CALL P(.('PRO  '));
242:         CALL P(.('RTN  '));
243:         CALL P(.('SAVP '));
244:         CALL P(.('UNSP '));
245:         CALL P(.('CNVB '));
246:         CALL P(.('CNVI '));
247:         CALL P(.('ALL  '));
248:         CALL P(.('LITA '));
249:         CALL P(.('ADDB '));
250:         CALL P(.('ADDI '));
251:         CALL P(.('SUBB '));
252:         CALL P(.('SUBI '));
```

```
253:          CALL P(.('MULB '));
254:          CALL P(.('MULI '));
255:          CALL P(.('DIVB '));
256:          CALL P(.('DIVI '));
257:          CALL P(.('MODX '));
258:          CALL P(.('EQLI '));
259:          CALL P(.('NEQI '));
260:          CALL P(.('LEQI '));
261:          CALL P(.('GEQI '));
262:          CALL P(.('LSSI '));
263:          CALL P(.('GTRI '));
264:          CALL P(.('XIN  '));
265:          CALL P(.('EQLB '));
266:          CALL P(.('NEQB '));
267:          CALL P(.('LEQB '));
268:          CALL P(.('GEQB '));
269:          CALL P(.('LSSB '));
270:          CALL P(.('GRTB '));
271:          CALL P(.('EQLS '));
272:          CALL P(.('NEQS '));
273:          CALL P(.('LEQS '));
274:          CALL P(.('GEQS '));
275:          CALL P(.('LSSS '));
276:          CALL P(.('GRTS '));
277:          CALL P(.('EQSET'));
278:          CALL P(.('NEQST'));
279:          CALL P(.('INCL1'));
280:          CALL P(.('INCL2'));
281:          CALL P(.('NEGB '));
282:          CALL P(.('NEGI '));
283:          CALL P(.('COMB '));
284:          CALL P(.('COMI '));
285:          CALL P(.('NOTX '));
286:          CALL P(.('ANDX '));
287:          CALL P(.('BOR  '));
288:          CALL P(.('STOB '));
```

```
289:          CALL P(.('STOI '));
290:          CALL P(.('STO  '));
291:          CALL P(.('STDB '));
292:          CALL P(.('STDI '));
293:          CALL P(.('STD  '));
294:          CALL P(.('UNION'));
295:          CALL P(.('STDIF'));
296:          CALL P(.('ISEC '));
297:          CALL P(.('CNAI '));
298:          CALL P(.('BRL  '));
299:          CALL P(.('BLC  '));
300:          CALL P(.('CN2I '));
301:          CALL P(.('MKSET'));
302:          CALL P(.('XCHG '));
303:          CALL P(.('PARM '));
304:          CALL P(.('PARMV'));
305:          CALL P(.('PARMX'));
306:          CALL P(.('INC  '));
307:          CALL P(.('DEC  '));
308:          CALL P(.('DEL  '));
309:          CALL P(.('WRT  '));
310:          CALL P(.('SUB  '));
311:          CALL P(.('LDSI '));
312:          CALL P(.('CASE '));
313:          CALL P(.('LOD  '));
314:          CALL P(.('LODB '));
315:          CALL P(.('LODI '));
316:          CALL P(.('RDVB '));
317:          CALL P(.('RDVI '));
318:          CALL P(.('RDVS '));
319:          CALL P(.('WRTB '));
320:          CALL P(.('WRTI '));
321:          CALL P(.('WRTS '));
322:          CALL P(.('DUMP '));
323:          CALL P(.('ABS  '));
324:          CALL P(.('SQR  '));
```

```
325:            CALL P(.('SIN   '));
326:            CALL P(.('COS   '));
327:            CALL P(.('ARCTN '));
328:            CALL P(.('EXP   '));
329:            CALL P(.('LN    '));
330:            CALL P(.('SQRT  '));
331:            CALL P(.('ODD   '));
332:            CALL P(.('EOLN  '));
333:            CALL P(.('EXF   '));
334:            CALL P(.('TRUNC '));
335:            CALL P(.('ROUND '));
336:            CALL P(.('ORD   '));
337:            CALL P(.('CHR   '));
338:            CALL P(.('SUCC  '));
339:            CALL P(.('PRED  '));
340:            CALL P(.('SEEK  '));
341:            CALL P(.('PUT   '));
342:            CALL P(.('GET   '));
343:            CALL P(.('RESET '));
344:            CALL P(.('REWRT '));
345:            CALL P(.('PAGE  '));
346:            CALL P(.('NEW   '));
347:            CALL P(.('DISPZ '));
348:            CALL P(.('FWD   '));
349:            CALL P(.('XTRNL '));
350:            CALL P(.('RDV   '));
351:        END;                          /* OF CASE STATEMENT */
352:        CALL PRINT$REST;
353: END;                              /* END OF DO WHILE */
354: EOF
355:
356: R;
357: >
```

