1979

NPS MICRO-COBOL : an implementation of Navy standard HYPO-COBOL for a microprocessor-based computer system

Farlee, Jim

Monterey, California. Naval Postgraduate School

NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

NPS MICRO-COBOL
an Implementation of
Navy Standard HYPO-COBOL
for a Microprocessor-Based Computer System

by

Jim Farlee Jr.
and
Michael L. Rice

June 1979

Thesis Advisor: M. S. Moranville

Approved for public release; distribution unlimited.

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) NPS MICRO-COBOL an Implementation of Navy Standard HYPO-COBOL for a Microprocessor-Based Computer System		5. TYPE OF REPORT & PERIOD COVERED Master's Thesis; June 1979 6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Jim Farlee Jr. Michael L. Rice		8. CONTRACT OR GRANT NUMBER(s)
9. PERFORMING ORGANIZATION NAME AND ADDRESS Naval Psotgraduate School Monterey, California 93940		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Naval Postgraduate School Monterey, California 93940		12. REPORT DATE June 1979 13. NUMBER OF PAGES
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)		15. SECURITY CLASS. (of this report) Unclassified 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

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

18. SUPPLEMENTARY NOTES

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

NPS MICRO-COBOL
Navy Standard HYPO-COBOL
Microcomputers
Compilers

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

A compiler for a subset of the Automated Data Processing Equipment Selection Office (ADPESO) HYPO-COBOL has been implemented on a Microcomputer. The implementation provides nucleus level constructs and file options from the ANSI COBOL package along with the PERFORM UNTIL construct from a higher level to give increased structural control. The language was

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE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

implemented through a compiler and run-time package executing under the CP/M operating system of an 8080 microcomputer-based system. Both the compiler and interpreter can be executed in 20K bytes of main memory. A program consisting of 8.5K bytes of intermediate code can be supported on this size machine. Modification of the compiler and interpreter programs can be accomplished to take advantage of larger machines. The programs that make up the compiler and interpreter package require 50K bytes of disk storage.

Approved for public release; distribution unlimited.

NPS MICRO-COBOL
an implemention of
Navy Standard HYPO-COBOL
for a microprocessor-based computer system

by

Jim Farlee Jr.
Captain, United States Marine Corps
B.S., University of Nebraska, 1970

Michael L. Rice
Captain, United States Marine Corps
B.S., Utah State University, 1973

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
June 1979

ABSTRACT

A compiler for a subset of the Automated Data Processing Equipment Selection Office (ADPESO) HYPO-COBOL has been implemented on a microcomputer. The implementation provides nucleus level constructs and file options from the ANSI COBOL package along with the PERFORM UNTIL construct from a higher level to give increased structural control. The language was implemented through a compiler and run-time package executing under the CP/M operating system of an 8080 microcomputer-based system. Both the compiler and interpreter can be executed in 20K bytes of main memory. A program consisting of 8.5K bytes of intermediate code can be supported on this size machine. Modification of the compiler and interpreter programs can be accomplished to take advantage of larger machines. The programs that make up the compiler and interpreter package require 50K bytes of disk storage.

TABLE OF CONTENTS

I. INTRODUCTION

A. BACKGROUND

The NPS MICRO-COBOL Compiler/Interpreter was initially (1976) developed to demonstrate that it was feasible to implement a COBOL compiler on a micro-computer. It was known that the COBOL language used would have to be a subset of ANSI COBOL because of the restriction imposed by the size of a micro-computer memory. A subset of ANSI COBOL, specifically ADPSO HYPO-COBOL, was selected as the basis for the implementation [3]. Additional motivation was provided by the DOD requirement that all computers used in a non-tactical environment be capable of executing COBOL.

The previous work was directed toward five major areas: 1.) selecting a suitable COBOL subset to operate on, 2.) develop the associated grammar for the language, 3.) determine what type of compiler to design, 4.) design and code the compiler, and 5.) design and code the interpreter. The interpreter performs the functions of a classical linking loader, resolving forward address references and establishing the run time intermediate code environment, as well as, executing the intermedite code.

The establishment of a suitable language was easily determined since HYPO-COBOL was a Department of the Navy approved subset of COBOL, designed to place minimal requirements on a system for compiler support. Where

possible, short constructs were used in the place of longer ones. Where more than one reserved word served the same function in COBOL the shortest form was used. There is no optional verbage in the language, and no duplicate constructs perform the same function. Limits were placed on all statements that had a variable input format so that all statements had a fixed maximum length. Where possible, such constructs were removed completely from the language. In addition, user defined identifier names were limited to twelve characters to reduce symbol table storage requirements.

Rather than include the standard levels of implementation for all of the modules, constructs were included only as required. In addition to low level constructs, the PERFORM UNTIL construct was included to allow better program structure. Further justification for the manner of subsetting and a highly detailed description of each element of the language is contained in the HYPO-COBOL language specifications reference 3. For a comparison of HYPO-COBOL constructs that are not supported by MICRO-COBOL see appendix G.

The grammar for the MICRO-COBOL language was defined as LALR(1). The compiler design was based on a table-driven parser for the LALR(1) grammar. The algorithm used to develope the parse tables for the compiler was developed by V. Lalonge [17].

The basic design and coding of the compiler and

interpreter was completed prior to the current thesis work by Scott Allan Craig [2]. Modification to the original thesis work was conducted by Phil Mylet [15].

B. OPERATING ENVIRONMENT

The NPS MICRO-COBOL compiler and interpreter are designed to run under the CP/M operating system on an 8080 or Z80 based microcomputer with at least 20K bytes of main memory. The compiler programs are designed to use no more than 12K bytes of main memory, while the interpreter program uses approximately 8K bytes. The compiler and interpreter require 50K bytes of disk storage for the programs that make up the compiler/interpreter package. For information on creating MICRO-COBOL source programs and CP/M see references 4 and 5.

C. GOALS AND OBJECTIVES

The primary goal of this thesis project was to complete the implementation of an 8080 microcomputer based compiler/interpreter, which could compile and execute a subset of the ANSI Standard HYPO-COBOL language specification. To achieve this goal both the compiler and interpreter would require testing, debugging, modification and implementation (extension) of any necessary additional language constructs. It was also decided that while testing and debugging, the documentation of the compiler's and

interpreter's internal structures, memory organization, interfaces and module functions would be accomplished.

Since the amount of testing and debugging effort could not be accurately determined, several subgoals were established which would be undertaken if adequate time was available. These time dependent goals included the validation of the compiler and interpreter and the inclusion of additional language constructs not previously implemented.

In addition to the above goals, it was considered beneficial to update and incorporate all previous thesis documentation into the present NPS MICRO-COBOL compiler and interpreter documentation. This documentation is appended to this thesis.

D. PROBLEM DEFINITION

For software performance assessment, a series of simple COBOL source programs and the Navy Automated Data Processing Equipment Selection Office HYPO-COBOL validation test programs (HCCVS) were compiled and execution was attempted. An evaluation of the test results indicated that the compiler and interpreter could only compile and execute very simple test programs. In particular, the compiler was unable to compile past the file section of the first validation program.

A review of the compiler and interpreter documentation led to several additional conclusions. The compiler and

interpreter were difficult to understand, and program logic flow was hard to follow, because: 1.) modular functions were not explained well, 2.) documentation on the module interfacing was inadequate, 3.) complete specifications describing the internal structures and memory organization did not exist, and 4.) few comments were included within the source code listings.

II. NPS MICRO-COBOL COMPILER

A. GENERAL DESCRIPTION

The MICRO-COBOL compiler is a one pass compiler that scans and parses MICRO-COBOL source programs, and generates intermediate code (pseudo-instructions) for the interpreter (pseudo-machine). The scanner design is similar to most other scanner implementations. The parser is an LALR(1) table-driven design, implemented in the PLM80 programming language [8]. The parse tables, as stated before, were generated using an algorithm developed at the University of Toronto [17].

The compiler reads the source program from a disk file, extracts the needed information for the symbol table and writes the pseudo-instructions to an intermediate code file. To accomplish this function, the compiler consists of three modules: PART ONE, IREADER, and PART TWO.

B. SYMBOL TABLE

The symbol table is the key data structure in the compiler. Information concerning identifiers, files, and records specified in the DATA DIVISION of the MICRO-COBOL source program is stored in the symbol table, along with labels specified in the PROCEDURE DIVISION.

The symbol table structure consists of: 1.) a sixty-four

address hash table, 2.) a fixed length field of thirteen bytes for each symbol table entry, and 3.) a variable length field to hold the name of each identifier. Since each identifier name is limited to twelve ASCII characters the symbol table entry for identifiers can vary in length from thirteen to twenty-five bytes. The bytes of each symbol table entry are grouped into various fields of either one or two bytes depending on the storage requirements. The thirteen bytes of the fixed length field entry are numbered from zero to twelve and the variable length field begins with byte thirteen. In referencing a specific field a byte index with a value from zero to thirteen is utilized.

The symbol table entry for a single identifier could contain up to nine different attributes of that identifier, although not all identifiers required the full range of attributes. The various fields in the symbol table contained different information depending on whether, for example, an identifier was a numeric or alphanumeric type. Four of the fields contained the same information for all identifiers. These fields were: 1.) field zero (bytes zero and one) contained the collision link, 2.) field one (byte two) contained the type of the identifier, 3.) field two (byte three) contained the length of the identifier name, and 4.) field thirteen (byte thirteen) was the beginning of the ASCII character representation. It should be noted that an identifier of type FILLER would not have a name associated with it, so field two would contain a zero and field

thirteen would not exist.

Entry into the symbol table is accomplished by using a HASH function on the ASCII character representation of the identifier name. This function generates an even number between zero and 126. The number is used as an index into the hash table by specifying an offset from the base of the hash table. The hash table can hold sixty-four uniquely determined address references to identifiers. The hash table entry associated with each index reference heads a linked list of identifiers with the same HASH function value. The linked list structure provides for additional identifier storage and therefore the number of unique identifiers is not limited by the sixty-four index values generated by the HASH function. A zero entry in the hash table indicates that there is no identifier with that HASH function value. In tracing through the linked list of identifiers the most recently declared variable appears at the end of the list. See figure [II-1] for an example of the computation of a hash value. See figure [II-2] for and example of the hash table indexing and linking of hash values.

HASH VALUE COMPUTATION

HASH Function value: sum of identifier ASCII characters logically and with 3FH then shifted left (SHL) one bit.

HASHBASE = 2000H

H.F.(AB) = HASHBASE + SHL(((41H + 42H) AND 3FH),1) = 2006H

H.F.(BA) = HASHBASE + SHL(((42H + 41H) AND 3FH),1) = 2006H

FIGURE II-1

HASH TABLE, SYMBOL TABLE LINKING

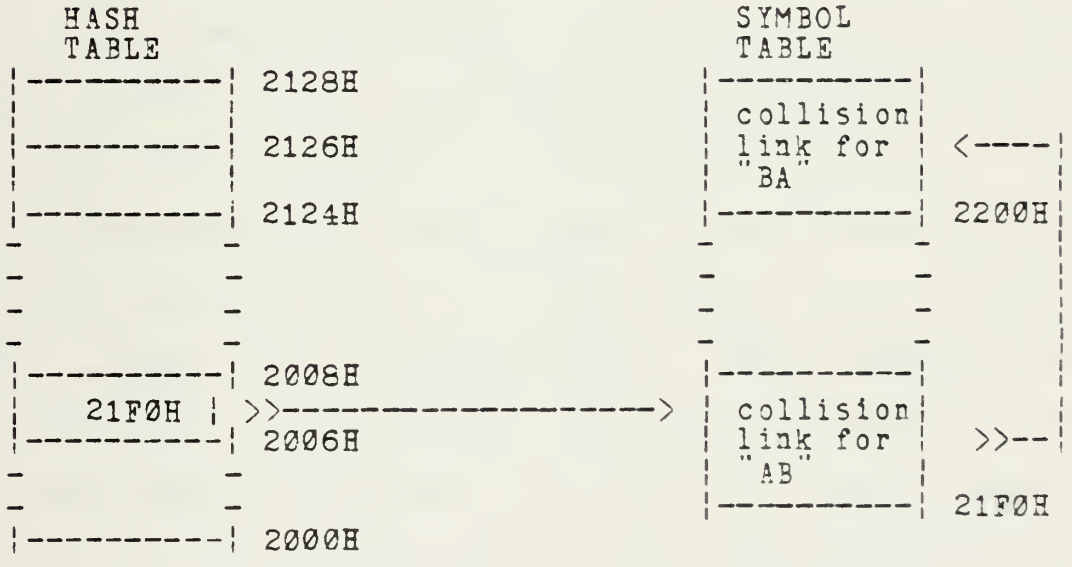

FIGURE II-2

1. Numeric Values

The symbol table entry for numeric values can contain up to seven attributes of the variable. These attributes are: 1.) identifier type, 2.) length of variable name 3.) beginning address of variable storage, 4.) numeric count (number of storage locations required by the identifier), 5.) level number, 6.) number of digits to the right of the decimal point, and 7.) the variable name. Figures [II-3] and [II-4] illustrate, respectively, the following two COBOL declarations:

```
Ø1 NUM PIC 9(9).
Ø1 NUM PIC 9(6).999 OCCURS 12 TIMES.
```

2. Numeric Edit

The numeric edit symbol table entry expands on the numeric symbol entry and utilizes bytes eight and nine to hold the beginning address, in the constants area, of the edit field mask. This mask allowed for the insertion of the following characters into and output display of a numeric number: fixed and floating dollar signs, credit(CR) and debit(DB) signs, asterisk fill, "Z" character fill, and plus ("+") and minus ("-") signs. It should be noted that an identifier with a numeric edit field value can not be used in an arithmetic statement.

NUMERIC SYMBOL TABLE ENTRY.

BYTE	SYMBOL TABLE VALUE
Ø-1	collision link (ØØ ØØ)
2	type identifier (1Ø)
3	length of identifier name (Ø3)
4-5	beginning address of identifier storage (Ø4 25)
6-7	length of identifier storage (Ø9 ØØ)
8-9	not used
1Ø	level entry (Ø1)
11	decimal count (ØØ)
12	occurances (ØØ)
13-15	identifier name (4E 55 4D)

Ø1 NUM PIC 9(9).

FIGURE II-3

NUMERIC SYMBOL TABLE ENTRY WITH DECIMAL
AND OCCURS CLAUSE

BYTE	SYMBOL TABLE VALUE
Ø-1	collision link (Ø9 2E)
2	type identifier (1Ø)
3	length of identifier name (Ø3)
4-5	beginning address of identifier storage (ØD 25)
6-7	length of identifier storage (Ø9 ØØ)
8-9	not used
1Ø	level entry (Ø1)
11	decimal count (Ø3)
12	occurances (ØC)
13-15	identifier name (4E 55 4D)

Ø1 NUM PIC 9(6).999 OCCURS 12 TIMES.

FIGURE II-4

3. Alpha or Alphanumeric

The alpha and alphanumeric symbol table entries appear similarly in the symbol table except for their type fields. Six entries appear in the symbol table for these identifiers: 1.) identifier type, 2.) length of identifier name, 3.) beginning address of storage, 4.) number of storage locations required by identifier, 5.) level entry, and 6.) identifer name. Figure [II-5] illustrates an alpha symbol table entry for the following identifier declaration:

Ø1 ALPHA PIC A(8).

4. Alpha Edit

The alpha edit symbol table entry expands on the alpha and alphanumeric edit types and utilizes bytes eight and nine to hold the beginning address of the edit field mask. These mask fields, which are stored in the constants area of the pseudo-machine, contain the characters necessary to edit an output so that, for example, slashes or blanks can be interspersed in the display output.

ALPHA SYMBOL TABLE ENTRY

BYTE	SYMBOL TABLE VALUE
0-1	collision link (00 00)
2	type identifier (08)
3	length of identifier (05)
4-5	beginning address of identifier storage (16 25)
6-7	length of identifier storage (08 00)
8-9	not used
10	level entry (01)
11	not used
12	not used
13-17	identifier name (41 4C 50 48 41)

01 ALPHA PIC A(8).

FIGURE II-5

5. Tables

NPS MICRO-COBOL was designed to support singly indexed tables. These tables are established by using an OCCURS clause with the PICTURE clause of an identifier. If an identifier is specified as a table the number of occurances of the table are placed in byte twelve of the symbol table entry for that identifier. The table identifier in COBOL is similar to the subscripted variable in other programming languages. For example, the statement, "Ø1 NUM PIC 9(9) OCCURS 12 TIMES", generates the symbol table entry illustrated in figure [II-4].

6. Labels

Labels generate the simplest of all symbol table entries, only four or five attributes are associated with the label. The variability depends on whether the label is declared in the source program before or after the label is referenced by a GO or PERFORM statement. In the event a label is specified before a GO or PERFORM statement references it, the symbol table would contain the following 1.) the type associated with label, 2.) the length of the identifier name, 3.) the address of the first intermediate code instruction following the appearance of the label in the source program (bytes four and five), 4.) the last executable instruction associated with the label (bytes eight and nine) This would be either the last executable instruction encountered before another label or the end of

the program., and 5.) the label name.

In the event a label is referenced by a GO or PERFORM statement before the label actually appears in the code, the symbol table entry performs a different function than just indicating the beginning and ending of the paragraph associated with the label. The same symbol table fields are used, however their meanings are different. The type is set to the unresolved label type of (ØFFH). The label remains unresolved until the beginning and the ending addresses of the associated paragraph are determined.

If a label is referenced for the first time by a GO statement the symbol table is initialized with the following: 1.) unresolved label type (ØFFH), 2.) the address of the GO statement (the intermediate code would be BRN ØØ ØØ where the zeros indicate where the address of the label is to be backstuffed). See section III-D for specific explanation of pseudo-machine instructions., 3.) the remainder of the label entries would be the same except no entry is made for the last executable instruction associated with the label. If an additional reference is made to the label by a subsequent GO statement the following action would occur: 1.) the current address (bytes four and five) would be placed in the branch address of the GO statement, 2.) the address of this branch statement would be placed in bytes four and five of the symbol table entry. This procedure facilitates linking together all unresolved references to labels so that when the label are resolved the

correct branch address could be placed into the intermediate code.

Encountering a PERFORM statement before a label is declared causes the following actions: 1.)bytes four and five contain the address of the next byte of intermediate code following the PER intermediate code instruction, 2.)and bytes eight and nine contain the address of the third byte following the PER instruction. If a subsequent PERFORM statement is encounted before the label is resolved the two address fields in the symbol table would be copied to the associated bytes following the most current PERFORM statement and the address of the first and third bytes following the PER instruction would be copied into the symbol table. It should be pointed out that any number of PERFORM and GO statements can be specified before a label is resolved.

7. Files

The symbol table entries for files are the most difficult to understand. The complexity of the entries is due to the way files and records are declared in a MICRO-COBOL program. The symbol table entry for a file consists of the following: 1.) byte two contains the type, 2.) byte three contains the length of the file name, 3.) bytes four and five contain the address in the symbol table of the first Ø1 level record associated with the file, 4.) bytes eight and nine contain the beginning address of the

file control block and input/output buffer for the file, (this would be the actual address in the data section of the pseudo-machine for the beginning of the 165 bytes associated with the file), 5.) if the file has a key entry associated with it (access via RANDOM or RANDOM RELATIVE) bytes ten and eleven contain the symbol table address of the access key variable, and 6.) the rest of the entry contains the file name. Figure [II-6] illustrates a file entry in the symbol table.

8. Records

This entry contains seven attributes of a record. Three are the same as all other entries type, name, and length of name. While the other four are: 1.) bytes four and five contain the initial storage address for the record, 2.) bytes six and seven contain the number of bytes of storage for the record, 3.) bytes eight and nine contain the symbol table address of the file associated with the record (this facilitates referencing the file when the record is written), and 4.) byte ten contains the level entry for the record.

FILE SYMBOL TABLE ENTRY

SAMPLE SOURCE PROGRAM FILE DECLARATION

```
INPUT-OUTPUT SECTION.
FILE-CONTROL.
      SELECT ROSTER-FIL
             ORGANIZATION RELATIVE
             ACCESS RANDOM RELATIVE NUM
             ASSIGN CS81-FIL.
```

BYTE	SYMBOL TABLE VALUE
Ø-1	collison link
2	type file (Ø3)
3	length of file name (Ø5)
4-5	symbol table address of first Ø1 level record (Ø9 2E)
6-7	not used
8-9	first address of FCB & buffer (ØE 26)
1Ø-11	symbol table address of key (33 27)
12	not used
13-17	file name (52 4F 53 54 45 52 5F 46 49 4C)

FIGURE II-6

C. COMPILER MODULE 'PART ONE"

1. Purpose

This first module of the compiler performs several functions. First, it establishes the interface between the compiler and: 1.) the input source file (of type 'CBL"), 2.) the output intermediate code file (of type "CIN"), and 3.) the IREADER module which reads and passes control to PART TWO of the compiler. Second, it scans and parses the source program statements up to the PROCEDURE DIVISION. Third, it generates output consisting of the symbol table entries (saved in memory) and data initialization intermediate code.

2. Control Actions

By executing the command COBOL <source program>, the object code for PART ONE of the compiler is loaded into memory starting at 100H (if necessary this can be modified for different machines) by the CP/M operating system. Execution of PART ONE loads the program name associated with the source program into the input file control block located at 5CH. This allows the source program name to be saved until actual source program compilation begins.

