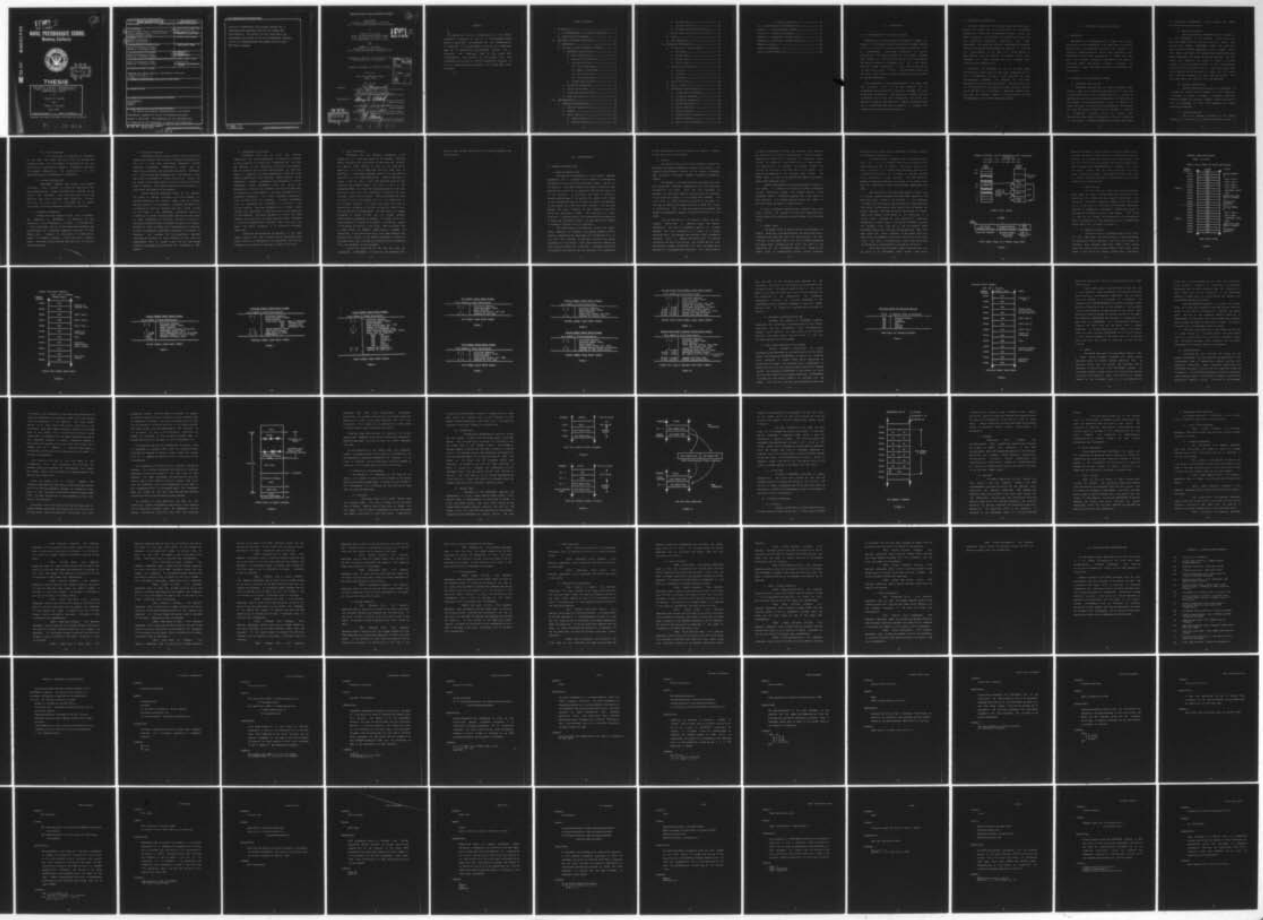
AD-A061 040 NAVAL POSTGRADUATE SCHOOL MONTEREY CALIF F/G 9/2
NPS-PASCAL. A PARTIAL IMPLEMENTATION OF PASCAL LANGUAGE FOR A M--ETC(U)
JUN 78 J C GRACIDA, R R STILWELL

UNCLASSIFIED NL

1 OF 3
AD A061040



DDC FILE COPY AD A061040

LEVEL II

(2)

NAVAL POSTGRADUATE SCHOOL

Monterey, California

DDC
RECEIVED
NOV 08 1978
E

THESIS

NPS-PASCAL: A Partial Implementation of PASCAL Language for a Microprocessor-based Computer System

by

Joaquin C. Gracida

and

Robert R. Stilwell

June 1978

Thesis Advisor: Gary A. Kildall

Approved for public release; distribution unlimited

78 10 30 068

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)

(6) NPS-PASCAL: A Partial Implementation of PASCAL Language for a Microprocessor-based Computer System

5. TYPE OF REPORT & PERIOD COVERED

(9) Master's Thesis; June 1978

6. PERFORMING ORG. REPORT NUMBER

7. AUTHOR(s)

(10) Joaquin C. Gracida
Robert R. Stilwell

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

(11) 12. REPORT DATE

June 1978

13. NUMBER OF PAGES

14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office)

Naval Postgraduate School
Monterey, California 93940

15. SECURITY CLASS. (of this report)

Unclassified

15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

(12) 228 p.

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)

The design and partial implementation of the PASCAL programming language for use on a microprocessor-based system is described. The framework for the implementation is comprised of two subsystems, a compiler which generates

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE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

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

251 450

code for a hypothetical zero-address machine and a translator that generates code for the target 8080 microcomputer. The portions of the system which are implemented are written in the PL/M programming language to run in a diskette-based environment with at least 32K bytes of memory.

Approved for public release; distribution unlimited

(2)

NPS-PASCAL
A Partial Implementation of PASCAL for a Microprocessor-based Computer System

LEVEL II

by

Joaquin C. Gracida
Major, United States Marine Corps
B.S., University of Idaho, 1966
M.A., Pepperdine University, 1976

and

Robert R. Stilwell
Lieutenant, United States Navy
B.S., United States Naval Academy, 1971

Submitted in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
JUNE, 1978

ACCESSION for
NTIS White Section
DDC Buff Section
UNANNOUNCED
JUSTIFICATION
BY
DISTRIBUTION/AVAILABILITY CODES
Dist. SPECIAL
A

Authors: _____

_____

Approved by: _____
Thesis Advisor

_____
Second Reader

_____
Chairman, Department of Computer Science

_____
Dean of Information and Policy Sciences

DDC
RECEIVED
NOV 08 1978
E

78 10 30 068

ABSTRACT

The design and partial implementation of the PASCAL programming language for use on a microprocessor-based system is described. The framework for the implementation is comprised of two subsystems, a compiler which generates code for a hypothetical zero-address machine and a translator that generates code for the target 8080 microcomputer. The portions of the system which are implemented are written in the PL/M programming language to run in a diskette-based environment with at least 32K bytes of memory.

TABLE OF CONTENTS

I. INTRODUCTION

A. AN INTRODUCTION TO THE PASCAL LANGUAGE

PASCAL was the first programming language to embody the concept of structured programming defined by Edsger Dykstra and C. A. R. Hoare [7], and was developed by Nicklaus Wirth at Eigenossisch Technische Hochschule in Zurich, Switzerland. Preliminary versions of the PASCAL language were drafted in 1968 following the spirit of the ALGOL-60 and ALGOL-W programming languages [10]. The first PASCAL compiler became operational in 1970 and its publication followed a year later. In 1973, a revised PASCAL report was published consolidating revisions resulting from two years of use and experience.

The development of the language PASCAL was based upon two principal aims: to be more powerful than its predecessors and to provide a suitable language to teach structured programming. The extensions of PASCAL relative to ALGOL-60 lie in the data structuring facilities that expand its range of applicability. PASCAL introduces record and file structures that make it possible to program both commercial and scientific applications.

8. OBJECTIVES OF NPS-PASCAL

The major objective of the project described here was to provide the basis for an implementation of the PASCAL language on an Intel 8080 microcomputer system. PASCAL was chosen because of its flexible type declarations as well as its structured programming constructs. The rapid development and decreasing costs of microcomputer hardware, coupled with sophisticated compatible software is allowing the microcomputer to be used in a large number of applications. The availability of another high level language for these systems can only increase their usefulness and acceptability.

NPS-PASCAL was developed to run on an 8080 based microcomputer system using the high level language PL/M [9]. PL/M is implemented through a cross compiler for 8080 microprocessor systems, and executes on the Naval Postgraduate School's IBM 360 computer. The availability of the 8080 based CP/M disk operating system simulator (CPSYM) on the IBM 360, with its powerful debugging capabilities, was also an important factor in the choice of the 8080 microprocessor and the CP/M operating system.

II. NPS-PASCAL LANGUAGE DESCRIPTION

A. BACKGROUND

NPS-PASCAL is an implementation of PASCAL with slight deviations to allow NPS-PASCAL to be specified by an LALR(1) grammar form. This permitted the use of the University of Toronto's compiler-compiler parse table generator [15]. NPS-PASCAL allows simple conversions to PASCAL and back which will become increasingly important as the number of available PASCAL application programs increases. The differences between the PASCAL structure and NPS-PASCAL are given below.

B. FEATURES OF THE NPS-PASCAL LANGUAGE

1. NPS-PASCAL Declarations

NPS-PASCAL requires that all labels, constants, user defined types, and variables be declared prior to their use in the program body. This is accomplished by a series of declarations and definitions in sequence at the beginning of the program. The program heading is used to declare the input and output files which can be accessed by the program. The input and output declarations are discussed in section 5. Procedures functions, which may contain local declarations, must also be defined prior to their invocation in the program. Although procedure and function calls were

not implemented in NPS-PASCAL, it was intended that these structures would be recursive.

a. Label Declarations

Labels are used by NPS-PASCAL as the target of all GOTO statements in the program. Labels can be any positive integer value up to thirty digits in length. This differs from PASCAL. NPS-PASCAL treats the label as a identifier while PASCAL recognizes it as an integer value. When labels are used in the program their declaration must appear immediately following the program heading. The only factor that restricts the number of labels in a given program is the available memory of the microcomputer executing the NPS-PASCAL compiler. Sufficient memory must be available after the compiler is loaded for the symbol table entries generated by each label declaration. This size restriction also applies to all declarations.

b. Constant Declarations

Constant declarations enable the programmer to introduce an identifier as a synonym for a valid constant. A constant is either a number, a constant identifier (possibly signed) or a string. Constant identifiers usually allow a programmer to write more readable and self-documenting programs.

c. Type Declarations

One of the greatest strengths of the PASCAL language is the ability to define unique data types. In

NPS-PASCAL there are four standard types: Boolean, Integer, Char (character), and Real. Integers may be any value between -32,768 and +32,767. Real values are represented internally in an exponential format and can take on any positive or negative value consisting of fourteen digits multiplied by ten to the -64th power through ten to the +63rd power. Characters may be any valid ASCII character. The two identifier constant values of TRUE and FALSE can be assigned to a variable of type Boolean.

In addition to the above scalar types, there exists the capability of declaring a user defined scalar type. A series of identifiers in sequence can be given an identifying name, making up this user defined type. A type may also be defined as a subrange of any previously defined scalar type, except Boolean and real, by indicating the smallest and largest value in the subrange. The order in which the user defined scalar type identifiers are declared determines the highest and lowest values of that type. The first identifier becomes the lowest value of that type, while the last becomes the highest.

Structured types are defined by describing the individual types of their components and by indicating which structuring method is to be applied. This determines each component's location in the structure. NPS-PASCAL supports the four type structures of PASCAL: array structures, record structures, set structures, and file structures.

(1) Array Structures

Array structures contain two or more components of the same declared type. Arrays are indexed with up to five dimensions. Each dimension is defined by a user defined scalar type or a subrange of the integers. Components may be declared as any valid type. If arrays are nested (the components of the array are arrays) then a maximum of five levels are allowed.

(2) Record Structures

Unlike the array structure, the record structure's components, called fields, are not necessarily of the same type. Each field in the record structure must be assigned a valid type, which possibly could be another record. Nesting of records is allowed up to four levels of declaration. Identifiers are associated with each field, and are used for selection rather than a computable index like that of an array. Records may vary in length with the restriction that the variable part of each record must be the last portion of that record. The variant part of a record cannot contain another record with a variant part.

(3) Set Structures

The set structure defines a set of values, which is the power set of a declared base type. The base type is usually a user defined scalar type or a subrange of the type integer. The maximum number of elements in the set cannot exceed sixteen including the null set.

(4) File Structures

A file structure is a sequence of components of the same type whose position in the file defines the element's order. Only one element is accessible at any one time during execution of a program. Subsequent items must be accessed sequentially. The manipulation of file structures was not implemented in NPS-PASCAL.

d. Variable Declarations

NPS-PASCAL supports both static and dynamic variables. Every variable is bound to a particular type during declaration. Dynamic variables are referred to as pointer type variables. They each may point to only one declared type which provides data protection. Pointer variables may also occur in structures, as in record structures, which are themselves dynamically generated.

2. Arithmetic Processing

Integer and binary coded decimal (BCD) arithmetic are supported by NPS-PASCAL. In addition, the set operations in (set membership), + (union), * (intersection), - (set difference), and all of the relational operators are also defined. The relational operators provided in NPS-PASCAL are: = (equal), <> (not equal), <= (less than or equal), >= (greater or equal), < (less than), and > (greater than). The three logical Boolean operators AND, OR, and NOT are also defined.

3. Control Structures

NPS-PASCAL employs several useful control structures consisting of BEGIN - END blocks for compound statements, IF THEN and IF THEN ELSE conditional statements, CASE-OF selective statements, REPEAT-UNTIL, WHILE, and FOR repetitive statements, and procedure statements. NPS-PASCAL is a block structured language, much like ALGOL-60 in that each block is bracketed by a BEGIN and an END. Blocks may be nested within other blocks up to and including the tenth level of nesting. This nesting restriction also applies to conditional and repetitive statements.

Unlike ALGOL-60, BEGIN-END blocks do not restrict the scope of defined variables within the program. All variable procedure and function names must be uniquely declared globally in the program. Variables declared within a procedure or function, however, have their scope limited to the range of that procedure. Procedure nesting is allowed with the same variable scope restrictions applicable to inner procedures or functions. Storage for statically declared variables remains allocated throughout the program. Storage for local variables within procedures and functions, as well as dynamically generated variables, is allocated as required. The storage for the dynamically assigned variables is returned to free storage at run time for reassignment when no longer in use. The run time storage allocation management routines were not implemented in NPS-PASCAL.

4. Procedures And Functions

Statements which can be given user defined identifiers are called procedures. A procedure is invoked in the program by its identifier. Procedures and functions are declared in the declaration portion of the program. Each may contain local variable declarations, as explained above, in addition to formal parameters. There are three types of acceptable formal parameter specifications: value parameters, variable parameters, and procedure or function parameters. Value parameters (call by value) are expressions in the calling statement that are evaluated once at activation of the procedure. The actual parameter is the evaluated expression. The formal parameter represents a local variable within the procedure. Variable parameters (call by reference) are addresses of variables in the calling program. The actual parameter is a variable whose indices are evaluated, if required, prior to the execution of the procedure. The formal parameter is given the same address and must be of the same type when used. Procedure or function parameters are evaluated each time they are used. The actual parameter is a function or procedure identifier.

Functions and procedures are declared in the same manner except that each function has an associated type. When a function is referenced in an expression, the function produces a value which is returned in place of the call.

5. Input And Output

NPS-PASCAL has four separate statements which accomplish all input and output for the program. READ and READLN statements are used either to read from the console, the default input device, or from a file when one is specified. If a file is specified, the file name must be declared in the program heading, and the file type must be specified in a file type declaration. These rules apply to output files as well. Text files declared to be of type character are a special case. For this, a call to the procedure READLN will read the characters specified and then skip to the next line (the character following the next carriage return and line feed). WRITE and WRITELN are used to either write to the console, or to a file specified in the program heading. The data to be read or written is specified by variables or strings in quotation marks which are enclosed within parentheses. Any combination of integer variables or quoted strings may be placed between parentheses and separated by commas. A a quoted string in a read statement is treated as a comment. The maximum number of characters printed on a line is 80. When the output from a given WRITE, or WRITELN specification reaches 80 characters, a carriage return and a line feed character are automatically issued. If the WRITELN statement is used, the carriage return and line feed are generated after processing the final parameter in the statement.

The built-in functions of PUT and GET were not implemented in NPS-PASCAL. In addition, the predicate EOF,

which is used to mark the end of a file being processed, was also omitted.

III. IMPLEMENTATION

A. COMPILER IMPLEMENTATION

1. Compiler Organization

The compiler was designed to read source language statements from a diskette and to produce an intermediate language file while printing an optional source listing at the console. A one pass approach was used to provide a fast compilation, as well as to reduce the required work and size of the compiler. To eliminate the need to perform a complete second pass of the source file, labels are placed in the intermediate code at the position where the execution of the program is to continue after a branch. This was done because the exact location of each branch is not known during the compilation phase. The resolution of label locations is accomplished by the code generating program as it scans the intermediate code. The code generating program is hereafter refered to as the translator.

The single pass of the compiler builds the symbol table, converts all numbers in the source program to their internal representation, generates the intermediate file on the disk, and provides an optional listing of the source statements at the console. Token and production numbers are also listed for each line if desired. If program errors are anticipated, the compiler can also surpress the generation

of the intermediate file by the setting of specific toggles at the beginning of the program.

2. Scanner

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

The scanner is divided into four main sections which are selectively executed depending on the first non-blank character of the token. After the first character is scanned, the individual section involved scans the remainder of the token and places it in the accumulator (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 which permits the scanner and parser to subsequently accept the remaining portion of the token.

The four sections of 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 process 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 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 position of the word in the VOCAB table to the parser. This position corresponds to the assigned token number. The VOCAB table is one of the tables provided by the LALR(1) parse table generator [15].

Special characters are handled separately, except in two cases. If a period is followed by numeric characters without intervening spaces, the program section which processes special characters assumes that a real number is being scanned. This program section handles the number in the same manner as does the number section.

If 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.

3. Symbol Table

The symbol table is used to store the attributes of labels, constants, type declarations, variable identifiers, procedures and functions, and file declarations. The main function of the symbol table is to verify program semantics and table symbol characteristics which are used in the generation of the intermediate code file. Access to the symbol table is accomplished through various primitive

subroutines using based global variables to uniquely address the elements of each entry.

The symbol table is modeled after the ALGOL-M symbol table [5]. It is an unordered linked list of entries which grows toward the top of memory. Individual entries are either accessed via a chained hash addressing technique as illustrated in Figure 1, or by means of address pointer fields contained in other entries. The later method of access is required since not all entries in the symbol table have an identifier, called the printname, associated with them.

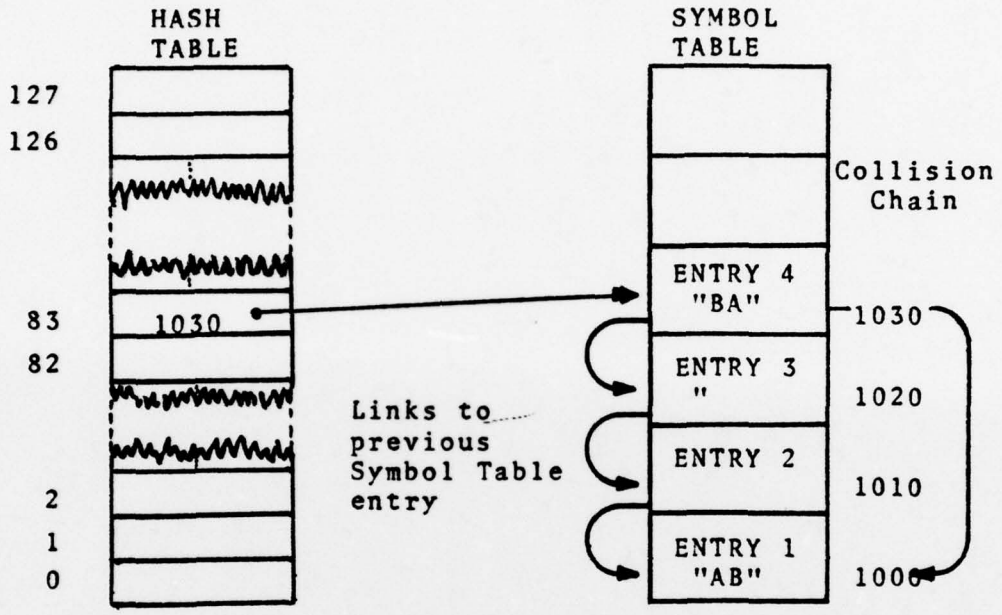
Each location in the hash table heads a linked list of entries whose printname, when evaluated, results in the same hash value. A zero in any location in the hashtable indicates that there are no entries whose printname produces that value. During symbol table construction or access, the global variable 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, 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.

There are eight different types of entries that can be found in the NPS-PASCAL symbol table. Each entry

HASHING FUNCTION: SUM OF PRINTNAMES ASCII CHARACTERS MODULO 128

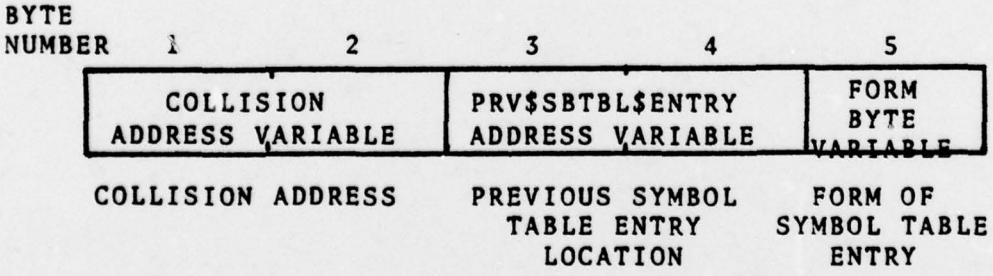
H.F.(AB) = (41 + 42) MOD 128 = 83

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



SYMBOL TABLE ACCESS

FIGURE 1



FIRST THREE FIELDS OF A SYMBOL TABLE ENTRY

FIGURE 2

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 (first two bytes), the previous symbol table entry address field (PRV$SBTBL$ENTRY - located in the third and fourth bytes), and the form field (FORM - the fifth byte), as shown on Figure 2. The remaining fields are used to uniquely describe each entry's attributes and particular identifying characteristics.

a. Label Entries

The form field of a label entry has the constant byte value of zero. A single byte follows the form field containing the length of the label's printname. 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 3.

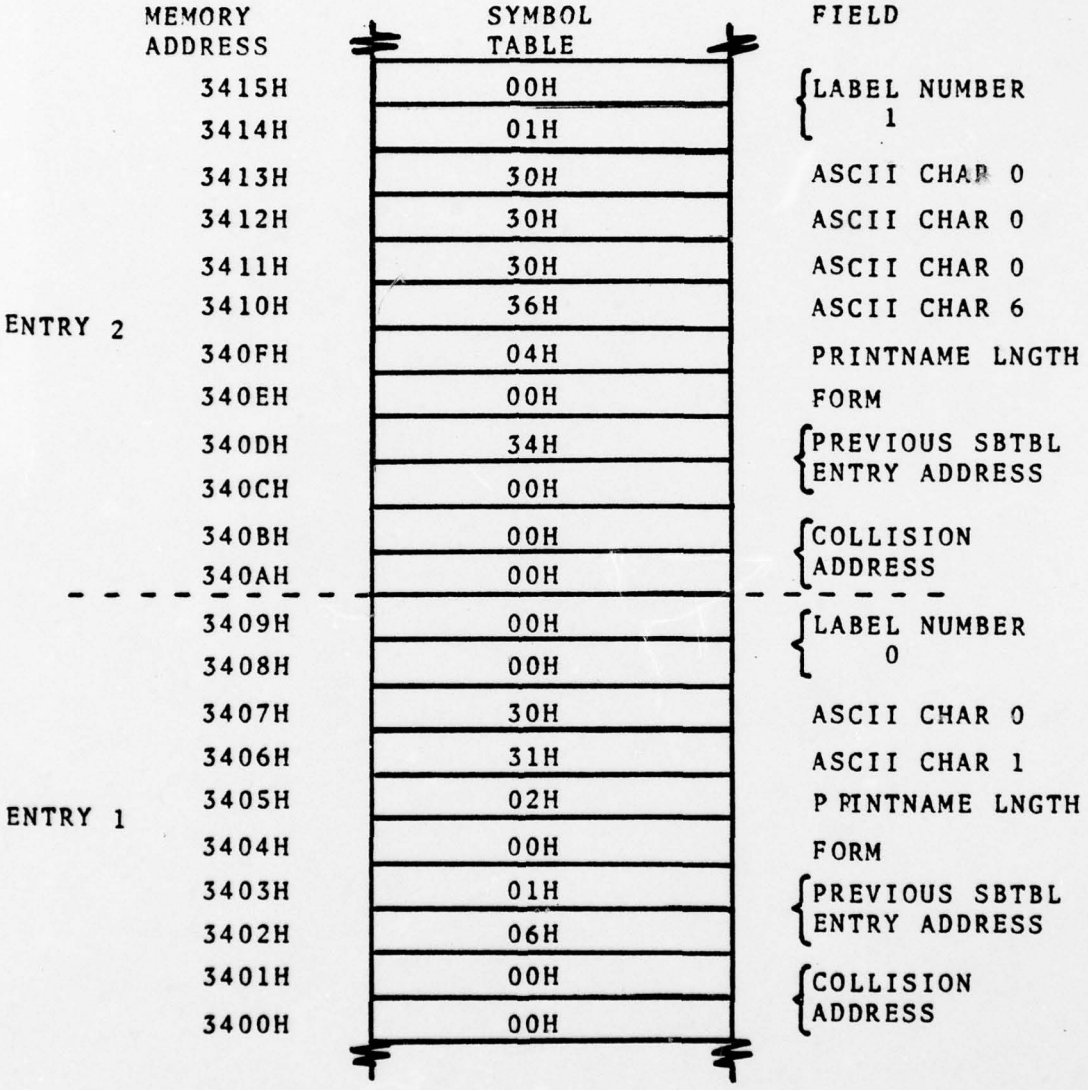
b. Constant Entries

The form field of a constant symbol table entry not only identifies the type of entry, it also designates the particular type of constant. The five valid types of constants are: unsigned identifier (FORM = 01H), signed identifier (FORM = 41H), integer (FORM = 09H), real value (FORM = 11H), and string constant (FORM = 19H). Each entry

EXAMPLE LABEL DECLARATION:

```
LABEL  10, 6000;
```

SYMBOL TABLE ENTRY FOR ABOVE DECLARATION



LABEL ENTRY SBTBL

FIGURE 3

in the symbol table has a unique three bit code in its one byte form field. The three bit code for constant entries, for example, is 001. The remaining bits in the FORM variable describe the particular characteristics of the type involved.

Following the form field of the constant entry are the printname length field and the printname characters. The value of the constant follows the printname characters. The value field may consist of another length field and 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).

c. Type Entries

There are two kinds of type entries that can be found in the NPS-PASCAL symbol table, a simple type entry or a type declaration entry. Simple type entries either indicate that a basic type is being assigned, or that a defined complex type declaration is to be evaluated. In the later case, the simple type entry will contain a pointer to a type declaration entry. Type declaration entries are generated from user defined types found in the source program. In some cases, a chain of type declarations can be defined. An example of this would be an array of the type array which is itself of type integer.

The form field of a simple type entry indicates which basic type is being entered or accessed. 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 the entry. The field following the form is a one byte field containing the length of the printname, 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 found in Figure 4.

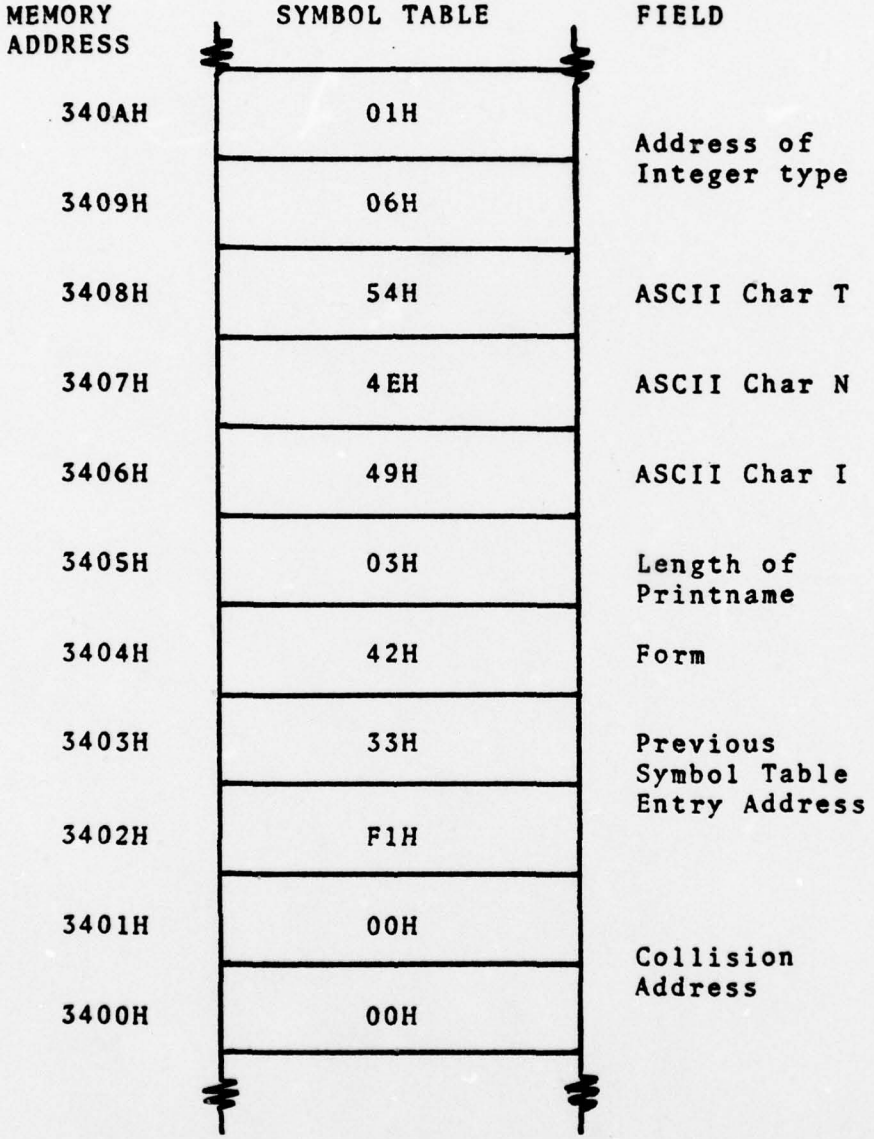
A type declaration entry is constructed for any of the seven different user definable types in NPS-PASCAL, consisting of scalar types, subrange types, array types, record types, set types, file types, and pointer types. Only the scalar entry contains an accessible printname. The rest of the entries must be accessed via a pointer field found in other entries. The format of these type declaration entries is found in Tables 1 through 7.

d. Variable Entries

The form field of the variable entry contains a value which describes the type of the variable. The values of the FORM and their associated types are shown in Table 8. Following the form field are the fields which contain the variable identifier's printname length and printname characters. A two byte field which contains the address of the variable's starting address in memory appears after the printname characters. This address is a relative offset

Simple Type Entry Example:

TYPE INT = INTEGER;



SYMPLE TYPE SYMBOL TABLE ENTRY

FIGURE 4

Scalar Symbol Table Entry Format

Byte Number	Field Description
0 - 1	Collision Address
2 - 3	Previous Symbol Table Entry Address
4	Form Field (Constant Value 07H)
5	Printname Length (LPN)
6 - 5+LPN	ASCII Representation of Printname
6+LPN	Ordinate Number Of Scalar Ident.
7+LPN - 8+LPN	Address Of Parent Type

SCALAR SYMBOL TABLE ENTRY FORMAT

TABLE 1

Subrange Symbol Table Entry Format

Byte Number	Field Description
0 - 1	Collision Address
2 - 3	Previous Symbol Table Entry Address
4	Form Field (Value Type of Subrange) 0FH Ordinate Elmt. 4FH Integer Elmt. 8FH Character Elmt.
5 - 6	Parent Address Field
7 - 8	Low Value Of Range
9 - 10	High Value Of Range
11 - 12	Difference + 1

SUBRANGE SYMBOL TABLE ENTRY FORMAT

TABLE 2

Array Symbol Table Entry Format

Byte Number	Field Description
0 - 1	Collision Address
2 - 3	Previous Symbol Table Entry Address
4	Form Field(Const. Val. 17H)
5	Number Of Dimensions
6 - 7	Address Of Component Type
8 - 9	Total Storage Required (No. Bytes)
10	Basic Type Of Components Value Type 00H Ordinate 01H Integer 02H Character 03H Real 04H Complex 05H Boolean
11 - 12	Address Of Dimension 1
13 - 14	Address Of Dimension 2
.	
.	

ARRAY SYMBOL TABLE ENTRY FORMAT

TABLE 3

Set Symbol Table Entry Format

Byte Number	Field Description
0 - 1	Collision Address
2 - 3	Previous Symbol Table Entry Address
4	Form Field (Const. Val 27H)
5 - 6	Address Of Set Type

SET SYMBOL TABLE ENTRY FORMAT

TABLE 4

File Symbol Table Entry Format

Byte Number	Field Description
0 - 1	Collision Address
2 - 3	Previous Symbol Table Entry Address
4	Form Field (Const. Val. 2FH)
5 - 6	Address Of File Type

FILE SYMBOL TABLE ENTRY FORMAT

TABLE 5

Pointer Symbol Table Entry Format

Byte Number	Field Description
0 - 1	Collision Address
2 - 3	Previous Symbol Table Entry Address
4	Form Field (Const. Val 37H)
5 - 6	Address Of Pointer Type

POINTER SYMBOL TABLE ENTRY FORMAT

TABLE 6

Record Symbol Table Entry Format

Byte Number	Field Description
0 - 1	Collision Address
2 - 3	Previous Symbol Table Entry Address
4	Form Field (Const. Val. 1FH)
5 - 6	Maximum Record Offset (Allocation)
7 - 8	Address Of Last Record Field

RECORD SYMBOL TABLE ENTRY FORMAT

TABLE 7

Record Fixed Field Symbol Table Entry Format

Byte Number	Field Description
0 - 1	Collision Address
2 - 3	Previous Symbol Table Entry Address
4	Form Field (Const. Val. 5FH)
5	Length Of Printname (LPN)
6 - 5+LPN	Printname Characters in ASCII
6+LPN - 7+LPN	Address Of Parent Record
8+LPN - 9+LPN	No. Of Bytes (Storage) in Field
10+LPN -11+LPN	Address Of Type Of Field
12+LPN -13+LPN	Offset From Record Base

RECORD FIXED FIELD SYMBOL TABLE ENTRY FORMAT

TABLE 7A

Record Tag Field & Variant Field Entry Format

Byte Number	Field Description
0 - 1	Collision Address
2 - 3	Previous Symbol Table Entry Address
4	Form: Variant (Const. Val. DFH) Tag (Const. Val. 9FH)
5	Length Of Printname (LPN)
6 - 5+LPN	Printname Characters in ASCII
6+LPN - 7+LPN	Address Of Parent Record
8+LPN - 9+LPN	No. Bytes in Field (Tag: No of Cases Cases)
10+LPN -11+LPN	Address Of Field Type
12+LPN -13+LPN	Offset From Record Base

RECORD TAG FIELD & VARIANT FIELD ENTRY FORMAT

TABLE 7B

from the base of the variable area assigned by the translator. The length of the variable is determined by the variable's type. The compiler keeps a count of the total amount of storage and passes this value to the translator at the completion of the compilation. The translator 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 5.

e. Procedure and Function Entries

Although the grammar of NPS-PASCAL supports procedures and functions, the construction of their symbol table entry was not implemented. Their implementation, however, would parallel the same format of all the other entries in the symbol table. The FORM values of 04H, and 05H were reserved for this purpose.

4. Symbol Table Construction and Access

Several standard construction and access 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 enter the printname length and printname characters. ENTERVARID is called with the value of FORM to be included with the symbol. The calling routines must subsequently enter the

The Form Field of Variable Entries

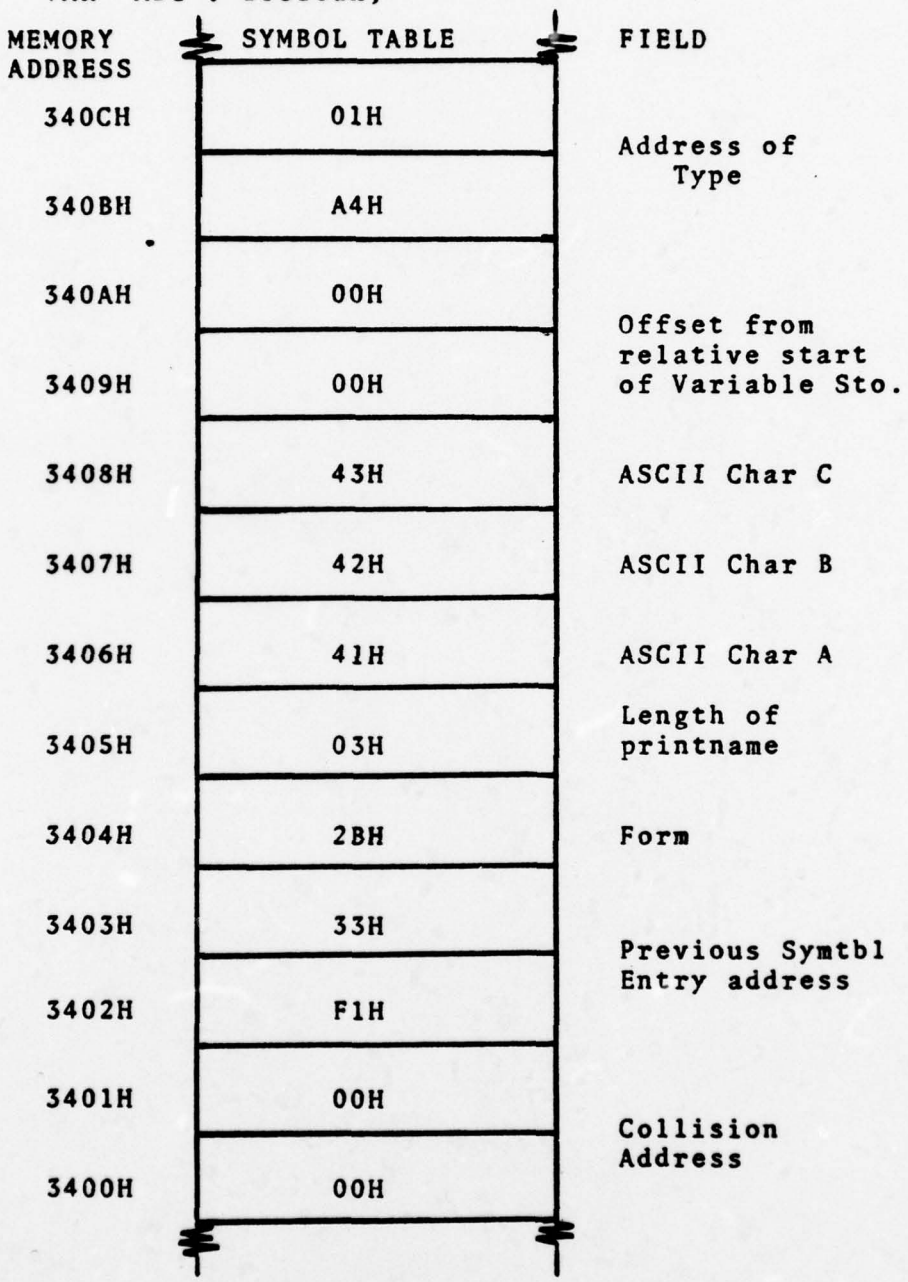
Value	Meaning (Type of Variable)
03H	Scalar-Ordinate
0BH	Integer
13H	Character
1BH	Real
23H	Complex
2BH	Boolean

FORM FIELD OF VARIABLE ENTRIES

TABLE 8

Variable Entry Example

VAR ABC : Boolean;



VARIABLE SYMBOL TABLE ENTRY

FIGURE 5

additional descriptive fields for the particular entry under construction.