```
 1:
 2:
 3:       /*
 4:        ***************************************************************
 5:        *                                                             *
 6:        *                                                             *
 7:        *                    SYMBOL$TABLE$PRINTOUT                    *
 8:        *                                                             *
 9:        *                                                             *
10:        ***************************************************************
11:       */
12:
13:
14:      /* THIS PROGRAM TAKES THE OUTPUT FROM THE PASCAL SYMBOL TABLE
15:      AND CONVERTS IT INTO A READABLE OUTPUT TO FACILITATE DEBUGGING */
16:
17: 100H:
18:
19: DECLARE
20:
21: LIT             LITERALLY       'LITERALLY',
22: BOOT            LIT             '0',
23: BDOS            LIT             '5H',
24: FCB             ADDRESS         INITIAL (5CH),
25: FCB$BYTE        BASED           FCB (1) BYTE,
26: I               BYTE,
27: TRUE            LIT             '1',
28: FALSE           LIT             '0',
29: COPYING         BYTE            INITIAL(TRUE),
30: ADDR1           ADDRESS         INITIAL (100H),
31: CHAR            BASED           ADDR1 BYTE,
32: BUFF$END        LIT             '0FFH',
33: ADDR            LIT             'ADDRESS',
34: FORMMASK        LIT             '07H',
35: DCL             LIT             'DECLARE',
36: PROC            LIT             'PROCEDURE',
```

```
37: EOFFILLER      LIT             '1AH',
38: BCDNUM(8)      BYTE,
39: FILE$TYPE                      DATA('S','Y','M'),
40: FORM           BYTE,
41: TABLE$START    ADDR, /* STARTING LOCATION AT COMPILATION */
42: OFFSET         ADDR, /* NEW VALUE OF TABLE ENTRY */
43: PARM$LISTING(10)  ADDR, /* LOCATION OF SUBRTN FORMAL PARAM LISTING */
44: SUBRTN  BYTE INITIAL(0),
45: PARM$NUM(10)  BYTE, /* KEEPS COUNT OF NUMBER OF PARAMETERS */
46: SAVE$BASE      ADDR, /* SAVES BASE LOCATION */
47: LPN            BYTE; /* LENGTH OF PRINTNAME */
48:
49:
50: DCL      BASE           ADDR,          /*BASE OF CURRENT ENTRY       */
51:          SBTBLTOP       ADDR,          /*CURRENT TOP OF TABLE (SYM)  */
52:          SBTBL          ADDR,
53:          PTR BASED      BASE BYTE,     /* 1ST BYTE OF ENTRY */
54:          APTRADDR       ADDR,     /* UTILITY VARIABLE TO ACCESS TABLE  */
55:          ADDRPTR BASED APTRADDR ADDR,
56:          BYTEPTR BASED APTRADDR BYTE,
57:          PRINTNAME      ADDR,     /* SET PRIOR TO LOOKUP OR ENTER      */
58:          SYMHASH        BYTE,
59:          LAST$SBTBL$ID ADDR,
60:          PARAMNUMLOC    ADDR,
61:          SBTBLSCOPE     ADDR;
62:
63:
64:
65:
66:
67: MON1: PROCEDURE (F,A);
68:     DECLARE F BYTE, A ADDRESS;
69:     GO TO BDOS;
70: END MON1;
71:
72:
```

```
 73:
 74: MON2: PROCEDURE (F,A) BYTE;
 75:     DECLARE F BYTE, A ADDRESS;
 76:     GO TO BDOS;
 77:     RETURN 0;
 78: END MON2;
 79:
 80:
 81:
 82: PRINT$CHAR: PROCEDURE (CHAR);
 83:     DECLARE CHAR BYTE;
 84:     CALL MON1(2,CHAR);
 85: END PRINT$CHAR;
 86:
 87:
 88:
 89: CRLF: PROCEDURE;
 90:     CALL PRINT$CHAR(13);
 91:     CALL PRINT$CHAR(10);
 92: END CRLF;
 93:
 94:
 95: PRINT: PROC(A);
 96:     DCL     A ADDR;
 97:     CALL MON1(9,A);
 98: END PRINT;
 99:
100:
101:
102: GET$CHAR: PROCEDURE BYTE;
103:     IF (ADDR1:=ADDR1+1) > BUFF$END THEN
104:     DO;
105:          IF MON2(20,FCB) <> 0 THEN
106:          DO;
107:               CALL PRINT(.'THE END  $');
108:          END;
```