Next, the control program, IREADER, is moved to high memory just below the BDOS (see reference 4 for an explanation of BDOS and other CP/M associated names). For example, using an INTEL Corporation 62K MDS microcomputer system with the CP/M operating system, the IREADER routine

is moved to high memory starting at ØDØØØH and continuing through ØDØFFH. This is done for two reasons: 1.) it allows the symbol table of the source program to begin at the next address following the object code for PART ONE, and 2.) places IREADER high enough in memory so that it is not destroyed by creation of the symbol table. See figures [II-7] and [II-8] for illustrations of the PART ONE memory organization before and after the IREADER routine is moved. The purpose of the IREADER routine will be explained in the next section.

MEMORY ORGANIZATION BEFORE IREADER ROUTINE MOVED

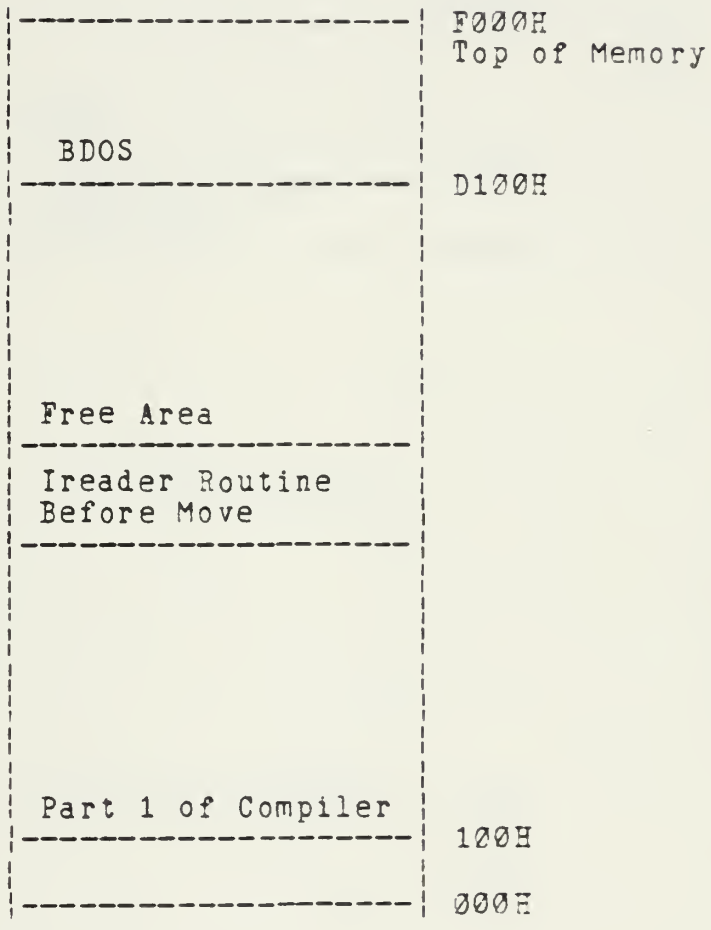

FIGURE II-7

MEMORY ORGANIZATION AFTER IREADER ROUTINE MOVED

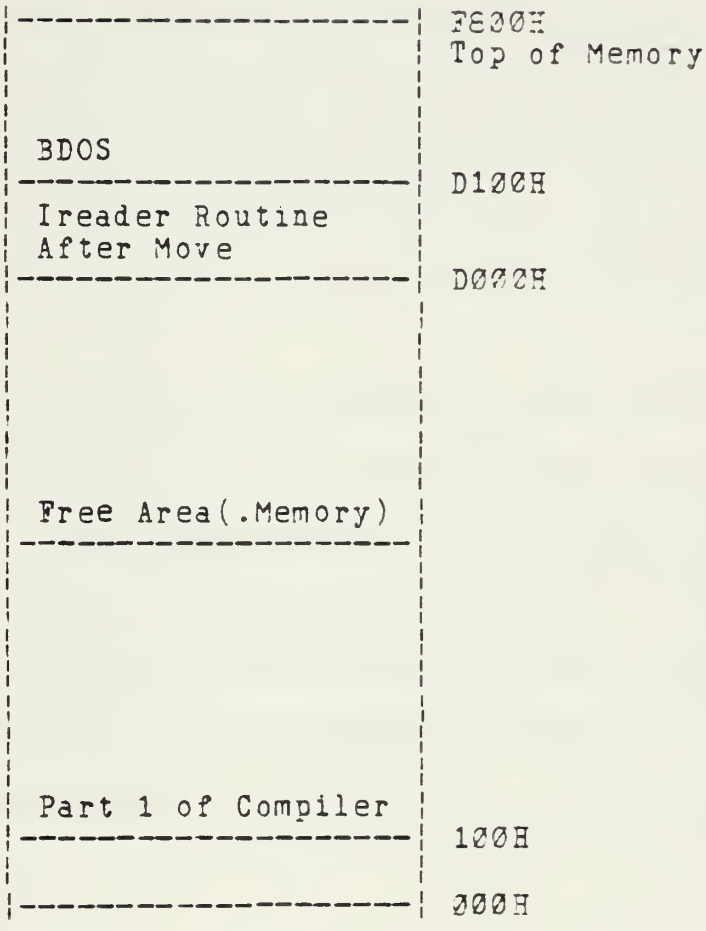

FIGURE II-8

Next, the interface between the compiler and the input file <source program> and the output file <intermediate code file> is established. The input file control block associated with the source file is initialized and the input file is opened. The input file name is copied to the output file control block (FCB) and if there is an intermediate code file already residing on the disk, it is erased. The output FCB is initialized and a file directory entry established for the new copy of the intermediate code file.

Prior to beginning the scanning and parsing actions, the first 128 byte record of the input file is read into the input buffer, located at 80H (default I/O buffer for CP/M). The scanner is primed with the first character of the input program, and scanning and parsing actions continue from this point in PART ONE until the PROCEDURE DIVISION of the source program is encountered; at this time compilation is suspended.

3. Symbol Table Entries

Entries made in the symbol table by PART ONE will consist of all identifiers declared in the DATA DIVISION of the source program. By refering to the Symbol Table Section above, an explanation may be obtained regarding the various types of symbol table entries.

4. Intermediate Code Generation

Pseudo-instructions are written to the intermediate code file for several different reasons while PART ONE is scanning and parsing the source program. The first intermediate code generated occurs when the INPUT-OUTPUT SECTION of a source program is nonempty. Within the FILE CONTROL PARAGRAPH of this section, instructions are generated to initialize the FCB for the file name associated with the SELECT statement. The name associated with the ASSIGN statement is placed in the FCB and is used in referencing the file on the disk.

Two other instances of intermediate code generation occur in the WORKING STORAGE SECTION of a source program. Anytime a record or elementary identifier entry has an edited PICTURE CLAUSE, code to initialize the storage beginning at the address specified in the formatted mask attribute of the symbol table entry will be written to the intermediate code file. When a record or elementary identifier entry has an associated numeric or nonnumeric VALUE CLAUSE, code to initialize the storage beginning at the address specified in the value location attribute of the symbol table entry will be written to the intermediate code file.

The final pseudo-instruction written to the intermediate code file is the SCD instruction. This occurs when the parser parses the word PROCEDURE in the source program; control is then passed to PART TWO and compilation

continues.

5. Parser Actions

The actions corresponding to each parse step are explained below. In each case, the grammar rule that is being applied is given, and an explanation of what program actions take place for that step has been included. In describing the actions taken for each parse step there has been no attempt to describe how the symbol table is constructed, what pseudo-instructions are generated or how the values are preserved on the stack. The intent of this section is to describe what information needs to be retained and at what point in the parse it can be determined. Where no action is required for a given statement, or where the only action is to save the contents of the top of the stack, no explanation is given. Questions regarding the actual manipulation of information should be resolved by consulting the programs.

1 <program> ::= <id-div> <e-div> <d-div> PROCEDURE

Reading the word PROCEDURE terminates the first part of the compiler.

2 <id-div> ::= IDENTIFICATION DIVISION. PROGRAM-ID.
<comment> . <auth> <date> <sec>

3 <auth> ::= AUTHOR . <comment> .

4 | <empty>

5 <date> ::= DATE-WRITTEN . <comment> .

6 | <empty>

```
7  <sec> ::= SECURITY . <comment> .
8          | <empty>
9  <comment> ::= <input>
10              | <comment> <input>
11  <e-div> ::= ENVIRONMENT DIVISION . CONFIGURATION
                SECTION.
                <scr-obj> <i-o>
12  <src-obj> ::= SOURCE-COMPUTER . <comment> <debug> .
                  OBJECT-COMPUTER . <comment> .
13  <debug> ::= DEBUGGING MODE
      Set a scanner toggle so that debug lines will be
      read.
14            | <empty>
15  <i-o> ::= INPUT-OUTPUT SECTION . FILE-CONTROL .
              <file-control-list> <ic>
16          | <empty>
17  <file-control-list> ::= <file-control-entry>
                          | <file-control-list>
                              <file-control-entry>
19  <file-control-entry> ::= SELECT <id> <attribute-list>.
      At this point all of the information about the file
      has been collected and the type of the file can be
      determined. File attributes are checked for
      compatability and entered in the symbol table.
20  <attribute-list> ::= <one attrib>
21                     | <attribute-list> <one attrib>
22  <one-attrib> ::= ORGANIZATION <org-type>
```

23 | ACCESS <acc-type> <relative>

24 | ASSIGN <input>

A file control block is built for the file using the INT operator.

25 <org-type> ::= SEQUENTIAL

No information needs to be stored since the default file organization is sequential.

26 | RELATIVE

The relative attribute is saved for production 19.

27 <acc-type> ::= SEQUENTIAL

This is the default.

28 | RANDOM

The random access mode is saved for production 19.

29 <relative> ::= RELATIVE <id>

The pointer to the identifier will be retained by the current symbol pointer, so this production only saves a flag on the value stack indicating that the production did occur.

30 | <empty>

31 <ic> ::= I-O-CONTROL . <same-list>

32 | <empty>

33 <same-list> ::= <same-element>

34 | <same-list> <same-element>

35 <same-element> ::= SAME <id-string> .

36 <id-string> ::= <id>

37 | <id-string> <id>

38 <d-div> ::= DATA DIVISION . <file-section> <work>

<link>

39 <file-section> ::= FILE SECTION . <file-list>

A flag needs to be set to indicate completion of the file section, so that the appropriate routine will be called when parsing level entries in the WORKING STORAGE SECTION.

40 | <empty>

The flag, indicated in production 39, is set.

41 <file-list> ::= <file-element>

42 | <file-list> <file-element>

43 <files> ::= FD <id> <file-control> .
<record-description>

This statement indicates the end of a record description, if there was an implied redefinition of the record, then the level stack (ID$STACK) must be reduced. The length of the first record description and its address can now be loaded into the symbol table for the file name.

44 <file-control> ::= <file-list>

The address of the symbol table entry for the record describing the file name is entered into an attribute of the file name symbol table entry, while the address of the file names symbol table entry is entered into an attribute of the same record.

45 | <empty>

Same as 44 above.

```
46 <file-list> ::= <file-element>
47                | <file-list> <file-element>
48 <file-element> ::= BLOCK <integer> RECORDS
49                   | RECORD <rec-count>
```

The record length is saved for comparison with the calculated length from the picture clauses.

```
50                   | LABEL RECORDS STANDARD
51                   | LABEL RECORDS OMITTED
52                   | VALUE OF <id-string>
53 <rec-count> ::= <integer>
54                | <integer> TO <integer>
```

The TO option is the only indication that the file will be variable length. The maximum length must be saved.

```
55 <work> ::= WORKING-STORAGE SECTION . <record-description>
```

If the level stack (ID$STACK) contains a record identifier with a level number greater than one, then the stack must be reduced. The reduction depends on whether the identifier on the top of the stack is a redefinition of the item beneath it or not. The primary action is to assign the proper amount of storage to the last record in the WORKING STORAGE SECTION.

```
56           | <empty>
57 <link> ::= LINKAGE SECTION . <record-description>
58           | <empty>
59 <record-description> ::= <level-entry>
```

60 | <record-description><level-entry>

61 <level-entry> ::= <integer> <data-id> <redefines>
<data-type> .

The symbol table address for the level entry identifier is loaded into the level stack (ID$STACK). The level stack keeps track of the nesting of field definitions (elementary items) in a record in the FILE and WORKING STORAGE SECTIONS. At this point there may be no information about the length of the item being defined and its attributes may depend entirely upon its constituent fields. Within the FILE SECTION, multiple record descriptions for a file are assumed to be redefinitions of the first record description. In the WORKING STORAGE SECTION, if there is a VALUE CLAUSE, the stack level to which it applies is saved in PENDING$LITERAL, the level entry number is saved in VALUE$LEVEL and a flag, VALUE$FLAG, is set.

62 <data-id> ::= <id>

63 | FILLER

An entry is built in the symbol table to record information about this record field. It cannot be used explicitly in a program because it has no name, but its attributes will need to be stored as part of the total record.

64 <redefines> ::= REDEFINES <id>

The redefines option gives new attributes to a previously defined record area. The symbol table pointer to the area being redefined is saved in an attribute of the redefining identifier's symbol table entry, so that information can be transferred to the area by either identifier. In addition to the information saved relative to the redefinition, it is necessary to check to see if the current identifier's level number is less than or equal to the level number of the identifier currently on the top of the level stack. If this is true, then all information for the item on top of the stack has been saved and the stack can be reduced. If the current identifier is a redefinition of another identifier, the stack entry for the record being redefined is not removed until the first non-redefinition of a current identifier at the same level.

65 | <empty>

As in production 64, the stack (ID$STACK) is checked to determine if the current level number indicates a reduction of the level stack is necessary. In addition, special action needs to be taken if the new level is Ø1. If an Ø1 level is encountered at this production prior to production 39 or 4Ø (the end of the file area), it is an implied redefinition of the previous Ø1 level record. In the WORKING STORAGE SECTION, it indicates the start of a new record.

66 <data-type> ::= <prop-list>

67 | <empty>

68 <prop-list> ::= <data-element>

69 | <prop-list> <data-element>

70 <data-element> ::= PIC <input>

The <input> at this point is the character string that defines record field. It is analyzed and the necessary extracted information is stored in the symbol table.

71 | USAGE COMP

The field is defined to a binary field; however, COMP has not been implemented, therefore, if there is an associated VALUE CLAUSE, the value is entered into the associated identifier's value storage location in display format.

72 | USAGE DISPLAY

The DISPLAY format is the default, and thus no special action occurs.

73 | SIGN LEADING <separate>

This production indicates the presence of a sign in a numeric field. The sign will be in a leading position. If the <separate> indicator is true, then the length will be one longer than the PICTURE CLAUSE, and the type will be changed to signed numeric leading and separate.

74 | SIGN TRAILING <separate>

The same information required by production 73 must

be recorded, but in this case the sign is trailing rather than leading.

75 | OCCURS <integer>

The type must be set to indicate multiple occurrences and the number of occurrences saved for computing the space defined by this field.

76 | SYNC <direction>

Syncronization with a natural boundary is not required by this machine.

77 | VALUE <literal>

The field being defined will be assigned an initial value determined by the value of the literal through the use of an INT operator. This is only valid in the WORKING-STORAGE SECTION. Note that numeric and signed numeric PICTURE CLAUSES will have a numeric -- no quotes delimiting -- VALUE CLAUSE, while alphanumeric and alpha types will have a nonnumeric -- literal delimited with quotes -- VALUE CLAUSE.

78 <direction> ::= LEFT

79 | RIGHT

80 | <empty>

81 <separate> ::= SEPARATE

The separate sign indicator is set.

82 | <empty>

83 <literal> ::= <input>

The input string is checked to see if it is a valid numeric literal, and if valid, it is stored to be

used in a value assignment.

84 ¦ <lit>

This literal is a quoted string.

85 ¦ ZERO

As the case of all literals, the fact that there is a pending literal needs to be saved. In this case and the three following cases, an indicator of which literal constant is being saved is all that is required. The literal value can be reconstructed later.

86 ¦ SPACE

87 ¦ QUOTE

88 <integer> ::= <input>

The input string is converted to an integer value for later internal use.

89 <id> ::= <input>

The input string is the name of an identifier and is checked aginst the symbol table. If it is in the symbol table, then a pointer to the entry is saved. If it is not in the symbol table, then it is entered and the address of the entry is saved.

D. INTERFACE ACTIONS

When compilation is suspended in PART ONE of the compiler certain key variables are saved for use in PART TWO. These variables are declared sequentially in PART ONE and are therefore located in contiguous memory in the

variable area of PART ONE. These variables consist of debugging toggles set when invoking the compiler , i.e. sequence or token numbers, a pointer to the next available address in the symbol table, a pointer to the next character in the input source file, the output file control block, the next address in the intermediate code area, the next address in the constants area, and the base address of the symbol table. These key variables, consisting of 48 bytes, are copied to the 48 bytes immediately below the IREADER routine to insure they are not destroyed when PART TWO of the compiler is brought into memory. Since the memory area required for PART ONE is larger than that required by PART TWO the symbol table does not need to be relocated. Since the symbol table is not altered when PART TWO of the compiler is brought into memory only the base address of the symbol table and the last address of the symbol table need be saved to insure that access to the symbol table can be continued in PART TWO. See Figure [II-9] for an illustration of the memory organization when control is transfered from PART ONE to IREADER. The IREADER rountine causes PART TWO of the compiler to be brought into memory starting at 100H and then transfers control to PART TWO of the Compiler.

E. COMPILER MODULE "PART TWO"

1. Purpose

The second part of the compiler scans and parses the

MICRO-COBOL source statements starting with the PROCEDURE DIVISION and generates the necessary intermediate code.

2. Control Actions

The first action after control is transfered to PART TWO from the IREADER routine is to copy the 48 bytes of the information saved from PART ONE into associated variables in PART TWO. After these variables are initialized all references to files, symbol table entries, etc. can be made in PART TWO and compilation can continue. See Figure [II-10] for an illustration of the memory organization at the time PART TWO begins compilation.

3. Symbol Table Entries

Entries made in the symbol table by PART TWO will be those for paragraph labels encountered within the PROCEDURE DIVISION of the source program.

4. Intermediate Code Generation

For an explanation of the pseudo-instructions that are generated by PART TWO refer to the compiler program listings and the parser actions below. Also, for general information on pseudo-instructions refer to section III-D.