Symbol table access is accomplished through the use of the standard lookup procedures, and pointers contained within entries. The procedure LOOKUP$ONLY can be called with the address of a printname as a parameter. The procedure calls CHECK$PRINT$NAME to compare the symbol table entry's printname with that of the parameter. The hashtable index of the parameter is used along with the symbol table collision fields to access the correct entries in the table. The procedure LOOKUPPNID was designed to accomplish the same task as LOOKUP$ONLY with the additional feature of checking the form field of the entry with a second parameter. If either procedure finds a match in the symbol table the global variable LOOKUP$ADDR is set to the location of the starting address in the symbol table of the matched entry, and the value TRUE is returned to the calling routine.

5. Parser

The parser was taken from the ALGOL-M compiler [5], a table driven pushdown automaton with parse tables generated using the LALR(k) parser generator [15]. It receives tokens from the scanner and analyzes them to determine if they are part of the NPS-PASCAL grammar. If the parser accepts a token, one of the following two actions is taken: it may save the token and continue to request tokens for the lookahead state, or it may recognize the

right part of a valid production and apply the production state causing a reduction to take place. If the parser 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.

When the RECOVER procedure is called, the parser backs up a state and attempts to continue parsing from that state. If this fails, the parser will continue to backup until the end of the currently pending reduction is encountered. At that point the invalid tokens are rejected and an attempt to parse the next token is made. This action continues until an acceptable token is found.

The major data structures in the parser are the LALR (1) parse tables and the parse stacks. The parse stacks consist of a state stack and a printname character stack. The printname character stack contains the individual characters of the tokens passed by the scanner.

6. Code Generation

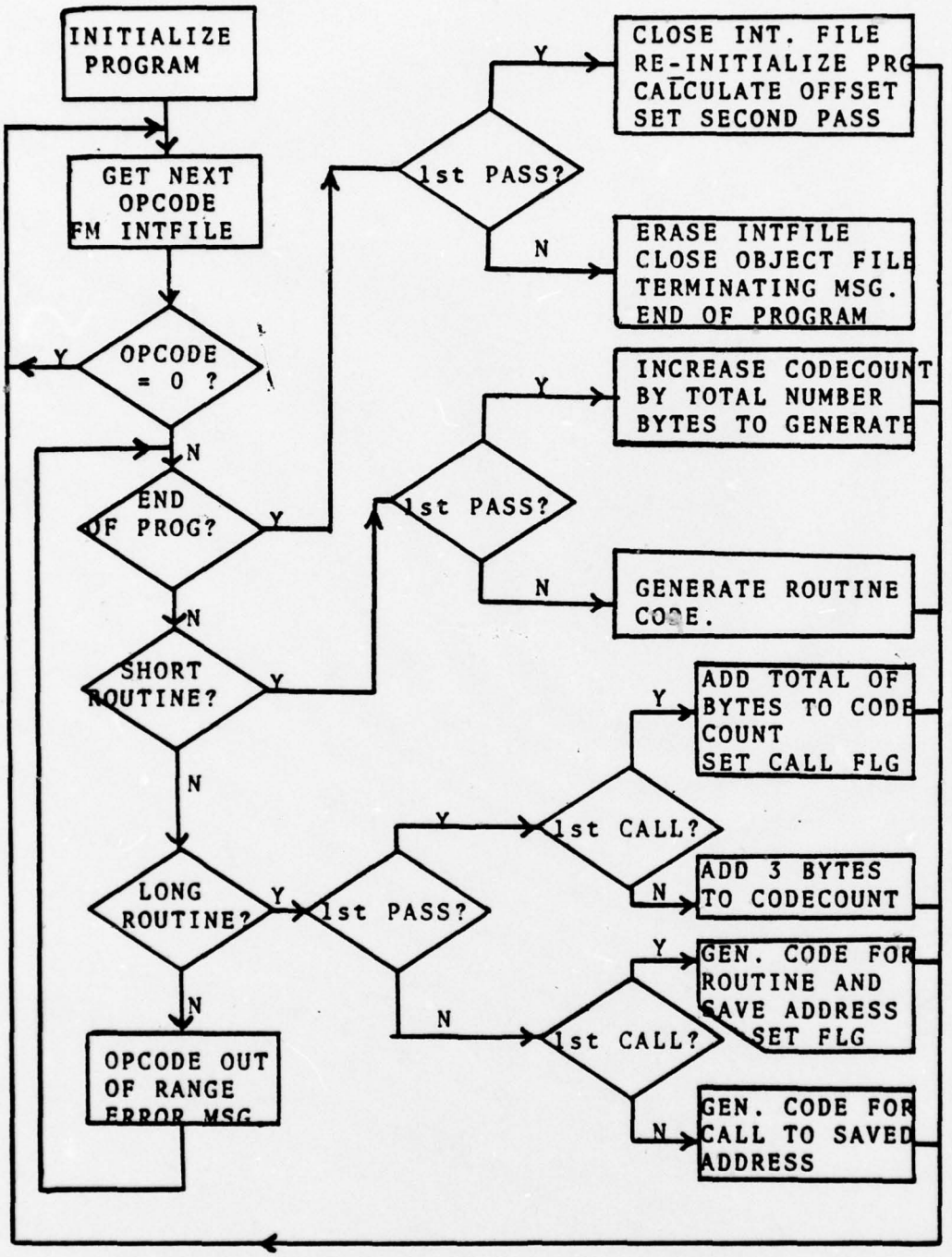
The parser not only verifies the syntax of the source statements, but also controls the generation of the intermediate code by associating semantic actions with production reductions. When a reduction takes place, the SYNTHESIZE procedure is called with the production number as a parameter. The SYNTHESIZE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

and the semantic actions for each reduction are listed in Appendix D.

B. TRANSLATOR ORGANIZATION

The zero-address machine code translator for NPS-PASCAL is a top down, modularized, program written in PL/M [9] and designed for easy modification. Modules whose future implementation are required for the completion of a full compiler were included in a stub form to indicate their absence in the program. As with any other program executed under the CP/M system, the translator is loaded, and starts execution at address 100 hexadecimal (100H). Its input is the intermediate file <filename>.PIN generated and stored on disk by the compiler. The translator makes two complete passes of this file. The intermediate file contains one byte numbers which represent either opcodes of the pseudo machine or operands sent from the compiler to the translator. The numerical value of the opcode is used to determine the proper entry point into a large case statement. Each case statement in the translator generates the portions of object code required to produce the output object module. A simplified flowchart of the intermediate code handling routine is shown in Figure 6.

On the first pass, the translator determines the size of the object module to be generated by summing the required number of bytes of object code needed to perform the instructions of each opcode used in the intermediate file.



FLOWCHART OF TRANSLATOR CASE STATEMENT

FIGURE 6

In addition, the translator's first pass also determines the relative addresses of all branching label (LBL) instructions which are generated by the compiler. The label number serves as an index into a label table, constructed by the translator, where each label's relative location in the code area of the object file is stored and retrieved. Label number zero is located in the two bytes following the end of the translator program in memory, denoted by .MEMORY in PL/M, while the location of the nth label is at bytes 2*n + .MEMORY and 2*n + .MEMORY + 1. Thus the label table is limited only by the size of the unused portion of memory in the host microcomputer.

Upon reaching the end of the first pass of the intermediate file, the translator re-initializes the program, closes and re-opens the intermediate file, opens the object module file, and begins the second phase of the machine code translation.

After the object file is initially opened, the translator generates the 8080 code to load the stack pointer (SP) to the maximum available address at execution time, which is the last byte of the CP/M Transient Program Area (TPA). A branch instruction is then generated to the first byte of the code area.

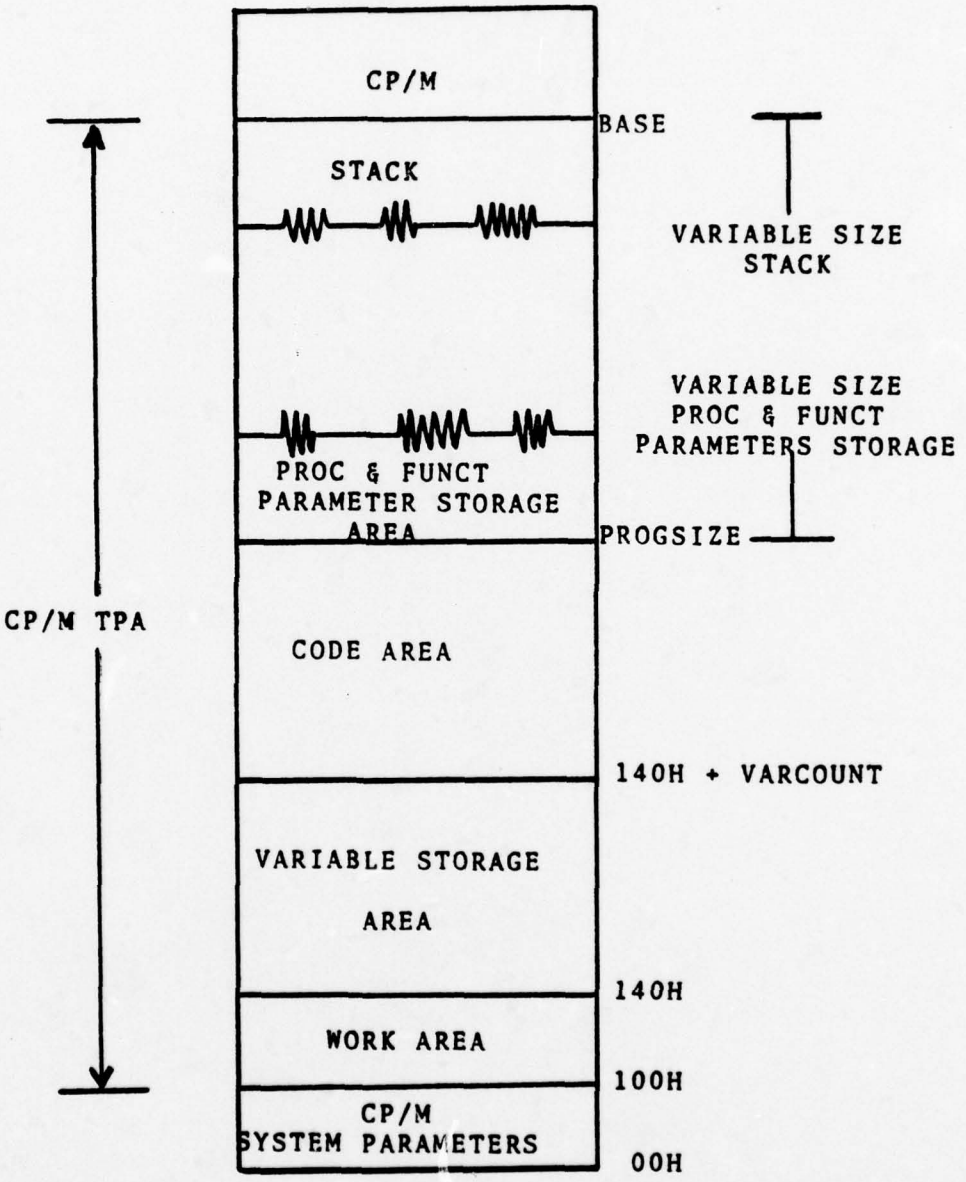
The object file's work area and Variable Storage Area is located between the branch instruction and the first byte of the code area. These two areas are initialized with zero

hexadecimal values. The work area, sixty bytes in length, is used for performing real arithmetic and as a scratch area for temporary variables. The Variable Storage Area is used for the storage of variables declared in the source program. The symbol table, which was generated by the compiler, is not available for use by the translator. Therefore, the access to variables in the Variable Storage Area is accomplished using the operands in the intermediate file.

On the second pass of the intermediate file object code is generated at address 140H plus the number of bytes allocated for program variables. Figure 7 shows the object file as it appears at execution time in the memory of the microcomputer.

The intermediate file contains two separate addressing schemes. The variable addresses are converted to absolute addresses by adding the offset of the work area to each address. The label addresses are evaluated on the first pass by counting the number of bytes of object code which must be generated prior to the appearance of the label in the intermediate file. On the second pass, the translator adds the offset of the work area size and the variable allocation size to the previously stored address.

Two methods of code generation are used by the translator. Simple intermediate instructions, which require ten or less bytes of object code, are generated in-line. Complex instructions requiring more than ten bytes are



OBJECT CODE IN TARGET COMPUTER

FIGURE 7

generated once when first encountered. Subsequent occurrences of the same intermediate file opcode causes the generation of a call to the code generated for the first invocation. This results in the generation of three bytes of object code for each succeding occurrence.

There are three routines used to generate the object module code. GENERATE is called for single byte generation, GEN$FIVE generates five bytes of code, and G$TEN generates ten bytes.

At the completion of the second pass the translator closes the object module file, erases the intermediate file and generates a completion message at the console. If an error is detected during translation, an error message is generated at the console and the program terminates.

1. Allocation of Storage Space

The opcode ALL is generated by the compiler to specify the number of bytes to be allocated in the object module's Variable Storage Area. In addition, each opcode in the intermediate file indicates the size and type of data that is to be operated upon.

a. Byte Data

Byte data items in the object module have two storage modes. The data is stored in byte locations when in memory. However, when a byte value is loaded into the stack, the byte data is preceeded by a zero value byte and loaded in the stack in a two byte format. A description

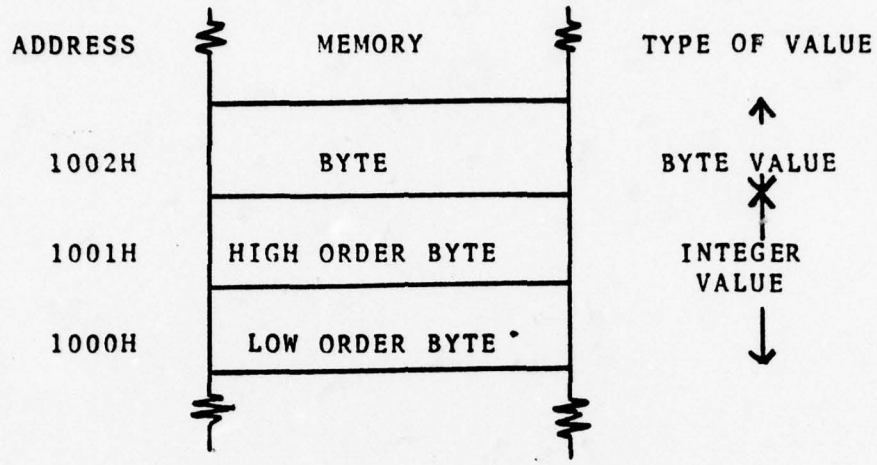
of the byte storage modes is shown in Figures 8 and 9. Byte data values are operated upon by the translator in byte format routines generated in the object file. Byte data may represent characters, numbers, or boolean data.

b. Integer Data

Integers are represented by two byte values and are stored in memory and the system stack in the same format. The high order byte is stored first followed by the low order byte of the integer number. The storage of integer numbers in memory and the stack is shown in Figures 8 and 9. The storage follows the processing requirements of the 8080 Microprocessor [8] to complete moves of data from memory or the stack into the processor double byte registers. An example of the POP and PUSH operation is shown in Figure 10. Integers are represented in two's complement form. They may take on values from -36,768 to +36,767. The high order bit of the integer representation is the sign bit. A zero high order bit indicates a positive integer value and a one indicates a negative value.

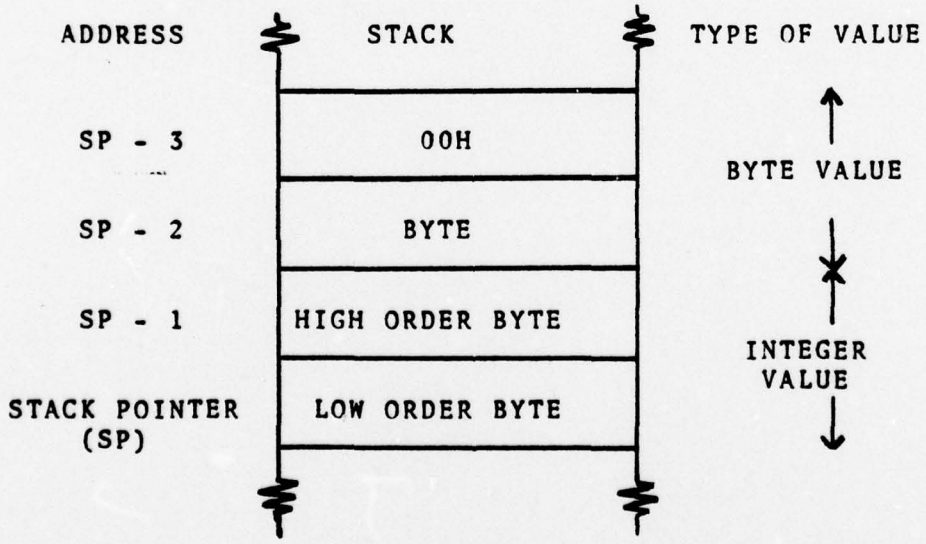
c. Decimal Data

Decimals in the NPS-PASCAL compiler are represented in binary coded decimal (BCD) format. Every decimal number is represented by 14 digits and is stored in eight contiguous bytes. The first 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. The byte



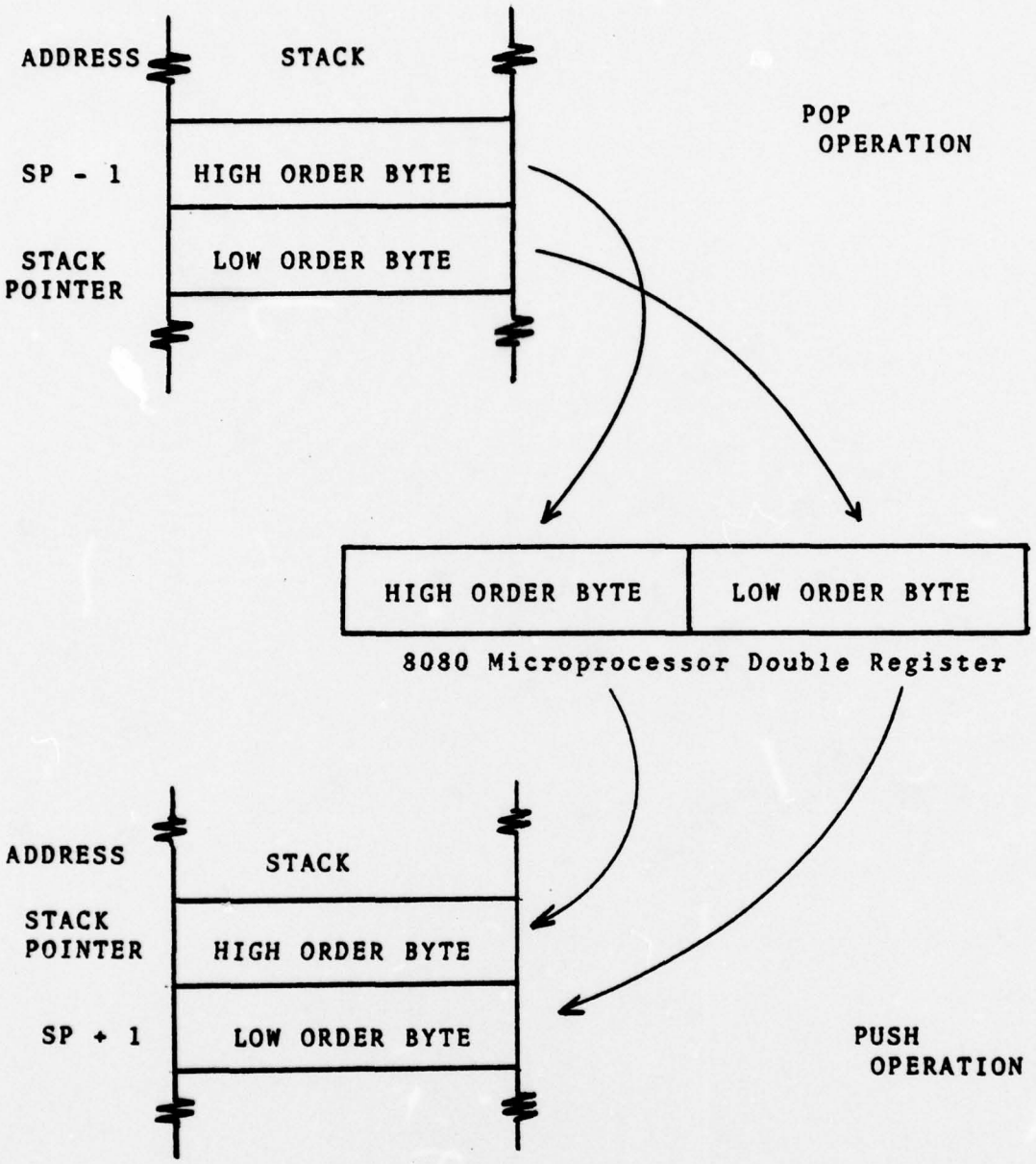
BYTE AND INTEGER VALUES IN MEMORY

FIGURE 8



BYTE AND INTEGER VALUES IN STACK

FIGURE 9



POP AND PUSH OPERATIONS

FIGURE 10

closest to the exponent byte represents the last two digits of the number while the last byte contains the first two digits of the number. Figure 11 shows a BCD number stored in memory.

The sign information byte uses the high order bit to indicate the sign of the number. A high order one bit indicates a negative number while a zero bit represents a positive number. The remaining seven bits represent the exponent and its sign, with a bias of 64. Values larger than 64 represent a biased positive exponent, while the values less than 64 represent exponents of negative sign with a result equal to the difference between 64 and the value. This reference point allows a range of exponent values from -64 to +63. The decimal point of the number is always assumed to be before the first digit.

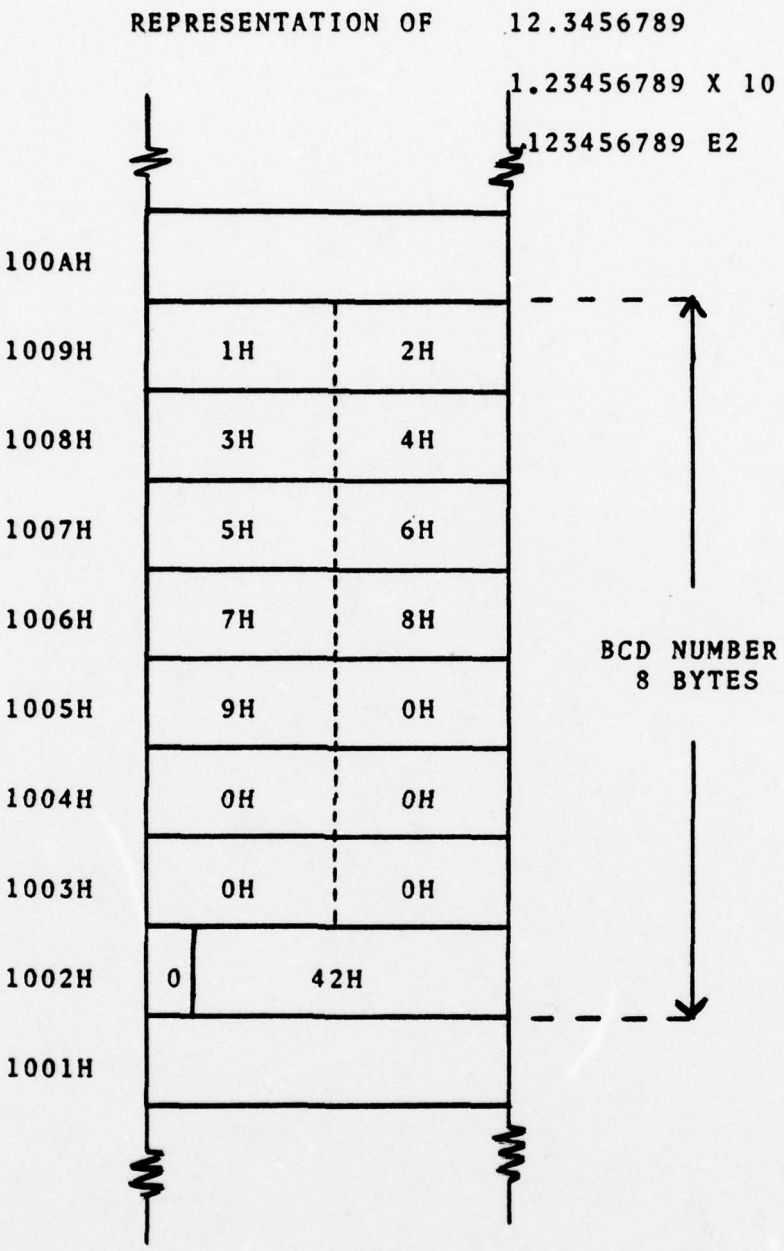
d. String Data

Strings in NPS-PASCAL are stored in memory sequentially. The first byte located at the starting address location indicates the length of the string. The string of ASCII characters that follow the length byte is arbitrarily limited to 80 characters, which is the length of an output line to the console.

2. Arithmetic Operations

a. Logicals

Logical operations or boolean operations act on byte values of zero and one only. A zero value indicates



BCD NUMBER IN MEMORY

FIGURE 11

a false while a non-zero value indicates true. Logical operations requiring comparison between two elements return the result of the operation in the form of a true or false value. Logical operations are also performed using boolean values to determine logical unions, disjunctions, or complements.

b. Integers

Operations with integers are straightforward. Both integers are removed from the stack and placed in double byte registers in the 8080 microprocessor where the requested operation is carried out. Operations with integers include addition, subtraction, multiplication, division, logical comparisons, and transformations to BCD format. Except for transformations, all results of integer operations are returned to the stack in the two byte integer format.

c. Decimals

Arithmetic operations with BCD numbers are more complex than with integers. However, the 8080 microprocessor provides the use of the DAA operator which simplifies the addition operation. BCD numbers and temporary results are stored in the work area during all real operations. Calculations are carried out by moving decimal number pairs into the 8080 registers. The required operation is applied repeatedly to successive bytes until completion. The resulting value of the operation is returned to the NPS-PASCAL stack in its eight byte BCD

format.

Since the default position of the decimal point in a BCD number is assumed to exist before the first digit, all operations must left justify each result. The functions of addition, subtraction, multiplication, and division were not included in the implementation at this point. However, the operations of complementing, transformation to integer format, and some logical comparisons were implemented.

3. String Operations

String operations were implemented for output and for comparison to determine logical equality or inequality. Comparisons take place either immediately between a string passed in the intermediate file and a string in memory or between two strings located in memory. Results of the comparison are returned to the stack in boolean form.

4. Input - Output

Input as well as output is done interactively through the console. The translator reads input numbers from the console and transforms them into the internal form of either an integer or a BCD number. Similarly, the work area is used to convert numbers into a printable form for output to the console. The routine WRITE$STRNG is used to generate the object code which performs all output operations, while the routines READ$INT and READ$BCD are used to perform all inout operations.

5. NPS-PASCAL Pseudo Operators

The pseudo code used in NPS-PASCAL, which differs from the PASCAL-<P> code [13], is listed below.

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 for the VSA. The size of the VSA is provided in the two byte operand.

LBL: (Label). This operator is used on the first pass of the translator 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 number.

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

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

LOD: (Load Byte). This operator generates code to move the top two bytes on the stack into the 8080 HL register. The byte is then moved from its location in memory to the stack preceeded by a high order zero byte.

LODB: (Load BCD). This operator generates

code to move the top two bytes on the stack into the 8080 HL register; it then increments the register by eight and moves eight bytes in decending order from memory onto the stack.

LODI: (Load Integer). This operator generates code to move the top two bytes on the stack into the 8080 HL register; it then moves two bytes from that location onto the stack.

c. Arithmetic Operators

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

convert the BCD number to an integer number. The resulting integer number 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 (not implemented).

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 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 number is returned to the top of the stack (not implemented).

SUBI: (Subtract 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 integer. The resulting integer number is returned to the stack.

MULB: (Multiply BCD). This operator generates code to move two BCD values from 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 (not implemented).

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 code to move two BCD values from the top of the stack into the work area where the second BCD is divided by the first BCD number. The quotient is returned to the top of the stack in BCD format (not implemented).

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 integer. 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 (not implemented).

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 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 BCD number, a one is returned to the stack. Otherwise a zero is returned (not implemented).

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 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 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 integer 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 (not implemented).

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 integer a one is returned to the stack. Otherwise a zero is returned.

NEGB: (Negate BCD). This operator

generates code to move the top two bytes on the stack to the 8080 registers where it complements the sign bit of the BCD sign byte then returns the two bytes to the stack.

NEGI: (Negate Integer). This operator generates code to move the integer number from the stack to the 8080 registers, complements the number to its negative number and returns it to the stack.

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

COMI: (Complement Integer). This operator generates code to move the top two byte integer number into the 8080 registers, finds the two's complement of the number and returns the value to the stack.

d. Boolean Operators

NOT: (Boolean Not). This operator generates code to move the two bytes at the top of the stack into the 8080 registers and compares the low order byte. If the byte is zero it returns a two byte value of one to the stack. If the byte is one it returns a two byte value of zero.

AND: (Boolean And). This operator generates code to move the next two integer numbers into the 8080 registers for logical AND comparison of their low order bytes. 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.

BOR: (Boolean Or). This operator generates code to move the next two integer numbers into the 8080 registers for logical OR comparison of their low order bytes. 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

EQLS: (Equal String). This operator generates code to compare a string whose length is given by the following byte. It moves the two bytes at the top of the stack into the 8080 HL register and compares the string one byte at the time for equality. If the string in the specified memory location is equal to the string in the intermediate file, a one is returned to the stack. Otherwise a zero is returned (not implemented).

NEQS: (Not Equal String). This operator generates code to compare a string whose length is given by the following byte. It moves the two bytes at the top of the stack into the 8080 HL register and compares the strings for equality. If the string in the specified memory location is equal to the string in the intermediate file, a zero is returned to the stack. Otherwise a one is returned (not implemented).

f. Stack Operators

DCRB: (Decrement Stack BCD). This operator generates code to decrement the size of the stack by eight bytes.

DCRI: (Decrement Stack Integer). This operator generates code to decrement the size of the stack by two bytes.

DCRI: (Decrement Stack Byte). This operator generates code to decrement the size of the stack by two bytes.

g. Program Control Operators

BRL: (Branch to Label). This operator calculates the label address in the label table using the next two byte label number and moves the codecount stored at the label table address and adds to it the address of the start of the code area. It then generates code to branch to the calculated address.

BCL: (Branch Conditional Label). This operator calculates the branching address in the same manner as the BRL code above. It then generates the code to move the two bytes on top of the stack to the 8080 registers to check the condition. If the low order byte removed from the stack is a one, the branching instruction is executed. If the low order byte is a zero the program continues without branching.

ENDP: (End of Program). At the end of the first pass of the translator this code reinitializes the

program, closes the intermediate file, and sets the second pass condition to true. On the second pass this opcode generates code to terminate the object code file and terminates compilation.

h. Store Operators

STOB: (Store BCD). This operator generates code to move the two bytes at the top the stack into the 8080 HL register then moves the next eight bytes from the stack to memory starting at the address indicated by the HL register. The value of the BCD number is preserved in the stack by incrementing the stack pointer by eight.

STOI: (Store Integer). This operator generates code to move the two bytes at the top of the stack into the 8080 HL register then moves the next two bytes from the stack to memory starting at the address indicated by the HL register. The value of the integer number is preserved in the stack by incrementing the stack pointer by two.

STOI: (Store Byte). This operator generates code to move the two bytes at the top of the stack into the 8080 HL register, then moves the next byte from the stack to memory at the address indicated by the HL register. The value of the byte value is preserved in the stack by incrementing the stack pointer by two.

STDB: (Store Destruct BCD). This operator generates code to move the two bytes at the top of the stack 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 the two bytes at the top of the stack into the 8080 HL register then moves the next two bytes from the stack to memory starting at the address indicated by the HL register.

STD: (Store Destruct Byte). This operator generates code to move the two bytes at the top of the stack into the 8080 HL register then moves the next byte from the stack to memory starting at the address indicated by the HL register.

i. Input - Output Operators

RDVB: (Read Variable BCD). This operator generates code to read a BCD number from the console, change it into its acceptable storage form, and place the eight byte internal form on top of the stack (not implemented).

RDVI: (Read Variable Integer). This operator generates code to read an integer number from the console, change it into its acceptable storage form, and place the two byte number on top of the stack (not implemented).

RDVS: (Read Variable String). This operator generates code to read a string variable from the console and stores it at a location in memory indicated by the two top bytes on the stack (not implemented).

WRVB: (Write Variable 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 prints the number to the console.

WRVI: (Write Variable Integer). This operator generates code to move an integer number into the work area, change the number into its printable form and print the number at the console.

WRVS: (Write Variable String). This operator generates code to print a string variable at the console equal in length to the next one byte integer. The string variable follows the size byte.

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

j. Routine Operators

PRO: (Procedure Call). This operator generates code to save the present address loaded in the program counter (PC) register and loads the PC register with the address contained in the next two bytes (not implemented).

RTN: (Return From Procedure). This operator generates code to retrieve the address stored by the previously executed procedure and loads the PC register to continue the program at this location (not implemented).

SAVP: (Save Parameters). This operator generates code to save the present value of the parameters in the next available area above the end of the object code (not implemented).

UNSP: (Unsave Parameters). This operator generates code to return the parameter values from the area above the object code (not implemented).

IV. CONCLUSIONS AND RECOMMENDATIONS

The NPS-PASCAL project described here is the first stage of a full PASCAL implementation for Intel 8080 based microcomputers. Although incomplete, the compiler structures are essentially intact, and the code generator is formulated.

Several features of the PASCAL language have not been implemented and are indicated in the program listings. The structure of the heap, which is required for recursive procedures and functions, as well as record manipulations, has not been designed nor implemented. Integrated program testing, including timing tests, will also be necessary to determine the correctness and efficiency of the overall system. Enhancements, such as formatted I/O, external subroutine library access, and run-time debugging, must also be designed and implemented. The structure of the symbol table should make run-time debugging relatively easy.

APPENDIX A - COMPILER ERROR MESSAGES

DE	Disk error : Recompile.
TO	Symbol table overflow : Reduce number of declaretions.
EE	Exponent size error : See user manual.
IE	Integer size error : See user manual.
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.
NP	No production : Syntax error in source line.
IC	Invalid constant variable : Constant entry in symbol table invalid - probably due to a prior error.
IE	Invalid expression type : The types of variables used in an expression are incompatible.
ES	Expression stack overflow : Simplify program.
IR	Invalid read variable : Only INTEGER or REAL values can be read.
LS	Label syntax error : All labels must be integers.
DC	Duplicate constant name : Constant identifiers must be unique.
DT	Duplicate type name : Type 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.
UL	Undefined label error : Label not declared in label statement.
AT	Assignment type error : Type of expression not compatible with assignment variable type.
CE	Invalid expression : The variable types within the expression are not compatible.
UO	Invalid unary operator : Variable type must be INTEGER, REAL, or subrange of INTEGER.
SO	State stack overflow : Simplify program.
VO	Variable stack overflow : Reduce the length of variable printnames.
IO	If statement stack overflow : Simplify program.

APPENDIX B - TRANSLATOR MESSAGES

MESSAGES

Disk file close error.

Disk file create error.

Disk file write error.

No internal file found.

IO Integer overflow.

EO Exponent overflow.

EU Exponent underflow.

DZ Division by zero attempted.

II Invalid Console Input

End of compilation. No program errors.

Compilation terminated due to error(s).

APPENDIX C NPS-PASCAL LANGUAGE MANUAL

This section describes the various elements of the NPS-PASCAL language. The format of the element will be shown, followed by a description and examples of its use. The following notation is used:

Braces {} indicate an optional entry.

A vertical bar | indicates alternate choices, one of which must appear.

Reserved words are indicated by capital letters.

Reserved words and other special symbols must appear as shown.

Items appearing in small letters are elements of the language which are defined and explained elsewhere in the language manual.

ELEMENT:

arithmetic expression

FORMAT:

integer¦decimal

variable

{(} arithmetic expression binary operator

arithmetic expression {)}

{(} unary operator arithmetic expression {)}

DESCRIPTION:

Arithmetic expressions consist of basic data elements combined with arithmetic operators in algebraic notation.

EXAMPLE:

```
(A + B)
-A
C + 12.6
```

ELEMENT:

Array Declaration

FORMAT:

```
VAR identifier: ARRAY (* index-type-string *)
          OF component-type
VAR identifier: ARRAY (* index-type-string
          {, index-type-string} *)
            OF component-type
```

DESCRIPTION:

Array types consist of a fixed number of declared components ; where all the components are of the same type. Each component of the array variable can be directly accessed by the name of the array variable followed by its index location in the array enclosed in the (* notation. See assignment statement.

EXAMPLE:

```
VAR temperature: ARRAY (* 1..10 *) OF REAL;
VAR grades: ARRAY (* 1..5,2..8 *) OF INTEGER;
```

ELEMENT:

assignment statement

FORMAT:

variable := expression

DESCRIPTION:

Assignment statements indicate a value to be assigned to a variable or a value to replace the present value of a variable. The symbol := is the assignment operator and must not be confused with the relational operator = indicating equality. The resulting value of the expression on the right side of the assignment statement must be consistent with the type of variable being assigned the new value. The only exception is that INTEGER expression types will be converted to REAL in the assignment to a real variable.

EXAMPLE:

```
x := 0;
temperature(* 2 *) := 103.7;
y := (15 DIV 4) * 2;
```

ELEMENT:

balanced statement

FORMAT:

simple statement

IF {(} boolean-expression {)} THEN balanced-statement
ELSE balanced-statement

DESCRIPTION:

Simple statements are statements of which no part constitutes another statement; therefore, it is considered a balanced statement. The IF conditional statement has parts constituting other statements. However, if the IF clause is balanced by an ELSE balanced-statement the statement is balanced.

EXAMPLE:

```
IF x > 0 THEN flag := TRUE; ELSE a := 2;
x := (y * 10) / b;
goto 300;
```

ELEMENT:

block

DESCRIPTION:

The block preceeded by a program-heading forms the PASCAL program. Similarly, the block preceeded by a procedure-heading or a function-heading forms parts of the Procedure and Function Declaration Part. The block consists of Label Declaration Part, Constant Definition Part, Type Definition Part, Variable Declaration Part, Procedure and Function Declaration Part, and Statement Part. All of the parts listed may be empty except the last.

EXAMPLE:

See individual part descriptions for specific information on each part.

ELEMENT:

boolean expression

FORMAT:

NOT boolean-expression

boolean-expression OR boolean-expression

boolean-expression AND boolean-expression

{(} expression relational-operator expression {)}

DESCRIPTION:

Comparison of constant to constant, INTEGER to INTEGER, REAL to REAL, REAL to INTEGER, and string to string are allowed in NPS-PASCAL. Comparison of numbers of different types are accomplished by changing the INTEGER number to REAL prior to comparison. The results of the comparison are recorded as a 1 if the comparison is TRUE and as a 0 if the comparison is FALSE.

EXAMPLE:

```
NOT errflag
( X - 3 > 0) OR ( errflag)
( Y > 0 ) AND ( Y < 10)
```

ELEMENT:

case statement

FORMAT:

CASE expression OF case-list-elements-list; END;

DESCRIPTION:

The case expression of the case statement is the selector for the case-list-elements-list. This list provides the choices of statements to select from. A statement whose label is equal to the current value of the selector is executed.

EXAMPLE:

```
CASE i OF
    1: x := 0;
    2: x := x;
    3: x := 2x;
    4: x := limit;
END
```

ELEMENT:

costant definition part

FORMAT:

empty

CONST constant-definition-list;

DESCRIPTION:

Constant Definition Part introduces identifiers as synonims for constants. The constant may be a number, signed or unsigned constant identifier, or a string.