```
109:          ADDR1=80H;
110:     END;
111:     RETURN CHAR;
112: END GET$CHAR;
113:
114:
115: D$CHAR: PROCEDURE(OUTPUT$BYTE);
116:     DECLARE OUTPUT$BYTE BYTE;
117:     IF OUTPUT$BYTE < 10 THEN CALL PRINT$CHAR(OUTPUT$BYTE + 30H);
118:     ELSE CALL PRINT$CHAR(OUTPUT$BYTE + 37H);
119: END D$CHAR;
120:
121:
122:
123: D: PROCEDURE (COUNT);
124:     DECLARE (COUNT, J) ADDRESS;
125:     DO J=1 TO COUNT;
126:          CALL D$CHAR(SHR(BYTEPTR,4));
127:          CALL D$CHAR(BYTEPTR AND 0FH);
128:          APTRADDR = APTRADDR + 1;
129:     END;
130: END D;
131:
132:
133:
134: PRINT$BCD: PROCEDURE (COUNT);
135:     DECLARE (COUNT, J, K, K1, K2) BYTE;
136:
137:    P$EXPON: PROCEDURE(VALU);
138:       DECLARE (VALU, X, COUNT1) BYTE;
139:       DECLARE DECI(3) BYTE INITIAL (100,10,1);
140:       DECLARE FLAG BYTE;
141:       FLAG = FALSE;
142:       DO X = 0 TO 2;
143:         COUNT1 = 30H;
144:         DO WHILE VALU >= DECI(X);
```

```
145:            VALU = VALU - DECI(X);
146:            FLAG = TRUE;
147:            COUNT1 = COUNT1 + 1;
148:          END;
149:          IF FLAG OR (X >= 2) THEN
150:            CALL PRINT$CHAR(COUNT1);
151:          ELSE CALL PRINT$CHAR(' ');
152:        END;
153:        RETURN;
154:      END P$EXPON;
155:
156:
157:       DO J = 0 TO (COUNT-1);
158:         BCDNUM(J) = BYTEPTR;
159:         APTRADDR = APTRADDR +1;
160:       END;
161:       K = BCDNUM(COUNT-1);
162:       CALL PRINT$CHAR(' ');
163:       K = (K AND 7FH);
164:       IF (SHR(K,4) < 4) THEN K1 = 40H - K;
165:       ELSE K1 = K - 40H;
166:       DO WHILE BCDNUM(COUNT-2) = 00H;
167:         COUNT = COUNT - 1;
168:       END;
169:       COUNT = COUNT - 2;
170:       DO K2 = 0 TO COUNT;
171:         CALL D$CHAR(SHR(BCDNUM(K2),4));
172:         IF K2 = 0 THEN CALL PRINT$CHAR(2EH);
173:         IF NOT((K2=COUNT) AND ((BCDNUM(K2) AND 0FH)=0)) THEN
174:           CALL D$CHAR(BCDNUM(K2) AND 0FH);
175:       END;
176:       CALL PRINT$CHAR(45H);
177:       IF (SHR(K,4) < 4) THEN
178:       DO;
179:         CALL PRINT$CHAR(2DH);
180:         CALL P$EXPON(K1 := K1 + 1);
```

```
181:      END;
182:      ELSE CALL P$EXPON(K1 := K1 - 1);
183: END PRINT$BCD;
184:
185:
186:
187:
188:
189:
190: FCB$BYTE(32), FCB$BYTE(0) = 0;
191: DO I = 0 TO 2;
192:      FCB$BYTE(I+9) = FILE$TYPE(I);
193: END;
194:
195: IF MON2(15,FCB) = 255 THEN
196: DO;
197:      CALL PRINT(.'ERROR--GONE TO BOOT $');
198:      GO TO BOOT;
199: END;
200:
201:
202:
203:
204:
205: DISKERR: PROC;
206:      DO;
207:         CALL PRINT(.'DE    $');
208:         GOTO BOOT;
209:      END;
210: END DISKERR;
211:
212:
213: PRINTDEC: PROC(VALUE);
214:      DCL      VALUE ADDR, I BYTE, COUNT BYTE;
215:      DCL      DECI(5) ADDR INITIAL(10000,1000,100,10,1);
216:      DCL      FLAG BYTE;
```