MEMORY ORGANIZATION WHEN CONTROL IS TRANSFERED TO IREADER

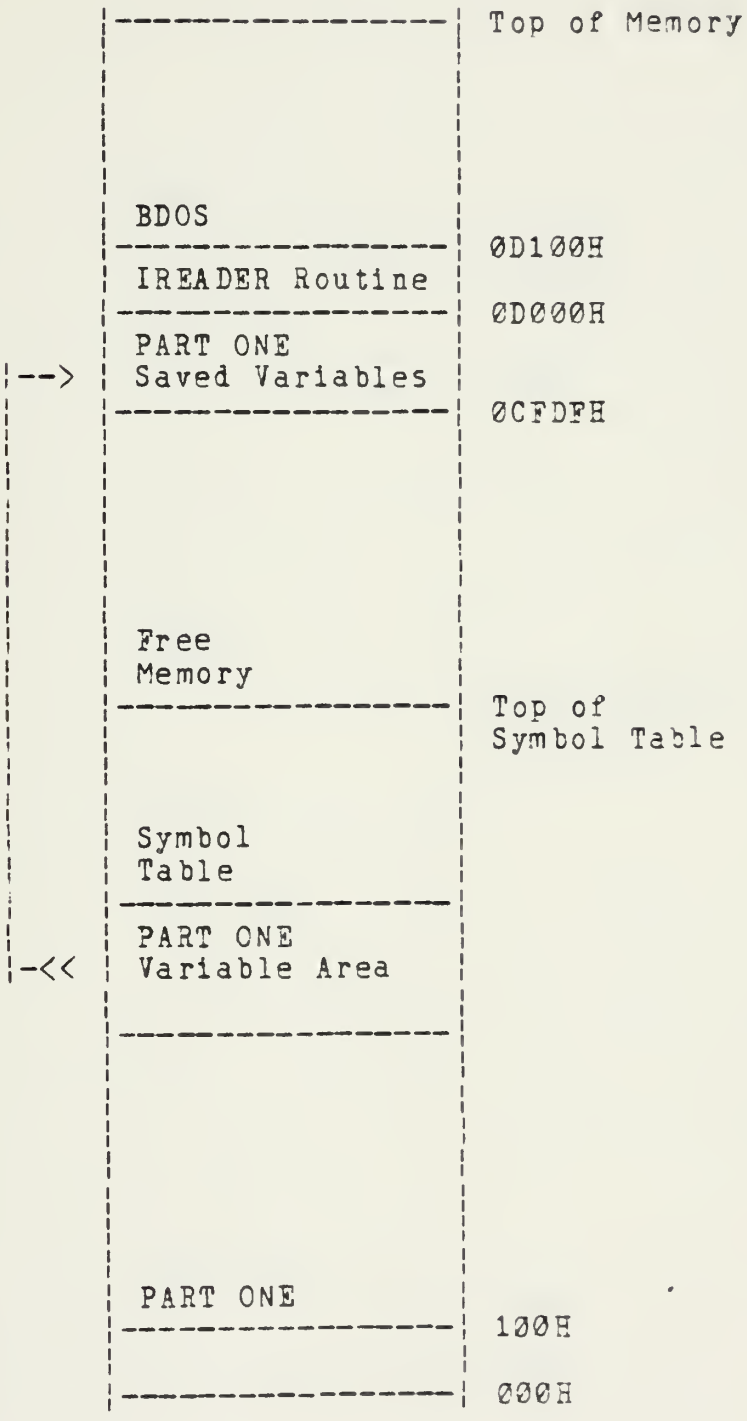

FIGURE II-9

MEMORY ORGANIZATION AFTER PART TWO IS COPIED INTO MEMORY

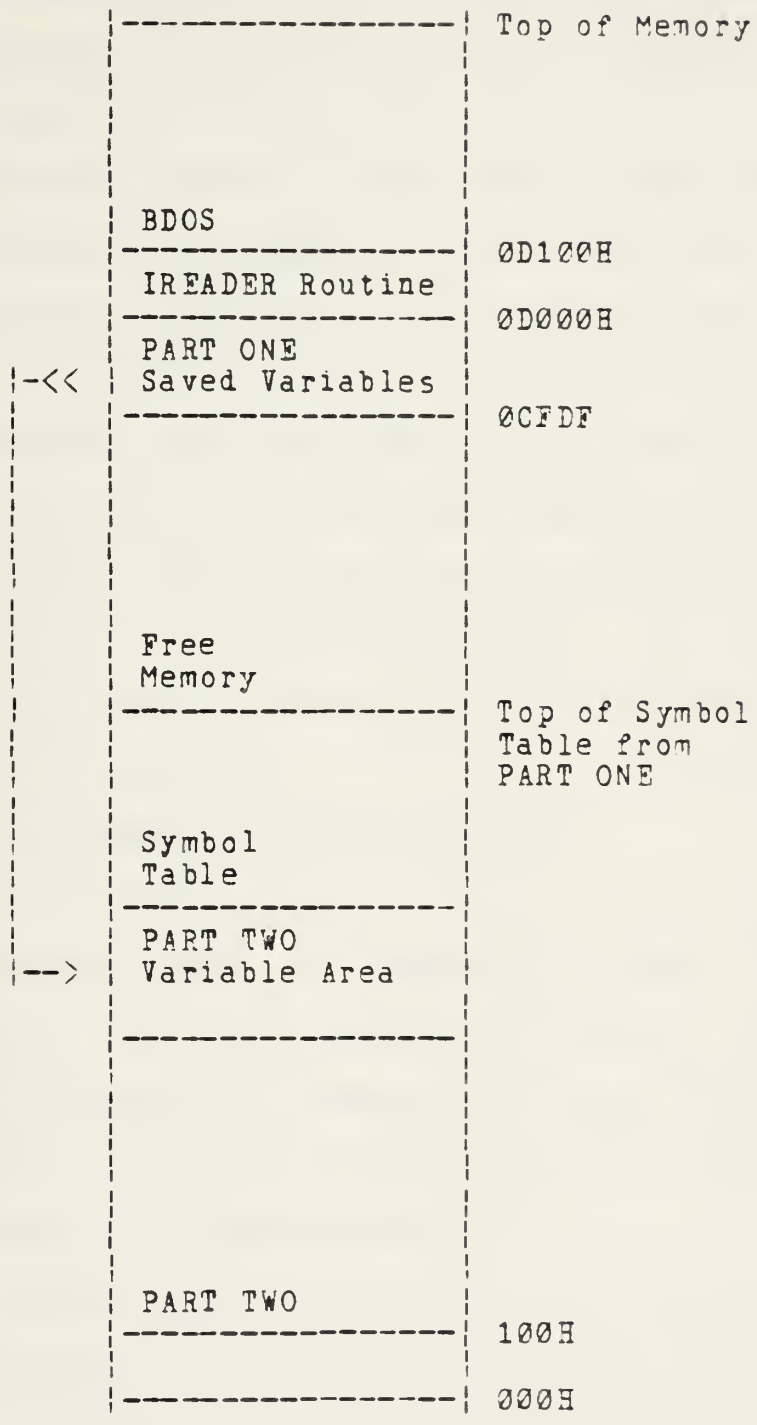

FIGURE II-10

5. Parser Actions

The actions corresponding to each parse step in PART TWO are explained below. In each case, the grammar action that is being applied is given, and an explanation of what program actions take place for that step has been included. In describing the actions taken for each parse step there has been no attempt to describe how the symbol table entries are made, what pseudo instructions are generated or how the values are preserved on the stack. The intent of this section is to describe what information needs to be retained and at what point in the parse it can be determined. Where no action is required for a given statement, or where the only action is to save the contents of the top of the stack, no explanation is given.

1 <p-div> ::= PROCEDURE DIVISION <using> .
<proc-body> EOF

This production indicates termination of the compilation. If the program has sections, then it will be necessary to terminate the last section with a RET Ø instruction. The code will be ended by the output of a TER operation.

2 <using> ::= USING <id-string>

Not implemented.

3 | <empty>

4 <id-string> ::= <id>

The identifier stack is cleared and the symbol table address of the identifier is loaded into

the first stack location.

5 | <id-string> <id>

The identifier stack is incremented and the symbol table pointer stacked.

6 <proc-body> ::= <paragraph>

7 | <proc-body> <paragraph>

8 <paragraph> ::= <id> . <sentence-list>

The starting and ending address of the paragraph are entered into the symbol table. A return is emitted as the last instruction in the paragraph (RET Ø). When the label is resolved, it may be necessary to produce a BST operation to resolve previous references to the label.

9 | <id> SECTION .

The starting address for the section is saved. If it is not the first, then the previous section ending address is loaded and a return (RET Ø) is output. As in production 8, a BST may be produced.

10 <sentence-list> ::= <sentence>.

11 | <sentence-list> <sentence> .

12 <sentence> ::= <imperative>

13 | <conditional>

14 | ENTER <id> <opt-id>

This construct is not implemented. An ENTER allows statements from another language to inserted in the source code.

15 <imperative> ::= ACCEPT <subid>

ACC <address> <length>

16 | <arithmetic>

17 | CALL <lit> <using>

This is not implemented.

18 | CLOSE <id>

CLS <file control block address>

19 | <file-act>

20 | DISPLAY <lit/id> <opt-lit/id>

The display operator is produced for the first literal or identifier (DIS <address> <length> <flag>). If the second value exists, the same code is also produced for it. The only difference in the two display outputs is the flag is set to zero on the first display to surpress the carriage return and line feed.

.

21 | EXIT <program-id>

RET Ø

22 | GO <id>

BRN <address>

23 | GO <id-string> DEPENDING <id>

GDP is output, followed by a number of parameters: <the number of entries in the identifier stack> <the length of the depending identifier> <the address of the depending identifier> <the address of each identifier in the stack>.

24 | MOVE <lit/id> TO <subid>

The types of the two fields determine the move that is generated. Numeric moves go through register two using a load and a store. Non-numeric moves depend upon the result field and may be either MOV, MED or MNE. Since all of these instructions have long parameter lists, they have not been listed in detail.

25 | OPEN <type-action> <id>

This produces either OPN, OP1, or OP2 depending upon the <type-action>. Each of these is followed by file control block address.

26 | PERFORM <id> <thru> <finish>

The PER operation is generated followed by the <branch address> <the address of the return statement to be set> and <the next instruction address>.

27 | <read-id>

28 | STOP <terminate>

If there is a terminate message, then STD is produced followed by <message address> <message length>. Otherwise STP is emitted.

29 <conditional> ::= <arithmetic> <size-error> <imperative>

A BST operator is output to complete the branch around the imperative from production 65.

30 | <file-act> <invalid> <imperative>

A BST operator is output to complete the branch from

production 64.

31 | <if-nonterminal> <condition> <action>
ELSE <imperative>

NEG will be emmitted unless <condition> is a 'NOT <cond-type>", in which case the two negatives will cancel each other. Two BST operators are required. The first fills in the branch to the ELSE action. The second completes the branch around the <imperative> which follows ELSE.

32 | <read-id> <special> <imperative>

A BST is produced to complete the branch around the <imperative>.

33 <Arithmetic> ::= ADD <1/id> <opt-1/id> TO <subid>
<round>

The existence of multiple load and store instructions make it difficult to indicate exactly what code will be generated for any of the arithmetic instructions. The type of load and store will depend on the nature of the number involved, and in each case the standard parameters will be produced. This parse step will involve the following actions: first, a load will be emitted for the first number into regster zero. If there is a second number, then a load into register one will be produced for it, followed by an ADD and a STI. Next a load into register one will be generated for the result number. Then an ADD instruction will be emitted. Finally, if the round indicator is set, a

RND operator will be produced prior to the store.

34 | DIVIDE <1/id> INTO <subid> <round>

The first number is loaded into register zero. The second operand is loaded into register one. A DIV operator is generated, followed by a RND operator prior to the store, if required.

35 | MULTIPLY <1/id> BY <subid> <round>

The multiply is the same as the divide except that a MUL operator is generated.

36 | SUBTRACT <1/id> <opt-1/id> FROM <subid> <round>

Subtaction generates the same code as the ADD except that a SUB is produced in place of the last ADD.

37 <file-act> ::= DELETE <id>

Either a DLS or a DLR will be produced along with the required parameters.

38 | REWRITE <id>

Either a RWS or a RWR is emitted, followed by parameters.

39 | WRITE <id> <special-act>

There are four possible write instructions: WTF, WV1, WRS, and WRR.

40 <condition> ::= <lit> <not> <cond-type>

One of the compare instructions is produced. They are CAL, CNS, CNU, RGT, RLT, REQ, SGT, SLT, and SEQ. Two load instructions and a SUB will also be generated if one of the register comparisons is required.

41 <cond-type> ::= NUMERIC

42 | ALPHABETIC

43 | <compare> <lit/id>

44 <not> ::= NOT

NEG is emitted unless the NOT is part of an IF statement in which case the NEG in the IF statement is cancelled.

45 | <empty>

46 <compare> ::= GREATER

47 | LESS

48 | EQUAL

49 <ROUND> ::= ROUNDED

50 | <empty>

51 <terminate> ::= <literal>

52 | RUN

53 <special> ::= <invalid>

54 | END

An ERO operator is emitted followed by a zero. The zero acts as a filler in the code and will be back-stuffed with a branch address. In this production and several of the following, there is a forward branch on a false condition past an imperative action. For an example of the resolution, examine production 32.

55 <opt-id> ::= <subid>

56 | <empty>

57 <action> ::= <imperative>

BRN Ø

```
58              | NEXT SENTENCE
      BRN Ø
59  <thru> ::= THRU <id>
60           | <empty>
61  <finish> ::= <l/id> TIMES
      LDI <address> <length> DEC Ø
62              | UNTIL <condition>
63              | <empty>
64  <invalid> ::= INVALID
      INV Ø
65  <size-error> ::= SIZE ERROR
      SER Ø
66  <special-act> ::= <when> ADVANCING <how-many>
67                  | <empty>
68  <when> ::= BEFORE
69           | AFTER
70  <how-many>::= <integer>
71               | PAGE
74                   | I-O
75  <subid> ::= <subscript>
76            | <id>
77  <integer> ::= <input>
      The value of the input string is saved as an internal
      number.
78  <id> ::= <input>
      The identifier is checked aginst the symbol table, if
      it  is not present, it is entered as an unresolved
```

label.

79 <l/id> ::= <input>

The input value may be a numeric literal. If so, it is placed in the constant area with an INT operand. If it is not a numeric literal, then it must be an identifier, and it is located in the symbol table.

80 | <subscript>

81 | ZERO

82 <subscript> ::= <id> (<input>)

If the identifier was defined with a USING option, then the input string is checked to see if it is a number or an identifier. If it is an identifier, then an SCR operator is produced.

83 <opt-l/id> ::= <l/id>

84 | <empty>

85 <nn-lit> ::= <lit>

The literal string is placed into the constant area using an INT operator.

86 | SPACE

87 | QUOTE

88 <literal> ::= <nn-lit>

89 | <input>

The input value must be a numeric literal to be valid and is loaded into the constant area using an INT.

90 | ZERO

91 <lit/id> ::= <l/id>

92 | <nn-lit>

```
93  <opt-lit/id> ::= <lit/id>
94                 | <empty>
95  <program-id> ::= <id>
96                 | <empty>
97  <read-id> ::= READ <id>
```

There are four read operations: RDF, RVL, RRS, and RRP.

```
98   <if-nonterminal>::=IF
```

III. NPS MICRO-COBOL INTERPRETER

A. GENERAL DESCRIPTION

The following sections describe the NPS MICRO-COBOL pseudo-machine in terms of the implementation, memory organization, interface actions and interpreter instructions. The pseudo-machine, which is constructed in the transient program area of CP/M, is the target machine for the compiler and is implemented through a programmed interpreter. The interpreter decodes each operation and either calls subroutines to perform the required actions or acts directly on the run time environment to control the actions of the interpreter. All communications between instructions is done through common areas in the program where information can be stored for later use. See figure [III-1] for an illustration of the pseudo-machine organization.

The machine contains a program counter and multiple parameter operations which contain all the information required to perform one complete action required by the language. Three eighteen digit registers are used for arithmetic operations, along with a subscript stack used to compute subscript locations, and a set of flags are used to pass branching information from one instruction to another.

Addresses in the pseudo-machine are represented by 16 bit values. Any memory address greater than 20 hexidecimal

is valid. Addresses less than 20 hexidecimal will be interpreted as having special significance. For example addresses one through eight are reserved for subscript stack references. All other addresses in the machine are absolute addresses.

The registers allow manipulation of signed numbers up to eighteen digits in length. Included in their representation is a sign indicator and the position of the assumed decimal point for the currently loaded number. Numbers are represented in standard COBOL "Display" format. These numbers may have separate signs indicated by "+" and "-" or may have a "zone" indicator, denoting a negative sign, in the most significant byte of a number's storage location. Before operations occur on any number, it is converted to a packed decimal format and entered into one of the pseudo-machine registers.

B. MEMORY ORGANIZATION

The memory of the pseudo-machine is divided into three major areas: 1.) the data area is established by the DATA DIVISION statements of the source program, 2.) the constants area which is established by both the DATA and PROCEDURE DIVISIONS of the source program, and 3.) the code area which is established by the PROCEDURE DIVISION.

The data area is the lowest area in the pseudo-machine. This area contains the storage for identifiers declared in the DATA DIVISION. Additionally, the data area contains the

File Control Block (FCB) and the buffer space (128 bytes) for all files declared in the source program.

Immediately following the data area is the code area. This contiguous area of storage contains all executable code generated. The constants area is located in high memory of the pseudo-machine. This area contains all edit field masks as well as all numeric and non-numeric literals. Figure [III-1] ilustrates the memory organization of the pseudo-machine.

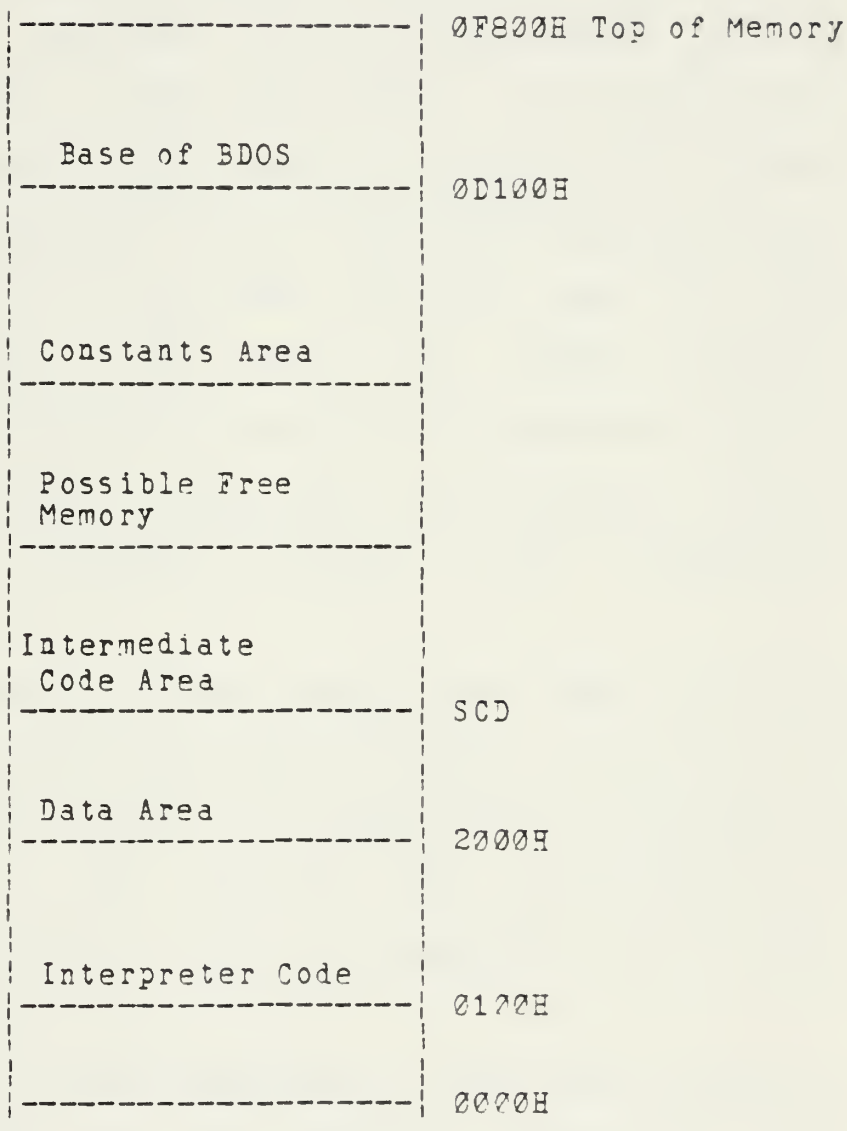
0F800H Top of Memory
Base of BDOS
0D100H
Constants Area
Possible Free Memory
Intermediate Code Area
SCD
Data Area
2000H
Interpreter Code
0100H
0000H

FIGURE III-1

C. INTERPRETER INTERFACE

The interpreter consists of two interface routines and the main interpreter program. To execute the interpreter the command EXEC <filename.filetype>, (where file type is CIN), is typed at the terminal. This action causes the two interface routines, BUILD and INTRDR, to be brought into memory. See figure [III-2] which illustrates the memory organization immediately after BUILD and INTRDR have been copied into memory. The BUILD routine reads in the intermediate code, initializes all memory locations requiring initalization, and resolves all unresolved address references. The INTRDR routine reads the interpreter program into memory and transfers control to the interpreter program.

The intermediate code instructions fall into two categories: 1.) instructions used by BUILD to establish the run time environment and, 2.) instructions to be executed by the interpreter. The following four instructions are generated in the compiler for use by the BUILD routine; SCD, INT, BST, and TER.

The SCD (start code) instruction is the last instruction generated by PART ONE and indicates where the first executable instruction for the intermediate code is to be loaded. This corresponds to the address immediately following the data area in the pseudo-machine. See Figure [III-1] which illustrates the relative location of the address that is associated with the SCD instruction.

MEMORY ORGANIZATION AFTER BUILD AND INTRDR

HAVE BEEN LOADED INTO MEMORY

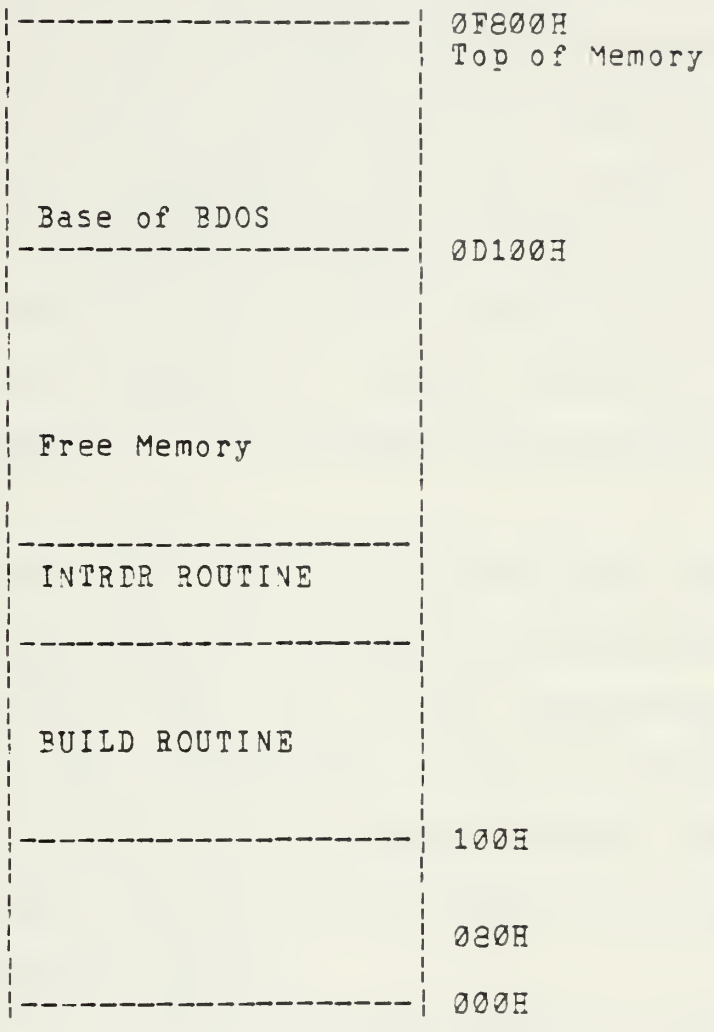

FIGURE III-2

The INT (initialize) instruction causes the BUILD routine to initialize the data area with the values associated with those identifiers in the DATA DIVISION of the source program that had VALUE CLAUSES. In addition, the INT instruction causes the BUILD routine to initialize the constants area with all the edit masks for those identifiers of the numeric and alphanumeric edit type, and all literals encounted in the PROCEDURE DIVISION of the source program.