EXAMPLE:

```
CONST least = 0; most = 50; next = a;
```

ELEMENT:

conditional statement

DESCRIPTION:

Conditional statements for NPS-PASCAL fall in two categories, the case statement and the IF statement. Language modification to have NPS-PASCAL parsable with one look ahead forced a distinction between two IF statements. See balanced statement and unbalanced statement. The case statement was included in the simple statements.

EXAMPLE:

See case statement, balanced statement, and unbalanced statement.

ELEMENT:

compound statement

FORMAT:

BEGIN statement-list END

DESCRIPTION:

Compound statements specify that the statements are executed in the same sequence as they are written. The BEGIN and END reserved words are the statement delimiters. A compound statement can be used whenever a statement is required.

EXAMPLE:

```
BEGIN
    a := least;
    b := a + 2;
    c := most
END
```

ELEMENT:

data type definition

DESCRIPTION:

A data type determines the set of values which variables of that type may assume. It also associates an identifier with the type Type.

EXAMPLE:

See simple type, structured type, and pointer type.

ELEMENT:

declarations

DESCRIPTION:

In NPS-PASCAL all the variables, labels, functions, procedures, constants, and data types to be used in the program must be declared at the beginning of the program.

EXAMPLE:

See block.

ELEMENT:

expression

DESCRIPTION:

There are two types of expressions in PASCAL. The boolean expression and the arithmetic expression.

EXAMPLE:

See arithmetic expression and boolean expression.

ELEMENT:

FOR statement

FORMAT:

FOR index-variable := initial-value DOWNTO final-value
DO statement

FOR index-variable := initial-value TO final-value
DO statement

DESCRIPTION:

FOR statements or FOR loops are iterative statements in PASCAL. The expression to assign the initial value to the index variable is only evaluated once before the first iteration of the body of the loop. At each iteration the value of the index variable is changed automatically therefore the value of the index variable can not be changed within the body of the loop. Index variables and the result of expressions giving the initial value and final value must be of type INTEGER.

EXAMPLE:

```
FOR i := 10 DOWNTO 1 DO
    totaltax := sales(* i *) * taxrate;
FOR n := (3x + 2) to (2x + 20) DO
    sum := sum + n;
```

ELEMENT:

file types

FORMAT:

TYPE identifier = FILE OF type;

(if type of file is CHAR then file is textfile)

DESCRIPTION:

NPS-PASCAL uses the word file to specify a structure consisting of a sequence of components all of which are of the same type. Declaring a file automatically introduces a buffer variable pointer that indicates the component to read or append in the file. All the operations of a sequential file generation and inspection can be expressed in terms of four primitive file operators reset, rewrite, get, and put, with a controlling symbol EOF.

EXAMPLE:

```
TYPE socsecno = FILE OF INTEGER;
TYPE text = FILE OF CHAR;
```

ELEMENT:

function call

FORMAT:

identifier := function-identifier

identifier := function-identifier
(formal-parameter(s))

DESCRIPTION:

Functions may appear as primary elements in arithmetic or boolean expressions. The type of the function must be scalar, subrange, or pointer type.

EXAMPLE:

Not implemented.

ELEMENT:

GOTO statement

FORMAT:

GOTO label

DESCRIPTION:

GOTO statements serve to indicate that further processing should continue at another part of the program text, at the location of the label. GOTO statements are restricted jumps within their scope. It is not possible to jump into procedures. Also, every label must be declared in a label declaration portion of the program.

EXAMPLE:

```
GOTO 300
GOTO 50
```

ELEMENT:

identifier

FORMAT:

letter

letter ¦{letter or digit ¦{letter or digit}}

DESCRIPTION:

Identifiers serve to denote constants, types, variables, procedures, and functions. They must begin with a letter and may be followed by any combination of letters and/or digits. Identifiers may consist of only one letter. Only the first eight characters of an identifier are significant; although, the lenghth of an identifier may be up to 32 characters. Two similar identifiers may be distinguishable if different in the first eight characters.

EXAMPLE:

```
a
sorted
test1
h2oweight
```

ELEMENT:

IF statement

FORMAT:

IF boolean-expression THEN unbalanced-statement

IF boolean-expression THEN balanced-statement

IF boolean-expression THEN balanced-statement
ELSE balanced-statement

DESCRIPTION:

IF statements are statements of conditional execution. If the boolean expression evaluates to TRUE the statement following the reserved word THEN is executed and the ELSE statement is ignored if applicable. If the boolean expression evaluates to FALSE the THEN statement is ignored and the ELSE statement is executed if applicable.

EXAMPLE:

```
IF errorflag THEN errorprocedure
IF (A + B) < C THEN C := A + B
    ELSE A := C
```

ELEMENT:

input

FORMAT:

READ(variable-name ![,variable-name])

READ(file-name, variable-name ![,variable-name])

READLN(variable-list)

READLN(file-name, variable-list)

DESCRIPTION:

The READ and READLN statements allow the user access to the input device to accept data for use in the execution of the NPS-PASCAL program. READLN allows to read and subsequently skip to the beginning of the next line, bypassing any further data on the current line.

EXAMPLE:

```
READ(x)
READLN(x,y,z)
```

ELEMENT:

label declaration part

FORMAT:

LABEL label-number {,label-number};

DESCRIPTION:

Any statement in a NPS-PASCAL program may be marked by prefixing it with a numerical label followed by a colon. This allows the statement to be referenced by a GOTO statement. All labels must be defined in the label declaration part before their use. A label is an unsigned integer consisting of no more than 32 digits.

EXAMPLE:

```
LABEL 3;
LABEL 5,15,20,25;
LABEL 123456789;
```

ELEMENT:

label

FORMAT:

unsigned integer 32 digits or less in length

DESCRIPTION:

See label declaration part

EXAMPLE:

```
50: x := y + 3
234568: if x < 0 then flag := TRUE
```

ELEMENT:

output

FORMAT:

```
WRITE(file-name, variable-list)
WRITE(variable-list)
WRITELN(file-name, variable-list)
WRITELN(variable-list)
```

DESCRIPTION:

The WRITE and WRITELN statements allow the program access to the output devices. WRITELN terminates the current line of the output file after its parameters have been acted upon. WRITE and WRITELN permit documentation of the output by outputting any information between quotations literally.

EXAMPLE:

```
WRITE('the value of x is',x)
WRITELN( x, y, 'the average is', z)
```

ELEMENT:

program heading

FORMAT:

```
PROGRAM identifier (file-identifier {,
                    file-identifier});
```

DESCRIPTION:

Program heading gives the NPS-PASCAL program a name and lists its input and output parameters. The name is only used for identification purposes and is not otherwise significant inside the program. The parameters indicate the files/devices through which the program communicates with its environment.

EXAMPLE:

```
PROGRAM bubble(input,output);
PROGRAM primes(output);
PROGRAM communicate(infile,outfile);
```

ELEMENT:

Procedure and Function Declaration Part

FORMAT:

not implemented

DESCRIPTION:

Every procedure or function used in a PASCAL-SM program must be defined before its use. Procedures are subroutines which are activated by procedure statements. Functions are subroutines that yield a resultant value, and therefore can be used as constituents of expressions.

EXAMPLE:

See procedure call, and function call.

AD-A061 040 NAVAL POS GRADUATE SCHOOL MONTEREY CALIF F/G 9/2
NPS-PASCAL. A PARTIAL IMPLEMENTATION OF PASCAL LANGUAGE FOR A M--ETC(U)
JUN 78 J C GRACIDA, R R STILWELL

UNCLASSIFIED NL

2 OF 3
AD
A061040

ELEMENT:

pointer type

FORMAT:

TYPE identifier = $type-identifier

DESCRIPTION:

A pointer type is a variable bound to another variable. It consists of an unbounded set of values pointing to its bound variable. The value NIL is always an element of a pointer variable and points to no element at all.

EXAMPLE:

```
TYPE link = $relative
```

ELEMENT:

Procedure call

FORMAT:

CALL procedure-name {(parameter {,parameter})}

DESCRIPTION:

Procedure calls are the statements which invoke the execution of a predefined procedure. Upon completion of the procedure the execution resumes at the next statement following the call.

EXAMPLE:

```
CALL printname
CALL largest (number1, number2)
```

ELEMENT:

Reserved words

DESCRIPTION:

Keywords used in NPS-PASCAL are reaserved words and are not allowed to be used as identifiers. The follwing list are NPS-PASCAL reserved words:

AND	END	NIL	SET
ARRAY	FILE	NOT	THEN
BEGIN	FOR	OF	TO
CASE	FUNCTION	OR	TYPE
CONST	GOTO	PACKED	UNTIL
DIV	IF	PROCEDURE	VAR
DO	IN	PROGRAM	WHILE
DOWNTO	LABEL	RECORD	WITH
ELSE	MOD	REPEAT	

ELEMENT:

Repeat statement

EXAMPLE:

REPEAT statement {;statement} UNTIL expression

DESCRIPTION:

The sequence of statements is executed at least once until the condition of the expression is met.

EXAMPLE:

```
REPEAT i UNTIL n = 10
REPEAT & initialize arrays &
  A := A + 1;
  B(* a *) = 0.0;
UNTIL  A = 10
```

ELEMENT:

Repetitive statement

DESCRIPTION:

Repetitive statements specify that certain statements are to be executed repeatedly. In some cases the number of repetitions may be known before the first repetitive execution. In other cases the condition to stop repeating the execution is determined after execution of a statement.

EXAMPLE:

See While statement, repeat statement, and For statement.

ELEMENT:

RECORD types

FORMAT:

TYPE record-name = RECORD field-list END

DESCRIPTION:

A record is a template for a structure whose parts may have quite distinct characteristics. It consists of a fixed number of components, called fields. Components can not be directly indexed.

EXAMPLE:

```
TYPE date = RECORD mo : (jan,apr,jul,oct);
                   day : 1..31
                  year : INTEGER
            END
```

ELEMENT:

remarks

FORMAT:

& comment &

DESCRIPTION:

Remarks are used to document programs. Any remark placed between the delimiters can be inserted in a program without causing an alteration in the program's meaning.

EXAMPLE:

```
& beginning of execution phase &
& this routine changes integer signs &
```

ELEMENT:

READ / READLN statements

FORMAT:

```
READ(variable-list)
READ(file-name, variable-list)
READLN(variable-list)
READLN(file-name, variable-list)
```

DESCRIPTION:

The READ and READLN procedures allow input from external files or the default input device to the program. The first parameter is the file from which to access data. If the first parameter is not a file identifier then the default file is the default input device. Read statements cause the next available value to be read from the file and assigned to the variable whose name is indicated as a parameter. When more than one variable value is read with the same statement the variable names in the variable list are separated by commas. See input.

EXAMPLE:

```
READ (x)
READ (x,y,z)
READLN (x(5))
READLN (test1,x,p,r)
```

ELEMENT:

SET types

FORMAT:

SET OF base-type

DESCRIPTION:

SET types define a range of values which is the power set of its base type. Base types must not be structured types. Operators applicable to all set types are:

+	union
-	set difference
*	intersection
IN	membership

EXAMPLE:

```
SET OF week
SET OF family
```

ELEMENT:

scalar types

FORMAT:

scalar-type-ident = (identifier {,identifier})

DESCRIPTION:

Scalar type defines an ordered set of values by enumeration of the identifiers which denote these values. The sequence in which the identifiers are declared is used in performing operations on the variables assigned this type.

EXAMPLE:

```
color = (blue,red,white)
card = (jack,queen,king,ace)
```

ELEMENT:

simple types

FORMAT:

scalar-type

subrange-type

identifier

DESCRIPTION:

Simple types in NPS-PASCAL are the basic elements of all type structures. They consist of the identifiers, INTEGER, REAL, CHAR, and BOOLEAN. Scalar defined types and subranges of INTEGER types, CHAR types, and scalar types are also included as simple types.

EXAMPLE:

See scalar types, subrange types, and identifier.

ELEMENT:

subrange types

FORMAT:

TYPE identifier : constant..constant

DESCRIPTION:

Subrange type is a subrange of another already defined scalar type called its associate scalar type. Subrange indicates the smallest and the largest value in the subrange. Subranges of REAL are not allowed.

EXAMPLE:

```
VAR small : 1..5;
    workday : mon..fry;
    initials : 'a'..'z';
```

ELEMENT:

structured types

DESCRIPTION:

Structured types are composed of other types. Options available to each structuring method indicate the prefered internal data representation. Type definition prefixed with the symbol PACKED economizes storage requirements (Packed option not implemented).

EXAMPLE:

See ARRAY type and RECORD type.

ELEMENT:

type BOOLEAN

DESCRIPTION:

BOOLEAN types assume a BOOLEAN value, one of the logical truth values denoted by the predefined identifiers TRUE or FALSE. Logical operators and relational operators yield BOOLEAN values when applied to BOOLEAN and arithmetic operands respectively.

EXAMPLE:

```
VAR errflg : BOOLEAN;
VAR signx, exponx : BOOLEAN;
```

ELEMENT:

type CHAR

FORMAT:

```
TYPE type-identifier = CHAR
VAR variable-identifier : CHAR
```

DESCRIPTION:

The value of type CHAR is an element of a finite and ordered set of characters. For NPS-PASCAL the CHAR set is the ASCII character set. Characters enclosed in single quotation marks denote a constant type.

EXAMPLE:

```
TYPE text = CHAR
VAR text : CHAR
```

ELEMENT:

type INTEGER

FORMAT:

```
TYPE type-identifier = INTEGER
VAR variable-identifier : INTEGER
```

DESCRIPTION:

The type INTEGER is an element of a defined subset of whole numbers. In NPS-PASCAL the range of integers is from -36,767 to 36,766. Arithmetic operators with integer operands yield integer values. Relational operators with integer operands yield BOOLEAN values.

EXAMPLE:

```
TYPE int = INTEGER
VAR a,b,c : INTEGER
```

ELEMENT:

type REAL

FORMAT:

```
TYPE type-identifier = REAL
VAR variable-identifier : REAL
```

DESCRIPTION:

A value of type REAL is an element of a defined subset of REAL numbers. In NPS-PASCAL the range of real numbers is expressed in BCD format, 14 decimal digits numbers, multiplied by a corresponding power of ten. The exponents of real numbers ranges between -63 and 64. Inputs may be expressed in exponential form, or real format. Decimal period must be preceded by at least one digit. Arithmetic operations using REAL number operands yield REAL value answers. Relational operators with real number operands yield BOOLEAN values.

EXAMPLE:

```
TYPE re = REAL
VAR re,im : REAL
```

ELEMENT:

type definition part

FORMAT:

TYPE identifier = type {;identifier = type}

DESCRIPTION:

Data types in NPS-PASCAL may be either described in the variable declaration part or referenced by a type identifier. The type definition allows the creation of new types. The definition determines a set of values and associates an identifier with the set.

EXAMPLE:

```
TYPE day = (mon,tue,wed,thu,fri,sat,sun);
TYPE color = (red, white, blue);
TYPE arry = ARRAY (* BOOLEAN,1 .. 10 *) OF
  INTEGER
```

ELEMENT:

unbalanced statement

FORMAT:

```
IF expression THEN statement
IF expression THEN balanced-statement
     ELSE unbalanced-statement
```

DESCRIPTION:

Unbalanced statement are a form of the IF conditional statement with an equal number of THEN and ELSE parts. This distiction was needed to form a LALR(1) parsable language.

EXAMPLE:

See IF statement, and balanced statement.

ELEMENT:

variable declaration part

FORMAT:

VAR identifier {,identifier}: type

{;identifier {,identifier}}: type

DESCRIPTION:

Every variable occuring in a statement must be declared in the variable declaration part prior to its use in the program. The variable declaration part associates the identifier with a data type.

EXAMPLE:

```
VAR counter, loop : INTEGER;
    errflg : BOOLEAN;
VAR pi : REAL;
```

APPENDIX D NPS-PASCAL LANGUAGE STRUCTURE

The following section describes the NPS-PASCAL language in BNF notation as modified to conform with the requirements of reference (10), Pascal User Manual, and reference (13), User's Guide To The LALR(k) Parser Generator. The description given describes the compiler data structures and the code generated. Numbered productions without a production result indicate empty productions. Items enclosed in brackets and separated by slants are alternative semantic actions. This notation is the same as provided by references 10 and 13.

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

2  <program heading> ::= PROGRAM <prog ident> (
                           <file ident> ) ;
 * <program identifier> <file identifier>

3                          PROGRAM <prog ident> (
                         <file ident> , <file ident> ) ;
 * <program identifier> <file identifier>
 * <file identifier>

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

5  <file ident> ::= <identifier>
 * { enter file identifier }

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

7  <ldp> ::=
 * < empty >
```

```
 8            LABEL <label string> ;
  * < label string >

 9  <label string> ::= <label>
  * < label > ; { enter label }

10                      <label string> , <label>
  * < label string > < label > ; { enter label }

11  <label> ::= <number>
  * < number > ; { check number type }
  .
12  <cdp> ::=
  * < empty >

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

14  <const def> ::= <ident const def>
  * < identifier constant definition >

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

16  <ident const def> ::= <ident const> = <constant>
  * < identifier constant > < constant >
  * { enter constant }

17  <ident const> ::= <identifier>
  * < identifier > ; { enter constant entry }

18  <constant> ::= <number>
  * < number > ; { assign constant attributes }

19                  <sign> <number>
  * < sign > < number > ; { set constant attributes }

20                  <constant ident>
  * < constant identifier > { set constant attributes }

21                  <sign> <constant ident>
  * < sign > < constant identifier >
  * { assign constant attributes }

22                  <string>
  * < string > ; { assign constant attributes }

23  <constant ident> ::= <identifier>
  * < identifier }

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

25            -
```

```
  * { assign SIGNTYPE value }

26  <tdp> ::=
  * < empty > ; { set CASE$STMT to FALSE }

27            TYPE <type def string> ;
  * < type definition string > { set CASE$STMT FALSE }

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

29                      <type def string> ; <type id>
  * < type definition string > < type identifier >

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

31  <type ids> ::= <identifier>
  * < identifier > ; { enter type }

32  <type> ::= <simple type>
  * < simple type >

33             <structured type>
  * < structured type >

34             <pointer type>
  * < pointer type >

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

36                      <subrange type>
  * < subrange type >

37                      <type ident>
  * < type identifier >

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

39  <scalar type> ::= ( <tident string> )
  * < type identifier string >

40  <tident string> ::= <identifier>
  * < identifier >
  * { enter type }

41                        <tident string> , <identifier>
  * < type identifier string > < identifier >
  * { enter type }

42  <subrange type> ::= <constant> .. <constant>
  * < constant > < constant > ; { enter subrange }
```

```
43  <structured type> ::= <unpacked structured type>
  * < unpacked structured type >

44                          PACKED
                            <unpacked structured type>
  * < unpacked structured type >

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

46                                  <record type>
  * < record type >

47                                  <set type>
  * < set type >

48                                  <file type>
  * < file type >

49  <array type> ::= ARRAY <lp> <index type string>
                     <rp> OF <component type>
  * < lp > < index type string > < rp >
  * < component type > ; { enter array type }

50  <lp> ::= (*
  N/A

51  <rp> ::= *)
  N/A

52  <index type string> ::= <index type>
  * < index type > ; { set array dimensions }

53                          <index type string> ,
                            <index type>
  * < index type string > < index type >
  * { set array dimensions }

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

55  <component type> ::= <type>
  * < type >

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

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

58                  <fixed part> ; <variant part>
  * < fixed part > < variant part >

59                  <variant part>
  * < variant part >
```

```
60  <fixed part> ::= <record section>
  * < record section >

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

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

63
  * < empty >

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

65                             <field ident string> ,
                               <field ident>
  * < field identifier string > < field identifier >

66  <field ident> ::= <identifier>
  * < identifier > ; { enter record field }

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

68                     CASE <type ident> OF
                       <variant string>
  * < type identifier > < variant string >

69  <variant string> ::= <variant>
  * < variant >

70                       <variant string> ; <variant>
  * < variant string > < variant >

71  <tag field> ::= <field ident> :
  * < field identifier > ; { set TAG$FD to TRUE }

72  <variant> ::= <case label list> : ( <field list> )
  * < case label list > < field list >

73
  * < empty >

74  <case label list> ::= <case label>
  * < case label >

75                        <case label list> ,
                          <case label>
  * < case label list > < case label >

76  <case label> ::= <constant>
  * < constant >
```

```
  * [ { set variant attributes } / { set CASE$STMT } ]

77  <set type> ::= SET OF <base type>
  * < base type > ; { enter type }

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

79  <file type> ::= FILE OF <type>
  * < type > ; { enter type }

80  <pointer type> ::= $ <type ident>
  * < type identifier > ; { enter type }

81  <vdp> ::=
  * < empty >

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

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

84                            <var declar string> ,
                              <var declar>
  * < variable declaration string >
  * < variable declaration >

85  <var declar> ::= <ident var string> : <type>
  * < identifier variable string > < type >
  * { set variable attributes }

86  <ident var string> ::= <identifier>
  * < identifier > ; { enter variable }

87                           <ident var string> ,
                             <identifier>
  * < identifier variable string > < identifier >
  * { enter variable }

88  <p fdp> ::=
  * < empty > { not implemented }

89              <porf declar>
  * < procedure or function declaration >
  * { not implemented }

90  <porf declar> ::= <proc or funct> ;
  * { not implemented }

91                    <porf declar> <proc or funct> ;
  * { not implemented }

92  <proc or funct> ::= <procedure declaration>
  * { not implemented }
```

```
93                          <function declaration>
  * { not implemented }

94  <procedure declaration> ::= <procedure heading>
                                <block>
  * { not implemented }

95  <procedure heading> ::= <proc id> ;
  * { not implemented }

96                            <proc id> (
                              <formal para sect list> ) ;
  * { not implemented }

97  <proc id> ::= PROCEDURE <identifier>
  * { not implemented }

98  <formal para sect list> ::= <formal para sect>
  * { not implemented }

99                              <formal para sect list>
                                ; <formal para sect>
  * { not implemented }

100 <formal para sect> ::= <para group>
  * { not implemented }

101                           VAR <para group>
  * { not implemented }

102                           FUNCTION <para group>
  * { not implemented }

103                           PROCEDURE <proc ident list>
  * { not implemented }

104 <proc ident list> ::= <identifier>
  * { not implemented }

105                         <proc ident list> ,
                            <identifier>
  * { not implemented }

106 <para group> ::= <para ident list> : <type ident>
  * { not implemented }

107 <para ident list> ::= <identifier>
  * { not implemented }

108                         <para ident list> ,
                            <identifier>
  * { not implemented }

109 <function declar> ::= <function heading> <block>
```

```
   * { not implemented }

110  <function heading> ::= <funct id> : <result type> ;
   * { not implemented }

111                          <funct id> (
                             <formal para list> )
                             : <result type> ;
   * { not implemented }

112  <funct id> ::= FUNCTION <identifier>
   * { not implemented }

113  <result type> ::= <type ident>
   * { not implemented }

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>
   * LBL ; { If Label address2 }

119                 <simple stmt>
   * < simple statement >

120  <unbal stmt> ::= <if clause> <stmt>
   * LBL ; { If Label address1 }

121                   <if clause> <true part> ELSE
                      <unbal stmt>
   * LBL ; { If Label address2 }

122  <if clause> ::= <if> <expression> THEN
   * NOT ; BLC ; { If Label address1 }

123  <if> ::= IF
   N/A

124  <true part> ::= <bal stmt>
   * < balanced statement >
   * BRL ; { If Label address2 } ; LBL ;
   * { If Label address1 }

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

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

127                    <procedure stmt>
   * < procedure statement >

128                    <repetitive stmt>
   * < repetitive statement >

129                    <case stmt>
   * < case statement >

130                    <with stmt>
   * < with statement >

131                    <read stmt>
   * < read statement >

132                    <write stmt>
   * < write statement >

133                    <goto stmt>
   * < goto statement >

134                    <compound stmt>
   * < compound statement >

135
   * < empty statement >

136  <assignment stmt> ::= <variable> := <expression>
   * < variable > < expression > ; LIT ;
   * { variable address }
   * [ STD/STDI/STDB/ ( CNAI/STDB ) ]

137  <variable> ::= <entire variable>
   * < entire variable >

138                <variable> $
   * { not implemented }

139                <variable> <lp> <express list> <rp>
   * { not implemented }

140                <variable> . <field ident>
   * { not implemented }

141  <entire variable> ::= <variable ident>
   * < variable identifier >

142  <variable ident> ::= <identifier>
   * < identifier > ; { set variable location/type }

143  <express list> ::= <expression>
   * { not implemented }
```

```
144                         <express list> , <expression>
   * { not implemented }

145  <expression> ::= <simple expression>
   * < simple expression >

146                      <simple expression>
                         <relational operator>
                          <simple expression>
    * < simple expression > < relational operator >
    * < simple expression > ; [ EQLI/NEQI/LEQI/GEQI/
    * LSSI/GRTI/ ( IN not implemented ) /EQLB/NEQB/
    * LEQB/GEQB/LSSB/GRTB ]

147  <relational operator> ::= =
   * { set operator type }

148                                  < >
   * { set operator type }

149                                  < =
   * { set operator type }

150                                  > =
   * { set operator type }

151                                  <
   * { set operator type }

152                                  >
   * { set operator type }

153                                  IN
   *  { set operator type }

154  <term> ::= <factor>
   * < factor >

155              <term> <multiplying operator> <factor>
   * < term > < mulitplying operator > < factor >
   * [ MULI/MULB/ ( CNVI ; CN2I ; DIVB ) /DIVB
   *   DIVI/DCRI/AND ] ; { MOD not implemented }

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

157                                   /
   * { set operator type }

158                                   DIV
   * { set operator type }

159                                   MOD
   * { set operator type }
```

```
160                                 AND
   * { set operator type }

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

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

163                                 <simple expression>
                                    <adding operator> <term>
   * < simple expression > < adding operator > < term >
   * [ ADDI/ADDB/SUBI/SUBB/BOR ]

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

165                                 -
   * { set operator type }

166                                 OR
   * { set operator type }

167  <factor> ::= <variable>
   * < variable > ; [ LDII ; { interger value }/
   * LDIB ; { BCD real value }/
   * NEGI/NEGB/LITA ; { variable address }
   * LOD/LODI/LODB ]

168                 <variable> ( <actual para list> )
   * { not implemented }

169                 ( <expression> )
   * < expression >

170                 <set>
   * { not implemented }

171                 NOT <factor>
   * < factor > ; NOT

172                 <number>
   * < number >
   * [ LDII ; { integer value } /
   *   LDIB ; { BCD real value } ]

173                 NIL
   N/A

174                 <string>
   * { not implemented }

175  <actual para list> ::= <actual para>
   * { not implemented }
```

```
176                          <actual para list> ,
                             <actual para>
   * { not implemented }

177  <set> ::= <lp> <element list> <rp>
   * { not implemented }

178  <element list> ::=
   * < empty >

179                       <xelement list>
   * { not implemented }

180  <xelement list> ::= <element>
   * { not implemented }

181                        <xelement list> , <element>
   * { not implemented }

182  <element> ::= <expression>
   * { not implemented }

183                 <expression> .. <expression>
   * { not implemented }

184  <goto stmt> ::= <goto> <label>
   * < goto > < label >
   * { label address }

185  <goto> ::= GOTO
   * BRL

186  <compound stmt> ::= <begin> <stmt lists> END
   * < begin > < statement lists >

187  <begin> ::= BEGIN
   N/A

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

189                    <stmt lists> ; <stmt>
   * < statement lists > < statement >

190  <procedure stmt> ::= <procedure ident>
   * { not implemented }

191                        <procedure ident> (
                           <actual para list> )
   * { not implemented }

192  <procedure ident> ::= <identifier>
   * { not implemented }

193  <actual para> ::= <expression>
```

```
   * { not implemented }

194  <rec variable list> ::= <variable>
   * { not implemented }

195                          <rec variable list> ,
                             <variable>
   * { not implemented }

196  <read stmt> ::= <read head> ( <io list> )
   * < read head > < io list >

197  <read head> ::= READ
   * { set WRITE$STMT to FALSE }
   * { set ALLOCATE to FALSE ( don't skip to new line) }

198                  READLN
   * { set WRITE$STMT to FALSE }
   * { set ALLOCATE to TRUE ( skip to new line ) }

199  <write stmt> ::= <write head> ( <io list> )
   * < write head > < io list >
   * [ if ALLOCATE then DUMP ]

200                   <write head>
   * < write head >
   * [ if ALLOCATE then DUMP ]

201  <write head> ::= WRITE
   * { set WRITE$STMT to TRUE }
   * { set ALLOCATE to FALSE }

202                   WRITELN
   * { set WRITE$STMT to TRUE }
   * { set ALLOCATE to TRUE }

203  <io list> ::= <file ident> * , <var list>
   * { not implemented }

204                <var list>
   * < variable list >

205  <var list> ::= <variable>
   * < variable > ; [ WRVI/WRVB/RDVI/RDVB
   * ( STDI/STDB - read statements only ) ]

206                   <string>
   * < string > ; WRVS ; { string }
   * { write statement only }

207                   <var list> , <variable>
   * < variable list > < variable >
   * [ WRVI/WRVB/RDVI/RDVB ( STDI/STDB -
   * read statement only ) ]
```

```
208                    <var list> , <string>
   * < variable list > < string >
   * WRVS ; { string } ( write statement only )

209  <case stmt> ::= <case express>
                     <case list elemt list> END
   * { not implemented }

210  <case express> ::= CASE <expression> OF
   * { not implemented }

211  <case list elemt list> ::= <case list elemt>
   * { not implemented }

212                                <case list elemt list> ;
                                   <case list elemt>
   * { not implemented }

213  <case list elemt> ::=
   * < empty >

214                           <case label list> : <stmt>
   * { not implemented }

215  <repetitive stmt> ::= <while stmt>
   * < while statement >

216                           <repeat stmt>
   * < repeat statement >

217                           <for stmt>
   * < for statement >

218  <with stmt> ::= <with> <rec variable list> <do>
                     <bal stmt>
   * { not implemented }

219  <with> ::= WITH
   * { not implemented }

220  <do> ::= DO
   * { not implemented }

221  <while stmt> ::= <while> <expression> <do>
                      <bal stmt>
   * { not implemented }

222  <while> ::= WHILE
   * { not implemented }

223  <for stmt> ::= <for> <control variable> :=
                    <for list> <do> <bal stmt>
   * { not implemented }

224  <for> ::= FOR
```

```
   * { not implemented }

225  <for list> ::= <initial list> <to> <final value>
   * { not implemented }

226                <initial value><downto> <final value>
   * { not implemented }

227  <control variable> ::= <identifier>
   * { not implemented }

228  <initial value> ::= <expression>
   * { not implemented }

229  <final value> ::= <expression>
   * { not implemented }

230  <repeat stmt> ::= <repeat> <stmt lists> <until>
                       <expression>
   * < repeat > < statement lists > < until >
   * < expression > ; NOT ; BLC ; { set REPEAT$LBL }

231  <repeat> ::= REPEAT
   * LBL ; { REPEAT$LBL }

232  <until> ::= UNTIL
   N/A

233  <to> ::= TO
   * { not implemented }

234  <downto> ::= DOWNTO
   * { not implemented }
```

```
*100h:   /*load point for compiler*/

/*****************************************************************************
/*****************************************************************************
/***                    s y s t e m   l i t e r a l s                      ***
/*****************************************************************************
/*****************************************************************************

declare lit     literally 'literally',
        cr              lit '13',
        lf              lit '0ah',
        dcl             lit 'declare',
        pos             lit '0',
        neg             lit '1',
        tab             lit '09h',
        proc            lit 'procedure',
        bdos            lit '5h',           /*entry point to disk op. sys*/
        boot            lit '0',            /* exit to return to op. sys. *
        true            lit '1',
        addr            lit 'address',
        minusx          lit '2dh',
        rfile           lit '20',
        false           lit '0',
        maxint          lit '32767',
        bcdsize         lit '8',
        andsign         lit '26h',
        fileeof         lit '1',
        eolchar         lit '0dh',
        max$nest        lit '3',
        forever         lit 'while true',
        comment         lit '0',
        varcsize        lit '100',
        hashmask        lit '127',
        formmask        lit '7',
        type$entry      lit '2',
        var$entry       lit '3',
        type$dcle       lit '7',
        typemask        lit '00111000b',
        contchar        lit '5ch',
        codesize        addr,       /* used to count size of code area */
        identsize       lit '32',
        eoffiller       lit '1ah',
        statesize       lit 'address',
        indexsize       lit 'address',
        pstacksize      lit '48',           /*'stack size for parser */
        first$time      byte  initial (true),
        identifier      lit '62',
        intrecsize      lit '128',
        file$entry      lit '6',
        labl$entry      lit '0',
         cons$entry      lit '1',
        maxoncount      lit '25',
        conbuffsize     lit '82',
        stringdelim     lit '27h',
        hashtblsize     lit '128',
        questionmark    lit '3fh',
        max$num$arry$dimen lit '5',
        arry$nest       lit '4',
        cons$str$type   lit '3',
        cons$num$type   lit '0',
        cons$ident$type lit '1',
        cons$sident$type lit '2',
        sourcerecsize   lit '128';

   /* number types */
dcl     ord$type        lit '0',
```

```
        charStype       lit '2',
        integerStype    lit '1',
        realStype       lit '2',
        unsignSexpon    lit '3',
        signedSexpon    lit '4',
        complexStype    lit '4',
        stringStype     lit '4',
        booleanStype    lit '5',
        signtype        byte,
        constStype      byte;     /* type of constant */

dcl     maxrno          lit '192',        /* max read count */
        maxlno          lit '251',        /* max look count */
        maxpno          lit '276',        /* max push count */
        maxsno          lit '510',        /* max state count */
        starts          lit '1',          /* start state */
        prodno          lit '234',        /* number of productions */
        eofc            lit '25',         /* eof */
        numberc         lit '58',         /* number */
        stringc         lit '59',         /* string */
        termno          lit '62';         /* terminal count */

dcl     sbloc           addr     initial(80h),
        form            byte,
        expon           byte,
        vecptr          byte,
        lineno          addr,
        typenum         byte,
        rfcbaddr        addr     initial (5ch),
        constSptr       byte,
        startbdos       addr initial(6h), /*addr of ptr to top of bdos*/
        lablcount       addr initial(0),  /* number of labels used */
        max             based startbdos addr,
        errorcount      addr     initial (0),
        typeSaddr       addr,
        typeSloct       addr,
        varSptr         byte,
        varStype(10) byte,
        varSsign(10) byte,
        expStype(11) byte, /* type of expression */
        expStypeSaddr(11) addr, /* addr of scalar parent type */
        expSptr         byte initial (0),
        opStype(10)  byte,
        opSptr          byte initial (0),
        caseSstmt       byte initial (false), /* in case stmt */
        writeSstmt      byte initial (false), /* in write stmt */
        repeatSlbl(10) addr,
        repeatSptr      byte initial (0),
        allocate        byte, /* true or false */
        allcSbasicStype byte,
        arrySqty(maxSnumSarrySdimen) addr,
        varSbase(10)  addr,
        varSbase1(10) addr,
        typeSindx       byte,
        allcSqty        addr,
        typeform        byte,
        allocSaddr      addr     initial (0),
        typeSordSnum    byte,
        parentStype     addr,
        constSindx      byte,
        lookupSaddr     addr,
        constSvec(4)    byte,
        constSvalue(162) byte,
        constSpnShash(4) byte,
        constSpnSptr    byte,
        constSpnSsize(4) byte,
        integerSdiff    addr,
        subrSval(2)     addr,
        subrStype(2)    byte,
        subrSptr        byte,
        subrStypeSaddr  addr,
        subrSform       byte,
```

```
        subr$pn$sign      byte,
        arry$base         addr,
        arry$ptr          byte,
        arry$dim$ptr      byte,
        ptrptr            byte,
        if$ptr            byte initial (255),
         if$lbl(1)        addr, /* if statement stack */
         tag$fd(max$nest) byte,
        var$cas$tp(max$nest) addr,
        var$cas$val(max$nest) addr,
        rec$var$typ(max$nest) byte,
        rec$nst      byte  initial (255),
        record$ptr       byte,
        rec$addr(6)      addr,
        rec$par$adr(max$nest) addr,
        variant$part(max$nest)     byte,
        fxd$ofst$bse(max$nest) addr,
        var$ofst$bse(max$nest) addr,
        cur$ofst(max$nest)       addr,
        num$arry$dimen(max$num$arry$dimen) byte,
        arry$dimen(25) addr,
        ary$dm$adr$ptr byte,
        const$num$type(4) byte,
        bcdnum(bcdsize) byte,
        rfcb              based    rfcbaddr (33) byte,
        no look           byte,
        lineptr           byte     initial (0),
        buffptr           byte     initial (255),
        wfcb(33)          byte     initial (0,'        ','pin',0,0,0,0),
        sourceptr         byte     initial (sourcerecsize),
        no intfile        byte     initial (false),
        sourcebuff        based    sbloc   (sourcerecsize)  byte,
        production        byte,
        prv$sbtbl$entry addr,
        cursourcerecsize byte  initial (sourcerecsize),
        linebuff(conbuffsize)  byte,
        diskoutbuff(intrecsize)  byte;

dcl     init$symb$tbl data(0,0,0,0,42h,7,'i','n','t','e','g','e',
          'r',0,0,0,0,4ah,4,'r','e','a','l',0,0,0,0,52h,4,'c','h','a',
          'r',0,0,0,0,5ah,7,'b','o','o','l','e','a','n',0,0,0,0,0eh,
          5,'i','n','p','u','t',0,0,0,0,1eh,6,'o','u','t','p','u','t',
          0,0,0,0,09h,4,'t','r','u','e',0,0,0,0,0,0,0,0,09h,5,'f','a',
          'l','s','e',0,0,1,0);


/*****************************************************************************
/*****************************************************************************
/***                    scanner  global  variables                         ***
/*****************************************************************************
/*****************************************************************************
dcl     token             byte,              /* type of token just scanned *
        hashcode          byte,              /* has value of current token *
        nextchar          byte,              /*current character fm getchar*
        cont              byte,              /*indx full accum. still more *
        accum(identsize)  byte;              /* holds current token */


/*****************************************************************************
/*****************************************************************************
/***                    symbol table global variables                      ***
/*****************************************************************************
/*****************************************************************************

dcl     base              addr,              /*base of current entry       */
        hashtable(hashtblsize) addr,
        sbtbltop          addr,              /*current top of table (sym)  */
        sbtbl             addr,
        ptr based         base byte,         /* 1st byte of entry          */
        aptraddr          addr,     /* utility variable to access table  */
        addrptr based aptraddr addr,
        byteptr based aptraddr byte,
```