```
217:      FLAG = FALSE;
218:      DO I = Ø TO 4;
219:      COUNT = 30H;
220:         DO WHILE VALUE >= DECI(I);
221:            VALUE = VALUE - DECI(I);
222:            FLAG= TRUE;
223:            COUNT = COUNT + 1;
224:         END;
225:         IF FLAG OR (I>= 4) THEN
226:            CALL PRINTCHAR(COUNT);
227:         ELSE
228:            CALL PRINTCHAR(' ');
229:      END;
230:      RETURN;
231: END PRINTDEC;
232:
233:
234: SETADDRPTR: PROC(OFFSET);
235:      DCL OFFSET BYTE;
236:      APTRADDR = BASE + OFFSET;
237: END SETADDRPTR;
238:
239:
240:
241: SET$PAST$PN: PROC(OFFSET);
242:      DCL OFFSET BYTE;
243:      CALL SETADDRPTR(6);
244:      CALL SETADDRPTR(BYTEPTR + OFFSET);
245: END SET$PAST$PN;
246:
247:
248: COPY$SBTBL: PROC;
249:      DCL K ADDR;
250:      K = Ø;
251:      DO WHILE COPYING;
252:           CALL SETADDRPTR(K);
```

```
253:          BYTEPTR = GETCHAR;
254:          K = K + 1;
255:          IF BYTEPTR = EOFFILLER THEN
256:          DO;
257:                K = K + 1;
258:                CALL SETADDRPTR(K);
259:                BYTEPTR = GETCHAR;
260:                IF BYTEPTR = EOFFILLER THEN
261:                DO;
262:                  COPYING = FALSE;
263:                  BYTEPTR = 00H;
264:                END;
265:          END;
266:     END;
267: END COPY$SBTBL;
268:
269:
270:
271: RESET$LOCATION: PROC(A) ADDR;
272:     DCL A ADDR;
273:     OFFSET = A - TABLE$START;
274:     RETURN OFFSET;
275: END RESET$LOCATION;
276:
277:
278: TAB1: PROC;
279:     CALL PRINT(.'     $');
280: END TAB1;
281:
282:
283: TAB2: PROC;
284:     CALL TAB1;
285:     CALL TAB1;
286: END TAB2;
287:
288:
```

```
289: WRITE$ENTRY: PROC;
290:     DO CASE (FORM AND 07H);
291:          CALL PRINT(.'LABEL ENTRY $');
292:          CALL PRINT(.'CONSTANT ENTRY $');
293:          CALL PRINT(.'TYPE ENTRY $');
294:          CALL PRINT(.'VARIABLE ENTRY $');
295:          CALL PRINT(.'PROCEDURE ENTRY $');
296:          CALL PRINT(.'FUNCTION ENTRY $');
297:          CALL PRINT(.'FILE ENTRY $');
298:          CALL PRINT(.'USER DECLARED ENTRY $');
299:     END;  /* CASE */
300: END WRITE$ENTRY;
301:
302:
303: PRINT$ID: PROC;
304:     DCL SIZE BYTE;
305:     CALL SETADDRPTR(6);
306:     SIZE = BYTEPTR;
307:     DO I = 1 TO SIZE;
308:          CALL SETADDRPTR(6+I);
309:          CALL PRINT$CHAR(BYTEPTR);
310:     END;
311:     CALL CRLF;
312: END PRINT$ID;
313:
314:
315: RANGER: PROC(A);
316:     DCL (A, BASE1) ADDR;
317:     BASE1 = A;
318:          CALL SET$PAST$PN(7);
319:          CALL CRLF;
320:          CALL TAB2;
321:          CALL PRINT(.'WITH LOW VALUE $');
322:          IF (SHR(FORM,7) AND FORMMASK) THEN
323:            CALL PRINT$CHAR(BYTEPTR);
324:          ELSE CALL PRINT$DEC(ADDRPTR);
```

```
325:          CALL PRINT(.' AND HIGH VALUE $');
326:          CALL SET$PAST$PN(9);
327:          IF (SHR(FORM,7) AND FORMMASK) THEN
328:            CALL PRINT$CHAR(BYTEPTR);
329:          FLSE CALL PRINT$DEC(ADDRPTR);
330: END RANGER;
331:
332:
333: USER$DEFINED: PROC;
334:       DO CASE (SHR(FORM,3) AND FORMMASK);
335:        DO;
336:          CALL PRINT(.'ENUMERATED TYPE - $');
337:          CALL PRINT$ID;
338:          CALL PRINT(.'THE VALUE IS $');
339:          CALL SET$PAST$PN(7);
340:          CALL PRINT$DEC(BYTEPTR);
341:        END;
342:        DO;
343:          DO CASE (SHR(FORM,6) AND FORMMASK);
344:            CALL PRINT(.'AN ENUMERATED SUBRANGE $');
345:            CALL PRINT(.'AN INTEGER SUBRANGE $');
346:            CALL PRINT(.'A CHARACTER SUBRANGE $');
347:          END; /* OF CASE */
348:          CALL RANGER(BASE);
349:        END;
350:        DO;
351:          CALL PRINT(.'AN ARRAY $');
352:          CALL SETADDRPTR(5);
353:          I = BYTEPTR;
354:          CALL CRLF;
355:          CALL TAB2;
356:          CALL PRINT(.'WITH COMPONENT TYPE $');
357:          CALL SETADDRPTR(10);
358:          DO CASE BYTEPTR;
359:            CALL PRINT(.'SCALAR $');
360:            CALL PRINT(.'INTEGER $');
```