The BST (backstuff) instruction resolves all unresolved references, i.e. branches to labels defined after the respective PERFORM or GO statement was encountered in the source program.

The TER (terminate) instruction is the last instruction generated by PART TWO of the compiler and indicates the end of the intermediate code file. Upon encountering a TER instruction in the intermediate code the BUILD routine inserts a STP instruction in its place. The STP instruction will cause the interpreter to terminate interpretation of the program when encountered.

All other code generated by the compiler is copied into the code area of the pseudo-machine by the BUILD routine. See Figure [III-3] for an illustration of the memory organization at this point in the initialization routine. The final action taken by the BUILD routine is to move the INTRDR routine into the input buffer at 80H and transfer control to INTRDR. This frees the area from 100H to the base of the data area for the interpreter.

The INTRDR routine reads the interpreter program into memory starting at 100H and transfers contol to it. From this point on the interpreter program executes the intermediate code that was loaded into the pseudo-machine.

MEMORY ORGANIZATION AFTER INTERMEDIATE CODE IS LOADED INTO MEMORY AND BEFORE THE INTERPRETER IS LOADED

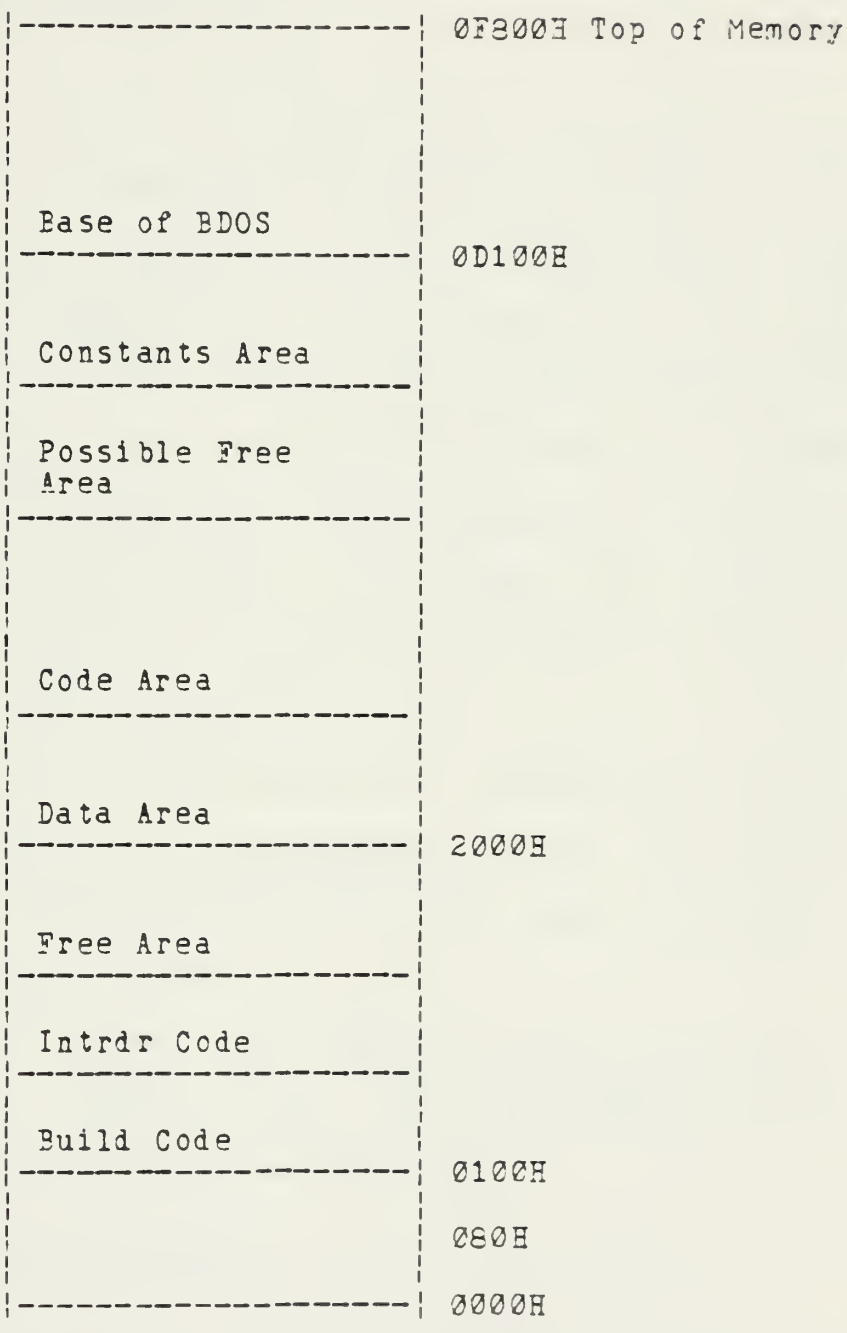

FIGURE III-3

D. PSEUDO-MACHINE INSTRUCTIONS

This section briefly covers the pseudo-machine instructions used in the interpreter, their format, and the actions which they accomplish.

1. Format

All of the interpreter instructions consist of an instruction number followed by a list of parameters. The following sections describe the instructions, list the required parameters, and describe the actions taken by the machine in executing each instruction. In each case, parameters are denoted informally by the parameter name enclosed in brackets. The BRN branching instruction, for example, uses the single parameter <branch address> which is the target of the unconditional branch.

As each instruction number is fetched from memory, the program counter is incremented by one. The program counter is then either incremented to the next instruction number, or a branch is taken.

The three eighteen digit registers which are used by the instructions covered in the following sections are referred to as registers zero, one, and two.

2. Arithmetic Operations

There are five arithmetic instructions which act upon the three registers. In all cases, the result is placed in register two. Operations are allowed to destroy

the input values during the process of creating a result, therefore, a number loaded into a register is not available for a subsequent operation.

ADD: (addition). Sum the contents of register zero and register one.

Parameters: no parameters are required.

SUB: (subtract). Subtract register zero from register one.

Parameters: no parameters are required.

MUL: (multiply). Multiply register zero by register one.

Parameters: no parameters are required.

DIV: (divide). Divide register one by the value in register zero. The remainder is not retained.

Parameters: no parameters are required

RND:(round). Round register two to the last significant decimal place.

Parameters: no parameters are required.

3. Branching

The machine contains the following flags which are used by the conditional instructions in this section.

BRANCH flag -- indicates if a branch is to be taken;

END OF RECORD flag -- indicates that an end of input condition has been reached when an attempt was made to read input;

OVERFLOW flag -- indicates the loss of information

from a register due to a number exceeding the available size;

INVALID flag -- indicates an invalid action in writing to a direct access storage device.

All of the branch instructions are executed by changing the value of the program counter. Some are unconditional branches and some test for condition flags which are set by other instructions. A conditional branch is executed by testing the branch flag which is initialized to false. A true value causes a branch by changing the program counter to the value of the branch address. The branch flag is then reset to false. A false value causes the program counter to be incremented to the next sequential instruction.

BRN: (branch to an address). Load the program counter with the <branch address>.

Parameters: <branch address>

The next three instructions share a common format. The memory field addressed by the <memory address> is checked for the <address length>, and if all the characters match the test condition, the branch flag is complemented

Parameters: <memory address> <address length> <branch address>

CAL: (compare alphabetic). Compare a memory field for alphabetic characters.

CNS: (compare numeric signed). Compare a field for numeric characters allowing for a sign character.

CNU: (compare numeric unsigned). Compare a field for numeric characters only.

DEC: (decrement a counter and branch if zero). decrement the value of the <address counter> by one; if the result is zero before or after the decrement, the program counter is set to the <branch address>. If the result is not zero, the program counter is incremented by four.

Parameters: <address counter> <branch address>

EOR: (branch on end of records flag). If the end-of-records flag is true, it is set to false and the program counter is set to the <branch address>. If false, the program counter is incremented by two.

Parameters: <branch address>

GDP: (go to - depending on). The memory location addressed by the <number address> is read for the number of bytes indicated by the <memory length>. This number indicates which of the <branch addresses> is to be used. The first parameter is a bound on the number of branch addresses. If the number is within the range, the program counter is set to the indicated address. An out-of-bounds value causes the program counter to be advanced to the next sequential instruction.

Parameters: <bound number - byte> <memory length> <memory address> <branch addr-1> <branch addr-2> ... <branch addr-n>

INV: (branch if invalid-file-action flag true). If the invalid-file-action flag is true, then it is set to false, and the program counter is set to the branch ad-

dress. If it is false, the program counter is incremented by two.

Parameters: <branch address>

PER: (perform). The code address addressed by the <change address> is loaded with the value of the <return address>. The program counter is then set to the <branch address>.

Parameters: <branch address> <change address> <return address>

RET: (return). If the value of the <branch address> is not zero, then the program counter is set to its value, and the <branch address> is set to zero. If the <branch address> is zero, the program counter is incremented by two.

Parameters: <branch address>

REQ: (register equal). This instruction checks for a zero value in register two. If it is zero, the branch flag is complemented. A conditional branch is taken.

Parameters: <branch address>

RGT: (register greater than). Register two is checked for a negative sign. If present, the branch flag is complemented. A conditional branch is taken.

Parameters: <branch address>

RLT: (register less than). Register two is checked for a positive sign, and if present, the branch flag is complemented. A conditional branch is taken.

Parameters: <branch address>

SER: (branch on size error). If the overflow flag is

true, then the program counter is set to the branch address, and the overflow flag is set to false. If it is false, then the program counter is incremented by two.

Parameters: <branch address>

The next three instructions are of similar form in that they compare two strings and set the branch flag if the condition is true.

Parameters: <string addr-1> <string addr-2> <length - address> <branch address>

SEQ: (strings equal). The condition is true if the strings are equal.

SGT: (string greater than). The condition is true if string one is greater than string two.

SLT: (string less than). The condition is true if string one is less than string two.

4. Moves

The machine supports a variety of move operations for various formats and types of data. It does not support direct moves of numeric data from one memory field to another Instead, all of the numeric moves go through the registers.

The next seven instructions all perform the same function. They load a register with a numeric value and differ only in the type of number that they expect to see in memory at the <number address>. All seven instructions cause the program counter to be incremented by five. Their

common format is given below.

Parameters: <number address> <byte length> <byte decimal count> <byte register to load>

LOD: (load literal). Register two is loaded with a constant value. The decimal point indicator is not set in this instruction. The literal will have an actual decimal point in the string if required.

LD1: (load numeric). Load a numeric field.

LD2: (load postfix numeric). Load a numeric field with an internal trailing sign.

LD3: (load prefix numeric). Load a numeric field with an internal leading sign.

LD4: (load separated postfix numeric). Load a numeric field with a separate leading sign.

LD5: (load separated prefix numeric). Load a numeric field with a separate trailing sign.

LD6: (load packed numeric). Load a packed numeric field.

MED: (move into alphanumeric edited field). The edit mask is loaded into the <to address> to set up the move, and then the <from address> information is loaded. The program counter is incremented by ten.

Parameters: <to address> <from address> <length of move> <edit mask address> <edit mask length>

MNE: (move into a numeric edited field). First the edit mask is loaded into the receiving field, and then the information is loaded. Any decimal point alignment required

will be performed. If truncation of significant digits is a side effect, the overflow flag is not set. The program counter is incremented by twelve.

Parameters: <to address> <from address> <address length of move> <edit mask address> <address mask length> <byte to decimal count> <byte from decimal count>

MOV: (move into an alphanumeric field). The memory field given by the <to address> is filled by the from field for the <move length> and then filled with blanks in the following positions for the <fill count>.

Parameters: <to address> <from address> <address move length> <address fill count>

STI: (store immediate register two). The contents of register two are stored into register zero and the decimal count and sign indicators are set.

Parameters: none.

The store instructions are grouped in the same order as the load instructions. Register two is stored into memory at the indicated location. Alignment is performed and any truncation of leading digits causes the overflow flag to be set. All five of the store instructions cause the program counter to be incremented by four. The format for these instructions is as follows.

Parameters: <address to store into> <byte length> <byte decimal count>

STO: (store numeric). Store into a numeric field.

ST1: (store postfix numeric). Store into a numeric

field with an internal trailing sign.

ST2: (store prefix numeric). Store into a numeric with an internal leading sign.

ST3: (store separated postfix numeric). Store into a numeric field with a separate trailing sign.

ST4: (store separated prefix numeric). Store into a numeric field with a separate leading sign.

ST5: (store packed numeric). Store into a packed numeric field.

5. Input-Output

The following instructions perform input and output operations. Files are defined as having the following characteristics: they are either sequential or random and, in general, files created in one mode are not required to be readable in the other mode. Standard files consist of fixed length records, and variable length files need not be readable in a random mode. Further, there must be some character or character string that delimits a variable length record.

ACC: (accept). Read from the system input device memory at the location given by the <memory address>. The program counter is incremented by three.

Parameters: <memory address> <byte length of read>

CLS: (close). Close the file whose file control block is addressed by the <fcb address>. The program counter is incremented by two.

Parameters: <fcb address>

DIS: (display). Print the contents of the data field pointed to by <memory address> on the system output device for the indicated length and advance the line output if <flag> is set. The program counter is incremented by four.

Parameters: <memory address> <byte length> <flag>

There are three open instructions with the same format. In each case, the file defined by the file control block referenced will be opened for the mode indicated. The program counter is incremented by two.

Parameters: <fcb address>

OPN: (open a file for input).

OP1: (open a file for output).

OP2: (open a file for both input and output). This is only valid for files on a random access device.

The following file actions all share the same format. Each performs a file action on the file referenced by the file control block. The record to be acted upon is given by the <record address>. The program counter is incremented by six.

Parameters: <FCB address> <record address> <record length - address>.

DLS: (delete a record from a sequential file). Remove the record that was just read from the file. The file is required to be open in the input-output mode.

RDF: (read a sequential file). Read the next record into the memory area.

WTF: (write a record to a sequential file). Append a new record to the file.

RVL: (read a variable length record).

WVL: (write a variable length record).

RWS: (rewrite sequential). The rewrite operation writes a record from memory to the file, overlaying the last record that was read from the device. The file must be open in the input-output mode.

The following file actions require random files rather than sequential files. They make use of a random file pointer which consists of a <relative address> and a <relative length>. The memory field holds the number to be used in disk operations or contains the relative record number of the last disk action. The relative record number is an index into the file which addresses the record being accessed. After the file action, the program counter is incremented by nine.

Parameters: <FCB address> <record address> <record length - address> <relative address> <relative length - byte>.

DLR: (delete a random record). Delete the record addressed by the relative record number.

RRR: (read random relative). Read a random record relative to the record number.

RRS: (read random sequential). Read the next sequential record from a random file. The relative record number of the record read is loaded into the memory reference.

RWR: (rewrite a random record).

WRR: (write random relative). Write a record into the area indicated by the memory reference.

WRS: (write random sequential). Write the next sequential record to a random file. The relative record number is returned.

6. Special Instructions

The remaining instructions perform special functions required by the machine that do not relate to any of the previous groups.

NEG: (negate). Complement the value of the branch flag.

Parameters: No parameters are required.

LDI: (load a code address direct). Load the code address located five bytes after the LDI instruction with the contents of <memory address> after it has been converted to hexidecimal.

Parameters: <memory address> <length - byte>

SCR: (calculate a subscript). Load the subscript stack with the value indicated from memory. The address loaded into the stack is the <initial address> plus an offset. Multiplying the <field length> by the number in the <memory reference> gives the offset value.

Parameters: <initial address> <field length> <memory reference> <memory length> <stack level>

STD: (stop display). Display the indicated information and then terminate the actions of the machine.

Parameters: <memory address> <length - byte>

STP: (stop). Terminate the actions of the machine. Parameters: no parameters are required. The following instructions are used in setting up the machine environment and cannot be used in the normal execution of the machine.

BST: (backstuff). Resolve a reference to a label. Labels may be referenced prior to their definition, requiring a chain of resolution addresses to be maintained in the code. The latest location to be resolved is maintained in the symbol table and a pointer at that location indicates the next previous location to be resolved. A zero pointer indicates no prior occurrences of the label. The code address referenced by <change address> is examined and if it contains zero, it is loaded with the <new address>. If it is not zero, then the contents are saved, and the process is repeated with the saved value as the change address after loading the <new address>.

Parameters: <change address> <new address>

INT: (initialize memory). Load memory with the <input string> for the given length at the <memory address>.

Parameters: <memory address> <address length> <input string>

SCD: (start code). Set the initial value of the program counter.

Parameters: <start address>

TER: (terminate). Terminate the initialization process and start executing code.

Parameters: no parameters are required.

IV. SYSTEM DEBUGGING METHODS AND TOOLS

Initially it appeared that the debugging of the compiler and interpreter would be straight forward. However, it became apparent that a systematic approach would have to be adopted in order to meet the objectives. As previously stated, the first step was to determine the degree to which the compiler had been developed. After accomplishing this task, the next step was to identify the means by which errors could be located and the methods by which solutions could be implemented and tested.

The method used to identify errors within the compiler consisted of the following: 1.) compiling test programs and denoting any compilation errors and 2.) examination of the symbol table construction and intermediate code instructions generated by compiling through the DATA DIVISION of a source program.

A minimum of forty-five minutes was required to recompile either module -- PART ONE or PART TWO -- of the compiler after making changes, because the object code produced by the compiler had to be linked and loaded. This indicated a need to find and use an alternative approach for testing proposed changes. The approach used, was to test compiler and interpreter modifications by using interactive debugging tools before changing the compiler's source code and recompiling. This reduced the amount of time that would

otherwise have been required by reducing the total number of recompilations.

A. DEBUGGING METHODOLOGY

The debugging methodology utilized, consisted of steps similar to those suggested by Polya's problem-solving technique [16]. First, upon encountering an occurrence of an error, the approach was to understand why the error occurred. This included determining what the compiler or interpreter had done right in its compilation or execution of a source program, followed by an analysis of what the compiler or interpreter had done incorrectly. Second, a theory was devised to explain the nature of the error(s), along with a devised method, such as a paper and pencil walk through using different variables or combinations of variables, to confirm the theory. Next, the plan concerning the error was implemented, usually this was accomplished by a paper and pencil code walk through followed by recompilation and reexecution of the program. Finally, a solution was determined, reviewed, and implemented.

It was observed, as in other program debugging efforts, that a few errors gave most of the difficulties encountered when debugging. Upon several occasions, it was thought that the origin and all side effects of an error had been discovered; later however, after having made a substantial coding change, it was realized that there was either another boundary condition, circumstance or combinatorial problem

giving rise to the error. The result was that of having to restudy and refix the error, which required additional time and effort.

To facilitate the testing and debugging of the compiler and interpreter several different software tools were utilized. It is difficult to say which was the most beneficial; however, when they were used together the task of testing and debugging was significantly enhanced.

B. INTERACTIVE TOOLS

Because the MICRO-COBOL compiler and interpreter were implemented under the CP/M operating system, two CP/M debugging facilities were used. First, the Dynamic Debugging Tool [7], DDT, is a dynamic interactive program which allows testing and debugging of programs in the CP/M operation system environment. The second was the Symbolic Instruction Debugger [6], SID, which expands upon the features of DDT. Specifically, SID includes real-time breakpoints, fully monitored execution, symbolic disassembly, assembly, and memory display and fill functions. Both debuggers were designed to operate in an interactive mode and each had several features and facilities in common which enhanced the debugging effort. One feature which allowed the setting of breakpoints at actual memory locations corresponding to a program's source lines and symbolic names was used quite extensively. Another useful facility was the ability to display and alter the programs symbolic values, which

enabled the substitution of values to check a proposed solution to an error.

C. CROSS REFERENCE LISTINGS

Another useful facility which eased the debugging effort was the cross reference listings produced by the PLM80 compiler used to compile the MICRO-COBOL compiler and interpreter. There were three different listings produced after each compilation: 1.) a line numbered source listing, 2.) a symbol address table, which included the name and actual memory address assigned for all symbols declared, and 3.) a line address table which cross referenced every line in the source listing with the 8080 code generated by the PLM80 compiler for that particular line. These listings were almost indispensable with regard to testing and debugging, and their contribution cannot be overemphasized.

D. VALIDATION TESTS

At the onset of this thesis project it was very difficult to decide how to test various constructs and features of the MICRO-COBOL compiler and interpreter and there were questions regarding test case design. During earlier work [15], the HYPO-COBOL Compiler Validation System (HCCVS) Tape (from the Automated Data Processing Equipment Selection Office (ADPESO)) was acquired -- to be used in validating the MICRO-COBOL compiler. However, the HCCVS was

never used and the tape had not been transferred to the appropriate media. This transfer was accomplished later [12]. By using the HCCVS as the evaluation package, the questions regarding test case construction and design were resolved and testing proceeded. The HCCVS was used primarily as a test bed for PART ONE of the compiler, having as an objective the goal of ensuring the proper construction of the symbol table and data initialization. Because some of the HYPO-COBOL constructs were not implemented in the MICRO-COBOL compiler (see Appendix E), the compilation of any HCCVS program past the PROCEDURE DIVISION statement was not successful.