```
        printname       addr,       /* set prior to lookup or enter      */
        symhash         byte;


declare read1 data(0,56,25,12,14,62,62,34,60,61,62,59,62,62,11,8,8,8,13
     ,62,62,62,62,62,59,62,59,62,14,3,4,9,58,59,62,3,4,5,9,33,37,43,51,5
     ,58,59,62,3,4,9,58,59,62,3,16,31,32,58,59,62,22,62,62,62,3,4,9,16,3
     ,32,58,59,62,22,62,16,62,58,33,37,43,53,35,62,62,62,62,62,20,29,35
     ,38,39,42,44,48,49,52,54,57,62,22,41,62,45,34,60,61,58,62,6,10,26,2
     ,30,62,20,29,35,38,39,42,44,48,49,52,54,57,58,62,1,13,44,7,7,3,8,15
     ,3,13,14,11,1,5,16,18,1,3,5,16,8,3,36,36,22,40,19,7,15,28,7,7,11,7,
     ,8,3,8,28,8,47,22,22,3,11,17,13,14,8,8,17,3,11,24,50,8,8,7,11,11,8
     ,11,13,11,13,11,13,11,3,46,8,7,11,7,11,18,11,13,11,17,11,19,2,4,9,1
     ,14,21,23,4,9,23,8,11,13,8,28,7,8,7,8,0,0,0,0,0);

declare look1 data(0,12,14,0,35,62,0,62,0,62,0,62,0,35,62,0,8,28,47,0,8
     ,28,0,7,8,28,0,14,0,8,28,0,7,8,28,0,7,8,28,0,8,28,36,47,0,36,0,35,6
     ,0,15,0,1,5,16,18,0,13,0,6,0,0,0,0,17,0,41,0,45,0,34,0,44,0,6,19,26
     ,27,30,0,6,10,26,27,30,0,6,10,26,27,30,0,8,28,0,8,47,0,36,0,11,0,11
     ,0,1,3,5,16,0,11,19,0,7,11,0,11,19,0,7,11,0,8,28,36,47,0,36,0,8,28
     ,47,0,15,0,8,0,3,0,44,0,8,28,0,11,0,8,0,8,0,11,0,3,0,46,0,46,0,46,0
     ,2,4,9,12,14,21,23,0,4,9,23,0);

declare apply1 data(0,0,0,0,5,30,0,179,182,0,0,0,29,75,97,0,0,0,21,0,0
     ,27,28,37,52,54,60,62,158,0,98,0,27,28,35,37,45,47,52,53,54,58,59,6
     ,61,62,87,158,0,0,0,22,0,0,45,47,59,61,0,35,58,87,0,15,46,48,50,67
     ,132,0,0,0,0,0,0,76,0,0,73,121,122,123,124,125,126,0,152,159,0,0,35,0
     ,0,0,14,0,0,24,0,0,3,33,67,0,24,0,62,0,0,28,0,27,158,0,37,0,0,0,0,0
     ,0,0,23,0,0,0,0,155,0,0,0,0,8,0,26,0,0,66,80,0,0,0,0,0,50,0,0,25,49
     ,128,130,0,69,70,83,84,85,128,129,0,69,0,70,83,84,85,129,0,0,129,0,
     ,0,0,10,11,25,36,41,43,49,69,70,83,84,85,103,104,111,128,129,130,0,
     ,0,0,9,39,40,55,56,57,68,86,88,89,91,108,109,110,0,0,99,168,0,0,188
     ,0,0,63,190,0,13,0,0,0,0,39,0,0,0,107,0,0,111,0,0,0,42,0,0,0,0,0,0
     ,11,0,43,0,0,0,0,27,0,0,0,0,116,137,0,0,0,0,0,0,0,0,110,0,0,0,0,0,0
     );

dcl     read2(236) addr initial
                   (0,78,277,424,425,281,316,66,80,82,383,482,212,314,43
     ,273,279,372,50,363,317,342,384,381,484,418,402,418,426,194,300,301
     ,294,298,299,6,300,15,301,64,71,73,75,208,294,298,209,6,300,301,294
     ,298,209,9,326,449,63,448,450,418,58,362,383,211,9,300,301,326,449
     ,63,448,450,418,59,307,326,293,287,64,71,73,208,67,342,280,388,373
     ,380,399,500,68,461,473,495,463,498,477,474,507,478,210,61,72,503,7
     ,65,79,81,295,299,432,433,436,434,435,418,399,500,68,461,473,495,46
     ,498,477,474,507,478,287,210,2,401,463,472,475,7,371,54,8,46,53,41,
     ,414,326,56,3,13,414,326,195,10,206,207,486,398,496,445,55,332,348
     ,17,30,18,16,198,5,199,462,199,508,204,205,11,40,327,347,52,386,387
     ,327,284,29,509,510,366,367,315,32,39,201,37,203,37,51,34,48,38,12
     ,75,196,467,42,444,42,57,31,45,35,327,36,496,193,440,441,202,423,42
     ,442,440,441,442,197,33,47,200,485,19,26,20,26,0,0,0,0,0);

dcl     look2(171) addr initial
                   (0,4,4,427,14,14,252,21,289,22,303,23,358,24,24,253
     ,254,254,254,25,255,255,27,256,256,256,28,44,428,257,257,49,258,258
     ,258,60,259,259,259,62,260,260,260,260,69,261,70,77,77,262,299,314
     ,418,418,418,418,468,342,314,281,418,83,84,85,263,88,92,264,94,265
     ,95,266,267,96,100,100,100,100,100,437,101,101,101,101,101,438,102
     ,102,102,102,102,439,268,268,104,269,269,111,400,391,119,480,120,47
     ,122,122,122,122,443,470,470,123,481,481,124,471,471,125,483,483,12
     ,270,270,270,270,128,271,129,272,272,272,130,139,458,145,333,151,47
     ,365,155,273,273,158,165,455,166,344,167,343,172,379,173,466,174,27
     ,179,275,182,276,184,184,184,184,184,184,184,421,185,185,185,422);

dcl     apply2(273) addr initial
                    (0,0,247,146,142,143,144,385,370,105,218,160,286,285
     ,460,106,217,127,291,290,154,352,352,352,292,318,352,352,352,115,29
     ,296,98,98,98,98,98,98,98,98,98,98,98,98,98,98,98,98,99,219,173,305
     ,304,118,361,338,355,331,306,330,354,330,308,356,389,382,389,149,15
     ,313,311,164,312,309,320,319,321,87,89,89,89,89,89,89,216,415,453,9
     ,181,329,328,325,322,141,140,238,337,336,187,416,341,153,340,334,33
     ,244,243,132,346,345,169,169,170,351,350,323,353,324,310,220,186,36
     ,359,180,107,240,163,162,368,249,114,192,191,375,374,245,377,378,37
```

```
,171,369,248,117,157,156,282,465,490,396,393,464,394,394,494,497,49
,226,400,391,397,392,285,235,235,235,234,36,134,133,236,395,402
,231,231,121,232,233,231,121,121,121,121,121,121,230,121,121,121,12
,121,229,413,417,152,138,237,420,459,412,504,135,136,237,419,505,50
,137,505,469,188,222,223,221,190,251,250,168,447,431,430,177,176,44
,159,242,457,456,409,97,390,410,224,148,147,403,246,452,451,183,407
,131,408,239,113,112,228,227,405,241,189,488,487,404,406,103,215,21
,213,491,109,498,98,116,179,161,502,501,492,225,103,91,110);

dcl     index1(511) addr initial
                    (0,1,2,21,3,5,6,7,7,64,11,11,64,64,82,13,14,15,16,17
,18,76,74,61,82,116,7,30,30,77,5,19,20,21,22,47,115,30,23,64,64,24
,64,26,28,35,13,35,13,116,13,29,30,35,30,64,64,64,47,35,30,35,30,53
,69,61,62,63,64,88,88,73,74,75,76,77,78,82,84,85,62,86,87,88,88,88
,64,47,64,64,101,64,102,103,104,105,106,77,108,53,110,110,110,115
,116,130,131,132,64,64,64,116,133,134,135,137,156,138,140,141,141
,142,146,142,142,142,142,150,116,88,116,151,13,152,153,154,155,156
,157,158,159,160,161,163,164,165,166,167,169,171,172,173,174,176,17
,106,178,179,30,180,181,183,185,186,187,189,190,190,53,191,193,195
,197,198,199,200,201,203,205,199,206,208,199,210,212,219,222,223,64
,225,53,227,229,1,4,7,9,11,13,16,20,23,27,29,32,36,40,45,47,50,52,5
,59,61,62,63,64,66,68,70,72,74,80,86,92,95,98,100,102,104,109,112
,115,118,121,126,128,132,134,136,138,140,143,145,147,149,151,153,15
,157,159,167,339,339,411,489,349,411,349,349,411,411,339,454,302,28
,357,364,411,411,411,411,411,489,283,283,283,1,2,2,3,4,7,10,10,11,1
,12,16,16,17,17,18,20,21,21,21,21,21,30,32,32,49,49,50,50,51,53,54
,54,54,59,59,59,63,70,71,71,72,73,73,74,74,74,74,76,77,85,88,88,89
,91,92,93,93,93,95,95,96,96,98,98,99,103,103,105,105,107,108,108,11
,110,113,115,116,117,118,119,119,120,120,121,123,123,124,124,125,12
,126,126,128,129,129,130,131,131,133,133,133,133,135,135,136,139,13
,140,141,141,142,143,145,146,146,146,151,151,159,159,161,167,168,17
,171,171,171,171,171,171,171,171,171,171,172,173,173,173,173,192,19
,171,171,171,171,171,171,171,171,171,171,172,173,173,173,173,192,19
,194,194,195,195,210,210,210,210,210,210,210,210,211,211,214,214,21
,214,215,215,215,217,217,217,218,218,218,218,229,230,232,233,233,235,235,236,23
,223,224,224,225,225,226,226,228,229,230,232,233,233,235,235,236,23
,239,239,240,241,241,242,242,243,243,244,244,246,246,246,246,246,248,24
,250,250,251,251,253,253,253,254,255,256,259,260,261,262,263,263,26
,263,266,268,269,270,271,272);

declare index2 data(0,1,1,1,2,1,1,4,4,9,2,2,9,9,2,1,1,1,1,1,1,1,1,2,1
,4,5,5,1,1,1,1,1,1,6,1,5,1,9,9,2,9,2,1,12,1,12,1,14,1,1,5,12,5,9,9,
,6,12,5,12,5,7,1,1,1,1,1,9,13,13,1,1,1,1,1,1,4,2,1,1,1,1,1,1,13,13,13,9,6
,9,9,1,9,1,1,1,1,2,1,2,7,5,5,5,5,1,14,1,1,1,1,1,1,1,1,2,1,1,1,1,2,1,1,
,4,4,3,3,3,3,1,14,13,14,1,1,1,1,1,1,1,1,1,1,1,1,1,2,1,1,1,2,2,1,2,2,2,2,
,1,1,2,1,1,5,1,2,2,1,1,2,1,1,1,1,7,2,2,2,1,1,1,1,1,1,1,2,2,1,1,2,2,1,2,7,3,
,2,9,2,7,2,2,3,3,2,2,2,3,4,3,4,2,3,4,4,5,2,3,2,5,2,2,1,1,1,2,2,1,2,2,2,2,
,6,6,6,3,3,2,2,2,5,3,3,3,3,3,5,2,4,2,2,2,2,2,3,2,2,2,2,2,2,2,2,8,4,14,2
,25,27,28,49,60,62,69,70,77,88,92,94,95,96,104,111,128,129,130,158
,174,179,182,3,5,7,0,0,5,0,2,0,2,0,0,2,0,0,2,2,0,0,5,0,0,0,2,0,0,2,0,0,2,0,0,1,2,0,0,1,1,
,2,2,0,0,0,0,0,0,0,0,2,0,2,2,0,1,0,0,0,0,5,0,0,0,2,0,0,2,0,0,1,2,0,0,0,0,0,
,0,0,2,0,4,3,0,2,1,4,0,0,2,0,2,0,2,0,2,1,0,2,0,2,2,0,2,0,0,2,0,0,0,0,0,0,2,
,1,0,2,0,1,1,1,1,0,2,2,2,0,2,1,3,6,1,0,0,0,1,1,1,0,0,0,0,2,0,3,0,0,0,2,3,0,
,0,0,0,0,0,2,0,1,3,2,0,0,0,2,0,2,0,0,0,2,0,2,1,0,2,0,0,2,0,3,0,0,0,1,2,
,0,0,0,3,2,0,1,0,0,0,0,2,2,0,0,2,0,2,0,0,0,3,0,0,3,0,5,0,2,2,0,0,0,3,0,
,3,0,0,0,3,0,0,0,0,2,2,2,2,0,2,0,2,0,0,0,3,0,0,0,3,0,5,0,2,2,0,0,0,3,0,
,0,0);

/*****************************************************************
/*****************************************************************
/***                  g l o b a l   p r o c e d u r e s        ***
/*****************************************************************
/*****************************************************************

mon1: proc(f,a);
    dcl     f byte,
    a addr;
    go to bdos;
end mon1;

mon2: proc (f,a) byte;
```

```
    dcl     f byte, a addr;
    go to bdos;
end mon2;


mon3:proc;                                  /*used to return to the system*
    goto boot;
end mon3;


move: proc (a,b,l);                         /*moves fm a to b for l bytes *
    dcl     (a,b) addr,                     /* l < 255 bytes */
    (s based a, d based b,1) byte;
    do while (l:=l - 1) <> 255;
       d=s;
       b=b + 1;
       a=a + 1;
    end;
end move;


fill: proc (a,char,n);                      /* move char to a n times */
    dcl a addr,(char,n,dest based a) byte;
    do while (n := n -1) <> 255;
       dest = char;
       a = a + 1;
    end;
end fill;


read: proc;
    dcl toggle(3) byte;
    toggle = 1;
    call mon1(10,.toggle);
end read;


printchar: proc(char);
    dcl     char byte;
    call mon1(2,char);
end printchar;


print: proc(a);
    dcl     a addr;
    call mon1(9,a);
end print;


diskerr: proc;
    do;
       call print(.'de    $');
5       goto boot;
    end;
end diskerr;

setup$int$file:proc;
   if nointfile then  /* only make file if this toggle off */
      return;
   call move(.rfcb,.wfcb,9);
      wfcb(32) = 0;
   call mon1(19,.wfcb);
   if mon2(22,.wfcb) = 255 then
      call diskerr;
end setup$int$file;

write$int$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 writeSintSfile;

emit: proc(objcode);
   dcl objcode byte;
   if (buffptr := buffptr+1) >= intrecsize then
   /* write to disk */
      do;
        call writeSintSfile;
         buffptr = 0;
      end;
   diskoutbuff(buffptr) = objcode;
end emit;

generate: proc(objcode);
   dcl objcode byte;
      codesize = codesize+1;
         call emit(objcode);
end generate;

closeSintSfile: proc;
  /* closes a file */
  if mon2(16,.wfcb) = 255 then
     call diskerr;
end closeSintSfile;


openSsourcefile: proc;
    call move(.'pas',rfcbaddr+9,3);
    rfcb(32),rfcb(12) = 0;
     if mon2(15,rfcbaddr) = 255 then
       do;
          call print(.'no source file $');
          go to boot;
       end;
end openSsourcefile;


rewindSsourceSfile:proc;      /* cp/m does not require any action */
    return;                   /* prior to reopening */
end rewindSsourceSfile;


readSsourceSfile:proc byte;
    dcl     dcnt byte;
    if (dcnt:=mon2(rfile,rfcbaddr)) > fileeof then
       call diskerr;
    return dcnt;
end readSsourceSfile;


crlf: proc;
    call printchar(cr);
    call printchar(lf);
end crlf;


printdec: proc(value);
    dcl     value addr, i byte, count byte;
    dcl     deci(4) addr initial(1000,100,10,1);
    dcl     flag byte;
    flag = false;
    do i = 0 to 3;
    count = 30h;
       do while value >= deci(i);
          value = value - deci(i);
          flag= true;
          count = count + 1;
       end;
       if flag or (i>= 3) then
          call printchar(count);
       else
```

```
            call printchar(' ');
        end;
        return;
    end printdec;


    print$prod:proc;
        call print(.' prod = $');
        call print$dec(production);
        call crlf;
    end print$prod;


    print$token:proc;
        call print(.' token = $');
        call print$dec(token);
        call crlf;
    end print$token;


    clear$line$buff:proc;
        call fill(.linebuff,' ',conbuffsize);
    end clear$line$buff;


    listline: proc(length);
        dcl     (length,i) byte;
        call print$dec(lineno);
        call print$char(' ');
        do i = 0 to length;
           call printchar(linebuff(i));
        end;
        call crlf;
    end listline;



    /*************************************************************************/
    /*************************************************************************/
    /***                    p a r s e r    v a r i a b l e s                 ***/
    /*************************************************************************/
    /*************************************************************************/

     dcl    listprod        byte        initial(false),
            lowertoupper    byte        initial(true),
            listsource      byte        initial(false),
            debugin         byte        initial(false),
            listtoken       byte        initial(false),
            compiling       byte;



    /*************************************************************************/
    /*************************************************************************/
    /***                  s c a n n e r     p r o c e d u r e s            ***/
    /*************************************************************************/
    /*************************************************************************/

    getchar: proc byte;
        dcl addeof data ('eof', eolchar,lf); /* add to end if left off */


        next$source$char: proc byte;
            return sourcebuff(sourceptr);
        end next$source$char;


        checkfile: proc byte;
            do forever;
               if (sourceptr:=sourceptr+1)>=cursourcerecsize then
                  do;
```

```
                sourceptr=0;
                if readSsourceSfile=fileeof then
                   return true;
             end;
          if (nextchar:=nextSsourceSchar)<>lf then
             return false;
       end;
    end checkfile;


    if checkfile or (nextchar = eoffiller) then
       do;                                  /* eof reached */
          call move(.addeof,sbloc,5);
          sourceptr = 0;
          nextchar=nextSsourceSchar;
       end;
    linebuff(lineptr:=lineptr + 1)=nextchar;          /*output line*/
    if nextchar  = eolchar  then
       do;
          lineno = lineno + 1;
          if listsource  then
             call listline(lineptr-1);
          lineptr = 0;
          call clearlinebuff;
       end;
    if nextchar = tab then
       nextchar = ' ';
    return nextchar;
end getchar;


getnoblank: proc;
    do while((getchar = ' ') or (nextchar = eoffiller));
    end;
end getnoblank;


title:proc;                                 /* compiler version */
    call crlf;
    call print(.'toggles set$');
    call crlf;
    call print(.'pascal-m vers 1.0$');
    call crlf;
    call crlf;
end title;


printSerror:proc;
    call printdec(errorcount);
    call printchar(' ');
    call print(.'error(s) detected$');
    call crlf;
end printSerror;


error: proc(errcode);
    dcl errcode addr,
    i       byte;
    errorcount=errorcount+1;
    call print(.'***$');
    call printSdec(lineno);
    call print(.'  error #');
    call printchar(' ');
    call printchar(high(errcode));
    call printchar(low(errcode));
    call print(.' near $');
    call printchar(' ');
    do i = 1 to accum;
       call printchar(accum(i));
    end;
    call crlf;
```

```
        call print(.'at error $');
        call printchar(' ');
        call printStoken;
        call print(.'at error $');
        call printchar(' ');
        call printSprod;
        if token=eofc then
           do;
              call printSerror;
              call mon3;
           end;
    end error;


    initializeSscanner: proc;
        dcl count byte;
        call openSsourcefile;
        lineno,lineptr = 0;
        call clearSlineSbuff;
        sourceptr = 128;
        call getnoblank;
        do while nextchar = '$';
           call getSnoSblank;
           if (count := (nextchar and 5fh) - 'a') <= 4 then
              do case count;
                 listsource = true;
                 listprod = true;
                 nointfile = true;
                 listtoken = true;
                 debugin = true;
              end;                              /* of case */
           call getnoblank;
        end;
    end initializeSscanner;



/*************************************************************************/
/*************************************************************************/
/***                          s c a n n e r                            ***/
/*************************************************************************/
/*************************************************************************/

    scanner: proc;
        dcl flag byte;


        putinaccum: proc;
            if not cont then
               do;
                  accum(accum := accum + 1) = nextchar;
                  hashcode = (hashcode+nextchar) and hashmask;
                  if accum = 31 then cont = true;
               end;
        end putinaccum;


        putandget: proc;
            call putinaccum;
            call getnoblank;
        end putandget;


        putandchar: proc;
            call putinaccum;
            nextchar = getchar;
        end putandchar;


        numeric: proc byte;
            return(nextchar - '0') <= 9;
        end numeric;
```

```
lowercase: proc byte;
    return (nextchar >= 61h) and (nextchar <= 7ah);
end lower$case;


decimalpt:proc byte;
    return nextchar='.';
end decimalpt;


conv$to$upper:proc;
    if lowercase and lowertoupper then
       nextchar=nextchar and 5fh;
end conv$to$upper;


letter: proc byte;
    call conv$to$upper;
    return ((nextchar - 'a') <= 25) or lowercase;
end letter;


alphanum: proc byte;
    return numeric or letter ;
end alphanum;


spoolnumeric: proc;
    do while numeric;
       call putandchar;
    end;
end spoolnumeric;


setup$next$call: proc;
    if nextchar = ' ' then
       call getnoblank;
    cont = false;
end setup$next$call;


lookup: proc byte;

   dcl maxrwlng lit  '9';

   dcl vocab data(0,'.','<','(','+','$','*',')',';','-','/',',','>'
   ,':','=','..','(*','*)',':=','do','if','in','of','or','to','eof'
   ,'and','div','end','for','mod','nil','not','set','var','case'
   ,'else','file','goto','read','then','type','with','array','begin'
   ,'const','label','until','while','write','downto','packed'
   ,'readln','record','repeat','<empty>','program','writeln'
   ,'function','procedure');

   dcl vloc data(0,1,15,35,65,97,132,162,183,191,200);

   dcl vnum data(0,1,15,25,35,43,50,55,60,61);

   dcl count data(0,13,9,9,7,6,4,2,0,0);

 dcl ptr addr, (field based ptr) (9) byte;
 dcl i byte;

     compare: proc byte;
         dcl i byte;
         i = 0;
         do while (field(i) = accum(i := i + 1)) and i <= accum;
         end;
         return i > accum;
     end compare;


     if accum > maxrwlng then
```

```
            return false;
         ptr=vloc(accum)+.vocab;
         do i=vnum(accum) to (vnum(accum)+count(accum));
            if compare then
               do;
                  token=i;
                  if i = 53 then

      /*****************************************/
      /* the following code sets up storage & */
      /* pointers for record entries in the   */
      /* symbol table.                        */
      /*****************************************/

           do;
              rec$nst=rec$nst+1;
              aptraddr,rec$par$adr(rec$nst)=sbtbl;
              addrptr=0000h;
              aptraddr=aptraddr+2;
              addrptr=prv$sbtbl$entry;
              prv$sbtbl$entry=sbtbl;
              aptraddr=aptraddr+2;
              byteptr=1fh;
              sbtbl=sbtbl+9;
     /* record initiazations */
     variant$part(rec$nst),tag$fd(rec$nst)=false;
     fxd$ofst$bse(rec$nst)=0000h;
     var$ofst$bse(rec$nst)=0000h;
     cur$ofst(rec$nst)=0000h;
     var$cas$val(rec$nst)=0000h;
     record$ptr=-1;
            end;
                  return true;
               end;
            ptr=ptr+accum;
         end;
         return false;
      end lookup;


/*************************************************************************/
/*************************************************************************/
/***                    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;
                 if cont then return;
              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$type;
      do while nextchar='0'; /*elim leading zeros*/
         nextchar=getchar;
      end;
      call spoolnumeric;
      if decimalpt then
         do;
            call putandchar;
            typenum = real$type;
            call spoolnumeric;
         end;
      /* this takes care of expon. form */
      if nextchar = 'e' then
         do;
            typenum = unsign$expon;
            call putandchar;
            if nextchar = '-' or nextchar = '+' then
               do;
                  call putandchar;
                  typenum = signed$expon;
               end;
            call spoolnumeric;
         end;
      if accum = 0 then
        hashcode,accum(accum:=1) = '0';
      call setup$next$call;
      return;
   end;                  /* of recognizing numeric constant */

else if letter then
        do;                     /* have a letter */
           do while alphanum;
              call putandchar;
           end;
           if not lookup then
              do;
                 token = identifier;
                 call setup$next$call;
                 return;
              end;
           else               /* is a rw but if comment skip */
              do;
                 call set$up$next$call;
                 return;
              end;
        end;           /* of recognizing rw or ident */


     else                              /* special character */
        do;
           if nextchar = andsign then
              do;
                 nextchar = getchar;
                 do while nextchar <> andsign;
                    nextchar = getchar;
                 end;
                 call get$no$blank;
              end;
           else
              do;
                 if nextchar = ':' then
                    do;
                       call putandchar;
                       if nextchar = '=' then
                          call putandget;
                    end;
                 else
                    if nextchar = '.' then
                       do;
                          call putandchar;
```

```
                                    if nextchar = '.' then
                                       call putandget;

                                    else
                                      if numeric then
                                       do;
                                          token = numberc ;
                                          typenum = real$type;
                                          call spoolnumeric ;
   /* check for exponent */
     if nextchar = 'e' then
        do;
           typenum=unsign$expon;
           call putandchar;
           if nextchar = '-' or nextchar = '+' then
             do;
                typenum=signed$expon;
                call putandchar;
             end;
           call spoolnumeric;
         end;
                                          call set$up$next$call ;
                                          return;
                                       end;
                                end;
                             else
                               if nextchar = '(' then
                                 do;
                                    call putandchar ;
                                    if nextchar = '*' then
                                      call putandget;
                                 end;
                              else
                                 if nextchar = '*' then
                                    do;
                                      call putandchar ;
                                      if nextchar = ')' then
                                        call putandget ;
                                    end;
                                 else
                                  call putandget;

                           if not lookup then
                              call error('ic');
                           call setup$next$call;
                           return;
                        end;
                  end;                      /* of recognizing special char */
     end;                                  /* of do forever */
 end scanner;                              /* end of scanner */


/*************************************************************************/
/*************************************************************************/
/***                    procedures for synthesizer                     ***/
/*************************************************************************/
/*************************************************************************/

initialize$symtbl: proc;

dcl     symbase             addr;

        do;
          call fill(.hashtable,0,shl(hashtblsize,2));
          symbase=.init$symb$tbl;
          sbtbl=.memory;
          init$symb$tbl(15)=high(symbase);
          init$symb$tbl(16)=low(symbase);
          init$symb$tbl(25)=high(symbase+13);
          init$symb$tbl(26)=low(symbase+13);
          init$symb$tbl(35)=high(symbase+23);
          init$symb$tbl(36)=low(symbase+23);
```

```
            initSsymbStbl(48)=high(symbase+33);
            initSsymbStbl(49)=low(symbase+33);
            initSsymbStbl(59)=high(symbase+46);
            initSsymbStbl(60)=low(symbase+46);
            initSsymbStbl(71)=high(symbase+57);
            initSsymbStbl(72)=low(symbase+57);
            initSsymbStbl(85)=high(symbase+69);
            initSsymbStbl(86)=low(symbase+69);
            hashtable(14)=symbase;
            hashtable(36)=symbase+13;
            hashtable(30)=symbase+23;
            hashtable(0)=symbase+33;
            hashtable(16)=symbase+46;
            hashtable(113)=symbase+57;
            hashtable(64)=symbase+69;
            hashtable(107)=symbase+83;
            prvSsbtblSentry = symbase+83;

        end;
        sbtbltop=max-2;
        vecptr=0; constSptr=0;
        constSindx=0;
        constSpnSptr=0;
        subrSptr=0;
        arySdmSadrSptr=-1;
        arrySptr=-1;
        variantSpart=false;
        arrySqty=0;
        allocSaddr=0;

end initializeSsymtbl;


/*****************************************************************************/
/*****************************************************************************/
/***                          p a r s e r                                  ***/
/*****************************************************************************/
/*****************************************************************************/


do;                                        /* block for parser */
dcl     state statesize,
        var(pstacksize) byte,
        hash(pstacksize) byte,
        varc(varcsize) byte,
        varindex byte,
        statestack(pstacksize) statesize,
        (sp,mp,mpp1,nolook) byte;

    /* nmumonics for pascal-sm machine */

dcl     endp  lit  '1',lbl   lit  '2',nop   lit  '0',ldib  lit  '3',
        ldii  lit  '4',cnvb  lit  '9',cnvi  lit '10',all   lit '11',
        lita  lit '12',addb  lit '13',addi  lit '14',subi  lit '16',
        muli  lit '18',divi  lit '20',lssi  lit '22',leqi  lit '24',
        eqlb  lit '25',eqli  lit '26',neqb  lit '28',neqi  lit '29',
        geqb  lit '31',geqi  lit '32',grti  lit '34',negb  lit '35',
        negi  lit '36',comb  lit '37',comi  lit '38',notx  lit '39',
        andx  lit '40',bor   lit '41',stob  lit '42',stoi  lit '43',
        sto   lit '44',stdb  lit '45',stdi  lit '46',std   lit '47',
        dcrb  lit '48',dcri  lit '49',dcr   lit '50',brl   lit '52',
        blc   lit '53',lod   lit '55',lodb  lit '56',lodi  lit '57',
        rdvb  lit '58',rdvi  lit '59',rdvs  lit '60',wrvb  lit '61',
        wrvi  lit '62',wrvs  lit '63',cnal  lit '51',cn2i  lit '54',

        dump  lit '64',grtb  lit '33',lssb  lit '21',leqb  lit '23',
        mulb  lit '17',divb  lit '19';

initializeSsynthesize: proc;
   codesize = 0;
end initializeSsynthesize;
```

```
/*****************************************************************************/
/*****************************************************************************/
/***                      code generating procedures                       ***/
/*****************************************************************************/
/*****************************************************************************/

synthesize: proc;                       /* synthesize local declarations */


setaddrptr: proc(offset);
    dcl offset byte;
    aptraddr = base + offset;
end setaddrptr;

calc$varc: proc(a) addr;
    dcl a byte;
    return var(a) + .varc;
end calc$varc;

setlookup: proc(a);
    dcl a byte;
    printname = calc$varc(a);
    symhash = hash(a); /* hashcode of pn */
end setlookup;


  /*************************************************/
  /* enter$links - this procedure enters in the    */
  /*  next four bytes of the symbol table the      */
  /*  collision field and the previous symbol      */
  /*  table entry address field for the next       */
  /*  symbol table entry. ( both in address var ) */
  /*************************************************/

enter$links: proc;
    base,aptraddr = sbtbl;
    addrptr = hashtable(symhash);
    call setaddrptr(2);
    addrptr = prv$sbtbl$entry;
    prv$sbtbl$entry = sbtbl;
    hashtable(symhash) = base;
end enter$links;

check$print$name: proc(a) byte;
/* a is offset from base to printname */
dcl n based printname byte;
dcl (len,a) byte;
call setaddrptr(a);
    if ( len := byteptr ) = n then
    do while (byteptr(len)=n(len));
    if ( len := len-1 ) = 0 then
    return true;
    end;
    return false;

end check$print$name;


  /*************************************************/
  /* lookup$printname$identity - this procedure    */
  /*  is passed the location of an identifier in   */
  /*  the production rule, and its target entry    */
  /*  type. if the identifier is found with the    */
  /*  correct type the procedure return true,      */
  /*  else false is returned.                      */
  /*************************************************/

lookup$pn$id: proc(a,id$entry) byte;
    dcl (a,id$entry) byte;
    call setlookup(a);
    base = hashtable(symhash);
```

```
      do while base <> 0;
        call setaddrptr(4);
        if (( byteptr and formmask ) = idSentry ) then
        do;
          if checkSprintSname(5) then
          lookupSaddr=base;
          return true;
        end;
        else do;
         call setaddrptr(0);
         base = addrptr;
        end;
       end;
    return false;
end lookupSpnSid;


  /*************************************************/
  /*  limits - this procedure ensures that the     */
  /*   symbol table entry about to be entered      */
  /*   will not exceed the upper limit of the      */
  /*   available symbol table addresses.           */
  /*   the parameter is the bytecount of the       */
  /*   entry to be entered.                        */
  /*************************************************/

limits: proc(count);
    dcl count byte;
    if sbtbltop <= (sbtbl + count ) then
      do;
        call error('to');
        call mon3;
      end;
end limits;


  /*************************************************/
  /*  enterSprintnameSidentity - this procedure    */
  /*   loads the symbol table with the following:  */
  /*    1. collision field                         */
  /*    2. previous symbol table entry address     */
  /*    3. form of entry ( preset byte "form" )    */
  /*    4. the length of the printname in one byte */
  /*    5. the printname characters                */
  /*   parameter: printname is set prior to call.  */
  /*************************************************/

enterSpnSid:proc;
    dcl (i,n based printname) byte;
    call limits(i:=n+6);
    call enterSlinks;
    call setaddrptr(4);
    byteptr = form;
    call setaddrptr(5);
    byteptr=n;
    call move(printname+1,sbtbl+6,n);
    sbtbl=sbtbl+i;
end enterSpnSid;


  /*************************************************/
  /*  enterSvariableSidentity - this procedure     */
  /*   calls enterSpnSid to load the symbol table  */
  /*   entry currently being scanned. it also      */
  /*   generates the entry's "form" by performing  */
  /*   a boolean 'or' operation on the idSentry    */
  /*   and the parameter "a".                      */
  /*************************************************/

enterSvarSid: proc(a,b,idSentry);
    dcl (a,b,idSentry) byte;
    if lookupSpnSid(b,idSentry) then
```

```
      return;
    /* else enter var name */
    form = a or idSentry;
    call enterSpnSid;
end enterSvarSid;


  /*************************************************/
  /*  setSlabel - this procedure assigns a label  */
  /*   to the current declared label and increment*/
  /*   the labelcount ( next to assign ).         */
  /*************************************************/

setSlabel: proc;
    addrptr=lablcount;
    lablcount=lablcount+1;
end setSlabel;


  /*************************************************/
  /*  enterSlabel - this procedure loads a label  */
  /*   entry into the symbol table. symhash and   */
  /*   printname must be set prior to calling     */
  /*************************************************/

enterSlabel: proc;
    call limits(2);
    aptraddr = sbtbl;
    call setSlabel;
    sbtbl = sbtbl+2;
end enterSlabel;



  /*************************************************/
  /*  lookupSonly - this procedure is passed the  */
  /*   position of a identifier just scanned in   */
  /*   the current production ( sp,mp,mpp1 ) and  */
  /*   returns true if the identifier is found in */
  /*   the symbol table.                          */
  /*************************************************/

lookupSonly: proc(a) byte;
   dcl a byte;
   call setlookup(a);
   base=hashtable(symhash);
     do while base <> 0;
      if checkSprintSname(5) then
       do;
         lookupSaddr=base;
         return true;
       end;
      else do;
       call setaddrptr(0);
       base=addrptr;
       end;
     end;
   return false;
end lookupSonly;

convrtbcd: proc(a,b); /* a=sp/mp/mpp1, b=pos/neg */

 /*************************************/
 /*  this procedure converts a real   */
 /*   number in the program to a bcd  */
 /*   representation.                 */
 /*************************************/

   dcl (i,j,dflag,eflag,sflag,a,b,n based printname) byte;
   dcl (exponloop,expsignloop) label;

     call setlookup(a);
```

```
 /*  initialize variables */
sflag=false; eflag=true; dflag=true; i=1;
do j=0 to 7; bcdnum(j)=0; end;
j=0; expon=64; /* e+00 */

 /* remove leading zeros */
do while ((n(i) - '0') = 0);
   i=i+1;
   if i=(n+1) then goto exponloop;
end;

 /* load bcdnum with significant digits */
do while ((n(i) - '0' ) <= 9 or n(i) = '.');
  if n(i) = '.' then
    do; eflag=false;
         if i=n then goto exponloop;
         i = i + 1;
    end;
  else
    do;
     do while j = 0 and dflag and (n(i) - '0') = 0;
       expon = expon-1;
       if i = n then goto exponloop;
       i = i + 1 ;
     end;
     if j = ( bcdsize-1 ) then goto exponloop;
     if dflag then /* first bcd pair */
       do;
         bcdnum(j)=rol((n(i)-'0'),4);
         dflag=false; i= i+1;
         if eflag then expon=expon+1;
       end;
     else
       do;
         bcdnum(j)=bcdnum(j)+(n(i)-'0');
         j = j + 1; i = i + 1;
         dflag=true; if eflag then expon=expon+1;
       end;
     if i=(n+1) then goto exponloop;
    end;
end;
exponloop:
  if i = (n+1) then goto expsignloop;
      if eflag then
       do;
        do while n(i) <> '.';
          expon = expon + 1;
          i = i + 1;
        end;
        i = i + 1;
       end;
      do while i < (n+1) and (n(i)-'0') <= 9 ;
        i = i + 1;
      end;
      if typenum = realtype then goto expsignloop;
      /* n(i) = e */ i = i+1;
      if typenum = signedexpon then
        do;
          if n(i) = minusx then sflag = true;
          i = i + 1 ;
        end;
      if  i = n+1 then
        do;
          call error('ee');
          return;
        end;
      dflag = 0;
      do j = i to n;
        dflag = (dflag*10)+(n(j)-'0');
      end;
      if sflag then /* exponent calculation */
         expon = expon-dflag;
```

```
      else expon = expon + dflag;

      expsignloop:
      bcdnum(bcdsize-1)=rol(b,7); /* sign of number */
      if expon > 127 then
       do;
         call error('ee');
         return;
       end;
      else bcdnum(bcdsize-1)=bcdnum(bcdsize-1)+expon;

end convrtbcd;

  /*************************************************/
  /*  converti - this procedure is passed "a", the*/
  /*   location of a constant in the production   */
  /*   and "b" the 'sign' of the integer. the     */
  /*   function generates a signed 16 bit repre-  */
  /*   sentation of the number and returns it in  */
  /*   an address variable.                       */
  /*************************************************/

converti: proc(a,b) address;
   dcl (i,a,b,n based printname) byte;
   dcl num addr;

     call setlookup(a); num=0;
     do i=1 to n;
       if (maxint/10) >= num then
         do;
           if (maxint/10) = num and (n(i)-'0') > 7 then
             do;
               call error('ie');
               return num;
             end;
           num=(num*10)+(n(i)-'0');
         end;
       else do;
             call error('ie');
          return num;
         end;
      end;
      if b = pos then return num;
      if num = maxint then
       do;
         call error('ie');
        return num;
       end;
      return ( - num);
end converti;

  /*************************************************/
  /*  convert$constant - this procedure is called */
  /*   with typenum set by the caller. the number */
  /*   must be pointed to by "sp" in the produc-  */
  /*   tion. the procedure returns with "const$   */
  /*   num$type" and "const$value" set with the   */
  /*   number in its internal form.               */
  /*************************************************/


convrt$const: proc(a); /* a=pos,neg */
   dcl a byte,int$addr addr;
     if typenum = integer$type then
      do;
        int$addr=converti(sp,a);
        const$num$type(const$ptr)=integer$type;
        const$ptr=const$ptr+1;
        call move(.int$addr,.const$value(const$indx),2);
        const$indx=const$indx+2;
      end;
     else do;
```

```
        call convrtbcd(sp,a);
        const$num$type(const$ptr)=real$type;
        const$ptr=const$ptr+1;
        call move(.bcdnum,.const$value(const$indx),bcdsize);
        const$indx=const$indx+bcdsize;
       end;
end convrt$const;

  /*****************************************************/
  /*  enter$constant$number - after the next entry*/
  /*   has had its links entered into the symbol  */
  /*   table, this procedure enters the constant  */
  /*   value into the symbol table and set the    */
  /*   entry's "form" to the appropriate type.    */
  /*****************************************************/


enter$cons$numb: proc;

   const$ptr=const$ptr-1;
   if const$num$type(const$ptr)= integertype then
     do;
       call setaddrptr(4); byteptr=8 or cons$entry;
       call limits(2); const$indx=const$indx-2;
       call move(.const$value(const$indx),sbtbl,2);
       sbtbl=sbtbl+2;
     end;
   else do;
       call setaddrptr(4); byteptr=10h or cons$entry;
       call limits(bcdsize); const$indx=const$indx-bcdsize;
       call move(.const$value(const$indx),sbtbl,bcdsize);
       sbtbl=sbtbl+bcdsize;
     end;
end enter$cons$numb;

  /*****************************************************/
  /*  enter$string - after the "links" and "form" */
  /*   are entered into the symbol table, this    */
  /*   procedure loads any identifier along with  */
  /*   its length. (used with constant strings    */
  /*   and constant identifiers )                 */
  /*****************************************************/


enter$string: proc(a);
   dcl (a,n based printname) byte;
     call setlookup(a);
     call limits(n+1);
     call move(printname,sbtbl,(n+1));
     sbtbl=sbtbl+(n+1);
end enter$string;

enter$cons$ident: proc(a,b); /* a=pos/neg , b=mp/mpp1/sp */
   dcl (a,b,c) byte;
      c=rol(a,6);
      call setaddrptr(4); byteptr=c or cons$entry;
      call enter$string(sp);
      const$pn$ptr=const$pn$ptr-1;
      const$indx=const$indx-const$pn$size(const$pn$ptr);
end enter$cons$ident;

enter$const$entry: proc;
   dcl ixindex byte;
     vecptr=vecptr-1;
      do case const$vec(vecptr);
        /* case constant number */
        call enter$cons$numb;
        /* case identifier constant */
        call enter$cons$ident(pos,sp);
        /* case signed identifier constant */
        call enter$cons$ident(neg,sp);
        /* case constant string */
```