```
361:            CALL PRINT(.'CHAR $');
362:            CALL PRINT(.'REAL $');
363:            CALL PRINT(.'STRING $');
364:            CALL PRINT(.'BOOLEAN $');
365:          END; /* OF CASE */
366:          CALL PRINT(.' AND REQUIRES $');
367:          CALL SETADDRPTR(8);
368:          CALL PRINT$DEC(ADDRPTR);
369:          CALL PRINT(.' BYTES OF STORAGE$');
370:          CALL CRLF;
371:          CALL TAB2;
372:          CALL PRINT(.'THERE IS/ARE $');
373:          CALL PRINT$DEC(I);
374:          CALL PRINT(.' DIMENSIONS IN THIS ARRAY $');
375:          CALL SETADDRPTR(9);
376:          DO WHILE I <> 0;
377:            APTRADDR = APTRADDR + 2;
378:            CALL RANGER(ADDRPTR);
379:            I = I - 1;
380:          END;
381:        END;
382:        DO;
383:        END;
384:        DO;
385:          CALL PRINT(.'A SET OF $');
386:          CALL SETADDRPTR(5);
387:          SAVEBASE = BASE;
388:          BASE = RESET$LOCATION(ADDRPTR);
389:          CALL PRINT$ID;
390:          BASE = SAVE$BASE;
391:        END;
392:        DO;
393:          CALL PRINT(.'A FILE OF $');
394:          CALL SETADDRPTR(5);
395:          SAVEBASE = BASE;
396:          BASE = RESET$LOCATION(ADDRPTR);
```

```
397:          CALL PRINT$ID;
398:          BASE = SAVEBASE;
399:        END;
400:        DO;
401:          CALL PRINT(.'A POINTER OF TYPE $');
402:          CALL SETADDRPTR(5);
403:          SAVE$BASE = BASE;
404:          BASE = RESET$LOCATION(ADDRPTR);
405:          CALL PRINT$ID;
406:          BASE = SAVE$BASE;
407:        END;
408:      END; /* OF CASE */
409: END USER$DEFINED;
410:
411:
412: CHECK$COLLISION: PROC;
413:      CALL SETADDRPTR(6);
414:      LPN = BYTEPTR;
415:      CALL TAB1;
416:      CALL PRINT(.'HASH VALUE = $');
417:      CALL SETADDRPTR(5);
418:      CALL PRINT$DEC(BYTEPTR);
419:      CALL SETADDRPTR(0);
420:      IF ADDRPTR = 00H THEN
421:           CALL PRINT(.' AND THERE ARE NO PREVIOUS COLLISIONS $');
422:      ELSE DO;
423:           DO WHILE ADDRPTR >= TABLE$START;
424:                APTRADDR = ADDRPTR;
425:                CALL PRINT(.' WHICH COLLIDES WITH $');
426:                CALL PRINT$ID;
427:                CALL SETADDRPTR(0);
428:                CALL CRLF;
429:                CALL TAB2;
430:           END;
431:           IF ADDRPTR = 00H THEN
432:                CALL PRINT(.' AND THERE ARE NO FURTHER COLLISIONS $');
```

```
433:          ELSE DO;
434:               CALL PRINT(.' ANY OTHER COLLISIONS OCCUR IN THE BUILT$');
435:               CALL PRINT(.'-IN SYMBOL TABLE WHICH IS INACCESSABLE $');
436:          END;
437:     END;
438:     CALL CRLF;
439: END CHECK$COLLISION;
440:
441:
442: ENTRY$HEAD: PROC;
443:     CALL WRITE$ENTRY;
444:     CALL PRINT$ID;
445:     CALL CHECK$COLLISION;
446:     CALL TAB1;
447: END ENTRY$HEAD;
448:
449:
450: CHECK$TYPE: PROC(A);
451:     DCL A BYTE;
452:     DCL TYPE BYTE;
453:     TYPE = (SHR(A, 3) AND FORMMASK);
454:     DO CASE TYPE;
455:          /* SCALAR-ORDINATE */
456:          CALL PRINT(.' SCALAR ORDINATE $');
457:          /* INTEGER */
458:          CALL PRINT(.' INTEGER $');
459:          /* CHARACTER */
460:          CALL PRINT(.' CHARACTER $');
461:          /* REAL */
462:          CALL PRINT(.' REAL $');
463:          /* COMPLEX */
464:          DO;
465:            SAVE$BASE = BASE;
466:            CALL SET$PAST$PN(9);
467:            BASE = RESET$LOCATION(ADDRPTR);
468:            CALL SETADDRPTR(4);
```