V. CONCLUSIONS AND RECOMMENDATIONS

A significant portion of the MICRO-COBOL Compiler/Interpreter has been tested, debugged and documented. The following specific language features and facilities previously not implemented, or implemented incorrectly, have been successfully implemented, tested and debugged during this project: 1.) the compiler's ability to handle any sequence of MICRO-COBOL language constructs (PICTURE CLAUSE, VALUE CLAUSE, OCCURS CLAUSE, and USAGE COMP CLAUSE) in the declaration of an identifier, 2.) record identifier declarations with up to ten levels of elementary field items, 3.) record and elementary field identifier redefinitions, 4.) nested redefinitions, and 5.) error message generation for duplicate identifier declarations within the DATA DIVISION.

Testing and debugging has been accomplished for all presently implemented MICRO-COBOL language constructs occurring in the DATA DIVISION of a source program. Specifically, testing was performed by compiling through the DATA DIVISION of the first ten HCCVS test programs.

In addition, the MICRO-COBOL compiler has been completely documented. This documentation includes the following: 1.) module organization, 2.) module interfaces, 3.) memory organization of the Interpreter, 4.) construction and data initialization of the symbol table, and 5.) key

internal data structures.

Several areas remain which could be improved, developed and implemented, to enhance the MICRO-COBOL Compiler/Interpreter system, these include: 1.) correction of the numerical algorithms in the interpreter to allow for signed-fractional arithmetic, 2.) implementation of numeric editing capabilities, 3.) implementation of a printer control feature and interface, and 4.) testing and debugging of the compiler's ability to compile the PROCEDURE DIVISION of the HCCVS test programs.

APPENDIX A

NPS MICRO-COBOL USER'S MANUAL

TABLE OF CONTENTS

I. ORGANIZATION

The compiler is designed to run on an 8080 system in an interactive mode through the use of a teletype or console. It requires at least 24K of main memory and a mass storage device for reading and writing. The compiler is composed of two parts , each of which reads a portion of the input file. Part One reads the input program to the end of the Data Division and builds the symbol table. At the end of the Data Division, Part One is overlayed by Part Two which uses the symbol table to produce the code. The output code is written as it is produced to minimize the use of internal storage.

The BUILD Program builds the core image for the intermediate code and performs such functions as backstuffing addresses. BUILD then loads the INTERPRETER addresses. BUILD then transfers control to the INTRDR routine. The INTRDR routine copies the interpreter into memory and transfers control to the Interpreter. The interpreter is controlled by a large case statement that decodes the instructions and performs the required actions.

As a tool for debugging the compiler the DECODE Program was created; it reads the intermediate code file and translates the instructions into mnemonics followed by parameters.

II. MICRO-COBOL ELEMENTS

This section contains a description of each element in the language and shows simple examples of their use. The following conventions are used in explaining the formats: Elements enclosed in broken braces < > are themselves complete entities and are described elsewhere in the manual. Elements enclosed in braces { } are choices, one of the elements which is to be used. Elements enclosed in brackets [ ] are optional. All elements in capital letters are reserved words and must be spelled exactly.

User names are indicated in lower case. These names have been restricted to 12 characters in length. There is only one restriction on user names, the first character must be an alpha character. The remainder of the user name can have any combination of representable character in it.

The input to the compiler does not need to conform to standard COBOL format. Free form input will be accepted as the default condition. If desired, sequence numbers can be entered in the first six positions of each line. However, a toggle needs to be set to cause the compiler to ignore the sequence numbers.

ELEMENT:

IDENTIFICATION DIVISION Format

FORMAT:

```
IDENTIFICATION DIVISION.

PROGRAM-ID. <comment>.

[AUTHOR. <comment>.]

[DATE-WRITTEN. <comment>.]

[SECURITY. <comment>.]
```

DESCRIPTION:

This division provides information for program identification for the reader. The order of the lines is fixed.

EXAMPLES:

```
IDENTIFICATION DIVISION.

PROGRAM-ID. SAMPLE.

AUTHOR. MICHAEL-L-RICE.
```

ELEMENT:

ENVIRONMENT DIVISION Format

FORMAT:

```
ENVIRONMENT DIVISION.

CONFIGURATION SECTION.

SOURCE-COMPUTER. <comment> [DEBUGGING MODE].

OBJECT-COMPUTER. <comment>.

[INPUT-OUTPUT SECTION.

FILE-CONTROL.

   <file-control-entry> . . .

[I-O-CONTROL.

   SAME file-name-1 file-name-2 [file-name-3]

           [file-name-4] [file-name-5]. ] ]
```

DESCRIPTION:

This division determines the external nature of a file. In the case of CP/M all of the files used can be accessed either sequentially or randomly except for variable length files which are sequential only. The debugging mode is also set by this section.

ELEMENT:

<file-control-entry>

FORMAT:

1.

```
SELECT file-name
    ASSIGN implementor-name
    [ORGANIZATION SEQUENTIAL]
    [ACCESS SEQUENTIAL].
```

2.

```
SELECT file-name
    ASSIGN implementor-name
    ORGANIZATION RELATIVE
    [ACCESS {SEQUENTIAL [RELATIVE data-name]}].
            {RANDOM RELATIVE data-name      }
```

DESCRIPTION:

The file-control-entry defines the type of file that the program expects to see. There is no difference on the diskette, but the type of reads and writes that are performed will differ. For CP/M the implementor

name needs to conform to the normal specifications.

EXAMPLES:

```
SELECT CARDS

   ASSIGN CARD.FIL.

SELECT RANDOM-FILE

   ASSIGN A.RAN

   ORGANIZATION RELATIVE

   ACCESS RANDOM RELATIVE RAND-FLAG.
```

ELEMENT:

DATA DIVISION Format

FORMAT:

```
DATA DIVISION.

[FILE SECTION.

[FD file-name

   [BLOCK integer-1 RECORDS]

   [RECORD [integer-2 TO] integer-3]

   [LABEL RECORDS {STANDARD}]

                  {OMITTED }

   [VALUE OF implementor-name-1 literal-1

       [implementor-name-2 literal-2] ... ].

[<record-description-entry>] ...] ...

[WORKING-STORAGE SECTION.

[<record-description-entry>] ... ]

[LINKAGE SECTION.

[<record-description-entry>] ... ]
```

DESCRIPTION:

This is the section that describes how the data is structured. There are no major differences from standard COBOL except for the following: 1. Label records make no sense on the diskette so no entry is required. 2. The VALUE OF clause likewise has no meaning for CP/M. 3. The linkage section has not been implemented.

If a record is given two lengths as in RECORD 12 TO 128, the file is taken to be variable length and can only be accessed in the sequential mode. See the section on files for more information.

ELEMENT:

<comment>

FORMAT:

any string of characters

DESCRIPTION:

A comment is a string of characters. It may include anything other than a period followed by a blank or a reserved word, either of which terminate the string. Comments may be empty if desired, but the terminator is still required by the program.

EXAMPLES:

this is a comment

anotheroneallruntogether

8080b 16K

ELEMENT:

<data-description-entry> Format

FORMAT:

```
level-number {data-name}
             {FILLER   }
[REDEFINES data-name]
[PIC character-string]
[USAGE {COMP   }]
       {DISPLAY}
[SIGN {LEADING} [SEPARATE]]
      {TRAILING}
[OCCURS integer]
[SYNC [LEFT ]]
      [RIGHT]
[VALUE literal].
```

DESCRIPTION:

This statement describes the specific attributes of the data. Since the 8080 is a byte machine, there was

no meaning to the SYNC clause, and thus it has not been implemented.

EXAMPLES:

```
01 CARD-RECORD.

   02 PART PIC X(5).

   02 NEXT-PART PIC 99V99 USAGE COMP.

   02 FILLER.

      03 NUMB PIC S9(3)V9 SIGN LEADING SEPARATE.

      03 LONG-NUMB 9(15).

      03 STRING REDEFINES LONG-NUMB PIC X(15).

   02 ARRAY PIC 99 OCCURS 100.
```

ELEMENT:

PROCEDURE DIVISION Format

FORMAT:

1.

```
PROCEDURE DIVISION [USING name1 [name2] ... [name5]].

section-name SECTION.

[paragraph-name. <sentence> [<sentence> ... ] ... ] ...
```

2.

```
PROCEDURE DIVISION [USING name1 [name2] ... [name5]].

paragraph-name. <sentence> [<sentence> ...] ...
```

DESCRIPTION:

As is indicated, if the program is to contain sections, then the first paragraph must be in a section. The USING option is part of the interprogram communication module and has not been implemented.

ELEMENT:

<sentence>

FORMAT:

<imperative-statement>

<conditional-statement>

ENTER verb

DESCRIPTION:

All sentences other than ENTER fall in one of the two main catigories. ENTER is part of the interprogram communication module.

ELEMENT:

<imperative-statement>

FORMAT:

The following verbs are always imperatives:

ACCEPT

CALL

CLOSE

DISPLAY

EXIT

GO

MOVE

OPEN

PERFORM

STOP

The following may be imperatives:
arithmetic verbs without the SIZE ERROR statement
and DELETE, WRITE, and REWRITE without the INVALID option.

ELEMENT:

<conditional-statements>

FORMAT:

IF

READ

arithmetic verbs with the SIZE ERROR statement

and DELETE, WRITE, and REWRITE with the INVALID option.

ELEMENT:

ACCEPT

FORMAT:

ACCEPT <identifier>

DESCRIPTION:

This statement reads up to 255 characters from the console. The usage of the item must be DISPLAY.

EXAMPLES:

```
ACCEPT IMMAGE

ACCEPT NUM(9)
```

ELEMENT:

ADD

FORMAT:

```
ADD {identifier} [{identifier-1}] TO identifier-2
    {literal   }  {literal    }

   [ROUNDED] [SIZE ERROR <imperative-statement>]
```

DESCRIPTION:

This instruction adds either one or two numbers to a third with the result being placed in the last location.

EXAMPLES:

```
ADD 10 TO NUMB1

ADD X Y TO Z ROUNDED.

ADD 100 TO NUMBER SIZE ERROR GO ERROR-LOC
```

ELEMENT:

CALL

FORMAT:

```
CALL literal [USING name1 [name2] ... [name5]]
```

DESCRIPTION:

CALL is not implemented.

ELEMENT:

CLOSE

FORMAT:

CLOSE file-name

DESCRIPTION:

Files must be closed if they have been written. However, the normal requirement to close an input file prior to the end of processing does not exist.

EXAMPLES:

```
CLOSE FILE1

CLOSE RANDFILE
```

ELEMENT:

DELETE

FORMAT:

DELETE file-name [INVALID <imperative-statement>]

DESCRIPTION:

This statement requires the file-name of the item to be deleted. The record is logically removed by filling it with a high value character, which is not displayable to the console or line printer. The logical record space can be used again by writing a valid record in its place.

EXAMPLES:

DELETE FILE-NAME

ELEMENT:

DISPLAY

FORMAT:

```
DISPLAY {identifier} [{identifier-1}]
        {literal   }  {literal     }
```

DESCRIPTION:

This displays the contents of an identifier or displays a literal on the console. Usage must be DISPLAY. The maximum length of the display is 80 characters for literal values and 255 characters for identifiers. Only two identifiers/literals are allowed for each DISPLAY command.

EXAMPLES:

```
DISPLAY MESSAGE-1

DISPLAY MESSAGE-3 10

DISPLAY 'THIS MUST BE THE END'
```

ELEMENT:

DIVIDE

FORMAT:

```
DIVIDE {identifier} INTO identifier-1 [ROUNDED]
       {literal   }

   [SIZE ERROR <imperative-statement>]
```

DESCRIPTION:

The result of the division is stored in identifier-1; any remainder is lost.

EXAMPLES:

```
DIVIDE NUMB INTO STORE

DIVIDE 25 INTO RESULT
```

ELEMENT:

ENTER

FORMAT:

ENTER language-name [routine-name]

DESCRIPTION:

This construct is not implimented.

ELEMENT:

EXIT

FORMAT:

EXIT [PROGRAM]

DESCRIPTION:

The EXIT command causes no action by the interpreter but allows for an empty paragraph for the construction of a common return point. The optional PROGRAM statement is not implemented as it is part of the interprogram communication module.

EXAMPLES:

```
RETURN.

   EXIT.
```

ELEMENT:

GO

FORMAT:

1.

GO procedure-name

2.

GO procedure-1 [procedure-2] ... procedure-20

DEPENDING identifier

DESCRIPTION:

The GO command causes an unconditional branch to the routine specified. The second form causes a forward branch depending on the value of the contents of the identifier. The identifier must be a numeric integer value. There can be no more than 20 procedure names.

EXAMPLES:

GO READ-CARD.

GO READ1 READ2 READ3 DEPENDING READ-INDEX.

ELEMENT:

IF

FORMAT:

IF <condition> {imperative } ELSE imperative-2

{NEXT SENTENCE}

DESCRIPTION:

This is the standard COBOL IF statement. Note that there is no nesting of IF statements allowed since the IF statement is a conditional.

EXAMPLES:

IF A GREATER B ADD A TO C ELSE GO ERROR-ONE.

IF A NOT NUMERIC NEXT SENTENCE ELSE MOVE ZERO TO A.

ELEMENT:

MOVE

FORMAT:

```
MOVE {identifier-1} TO identifier-2
     {literal     }
```

DESCRIPTION:

The standard list of allowable moves applies to this action. As a space saving feature of this implementation, all numeric moves go through the accumulators. This makes numeric moves slower than alpha-numeric moves, and where possible they should be avoided. Any move that involves picture clauses that are exactly the same can be accomplished as an alpha-numeric move if the elements are redefined as alpha-numeric; also all group moves are alpha-numeric.

EXAMPLES:

```
MOVE SPACE TO PRINT-LINE.

MOVE A(10) TO B(PTR).
```

ELEMENT:

MULTIPLY

FORMAT:

```
MULTIPLY {identifier} BY identifier-2 [ROUNDED]
         {literal   }

   [SIZE ERROR <imperative-statement>]
```

DESCRIPTION:

The multiply routine requires enough space to calculate the result with the full number of decimal digits prior to moving the result into identifier-2. This means that a number with 5 places after the decimal multiplied by a number with 6 places after the decimal will generate a number with 11 decimal places which would overflow if there were more than 7 digits before the decimal place.

EXAMPLES:

```
MULTIPLY X BY Y.

MULTIPLY A BY B(7) SIZE ERROR GO OVERFLOW.
```

ELEMENT:

OPEN

FORMAT:

```
OPEN {INPUT file-name }
     {OUTPUT file-name}
     {I-O file-name    }
```

DESCRIPTION:

The three types of OPENS have exactly the same effect on the diskette. However, they do allow for internal checking of the other file actions. For example, a write to a file set open as input will cause a fatal error.

EXAMPLES:

```
OPEN INPUT CARDS.

OPEN OUTPUT REPORT-FILE.
```

ELEMENT:

PERFORM

FORMAT:

1.

```
PERFORM procedure-name [THRU procedure-name-2]
```

2.

```
PERFORM procedure-name [THRU procedure-name-2]
   {identifier} TIMES
    {integer   }
```

3.

```
PERFORM procedure-name [THRU procedure-name-2]
   UNTIL <condition>
```

DESCRIPTION:

All three options are supported. Branching may be either forward or backward, and the procedures called may have perform statements in them as long as the end points do not coincide or overlap.

EXAMPLES:

```
PERFORM OPEN-RCUTINE.
```

```
PERFORM TOTALS THRU END-REPORT.

PERFORM SUM 10 TIMES.

PERFORM SKIP-LINE UNTIL PG-CNT GREATER 60.
```

ELEMENT:

READ

FORMAT:

1.

READ file-name INVALID <imperative-statement>

2.

READ file-name END <imperative-statement>

DESCRIPTION:

The invalid condition is only applicable to files in a random mode. All sequential files must have an END statement.

EXAMPLES:

READ CARDS END GO END-OF-FILE.

READ RANDOM-FILE INVALID MOVE SPACES TO REC-1.

ELEMENT:

REWRITE

FORMAT:

REWRITE record-name [INVALID <imperative>]

DESCRIPTION:

REWRITE is only valid for files that are open in the I-O mode. The INVALID clause is only valid for random files. This statement results in the current record being written back into the place that it was just read from, the last executed read.

EXAMPLES:

REWRITE CARDS.

REWRITE RAND-1 INVALID PERFORM ERROR-CHECK.

ELEMENT:

STOP

FORMAT:

```
STOP {RUN    }
     {literal}
```

DESCRIPTION:

This statement ends the running of the interpreter. If a literal is specified, then the literal is displayed on the console prior to termination of the program.

EXAMPLES:

```
STOP RUN.

STOP 1.

STOP "INVALID FINISH".
```

ELEMENT:

SUBTRACT

FORMAT:

```
SUBTRACT {identifier-1} [identifier-2] FROM identifier-3

         {literal-1   } [literal-2   ]

   [ROUNDED] [SIZE ERROR <imperative-statement>]
```

DESCRIPTION:

Identifier-3 is decremented by the value of identifier/literal one, and, if specified, identifier/literal two. The results are stored back in identifier-3. Rounding and size error options are available if desired.

EXAMPLES:

```
SUBTRACT 10 FROM SUB(12).

SUBTRACT A B FROM C ROUNDED.
```

ELEMENT:

WRITE

FORMAT:

1.

```
WRITE record-name [{BEFORE} ADVANCING {INTEGER}]
                  {AFTER }            {PAGE   }
```

2.

```
WRITE record-name INVALID <imperative-statement>
```

DESCRIPTION:

The record specified is written to the file specified in the file section of the source program. The INVALID option only applies to random files.

EXAMPLES:

```
WRITE OUT-FILE.

WRITE RAND-FILE INVALID PERFORM ERROR-RECOV.
```

ELEMENT:

<condition>

FORMAT:

RELATIONAL CONDITION:

```
{identifier-1} [NOT] {GREATER} {identifier-2}
{literal-1}          {LESS   } {literal-2   }
                     {EQUAL  }
```

CLASS CONDITION:

```
identifier [NOT] {NUMERIC   }
                 {ALPHABETIC}
```

DESCRIPTION:

It is not valid to compare two literals. The class condition NUMERIC will allow for a sign if the identifier is signed numeric.

EXAMPLES:

```
A NOT LESS 10.

LINE GREATER 'C'.

NUMB1 NOT NUMERIC
```

ELEMENT:

Subscripting

FORMAT:

data-name (subscript)

DESCRIPTION:

Any item defined with an OCCURS may be referenced by a subscript. The subscript may be a literal integer, or it may be a data item that has been specified as an integer. If the subscript is signed, the sign must be positive at the time of its use.

EXAMPLES:

A(10)

ITEM(SUB)

III. COMPILER TOGGLES

There are four compiler toggles which are controlled by an entry following the compiler activation command, COBOL <filename>. The format of the entry consists of following <filename> by one space and then entering a "$" followed immediately by the desired toggles. There must be only one space after <filename> and no spaces between the "$" and the toggles. The following is an example of a typical entry:

COBOL EXAMPLE $ST

This entry would cause the compiler to ignore the sequence numbers entered at the beginning of each input file line and print the token numbers to the output device. In each case the toggle reverses the default value.

$L -- list the input code on the screen as the program is compiled. Default is on. Error messages will be difficult to understand if this toggle is turned off, but if the interface device is a teletype, it may be desired in certain situations.

$S -- sequence numbers are in the first six positions of each record. Default is off.

$P -- list productions as they occur. Default is off.

$T -- list tokens from the scanner. Default is off.

IV. RUN TIME CONVENTIONS

This section explains how to run the compiler on the current system. The compiler expects to see a file with a type of CBL as the input file. In general, the input is free form. If the input includes sequence numbers then the compiler must be notified by setting the appropriate toggle. The compiler is started by typing COBOL <file-name>. Where the file name is the system name of the input file. There is no interaction required to start the second part of the compiler. The output file will have the same <file-name> as the input file, and will be given a file type of CIN. Any previous copies of the file will be erased.

The interpreter is started by typing EXEC <filename>. The first program is a loader, and it will display "LOAD FINISHED' to indicate successful completion. The run-time package will be brought in by the INTRDR routine, and execution should continue without interuption.

V. FILE INTERACTIONS WITH CP/M

The file structure that is expected by the program imposes some restrictions on the system. References 4 and 5 contain detailed information on the facilities of CP/M, and should be consulted for details. The information that has been included in this section is intended to explain where limitations exist and how the program interacts with the system.

All files in CP/M are on a random access device, and there is no way for the system to distinguish sequential files from files created in a random mode. This means that the various types of reads and writes are all valid to any file that has fixed length records. The restrictions of the ASSIGN statement do prevent a file from being open for both random and sequential actions during one program.

Each logical record is terminated by a carriage return and a line feed. In the case of variable length records, this is the only end mark that exists. This convention was adopted to allow the various programs which are used in CP/M to work with the files. Files created by the editor, for example, will generally be variable length files. This convention does remove the capability of reading variable length files in a random mode.