```
        do;
        call setaddrptr(4); byteptr=18h or consSentry;
        call enterSstring(sp);
        constSpnSptr=constSpnSptr-1;
        constSindx=constSindx-constSpnSsize(constSpnSptr);
        end;
      end; /* of case constStype */
end enterSconstSentry;

  /*************************************************/
  /*  enterScomplexStype - this procedure is       */
  /*   called to enter the "links" and "form" for  */
  /*   the 'complex type' symbol table entries.    */
  /*   note* that this entry never has a print-    */
  /*   name assigned.                              */
  /*************************************************/


enterScomplexStype: proc(a);
   dcl a byte;
     call limits(5);
     base,aptraddr=sbtbl;
     addrptr=0000h;
     call setaddrptr(2);
     addrptr=prvSsbtblSentry;
     prvSsbtblSentry=base;
     call setaddrptr(4);
     byteptr=a;
     sbtbl=sbtbl+5;
end enterScomplexStype;

  /*************************************************/
  /*  enterSstructStype - this procedure is        */
  /*   called by the 'type' productions:           */
  /*     1. set type                               */
  /*     2. file type                              */
  /*     3. pointer type                           */
  /*   it calls enterScomplexStype to set up its   */
  /*   "links" and "form", then it sets a pointer  */
  /*   to the associated complex type.             */
  /*************************************************/


enterSstructStype: proc(a);
   dcl a byte;
     call enterScomplexStype(a);
     call limits(2);
     call setaddrptr(5);
     addrptr=typeSloct;
     sbtbl=sbtbl+2;
     typeSloct=base;
end enterSstructStype;

  /*************************************************/
  /*  lookupSidentifier - this procedure is called*/
  /*   with 'symhash' and printname set. it will   */
  /*   return true if the identifier can be found  */
  /*************************************************/


lookupSident: proc byte;
   base=hashtable(symhash);
     do while base <> 0;
       if checkSprintSname(5) then
         do;
            lookupSaddr=base;
            return true;
         end;
       else do;
         call setaddrptr(0);
         base=addrptr;
         end;
```

```
      end;
    return false;
end lookup$ident;

  /*************************************************/
  /*  lookup$printname$only - this procedure sets */
  /*   the "symhash" and calls lookup$ident to    */
  /*   determine if the entry is in the symbol    */
  /*   table.  the address of the printname is    */
  /*   passed as a parameter. if the entry is     */
  /*   found, true is returned.                   */
  /*************************************************/


lookup$pn$only: proc(a) byte;
   dcl a addr; /* addr of print-name */
   dcl (b,n based a) byte;
      hashcode=0;
      do b=1 to n;
        hashcode=(hashcode+n(b)) and hashmask;
      end;
      symhash=hashcode;
      printname=a;
      if lookup$ident then
         return true;
      else return false;
end lookup$pn$only;

 /*************************************************/
 /*  store$constant identifier - this routine is  */
 /*   called with printname set to load an        */
 /*   identifier in the 'constant value' variable.*/
 /*************************************************/


store$const$ident: proc;
   dcl n based printname byte;
     call setlookup(sp);
     call move(printname,.const$value(const$indx),(n+1));
     const$indx=const$indx+(n+1);
     const$pn$hash(const$pn$ptr)=symhash;
     const$pn$size(const$pn$ptr)=n+1;
     const$pn$ptr=const$pn$ptr+1;
end store$const$ident;

subr$error: proc;
  call error('is');
  subr$type(subr$ptr)=integer$type;
  subr$val(subr$ptr)=0000h;
end subr$error;

ord$hi$low$check: proc;
  if subr$ptr=0 then return;
  if subr$type=subr$type(1) then
    do;
       if subr$val > subr$val(1) then return;
    end;
  call error('is');
end ord$hi$low$check;

subr$int$hi$low$check: proc;
  if subr$ptr=0 then return;
  if subr$type <> subr$type(1) then
    do;
       call subr$error;
       return;
    end;
  if subr$val < 32768 and subr$val(1) >32767 then
    do;
      integer$diff = subr$val+( -subr$val(1))+1;
      return;
    end;
```

```
  if subr$val > 32767 and subr$val(1) < 32768 then
    do;
      call subr$error;
      return;
    end;
  if subr$val <32768 then /* both positive */
    do;
      if(subr$val-(subr$val(1)+1)) < 32768 then
      do; integer$diff=subr$val-(subr$val(1))+1;
        return;
      end;
     call subr$error;
     return;
    end;
    else /* both negative */
      if ( - subr$val(1)-( - subr$val +1)) < 32768 then
        do;
          integer$diff=( - subr$val(1))-( - subr$val)+1;
          return;
        end;
    call subr$error;
end subr$int$hi$low$check;

 /*****************************************************/
 /*  subrange$identifer$procedure - this routine  */
 /*   is called to determine the offset ( number  */
 /*   of entries in a subrange ) and the type of  */
 /*   subrange, given that the subrange type is   */
 /*   a named identifier.                         */
 /*****************************************************/


subr$ident$proc: proc;
    const$pn$ptr=const$pn$ptr-1;
    const$indx=const$indx-const$pn$size(const$pn$ptr);
    printname=.const$value(const$indx);
    symhash=const$pn$hash(const$pn$ptr);
    if not lookup$ident then call subr$error;
    else /* found constant identifier */
      do;
        base=lookup$addr;
        call setaddrptr(4); /* points to form(byteptr) */
        subr$form=byteptr;
        if subr$form <> 07h and (subr$form and formmask) <> cons$entry
           then call subr$error;
        else do;
         if subr$form = 07h then
           do;
             subr$type(subr$ptr)=ord$type;
             call setaddrptr(5);
             subr$form=byteptr; /* length of p.name */
             call setaddrptr(6+subr$form);
             subr$val(subr$ptr)=double(byteptr);
             call setaddrptr(7+subr$form);
             subr$type$addr(subr$ptr)=addrptr;
             call ord$hi$low$check;
           end;
         else
           do;
             do while ((shr(subr$form,3) and 3h)=0);
               if shr(subr$form,5)=neg then
                 if subr$pn$sign=pos then subr$pn$sign=neg;
                 else subr$pn$sign=pos;
               call setaddrptr(5);
               subr$form=byteptr;
               call setaddrptr(6+subr$form);
               if not lookup$only(aptraddr) then
                 do;
                   call subr$error;
                   subr$ptr=subr$ptr+1;
                   return;
                 end;
```

```
                else do;
                  base=lookup$addr;
                  call setaddrptr(4);
                  subr$form=byteptr;
                  end;
               end;
               if (shr(subr$form,3)and 3h) = 2 then
                 do;
                   call subr$error;
                   subr$ptr=subr$ptr+1;
                   return;
                 end;
               /* here we have either an integer or char */
               if (shr(subr$form,3) and 3h) =  1 then
                 do; /* integer */
                     call setaddrptr(5);
                   subr$form=byteptr;
                   call setaddrptr(6+subr$form);
                   if subr$pn$sign = neg then
                     subr$val(subr$ptr)=  - addrptr;
                   else subr$val(subr$ptr)=addrptr;
                   subr$type(subr$ptr)=integer$type;
                   call subr$int$hi$low$check;
                 end;
              else
                do;
                  call setaddrptr(5);
                  subr$form=byteptr;
                  call setaddrptr(6+subr$form);
                  if byteptr <> 1 then
                    do;
                      call subr$error;
                      subr$ptr=subr$ptr+1;
                      return;
                    end;
                  call setaddrptr(7+subr$form);
                  if byteptr <41h or byteptr > 5ah then
                     call subr$error;
                  else do;
                     subr$val(subr$ptr)=double(byteptr-41h);
                     subr$type(subr$ptr)=char$type;
                     call ord$hi$low$check;
                     end;
                end;
             end;
           end;
         end;
       subr$ptr=subr$ptr+1;
end subr$ident$proc;

 /*****************************************************/
 /*  subrange$case - this procedure is used to     */
 /*  determine the number of entries in a subrange*/
 /*****************************************************/


subr$case: proc;
   subr$pn$sign=pos;
   do case const$vec(vecptr);
    /* case const number */
    do; const$ptr=const$ptr-1;
      if const$num$type(const$ptr)=real$type then
        do;
           call subr$error;
           const$indx=const$indx-bcdsize;
        end;
      else
        do; /* integer type */
          const$indx=const$indx-2;
          call move(.const$value(const$indx),.subr$val(subr$ptr),2);
          subr$type(subr$ptr)=integer$type;
          call subr$int$hi$low$check;
```

```
        end;
      subr$ptr=subr$ptr+1; /* next to fill */
    end;
    /* case ident constant */
    call subr$ident$proc;
    /* case signed ident constant */
    do;
      subr$pn$sign=neg;
      call subr$ident$proc;
    end;
    /* case constant string */
    do;
       const$pn$ptr=const$pn$ptr-1;
      const$indx=const$indx-const$pn$size(const$pn$ptr);
      printname=.const$value(const$indx);
      if const$pn$size(const$pn$ptr) <> 2 then
        call subr$error;
      else
        do;
          base=printname;
          call setaddrptr(1);
          if byteptr < 41h or byteptr > 5ah then
            call subr$error;
          else
            do;
              subr$val(subr$ptr)=double(byteptr-41h);
              subr$type(subr$ptr)=char$type;
              call ord$hi$low$check;
            end;
        end;
      subr$ptr=subr$ptr+1;
    end;
  end; /* of case const$vec(vecptr) */
end subr$case;

/*************************************************/
/*  enter$subrange$entry - this procedure is     */
/*   used to enter a subrange type entry into    */
/*   the symbol table. this symbol table entry   */
/*   has no printname associated with it.        */
/*************************************************/


enter$subr$entry: proc;
  type$loct=sbtbl;
  call limits(14);
  vecptr=vecptr-1;
  call subr$case;
  vecptr=vecptr-1;
  call subr$case;
  call enter$complex$type(shl(subr$type,6)or 0fh);
  call setaddrptr(5);
  if subr$type=integer$type then
    addrptr=.init$symb$tbl;
  if subr$type=char$type then addrptr=(.init$symb$tbl+23);
  if subr$type=ord$type then addrptr=subr$type$addr;
  call setaddrptr(7);
  addrptr=subr$val(1);
  call setaddrptr(9);
  addrptr=subr$val;
  call setaddrptr(11);
  if subr$type=integer$type then /* range 0 to 64k */
     addrptr=integer$diff; /* may be greater than 32767 */
  else
     addrptr=((subr$val-subr$val(1))+1);
  subr$ptr=0;
  sbtbl=sbtbl+8;
end enter$subr$entry;

type$error: proc;
  allocate=false;
  call error('it');
```

```
end typeSerror;

 /*****************************************************/
 /*  allocate offset - this procedure is called to*/
 /*   determine the number of bytes required for  */
 /*   storage of a variable of the type given in  */
 /*   the parameter 'a'. the variable's allcSqty  */
 /*   and allcSform are set upon return.          */
 /*****************************************************/

allcSoffset: proc(a); /* typeSloct */
   dcl a addr;
   dcl (allcSform,b) byte;
     base=a;
     call setaddrptr(4); /* points to form of type */
     allcSform= byteptr and formmask;
     if allcSform <> typeSentry and allcSform <> typedcle then
      do;
       call typeSerror;
       allcSqty=1;
       allcSbasicStype=0;
       return;
      end;
      do while((shr(byteptr,3)and formmask)=7 and allcSform=typeSentry);
       call setaddrptr(5);
       call setaddrptr(6+byteptr);
       base=addrptr; call setaddrptr(4);
       allcSform=byteptr and formmask;
       if allcSform <> typeSentry and allcSform <> typedcle then
        do; call typeSerror;
          allcSqty=1;
          allcSbasicStype=0; return;
        end;
      end;
 /* here exists either a basic type or a type declaration */
      if allcSform = typeSentry then
       do; /* basic type */
        do case (shr(byteptr,3) and formmask);
          /* integer */
         do;
           allcSqty=2;
           allcSbasicStype=integerStype;
         end;
          /* bcd real */
         do;
           allcSqty=8;
           allcSbasicStype=unsignSexpon;
         end;
          /* character */
         do;
           allcSqty=1;
           allcSbasicStype=charStype;
         end;
          /* boolean */
         do;
           allcSqty=1;
           allcSbasicStype=booleanStype;
         end;
       end; /* of case */
       allocate=true;
       return;
      end;
/* here exists a type declaration */
   allcSform=(shr(byteptr,3)and formmask);
   if allcSform=0 then
    do; /* scalar */
     allocate=true;
     allcSqty=double(allcSform+1);
     allcSbasicStype=ordStype; return;
    end;
   if allcSform=1 then
    do; /* subrange */
```

```
      allocate=true;
      allc$basic$type=complex$type;
      b=shr(byteptr,6);
      if b = 1 then allc$qty=double(allc$form+1);
      else allc$qty=double(allc$form); return;
     end;
    if allc$form=2 then
     do; /* array */
      allocate=true;
      allc$basic$type=complex$type;
      call setaddrptr(8);
      allc$qty=addrptr; return;
     end;
    b=2;
    /* all other cases allocate an address field */
    allc$qty=double(b);
    allc$basic$type=complex$type;
    allocate=true;
end allc$offset;

 /*************************************************/
 /*  allc$index$offset - this procedure is called */
 /*    to determine the number of bytes required  */
 /*    by an array to store the array's components */
 /*    type$loct is set prior to calling this     */
 /*    routine. an address variable containing the */
 /*    byte count is returned.                    */
 /*************************************************/


allc$index$offset: proc addr;
    dcl a addr,b byte;
     a,base=type$loct;
     call setaddrptr(4);
     do while (shr(byteptr,3) and formmask) = 7 and
              ( byteptr and formmask ) = type$entry;
      call setaddrptr(5);
      call setaddrptr(6+byteptr);
      base=addrptr; call setaddrptr(4);
     end;
 /* here we have either a scalar,subrange,boolean, or char type */
     b= shr(byteptr,3) and formmask;
     if (byteptr and formmask) = type$entry then
      do;
        if b = 0 or b = 1 then
          do;
            call error('ia');
            b=2;
            return double(b);
          end;
        if b=2 then /* character subrange */
          do;
            b = 26;
            rec$var$typ(rec$nst)=char$type;
            return double(b);
          end;
        /* boolean */
        rec$var$typ(rec$nst)=boolean$type;
        b = 2; return double(b);
      end;
 /* complex type */
    if (( byteptr and formmask) <> type$dcle or
       (( b <> 0 ) and ( b <> 1 ))) then
      do;
        call error('ia');
        b=2; return double(b);
      end;
    if b=0 then
      do; /* scalar type */
       rec$var$typ(rec$nst)=complex$type;
       call setaddrptr(5);
       call setaddrptr(6+byteptr);
```

```
      return double(byteptr + 1);
    end;
 /* subrange type */
    recSvarStyp(recSnst)=ordStype;
    call setaddrptr(11);
    return addrptr;
end allcSindexSoffset;

 /*************************************************/
 /*  setSvariableStype - this procedure is called */
 /*   to set the variable type, variable sign,and */
 /*   address of the basic type given. the address*/
 /*   variable 'lookupSaddr' is set prior to the  */
 /*   call.                                       */
 /*************************************************/


setSvarStype: proc;

varSptr=varSptr+1; base=lookupSaddr;
call setaddrptr(4);
if (byteptr and formmask) = consSentry then
   do;  /* constant variable */
     subrSpnSsign=pos;
     do while (shr(byteptr,3) and 03h) = 0;
      if (shr(byteptr,5) and 01h) = 1 then
       do;
         if subrSpnSsign=pos then subrSpnSsign=neg;
         else subrSpnSsign=pos;
       end;
      call setaddrptr(5);
      if not lookupSpnSonly(aptraddr) then
       do;
         call error('ic');
        /* put in default values to return with */
        return;
       end;
      call setaddrptr(4);
      if (byteptr and formmask) <> consSentry then
        do;
          call error('ic');
           /* put in default values to return with */
          return;
        end;
     end;
     /* here we have a non-identifier constant variable */
     if (shr(byteptr,3) and 3h) =1 then
      do; /* integer or boolean constant */
       if base < 1000h then
        do; /* boolean */
         call setaddrptr(5);
         call setaddrptr(8+byteptr);
         varSbase(varSptr)=aptraddr; varSsign(varSptr)=pos;
         varStype(varSptr)=4h;
        end;
       else do; /* integer constant */
         call setaddrptr(5);
         call setaddrptr(6+byteptr);
         varSbase(varSptr)=aptraddr; varStype(varSptr)=5h;
         varSsign(varSptr)=subrSpnSsign;
        end;
       return;
      end;
     if (shr(byteptr,3) and 3h) = 2 then
       do; /* real constant */
        call setaddrptr(5);
        call setaddrptr(6+byteptr);
        varSbase(varSptr)=aptraddr; varStype(varSptr)=6h;
        varSsign(varSptr)=subrSpnSsign;
        return;
       end;
     /* default constant of string */
```

```
      call setaddrptr(5); call setaddrptr(6+byteptr);
      var$base(var$ptr)=aptraddr; var$type(var$ptr)=7h;
      return;
     end; /* of constant variables */
    if (byteptr and formmask) = var$entry then
     do; /* declared variables */
      ptrptr=shr(byteptr,3) and formmask; /* type of var */
      call setaddrptr(5); call setaddrptr(6+byteptr);
      var$base(var$ptr)=addrptr; /* relative addr of var */
      /* sign is always ignored */
      do case ptrptr;
       /* case 0 ord variable */
       do;
         var$type(var$ptr)=10h;
         aptraddr=aptraddr+2;
         var$base1(var$ptr)=addrptr; /* addr of parent */
       end;
       /* case 1 integer variable */
       var$type(var$ptr)=09h;
       /* case 2 char variable */
       var$type(var$ptr)=0bh;
       /* case 3 real variable */
       var$type(var$ptr)=0ah;
       /* case 4 complex variable */
         do; /* not implimented */
           /* insert complex variable routines here */
         end;
       /* case 5 boolean variable */
       var$type(var$ptr)=08h;
      end; /* of variable case */
      return;
     end;
     if byteptr = 7h then
       do; /* scalar constant */
         call setaddrptr(5); call setaddrptr(6+byteptr);
         var$base(var$ptr)=aptraddr; aptraddr=aptraddr+1;
         var$base1(var$ptr)=addrptr; /* parent type of scalar */
         var$type(var$ptr)=11h;
         return;
       end;

end set$var$type;

 /*******************************************************/
 /*  load$variable - this procedure generates the */
 /*   intermediate code to load the next variable */
 /*   on the execution stack of the object file   */
 /*******************************************************/


load$variable: proc;

  if var$type(var$ptr) < 08h then
    do; /* constant variable */
     aptraddr=var$base(var$ptr);
     if var$type(var$ptr)=04h then
      do; /* boolean constant */
       call generate(ldii); call generate(byteptr);
       call generate(nop); /* high byte zero */
       exp$type(exp$ptr)=boolean$type;
      end;
     if var$type(var$ptr)=05h then
      do; /* integer constant */
       call generate(ldii); call generate(byteptr);
       call generate(high(addrptr));
       if var$sign(var$ptr)=neg then
        call generate(negi);
       exp$type(exp$ptr)=integer$type;
      end;
     if var$type(var$ptr)=06h then
      do; /* bcd constant */
       call generate(ldib);
```

```
        do ptrptr=1 to (bcdsize/2);
         call generate(byteptr); call generate(high(addrptr));
         aptraddr=aptraddr+2;
        end;
        if varSsign(varSptr)=neg then
         call generate(negb);
        expStype(expSptr)=unsignSexpon;
       end; /* of load bcd */
      if varStype(varSptr)=07h then
       do; /* string constant */
        call generate(nop); /* not implimented */
        expStype(expSptr)=stringStype;
       end;
      varSptr=varSptr-1;
      return;
     end; /* of constant variable load */
   if varStype(varSptr) < 11h then
    do; /* simple variables */
     call generate(lita); /* load addr of variable */
     call generate(low(varSbase(varSptr)));
     call generate(high(varSbase(varSptr)));
     if varStype(varSptr)= 08h then
      do; /* boolean variable */
        call generate(lod);
        expStype(expSptr)=booleanStype;
      end;
     if varStype(varSptr)=09h then
      do; /* integer variable */
       call generate(lodi);
       expStype(expSptr)=integerStype;
      end;
     if varStype(varSptr)= 0ah then
      do; /* real variable */
       call generate(lodb);
       expStype(expSptr)=unsignSexpon;
      end;
     if varStype(varSptr)= 0bh then
      do; /* char variable */
       call generate(lod);
       expStype(expSptr)=charStype;
      end;
     if varStype(varSptr)= 10h then
      do; /* ord variable */
       call generate(lod);
       expStype(expSptr)=ordStype;
       expStypeSaddr(expSptr)=varSbase1(varSptr);
      end;
     varSptr=varSptr-1;
     return;
    end;
    if varStype(varSptr)= 11h then
     do; /* ord constant */
      aptraddr=varSbase(varSptr);
      call generate(ldii);
      call generate(byteptr); call generate(nop);
      expStype(expSptr)= ordStype;
      expStypeSaddr(expSptr)= varSbase1(varSptr);
      varSptr=varSptr-1;
     end;
end loadSvariable;

 /*:::::::::::::::::::::::::::::::::::::*/
 /* this procedure checks the top two */
 /* variables on the execution stack  */
 /* for proper type.                  */
 /*:::::::::::::::::::::::::::::::::::::*/

checkSexprsStype: proc byte;

  if (expStype(expSptr)=expStype(expSptr-1)) and expStype(expSptr)<>0h
    then return true;
  if expStype(expSptr)=1h then
```

```
    do;
      if expStype(expSptr-1)=3h then
       do;
         call generate(cnvi); /* convert int to bcd */
         expStype(expSptr)=3h;
         return true;
       end;
      else return false;
    end;
  if expStype(expSptr)=3h then
    do;
     if expStype(expSptr-1)=1h then
       do;
         call generate(cn2i); /* convert second int to bcd */
         expStype(expSptr-1)=3h;
         return true;
       end;
     else return false;
   end;
  if expStype(expSptr)=0h then
   do;
     if expStype(expSptr-1)<>0h then
       return false;
     else
      do;
       if expStypeSaddr(expSptr)=expStypeSaddr(expSptr-1) then
          return true;
      end;
   end;
   return false;
end checkSexprsStype;


  /*****************************************/
  /*  writeSstring - this procedure writes */
  /*    a string to the intermed. code     */
  /*****************************************/

writeSstring: proc;

dcl n based printname byte;
  call setlookup(sp);
  call generate(wrvs);
  call generate(n);
  do ptrptr = 1 to n;
   call generate(n(ptrptr));
  end;
end writeSstring;


  /*****************************************/
  /*  writeSvariable - this procedure will */
  /*    write a variable to the console via */
  /*    the intermed. code.                */
  /*****************************************/

writeSvar: proc;
  if expSptr = 11 then
    do;
       call error('es');
       call mon3;
    end;
  expSptr=expSptr+1;
  call loadSvariable;
  do case expStype(expSptr);
    call generate(wrvi);
    call generate(wrvi);
    call generate(wrvi);
    call generate(wrvb);
    call generate(wrvi);
    call generate(wrvi);
  end; /* of expStype case */
```

```
  expSptr=expSptr-1;
end writeSvar;

  /*************************************************/
  /*  readSvariable - this procedure generates */
  /*    the intermediate code to read a variable*/
  /*    from the console.                        */
  /*************************************************/

readSvar: proc;
  if varStype(varSptr) < 08h then
   call error('ir');
  else do;
     if varStype(varSptr) < 11h then
      do;
        if varStype(varSptr)=08h then
         do; /* read boolean not implimented */
         end;
        if varStype(varSptr)=09h then
         do;
          call generate(rdvi);
          call generate(stdi);
          call generate(low(varSbase(varSptr)));
          call generate(high(varSbase(varSptr)));
         end;
        if varStype(varSptr)=0ah then
         do; /* real */
          call generate(rdvb);
          call generate(stdb);
          call generate(low(varSbase(varSptr)));
          call generate(high(varSbase(varSptr)));
         end;
        if varStype(varSptr)=0bh then
         do; /* read char not implimented */
         end;
        if varStype(varSptr)=10h then
         do; /* read ord not implimented */
         end;
      end;
      if varStype(varSptr)=11h then
        call error('ir');
     varSptr=varSptr-1;
    end;
end readSvar;



    if listprod then
       call printSprod;


       do case production;


/*************************************************************************/
/*************************************************************************/
/***                    p r o d u c t i o n s                          ***/
/*************************************************************************/
/*************************************************************************/
/***                the following is the input grammar                 ***/
/*************************************************************************/
/*************************************************************************/


/*      case 0 not used    */  ;

 /*      1   <program> ::= <program heading> <block> . _|_            */
    do;
       call printSerror;
       call printchar(' ');
```

```
      call crlf;
      call print(.' compilation complete.$');
      call crlf;
      if not (errorcount > 0) then
        do;
          call generate(all);
          call generate(low(alloc$addr));
          call generate(high(alloc$addr));
          call generate(endp);
        end;
      call write$int$file;
      call close$int$file;
      call mon3;
    end;

/*      2   <program heading> ::= program <prog ident> (                  */
/*      2                           <file ident> ) ;                      */
            ;
/*      3                         | program <prog ident> (                */
/*      3                           <file ident> , <file ident> ) ;       */
            ;

/*      4   <prog ident> ::= <identifier>                                 */
            ;

/*      5   <file ident> ::= <identifier>                                 */
    if first$time then
      do;
        first$time = false;
        call enter$var$id(0,sp,file$entry);
      end;
    else
      call enter$var$id(16,sp,file$entry);

/*      6   <block> ::= <ldp> <cdp> <tdp> <vdp> <p$fdp> <stmtp>           */
            ;

/*      7   <ldp> ::=                                                     */
            ;
/*      8           | label <label string> ;                            */
            ;

/*      9   <label string> ::= <label>                                    */
    if typenum = integer$type then
      do;
        call enter$var$id(0,sp,lab1$entry);
        call enter$label;
      end;
/*     10                    | <label string> , <label>                 */
    if typenum = integer$type then
      do;
        call enter$var$id(0,sp,lab1$entry);
        call enter$label;
      end;

/*     11   <label> ::= <number>                                          */
    if typenum <> integer$type then
      call error('ls');

/*     12   <cdp> ::=                                                    */
    ;
/*     13          | const <const def> ;                                 */
    ;

/*     14   <const def> ::= <ident const def>                            */
    ;
/*     15                 | <const def> ; <ident const def>             */
    ;

/*     16   <ident const def> ::= <ident const> = <constant>             */
    call enter$const$entry;
```

```
/*     17   <ident const> ::= <identifier>                                  */
    do;
      if lookup$only(sp) then
        call error('dc');
      call enter$var$id(0,sp,cons$entry);
     end;

/*     18   <constant> ::= <number>                                         */
    do;
     call convrt$const(pos);
     const$vec(vecptr)=cons$num$type;
     vecptr=vecptr+1;
    end;
/*     19                 | <sign> <number>                                */
    do;
     if signtype=neg then
       call convrt$const(neg);
     else call convrt$const(pos);
     const$vec(vecptr)=cons$num$type;
     vecptr=vecptr+1;
    end;
/*     20                 | <constant ident>                               */
    do;
     const$vec(vecptr)=cons$ident$type;
     vecptr=vecptr+1;
    call store$const$ident;
    end;
/*     21                 | <sign> <constant ident>                        */
    do;
     if signtype=neg then
      const$vec(vecptr)=cons$ident$type;
     else const$vec(vecptr)=cons$ident$type;
     vecptr=vecptr+1;
    call store$const$ident;
    end;
/*     22                 | <string>                                       */
    do;
      const$vec(vecptr)=cons$str$type;
      vecptr=vecptr+1;
     call store$const$ident;
    end;

/*     23   <constant ident> ::= <identifier>                               */
    ;

/*     24   <sign> ::= +                                                    */
    signtype=pos;
/*     25            | -                                                    */
    signtype=neg;

/*     26   <tdp> ::=                                                       */
    case$stmt=false;
/*     27           | type <type def string> ;                             */
    case$stmt=false;

/*     28   <type def string> ::= <type id>                                 */
    ;
/*     29                       | <type def string> ; <type id>            */
    ;

/*     30   <type id> ::= <type ids> = <type>                               */
    do;
      aptraddr=type$addr;
      addrptr=type$loct;
    end;

/*     31   <type ids> ::= <identifier>                                     */
    do;
      if lookup$only(sp) then
        call error('dt');
      parent$type=sbtbl;
      call enter$var$id(78h,sp,type$entry);
```

```
      call limits(2);
      type$addr=sbtbl;
      sbtbl=sbtbl+2;
    end;

/*     32   <type> ::= <simple type>                                    */
    ;
/*     33            | <structured type>                               */
    ;
/*     34            | <pointer type>                                  */
    ;

/*     35   <simple type> ::= <scalar type>                             */
    ;
/*     36                   | <subrange type>                          */
    ;
/*     37                   | <type ident>                             */
    ;

/*     38   <type ident> ::= <identifier>                               */
  if lookup$pn$id(sp,type$entry) then
    type$loct=lookup$addr;
  else
    do;
      call error('ti');
       type$loct=.init$symb$tbl;  /* integer default */
    end;

/*     39   <scalar type> ::= ( <tident string> )                       */
    ;

/*     40   <tident string> ::= <identifier>                            */
    do;
      type$ord$num=0;
      type$loct=sbtbl;
     if  lookup$only(sp) then
      call error('dt');
      call enter$var$id(0,sp,type$dcle);
      call limits(3);
      aptraddr=sbtbl;
      byteptr=0;
      aptraddr=aptraddr+1;
      addrptr=parent$type;
      sbtbl=sbtbl+3;
    end;
/*     41                     | <tident string> , <identifier>          */
    do;
      type$ord$num=type$ord$num+1;
      type$loct=sbtbl;
      if  lookup$only(sp) then
        call error('dt');
      call enter$var$id(0,sp,type$dcle);
      call limits(3);
      aptraddr=sbtbl;
      byteptr=type$ord$num;
      aptraddr=aptraddr+1;
      addrptr=parent$type;
      sbtbl=sbtbl+3;
    end;

/*     42   <subrange type> ::= <constant> .. <constant>                */
    call enter$subr$entry;

/*     43   <structured type> ::= <unpacked structured type>            */
    ;
/*     44                       | packed                               */
/*     44                         <unpacked structured type>           */
    ;

/*     45   <unpacked structured type> ::= <array type>                 */
    ;
/*     46                                  | <record type>              */
```

```
    ;
/*     47                                  | <set type>                  */
    ;
/*     48                                  | <file type>                 */
    ;

/*     49   <array type> ::= array <lp> <index type string> <rp>       */
/*     49                    of <component type>                       */
    do;
     if arry$ptr = -1 then arry$ptr=0;
     call enter$complex$type(17h);
     ary$dm$adr$ptr=ary$dm$adr$ptr-num$arry$dimen(arry$ptr);
     arry$base=base;
     call limits((num$arry$dimen(arry$ptr)*2)+3);
     call setaddrptr(5);
     byteptr=num$arry$dimen(arry$ptr);
     call setaddrptr(6);
     addr$ptr=type$loct;
     call allc$offset(type$loct);
     base=arry$base;
     call setaddrptr(8);
     addrptr=arry$qty(arry$ptr)*allc$qty;
     call setaddrptr(10);
     byteptr=allc$basic$type;
     do subr$form=0 to (num$arry$dimen(arry$ptr)-1);
       call setaddrptr(11+(2*subr$form));
       addrptr=arry$dimen(ary$dm$adr$ptr+subrform+1);
     end;
     type$loct=base;
     sbtbl=sbtbl+((num$arry$dimen(arry$ptr)*2)+6);
    arry$ptr=arry$ptr-1;
    end;

/*     50   <lp> ::= (*                                                */
    ;

/*     51   <rp> ::= *)                                                */
    ;

/*     52   <index type string> ::= <index type>                       */
    do;
     if arry$ptr=arry$nest-1 then
       do;
         call error('an');
         ary$dm$adr$ptr=ary$dm$adr$ptr-num$arry$dimen(arry$ptr);
       end;
     else arry$ptr= arry$ptr+1;
         arry$dim$ptr=0;
         ary$dm$adr$ptr=ary$dm$adr$ptr+1;
         arry$dimen(ary$dm$adr$ptr)=type$loct;
         arry$qty(arry$ptr)=allc$index$offset;
         num$arry$dimen(arry$ptr)=1;
    end;
/*     53                          | <index type string> ,              */
/*     53                            <index type>                      */
    do;
    if arry$dim$ptr=max$num$arry$dimen-1 then
      call error('ad');
    else     arry$dim$ptr=arry$dim$ptr+1;
             ary$dm$adr$ptr=ary$dm$adr$ptr+1;
             arry$dimen(ary$dm$adr$ptr)=type$loct;
             arry$qty(arry$ptr)=arry$qty(arry$ptr)*allc$index$offset;
             num$arry$dimen(arry$ptr)=num$arry$dimen(arry$ptr)+1;
    end;

/*     54   <index type> ::= <simple type>                             */
    ;

/*     55   <component type> ::= <type>                                */
   ;

/*     56   <record type> ::= record <field list> end                  */
```

```
   do;
  variant$part(rec$nst)=false;
  base,type$loct=rec$par$adr(rec$nst);
  if var$cas$val(rec$nst) <> 0 then
   call error('iv');
  call setaddrptr(5);
  addrptr=fxd$ofst$bse(rec$nst)-1;
  call setaddrptr(7);
  addrptr= prv$sbtbl$entry;
  rec$nst=rec$nst-1;
 end;

/*     57   <field list> ::= <fixed part>                              */
   ;
/*     58                  | <fixed part> ; <variant part>             */
   ;
/*     59                  | <variant part>                            */
   ;

/*     60   <fixed part> ::= <record section>                          */
   ;
/*     61                  | <fixed part> ; <record section>           */
   ;

/*     62   <record section> ::= <field ident string> : <type>         */
   do;
    call allc$offset(type$loct);
    /* allc$basic$type and allc$qty are set */
    do ptrptr = 0 to record$ptr;
      base = rec$addr(ptrptr);
      call setaddrptr(5);
      call setaddrptr(8+byteptr);
      addrptr=allc$qty;
      aptraddr=aptraddr+2;
      addrptr=type$loct;
      aptraddr=aptraddr+2;
      addrptr=cur$ofst(rec$nst);
      cur$ofst(rec$nst)=cur$ofst(rec$nst)
        + allc$qty;
   end;
   record$ptr=0;
   if fxd$ofst$bse(rec$nst) < cur$ofst(rec$nst)
   then fxd$ofst$bse(rec$nst)=cur$ofst(rec$nst);
  end;
/*     63                       |                                      */
   ;

/*     64   <field ident string> ::= <field ident>                     */
   ;
/*     65                          | <field ident string> ,            */
/*     65                            <field ident>                     */
   ;

/*     66   <field ident> ::= <identifier>                             */
   do;
    if record$ptr <> 5 then record$ptr=record$ptr+1;
    else call error('rn');
    rec$addr(record$ptr)=sbtbl;
    call enter$var$id(58h,sp,type$dcle);
    call limits(8);
    aptraddr=sbtbl;
    addrptr=rec$par$adr(rec$nst);
    sbtbl=sbtbl+8;
    if variant$part(rec$nst) then
        do;
         base=rec$addr(record$ptr);
         call limits(2);
         call setaddrptr(4);
         byteptr=0dfh;
        end;
    end;
```

```
/*     67   <variant part> ::= case <tag field> <type ident> of      */
/*     67                        <variant string>                    */
   ;
/*     68                      | case <type ident> of                */
/*     68                        <variant string>                    */
   ;

/*     69   <variant string> ::= <variant>                          */
   ;
/*     70                        | <variant string> ; <variant>     */
   ;

/*     71   <tag field> ::= <field ident> :                         */
   tag$fd(rec$nst)=true;

/*     72   <variant> ::= <case label list> : ( <field list> )      */
   ;
/*     73               |                                           */
   ;

/*     74   <case label list> ::= <case label>                      */
   ;
/*     75                         | <case label list> , <case label> */
   ;

/*     76   <case label> ::= <constant>                             */
 if case$stmt then
   do;
     /* insert case stmt routines */
   end;
 else
   do;
    if not variant$part(rec$nst) then
     do;
      variant$part(rec$nst)=true;
      var$cas$tp(rec$nst)=type$loct;
      var$cas$val(rec$nst)=allc$index$offset;
      call allc$offset(type$loct);
      if tag$fd(rec$nst) then
       do;
         tag$fd(rec$nst)=false;
         base=rec$addr(record$ptr);
         call setaddrptr(4);
         byteptr=9fh;
         call setaddrptr(5);
         call setaddrptr(8+byteptr);
         addrptr=var$cas$val(rec$nst);
         aptraddr=aptraddr+2;
         addrptr=var$cas$tp(rec$nst);
         aptraddr=aptraddr+2;
         addrptr=cur$ofst(rec$nst);
      cur$ofst(rec$nst)=cur$ofst(rec$nst)+allc$qty;
       end;
      var$ofst$bse(rec$nst)=cur$ofst(rec$nst);
      fxd$ofst$bse(rec$nst)=cur$ofst(rec$nst);
     end;
    /* call compare$const$variant; */
    /* the routine above checks the case lable with the variant type */

    cur$ofst(rec$nst)=var$ofst$bse(rec$nst);
    vecptr=vecptr-1;
    const$ptr,const$indx,const$pn$ptr=0;
   end;

/*     77   <set type> ::= set of <base type>                       */
   call enter$struct$type(27h);

/*     78   <base type> ::= <simple type>                           */
   ;

/*     79   <file type> ::= file of <type>                          */
   call enter$struct$type(2fh);
```

```
/*     80   <pointer type> ::= $ <type ident>                          */
   call enter$struct$type(37h);

/*     81   <vdp> ::=                                                  */
   ;
/*     82           | var <var declar string> ;                       */
   ;

/*     83   <var declar string> ::= <var declar>                       */
   ;
/*     84                         | <var declar string> ;             */
/*     84                           <var declar>                       */
   ;

/*     85   <var declar> ::= <ident var string> : <type>               */
   do;
    call allc$offset(type$loct);
    if not allocate then
      do;
       allc$qty=1;
       allc$basic$type=0;
      end;
    do while var$ptr <> -1;
      base=var$base(var$ptr);
      call setaddrptr(4);
      byteptr=shl(allc$basic$type,3) or var$entry;
      aptraddr=var$base1(var$ptr);
      addrptr=alloc$addr;
      alloc$addr=alloc$addr+allc$qty;
      aptraddr=aptraddr+2;
      addrptr=type$loct;
      var$ptr=var$ptr-1;
    end;
   end;

/*     86   <ident var string> ::= <identifier>                        */
   do;
     var$ptr=0;
     var$base=sbtbl;
     call enter$var$id(0,sp,var$entry);
     call limits(4);
     var$base1=sbtbl;
     sbtbl=sbtbl+4;
   end;
/*     87                       | <ident var string> ,                */
/*     87                         <identifier>                        */
   do;
     if var$ptr <> 10 then
     do;
     var$ptr=var$ptr+1;
     var$base(var$ptr)=sbtbl;
     call enter$var$id(0,sp,var$entry);
     call limits(4);
     var$base1(var$ptr)=sbtbl;
     sbtbl=sbtbl+4;
   end;
     else call error('vn');
     end;

/*     88   <p$fdp> ::=                                                */
   ;
/*     89             | <porf declar>                                  */
   ;

/*     90   <porf declar> ::= <proc or funct> ;                        */
   ;
/*     91                   | <porf declar> <proc or funct> ;          */
   ;

/*     92   <proc or funct> ::= <procedure declaration>                */
   ;
/*     93                     | <function declar>                      */
```