```
469:              IF (BYTEPTR AND FORM$MASK) = 07H THEN
470:                CALL USER$DEFINED;
471:              ELSE CALL PRINT$ID;
472:              BASE = SAVE$BASE;
473:            END;
474:            /* BOOLEAN */
475:            CALL PRINT(.'BOOLEAN $');
476:      END;    /* CASE TYPE */
477:      CALL CRLF;
478:      CALL TAB1;
479: END CHECK$TYPE;
480:
481:
482: PRINT$PRT: PROC(A);
483:      DCL A BYTE;
484:      IF A = 12 THEN
485:           CALL PRINT(.'THE ASSIGNED PRT LOCATION FOR THE SBP IS $');
486:      ELSE CALL PRINT(.'THE ASSIGNED PRT LOCATION IS $');
487:      CALL SET$PAST$PN(A);
488:      CALL PRINT$DEC(ADDRPTR);
489:      CALL CRLF;
490: END PRINT$PRT;
491:
492:
493: PRINT$LABEL: PROC;
494:      CALL ENTRY$HEAD;
495:      CALL PRINT(.'THE ASSIGNED LABEL VALUE IS $');
496:      CALL SET$PAST$PN(7);
497:      CALL PRINT$DEC(ADDRPTR);
498:      CALL CRLF;
499: END PRINT$LABEL;
500:
501:
502: PRINT$CONST: PROC;
503:      CALL ENTRY$HEAD;
504:      CALL PRINT(.'THE CONSTANT TYPE IS $');
```

```
505:      CALL PRINT(.'THE CONSTANT VALUE = $');
506:      CALL SET$PAST$PN(7);
507:      CALL PRINT$DEC(ADDRPTR);
508:  END PRINT$CONST;
509:
510:
511:  PRINT$TYPE: PROC;
512:      CALL ENTRY$HEAD;
513:      CALL PRINT(.'THE PARENT TYPE IS $');
514:      DO CASE (SHR(FORM,3) AND FORMMASK);
515:        CALL PRINT(.'INTEGER $');
516:        CALL PRINT(.'REAL $');
517:        CALL PRINT(.'CHAR $');
518:        CALL PRINT(.'BOOLEAN $');
519:        DO;
520:          CALL SET$PAST$PN(7);
521:          SAVE$BASE = BASE;
522:          BASE = RESET$LOCATION(ADDRPTR);
523:          CALL SETADDRPTR(4);
524:          IF (BYTEPTR AND FORMMASK) = 07H THEN
525:            CALL USER$DEFINED;
526:          ELSE CALL PRINT$ID;
527:          BASE = SAVE$BASE;
528:        END;
529:      END;  /* OF CASE */
530:  END PRINT$TYPE;
531:
532:
533:  PRINT$VARIABLE: PROC;
534:      CALL ENTRY$HEAD;
535:      CALL PRINT(.'THE VARIABLE TYPE IS $');
536:      CALL CHECK$TYPE(FORM);
537:      CALL PRINT$PRT(7);
538:  END PRINT$VARIABLE;
539:
540:
```

```
SUBROUTINE: PROC;
    DCL J BYTE;
    CALL PRINT(.'THERE ARE $');
    CALL SET$PAST$PN(7);
    J = BYTEPTR;
    CALL PRINT$DEC(BYTEPTR);
    CALL PRINT(.' PARAMETERS $');
    CALL CRLF;
    CALL SET$PAST$PN(8);
    PARM$LISTING(SUBRTN:=SUBRTN+1), APTRADDR = RESET$LOCATION(ADDRPTR);
    PARM$NUM(SUBRTN) = J;
    DO I = 1 TO J;
        CALL TAB2;
        CALL PRINT(.'NO. $');
        CALL PRINT$DEC(I);
        CALL TAB1;
        IF SHR(BYTEPTR,7) THEN
        DO;
            IF SHR(BYTEPTR,6) THEN CALL PRINT(.' FUNCTION $');
            ELSE CALL PRINT(.' VAR $');
        END;
        ELSE IF BYTEPTR = 4 THEN CALL PRINT(.' PROCEDURE $');
            ELSE CALL PRINT(.' VALUE $');
        CALL PRINT(.'PARAMETER OF TYPE $');
        CALL CHECK$TYPE(FORM);
        APTRADDR = APTRADDR + 3;
    END; /* DO I */
    CALL PRINT$PRT(10);
    CALL PRINT$PRT(12);
    CALL TAB1;
    CALL PRINT(.'THE LABEL VALUE PRECEDING THE CODE IS $');
    CALL SET$PAST$PN(14);
    CALL PRINT$DEC(ADDRPTR);
    CALL CRLF;
END SUBROUTINE;
```