All of the physical records are 128 bytes in length, and the program supplies buffer space for these records in addition to the logical records. Logical records may be of

any desired length.

VI. ERROR MESSAGES

A. COMPILER FATAL MESSAGES

BR Bad read -- disk error, no corrective action can be taken in the program.

CL Close error -- unable to close the output file.

MA Make error -- could not create the output file.

MO Memory overflow -- the code and constants generated will not fit in the alloted memory space.

OP Open error -- can not open the input file, or no such file present.

SO Stack overflow -- the LALR(1) parsing stack has exceeded its maximum allowable size.

ST Symbol table overflow -- symbol table is too large for the allocated space.

WR Write error -- disk error, could not write a code record to the disk.

B. COMPILER WARNINGS

CE Close error -- attempted to close a non-existing file.

DC Decimal count error -- decimal significance is greater than 18 digits.

DI Duplicate identifier -- the identifier name has been

previously declared in the WORKING STORAGE area of the program.

EF Excess files -- the number of files declared in the source program exceeds 24.

EL Extra levels -- only 10 levels are allowed.

FT File type -- the data element used in a read or write statement is not a file name.

IA Invalid access -- the specified options are not an allowable combination.

ID Identifier stack overflow -- more than 20 items in a GO TO -- DEPENDING statement.

IS Invalid subscript -- an item was subscripted but it was not defined by an OCCURS.

IT Invalid type -- the field types do not match for this statement.

LE Literal error -- a literal value was assigned to an item that is part of a group item previously assigned a value.

LV Literal value error -- the PICTURE clause field type does not match the VALUE clause literal type.

MD Multiple decimals -- a numeric literal in a VALUE clause contains more than one decimal point.

MS Multiple signs -- a signed numeric literal in a VALUE clause contains more than one sign.

NF No file assigned -- there was no SELECT clause for this file.

NI Not implemented -- a production was used that is not implemented.

NN Non-numeric -- an invalid character was found in a numeric string.

NP No production -- no production exists for the cuurrent parser configuration; error recovery will automatically occur.

NV Numeric value -- a numeric value was assigned to a non-numeric item.

OE Open error -- attempt to open a file that was not declared; or attempted to open a file for I-O that was not a RELATIVE file.

PC Picture clause -- an invalid character or set of characters exists in the picture clause.

PF Paragraph first -- a section header was produced after a paragraph header, which is not in a section.

R1 Redefine nesting -- a redefinition was made for an item which is part of a redefined item.

R2 Redefine length -- the length of the redefinition item was greater than the item that it redefined. This error message may be printed out one identifier past the redefining identifier record in which it occurred.

R3 Redefines misplaced -- a redefines was attempted in the FILE SECTION of the source program.

SE Scanner error -- the scanner was unable to read an identifier due to an invalid character.

SG Sign error -- either a sign was expected and not

found, or a sign was present when not valid.

SL Significance loss -- the number assigned as a value is larger than the field defined.

TE Type error -- the type of a subscript index is not integer numeric.

UI Undeclared identifier -- the identifier was not declared in WORKING STORAGE area of the source program.

VE Value error -- a value statement was assigned to an item in the file section.

WL Wrong level error -- program attempted to write a record other than an Ø1 level record to an output file.

C. INTERPRETER FATAL ERRORS

CL Close error -- the system was unable to close an output file.

ME Make error -- the system was unable to make an input file on the disk.

NF No file -- an input file could not be opened.

W1 Write non-sequential -- attempted to WRITE to a file opened for INPUT or a file opened for I-O when ACCESS was SEQUENTIAL.

W2 Wrong key -- attempted to change the key value to a lower value than the number of the last record written.

W3 Write input -- attempted to WRITE to a file opened for INPUT.

W4 Write non-empty -- attempted to WRITE to a non-empty record.

W5 Read output -- attempted to READ a file opened for OUTPUT.

W6 Rewrite error -- attempted to REWRITE to a file not opened fo I-O.

W7 Rewrite error -- attempted to REWRITE a record before reading the file; or multiple REWRITE attempts without doing a READ between each.

D. INTERPRETER WARNING MESSAGES

EM End mark -- a record that was read did not have a carriage return or a line feed in the expected location.

GD Go to depending -- the value of the depending indicator was greater than the number of available branch addresses.

IC Invalid character -- an invalid character was loaded into an output field during an edited move. For example, a numeric character into an alphabetic-only field.

SI Sign Invalid -- the sign is not a "+" or a "-".

WE Write Error -- attempted to write to an output file.

APPENDIX B

LIST OF MICRO-COBOL RESERVED WORDS

The following is a list of reserved words for MICRO-COBOL. The reserved words are the same as those specified for the HYPO-COBOL language, except where noted with an asterisk (*).

ACCEPT	ENVIRONMENT	MULTIPLY	RUN
ACCESS	EOF *	NEXT	SAME
ADD	EQUAL	NOT	SECTION
ADVANCING	ERROR	NUMERIC	SECURITY
AFTER	EXIT	OBJECT-COMPUTER	SELECT
ALPHABETIC	FD	OCCURS	SENTENCE
ASSIGN	FILE	OF	SEPARATE
AUTHOR	FILE-CONTROL	OMITTED	SEQUENTIAL
BEFORE	FILLER	OPEN	SIGN
BLOCK	FROM	ORGANIZATION	SIZE
BY	GO	OUTPUT	SOURCE-COMPUTER
CALL	GREATER	PAGE	SPACE
CLOSE	I-O	PERFORM	STANDARD
COBOL	I-O-CONTROL	PIC	STOP
COMP	IDENTIFICATION	PROCEDURE	SUBTRACT
CONFIGURATION	IF	PROGRAM	SYNC
DATA	INPUT	PROGRAM-ID	THRU
DATE-WRITTEN	INPUT-OUTPUT	QUOTE	TIMES
DEBUGGING	INVALID	RANDOM	TO
DELETE	INTO	READ	TRAILING
DEPENDING	LABEL	RECORD	UNTIL
DISPLAY	LEADING	RECORDS	USAGE
DIVIDE	LEFT	REDEFINES	USING
DIVISION	LESS	RELATIVE	VALUE
ELSE	LINKAGE	REWRITE	WORKING-STORAGE
END	MODE	RIGHT	WRITE
ENTER	MOVE	ROUNDED	ZERO

APPENDIX C

The MICRO-COBOL compiler and interpreter source files currently exist in the high level language PLM80 and are edited and compiled under the ISIS operating system on a INTEL Corporation MDS system. This is a description of the procedures required to compile and establish the programs to compile and interpret a MICRO-COBOL program. The MICRO-COBOL compiler and interpreter run on any 8080 or Z-80 based microcomputer that operates under CP/M. The execution of the following four files will cause a MICRO-COBOL program to be compiled and executed:

1. COBOL.COM
2. PART2.COM
3. EXEC.COM
4. CINTERP.COM

These four files are created from the following six PLM80 source programs.

1. PART1.PLM
2. PART2.PLM
3. BUILD.PLM
4. IREADER.PLM
5. INTRDR.PLM
6. INTERP.PLM

The procedures used to create the four object files (COM files) involve compiling, linking, and locating each of the

six source files under ISIS. The SID program is then used under CP/M to construct the executable files. Each of the following steps describe the action(s) to be taken and, where appropriate, the command string to be entered into the computer.

1. An ISIS system disk containing the PLM80 compiler is placed into drive A and a non-system disk containing the source programs is placed into drive B. It should be noted that drive A and B are the CP/M reference names for the drives while F1 and F2 are the ISIS reference names used for the associated disk drives.

2. Compile the PLM source program under ISIS using the the following command:

```
PLM80 :F1:<filename>.PLM DEBUG XREF
```

DEBUG saves the symbol table and line files for later use during debugging sessions. XREF causes a cross-reference listing, of all identifiers in the source program, to be created. The cross-reference listing includes each identifier and the associated line number where the identifer was declared and the line number of each occurence of the identifier in the source program [9].

3. Link the PLM80 object file.

```
LINK :F1:<filename>.OBJ, TRINT.OBJ, PLM80.LIB, TO
:F1:<filename>.MOD
```

See reference 10 for an explanation of PLM80.LIB. The

TRINT.OBJ program interfaces the MON1 and MON2 functions of CP/M to the source program, allowing for the use of absolute addresses in referencing these functions.

4. Locate the object file.

LOCATE :F1:<filename>.MOD CODE(org address)

The "org address" is the address where the program will begin to be loaded into memory. The following are "org addresses" for the associated program:

PART1.MOD	100H
PART2.MOD	100H
INTERP.MOD	100H
INTRDR.MOD	80H
BUILD.MOD	100H
IREADER.MOD	D000H

The "org addresses" above represent the ones used with a 62K byte CP/M system. The only address that would need to be changed if a different size system was used would be the one for IREADER.MOD. See appendix E for specifics on the address to use for IREADER.

4a. The two files INTRDR and IREADER just created by the LOCATE command must be converted to "HEX FILES". By using the ISIS command OBJHEX <filename> the file will be converted to the "HEX file" <filename>.HEX.

5. Replace the ISIS system disk in drive A with a CP/M system disk and reboot the system.

6. Transfer the located ISIS file from the ISIS disk on drive B to the CP/M disk on drive A.

FROMISIS <filename>

6a. When transfering the "HEX files" to the CP/M disk use the following:

FROMISIS <filename>.HEX

7. Convert the ISIS file to a CP/M executable form.

OBJCPM <filename>

7a. The "HEX files" are not coverted to a CP/M format, but are left in the HEX format.

At this point the object file is in machine readable form and will run under CP/M when called properly. PART2.COM and CINTERP.COM are called by PART1.COM (COBOL.COM) and EXEC.COM, respectively and need no further work. PART1.COM and EXEC.COM need to be constructed from the remaining four files.

PART1.COM is created by entering the following commands:

1. SID PART1.COM
2. IREADER.HEX
3. R6200
4. A2A9A
5. JMP 0D000
6. Control-C
7. Save 52 COBOL.COM

See reference 6 for an explanation of the "I", "R", and "A" commands used above and ref 4 for an explanation of the "SAVE" command. Steps four and five above are used to patch the JUMP to IREADER referred to in the PART1.PLM program into the PART1.COM program.

EXEC.COM us created by entering the following commands:

1. SID BUILD.COM
2. INTRDR.HEX
3. R1C00
4. A1CB5
5. JMP 5
6. A1CC1
7. JMP 5
8. CONTROL-C
9. SAVE 31 EXEC.COM

Statements 4, 5, 6, and 7 above are used to patch the JUMP to BDOS referred to in the INTRDR.PLM program into the INTRDR.HEX program.

NPS MICRO-COBOL programs may now be executed in the following manner. The source program is named, <filename>.CBL. The command "COBOL <filename>", causes the MICRO-COBOL source program to be read into memory and compiled. During the compilation, the intermediate code file, <filename>.CIN, is written out to the disk as the code is generated. The command "EXEC <filename>", causes the file, <filename>.CIN, to be executed.

APPENDIX D

PART ONE AND PART TWO INTERNAL DATA STRUCTURES AND SIGNIFICANT VARIABLES

Within PART ONE and PART TWO, many significant data structures are used by the procedures which constitute the scanner and parser. Descriptions are given below for those structures regarded as important and necessary for future compiler development.

1. Interfacing Structures

ADD$END -- this variable is used to hold the end of file filler for the end of the source program.

BUFFER(11) -- byte array used to hold the filename and filetype if declared, of an input or output file in the SELECT CLAUSE of the FILE SECTION of a MICRO-COBOL source program.

BUFFER$END -- address variable which marks the last byte of the compiler input buffer which is a 128 byte buffer used for reading the source program.

IN$ADDR -- address variable, default file control block used initially to hold the <filename.CBL> of the source program to be compiled.

IN$BUFF -- literal value, marks the first byte of the compiler input buffer.

INPUT$FCB -- byte value, based at IN$ADDR(33), the base address of the default file control block of the source

program.

OUTPUT$BUFF(128) -- byte array, used as a 128 byte output buffer for loading the generated output (pseudo instructions) when writing to the intermediate code file.

OUTPUT$CHAR -- byte value, based at the OUTPUT$PTR; used to identify the particular byte of the output buffer (OUTPUT$BUFF) to which the next intermediate code instruction is to be written.

OUTPUT$END -- address variable, pointer to the end of the output buffer (OUTPUT$BUFF).

OUTPUT$FCB(33) -- byte array, the FCB for the intermediate code file <filename.CIN> established in PART ONE of the compiler and pasted to PART TWO of the compiler by IREADER module.

OUTPUT$PTR -- address value, used as an index into the output buffer (OUTPUT$BUFF).

POINTER -- address value, the address of the byte holding the next input character of the source program.

2. Debugging Structures

DEBUGGING -- logical byte value, toggle used in conjunction with ":" in a MICRO-COBOL source program text; allows for the compilation or non-compilation of the deugging statements following the ":".

LIST$INPUT -- logical byte value, toggle used to display or not display a source program to the CRT during compilation.

PARMLIST(9) -- byte array used to hold the toggles set by the compiler developer or user upon execution of the command: COBOL <filename.CBL> $TOGGLES.

PRINT$PROD -- logical byte value, toggle used to print, in chronological order, at the CRT the production numbers of the compiler grammar rules used during a compilation of the source program.

SEQ$NUM -- logical byte value, toggle used to indicate the presence of sequence numbers in the first six positions of each line of a source program being compiled.

3. Memory Structures

EOFFILLER -- literal value, used to test for the occurrence of an end of file character ("1AH" in CP/M), when reading the source program.

FREE$STORAGE -- first free address following PART ONE of the compiler; utilized as the base of the symbol table. This is the same value as HASHTABADDR in PART TWO of the compiler.

INITIAL$POS -- address value, the initial location of the IREADER module before it is copied to high memory at location MAX$MEMORY.

MAX$MEMORY -- address value, the location in high memory where the IREADER module is to be moved.

NEXT$AVAILABLE -- address value, the pseudo machine memory address for the next machine instruction.

PART1$LEN -- the number of bytes of information

saved in high memory after execution of PART ONE and used to initialize PART TWO module variables of the compiler.

PASS1$TOP -- this address is used in conjunction with PASS1$LEN for locating the fourty-eight bytes of information saved in PART ONE for use in PART TWO of the compiler.

RDR$LENGTH -- literal value representing the 255 bytes of the IREADER module to be moved from INITIAL$POS to MAX$MEMORY.

4. Scanner Structures:

ACCUM(51) -- an array of 51 bytes; the first byte contains a count of the total number of characters currently in the accumulator. This structure holds tokens as they are scanned, and will hold either a reserved word, a user defined identifier, or a literal.

COLLISION -- address varible, contained in first two bytes of an identifier's symbol table entry and indicates whether there is another identifier which hashes to the same hash table address. This address points to that identifier's address in the symbol table.

DISPLAY(74) -- an array of 74 bytes; the first byte contains a count of the total number of characters (1-73) currently in the display buffer. Every line within a source program is loaded into this structure for subsequent printing to the CRT terminal during compilation.

EDIT$FLAG -- logical flag which denotes the fact

that a '$' symbol has been loaded into the DISPLAY array during compilation. When set the characters within DISPLAY will be printed one at a time, until the entire line is printed.

HASH$TABLE$ADDR -- the base of the symbol table generated in PART ONE, used as the base of the hashtable.

HASHTABADDR -- this was the address of the bottom of the symbol table generated in PART ONE of the compiler, and saved for Part two.

INPUT$STR -- literal value (32), returned to the LALR(1) parser anytime the token contained in the ACCUM is not a reserved word or literal.

LITERAL -- literal value (15), returned to the LALR(1) parser anytime the first character encountered by the scanner is a quote ('), prior to loading the ACCUM.

MAX$LEN -- length of the longest reserved word allowed by MICRO-COBOL.

5. Parser Structures:

BUFFER(31) -- byte array used to store edited PICTURE CLAUSE characters for subsequent intermediated code generation.

COMPILING -- logical byte value which indicates that compiling is taking place or not in PART ONE or PART TWO; set to FALSE whenever the statestack of the LALR(1) parser is reduced to a recognizable finished state.

CUR$SYM -- address variable that holds the address

of the current symbol being accessed in the symbol table.

DUP$IDEN$ARRAY(24) -- address array that holds the symbol address for all files declared in the INPUT-OUTPUT SECTION of a source program. When the FILE SECTION entry for the file is encountered the array is searched to determine if the file was declared and to insure that a FILE SECTION entry had not been previously made.

FILE$DESC$FLAG -- logical byte value; indicates whether the compiler is compiling the FILE DESCRIPTION SECTION of a source program or not.

FILESECEND --logical byte value set whenever the parser has parsed passed the FILE SECTION of a source program.

HOLD$LIT(51) -- byte array, first byte contains a count of the total number of characters currently stored in the HOLDLIT buffer which is used to hold characters for a VALUE CLAUSE.

ID$STACK(10) -- address array which functions as a stack and is used to hold the addresses of identifiers at both the record and elementary levels. Whenever a record identifier has nested elementary field identifiers it is saved on the ID$STACK. Also, anytime a record identifier has succeeding record identifiers redefining it, it is saved on the ID$STACK. In the case of multiple record descriptions in a file description of the FILE SECTION, the record descriptions following the first record are assumed redefinitions.

ID$STACK$PTR -- a byte index variable into the ID$STACK array.

MAXIDLEN -- a numeric value (12), maximum length of any user defined identifier.

MP -- byte index variable into the VALUE array.

MPP1 -- byte index variable into the VALUE array, one byte above MP index.

NEXT$SYM -- this address indicates the next available free space for a symbol table entry.

PENDING$LITERAL -- byte value (Ø,1,2,3,4,5), indicates the category of the target input to a VALUE CLAUSE.

PENDINGLITID -- byte value (Ø,1,2,3,4,5), which is saved to indicate the category of the most recently encountered target input to a VALUE CLAUSE.

PRODUCTION -- byte value, determined by the parser and indicates the next semantic action to be taken by the compiler.

REDEF -- logical byte value which allows the testing of an identifier's storage value size against the storage value size of a second identifier that redefines the first. Set to TRUE when there are multiple record descriptions within a FD BLOCK in the FILE SECTION, or when a record or elementary identifier declaration in the WORKING STORAGE SECTION contains a REDEFINES CLAUSE.

REDEF$FLAG -- logical byte value, used to denote the scanning and parsing of the FILE SECTION of a source

program, helps in identifying duplicate identifiers within this section.

REDEF$ONE -- address variable that holds the symbol table address of the identifier being redefined by another identifier.

REDEF$TWO -- an address variable that contains the symbol table address of an identifier which redefines another identifier.

SP -- a byte index for the STATESTACK array and the VALUE array; points to the top of the STATESTACK array.

STATE -- a byte value numeric quantity that indicates the current parser state.

STATESTACK(30) -- a byte array which stacks the states (production sequences) the parser passes through while compiling a source program.

TRUNC$FLAG -- logical byte value that indicates numeric truncation of an identifier's VALUE CLAUSE input hasn't occurred, because the identifier's associated PICTURE CLAUSE has not been scanned and parsed.

VALUE(30) -- an address array that holds addresses of identifiers, specific attributes of these identifiers and attributes of the current source program statement or sentence being parsed.

VARC(51) -- a byte array, the first byte holds the count of the total number of characters within it, used to hold all the ASCII characters of tokens scanned within the source program, excluding reserved words; for subsequent

analysis and processing.

VALUE$FLAG -- a logical byte that is set anytime an identifier has an associated VALUE CLAUSE; used primarily to recognize the occurrence of a PICTURE CLAUSE before the VALUE CLAUSE or when a record entry has a VALUE CLAUSE, but no associated PICTURE CLAUSE except for those in its elementary field identifiers.

VALUE$LEVEL -- a byte value which saves the level number of a record identifier which doesn't have an associated PICTURE CLAUSE.

APPENDIX E

The NPS MICRO-COBOL compiler/interpreter is designed to operate on any 8080 or Z80 based microcomputer operating under CP/M with at least 20K bytes of memory. The PLM80 source files have been written in such a way, that certain variables must be altered in the source code to take advantage of the machine that the programs are going to be operating on. This appendix covers those programs and the variables that must be altered.

1. PART1.PLM

This program has two variables that are memory size dependent, MAX$MEMORY and MAX$INT$MEMORY. The variable MAX$MEMORY is set to 100H bytes below the base of the BDOS and is used for the beginning address of the IREADER routine. The variable MAXINTMEMORY is set to the base address of the BDOS and is used as the upper limit for the intermediate code file.

2. PART2.PLM

This program also has two variables that are memory size dependent, MAX$MEMORY and PASS1$TOP. In this program MAX$MEMORY is set to the base address of the BDOS while PASS1$TOP is set to 100H bytes below the base of the BDOS.

3. IREADER.PLM

Although, this program does not have any memory size dependent variables the program must be modified to execute properly. When using the LOCATE command, under ISIS, this routine must be located 100H bytes below the BDOS of the

system. This address would correspond to the values of MAX$MEMORY in PART2.PLM and MAXINTMEMORY in PART1.PLM.

4. INTERP.PLM, INTRDR.PLM, and BUILD.PLM

These three programs have no variables that need to be altered.