```
   ;
/*     94   <procedure declaration> ::= <procedure heading>          */
/*     94                               <block>                      */
   ;
/*     95   <procedure heading> ::= <proc id> ;                      */
   ;
/*     96                         | <proc id> (                      */
/*     96                           <formal para sect list> ) ;      */
   ;
/*     97   <proc id> ::= procedure <identifier>                     */
   ;
/*     98   <formal para sect list> ::= <formal para sect>           */
   ;
/*     99                             | <formal para sect list> ;    */
/*     99                               <formal para sect>           */
   ;
/*    100   <formal para sect> ::= <para group>                      */
   ;
/*    101                        | var <para group>                 */
   ;
/*    102                        | function <para group>             */
   ;
/*    103                        | procedure <proc ident list>       */
   ;
/*    104   <proc ident list> ::= <identifier>                       */
   ;
/*    105                       | <proc ident list> , <identifier>   */
   ;
/*    106   <para group> ::= <para ident list> : <type ident>        */
   ;
/*    107   <para ident list> ::= <identifier>                       */
   ;
/*    108                       | <para ident list> , <identifier>   */
   ;
/*    109   <function declar> ::= <function heading> <block>         */
   ;
/*    110   <function heading> ::= <funct id> : <result type> ;      */
   ;
/*    111                        | <funct id> (                     */
/*    111                          <formal para sect list> ) :      */
/*    111                          <result type> ;                  */
   ;
/*    112   <funct id> ::= function <identifier>                     */
   ;
/*    113   <result type> ::= <type ident>                           */
   ;
/*    114   <stmtp> ::= <compound stmt>                              */
   ;
/*    115   <stmt> ::= <bal stmt>                                    */
   ;
/*    116            | <unbal stmt>                                  */
   ;
/*    117            | <label def> <stmt>                            */
   ;
/*    118   <bal stmt> ::= <if clause> <true part> else <bal stmt>   */
   do;
    call generate(lbl);
```

```
    call generate(low(ifSlbl(ifSptr)+1));
    call generate(high(ifSlbl(ifSptr)+1));
    ifSptr=ifSptr-1;
   end;
/*    119                 | <simple stmt>                                  */
   ;

/*    120   <unbal stmt> ::= <if clause> <stmt>                            */
   do;
    call generate(lbl);
    call generate(low(ifSlbl(ifSptr)));
    call generate(high(ifSlbl(ifSptr)));
    ifSptr=ifSptr-1;
   end;
/*    121                   | <if clause> <true part> else                 */
/*    121                     <unbal stmt>                                 */
   do;
    call generate(lbl);
    call generate(low(ifSlbl(ifSptr)+1));
    call generate(high(ifSlbl(ifSptr)+1));
    ifSptr=ifSptr-1;
   end;

/*    122   <if clause> ::= <if> <expression> then                         */
   do;
    if expStype(expSptr)=booleanStype then
      do;
        expSptr=expSptr-1;
        call generate(notx);
        call generate(blc);
        call generate(low(ifSlbl(ifSptr)));
        call generate(high(ifSlbl(ifSptr)));
      end;
     else call error('ce');
   end;

/*    123   <if> ::= if                                                    */
   do;
     if ifSptr=9 then
      do;
       call error('io');
       call mon3;
      end;
    ifSptr=ifSptr+1;
    ifSlbl(ifSptr)=lablcount;
    lablcount=lablcount=2;
   end;

/*    124   <true part> ::= <bal stmt>                                     */
   do;
     call generate(br1);
     call generate(low(ifSlbl(ifSptr)+1));
     call generate(high(ifSlbl(ifSptr)+1));
     call generate(lbl);
     call generate(low(ifSlbl(ifSptr)));
     call generate(high(ifSlbl(ifSptr)));
   end;

/*    125   <label def> ::= <label> :                                      */
   if lookupSpnSid(mp,lablSentry) then
     do;
      call setaddrptr(5);
      call setaddrptr(6+byteptr);
      call generate(lbl);
      call generate(low(addrptr));
      call generate(high(addrptr));
    end;
   else call error('ul');

/*    126   <simple stmt> ::= <assignment stmt>                            */
   ;
/*    127                   | <procedure stmt>                             */
```

```
   ;
/*    128                    | <repetitive stmt>                          */
   ;
/*    129                    | <case stmt>                                */
   ;
/*    130                    | <with stmt>                                */
   ;
/*    131                    | <read stmt>                                */
   ;
/*    132                    | <write stmt>                               */
   ;
/*    133                    | <goto stmt>                                */
   ;
/*    134                    | <compound stmt>                            */
   ;
/*    135                    |                                            */
   ;

/*    136   <assignment stmt> ::= <variable> := <expression>             */
   do;
    call generate(lita);
    call generate(low(var$base(var$ptr)));
    call generate(high(var$base(var$ptr)));
    do case exp$type(exp$ptr);
     /* case 0 - ord type */
     if var$type(var$ptr)<> 11h or exp$type$addr(exp$ptr)<>
      var$base1(var$ptr) then
        goto error$loop;
     else call generate(std);
     /* case 1 - integer type */
     if var$type(var$ptr)=09h then
      call generate(stdi);
     else do;
       if var$type(var$ptr)=0ah then
       do;
        call generate(cnai);
        call generate(stdb);
       end;
      else goto error$loop;
     end;
     /* case 2 - char$type */
     if var$type(var$ptr)=0bh then
      call generate(std);
     else goto error$loop;
     /* case 3 - real type */
     if var$type(var$ptr)=0ah then
      call generate(stdb);
     else goto error$loop;
     /* case 4 - string type */
     ; /* not implimented */
     /* case 5 - boolean type */
     if var$type(var$ptr)=08h then
      call generate(std);
     else goto error$loop;
   end; /* of case */
   goto second$loop;
    error$loop: call error('at');
   second$loop:
     exp$ptr=exp$ptr-1;
     var$ptr=var$ptr-1;
   end;

/*    137   <variable> ::= <entire variable>                             */
   ;
/*    138                | <variable> $                                  */
   ;
/*    139                | <variable> <lp> <expres list> <rp>            */
   ;
/*    140                | <variable> . <field ident>                    */
   ;

/*    141   <entire variable> ::= <variable ident>                       */
```

```
   ;
/*    142   <variable ident> ::= <identifier>                          */
   if not lookup$only(sp) then
     call error('dt');
   else do;
    call set$var$type; /* lookup$addr set here */
   end;
/*    143   <expres list> ::= <expression>                             */
   ;
/*    144                     | <expres list> , <expression>           */
   ;
/*    145   <expression> ::= <simple expression>                       */
   ;
/*    146                    | <simple expression>                     */
/*    146                      <relational operator>                   */
/*    146                      <simple expression>                     */
   do;
    op$ptr=op$ptr-1;
    if check$exprs$type then
     do;
     if (exp$type(exp$ptr)<>4h)or(exp$type(exp$ptr)<>3h)then
      do case (op$type(op$ptr)-8h);
       /* case 0 - * */
       call generate(eqli);
       call generate(neqi);
       call generate(leqi);
       call generate(geqi);
       call generate(lssi);
       call generate(grti);
       ; /* "in" not implimented */
      end; /* of case for integers */
     if exp$type(exp$ptr)=3h then
      do case (op$type(op$ptr)-8h);
       call generate(eqlb);
       call generate(neqb);
       call generate(leqb);
       call generate(geqb);
       call generate(lssb);
       call generate(grtb);
       ; /* "in" not implimented */
      end; /* of case for reals */
     if exp$type(exp$ptr)=4h then
       do; /* tests for strings not implimented */
       end;
    exp$type(exp$ptr)=boolean$type;
   end;
   else call error('ce');
      exp$ptr=exp$ptr-1;
   end;
/*    147   <relational operator> ::= =                                */
   do;
    op$type(op$ptr)=08h;
    op$ptr=op$ptr+1; /* next to fill */
   end;
/*    148                             | < >                            */
   do;
    op$type(op$ptr)=09h;
    op$ptr=op$ptr+1;
   end;
/*    149                             | < =                            */
   do;
    op$type(op$ptr)=0ah;
    op$ptr=op$ptr+1;
   end;
/*    150                             | > =                            */
   do;
    op$type(op$ptr)=0bh;
    op$ptr=op$ptr+1;
```

```
   end;
/*    151                              | <                                  */
   do;
    op$type(op$ptr)=0ch;
    op$ptr=op$ptr+1;
   end;
/*    152                              | >                                  */
   do;
    op$type(op$ptr)=0dh;
    op$ptr=op$ptr+1;
   end;
/*    153                              | in                                 */
   do;
    op$type(op$ptr)=0eh;
    op$ptr=op$ptr+1;
   end;

/*    154   <term> ::= <factor>                                             */
   ;
/*    155            | <term> <multiplying operator> <factor>               */
   do;
    op$ptr=op$ptr-1;
    if check$exprs$type then
     do;
      if op$type(op$ptr)=0h then /* multiplication */
       do case exp$type(exp$ptr);
        call error('ce'); /* case 0 - ord */
        call generate(muli); /* case 1 - integer */
        call error('ce');
        call generate(mulb); /* case 3 - real */
        call error('ce'); /* case 4 - string */
        call error('ce'); /* case 5 - boolean */
       end; /* of mul case */
      if op$type(op$ptr)=1h then /* real division */
       do case exp$type(exp$ptr);
        call error('ce'); /* case 0 - ord */
        do; /* case 1 - integer with real result */
         call generate(cnvi); /* convert 1st integer */
         call generate(cn2i); /* convert 2nd integer */
         call generate(divb);
         exp$type(exp$ptr-1)=unsign$expon;
        end;
        call error('ce'); /* case 2 - char */
        call generate(divb); /* case 3 - real */
        call error('ce'); /* case 4 - string */
        call error('ce'); /* case 5 - boolean */
       end; /* of div case */
      if op$type(op$ptr)=2h then /* integer divide */
       if exp$type(exp$ptr)= integer$type then
        call generate(divi);
       else call error('ce');
      if op$type(op$ptr)=3h then /* integer mod */
       if exp$type(exp$ptr)=integer$type then
        call generate(dcri); /* "mod" not implimented */
       else call error('ce');
      if op$type(op$ptr)=4h then /* boolean and */
       if exp$type(exp$ptr)=boolean$type then
        call generate(andx);
       else call error('ce');
     end;
    else call error('ce');
    exp$ptr=exp$ptr-1;
   end;

/*    156   <multiplying operator> ::= *                                    */
   do;
    op$type(op$ptr)=0h;
    op$ptr=op$ptr+1;
   end;
/*    157                              | /                                  */
   do;
    op$type(op$ptr)=01h;
```

```
    op$ptr=op$ptr+1;
   end;
/*     158                                | div                                */
   do;
    op$type(op$ptr)=02h;
    op$ptr=op$ptr+1;
   end;
/*     159                                | mod                                */
   do;
    op$type(op$ptr)=03h;
    op$ptr=op$ptr+1;
   end;
/*     160                                | and                                */
   do;
    op$type(op$ptr)=04h;
    op$ptr=op$ptr+1;
   end;

/*     161   <simple expression> ::= <term>                                    */
   ;
/*     162                             | <sign> <term>                        */
 if signtype = neg then
   do;
    if exp$type(exp$ptr)=unsign$expon then
      call generate(negb);
    else if exp$type(exp$ptr)=integer$type then
      call generate(negi);
    else call error('uo');
   end;
/*     163                             | <simple expression>                  */
/*     163                               <adding operator> <term>             */
   do;
    op$ptr=op$ptr-1;
    if check$exprs$type then
      do;
        if op$type(op$ptr)=5h then /* arith add */
        do case exp$type(exp$ptr);
          call error('ce');
          call generate(addi);/* case 1 - integer */
          call error('ce'); /* case 2 - char */
          call generate(addb); /* case3 - real */
          call error('ce'); /* case 4 - string */
          call error('ce'); /* case 5 - boolean */
        end; /* case */
        if op$type(op$ptr)= 6h then /* arith subtrc */
        do case exp$type(exp$ptr);
          call error('ce'); /* case 0 - ord type */
          call generate(subi);
          call error('ce');
          ; /* call generate(subb); */
            /* not implimented */
          call error('ce');
          call error('ce');
        end;
        if op$type(op$ptr)=7h then /* boolean or */
        do;
         if exp$type(exp$ptr)=boolean$type then
             call generate(bor);
         else call error('ce');
        end;
      end;
      else call error('ce');
      exp$ptr=exp$ptr-1;
     end;

/*     164   <adding operator> ::= +                                           */
   do;
      op$type(op$ptr)=05h;
      op$ptr=op$ptr+1;
     end;
/*     165                          | -                                       */
   do;
```

```
    op$type(op$ptr)=06h;
    op$ptr=op$ptr+1;
   end;
/*    166                          | or                                    */
   do;
    op$type(op$ptr)=07h;
    op$ptr=op$ptr+1;
   end;

/*    167   <factor> ::= <variable>                                        */
   do;
    if exp$ptr=11 then
      do;
       call error('es');
        call mon3;
      end;
    exp$ptr=exp$ptr+1;
    call load$variable;
   end;
/*    168              | <variable> ( <actual para list> )                */
   ;
/*    169              | ( <expression> )                                 */
   ;
/*    170              | <set>                                            */
   ;
/*    171              | not <factor>                                     */
  if exp$type(exp$ptr)=boolean$type then
   call generate(notx);
  else call error('ce');
/*    172              | <number>                                         */
   do;
    if exp$ptr=11 then
      do;
        call error('es');
        call mon3;
      end;
    exp$ptr=exp$ptr+1;
    if typenum=integer$type then
     do;
       exp$type(exp$ptr)=integer$type;
       allc$qty=converti(sp,pos);
       call generate(ldii);
       call generate(low(allc$qty));
       call generate(high(allc$qty));
     end;
    else do;
       exp$type(exp$ptr)=unsign$expon;
       call convrtbcd(sp,pos);
       call generate(ldib);
       do ptrptr=0 to bcdsize-1;
         call generate(bcdnum(ptrptr));
       end;
      end;
    end;
/*    173              | nil                                              */
   ;
/*    174              | <string>                                         */
   ;

/*    175   <actual para list> ::= <actual para>                           */
   ;
/*    176                          | <actual para list> ,                  */
/*    176                            <actual para>                       */
   ;

/*    177   <set> ::= <lp> <element list> <rp>                             */
   ;

/*    178   <element list> ::=                                             */
   ;
/*    179                       | <xelement list>                          */
   ;
```

```
/*    180   <xelement list> ::= <element>                                  */
   ;
/*    181                     | <xelement list> , <element>                */
   ;

/*    182   <element> ::= <expression>                                     */
   ;
/*    183               | <expression> .. <expression>                    */
   ;

/*    184   <goto stmt> ::= <goto> <label>                                 */
   if lookup$pn$id(sp,labl$entry) then
     do;
     call setaddrptr(5);
     call setaddrptr(6+byteptr);
     call generate(low(addrptr));
     call generate(high(addrptr));
    end;
   else do;
     call error('ul');
     call generate(nop); call generate(nop);
   end;

/*    185   <goto> ::= goto                                                */
   call generate(br1);

/*    186   <compound stmt> ::= <begin> <stmt lists> end                   */
   ;

/*    187   <begin> ::= begin                                              */
   ;

/*    188   <stmt lists> ::= <stmt>                                        */
   ;
/*    189                  | <stmt lists> ; <stmt>                        */
   ;

/*    190   <procedure stmt> ::= <procedure ident>                         */
   ;
/*    191                      | <procedure ident> (                      */
/*    191                        <actual para list> )                     */
   ;

/*    192   <procedure ident> ::= <identifier>                             */
   ;

/*    193   <actual para> ::= <expression>                                 */
   ;

/*    194   <rec variable list> ::= <variable>                             */
   ;
/*    195                         | <rec variable list> ,                  */
/*    195                           <variable>                             */
   ;

/*    196   <read stmt> ::= <read head> ( <io list> )                      */
   ;

/*    197   <read head> ::= read                                           */
   do;
    write$stmt=false;
    allocate=false;
   end;
/*    198                 | readln                                        */
   do;
    write$stmt=false;
    allocate=true;
   end;

/*    199   <write stmt> ::= <write head> ( <io list> )                    */
   if write$stmt then call generate(dmp);
/*    200                  | <write head>                                 */
```

```
   if allocate then call generate(dump);

/*    201   <write head> ::= write                                          */
   do;
    allocate=false;
    write$stmt=true;
   end;
/*    202                  | writeln                                       */
   do;
    allocate=true;
    write$stmt=true;
   end;

/*    203   <io list> ::= <file ident> * , <var list>                       */
   ;
/* not implimented */
/*    204               | <var list>                                       */
   ;

/*    205   <var list> ::= <variable>                                       */
   if write$stmt then call write$var;
   else call read$var;
/*    206               | <string>                                         */
   if write$stmt then call write$string;
   else do; /* not implimented */
        end;
/*    207               | <var list> , <variable>                          */
   if write$stmt then call write$var;
   else call read$var;
/*    208               | <var list> , <string>                            */
   if write$stmt then call write$string;
   else do; /* not implimented */
        end;

/*    209   <case stmt> ::= <case express> <case list elemt list>           */
/*    209                   end                                             */
   ;

/*    210   <case express> ::= case <expression> of                         */
   do;
    case$stmt=true;
   end;

/*    211   <case list elemt list> ::= <case list element>                 */
   ;
/*    212                            | <case list elemt list> ;            */
/*    212                              <case list element>                 */
   ;

/*    213   <case list element> ::=                                         */
   ;
/*    214                         | <case label list> : <stmt>             */
   ;

/*    215   <repetitive stmt> ::= <while stmt>                              */
   ;
/*    216                       | <repeat stmt>                             */
   ;
/*    217                       | <for stmt>                                */
   ;

/*    218   <with stmt> ::= <with> <rec variable list> <do>                 */
/*    218                   <bal stmt>                                      */
   ;

/*    219   <with> ::= with                                                 */
   ;

/*    220   <do> ::= do                                                     */
   ;

/*    221   <while stmt> ::= <while> <expression> <do> <bal stmt>           */
```

```
   ;
/*    222   <while> ::= while                                          */
   ;
/*    223   <for stmt> ::= <for> <control variable> := <for list>      */
/*    223                  <do> <bal stmt>                             */
   ;
/*    224   <for> ::= for                                              */
   ;
/*    225   <for list> ::= <initial value> <to> <final value>         */
   ;
/*    226                | <initial value> <downto> <final value>     */
   ;
/*    227   <control variable> ::= <identifier>                        */
   ;
/*    228   <initial value> ::= <expression>                           */
   ;
/*    229   <final value> ::= <expression>                             */
   ;
/*    230   <repeat stmt> ::= <repeat> <stmt lists> <until>            */
/*    230                     <expression>                             */
   do;
    repeat$ptr=repeat$ptr-1;
    if exp$type(exp$ptr)=boolean$type then
     do;
       exp$ptr=exp$ptr-1;
       call generate(notx);
       call generate(blc);
       call generate(low(repeat$lbl(repeat$ptr)));
       call generate(high(repeat$lbl(repeat$ptr)));
     end;
    else call error('ce');
   end;
/*    231   <repeat> ::= repeat                                        */
   do;
     call generate(lbl);
     call generate(low(lablcount));
     call generate(high(lablcount));
     repeat$lbl(repeat$ptr)=lablcount;
     lablcount=lablcount+1; repeat$ptr=repeat$ptr+1;
   end;
/*    232   <until> ::= until                                          */
   ;
/*    233   <to> ::= to                                                */
   ;
/*    234   <downto> ::= downto                                        */
   ;
/*    235      not used, overflow indicator                          */
   ;
end;                                          /* of case statement */

end synthesize;


/*****************************************************************/
/*****************************************************************/
/***                  error recovery routines                  ***/
/*****************************************************************/
/*****************************************************************/
```

```
noconflict: proc (cstate) byte;
    dcl cstate statesize, (i,j,k) indexsize;
    j= index1(cstate);
    k= j + index2(cstate) - 1;
    do i = j to k;
       if read1(i) = token then return true;
    end;
    return false;
end noconflict;


recover: proc statesize;
    dcl tsp byte, rstate statesize;
    do forever;
       tsp = sp;
       do while tsp <> 255;
          if noconflict(rstate:=statestack(tsp)) then
             do;                    /* state will read token */
                if sp <> tsp then sp = tsp - 1;
                return rstate;
             end;
          tsp = tsp - 1;
       end;
       call scanner;
    end;
end recover;



/*****************************************************************************/
/*****************************************************************************/
/***                        lalr  parser  routines                         ***/
/*****************************************************************************/
/*****************************************************************************/


    do;                                         /*block for declarations*/
    dcl    (i,j,k) indexsize, index byte;


    initialize: proc;
        call initialize$scanner;
        call initialize$symtbl;
        call initialize$synthesize;
        call title;
    end initialize;


    getin1: proc indexsize;
        return index1(state);
    end getin1;


    getin2: proc indexsize;
        return index2(state);
    end getin2;


    incsp: proc;
        if (sp := sp + 1) = length(statestack) then
            call error('so');
    end incsp;


    lookahead: proc;
        if nolook then
           do;
              call scanner;
              nolook = false;
              if listtoken then
                 call print$token;
           end;
```

```
    end lookahead;


    set$varc$i: proc(i);          /* set varc, and incrmnt varindex */
        dcl   i byte;
        varc(varindex)=i;
        if (varindex:=varindex+1) > length(varc) then
           call error('vo');
    end set$varc$i;


              /* initialize for input - output operations  */
call move(.rfcb,.wfcb,9);      /* put filename in write fcb */
call setup$int$file; /* creates output file for generated code */
call initialize;

do forever;
   do while true;                          /* initialize variables */
      compiling,nolook=true;
      state=starts;
      sp=255;
      varindex,var = 0;

      do while compiling;
         if state<=maxrno then                /* read state */
            do;
               call incsp;
               statestack(sp)=state;
               i=getin1;
               call lookahead;
               j=i+getin2-1;
               do i=i to j;
                  if read1(i)=token then             /* save token */
                     do;           /* copy accum to proper position */
                        var(sp)=varindex;
                        do index = 0 to accum;
                           call set$varc$i(accum(index));
                        end;
                        hash(sp) = hashcode;
                                  /* save relative table location */
                        state=read2(i);
                        nolook=true;
                        i=j;
                     end;
                  else if i=j then
                          do;
                             call error('np');
                             if (state := recover)=0 then
                                compiling = false;
                          end;
               end;
            end;
         else if state>maxpno then         /* apply production state */
                 do;
                    mp=sp-getin2;
                    mpp1=mp+1;
                    production = state-maxpno;
                    call synthesize;
                    sp=mp;
                    i=getin1;
                    varindex=var(sp);
                    j=statestack(sp);
                    do while (k:=apply1(i)) <> 0 and j <> k;
                       i=i+1;
                    end;
                    if (state:= apply2(i))=0 then compiling = false;
                 end;
              else if state<= maxlno then      /* lookahead state */
                      do;
                         i=getin1;
                         call lookahead;
                         do while (k:=look1(i)) <> 0 and token <> k;
```

```
                    i=i+1;
                  end;
                  state=look2(i);
                end;
                                                  /* push state  */
              else do;
                    call incsp;
                    statestack(sp)= getin2;
                    state=getin1;
                  end;
          end;                                      /* of while compiling  */
       end;                                        /*of while      true     */
    end;                                         /*of do forever*/

end;                                          /* of block for parser  */
end;                                         /*of block for declarations*/
eof
```

```
100h:   /*load point for translator*/


/*****************************************************************************/
/*****************************************************************************/
/***                    s y s t e m   l i t e r a l s                      ***/
/*****************************************************************************/
/*****************************************************************************/

declare lit     literally 'literally',
        dcl             lit 'declare',
        proc            lit 'procedure',
        bdos            lit '5h',           /*entry point to disk op. sys*/
        boot            lit '0',           /* exit to return to op. sys. */
        true            lit '1',
        addr            lit 'address',
        false           lit '0',
        bdSlen          lit '8',
        intSlen         lit '2',
        fileeof         lit '1',
        comrecsize      lit '128',
        eoffiller       lit '1ah',
        pinrecsize      lit '128',
        forever         lit 'while true';

dcl     sbloc           addr     initial(80h),
        codestrt        addr     initial(0),
        varstrt         addr     initial(140h),
        progSsize       addr     initial(0),
        nextchar        byte,
        codecount       addr     initial(0),
        varcount        addr     initial(0),
        genSbuff(80)    byte,
        codesize        addr    initial(100h),/* adds bytes generated */
        tempaddr        addr,
        tempbyte        byte,
        combuff(comrecsize)     byte,
        pinptr          byte     initial(pinrecsize),
        errcount        addr     initial(0),
        pincode         byte,
        pinbuff         based    sbloc   (pinrecsize)  byte,
        startbdos       addr     initial(6h),      /*ptr to addr of bdos*/
        base            addr,
        max             based    startbdos addr,
        comptr          byte     initial(255),
        curpinrecsize   byte     initial(pinrecsize),
        rfcbaddr        addr     initial(5ch),
        loop            byte,
        wfcb(33)        byte     initial(0,'        ','com',0,0,0,0),
        rfcb            based    rfcbaddr(33) byte,
        cspc(66)        byte,
        cspaddr(66)     addr,
        spSmax          addr,       /* stack pointer max */
        nolook          byte;


dcl     pass1           addr     initial(true),
        pass2           byte     initial(false),
        nopinfile       byte     initial(false);

/*****************************************************************************/
/*****************************************************************************/
/***                g l o b a l    p r o c e d u r e s                     ***/
/*****************************************************************************/
/*****************************************************************************/

mon1: proc(f,a);
    dcl   f byte,
    a addr;
    go to bdos;
end mon1;
```

```
mon2: proc (f,a) byte;
    dcl     f byte, a addr;
    go to bdos;
end mon2;


deleteSfile: proc(a);
    dcl a addr;
    call mon1(19,a);
end deleteSfile;


makeSfile: proc(a) byte;
    dcl a addr;
    return mon2(22,a);
end makeSfile;


setSdma: proc(a);
    dcl a addr;
    call mon1(26,a);
end setSdma;


wrtSconSrcrd: proc(a) byte;
    dcl a addr;
    return mon2(21,a);
end wrtSconSrcrd;


closeSfile: proc(a) byte;
    dcl a addr;
    return mon2(16,a);
end closeSfile;


openSfile: proc(a) byte;
    dcl a addr;
    return mon2(15,a);
end openSfile;


readSfile: proc(a) byte;
    dcl a addr;
    return mon2(20,a);
end readSfile;


deleteSpinSfile: proc;
    call deleteSfile(rfcbaddr);
end deleteSpinSfile;


move: proc (a,b,l);                     /*moves fm a to b for l bytes */
    dcl     (a,b) addr,                 /* l < 255 bytes */
    (s based a, d based b,l) byte;
    do while (l:=l - 1) <> 255;
       d=s;
       b=b + 1;
       a=a + 1;
    end;
end move;


print: proc(a);
    dcl     a addr;
    call mon1(9,a);
end print;


printchar: proc(msg);
    dcl  msg byte;
```

```
    call mon1(2,msg);
end printchar;


error: proc(errcode);
    dcl errcode  addr,
    i  byte;
    errcount = errcount + 1;
    call print(.'***$');
    call print(.'  error$');
    call printchar(' ');
    call printchar(high(errcode));
    call printchar(low(errcode));
end error;


diskerr: proc;
    do;
       call print(.'disk error    $');
       goto boot;
    end;
end diskerr;

/*****************************************************************************/
/*****************************************************************************/
/***          f i l e   m a n i p u l a t i n g   r o u t i n e s     ***/
/*****************************************************************************/
/*****************************************************************************/

setup$com$file:proc;
   if nopinfile then  /* only make file if this toggle off */
      return;
   call move(.rfcb,.wfcb,9);
      wfcb(32) = 0;
   call delete$file(.wfcb);
   if make$file(.wfcb) = 255 then
      call diskerr;
end setup$com$file;


write$com$file: proc;
   if nopinfile then
      return;
   call set$dma(.combuff);
   if wrt$com$rcrd(.wfcb) <> 0 then
      call diskerr;
   call set$dma(sbloc);           /* reset dma addr */
end write$com$file;


emit: proc(objcode);
   dcl objcode byte;
   if (comptr := comptr+1) >= comrecsize then
   /* write to disk */
      do;
        call write$com$file;
         comptr = 0;
      end;
   combuff(comptr) = objcode;
end emit;


generate: proc(objcode);
   dcl objcode byte;
      codesize = codesize+1;
      call emit(objcode);
end generate;


gen$five: proc(a,b,c,d,e);
    dcl (a,b,c,d,e) byte;
    codesize = codesize + 5;
```

```
    call emit(a);
    call emit(b);
    call emit(c);
    call emit(d);
    call emit(e);
end gen5five;


gten: proc(a,b,c,d,e,f,g,h,i,j);
    dcl (a,b,c,d,e,f,g,h,i,j) byte;
    codesize = codesize + 10;
    call emit(a);
    call emit(b);
    call emit(c);
    call emit(d);
    call emit(e);
    call emit(f);
    call emit(g);
    call emit(h);
    call emit(i);
    call emit(j);
end gten;


close5com5file: proc;
  /* closes a file */
  if close5file(.wfcb) = 255 then
     call diskerr;
end close5com5file;


open5pin5file: proc;
    call move(.'pin',rfcbaddr+9,3);
    rfcb(32) = 0;
    if open5file(rfcbaddr) = 255 then
       do;
          call print(.'no intermediate file found $');
          go to boot;
       end;
end open5pin5file;


rewind5pin5file:proc;      /* cp/m does not require any action */
    return;                   /* prior to reopening */
end rewind5pin5file;


read5pin5file:proc byte;
    dcl     dcnt byte;
    if (dcnt:=read5file(rfcbaddr)) > fileeof then
       call diskerr;
    return dcnt;
end read5pin5file;


get5next5byte: proc byte;
    dcl  addeof data('1');
    next5pin5char: proc byte;
        return pinbuff(pinptr);
    end next5pin5char;

    checkfile: proc byte;
        do forever;
           if (pinptr := pinptr + 1) >= curpinrecsize then
              do;
                 pinptr = 0;
                 if read5pin5file = fileeof then
                    return true;
              end;
        nextchar = next5pin5char;
        return false;
        end;
```

```
    end checkfile;

    if checkfile or (nextchar = eoffiller)  then
       do;
          call move(.addeof, sbloc, 1);
          pinptr = 0;
          nextchar = nextSpinSchar;
       end;
    return nextchar;
end getSnextSbyte;


getSnextSaddr: proc addr;
    dcl (lowa,higha,tadd) addr;
    lowa = double(getSnextSbyte);
    higha = double(getSnextSbyte);
    tadd = shl(higha,8);
    tadd = tadd + lowa;
    return tadd;
end getSnextSaddr;


/*****************************************************************************/
/*****************************************************************************/
/***              g e n e r a l   p r o c e d u r e s                      ***/
/*****************************************************************************/
/*****************************************************************************/


/*******************************************/
/* popSsvSaddr removes the first two bytes */
/* from the stack and saves them in the    */
/* address indicated as a parameter.       */
/* total number of bytes generated = 7     */
/*******************************************/
popSsvSaddr: proc(a);
    dcl a addr;
    call generate(21h);         /* lxi */
    call generate(low(a));      /* storage place */
    call generate(high(a));
    call generate(0c1h);        /* pop b */
    call generate(71h);         /* movm c */
    call generate(23h);         /* inx h */
    call generate(70h);         /* movm b */
end popSsvSaddr;


/*******************************************/
/*  pushSsvSaddr returns the address from  */
/* the specified address to the stack.     */
/* total number of bytes generated = 7     */
/*******************************************/
pushSsvSaddr: proc(a);
    dcl a addr;
    call generate(21h);         /* lxi */
    call generate(low(a));      /* retrieving addr */
    call generate(high(a));
    call generate(4eh);         /* movc m */
    call generate(23h);         /* inx h */
    call generate(46h);         /* movb m */
    call generate(0c5h);        /* push b */
end pushSsvSaddr;


/*******************************************/
/* popSint     removes the first two bytes */
/* from the stack and saves them in the    */
/* address indicated as a parameter.       */
/* total number of bytes generated = 7     */
/*******************************************/
popSint: proc(a);
    dcl a addr;
```

```
    call generate(21h);      /* lxi h */
    call generate(low(a));   /* addr of int */
    call generate(high(a));
    call generate(0c1h);     /* pop b */
    call generate(71h);      /* movm c */
    call generate(23h);      /* inx h */
    call generate(70h);      /* movm b */
end pop$int;

/*********************************************/
/*  push$int     returns the integer from  */
/* the specified address to the stack.      */
/* total number of bytes generated = 7      */
/*********************************************/
push$int: proc(a);
    dcl a addr;
    call generate(21h);      /* lxi h */
    call generate(low(a));   /* addr of int */
    call generate(high(a));
    call generate(4eh);      /* movc m */
    call generate(23h);      /* inx h */
    call generate(46h);      /* movb m */
    call generate(0c5h);     /* push b */
end push$int;

/*********************************************/
/* pop$bcd removes the last 8 bytes from the */
/* stack and places them in the working area */
/* starting at address a which is passed as  */
/* a parameter by the user.                  */
/* total number of bytes generated = 23      */
/*********************************************/
pop$bcd: proc(a);
    dcl i byte, a addr;
    call generate(21h);              /* lxi  h */
    call generate(low(a));              /* addr of bcd */
    call generate(high(a));
    do i=1 to 4;
        call generate(0c1h);         /* pop b */
        call generate(71h);          /* movm c */
        call generate(23h);          /* inx h */
        call generate(70h);          /* movm b */
        call generate(23h);          /* inx h */
    end;
end pop$bcd;

/*********************************************/
/* push$bcd places the 8 bytes of a bcd num- */
/* ber, whose last array address a is passed */
/* as a parameter by the user, into the stack*/
/* total number of bytes generated = 23      */
/*********************************************/
push$bcd: proc(a);
    dcl i byte, a addr;
    call generate(21h);          /* lxi h */
    call generate(low(a));       /* addr of bcd */
    call generate(high(a));
    do i=1 to 4;
        call generate(46h);      /* movb m */
        call generate(2bh);      /* dcx h */
        call generate(4eh);      /* movc m */
        call generate(2bh);      /* dcx h */
        call generate(0c5h);     /* push b */
    end;
end push$bcd;

/*********************************************/
/* compl$bcd complements a bcd number located */
```

```
/* at the working area in location of x, the  */
/* address a passed as a parameter is the ad- */
/* dress of the first byte of the number array*/
/* total number of bytes generated = 43       */
/**********************************************/
compl8bcd: proc(a);
    dcl i byte, a addr;
    call generate(21h);          /* lxi h */
    call generate(low(a));       /* addr of bcd */
    call generate(high(a));
    call generate(3ah);          /* ldi */
    call generate(80h);          /* 10000000 */
    call generate(86h);          /* add m */
    call generate(77h);          /* movm a */
    call generate(23h);          /* inx h */
    do i=1 to 7;
       call generate(3ah);       /* lda */
       call generate(99h);       /* 99  */
       call generate(96h);       /* sub m */
       call generate(77h);       /* movm a */
       call generate(23h);       /* inx h */
    end;
end compl8bcd;


/**********************************************/
/* mult8int pops two integer numbers from the */
/* stack and multiplies them together . then  */
/* pushes the resulting integer back in the stack */
/* total number of bytes generated = 40       */
/**********************************************/
mult8int: proc(a);
    dcl a addr;
    call generate(0d1h);         /* pop d */
    call generate(0c1h);         /* pop b */
    call generate(0c3h);         /* jmp */
    call generate(low(a + 24h));
    call generate(high(a + 24h));
    call generate(79h);          /* mova c */
    call generate(93h);          /* sub e */
    call generate(78h);          /*mova b */
    call generate(9ah);          /* sbb d */
    call generate(0f2h);         /* jp */
    call generate(low(a + 11h));
    call generate(high(a + 11h));
    call generate(60h);          /* movh b */
    call generate(69h);          /* movl c */
    call generate(0ebh);         /* xchg */
    call generate(44h);          /* movb h */
    call generate(4dh);          /* movc l */
    call generate(21h);          /* lxi h */
    call generate(00h);
    call generate(00h);
    call generate(0ebh);         /* xchg */
    call generate(78h);          /* mova b */
    call generate(0b1h);         /* ora c */
    call generate(0c8h);         /* rz */
    call generate(0ebh);         /* xchg */
    call generate(78h);          /* mova b */
    call generate(1fh);          /* rar */
    call generate(47h);          /* movb a */
    call generate(79h);          /* mova c */
    call generate(1fh);          /* rar */
    call generate(4fh);          /* movc a */
    call generate(0c2h);         /* jnc */
    call generate(19h);          /* dad d */
    call generate(0ebh);         /* xchg */
    call generate(29h);          /* dad h */
    call generate(0c3h);         /* jmp */
    call generate(0f8h);         /* call */
    call generate(low(a + 3));
    call generate(high(a + 3));
```

```
    call generate(0d5h);        /* push d */
end mult3int;


/*********************************************/
/* div3int pops two integer numbers from */
/* the stack, divides the second number  */
/* removed by the first number removed   */
/* and returns the result to the stack   */
/* total number of bytes generated = 54  */
/*********************************************/
div3int: proc(a);
    dcl a addr;
    call gten(0d1h,0c1h,0c3h,low(a+50),high(a+50),7ah,2fh,57h,7bh,2fh);
    call gten(5fh,13h,21h,00h,00h,3eh,11h,0e5h,19h,0d2h);
    call generate( low(a + 23));
    call generate( high(a + 23));
    call gten(0e3h,0e1h,0f5h,79h,17h,4fh,78h,17h,47h,7dh);
    call gten(17h,5fh,7ch,17h,57h,0f1h,3dh,0c2h,low(a+17),high(a+17));
    call gten(0b7h,7ch,1fh,57h,7dh,1fh,5fh,0c9h,0cdh,low(a+5));
    call generate( high(a + 5));
    call generate(0c5h);
end div3int;


/*************************************************/
/* lt3int compares the next two integers in   */
/* the stack and returns a 1 to the stack     */
/* if the comparison is true or a 0 if the    */
/* comparison is false.                       */
/* total number of bytes generated = 23       */
/*************************************************/
lt3int: proc(a);
    dcl a addr;
    call generate(0d1h);        /* pop d */
    call generate(0c1h);        /* pop b */
    call generate(79h);         /* mova c */
    call generate(93h);         /* sub e */
    call generate(4fh);         /* movc a */
    call generate(78h);         /* mova b */
    call generate(9ah);         /* sbb d */
    call generate(0d2h);        /* jnc */
    call generate(low(a + 18));
    call generate(high(a + 18));
    call generate(0eh);         /* mvi c */
    call generate(01h);
    call generate(06h);         /* mvi b */
    call generate(00h);
    call generate(0c5h);        /* push b */
    call generate(0c3h);        /* jmp */
    call generate(low(a + 23));
    call generate(high(a + 23));
    call generate(0eh);         /* mvi c */
    call generate(00h);
    call generate(06h);         /* mvi b */
    call generate(00h);
    call generate(0c5h);        /* push b */
end lt3int;


/*************************************************/
/* le3int compares the next two integers in   */
/* the stack and returns a 1 to the stack     */
/* if the comparison is true or a 0 if the    */
/* comparison is false.                       */
/* total number of bytes generated = 23       */
/*************************************************/
le3int: proc(a);
    dcl a addr;
    call generate(0c1h);        /* pop b */
    call generate(0d1h);        /* pop d */
    call generate(79h);         /* mova c */
```

AD-A061 040 NAVAL POSTGRADUATE SCHOOL MONTEREY CALIF F/G 9/2
NPS-PASCAL. A PARTIAL IMPLEMENTATION OF PASCAL LANGUAGE FOR A M--ETC(U)
JUN 78 J C GRACIDA, R R STILWELL

UNCLASSIFIED NL

3 OF 3
AD
A061040

END
DATE
FILMED
1-79
DDC

```
    call generate(93h);       /* sub e */
    call generate(4fh);       /* movc a */
    call generate(78h);       /* mova b */
    call generate(9ah);       /* sbb d */
    call generate(0dah);       /* jc */
    call generate(low(a + 18));
    call generate(high(a + 18));
    call generate(0eh);       /* mvi c */
    call generate(01h);
    call generate(06h);       /* mvi b */
    call generate(00h);
    call generate(0c5h);      /* push b */
    call generate(0c3h);      /* jmp */
    call generate(low(a + 23));
    call generate(high(a + 23));
    call generate(0eh);       /* mvi c */
    call generate(00h);
    call generate(06h);       /* mvi b */
    call generate(00h);
    call generate(0c5h);      /* push b */
end le3int;