```
577:
578: BRANCH: PROC;
579:     SBTBL = SBTBL + (3 * PARM$NUM(SUBRTN));
580:     SUBRTN = SUBRTN - 1;
581: END BRANCH;
582:
583:
584: PRINT$PROC: PROC;
585:     CALL ENTRY$HEAD;
586:     CALL SUBROUTINE;
587: END PRINT$PROC;
588:
589:
590: PRINT$FUNC: PROC;
591:     CALL ENTRY$HEAD;
592:     CALL PRINT(.'THE FUNCTION TYPE IS $');
593:     CALL SET$PAST$PN(16);
594:     FORM = BYTEPTR;
595:     CALL CHECK$TYPE(FORM);
596:     CALL SUBROUTINE;
597: END PRINT$FUNC;
598:
599:
600: PRINT$FILE: PROC;
601:     CALL ENTRY$HEAD;
602: END PRINT$FILE;
603:
604:
605: SKIPPER: PROC;
606:     DO CASE(SHR(FORM,3) AND FORMMASK);
607:       DO;
608:         CALL SETADDRPTR(6);
609:         SBTBL = SBTBL + 10 + BYTEPTR;
610:       END;
611:       SBTBL = SBTBL + 16;
612:       DO;
```

```
613:          CALL SETADDRPTR(5);
614:          SBTBL = SBTBL + 10 + (2 * BYTEPTR);
615:        END;
616:        DO;
617:          IF FORM = 1FH THEN SBTBL = SBTBL + 9;
618:          ELSE DO;
619:            CALL SETADDRPTR(6);
620:            SBTBL = SBTBL + 14 + BYTEPTR;
621:          END;
622:        END;
623:        SBTBL = SBTBL + 7;
624:        SBTBL = SBTBL + 7;
625:        SBTBL = SBTBL + 7;
626:      END;  /* OF CASE */
627: END SKIPPER;
628:
629:
630: STARS: PROC;
631:      CALL CRLF;
632:      CALL PRINT(.'**************************************************$');
633:      CALL CRLF;
634: END;
635:
636:
637:
638: BASE, SBTBL = .MEMORY;
639: CALL COPY$SBTBL; /* PLACE SYMBTBL AT TOP OF MEMORY */
640: CALL SETADDRPTR(4);
641: FORM = BYTEPTR;
642: DO CASE (FORM AND FORMMASK);
643:      CALL SET$PAST$PN(11);
644:      DO;
645:        CALL SETADDRPTR(4);
646:        IF SHR(BYTEPTR,4) THEN CALL SET$PAST$PN(17);
647:        ELSE CALL SET$PAST$PN(11);
648:      END;
```

```
649:      CALL SET$PAST$PN(11);
650:      CALL SET$PAST$PN(13);
651:      CALL SET$PAST$PN(18);
652:      CALL SET$PAST$PN(19);
653:      CALL SET$PAST$PN(9);
654:      ; /* THIS ENTRY IS IMPOSSIBLE FOR THE FIRST ENTRY */
655: END; /* CASE FORM */
656: /* SET THE VALUE FOR THE STARTING LOCATION OF THE SYMBOL TABLE */
657: TABLE$START = ADDRPTR;
658: /* START */
659: CALL SETADDRPTR(2);
660: DO WHILE ADDRPTR <> 00H;
661:      CALL SETADDRPTR(4);
662:      FORM = BYTEPTR;
663:      CALL STARS;
664:      DO CASE (BYTEPTR AND FORMMASK);
665:      /* LABEL */
666:           DO;
667:                CALL PRINT$LABEL;
668:                SBTBL = SBTBL + 9 + LPN;
669:           END;
670:           /* CONSTANT */
671:           DO;
672:                CALL PRINT$CONST;
673:                SBTBL = SBTBL + 9 + LPN;
674:           END;
675:           /* TYPE */
676:           DO;
677:                CALL PRINT$TYPE;
678:                SBTBL = SBTBL + 9 + LPN;
679:           END;
680:           /* VARIABLE */
681:           DO;
682:                CALL PRINT$VARIABLE;
683:                SBTBL = SBTBL + 11 + LPN;
684:           END;
```