5. GENERAL INFORMATION

The current version of the NPS MICRO-COBOL compiler/interpreter is designed for continued development and certain variables are not set to make optimal use of memory. The variable NEXT$AVAILABLE, in PART1.PLM, is set to 3002H and CODE$START, in INTERP.PLM, is set to 3000H. Normally, CODE$START would be set to the address immediately following the last address in CINTERP.COM and NEXT$AVAILABLE would be set two bytes above that address. These address are currently set approximately 950H bytes above where they should be located, to allow for testing and expansion of the interpreter. As soon as implementation is completed these two addresses can be reset to appropriate values.

APPENDIX F

MICRO-COBOL Parse Table Generation

The parse tables for NPS Micro-Cobol were generated on the IBM 360 using the LALR(1) parse table generater described in reference 17. There are basically two steps involved in generating the tables. First, a deck of cards containing the grammar is entered into the computer using the following JCL:

```
//GO EXEC PGM= LALR,REGION=220K
//STEPLIB DD DSN=F0963.LALR,UNIT=2314,
      VOL=SER=LINDA,DISP=SHR
//SYSPRINT DD SYSOUT=A,DCB=(RECFM=FB,
      LRECL=133,BLKSIZE=3325),
//SPACE=(CYL,(1,1))
//NONTERM DD SPACE=(CYL,(1,1)),UNIT=SYSDA
//FSMDATA DD SPACE=(CYL,(1,1)),UNIT=SYSDA
//PTABLES DD SYSOUT=B,
      DCB=(RECFM=FB,LRECL=80,BLKSIZE=800)
//SYSIN DD *
```

The ouput from this run is a listing and a card deck containing the tables in XPL compatable format. This deck is then translated into PLM compatible format using the following JCL and an XPL program which is available in the card deck library in the Computer Science Department at the Naval Postgraduate School.

```
//EXEC XCOM
```

```
//COMP.SYSIN DD *
//GO.SYSPUNCH DD SYSOUT=B,
      DCB=(RECFM=FB,LRECL=80,BLKSIZE=800)
//GO.SYSIN DD *
```

The tables are then transferred to a diskette and edited into the PLM80 source program using the ISIS COPY and EDIT features on the INTEL MDS System.

APPENDIX G

LIST OF INOPERATIVE CONSTRUCTS

The following is a list of MICRO-COBOL elements that either have not been implemented or have been implemented incorrectly.

LINKAGE SECTION

USAGE COMP

SIGN {LEADING} SEPARATE
{TRAILING}

SYNC {LEFT}
{RIGHT}

ADD

DIVIDE

DELETE

EXIT

MOVE

MULTIPLY

SUBTRACT

The following HYPO-COBOL elements are part of MICRO-COBOL only to the extent that they are defined in the grammar. No code has been written to support them.