/*************************************************/
/* gt3int compares the next two integers in      */
/* the stack and returns a 1 to the stack        */
/* if the comparison is true or a 0 if the       */
/* comparison is false.                          */
/* total number of bytes generated = 23          */
/*************************************************/
gt3int: proc(a);
    dcl a addr;
    call generate(0c1h);      /* pop b */
    call generate(0d1h);      /* pop d */
    call generate(79h);       /* mova c */
    call generate(93h);       /* sub e */
    call generate(4fh);       /* movc a */
    call generate(78h);       /* mova b */
    call generate(9ah);       /* sbb d */
    call generate(0d2h);      /* jnc */
    call generate(low(a + 18));
    call generate(high(a + 18));
    call generate(0eh);       /* mvi c */
    call generate(01h);
    call generate(06h);       /* mvi b */
    call generate(00h);
    call generate(0c5h);      /* push b */
    call generate(0c3h);      /* jmp */
    call generate(low(a + 23));
    call generate(high(a + 23));
    call generate(0eh);       /* mvi c */
    call generate(00h);
    call generate(06h);       /* mvi b */
    call generate(00h);
    call generate(0c5h);      /* push b */
end gt3int;

/*************************************************/
/* ge3int compares the next two integers in      */
/* the stack and returns a 1 to the stack        */
/* if the comparison is true or a 0 if the       */
/* comparison is false.                          */
/* total number of bytes generated = 23          */
/*************************************************/
ge3int: proc(a);
    dcl a addr;
    call generate(0d1h);      /* pop d */
    call generate(0c1h);      /* pop b */
    call generate(79h);       /* mova c */
    call generate(93h);       /* sub e */
    call generate(4fh);       /* movc a */
```

```
    call generate(78h);       /* mova b */
    call generate(9ah);       /* sbb d */
    call generate(0dah);      /* jc */
    call generate(low(a + 18));
    call generate(high(a + 18));
    call generate(0eh);       /* mvi c */
    call generate(01h);
    call generate(06h);       /* mvi b */
    call generate(00h);
    call generate(0c5h);      /* push b */
    call generate(0c3h);      /* jmp */
    call generate(low(a + 23));
    call generate(high(a + 23));
    call generate(0eh);       /* mvi c */
    call generate(00h);
    call generate(06h);       /* mvi b */
    call generate(00h);
    call generate(0c5h);      /* push b */
end ge3int;


/*************************************************/
/* eq3int compares the next two integers in      */
/* the stack and returns a 1 to the stack        */
/* if the comparison is true or a 0 if the       */
/* comparison is false.                          */
/* total number of bytes generated = 24          */
/*************************************************/
eq3int: proc(a);
    dcl a addr;
    call generate(0d1h);      /* pop d */
    call generate(0c1h);      /* pop b */
    call generate(79h);       /* mova c */
    call generate(93h);       /* sub e */
    call generate(4fh);       /* movc a */
    call generate(78h);       /* mova b */
    call generate(9ah);       /* sbb d */
    call generate(0b1h);      /* ora c */
    call generate(0c2h);      /* jnz */
    call generate(low(a + 19));
    call generate(high(a + 19));
    call generate(0eh);       /* mvi c */
    call generate(01h);
    call generate(06h);       /* mvi b */
    call generate(00h);
    call generate(0c5h);      /* push b */
    call generate(0c3h);      /* jmp */
    call generate(low(a + 24));
    call generate(high(a + 24));
    call generate(0eh);       /* mvi c */
    call generate(00h);
    call generate(06h);       /* mvi b */
    call generate(00h);
    call generate(0c5h);      /* push b */
end eq3int;


/*************************************************/
/* ne3int compares the next two integers in      */
/* the stack and returns a 1 to the stack        */
/* if the comparison is true or a 0 if the       */
/* comparison is false.                          */
/* total number of bytes generated = 24          */
/*************************************************/
ne3int: proc(a);
    dcl a addr;
    call generate(0d1h);      /* pop d */
    call generate(0c1h);      /* pop b */
    call generate(79h);       /* mova c */
    call generate(93h);       /* sub e */
    call generate(4fh);       /* movc a */
    call generate(78h);       /* mova b */
```

```
    call generate(9ah);       /* sbb d */
    call generate(0b1h);      /* ora c */
    call generate(0c2h);      /* jnz */
    call generate(low(a + 19));
    call generate(high(a + 19));
    call generate(0eh);       /* mvi c */
    call generate(01h);
    call generate(06h);       /* mvi b */
    call generate(00h);
    call generate(0c5h);      /* push b */
    call generate(0c3h);      /* jmp */
    call generate(low(a + 24));
    call generate(high(a + 24));
    call generate(0eh);       /* mvi c */
    call generate(00h);
    call generate(06h);       /* mvi b */
    call generate(00h);
    call generate(0c5h);      /* push b */
end neSint;

/*********************************************/
/* notSbool negates a '0' to a '1' and a     */
/* '1' to a '0' taking the last byte of the */
/* stack and returning its complement to the*/
/* stack. total number of bytes = 19         */
/*********************************************/
notSbool: proc(a);
    dcl a addr;
    call generate(0c1h);      /* pop b */
    call generate(79h);       /* mova c */
    call generate(0fh);       /* rrc */
    call generate(0d2h);      /* jnc */
    call generate(low(a + 14));
    call generate(high(a + 14));
    call generate(0eh);       /* mvi c */
    call generate(00h);
    call generate(06h);       /* mvi b */
    call generate(00h);
    call generate(0c5h);      /* push b */
    call generate(0c3h);      /* jmp */
    call generate(low(a + 19));
    call generate(high(a + 19));
    call generate(0eh);       /* mvi c */
    call generate(00h);
    call generate(06h);       /* mvi b */
    call generate(00h);
    call generate(0c5h);      /* push b */
end notSbool;

/*********************************************/
/* andSbool pops the last two integers from */
/* the stack calculates their logical 'and' */
/* and returns the new value to the stack.  */
/* total number of bytes generated = 26      */
/*********************************************/
andSbool: proc(a);
    dcl a addr;
    call generate(0d1h);      /* pop d */
    call generate(0c1h);      /* pop b */
    call generate(79h);       /* mova c */
    call generate(0a3h);       /* ana e */
    call generate(4fh);       /* movc a */
    call generate(78h);       /* mova b */
    call generate(0a1h);      /* ana c */
    call generate(47h);       /* movb a */
    call generate(79h);       /* mova c */
    call generate(0fh);       /* rrc */
    call generate(0d2h);      /* jnc */
    call generate(low(a + 21));
    call generate(high(a + 21));
```

```
    call generate(0eh);       /* mvi c */
    call generate(01h);
    call generate(06h);       /* mvi b */
    call generate(00h);
    call generate(0c5h);      /* push b */
    call generate(0c3h);      /* jmp */
    call generate(low(a + 26));
    call generate(high(a + 26));
    call generate(0eh);       /* mvi c */
    call generate(00h);
    call generate(06h);       /* mvi b */
    call generate(00h);
    call generate(0c5h);      /* push b */
end andSbool;


/*********************************************/
/* orSbool pops the last two integers from   */
/* the stack calculates their logical 'or'   */
/* and returns the new value to the stack.   */
/* total number of bytes generated = 26      */
/*********************************************/
orSbool: proc(a);
    dcl a addr;
    call generate(0d1h);      /* pop d */
    call generate(0c1h);      /* pop b */
    call generate(79h);       /* mova c */
    call generate(0b3h);       /* ora e */
    call generate(4fh);       /* movc a */
    call generate(78h);       /* mova b */
    call generate(0b2h);      /* ora d */
    call generate(47h);       /* movb a */
    call generate(79h);       /* mova c */
    call generate(0fh);       /* rrc */
    call generate(0d2h);      /* jnc */
    call generate(low(a + 21));
    call generate(high(a + 21));
    call generate(0eh);       /* mvi c */
    call generate(01h);
    call generate(06h);       /* mvi b */
    call generate(00h);
    call generate(0c5h);      /* push b */
    call generate(0c3h);      /* jmp */
    call generate(low(a + 26));
    call generate(high(a + 26));
    call generate(0eh);       /* mvi c */
    call generate(00h);
    call generate(06h);       /* mvi b */
    call generate(00h);
    call generate(0c5h);      /* push b */
end orSbool;


/*********************************************/
/* svSstack increses the size of the stack */
/* by moving the stack pointer (b) times   */
/* total number of bytes generated = b     */
/*********************************************/
svSstack: proc(b);
    dcl (i,b) byte;
    do i=1 to b;
    call generate(3bh);    /* dcx sp */
    end;
end svSstack;


/*********************************************/
/* unsvSstack  decreases size of the stack */
/* by moving the stack pointer (b) times   */
/* total number of bytes generated = b     */
/*********************************************/
unsvSstack: proc(b);
```

```
    dcl (i,b) byte;
    do i=1 to b;
    call generate(33h);     /* inx sp */
    end;
end unsv0stack;


/*********************************************/
/* sto0bcd stores eight bytes from the       */
/* stack into an address calculated from     */
/* the first two bytes taken from the stack  */
/* total number of bytes generated = 21      */
/*********************************************/
sto0bcd: proc;
    dcl i byte;
    call generate(0e1h);      /* pop h */
    do i=1 to 4;
    call generate(0c1h);      /* pop b */
    call generate(71h);       /* movm c */
    call generate(23h);       /* inx h */
    call generate(70h);       /* movm b */
    call generate(23h);       /* inx h */
    end;
end sto0bcd;


/*********************************************/
/* lod0bcd removes the first two bytes from  */
/* the stack , calculates the address of     */
/* the bcd number and moves 8 bytes into     */
/* the stack. total bytes generated = 21     */
/*********************************************/
lod0bcd: proc;
    dcl i byte;
    call generate(0e1h);       /* pop h */
    do i=1 to 4;
        call generate(46h);    /* movbm */
        call generate(2bh);    /* dcx h */
        call generate(4eh);    /* movc m */
        call generate(2bh);    /* dcx h */
        call generate(0c5h);   /* push b */
    end;
end lod0bcd;


/*********************************************/
/* print0int prints to the console the       */
/* integer specified by the calling routine  */
/* total bytes generated = 570               */
/*********************************************/
print0int: proc(a);
    dcl a addr;
    call gen0five(0c3h,low(a+3dh),high(a+3dh),21h,0fh);
    call gen0five(01h,71h,2ch,73h,23h);
    call gten(72h,0c3h,05h,00h,0c9h,21h,12h,01h,71h,0eh);
    call gten(02h,5eh,16h,00h,0cdh,low(a+3h),high(a+3h),21h,3fh,01h);
    call gten(34h,3eh,4fh,96h,0d2h,low(a+3ch),high(a+3ch),0eh,02h,1eh);
    call gten(0dh,16h,00h,0cdh,low(a+03h),high(a+03h),0eh,02h,1eh,0ah);
    call gten(16h,00h,0cdh,low(a+03h),high(a+03h),21h,3fh,01h,36h,00h);
    call gten(0c9h,21h,0eh,01h,36h,00h,3eh,0ffh,06h,7fh);
    call gten(2eh,0eeh,96h,2ch,4fh,78h,9eh,0d2h,low(a+66h),high(a+66h));
    call gten(0eh,2dh,0cdh,low(a+0fh),high(a+0fh),0afh,21h,08h,01h,96h);
    call gten(2ch,4fh,3eh,00h,9eh,2dh,71h,23h,77h,0c3h);
    call gen0five(low(a+6bh),high(a+6bh),0eh,20h,0cdh);
    call gen0five(low(a+0fh),high(a+0fh),11h,10h,27h);
   call gten(21h,08h,01h,4eh,2ch,46h,0c3h,low(a+0a4h),high(a+0a4h),7ah);
    call gten(2fh,57h,7bh,2fh,5fh,13h,21h,00h,00h,3eh);
    call gen0five(11h,0e5h,19h,0d2h,low(a+89h));
    call gen0five(high(a+89h),0e3h,0e1h,0f5h,79h);
    call gten(17h,4fh,78h,17h,47h,7dh,17h,6fh,7ch,17h);
   call gten(67h,0f1h,3dh,0c2h,low(a+83h),high(a+83h),0b7h,7ch,1fh,57h);
   call gten(7dh,1fh,5fh,0c9h,0cdh,low(a+77h),high(a+77h),0afh,91h,5fh);
```

```
    call gten(3eh,00h,98h,0d2h,low(a+10fh),high(a+10fh),21h,0eh,01h,36h);
     call gten(01h,11h,10h,27h,21h,08h,01h,4eh,2ch,46h);
     call gten(0cdh,low(a+77h),high(a+77h),21h,0ah,01h,71h,11h,10h,27h);
    call gten(21h,0ah,01h,4eh,06h,00h,0c3h,low(a+0f4h),high(a+0f4h),79h);
   call gten(93h,78h,9ah,0f2h,low(a+0ddh),high(a+0ddh),60h,69h,0ebh,44h);
     call gten(4dh,21h,00h,00h,0ebh,78h,0b1h,0c8h,0ebh,78h);
   call gten(1fh,47h,79h,1fh,4fh,0d2h,low(a+0efh),high(a+0efh),19h,0ebh);
     call gen3five(29h,0c3h,low(a+0e1h),high(a+0e1h),0cdh);
     call gen3five(low(a+0d1h),high(a+0d1h),21h,08h,01h);
     call gten(7eh,2ch,46h,93h,4fh,78h,9ah,2dh,71h,23h);
     call gten(77h,2eh,0dh,7eh,0c6h,30h,77h,4eh,0cdh,low(a+0fh));
     call gten(high(a+0fh),11h,0e8h,03h,21h,08h,01h,4eh,2ch,0cdh);
    call gten(0cdh,low(a+77h),high(a+77h),0afh,91h,5fh,3eh,00h,98h,0d2h);
    call gten(low(a+160h),high(a+160h),21h,0eh,01h,36h,01h,11h,0e8h,03h);
     call gten(21h,08h,01h,4eh,2ch,46h,0cdh,low(a+77h),high(a+77h),21h);
     call gten(0dh,01h,71h,11h,0e8h,03h,21h,0dh,01h,04h);
    call gten(06h,00h,0cdh,low(a+0d1h),high(a+0d1h),21h,08h,01h,7eh,2ch);
     call gten(46h,93h,4fh,78h,9ah,2dh,71h,23h,77h,2eh);
    call gten(0dh,7eh,0c6h,30h,77h,4eh,0cdh,low(a+0fh),high(a+0fh),0c3h);
     call gen3five(low(a+16dh),high(a+16dh),21h,0eh,01h);
     call gen3five(7eh,0fh,0d2h,low(a+16dh),high(a+16dh));
     call gten(0eh,30h,0cdh,low(a+0fh),high(a+0fh),1eh,64h,16h,00h,21h);
     call gten(08h,01h,4eh,2ch,46h,0cdh,low(a+77h),high(a+77h),0afh,91h);
    call gten(5fh,3eh,00h,98h,0d2h,low(a+1c1h),high(a+1c1h),21h,0eh,01h);
     call gten(36h,01h,1eh,64h,16h,00h,21h,08h,01h,4eh);
     call gten(2ch,46h,0cdh,low(a+77h),high(a+77h),21h,0dh,01h,71h,1eh);
     call gten(64h,16h,00h,21h,0dh,01h,4eh,06h,00h,0cdh);
     call gten(low(a+0d1h),high(a+0d1h),21h,08h,01h,7eh,2ch,46h,93h,4fh);
     call gten(78h,9ah,2dh,71h,23h,77h,2eh,0dh,7eh,0c6h);
     call gen3five(30h,77h,4eh,0cdh,low(a+0fh));
     call gen3five(high(a+0fh),0c3h,low(a+1ceh),high(a+1ceh),21h);
   call gten(0eh,01h,7eh,0fh,0d2h,low(a+1ceh),high(a+1ceh),0eh,30h,0cdh);
     call gten(low(a+0fh),high(a+0fh),1eh,0ah,16h,00h,21h,08h,01h,4eh);
     call gten(2ch,46h,0cdh,low(a+77h),high(a+77h),0afh,91h,5fh,3eh,00h);
    call gten(98h,0d2h,low(a+21dh),high(a+21dh),1eh,0ah,16h,00h,21h,08h);
     call gten(01h,4eh,2ch,46h,0cdh,low(a+77h),high(a+77h),21h,0dh,01h);
     call gten(71h,1eh,0ah,16h,00h,21h,0dh,01h,4eh,06h);
    call gten(00h,0cdh,low(a+0d1h),high(a+0d1h),21h,08h,01h,7eh,2ch,46h);
     call gten(93h,4fh,78h,9ah,2dh,71h,23h,77h,2eh,0dh);
    call gten(7eh,0c6h,30h,77h,4eh,0cdh,low(a+0fh),high(a+0fh),0c3h,00h);
   call gten(00h,21h,0deh,01h,7eh,0fh,0d2h,low(a+22ah),high(a+22ah),0eh);
     call gten(30h,0cdh,low(a+0fh),high(a+0fh),01h,30h,09h,2ah,08h,01h);
     call gten(09h,0ebh,21h,0ah,01h,73h,4eh,0cdh,low(a+0fh),high(a+0fh));
end print3int;


/*********************************************/
/* print3bcd prints to the console a bcd     */
/* number moved to the working area by the   */
/* calling routine.                          */
/* total number of bytes generated = 464     */
/*********************************************/
print3bcd: proc(a);
    dcl a addr;
    call gten(0c3h,low(a+3dh),high(a+3dh),21h,13h,01h,71h,2ch,73h,23h);
    call gten(72h,0c3h,05h,00h,0c9h,21h,16h,01h,71h,0eh);
    call gten(02h,5eh,16h,00h,0cdh,low(a+3h),high(a+3h),21h,3fh,01h);
    call gten(34h,3eh,4fh,96h,0d2h,low(a+3ch),high(a+3ch),0eh,02h,1eh);
    call gten(0dh,16h,00h,0cdh,low(a+3h),high(a+3h),0eh,02h,1eh,0ah);
    call gten(16h,00h,0cdh,low(a+3h),high(a+3h),21h,3fh,01h,36h,00h);
    call gten(0c9h,3eh,3ch,21h,3fh,01h,96h,0d2h,low(a+5dh),high(a+5dh));
    call gten(0eh,02h,1eh,9dh,16h,00h,0cdh,low(a+3h),high(a+3h),0eh);
    call gten(02h,1eh,0ah,16h,00h,0cdh,low(a+3h),high(a+3h),21h,3fh);
    call gten(01h,36h,00h,2eh,0fah,36h,07h,3eh,7fh,2eh);
    call gen3five(08h,96h,0d2h,low(a+78h),high(a+78h));
    call gen3five(0eh,2dh,0cdh,low(a+0fh),high(a+0fh));
    call gten(21h,08h,01h,7eh,0e6h,80h,77h,0c3h,low(a+7dh),high(a+7dh));
    call gten(0eh,20h,0cdh,low(a+0fh),high(a+0fh),21h,12h,01h,4eh,06h);
    call gten(00h,2eh,08h,09h,7eh,1eh,04h,0b7h,1fh,1dh);
    call gten(0c2h,low(a+89h),high(a+89h),0c6h,30h,21h,10h,01h,77h,4eh);
    call gten(0cdh,low(a+0fh),high(a+0fh),0eh,2eh,0cdh,
    low(a+0fh),high(a+0fh),21h,11h);
```

```
    call gten(01h,36h,01h,3eh,06h,21h,11h,01h,96h,0dah);
   call gten(low(a+0e4h),high(a+0e4h),02ch,4eh,06h,00h,2eh,08h,09h,7eh);
    call gten(0e6h,0fh,0c6h,30h,21h,10h,01h,77h,4eh,0cdh);
    call gten(low(a+0fh),high(a+0fh),21h,12h,01h,35h,4eh,06h,00h,2eh);
    call gten(08h,09h,7eh,1eh,04h,0b7h,1fh,1dh,0c2h,low(a+0cdh));
   call gten(high(a+0cdh),0c6h,30h,21h,10h,01h,77h,4eh,0cdh,low(a+0fh));
    call gten(high(a+0fh),21h,11h,01h,34h,0c2h,low(a+0a3h),high(a+0a3h),
    2eh,08h);
    call gten(23h,7eh,0e6h,0fh,0c6h,30h,21h,10h,01h,77h);
    call gten(4eh,0cdh,low(a+0fh),high(a+0fh),0fh,45h,0cdh,
    low(a+0fh),high(a+0fh),21h);
  call gten(08h,01h,7eh,0d6h,89h,0d2h,low(a+111h),high(a+111h),0eh,2dh);
    call gten(0cdh,low(a+0fh),high(a+0fh),3eh,41h,21h,08h,01h,96h,77h);
    call gten(0c3h,low(a+11dh),high(a+11dh),0eh,20h,
    0cdh,low(a+0fh),high(a+0fh),21h,08h);
    call gten(01h,7eh,0d6h,41h,77h,1eh,0ah,16h,00h,21h);
   call gten(08h,01h,4eh,06h,00h,0c3h,low(a+157h),high(a+157h),7ah,2fh);
    call gten(57h,7bh,2fh,5fh,13h,21h,00h,00h,3eh,11h);
    call gten(0e5h,19h,0d2h,low(a+13ch),high(a+13ch),0e3h,0e1h,
    0e5h,79h,17h);
    call gten(4fh,78h,17h,47h,7dh,17h,6fh,7ch,17h,67h);
    call gten(0f1h,3dh,0c2h,low(a+136h),high(a+136h),0b7h,7ch,
    1fh,57h,7dh);
    call gten(1fh,5fh,0c9h,0cdh,low(a+12ah),high(a+12ah),0afh,
    91h,5fh,3eh);
   call gten(00h,98h,0d2h,low(a+1beh),high(a+1beh),1eh,0ah,16h,00h,21h);
    call gten(08h,01h,4eh,06h,00h,0cdh,low(a+12ah),
    high(a+12ah),21h,low(a+1f3h));
    call gten(high(a+1f3h),71h,1eh,0ah,16h,00h,21h,10h,01h,4eh);
  call gten(06h,00h,0c3h,low(a+1a4h),high(a+1a4h),79h,93h,78h,9ah,0f2h);
   call gten(low(a+18dh),high(a+18dh),60h,69h,0ebh,44h,4dh,21h,00h,00h);
    call gten(0ebh,78h,0b1h,0c8h,0ebh,78h,1fh,47h,79h,1fh);
    call gten(4fh,0d2h,low(a+19fh),high(a+19fh),19h,0ebh,29h,
    0c3h,low(a+191h),high(a+191h));
   call gten(0cdh,low(a+181h),high(a+181h),21h,08h,01h,7eh,93h,4fh,3eh);
    call gten(00h,9ah,71h,2eh,10h,7eh,0c6h,30h,77h,4eh);
  call gten(0cdh,low(a+0fh),high(a+0fh),0c3h,low(a+1c3h),high(a+1c3h),
    0eh,20h,0cdh,low(a+0fh));
    call gten(high(a+0fh),21h,08h,01h,7eh,0c6h,30h,2eh,10h,77h);
    call generate(4eh);
    call generate(0cdh);
    call generate(low(a+0fh));
    call generate(high(a+0fh));
end print3bcd;


/***********************************************/
/*  ne3bcd logically compares two bcd numbers */
/* taken from the stack returns a value of one*/
/* if the numbers are not equal, a zero if     */
/* equal. total number of bytes gen = 67       */
/***********************************************/
ne3bcd: proc(a);
    dcl a addr;
    call gten(21h,1ah,11h,36h,00h,2dh,36h,00h,21h,19h);
    call gten(11h,7eh,0d6h,08h,0d2h,low(a+3eh),high(a+3eh),4eh,06h,00h);
    call gten(2eh,08h,09h,7eh,21h,19h,11h,5eh,16h,00h);
    call gten(2eh,10h,19h,4fh,7eh,91h,0cah,low(a+34h),high(a+34h),21h);
    call gten(1ah,11h,36h,01h,2dh,7eh,0c6h,08h,77h,0c3h);
    call gten(low(a+8h),high(a+8h),21h,18h,11h,36h,00h,2dh,34h,0c3h);
    call gen3five(low(a+8h),high(a+8h),2ch,4eh,06h);
    call generate(00h);
    call generate(0c5h);
end ne3bcd;


/***********************************************/
/* eq3bcd logically compares two bcd numbers  */
/* taken from the stack returns a value of one*/
/* if the numbers are equal, a zero if not.   */
/*total number of bytes generated = 66         */
/***********************************************/
```

```
eq3bcd: proc(a);
    dcl a addr;
    call gten(21h,1ah,11h,36h,00h,2dh,36h,00h,21h,19h);
    call gten(7eh,0d6h,08h,0d2h,low(a+3eh),high(a+3eh),4eh,06h,00h,2eh);
    call gten(08h,09h,7eh,21h,19h,11h,5eh,16h,00h,2eh);
    call gten(10h,19h,4fh,7eh,91h,0c2h,low(a+31h),high(a+31h),21h,1ah);
    call gten(11h,36h,01h,2dh,34h,0c3h,low(a+8h),high(a+8h),21h,1ah);
    call gten(11h,36h,00h,2dh,7eh,0c6h,08h,77h,0c3h,low(a+8h));
    call gen3five(high(a+8h),2ch,4eh,06h,00h);
    call generate(0c5h);
end eq3bcd;


/*************************************/
/* ge3bcd logically compares two bcd */
/* numbers taken from the stack and  */
/* returns a value of one if the 1st */
/* number is greater or equal to the */
/* second number. bytes generated=324*/
/*************************************/
ge3bcd: proc(a);
    dcl a addr;
    call gten(0c3h,low(a+0ech),high(a+0ech),21h,1eh,01h,
    36h,01h,2eh,1ch);
    call gten(7eh,0fh,0d2h,low(a+80h),high(a+80h),2eh,19h,7eh,2ch,96h);
    call gten(0c2h,low(a+67h),high(a+67h),21h,1eh,01h,7eh,0d6h,
    08h,0d2h);
    call gten(low(a+72h),high(a+72h),4eh,06h,00h,2eh,08h,09h,7eh,21h);
    call gten(1eh,01h,5eh,16h,00h,2eh,10h,19h,4fh,7eh);
    call gten(91h,0c2h,low(a+3dh),high(a+3dh),21h,1eh,01h,34h,
    0c3h,low(a+17h));
    call gten(high(a+17h),21h,1eh,01h,4eh,06h,00h,2eh,08h,09h);
     call gten(7eh,21h,1eh,01h,5eh,16h,00h,2eh,10h,19h);
    call gten(4fh,7eh,5fh,79h,93h,0d2h,low(a+5dh),high(a+5dh),21h,1dh);
    call gten(01h,36h,01h,21h,1eh,01h,7eh,0c6h,08h,77h);
    call gten(0c3h,low(a+17h),high(a+17h),2dh,7eh,2ch,96h,0d2h,
    low(a+72h),high(a+72h));
    call gten(2eh,1dh,36h,01h,2eh,1eh,7eh,0d6h,08h,0c2h);
    call gten(low(a+0ebh),high(a+0ebh),2dh,36h,01h,0c3h,low(a+0ebh),
    high(a+0ebh),2eh,19h);
   call gten(7eh,2ch,96h,0c2h,low(a+0d6h),high(a+0d6h),21h,1eh,01h,7eh);
    call gten(0d6h,08h,0d2h,low(a+0e0h),high(a+0e0h),4eh,06h,00h,
    2eh,08h);
    call gten(09h,7eh,21h,1eh,01h,5eh,16h,00h,2eh,10h);
    call gten(19h,4fh,7eh,91h,0c2h,low(a+0aeh),high(a+0aeh),21h,
    1eh,01h);
    call gten(34h,0c3h,low(a+88h),high(a+88h),21h,1eh,01h,4eh,06h,00h);
    call gten(2eh,08h,09h,7eh,21h,1eh,01h,5eh,16h,00h);
    call gten(2eh,10h,19h,4fh,7eh,91h,0d2h,low(a+0cch),high(a+0cch),
    21h);
    call gten(1dh,01h,36h,01h,21h,1eh,01h,7eh,0c6h,08h);
    call gten(77h,0c3h,low(a+88h),high(a+88h),7eh,2dh,96h,0d2h,
    low(a+0e0h),high(a+0e0h));
    call gten(2eh,1dh,36h,01h,2eh,1eh,7eh,0d6h,08h,0c2h);
   call gten(low(a+0ebh),high(a+0ebh),2dh,36h,01h,0c9h,21h,1dh,01h,36h);
    call gten(00h,2eh,1bh,36h,00h,2ch,36h,00h,2eh,08h);
    call gten(7eh,0e6h,79h,2eh,19h,77h,2eh,10h,7eh,0e6h);
    call gten(79h,2eh,1ah,77h,0afh,0d6h,80h,9fh,2eh,08h);
    call gten(0a6h,0fh,0d2h,low(a+117h),high(a+117h),2eh,1bh,36h,01h,
    0afh);
    call gten(0d6h,80h,9fh,2eh,10h,0a6h,0fh,0d2h,low(a+126h),
    high(a+126h));
    call gten(2eh,1ch,36h,01h,2eh,1bh,7eh,2ch,0a6h,0fh);
    call gten(0d2h,low(a+135h),high(a+135h),0cdh,low(a+3h),high(a+3h),
    0c3h,low(a+13dh),high(a+13dh),7eh);
   call gten(0fh,0d2h,low(a+13dh),high(a+13dh),2ch,36h,01h,21h,1dh,01h);
    call generate(4eh);
    call generate(06h);
    call generate(00h);
    call generate(0c5h);
end ge3bcd;
```

```
/*****************************************/
/* write$string calls the routine that   */
/* prints the characters followed by the */
/* opcode. total no. bytes gen = 61      */
/*****************************************/
write$strng: proc;
    dcl a addr;
    a = cspaddr(63);
    call gten(0c3h,low(a+2eh),high(a+2eh),21h,08h,01h,71h,2ch,73h,23h);
    call gten(72h,0c3h,05h,00h,0c9h,21h,0ch,01h,71h,0eh);
    call gten(02h,5eh,16h,00h,0cdh,low(a-12),high(a-12),21h,3fh,01h);
    call gten(34h,3eh,4fh,96h,0d2h,low(a+2dh),high(a+2dh),0eh,02h,1eh);
    call gten(0dh,16h,00h,0cdh,low(a-12),high(a-12),0eh,02h,1eh,0ah);
    call gten(16h,00h,0cdh,low(a-12),high(a-12),21h,3fh,01h,36h,00h);
    call generate(0c9h);
end write$strng;


/*****************************************/
/* convert$int removes the first two     */
/* bytes from the stack changes them to  */
/* a bcd number and returns 8 bytes to   */
/* the stack. total bytes = 383          */
/*****************************************/
convert$int: proc(a);
    dcl a addr;
    call gten(21h,17h,01h,36h,01h,3eh,07h,21h,17h,01h);
    call gten(96h,0dah,low(a+1dh),high(a+1dh),4eh,06h,00h,2eh,0ah,09h);
    call gten(36h,00h,21h,17h,01h,34h,0c2h,low(a+5h),high(a+5h),36h);
    call gten(00h,3eh,04h,21h,17h,01h,96h,0dah,low(a+37h),high(a+37h));
    call gten(4eh,06h,00h,2eh,12h,09h,36h,00h,21h,17h);
    call gten(01h,34h,0c2h,low(a+1fh),high(a+1fh),2ch,36h,00h,3eh,0ffh);
    call gten(06h,7fh,2eh,08h,96h,2ch,4fh,78h,9eh,0d2h);
    call gten(low(a+59h),high(a+59h),2eh,18h,36h,01h,0afh,2eh,08h,96h);
    call gten(2ch,4fh,3eh,00h,9eh,2dh,71h,23h,77h,2eh);
    call gten(1ah,36h,10h,23h,36h,27h,2eh,19h,36h,01h);
    call gten(2eh,0ah,36h,45h,21h,19h,01h,7eh,0fh,0d2h);
    call gten(low(a+0cfh),high(a+0cfh),2eh,08h,7ch,2ch,46h,2eh,1ah,96h);
    call gten(2ch,4fh,78h,9eh,0dah,low(a+86h),high(a+86h),2eh,19h,36h);
    call gten(00h,0c3h,low(a+68h),high(a+68h),1eh,0ah,16h,00h,21h,1ah);
   call gten(01h,4eh,2ch,46h,0c3h,low(a+0c0h),high(a+0c0h),7ah,2fh,57h);
    call gten(7bh,2fh,5fh,13h,21h,00h,00h,3eh,11h,0e5h);
    call gten(19h,0d2h,low(a+0a5h),high(a+0a5h),0e3h,0e1h,0f5h,79h,
    17h,4fh);
    call gten(78h,17h,47h,7dh,17h,6fh,7ch,17h,67h,0f1h);
    call gten(3dh,0c2h,low(a+9fh),high(a+9fh),0b7h,7ch,1fh,57h,7dh,1fh);
    call gten(5fh,0c9h,0cdh,low(a+93h),high(a+93h),21h,1ah,01h,71h,23h);
    call gten(70h,2eh,0ah,35h,0c3h,low(a+68h),high(a+68h),2dh,7eh,0fh);
  call gten(0d2h,low(a+0dbh),high(a+0dbh),2eh,0ah,7eh,0c6h,80h,77h,2eh);
    call gten(17h,36h,00h,0afh,21h,1ah,01h,96h,2ch,4fh);
   call gten(3eh,00h,9eh,0d2h,low(a+13bh),high(a+13bh),2eh,17h,4eh,06h);
    call gten(00h,2eh,12h,09h,0ebh,21h,0eah,03h,73h,2ch);
    call gten(72h,21h,1ah,01h,5eh,2ch,56h,2eh,08h,4eh);
    call gten(2ch,46h,0cdh,low(a+93h),high(a+93h),2ah,0eah,03h,71h,21h);
    call gten(17h,01h,34h,21h,1ah,01h,5eh,2ch,56h,2eh);
    call gten(08h,4eh,2ch,46h,0cdh,low(a+93h),high(a+93h),21h,08h,01h);
    call gten(73h,23h,72h,1eh,0ah,16h,00h,21h,1ah,01h);
    call gten(4eh,2ch,46h,0cdh,low(a+93h),high(a+93h),21h,1ah,01h,71h);
   call gten(23h,70h,0c3h,low(a+0dfh),high(a+0dfh),01h,07h,00h,2eh,0ah);
    call gten(09h,0ebh,21h,12h,01h,7eh,87h,87h,87h,87h);
    call gten(0d5h,4fh,0c5h,23h,7eh,0d1h,83h,0e1h,77h,01h);
    call gten(06h,00h,21h,0ah,01h,09h,0e5h,01h,02h,00h);
    call gten(21h,12h,01h,09h,7eh,87h,87h,87h,87h,11h);
    call gten(03h,00h,21h,12h,01h,19h,4fh,7eh,81h,0e1h);
    call gten(77h,01h,05h,00h,21h,0ah,01h,09h,0e5h,01h);
    call gten(04h,00h,21h,12h,01h,09h,7eh,87h,87h,87h);
    call generate(87h);
    call generate(0e1h);
    call generate(77h);
end convert$int;
```

```
/*****************************************/
/* convert8bcd pops the next eight bytes */
/* from the stack, converts the bcd num  */
/* to an integer number and returns it to*/
/* the stack. bytes generated = 313      */
/*****************************************/
convert8bcd: proc(a);
    dcl a addr;
   call gten(0c3h,low(a+1fh),high(a+1fh),21h,15h,01h,71h,2ch,73h,23h);
   call gten(72h,0c3h,05h,00h,0c9h,21h,17h,01h,71h,23h);
   call gten(70h,0eh,09h,2dh,5eh,2ch,56h,0cdh,low(a+3h),high(a+3h));
   call gten(0c9h,21h,12h,01h,36h,00h,2eh,10h,36h,00h);
   call gten(23h,36h,00h,2eh,08h,7eh,0e6h,7fh,4fh,3eh);
   call gten(45h,91h,0d2h,low(a+4dh),high(a+4dh),0c3h,low(a+44h),
        high(a+44h),45h,52h);
   call gten(52h,4fh,52h,20h,49h,4fh,20h,24h,01h,3ah);
   call gten(02h,0cdh,low(a+0fh),high(a+0fh),0c3h,low(a+139h),
        high(a+139h),7eh,0e6h,80h);
  call gten(0d6h,80h,0c2h,low(a+59h),high(a+59h),2eh,12h,36h,01h,2eh);
   call gten(08h,7eh,0e6h,7fh,0d6h,40h,2eh,13h,77h,3eh);
   call gten(7fh,96h,0d2h,low(a+6bh),high(a+6bh),36h,00h,2ch,36h,07h);
   call gten(0afh,21h,13h,01h,96h,0d2h,low(a+103h),high(a+103h),
        2ch,4eh);
   call gten(06h,00h,2eh,08h,09h,7eh,1eh,04h,0b7h,1fh);
   call gten(1dh,0c2h,low(a+80h),high(a+80h),06h,00h,4fh,21h,18h,01h);
   call gten(71h,2ch,70h,1eh,0ah,16h,00h,21h,10h,01h);
   call gten(4eh,2ch,46h,0c3h,low(a+0bfh),high(a+0bfh),79h,93h,
        78h,9ah);
   call gten(0f2h,low(a+0a8h),high(a+0a8h),60h,69h,0ebh,44h,4dh,
        21h,00h);
   call gten(00h,0ebh,78h,0b1h,0c8h,0ebh,78h,1fh,47h,79h);
   call gten(1fh,4fh,0d2h,low(a+0bah),high(a+0bah),19h,0ebh,29h,
        0c3h,low(a+0ach));
   call gten(high(a+0ach),0cdh,low(a+9ch),high(a+9ch),21h,18h,01h,
        4eh,2ch,46h);
   call gten(0ebh,09h,22h,10h,01h,21h,0f9h,03h,35h,0afh);
   call gten(96h,0d2h,low(a+6eh),high(a+6eh),1eh,0ah,16h,00h,21h,10h);
   call gten(01h,4eh,2ch,46h,0cdh,low(a+9ch),high(a+9ch),0d5h,
        21h,14h);
   call gten(01h,4eh,06h,00h,2eh,08h,09h,7eh,0e6h,0fh);
   call gten(06h,00h,4fh,0d1h,69h,60h,19h,22h,10h,01h);
   call gten(21h,14h,01h,35h,2dh,35h,0c3h,low(a+6eh),high(a+6eh),3eh);
   call gten(0ffh,06h,7fh,2eh,10h,96h,2ch,4fh,78h,9eh);
   call gten(0d2h,low(a+124h),high(a+124h),0c3h,low(a+11eh),
        high(a+11eh),45h,52h,52h,4fh);
   call gten(52h,20h,49h,4fh,20h,24h,01h,low(a+114h),high(a+114h),
        0cdh);
   call gten(low(a+0fh),high(a+0fh),21h,12h,01h,7eh,0fh,0d2h,
        low(a+139h),high(a+139h));
   call gten(0afh,2eh,10h,96h,2ch,4fh,3eh,00h,9eh,2dh);
   call generate(71h);
   call generate(23h);
   call generate(77h);
end convert8bcd;


/*************************************/
/* dump generates code for a carriage */
/* return and line feed character to  */
/* the screen to start a new line.    */
/*  total number of bytes generated=38*/
/*************************************/
dump: proc;
    dcl a addr;
    a = cspaddr(64);
   call gten(0c3h,low(a+0fh),high(a+0fh),21h,08h,01h,71h,2ch,73h,23h);
   call gten(72h,0c3h,05h,00h,0c9h,0eh,02h,1eh,0dh,16h);
   call gten(00h,0cdh,low(a+3h),high(a+3h),0eh,02h,1eh,0ah,16h,00h);
   call gen5five(0cdh,low(a+3h),high(a+3h),21h,3fh);
   call generate(01h);
   call generate(36h);
   call generate(00h);
```

```
end dump;