```
685:            /* PROCEDURE */
686:            DO;
687:                 CALL PRINT$PROC;
688:                 SBTBL = SBTBL + 16 + LPN;
689:            END;
690:            /* FUNCTION */
691:            DO;
692:                 CALL PRINT$FUNC;
693:                 SBTBL = SBTBL + 17 + LPN;
694:            END;
695:            /* FILE */
696:                 DO;
697:                 CALL PRINT$FILE;
698:                 SBTBL = SBTBL + 7 + LPN;
699:            END;
700:            /* USER DEFINED ENTRY */
701:            DO;
702:              CALL SKIPPER;
703:            END;
704:       END; /* OF CASE */
705:       IF SBTBL = PARM$LISTING(SUBRTN) THEN CALL BRANCH;
706:       BASE = SBTBL;
707:       CALL SETADDRPTR(2);
708: END;
709: CALL CRLF;
710: CALL PRINT(.'THE CONTENTS OF THE SYMBOL TABLE HAVE BEEN PRINTED. $');
711: GO TO BOOT;
712: EOF
```

LIST OF REFERENCES

1. Gracida, J.C. and Stilwell, R.R., NPS-PASCAL: A Partial Implementation of PASCAL Language for a Microprocessor-based Computer System, Master's Thesis, Naval Postgraduate School, Monterey, Ca., June 1978.

2. BSI DPS/13/4 Working Group, "The BSI/ISO Working Draft of Standard Pascal," Pascal News, number 14, p. 4-54, January 1979.

3. Strutynski, Kathryn B., CPM80 User's Guide, internally distributed technical note.

4. W. R. Church Computer Center TN No. 0141-31, PL/M 8080 Compiler User's Guide at NPS, Bernadette Peavey, March 1977.

5. University of Toronto, Computer Systems Research Group Technical Report CSRG-2, "An Efficient LALR Parser Generator," by W. R. Lalonde, April 1971.

6. UCSD (Mini-Micro Computer) Pascal, Revised Version 1.5, Institute for Information Systems, Regents of the University of California, San Diego Campus, La Jolla, Ca., August 1978.

7. Jensen, K., and Wirth, N., "Pascal User Manual and Report," 2nd ed., Springer-Verlag, New York-Heidelberg-Berlin, 1974.

8. NPS-PASCAL User's Manual (to be published).

9. Flynn, J.P. and Moranville, M.S., ALGOL-M An Implementation of a High-level Block Structured Language for a Microprocessor-based Computer System, Master's Thesis, Naval Postgraduate School, Monterey, Ca., September 1977.

10. Naval Postgraduate School Report NPS-53KD72 11A, ALGOL-E: An Experimental Approach to the Study of Programming Languages, by Gary A. Kildall, 7 January 1972.

11. Intel Corporation, 8080/8085 Assembly Language Programming Manual, 1977.

12. Eubanks, Gordon E. Jr., A Microprocessor Implementation of Extended Basic, Master's Thesis, Naval Postgraduate School, December 1976.

INITIAL DISTRIBUTION LIST

		No. Copies
1.	Defense Documentation Center Cameron Station Alexandria, Virginia 22314	2
2.	Library, Code 0142 Naval Postgraduate School Monterey, California 93940	2
3.	Department Chairman, Code 52 Department of Computer Science Naval Postgraduate School Monterey, California 93940	4
4.	Assoc Professor Gary A. Kildall, Code 52Kd Department of Computer Science Naval Postgraduate School Monterey, California 93940	1
5.	LT Mark S. Moranville, USN, Code 52Mv Department of Computer Science (less p. 117-282) Naval Postgraduate School Monterey, California 93940	1 25
6.	Microcomputer Laboratory, Code 52ec Department of Computer Science Naval Postgraduate School Monterey, California 93940	2
7.	MAJ Joaquin C. Gracida, USMC 209 So. Adams Street Arlington, Virginia 22204	1
8.	LT Robert R. Stilwell, SC, USN NSD Guam Code 60 FPO San Francisco 96630	1
9.	LCDR Antonio L. S. Goncalves, Brazilian Navy Rua Prudente de Moraes, 660 Apt. 202 Ipanema, Rio de Janeiro 20000 RJ Brazil	1
10.	LT(JG) Javier E. De La Cuba, Peruvian Navy Direccion de Instruccion de la Marina Ministerio de Marina Ave. Salaverry S/N Lima, PERU	1

11. LT John L. Byrnes, USN 6
Class 63, SWOSCOLCOM
Bldg. 446
Newport, Rhode Island 02840