/*****************************************************************************/
/*****************************************************************************/
/***        p r o c e d u r e s   c a l l e d   b y   c a s e   s t m t ***/
/*****************************************************************************/
/*****************************************************************************/

set$flags: proc;
    dcl i byte;
    do i=1 to 65;
       cspc(i)= true;
    end;
end set$flags;


endp1: proc;
    pass1= false;
    pass2= true;
    call set$flags;
    call setup$com$file;
    call rewind$pin$file;
    call open$pin$file;
    codestrt = varstrt + varcount;
    prog$size = codestrt + codecount;
    pinptr = pinrecsize;
    sp$max = max - 2;
    call generate(31h);      /* lxi sp */
    call generate(low(sp$max));     /* max stack ptr */
    call generate(high(sp$max));
    call generate(0c3h);     /* jmp to code area */
    call generate(low(codestrt));
    call generate(high(codestrt));
    do loop=1 to (varcount + 58);      /* initialize work area */
       call generate(00h);
    end;
end endp1;


endprog: proc;
    pass2 = false;
    call generate(0fbh);     /* ei */
    call generate(76h);      /* hlt */
    call generate(76h);      /* hlt */
    call delete$pin$file;
    call write$com$file;
    call close$com$file;
    if errcount = 0 then do;
        call print(.'end of compilation. no program errors.$');
        call printchar(0dh);           /* cr */
        call printchar(0ah);           /* lf */
        end;
    else call print(.' compilation terminated due to error(s). $');
    call printchar(0dh);
    call printchar(0ah);
    goto boot;
end endprog;


sec$pass: proc(a);
    dcl a addr;
    call generate(0cdh);       /* call */
    call generate(low(a));
    call generate(high(a));
end sec$pass;


lb12: proc;
    dcl (lblnumber,lbladdr, n based lbladdr) addr;
    lblnumber = get$next$addr;
    lbladdr = .memory + (2* lblnumber);
```

```
    n = codecount;
end lb12;


ldib3: proc;
    dcl i byte, a addr;
    do i=1 to 4;
    a = get$next$addr;
    call gen3five(0eh,low(a),06h,high(a),0c1h);
    end;
end ldib3;


ldii4: proc;
    dcl a addr;
    a = get$next$addr;
    call gen3five(0eh,low(a),06h,high(a),0c1h);
end ldii4;


cnvb9: proc;
    dcl a addr;
    call generate(0c3h);        /* jmp */
    call generate(low(cspaddr(9) + 358));
    call generate(high(cspaddr(9) + 358));
    call pop$sv$addr(106h);                 /* 7 bytes */
    call pop$bcd(108h);                     /* 23 bytes */
    a = cspaddr(9) + 30;
    call convert$bcd(a);                    /* 313 bytes */
    call push$int(110h);                    /* 7 bytes */
    call push$sv$addr(106h);                /* 7 bytes */
    call generate(0c9h);        /* ret */
    call generate(0cdh);        /* call */
    call generate(low(cspaddr(9)));
    call generate(high(cspaddr(9)));
end cnvb9;


cnvi10: proc;
    dcl a addr;
    call generate(0c3h);        /* jmp */
    call generate(low(cspaddr(10) + 428));
    call generate(high(cspaddr(10) + 428));
    call pop$sv$addr(106h);                 /* 7 bytes */
    call pop$int(108h);                     /* 7 bytes */
    a = cspaddr(10) + 14;
    call convert$int(a);                    /* 383 bytes */
    call push$bcd(10ah);                    /* 23 bytes */
    call push$sv$addr(106h);                /* 7 bytes */
    call generate(0c9h);        /* ret */
    call generate(0cdh);        /* call */
    call generate(low(cspaddr(10)));
    call generate(high(cspaddr(10)));
end cnvi10;


lita12: proc;
    dcl taddr addr;
    taddr = get$next$addr + varstrt;
    call generate(01h);     /* lxi b */
    call generate(low(taddr));
    call generate(high(taddr));
    call generate(0c5h);    /* push b */
end lita12;


addb13: proc;
    ;
end addb13;


addi14: proc;
```

```
    call generate(0c1h);          /* pop b */
    call generate(0e1h);          /* pop h */
    call generate(09h);           /* dad b */
    call generate(0e5h);        /* push h */
end addi14;


subb15: proc;
    ;
end subb15;


subi16: proc;
    call generate(0c1h);          /* pop b */
    call generate(0d1h);          /* pop d */
    call generate(79h);           /* mova c */
    call generate(93h);           /* sub e */
    call generate(4bh);           /* movc e */
    call generate(78h);           /* mova b */
    call generate(9ah);           /* sbb b */
    call generate(47h);           /* movb a */
    call generate(0c5h);          /* push b */
end subi16;


mulb17: proc;
    ;
end mulb17;


muli18: proc;
    dcl a  addr;
    call generate(0c3h);          /* jmp */
    call generate(low(cspaddr(18) + 55));
    call generate(high(cspaddr(18) + 55));
    call popCsvCaddr(106h);           /* 7 bytes */
    a = cspaddr(18) + 7;
    call multCint(a);            /*40 bytes */
    call pushCsvCaddr(106h);                /* 7 bytes */
    call generate(0c9h);      /* rtn */
    call generate(0cdh);      /* call */
    call generate(cspaddr(18));
    call generate(cspaddr(18));
end muli18;


divb19: proc;
    ;
end divb19;


divi20: proc;
    dcl a  addr;
    call generate(0c3h);          /* jmp */
    call generate(low(cspaddr(20) + 69));
    call generate(high(cspaddr(20) + 69));
    call popCsvCaddr(106h);             /* 7 bytes */
    a = cspaddr(20) + 7;
    call divCint(a);             /* 54 bytes */
    call pushCsvCaddr(106h);               /* 7 bytes */
    call generate(0c9h);      /* rtn */
    call generate(0cdh);      /* call */
    call generate(cspaddr(20));
    call generate(cspaddr(20));
end divi20;


lssb21: proc;
    ;
end lssb21;
```

```
lssi22: proc;
    dcl a addr;
    call generate(0c3h);      /* jmp */
    call generate(low(cspaddr(22) + 38));
    call generate(high(cspaddr(22) + 38));
    call pop$sv$addr(106h);       /*  7 bytes */
    a = cspaddr(22) + 7;
    call lt$int(a);       /* 23 bytes */
    call push$sv$addr(106h);        /* 7 bytes */
    call generate(0c9h);      /* ret */
    call generate(0cdh);      /* call */
    call generate(low(cspaddr(22)));
    call generate(high(cspaddr(22)));
end lssi22;


leqb23: proc;
    ;
end leqb23;


leqi24: proc;
    dcl a addr;
    call generate(0c3h);        /* jmp */
    call generate(low(cspaddr(24) + 38));
    call generate(high(cspaddr(24) + 38));
    call pop$sv$addr(106h);     /* 7 bytes */
    a = cspaddr(24) + 7;
    call le$int(a);             /* 23 bytes */
    call push$sv$addr(106h);      /* 7 bytes */
    call generate(0c9h);       /* ret */
    call generate(0cdh);         /* call */
    call generate(low(cspaddr(24)));
    call generate(high(cspaddr(24)));
end leqi24;

eqlb25: proc;
    dcl a addr;
    call generate(0c3h);        /* jmp */
    call generate(low(cspaddr(25) + 81));
    call generate(high(cspaddr(25) + 81));
    call pop$sv$addr(106h);     /* 7 bytes */
    a = cspaddr(25) + 7;
    call eq$bcd(a);             /* 66 bytes */
    call push$sv$addr(106h);      /* 7 bytes */
    call generate(0c9h);       /* ret */
    call generate(0cdh);         /* call */
    call generate(low(cspaddr(25)));
    call generate(high(cspaddr(25)));
    ;
end eqlb25;


eqli26: proc;
    dcl a addr;
    call generate(0c3h);        /* jmp */
    call generate(low(cspaddr(26) + 39));
    call generate(high(cspaddr(26) + 39));
    call pop$sv$addr(106h);      /* 7 bytes */
    a = cspaddr(26) + 7;
    call eq$int(a);            /* 24 bytes */
    call push$sv$addr(106h);    /*  7 bytes */
    call generate(0c9h);     /* ret */
    call generate(0cdh);       /* call */
    call generate(low(cspaddr(26)));
    call generate(high(cspaddr(26)));
end eqli26;


eqls27: proc;
    ;
end eqls27;
```

```
neqb28: proc;
    dcl a addr;
    call generate(0c3h);          /* jmp */
    call generate(low(cspaddr(28) + 82));
    call generate(high(cspaddr(28) + 82));
    call pop$sv$addr(106h);       /* 7 bytes */
    a = cspaddr(28) + 7;
    call ne$bcd(a);               /* 67 bytes */
    call push$sv$addr(106h);        /* 7 bytes */
    call generate(0c9h);         /* ret */
    call generate(0cdh);          /* call */
    call generate(low(cspaddr(28)));
    call generate(high(cspaddr(28)));
    ;
end neqb28;


neqi29: proc;
    dcl a addr;
    call generate(0c3h);          /* jmp */
    call generate(low(cspaddr(29) + 39));
    call generate(high(cspaddr(29) + 39));
    call pop$sv$addr(106h);       /* 7 bytes */
    a = cspaddr(29) + 7;
    call ne$int(a);             /* 24 bytes */
    call push$sv$addr(106h);/*  7 bytes */
    call generate(0c9h);      /* ret */
    call generate(0cdh);       /* call */
    call generate(low(cspaddr(29)));
    call generate(high(cspaddr(29)));
end neqi29;


neqs30: proc;
    ;
end neqs30;


geqb31: proc;
    ;
end geqb31;


geqi32: proc;
    dcl a addr;
    call generate(0c3h);          /* jmp */
    call generate(low(cspaddr(32) + 38));
    call generate(high(cspaddr(32) + 38));
    call pop$sv$addr(106h);       /* 7 bytes */
    a = cspaddr(32) + 7;
    call ge$int(a);               /* 23 bytes */
    call push$sv$addr(106h);        /* 7 bytes */
    call generate(0c9h);         /* ret */
    call generate(0cdh);          /* call */
    call generate(low(cspaddr(32)));
    call generate(high(cspaddr(32)));
end geqi32;


grtb33: proc;
    ;
end grtb33;


grti34:proc;
    dcl a addr;
    call generate(0c3h);          /* jmp */
    call generate(low(cspaddr(34) + 38));
    call generate(high(cspaddr(34) + 38));
    call pop$sv$addr(106h);       /* 7 bytes */
    a = cspaddr(34) + 7;
    call gt$int(a);               /* 23 bytes */
```

```
    call pushSsvSaddr(106h);      /* 7 bytes */
    call generate(0c9h);        /* ret */
    call generate(0cdh);          /* call */
    call generate(low(cspaddr(34)));
    call generate(high(cspaddr(34)));
end grti34;


negb35: proc;
    call generate(0c1h);        /* pop b */
    call generate(3ah);         /* lda */
    call generate(80h);         /* 10000000 */
    call generate(81h);         /* add c */
    call generate(4fh);         /* movc a */
    call generate(0c5h);        /* push b */
end negb35;


negi36: proc;
    call generate(0c1h);        /* pop b */
    call generate(0afh);         /* xra a */
    call generate(91h);         /* sub c */
    call generate(4fh);         /* movc a */
    call generate(3eh);         /* mvi a */
    call generate(00h);        /* ooh */
    call generate(98h);        /* sbc b */
    call generate(47h);        /* movb a */
    call generate(0c5h);        /* push b */
end negi36;


comb37: proc;
    call generate(0c3h);        /* jmp */
    call generate(low(cspaddr(37) + 104));
    call generate(high(cspaddr(37) + 104));
    call popSsvSaddr(10eh);      /* 7 bytes */
    call popSbcd(106h);      /* 23 bytes */
    call compISbcd(106h);    /* 43 bytes */
    call pushSbcd(106h);     /* 23 bytes */
    call pushSsvSaddr(10eh);     /* 7 bytes */
    call generate(0c9h);         /* ret */
    call generate(0cdh);        /* call */
    call generate(low(cspaddr(37)));
    call generate(high(cspaddr(37)));
end comb37;


comi38: proc;
    call generate(0c1h);        /* pop b */
    call generate(79h);         /* mova c */
    call generate(0eeh);        /* xri */
    call generate(0ffh);        /* 11111111 */
    call generate(4fh);         /* movc a */
    call generate(78h);         /* mova b */
    call generate(0eeh);        /* xri */
    call generate(0ffh);        /* 11111111 */
    call generate(47h);         /* movb a */
    call generate(0c5h);        /* push b */
end comi38;


not39: proc;
    dcl a addr;
    call generate(0c3h);          /* jmp */
    call generate(low(cspaddr(39) + 34));
    call generate(high(cspaddr(39) + 34));
    call popSsvSaddr(106h);      /* 7 bytes */
    a = cspaddr(39) + 7;
    call notSbool(a);               /* 19 bytes */
    call pushSsvSaddr(106h);       /* 7 bytes */
    call generate(0c9h);        /* ret */
    call generate(0cdh);          /* call */
```

```
    call generate(low(cspaddr(39)));
    call generate(high(cspaddr(39)));
end not39;


and40: proc;
    dcl a addr;
    call generate(0c3h);          /* jmp */
    call generate(low(cspaddr(40) + 41));
    call generate(high(cspaddr(40) + 41));
    call pop$sv$addr(106h);       /* 7 bytes */
    a = cspaddr(40) + 7;
    call and$bool(a);              /* 26 bytes */
    call push$sv$addr(106h);      /* 7 bytes */
    call generate(0c9h);         /* ret */
    call generate(0cdh);          /* call */
    call generate(low(cspaddr(40)));
    call generate(high(cspaddr(40)));
end and40;


bor41: proc;
    dcl a addr;
    call generate(0c3h);          /* jmp */
    call generate(low(cspaddr(41) + 41));
    call generate(high(cspaddr(41) + 41));
    call pop$sv$addr(106h);       /* 7 bytes */
    a = cspaddr(41) + 7;
    call or$bool(a);               /* 26 bytes */
    call push$sv$addr(106h);      /* 7 bytes */
    call generate(0c9h);         /* ret */
    call generate(0cdh);          /* call */
    call generate(low(cspaddr(41)));
    call generate(high(cspaddr(41)));
end bor41;


stob42: proc;
    call generate(0c3h);       /* jmp */
    call generate(low(cspaddr(42) + 44));
    call generate(high(cspaddr(42) + 44));
    call pop$sv$addr(106h);       /* 7 bytes */
    call sto$bcd;                 /* 21 bytes */
    call sv$stack(bcd$len);       /* 8 bytes */
    call push$sv$addr(106h);      /* 7 bytes */
    call generate(0c9h);          /* ret */
    call generate(0cdh);          /* call */
    call generate(low(cspaddr(42)));
    call generate(high(cspaddr(42)));
end stob42;


stoi43: proc;
    call generate(0e1h);          /* pop h */
    call generate(0c1h);          /* pop b */
    call generate(71h);          /* movm c */
    call generate(23h);          /* inx h */
    call generate(70h);          /* movm b */
    call sv$stack(int1en); /* 2 bytes */
end stoi43;


sto44: proc;
    call generate(0e1h);       /* pop h */
    call generate(0c1h);      /* pop b */
    call generate(71h);       /* movm c */
    call generate(3bh);       /* dcx sp */
    call generate(3bh);       /* dcx sp */
end sto44;


stdb45:proc;
```

```
    call generate(0c3h);      /* jmp */
    call generate(low(cspaddr(45) + 36));
    call generate(high(cspaddr(45) + 36));
    call pop$sv$addr(106h);      /* 7 bytes */
    call sto$bcd;                /* 21 bytes */
    call push$sv$addr(106h);     /* 7 bytes */
    call generate(0c9h);         /* ret */
    call generate(0cdh);         /* call */
    call generate(low(cspaddr(45)));
    call generate(high(cspaddr(45)));
end stdb45;

stdi46: proc;
    call generate(0e1h);        /* pop h */
    call generate(0c1h);         /* pop b */
    call generate(71h);         /* movm c */
    call generate(23h);         /* inx h */
    call generate(70h);         /* movm b */
end stdi46;


std47: proc;
    call generate(0e1h);      /* pop h */
    call generate(0c1h);     /* pop b */
    call generate(71h);      /* movm c */
end std47;


cnai51: proc;
    dcl a addr;
    call generate(0c3h);        /* jmp */
    call generate(low(cspaddr(51) + 442));
    call generate(high(cspaddr(51) + 442));
    call pop$sv$addr(106h);              /* 7 bytes */
    call pop$sv$addr(11bh);            /* 7 bytes */
    call pop$int(108h);                  /* 7 bytes */
    a = cspaddr(51) + 21;
    call convert$int(a);                 /* 383 bytes */
    call push$bcd(10ah);                 /* 23 bytes */
    call push$sv$addr(11bh);         /* 7 bytes */
    call push$sv$addr(106h);             /* 7 bytes */
    call generate(0c9h);        /* ret */
    call generate(0cdh);        /* call */
    call generate(low(cspaddr(51)));
    call generate(high(cspaddr(51)));
end cnai51;


br152: proc;
    dcl (lbl,tot,lbladdr,n based lbladdr) addr;
    lbl = get$next$addr;
    lbladdr = .memory + (2 * lbl);
    tot = n + codestrt;
    call generate(0c3h);      /* jmp */
    call generate(low(tot));
    call generate(high(tot));
end br152;


blc53: proc;
    dcl (lbl,tot,lbladdr,n based lbladdr) addr;
    lbl = get$next$addr;
    lbladdr = .memory + (2 * lbl);
    tot = n + codestrt;
    call generate(0c1h);      /* pop b */
    call generate(79h);       /* mova c */
    call generate(0fh);       /* rrc */
    call generate(0dah);      /* jc */
    call generate(low(tot));
    call generate(high(tot));
end blc53;
```

```
cn2i54: proc;
    dcl a addr;
    call generate(0c3h);        /* jmp */
    call generate(low(cspaddr(54) + 474));
    call generate(high(cspaddr(54) + 474));
    call pop$sv$addr(106h);                 /* 7 bytes */
    call pop$bcd(11bh);             /* 23 bytes */
    call pop$int(108h);                     /* 7 bytes */
    a = cspaddr(54) + 37;
    call convert$int(a);                    /* 383 bytes */
    call push$bcd(10ah);                    /* 23 bytes */
    call push$bcd(11bh);             /* 23 bytes */
    call push$sv$addr(106h);                /* 7 bytes */
    call generate(0c9h);        /* ret */
    call generate(0cdh);        /* call */
    call generate(low(cspaddr(54)));
    call generate(high(cspaddr(54)));
end cn2i54;


lod55: proc;
    call generate(0e1h);      /* pop h */
    call generate(4eh);      /* movc m */
    call generate(06h);      /* mvi b */
    call generate(00h);      /* 0 */
    call generate(0c5h);     /* push b */
end lod55;


lodb56: proc;
    call generate(0c3h);      /* jmp */
    call generate(low(cspaddr(56) + 36));
    call generate(high(cspaddr(56) + 36));
    call pop$sv$addr(106h);     /* 7 bytes */
    call lod$bcd;               /* 21 bytes */
    call push$sv$addr(106h);    /* 7 bytes */
    call generate(0c9h);     /* ret */
    call generate(0cdh);        /* call */
    call generate(low(cspaddr(56)));
    call generate(high(cspaddr(56)));
end lodb56;


lodi57: proc;
    call generate(0e1h);      /* pop h */
    call generate(4eh);      /* movc m */
    call generate(23h);      /* inx h */
    call generate(46h);       /* movb m */
    call generate(0c5h);     /* push b */
end lodi57;


rdvb58: proc;
    ;
end rdvb58;


rdvi59: proc;
    ;
end rdvi59;


rdvs60: proc;
    ;
end rdvs60;


wrvb61: proc;
    dcl a addr;
    call generate(0c3h);      /* jmp */
    call generate(low(cspaddr(61) + 502));
    call generate(high(cspaddr(61) + 502));
```

```
    call popSsvSaddr(106h);      /* 7 bytes */
    call popSbcd(108h);          /* 23 bytes */
    a = cspaddr(61) + 30;
    call printSbcd(a);           /* 464 bytes */
    call pushSsvSaddr(106h);     /* 7 bytes */
    call generate(0c9h);         /* ret */
    call generate(0cdh);         /* call */
    call generate(low(cspaddr(61)));
    call generate(high(cspaddr(61)));
end wrvb61;


wrvi62: proc;
    dcl a addr;
    call generate(0c3h);      /* jmp */
    call generate(low(cspaddr(62) + 592));
    call generate(high(cspaddr(62) + 592));
    call popSsvSaddr(106h);      /* 7 bytes */
    call popSint(108h);          /* 7 bytes */
    a = cspaddr(62) + 14;
    call printSint(a);           /* 570 bytes */
    call pushSsvSaddr(106h);     /* 7 bytes */
    call generate(0c9h);         /* ret */
    call generate(0cdh);         /* call */
    call generate(low(cspaddr(62)));
    call generate(high(cspaddr(62)));
end wrvi62;


wrvs63: proc(n);
    dcl (n,i,ch) byte;
    i = 0;
    do while i < n;
        ch = getSnextSbyte;
        call generate(0eh);     /* movi c */
        call generate(ch);
        call generate(0cdh);    /* call */
        call generate(low(cspaddr(63)));
        call generate(high(cspaddr(63)));
        i = i + 1;
    end;
end wrvs63;


dmp64: proc;
    call generate(0c3h);         /* jmp */
    call generate(low(cspaddr(64) + 39));
    call generate(high(cspaddr(64) + 39));
    call dump;                   /* 38 bytes */
    call generate(0c9h);         /* ret */
    call generate(0cdh);         /* call */
    call generate(low(cspaddr(64)));
    call generate(high(cspaddr(64)));
end dmp64;

/*****************************************************************************/
/*****************************************************************************/
/***      i n t e r p r e t e r    m a i n    p r o g r a m        ***/
/*****************************************************************************/
/*****************************************************************************/

do;
   call openSpinSfile;
   call setSflags;
   do while pass1 or pass2;
      pincode = getSnextSbyte;
      do case pincode;
```

```
/*          0      nop      no operation    */
        ;

/*          1      endp    -  end of program    */
    if pass1 then call endp1;
    else call endprog;

/*          2      lbl    -  label  */

   if pass1 then call lbl2;
     else
          tempaddr = getSnextSaddr;

/*          3       ldib -    load immediate bcd number    */
   if pass1 then do;
          codecount = codecount + 20;
          tempaddr = getSnextSaddr;
          tempaddr = getSnextSaddr;
          tempaddr = getSnextSaddr;
          tempaddr = getSnextSaddr;
          end;
    else call ldib3;

/*          4       ldii -    load immediate integer          */
   if pass1 then do;
        codecount = codecount + 5;
        tempaddr = getSnextSaddr;
        end;
    else call ldii4;

/*          5      savp   -    save parameters (not implemented)    */
    ;

/*          6      unsp   -    unsave parameters (not implemented)    */
    ;

/*          7      pro    -    procedure call (not implemented)    */
    ;

/*          8      rtn    -    return from procedure (not implemented)    */
    ;

/*          9      cnvb   -    convert bcd to integer    */
   if pass1 then
      do;
         if cspc(9) then
            do;
               codecount= codecount + 364;
               cspc(9)= false;
            end;
         else codecount = codecount + 3;
      end;
   else if cspc(9) then
            do;
                cspaddr(9 )= codesize + 3;
                call cnvb9;
                cspc(9)= false;
            end;
        else call sec3pass(cspaddr(9));


/*         10   .  cnvi   -    convert integer to bcd    */

   if pass1 then
      do;
         if cspc(10) then
            do;
               codecount= codecount + 434;
               cspc(10)= false;
            end;
         else codecount = codecount + 3;
      end;
```

```
   else if cspc(10) then
           do;
               cspaddr(10)= codesize + 3;
               call cnvi10;
               cspc(10)= false;
           end;
        else call sec3pass(cspaddr(10));

/*         11      all   -   allocate variable      */

   if pass1 then do;
     tempaddr = get3next3addr;
     varcount = varcount + tempaddr;
     end;
    else tempaddr = get3next3addr;

/*         12      lita  -   literal address          */

   if pass1 then codecount = codecount + 4;
    else call lita12;

/*         13      addb  -   add bcd numbers    */

     ;

/*         14      addi  -   add integer numbers    */

   if pass1 then codecount = codecount + 4;
    else call addi14;

/*         15      subb  -   subtract bcd numbers    */

     ;

/*         16      subi  -   subtract integer numbers    */

   if pass1 then codecount = codecount + 9;
    else call subi16;

/*         17      mulb  -   multiply bcd numbers    */
    ;

/*         18      muli  -   multiply integer numbers    */

   if pass1 then
      do;
         if cspc(18) then
           do;
               codecount= codecount + 61;
               cspc(18)= false;
           end;
        else codecount = codecount + 3;
      end;
   else if cspc(18) then
           do;
               cspaddr(18)= codesize + 3;
               call muli18;
               cspc(18)= false;
           end;
        else call sec3pass(cspaddr(18));

/*         19      divb  -   divide bcd numbers    */
    ;

/*         20      divi  -   divide integer numbers  */

   if pass1 then
      do;
         if cspc(20) then
           do;
```

```
                codecount= codecount + 75;
                cspc(20)= false;
             end;
          else codecount = codecount + 3;
       end;
    else if cspc(20) then
             do;
                cspaddr(20)= codesize + 3;
                cspc(1)= false;
                call divi20;
             end;
          else call sec3pass(cspaddr(20));

/*        21     lssb  -   less than compare, bcd    */

     ;

/*        22     lssi  -   less than compare, integer   */

    if pass1 then
       do;
          if cspc(22) then
             do;
                codecount= codecount + 44;
                cspc(22)= false;
             end;
          else codecount = codecount + 3;
       end;
    else if cspc(22) then
             do;
                cspaddr(22)= codesize + 3;
                call lssi22;
                cspc(22)= false;
             end;
          else call sec3pass(cspaddr(22));

/*        23     leqb  -   less than or equal compare, bcd   */

     ;

/*        24     leqi  -   less than or equal compare, integer   */

    if pass1 then
       do;
          if cspc(24) then
             do;
                codecount= codecount + 44;
                cspc(24)= false;
             end;
          else codecount = codecount + 3;
       end;
    else if cspc(24) then
             do;
                cspaddr(24)= codesize + 3;
                call leqi24;
                cspc(24)= false;
             end;
          else call sec3pass(cspaddr(24));

/*        25     eqlb  -   equal compare, bcd   */

    if pass1 then
       do;
          if cspc(25) then
             do;
                codecount= codecount + 87;
                cspc(25)= false;
             end;
          else codecount = codecount + 3;
```

```
      end;
   else if cspc(25) then
           do;
               cspaddr(25)= codesize + 3;
               call eqlb25;
               cspc(25)= false;
           end;
         else call sec3pass(cspaddr(25));

/*        26     eqli  -   equal compare, integer   */

   if pass1 then
      do;
         if cspc(26) then
           do;
               codecount= codecount + 45;
               cspc(26)= false;
           end;
         else codecount = codecount + 3;
      end;
   else if cspc(26) then
           do;
               cspaddr(26)= codesize + 3;
               call eqli26;
               cspc(26)= false;
           end;
         else call sec3pass(cspaddr(26));

/*        27     eqls  -   equal compare, string   */
    ;

/*        28     neqb  -   not equal compare, bcd   */
   if pass1 then
      do;
         if cspc(28) then
           do;
               codecount= codecount + 88;
               cspc(28)= false;
           end;
         else codecount = codecount + 3;
      end;
   else if cspc(28) then
           do;
               cspaddr(28)= codesize + 3;
               call neqb28;
               cspc(28)= false;
           end;
         else call sec3pass(cspaddr(28));

/*        29     neqi  -   not equal compare, integer   */
   if pass1 then
      do;
         if cspc(29) then
           do;
               codecount= codecount + 45;
               cspc(29)= false;
           end;
         else codecount = codecount + 3;
      end;
   else if cspc(29) then
           do;
               cspaddr(29)= codesize + 3;
               call  neqi29;
               cspc(29)= false;
           end;
         else call sec3pass(cspaddr(29));

/*        30     neqs  -   not equal compare, string   */
```

```
   ;
/*        31     geqb  -   greater or equal compare, bcd   */

    ;

/*        32     geqi  -   greater or equal compare, integer   */
   if pass1 then
      do;
         if cspc(32) then
            do;
               codecount= codecount + 44;
               cspc(32)= false;
            end;
         else codecount = codecount + 3;
      end;
   else if cspc(32) then
            do;
               cspaddr(32)= codesize + 3;
               call geqi32;
               cspc(32)= false;
            end;
         else call sec3pass(cspaddr(32));


/*        33     grtb  -   greater than compare, bcd   */
    ;

/*        34     grti  -   greater than compare, integer   */
   if pass1 then
      do;
         if cspc(34) then
            do;
               codecount= codecount + 44;
               cspc(34)= false;
            end;
         else codecount = codecount + 3;
      end;
   else if cspc(34) then
            do;
               cspaddr(34)= codesize + 3;
               call grti34;
               cspc(34)= false;
            end;
         else call sec3pass(cspaddr(34));


/*        35     negb  -   change sign of bcd   */
   if pass1 then codecount = codecount + 6;
   else call negb35;


/*        36     negi  -   change sign of integer   */

   if pass1 then codecount= codecount + 9;
   else call negi36;

/*        37     comb  -   complement (9's) bcd   */

   if pass1 then
      do;
         if cspc(37) then
            do;
               codecount= codecount +  110;
               cspc(37)= false;
            end;
         else codecount = codecount + 3;
      end;
   else if cspc(37) then
            do;
               cspaddr(37)= codesize + 3;
               call comb37;
               cspc(37)= false;
```

```
            end;
         else call sec3pass(cspaddr(37));

/*        38     comi  -   complement (2's) integer   */

   if pass1 then codecount = codecount + 10;
   else call comi38;

/*        39     not   -   boolean negative   */

   if pass1 then
      do;
         if cspc(39) then
            do;
               codecount= codecount + 40;
               cspc(39)= false;
            end;
         else codecount = codecount + 3;
      end;
   else if cspc(39) then
            do;
               cspaddr(39)= codesize + 3;
               call not39;
               cspc(39)= false;
            end;
         else call sec3pass(cspaddr(39));

/*        40     and   -   logical and   */
   if pass1 then
      do;
         if cspc(40) then
            do;
               codecount= codecount + 47;
               cspc(40)= false;
            end;
         else codecount = codecount + 3;
      end;
   else if cspc(40) then
            do;
               cspaddr(40)= codesize + 3;
               call and40;
               cspc(40)= false;
            end;
         else call sec3pass(cspaddr(40));

/*        41     bor   -   logical or   */

   if pass1 then
      do;
         if cspc(41) then
            do;
               codecount= codecount + 47;
               cspc(41)= false;
            end;
         else codecount = codecount + 3;
      end;
   else if cspc(41) then
            do;
               cspaddr(41)= codesize + 3;
               call bor41;
               cspc(41)= false;
            end;
         else call sec3pass(cspaddr(41));

/*        42     stob  -   store bcd   */

   if pass1 then
      do;
         if cspc(42) then
```

```
            do;
               codecount= codecount + 50;
               cspc(42)= false;
            end;
         else codecount = codecount + 3;
      end;
   else if cspc(42) then
            do;
               cspaddr(42)= codesize + 3;
               call stob42;
               cspc(42)= false;
            end;
         else call sec3pass(cspaddr(42));

/*        43     stoi  -   store integer   */

   if pass1 then codecount = codecount + 7;
     else call stoi43;

/*        44     sto   -   store byte    */

   if pass1 then codecount = codecount + 5;
     else call sto44;

/*        45     stdb  -   store destruct bcd    */

   if pass1 then
      do;
         if cspc(45) then
            do;
               codecount= codecount + 42;
               cspc(45)= false;
            end;
         else codecount = codecount + 3;
      end;
   else if cspc(45) then
            do;
               cspaddr(45)= codesize + 3;
               call stdb45;
               cspc(45)= false;
            end;
         else call sec3pass(cspaddr(45));

/*        46     stdi  -   store destruct integer    */

   if pass1 then codecount = codecount + 5;
    else call stdi46;

/*        47     std   -   store destruct byte    */

   if pass1 then codecount = codecount + 3;
    else call std47;

/*        48     dcrb  -   decrement stack bcd    */

   if pass1 then codecount = codecount + 8;
     else call unsv3stack(bcd$len);

/*        49     dcri  -   decrement stack integer    */

   if pass1 then codecount = codecount + 2;
    else call unsv3stack(int$len);

/*        50     dcr   -   decrement stack byte    */

   if pass1 then codecount = codecount + 2;
     else call unsv3stack(int$len);

/*        51      cnai - convert integer preceeded by address     */
   if pass1 then
```

```
      do;
         if cspc(51) then
            do;
               codecount= codecount + 448;
               cspc(51)= false;
            end;
         else codecount = codecount + 3;
      end;
   else if cspc(51) then
            do;
               cspaddr(51)= codesize + 3;
               call cnai51;
               cspc(51)= false;
            end;
         else call sec$pass(cspaddr(51));

/*        52     brl   -   branch label absolute   */

   if pass1 then
      do;
      codecount = codecount + 3;
      tempaddr = get$next$addr;
      end;
   else call brl52;

/*        53     blc   -   branch label conditional   */

   if pass1 then
      do;
      codecount = codecount + 6;
      tempaddr = get$next$addr;
      end;
   else call blc53;

/*        54      cn2i - convert integer preceeded by bcd    */
   if pass1 then
      do;
         if cspc(54) then
            do;
               codecount= codecount + 480;
               cspc(54)= false;
            end;
         else codecount = codecount + 3;
      end;
   else if cspc(54) then
            do;
               cspaddr(54)= codesize + 3;
               call cn2i54;
               cspc(54)= false;
            end;
         else call sec$pass(cspaddr(54));

/*        55     lod   -   load byte           */

   if pass1 then codecount = codecount + 5;
     else call lod55;


/*        56     lodb  -   load bcd number    */

   if pass1 then
      do;
         if cspc(56) then
            do;
               codecount= codecount + 42;
               cspc(56)= false;
            end;
         else codecount = codecount + 3;
      end;
   else if cspc(56) then
            do;
               cspaddr(56)= codesize + 3;
```

```
            call lodb56;
            cspc(56)= false;
         end;
      else call sec3pass(cspaddr(56));

/*       57     lodi  -  load integer number    */

   if pass1 then codecount = codecount + 5;
    else call lodi57;

/*       58     rdvb  -   read variable bcd   */
   if pass1 then
      do;
         if cspc(58) then
            do;
               codecount= codecount + 1;
               cspc(58)= false;
            end;
         else codecount = codecount + 3;
      end;
   else if cspc(58) then
            do;
               cspaddr(58)= codesize + 3;
               call rdvb58;
               cspc(58)= false;
            end;
         else call sec3pass(cspaddr(58));

/*       59     rdvi  -   read variable integer   */
   if pass1 then
      do;
         if cspc(59) then
            do;
               codecount= codecount + 1;
               cspc(59)= false;
            end;
         else codecount = codecount + 3;
      end;
   else if cspc(59) then
            do;
               cspaddr(59)= codesize + 3;
               call rdvi59;
               cspc(59)= false;
            end;
         else call sec3pass(cspaddr(59));

/*       60     rdvs  -   read variable string   */

    ;

/*       61     wrvb  -   write variable bcd   */
   if pass1 then
      do;
         if cspc(61) then
            do;
               codecount= codecount + 508;
               cspc(61)= false;
            end;
         else codecount = codecount + 3;
      end;
   else if cspc(61) then
            do;
               cspaddr(61)= codesize + 3;
               call wrvb61;
               cspc(61)= false;
            end;
         else call sec3pass(cspaddr(61));
```

```
/*        62     wrvi  -   write variable integer   */
   if pass1 then
      do;
         if cspc(62) then
            do;
               codecount= codecount + 598;
               cspc(62)= false;
            end;
         else codecount = codecount + 3;
      end;
   else if cspc(62) then
           do;
               cspaddr(62)= codesize + 3;
               call wrvi62;
               cspc(62)= false;
           end;
        else call sec3pass(cspaddr(62));


/*        63     wrvs  -   write variable string   */
   if pass1 then
      do;
         if cspc(63) then
            do;
               tempbyte = get$next$byte;
               codecount = codecount + 61 + double(tempbyte)* 5;
               cspc(63)= false;
            end;
         else do;
             tempbyte = get$next$byte;
             codecount = codecount + double(tempbyte) * 5;
             end;
      do  loop = 0 to tempbyte;
          pincode = get$next$byte;
       end;
      end;
   else if cspc(63) then
           do;
               cspaddr(63)= codesize + 0fh;
               call write$strng;       /* 61 bytes */
               tempbyte = get$next$byte;
               call wrvs63(tempbyte);
               cspc(63)= false;
           end;
        else do;
            tempbyte = get$next$byte;
            call wrst63(tempbyte);
            end;

/*       64     dmp -  start new output line   */

   if pass1 then
      do;
         if cspc(64) then
            do;
               codecount= codecount + 45;
               cspc(64)= false;
            end;
         else codecount = codecount + 3;
      end;
   else if cspc(64) then
           do;
               cspaddr(64)= codesize + 3;
               call dmp64;
               cspc(64)= false;
           end;
        else call sec3pass(cspaddr(64));

/*       65     not used                        */

    do;
        call error('co');       /* code overflow */
```

```
        call endprog;
    end;


      end;                          /* case pincode  */
   end;                             /* do while pass1 */
end;
eof
```

LIST OF REFERENCES

1. Aho, A V. and Ullman, J. D., "Principles . of Compiler Design", Addison-Wesley Publishing Company, Reading, Mass. 1977.

2. Conway, R., Gries, D., and Zimmerman, E. C., "A Primer On Pascal", Winthrop Publishers, Inc. Cambridge, Mass. 1976.

3. Digital Research, An Introduction to CP/M Features and Facilities, 1976, Box 579, Pacific Grove, CA., 93959.

4. Digital Research, CP/M Interface Guide, 1976.

5. 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 Thesis, Naval Postgraduate School, Monterey Ca., September 1977.

6. Gries, D., "Compiler Construction For Digital Computers", John Wiley & Sons, Inc. New York N.Y., 1971.

7. Grogono, P., "Programming In Pascal", Addison-Wesley Publishing Company, Reading, Mass. Copywright 1978.

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

9. Intel Corporation, 8008 and 8080 PL/M Programming Manual, 1975, 3065 Bowers Ave, Santa Clara, CA., 95051.

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

11. Naur, P. (ed.), "Report on the Algorithmic Language ALGOL 60", Comm. ACM, Vol. 3, No. 5, May 1960, pp. 299-314.

12. 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

13. Nori, K. V. and others, "The PASCAL <P> Compiler: Implementation Notes", Eidgenossische Technische Hochschule, Zurich, Revised Edition, July 1976.

14. Strutynski, Kathryn B. Information on the CP/M

Interface Simulator, internally distributed technical note.

15. Lalonde, W. R., User's Guide To The LALR(k) Parser Generator, University of Toronto Computer Systems Research Group, Toronto Canada, 8 April 1971.

16. Lee, J. A. N., "The Anatomy Of A Compiler", D. Van Nostrand Company, New York N.Y., Copyright 1974.

17. McKeeman, W. M., Horning, J. J., and Wortman, D. B., "A Compiler Generator", Prentice-Hall Inc., 1970.

18. Pollack, B. N., "Compiler Techniques", Copyright Averbach Publishers Inc., 1972.

19. Rohl, J. S., "An Introduction To Compiler Writing", Mac Donald and Jane's, London and American Elsevier Inc. New York, 1975.

20. Soucek, B., "Microprocessors & Microcomputers", John Wiley and Sons Inc. New York, London, Sydney, Toronto, 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 Naval Postgraduate School Monterey, California 93940	1
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	2
8.	Lt Robert R. Stilwell, SC, USN NSD Guam Code 60 FPO San Francisco 96630	2

9. LCDR Antonio L. S. Goncalves, Brazilian Navy 1
Rua Prudente de Moraes, 660 Apt. 202
Ipanema, Rio de Janeiro 20000
RJ, BRAZIL

10. LT(JG) Javier E. De La Cuba, Peruvian Navy 1
Direccion de Instruccion de la Marina
Ministerio de Marina
Ave. Salaverry S/N
Lima, PERU