USING

CALL

ENTER

```
                    {BEFORE}                    {INTEGER}
WRITE  record-name  {AFTER}   ADVANCING        {PAGE}
```

```
PART1:
DO;
/* NORMALLY ORG'ED AT 100H */

    /*        COBOL COMPILER - PART 1                */

       /*    GLOBAL DECLARATIONS AND LITERALS    */

DECLARE LIT LITERALLY 'LITERALLY';
DECLARE
    PARMS           LIT         '6DH',
    PARMLIST(9)     BYTE         INITIAL('          '),
    EOFFILLER       LIT         '1AH',
                               /* END OF RECORD FILLER */
    MAX$MEMORY      LIT         '0D000H',
                               /* TOP OF USEABLE MEMORY */
    INITIAL$POS     LIT         '3200H',
    RDR$LENGTH      LIT         '255',
    PASS1$LEN       LIT         '48',
    CR              LIT         '13',
    LF              LIT         '10',
    QUOTE           LIT         '27H',
    POUND           LIT         '23H',
    TRUE            LIT         '1',
    FALSE           LIT         '0',
    FILE$DESC$FLAG              BYTE       INITIAL(FALSE),
    REDEF$FLAG                  BYTE       INITIAL(FALSE),
    DUP$IDEN$ARRAY(24)        ADDRESS
INITIAL(0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0),
    FOREVER         LIT         'WHILE TRUE';

DECLARE MAXRNO LIT        '104',/* MAX READ COUNT */
        MAXLNO LIT        '129',/* MAX LOOK COUNT */
        MAXPNO LIT        '145',/* MAX PUSH COUNT */
        MAXSNO LIT        '234',/* MAX STATE COUNT */
        STARTS LIT        '1';/* START STATE */

DECLARE READ1 (*) BYTE
DATA(0,57,48,56,32,8,25,59,2,16,17,22,29,53,58,11,32,32,39
,38,34,44,9,19,32,37,6,33,3,14,15,18,20,32,28,49,32,1,42
,38,36,43,1,1,1,1,1,1,1,1,1,10,1,39,1,1,1,38,40,49,38,39,1
,1,38,23,24,55,52,41              ,35,46,1,7,50,1,32,1,32,32,45
,1,32,1,32,1,32,47,37,4,26,32,54,40,1,1
,32,5,12,13,21,22,27,1,60,1,23,24,55,30,51);
DECLARE LOOK1(*) BYTE
DATA(0,8,0,25,0,9,19,0,42,0,42,0,1,0,52,0,41,0,35,0,1,0,47
,0,4,0,54,0,40,0,35,46,60,0,1,0,32,0,1,0,1,0,11,0,60,0,7,0
32,0,32,0,32,0);
DECLARE APPLY1(*) BYTE
DATA(0,0,0,0,0,0,9,10,12,14,19,0,0,0,0,0,0,101,0,0,100,0
,0,0,0,0,0,97,0,27,0,0,0,69,0,91,92,0,0,91,92,0,0,0,0,13
```

```
,17,0,102      ,103,104,0,0,0,0,0,95,0,0,54,0,0,23,30,38,39,0
,21,40,52,56,87,93,94,0);
DECLARE READ2(*) BYTE
DATA(0,65,57,64,154,26,37,67,21,30,31,33,39,61,66,27,234
,215,51,45,108,109,223,224,233,43,216,217,22,230,229,232
,231,228,173  ,172,169,9,226,47,196,195,7,8,11,13,15,2,3,105
,14,158,4,50,20,12,18        ,48,171,170,44,49,19,10,46,35,36
,63,60,53,42,146,16,25,58,106,155     ,148,155,155,55,150,155
,152,155,157,155,56,193,23,208,234,62,52,206
,180,234,24,28,107,32,34,38,17,68,164,35,36,63,40,59);
DECLARE LOOK2(*) BYTE
DATA(0,5,130,6,131,29,29,132,41,133,54,134,135,69,71,136
,72,137,73,138,139,80,84,140,86,198,88,141,89,142184,184
,184,91,189   ,92,93,197,211,95,143,96,97,176,99,144,145,101
,102,200,103,202,104,188);
DECLARE APPLY2(*) BYTE
DATA(0,0,77,111,112,147,79,114,81,82,83,78,76,117,75,156
,126,163,162,100,166,165,167,118,168,160,124,179,178,94
,121,74,125       ,120,119,187,187,186,98,192,192,191,194,113
,183,128,129,127,205,205 ,205,204,115,123,90,122,214,213,221
,219,218,222,199,85,220,116,87         ,110,70,174,209,207,182
,182,181);
DECLARE INDEX1(*) BYTE
DATA(0,1,2,3,4,5,6,7,8,4,4,24,4,24,4,13,14,24,109,4,15,16
,16,24,17,18,19,16,20,22,24,25,26,28,29,34,36,37,24,24,16
,38,39,40    ,42,43,44,45,46,47,48,49,16,50,38,51,16,52,53,54
,55,56,57,58,60,61 ,62,63,64,8,65,68,69,70,71,72,73,74,75,77
,79,81,83,85,87,88,89,90,92     ,93,94,8,8,16,95,97,97,15,103
,104,105,109,24,24,24,1,3,5,8,10,12,14 ,16,18,20,22,24,26,28
,30,34,36,38,40,42,44,46,48,50,52,185,149,225
,227,227,190,151
,153,203,159,210,161,175,212,201,177,1,2,3,3,4,4,5,5 ,6,6,12
,13,14,14,15,15,16,16,17,19,19,20,20,20,22,22,23,23,24,24,25
,25,26,26,27,29,29,31,32,32,33,33,35,38,38,33,33,39,39,39
,39,39,42    ,42,43,43,44,44,45,45,48,52,52,53,53,54,54,55,55
,56,56,56,56,56,56       ,56,56,58,58,58,59,59,61,61,61,61,61
,62,67);
DECLARE INDEX2(*) BYTE
DATA(0,1,1,1,1,1,1,1,5,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1
,1,1,2,2,1,1,2,1,5,2,1,1,1,1,1,1,1,2,1,1,1,1,1,1,1,1,1,1,1
,1,1,1,1,1  ,1,1,1,2,1,1,1,1,1,5,3,1,1,1,1,1,1,1,2,2,2,2,2,2
,1,1,1,2,1,1,1,5,5,1     ,2,6,6,1,1,1,4,2,1,1,1,2,2,3,2,2,2,2
,2,2,2,2,2,2,2,4,2,2,2,2,2,2,2,2  ,2,2,5,6,29,41,54,69,71,72
,73,80,84,88,89,96,99,101,3,9,3,0,3,0,3,0   ,0,1,7,8,1,0,6,0
,0,1,3,0,1,1,2,1,0,0,0,0,1,0,2,0,0,1,2,0,1,5,3,0,0,1  ,4,0,0
,0,1,2,1,2,2,2,0,2,3,0,3,0,0,1,4,0,0,1,0,0,0,0,1,1,1,1,2,2,1
,1,1,0,0,0,0,0,0,0,0,0,0,0,0); /* END OF TABLES */
DECLARE
    /* JOINT DECLARATIONS
    THESE ITEMS ARE DECLARED TOGETHER IN THIS SECTION
    IN ORDER TO FACILITATE THEIR BEING SAVED FOR
    THE SECOND PART OF THE COMPILER. */

    OUTPUT$FCB     (33) BYTE
```

```
    INITIAL(0,'         ','CIN',0,0,0,0),
    DEBUGGING       BYTE        INITIAL (FALSE),
    PRINT$PROD      BYTE        INITIAL(FALSE),
    PRINT$TOKEN     BYTE        INITIAL(FALSE),
    LIST$INPUT      BYTE        INITIAL (TRUE),
    SEQ$NUM         BYTE        INITIAL (FALSE),
    NEXT$SYM        ADDRESS,
    POINTER         ADDRESS     INITIAL (100H),
    NEXT$AVAILABLE  ADDRESS     INITIAL (3002H),
    MAX$INT$MEM     ADDRESS     INITIAL (0D100H),
    FREE$STORAGE    ADDRESS,
    FILE$SEC$END    BYTE        INITIAL (FALSE),

    /*  I O BUFFERS AND GLOBALS */
    IN$ADDR ADDRESS INITIAL (5CH),
    INPUT$FCB BASED INADDR (33) BYTE,
    OUTPUT$PTR  ADDRESS,
    OUTPUT$BUFF (128) BYTE,
    OUTPUT$END ADDRESS,
    OUTPUT$CHAR BASED OUTPUT$PTR BYTE;


MON1: PROCEDURE (F,A) EXTERNAL;
    DECLARE A ADDRESS, F BYTE;
END MON1;

MON2: PROCEDURE (F,A) BYTE EXTERNAL;
    DECLARE F BYTE, A ADDRESS;
END MON2;

BOOT: PROCEDURE EXTERNAL;
    DECLARE A ADDRESS;
    END BOOT;

PRINTCHAR: PROCEDURE (CHAR);
    DECLARE CHAR BYTE;
    CALL MON1 (2,CHAR);
END PRINTCHAR;

CRLF: PROCEDURE;
    CALL PRINTCHAR(CR);
    CALL PRINTCHAR(LF);
END CRLF;

PRINT: PROCEDURE (A);
    DECLARE A ADDRESS;
    CALL MON1 (9,A);
END PRINT;

PRINT$ERROR: PROCEDURE (CODE);
/* THIS PROCEDURE IS USED TO PRINT COMPILER ERRORS TO
   CONSOL */
    DECLARE CODE     ADDRESS,
             I       BYTE,
```

```
            CODE1(6) ADDRESS;
    IF CODE = FALSE THEN
    DO;
        DO I = 0 TO 5;
            CODE1(I) = 0;
        END;
        I = 0;
    END;
    ELSE
    IF CODE = TRUE THEN
    DO;
        I = 0;
        DO WHILE((I <> 6) AND ( CODE1(I) <> 0));
            CALL CRLF;
            CALL PRINTCHAR(HIGH(CODE1(I)));
            CALL PRINTCHAR(LOW (CODE1(I)));
            CODE1(I) = 0;
            I = I + 1;
        END;
        I = 0;
    END;
    ELSE
    IF (CODE = 'NP') OR (CODE = 'SL') OR (CODE = 'NV') THEN
    DO;
        CALL CRLF;
        CALL PRINTCHAR(HIGH(CODE));
        CALL PRINTCHAR(LOW(CODE));
    END;
    ELSE
    DO;
        IF I <> 6 THEN
         DO;
             CODE1(I) = CODE;
             I = I + 1;
         END;
   END;
END PRINT$ERROR;

FATAL$ERROR: PROCEDURE(REASON);
    DECLARE REASON ADDRESS;
    CALL PRINT$ERROR(REASON);
    CALL PRINT$ERROR(TRUE);
    CALL TIME(10);
    CALL BOOT;
END FATAL$ERROR;

OPEN: PROCEDURE;
    IF MON2 (15,IN$ADDR)=255 THEN CALL FATAL$ERROR('OP');
END OPEN;

MORE$INPUT: PROCEDURE BYTE;
    /* READS THE INPUT FILE AND RETURNS TRUE IF A RECORD
         WAS READ.  FALSE IMPLIES END OF FILE */
    DECLARE DCNT BYTE;
```

```
    IF (DCNT:=MON2(20,.INPUT$FCB))>1
       THEN CALL FATAL$ERROR('BR');
    RETURN NOT(DCNT);
END MORE$INPUT;

MAKE: PROCEDURE;
    /* DELETES ANY EXISTING COPY OF THE OUTPUT FILE
         AND CREATES A NEW COPY*/
    CALL MON1(19,.OUTPUT$FCB);
    IF MON2(22,.OUTPUT$FCB)=255 THEN CALL FATAL$ERROR('MA');
END MAKE;

WRITE$OUTPUT: PROCEDURE;
    /* WRITES OUT A BUFFER */
    CALL MON1(26,.OUTPUT$BUFF);     /* SET DMA */
    IF MON2(21,.OUTPUT$FCB)<>0 THEN CALL FATAL$ERROR('WR');
    CALL MON1(26,80H);     /* RESET DMA */
END WRITE$OUTPUT;

MOVE: PROCEDURE(SOURCE, DESTINATION, COUNT);
    /* MOVES FOR THE NUMBER OF BYTES SPECIFIED BY COUNT */
    DECLARE (SOURCE,DESTINATION) ADDRESS,
    (S$BYTE BASED SOURCE, D$BYTE BASED DESTINATION, COUNT)
       BYTE;
    DO WHILE (COUNT:=COUNT - 1) <> 255;
         D$BYTE=S$BYTE;
         SOURCE=SOURCE +1;
         DESTINATION = DESTINATION + 1;
    END;
END MOVE;

FILL: PROCEDURE(ADDR,CHAR,COUNT);
    /* MOVES CHAR INTO ADDR FOR COUNT BYTES */
    DECLARE ADDR ADDRESS,
    (CHAR,COUNT,DEST BASED ADDR) BYTE;
    DO WHILE (COUNT:=COUNT -1)<>255;
         DEST=CHAR;
         ADDR=ADDR + 1;
    END;
END FILL;

    /*   *  *  *  *  *  SCANNER LITS *  *  *  *  */
DECLARE
    LITERAL         LIT       '15',
    INPUT$STR       LIT       '32',
    PERIOD          LIT       '1',
    INVALID         LIT       '0';

    /*  * * * * SCANNER TABLES * * * *    */
DECLARE TOKEN$TABLE (*) BYTE DATA
    /* CONTAINS THE TOKEN NUMBER ONE LESS THAN THE
    FIRST RESERVED WORD FOR EACH LENGTH OF WORD  */
    (0,0,1,4,5,15,22,32,38,44,47,49,51,55,56,57),
```

```
TABLE (*) BYTE DATA('FD','OF','TO','PIC','COMP','DATA','FILE'
,'LEFT','MODE','SAME','SIGN','SYNC','ZERO','BLOCK','LABEL'
,'QUOTE','RIGHT','SPACE','USAGE','VALUE','ACCESS','ASSIGN'
,'AUTHOR','FILLER','OCCURS','RANDOM','RECORD','SELECT'
,'DISPLAY','LEADING','LINKAGE','OMITTED','RECORDS'
,'SECTION','DIVISION','RELATIVE','SECURITY','SEPARATE'
,'STANDARD','TRAILING','DEBUGGING','PROCEDURE','REDEFINES'
,'PROGRAM-ID','SEQUENTIAL','ENVIRONMENT','I-O-CONTROL'
,'DATE-WRITTEN','FILE-CONTROL','INPUT-OUTPUT','ORGANIZATION'
,'CONFIGURATION','IDENTIFICATION','OBJECT-COMPUTER'
,'SOURCE-COMPUTER','WORKING-STORAGE'),

OFFSET  (16) ADDRESS
    /*  NUMBER OF BYTES TO INDEX INTO THE TABLE
        FOR EACH LENGTH */
    INITIAL (0,0,0,6,9,45,80,128,170,218,245,265,
         287,335,348,362),

WORD$COUNT (*) BYTE DATA
    /* NUMBER OF WORDS OF EACH SIZE */
    (0,0,3,1,9,7,8,6,6,3,2,2,4,1,1,3),

    MAX$LEN         LIT         '16',
    ADD$END(*) BYTE DATA         ('PROCEDURE '),
    LOOKED          BYTE        INITIAL (0),
    HOLD            BYTE,
    BUFFER$END      ADDRESS     INITIAL   (100H),
    NEXT            BASED        POINTER  BYTE,
    INBUFF          LIT         '80H',
    CHAR            BYTE,
    ACCUM$LENG      LIT         '50',
    ACCUM$LEN$P$1   LIT         '51',
    /* = TO ACCUM$LENG PLUS 1 */
    ACCUM (ACCUM$LEN$P$1) BYTE,
    DISPLAY(74)     BYTE        INITIAL (0),
    TOKEN           BYTE,            /*RETURNED FROM SCANNER */
    EDIT$FLAG       BYTE      INITIAL(FALSE);


    /* * * * *   PROCEDURES USED BY THE SCANNER * * * */

NEXT$CHAR: PROCEDURE BYTE;
    IF LOOKED THEN
    DO;
         LOOKED=FALSE;
         RETURN (CHAR:=HOLD);
    END;
    IF (POINTER:=POINTER + 1) >= BUFFER$END THEN
    DO;
        IF NOT MORE$INPUT THEN
         DO;
              BUFFER$END=.MEMORY;
              POINTER=.ADD$END;
         END;
```

```
        ELSE POINTER=INBUFF;
    END;
    IF NEXT = EOFFILLER THEN
    DO;
        BUFFER$END = .MEMORY;
        POINTER = .ADD$END;
    END;
    RETURN (CHAR:=NEXT);
END NEXT$CHAR;

GET$CHAR: PROCEDURE;
    /* THIS PROCEDURE IS CALLED WHEN A NEW CHAR IS
    NEEDED WITHOUT THE DIRECT RETURN OF THE CHARACTER*/
    CHAR=NEXT$CHAR;
END GET$CHAR;

DISPLAY$LINE: PROCEDURE;
    DECLARE  I BYTE;
    IF NOT LIST$INPUT THEN RETURN;
    IF NOT EDIT$FLAG THEN
      DO;
      DISPLAY(DISPLAY(0) + 1) = '$';
      CALL PRINT(.DISPLAY(1));
      END;
    ELSE DO I = 1 TO DISPLAY(0);
           CALL PRINTCHAR(DISPLAY(I));
         END;
    DISPLAY(0) = 0;
    EDIT$FLAG = FALSE;
END DISPLAY$LINE;

LOAD$DISPLAY: PROCEDURE;
    IF DISPLAY(0) < 72 THEN
        DISPLAY(DISPLAY(0):=DISPLAY(0) + 1) = CHAR;
    IF CHAR = '$' THEN EDIT$FLAG = TRUE;
    CALL GET$CHAR;
END LOAD$DISPLAY;

PUT: PROCEDURE;
    IF ACCUM(0) < ACCUM$LENG THEN
    ACCUM(ACCUM(0):=ACCUM(0)+1)=CHAR;
    CALL LOAD$DISPLAY;
END PUT;

EAT$LINE: PROCEDURE;
    DO WHILE CHAR<>CR;
        CALL LOAD$DISPLAY;
    END;
END EAT$LINE;

GET$NO$BLANK: PROCEDURE;
    DECLARE (N,I) BYTE;
    DO FOREVER;
        IF CHAR = ' ' THEN CALL LOAD$DISPLAY;
```

```
          ELSE
          IF CHAR=CR THEN
          DO;
               CALL DISPLAY$LINE;
               CALL PRINT$ERROR(TRUE);
               IF SEQ$NUM THEN N=8; ELSE N=2;
               DO I = 1 TO N;
                    CALL LOAD$DISPLAY;
               END;
               IF CHAR = '*' THEN CALL EAT$LINE;
               ELSE
               IF CHAR = ':' THEN
                    DO;
                         IF NOT DEBUGGING THEN CALL EAT$LINE;
                         ELSE CALL LOAD$DISPLAY;
                    END;
          END;
          ELSE
          RETURN;
     END;    /* END OF DO FOREVER */
END GET$NO$BLANK;

SPACE: PROCEDURE BYTE;
     RETURN (CHAR=' ') OR (CHAR=CR);
END SPACE;

DELIMITER: PROCEDURE BYTE;
     /* CHECKS FOR A PERIOD FOLLOWED BY A SPACE OR CR*/
     IF CHAR <> '.' THEN RETURN FALSE;
     HOLD=NEXT$CHAR;
     LOOKED=TRUE;
     IF SPACE THEN
     DO;
          CHAR = '.';
          RETURN TRUE;
     END;
     CHAR='.';
     RETURN FALSE;
END DELIMITER;

END$OF$TOKEN: PROCEDURE BYTE;
     RETURN SPACE OR  DELIMITER;
END END$OF$TOKEN;

GET$LITERAL: PROCEDURE BYTE;
     CALL LOAD$DISPLAY;
     DO FOREVER;
          IF CHAR= QUOTE THEN
          DO;
               CALL LOAD$DISPLAY;
               RETURN LITERAL;
          END;
          CALL PUT;
     END;
```

```
END GET$LITERAL;

LOOK$UP: PROCEDURE BYTE;
    DECLARE POINT ADDRESS,
    HERE BASED POINT (1) BYTE,
    I                    BYTE;

    MATCH: PROCEDURE BYTE;
         DECLARE J BYTE;
         DO J=1 TO ACCUM(0);
              IF HERE(J - 1) <> ACCUM(J) THEN RETURN FALSE;
         END;
         RETURN TRUE;
    END MATCH;

    POINT=OFFSET(ACCUM(0))+ .TABLE;
    DO I=1 TO WORD$COUNT(ACCUM(0));
         IF MATCH THEN RETURN I;
         POINT = POINT + ACCUM(0);
    END;
    RETURN FALSE;
END LOOK$UP;

RESERVED$WORD: PROCEDURE BYTE;
    /* RETURNS THE TOKEN NUMBER OF A RESERVED WORD IF THE
    CONTENTS OF THE ACCUMULATOR IS A RESERVED WORD, OTHERWISE
    RETURNS ZERO */
    DECLARE VALUE BYTE;
    DECLARE NUMB BYTE;
         IF ACCUM(0) > MAX$LEN THEN RETURN 0;
         IF (NUMB:=TOKEN$TABLE(ACCUM(0)))=0 THEN RETURN 0;
         IF (VALUE:=LOOK$UP)=0 THEN RETURN 0;
         RETURN (NUMB + VALUE);
END RESERVED$WORD;

GET$TOKEN: PROCEDURE BYTE;
    ACCUM(0)=0;
    CALL GET$NO$BLANK;
    IF CHAR=QUOTE THEN RETURN GET$LITERAL;
    IF DELIMITER THEN
    DO;
         CALL PUT;
         RETURN PERIOD;
    END;
    DO FOREVER;
         CALL PUT;
         IF END$OF$TOKEN THEN RETURN INPUT$STR;
    END; /* OF DO FOREVER */
END GET$TOKEN;

SCANNER: PROCEDURE;
    DECLARE CHECK BYTE;
    DO FOREVER;
         IF(TOKEN:=GET$TOKEN) = INPUT$STR THEN
```

```
          IF (CHECK:=RESERVED$WORD) <> 0 THEN TOKEN=CHECK;
        IF TOKEN <> 0 THEN RETURN;
        CALL PRINT$ERROR ('SE');
        DO WHILE NOT END$OF$TOKEN;
             CALL GET$CHAR;
        END;
    END;
END SCANNER;

PRINT$ACCUM: PROCEDURE;
    ACCUM(ACCUM(0)+1)='$';
    CALL PRINT(.ACCUM(1));
END PRINT$ACCUM;

PRINT$NUMBER: PROCEDURE(NUMB);
    DECLARE(NUMB,I,CNT,K) BYTE, J(*) BYTE DATA(100,10);
    DO I=0 TO 1;
        CNT=0;
        DO WHILE NUMB >= (K:=J(I));
             NUMB=NUMB - K;
             CNT=CNT + 1;
        END;
        CALL PRINTCHAR('0' + CNT);
    END;
    CALL PRINTCHAR('0' + NUMB);
END PRINT$NUMBER;

INIT$SCANNER: PROCEDURE;
    /*    INITIALIZE FOR INPUT - OUTPUT OPERATIONS    */
    DECLARE CON$CBL (*) BYTE DATA ('CBL'),
            I           BYTE,
            TESTFLAG    BYTE;
    CALL MOVE(PARMS,.PARMLIST,8);
    IF PARMLIST(0) = '$' THEN
    DO;
     I = 0;
     DO WHILE (TESTFLAG:=PARMLIST(I:=I+1)) <> ' ';
       IF TESTFLAG = 'L' THEN LIST$INPUT=NOT LIST$INPUT;
       IF TESTFLAG = 'S' THEN SEQ$NUM= NOT SEQ$NUM;
       IF TESTFLAG = 'P' THEN PRINT$PROD = NOT PRINT$PROD;
       IF TESTFLAG = 'T' THEN PRINT$TOKEN = NOT PRINT$TOKEN;
     END;
    END;
    CALL MOVE (.CON$CBL, IN$ADDR + 9, 3);
    CALL FILL(IN$ADDR + 12,0,5);
    CALL OPEN;
    CALL MOVE(INADDR,.OUTPUT$FCB,9);
    OUTPUT$FCB(32) = 0;
    OUTPUT$END=(OUTPUT$PTR:=.OUTPUT$BUFF - 1) + 128;
    CALL MAKE;
    CALL GET$CHAR;    /* PRIME THE SCANNER */
    IF SEQ$NUM THEN
    DO I = 1 TO 6;
        CALL LOAD$DISPLAY;
```

```
    END;
    IF CHAR = '*' THEN CALL EAT$LINE;
    CALL GET$NO$BLANK;
    CALL PRINT$ERROR(FALSE);    /* INITIALIZES ERROR
                                   MSG OUTPUT */
END INIT$SCANNER;

    /*  *  *  *  END OF SCANNER PROCEDURES  *  *  *  */

    /*  *  *  *  *  SYMBOL TABLE DECLARATIONS * * *  */

DECLARE
CUR$SYM             ADDRESS,        /*SYMBOL BEING ACCESSED*/
SYMBOL              BASED CUR$SYM (1)  BYTE,
SYMBOL$ADDR         BASED CUR$SYM (1) ADDRESS,
NEXT$SYM$ENTRY      BASED NEXT$SYM ADDRESS,
HASH$PTR            ADDRESS,
SAVE$ADDR           ADDRESS,
DISPLACEMENT        LIT             '13',
HASH$MASK           LIT             '3FH',
S$TYPE              LIT             '2',
OCCURS              LIT             '12',
ADDR2               LIT             '4',
P$LENGTH            LIT             '3',
S$LENGTH            LIT             '3',
LEVEL               LIT             '10',
DECIMAL             LIT             '11',
LOCATION            LIT             '2',
REL$ID              LIT             '5',
START$NAME          LIT             '12',  /*1 LESS*/
MAX$ID$LEN          LIT             '12';

    /*  *  *  *  *  TYPE LITERALS * * * * * * * * */

DECLARE
SEQUENTIAL     LIT      '1',
SEQ$RELATIVE   LIT      '2',
RANDOM         LIT      '3',
VARIABLE$LENG  LIT      '4',
GROUP          LIT      '6',
COMP           LIT      '21';

    /*  *  *  *  SYMBOL TABLE ROUTINES  *  *  *  */

INIT$SYMBOL: PROCEDURE;
    /* INITIALIZE HASH TABLE AND FIRST COLLISION FIELD */
    FREE$STORAGE = .MEMORY;
    CALL FILL (FREE$STORAGE,0,130);
    NEXT$SYM=FREE$STORAGE+128;
    NEXT$SYM$ENTRY=0;
END INIT$SYMBOL;

GET$P$LENGTH: PROCEDURE BYTE;
    RETURN SYMBOL(P$LENGTH);
```

```
END GET$P$LENGTH;

SET$ADDRESS: PROCEDURE(ADDR);
DECLARE ADDR ADDRESS;
    SYMBOL$ADDR(LOCATION)=ADDR;
END SET$ADDRESS;

GET$ADDRESS: PROCEDURE ADDRESS;
    RETURN SYMBOL$ADDR(LOCATION);
END GET$ADDRESS;

GET$TYPE: PROCEDURE BYTE;
    RETURN SYMBOL(S$TYPE);
END GET$TYPE;

SET$TYPE: PROCEDURE(TYPE);
    DECLARE TYPE BYTE;
    SYMBOL(S$TYPE)=TYPE;
END SET$TYPE;

OR$TYPE: PROCEDURE(TYPE);
    DECLARE TYPE BYTE;
    SYMBOL(S$TYPE)=TYPE OR GET$TYPE;
END OR$TYPE;

GET$LEVEL: PROCEDURE BYTE;
    RETURN SYMBOL(LEVEL);
END GET$LEVEL;

SET$LEVEL: PROCEDURE (LVL);
    DECLARE LVL BYTE;
    SYMBOL(LEVEL)=LVL;
END SET$LEVEL;

GET$DECIMAL: PROCEDURE BYTE;
    RETURN SYMBOL(DECIMAL);
END GET$DECIMAL;

SET$DECIMAL: PROCEDURE (DEC);
    DECLARE DEC BYTE;
    SYMBOL(DECIMAL)=DEC;
END SET$DECIMAL;

SET$S$LENGTH: PROCEDURE(HOW$LONG);
    DECLARE HOW$LONG ADDRESS;
    SYMBOL$ADDR(S$LENGTH) = HOW$LONG;
END SET$S$LENGTH;

GET$S$LENGTH: PROCEDURE ADDRESS;
    RETURN SYMBOL$ADDR(S$LENGTH);
END GET$S$LENGTH;

SET$ADDR2: PROCEDURE (ADDR);
    DECLARE ADDR ADDRESS;
```

```
SYMBOL$ADDR(ADDR2)=ADDR;
END SET$ADDR2;

GET$ADDR2: PROCEDURE ADDRESS;
    RETURN SYMBOL$ADDR(ADDR2);
END GET$ADDR2;

SET$OCCURS: PRCCEDURE(OCCUR);
    DECLARE OCCUR BYTE;
    SYMBOL(OCCURS)=OCCUR;
END SET$OCCURS;

GET$OCCURS: PROCEDURE BYTE;
    RETURN SYMBOL (OCCURS);
END GET$OCCURS;

SET$IO$ADDRS: PROCEDURE;
    SYMBOL$ADDR(LOCATION) = NEXT$SYM;
    SAVE$ADDR = CUR$SYM;
END SET$IO$ADDRS;
    /*  *  *  *  PARSER DECLARATIONS  *  *  *  */
DECLARE
INT                 LIT    '63',  /* CODE FOR INITIALIZE */
SCD                 LIT    '66', /* CODE FOR SET CODE START */
PSTACKSIZE          LIT    '30', /* SIZE OF PARSE STACKS*/
STATESTACK          (PSTACKSIZE) BYTE, /* SAVED STATES */
VALUE               (PSTACKSIZE) ADDRESS, /* TEMP VALUES */
VARC                (51)          BYTE, /*TEMP CHAR STORE*/
ID$STACK            (10)       ADDRESS    INITIAL (0),
ID$STACK$PTR        BYTE      INITIAL(0),
HOLD$LIT (ACCUM$LEN$P$1)    BYTE,
HOLD$SYM            ADDRESS,
PENDING$LITERAL     BYTE INITIAL(FALSE),
PENDING$LIT$ID      ADDRESS,
REDEF               BYTE      INITIAL (FALSE),
REDEF$ONE           ADDRESS,
REDEF$TWO           ADDRESS,
TEMP$HOLD           ADDRESS,
TEMP$TWO            ADDRESS,
COMPILING           BYTE    INITIAL(TRUE),
SP                  BYTE    INITIAL (255),
MP                  BYTE,
MPP1                BYTE,
NOLOOK              BYTE    INITIAL(TRUE),
(I,J,K)             BYTE,        /*INDICIES FOR THE PARSER*/
STATE               BYTE    INITIAL(STARTS),
VALUE$FLAG          BYTE    INITIAL(FALSE),
VALUE$LEVEL         BYTE    INITIAL(0),
TRUNC$FLAG          BYTE    INITIAL(TRUE);

    /*  *  *  *  PARSER ROUTINES  *  *  *  *  */

BYTE$OUT: PROCEDURE(ONE$BYTE);
/* THIS PROCEDURE WRITES ONE BYTE OF OUTPUT ONTO THE DISK
```

```
IF REQUIRED THE OUTPUT BUFFER IS DUMPED TO THE DISK */
    DECLARE ONE$BYTE BYTE;
    IF (OUTPUT$PTR:=OUTPUT$PTR + 1)> OUTPUT$END THEN
    DO;
         CALL WRITE$OUTPUT;
         OUTPUT$PTR=.OUTPUT$BUFF;
    END;
    OUTPUT$CHAR=ONE$BYTE;
END BYTE$OUT;

STRING$OUT: PROCEDURE (ADDR,COUNT);
    DECLARE (ADDR,I,COUNT) ADDRESS, (CHAR BASED ADDR) BYTE;
    DO I=1 TO COUNT;
         CALL BYTE$OUT(CHAR);
         ADDR=ADDR+1;
    END;
END STRING$OUT;

ADDR$OUT: PROCEDURE(ADDR);
    DECLARE ADDR ADDRESS;
    CALL BYTE$OUT(LOW(ADDR));
    CALL BYTE$OUT(HIGH(ADDR));
END ADDR$OUT;

FILL$STRING: PROCEDURE(COUNT,CHAR);
    DECLARE (I,COUNT) ADDRESS, CHAR BYTE;
    DO I=1 TO COUNT;
         CALL BYTE$OUT(CHAR);
    END;
END FILL$STRING;

START$INITIALIZE: PROCEDURE(ADDR,CNT);
    DECLARE (ADDR,CNT) ADDRESS;
    CALL BYTEOUT(INT);
    CALL ADDR$OUT(ADDR);
    CALL ADDR$OUT(CNT);
END START$INITIALIZE;

BUILD$SYMBOL: PROCEDURE(LEN);
    DECLARE LEN BYTE, TEMP ADDRESS;
    TEMP=NEXT$SYM;
    IF (NEXT$SYM:=.SYMBOL(LEN:=LEN+DISPLACEMENT))
         > MAX$MEMORY THEN CALL FATAL$ERROR('ST');
    CALL FILL (TEMP,0,LEN);
END BUILD$SYMBOL;

DUP$IDEN$TEST:   PROCEDURE;
DECLARE   I    BYTE;

IF REDEF$FLAG THEN
DO;
     REDEF$FLAG = FALSE;
     RETURN;
END;
```

```
ELSE
IF FILE$DESC$FLAG THEN
DO;
     FILE$DESC$FLAG = FALSE;
     I = Ø;
     DO WHILE DUP$IDEN$ARRAY(I) <> Ø;
          IF DUP$IDEN$ARRAY(I) = CUR$SYM THEN
          DO;
               CALL PRINT$ERROR('DI');
               RETURN;
          END;
          I = I + 1;
          IF I > 23 THEN
          DO;
           CALL PRINT$ERROR('EF');
           RETURN;
          END;
     END;
     DUP$IDEN$ARRAY(I) = CURSYM;
     RETURN;
END;
ELSE
    CALL PRINT$ERROR('DI');
END DUP$IDEN$TEST;

MATCH: PROCEDURE ADDRESS;
    /* CHECKS AN IDENTIFIER TO SEE IF IT IS IN THE SYMBOL
    TABLE.  IF IT IS PRESENT, CUR$SYM IS SET FOR ACCESS.
    OTHERWISE A NEW ENTRY IS MADE AND THE PRINT NAME
    IS ENTERED.  ALL NAMES ARE TRUNCATED TO MAX$ID$LEN*/
    DECLARE POINT                  ADDRESS,
            COLLISION BASED POINT ADDRESS,
            (HOLD,I)               BYTE;
    IF VARC(Ø)>MAX$ID$LEN
         THEN VARC(Ø) = MAX$ID$LEN;
         /* TRUNCATE IF REQUIRED */
    HOLD = Ø;
    DO I=1 TO VARC(Ø);     /* CALCULATE HASH CODE */
         HOLD=HOLD + VARC(I);
    END;
    POINT=FREE$STORAGE + SHL((HOLD AND HASH$MASK),1);
    DO FOREVER;
        IF COLLISION=Ø THEN
         DO;
               IF FILE$DESC$FLAG THEN
               DO;
                   FILE$DESC$FLAG = FALSE;
                   CALL PRINT$ERROR('UI');
               END;
               ELSE
               IF REDEF$FLAG THEN
               DO;
                   REDEF$FLAG = FALSE;
                   CALL PRINT$ERROR('UI');
```

```
                END;
                CUR$SYM,COLLISION=NEXT$SYM;
                CALL BUILD$SYMBOL(VARC(0));
                /* LOAD PRINT NAME */
                SYMBOL(P$LENGTH)=VARC(0);
                DO I = 1 TO VARC(0);
                     SYMBOL(START$NAME + I)=VARC(I);
                END;
                RETURN CUR$SYM;
           END;
           ELSE
           DO;
                CUR$SYM=COLLISION;
                IF (HOLD:=GET$P$LENGTH)=VARC(0) THEN
                DO;
                     I=1;
                     DO WHILE SYMBOL(START$NAME + I)= VARC(I);
                     IF (I:=I+1)>HOLD THEN
                     DO;
                           CALL DUP$IDEN$TEST;
                           RETURN (CUR$SYM:=COLLISION);
                     END;
                     END;
               END;
          END;
          POINT=COLLISION;
     END;
END MATCH;

ALLOCATE: PROCEDURE(BYTES$REQ) ADDRESS;
     /* THIS ROUTINE CONTROLS THE ALLOCATION OF SPACE
     IN THE MEMORY OF THE INTERPRETER.  */

     DECLARE (HOLD,BYTES$REQ) ADDRESS;
     HOLD=NEXT$AVAILABLE;
     IF (NEXT$AVAILABLE:=NEXT$AVAILABLE + BYTES$REQ)
        >MAX$INT$MEM
          THEN CALL FATAL$ERROR('MO');
     RETURN HOLD;
END ALLOCATE;

DIGIT: PROCEDURE (CHAR) BYTE;
     DECLARE CHAR BYTE;
     RETURN (CHAR <= '9') AND (CHAR >= '0');
END DIGIT;

SET$REDEF: PROCEDURE(OLD,NEW);
     DECLARE (OLD,NEW) ADDRESS;
          REDEF$ONE=OLD;
          REDEF$TWO=NEW;
          REDEF=TRUE;
END SET$REDEF;

SET$CUR$SYM: PROCEDURE;
```

```
    CUR$SYM=ID$STACK(ID$STACK$PTR);
END SET$CUR$SYM;

STACK$LEVEL: PROCEDURE BYTE;
    CALL SET$CUR$SYM;
    RETURN GET$LEVEL;
END STACK$LEVEL;

LOAD$LEVEL: PROCEDURE;
    DECLARE HOLD ADDRESS;

    LOAD$REDEF$ADDR: PROCEDURE;
         CUR$SYM=REDEF$ONE;
         HOLD=GET$ADDRESS;
    END LOAD$REDEF$ADDR;

    IF ID$STACK(0) <> 0 THEN
    DO;
         IF VALUE(SP-2)=0 THEN
         DO;
              CALL SET$CUR$SYM;
              HOLD=GET$S$LENGTH + GET$ADDRESS;
         END;
         ELSE DO;
              IF FILE$SEC$END THEN
                 DO;
                 IF ID$STACK(ID$STACK$PTR) <> REDEF$ONE
                 THEN
                    DO;
                    CALL PRINT$ERROR('R1');
                    REDEF$ONE=ID$STACK(ID$STACK$PTR);
                    END;
                 END;
              CALL LOAD$REDEF$ADDR;
              END;
         IF (ID$STACK$PTR:=ID$STACK$PTR+1)>9 THEN
         DO;
              CALL PRINT$ERROR('EL');
              ID$STACK$PTR=9;
         END;
    END;
    ELSE HOLD=NEXT$AVAILABLE;
    ID$STACK(ID$STACK$PTR)=VALUE(MPP1);
    CALL SET$CUR$SYM;
    IF (GET$LEVEL = 1) AND (NOT FILE$SEC$END) THEN
        CALL SET$ADDR2(SAVE$ADDR);
    CALL SET$ADDRESS(HOLD);
END LOAD$LEVEL;

REDEF$OR$VALUE: PROCEDURE;
    DECLARE HOLD ADDRESS,
         (DEC,K,J,SIGN,CHAR) BYTE;
    IF REDEF THEN
      DO;
```

```
      IF REDEF$TWO=CUR$SYM THEN
        DO;
        HOLD=GET$S$LENGTH;
        CUR$SYM=REDEF$ONE;
        IF HOLD>GET$S$LENGTH THEN
          DO;
          CALL PRINT$ERROR('R2');
          HOLD=GET$S$LENGTH;
          CUR$SYM=REDEF$TWO;
          CALL SET$S$LENGTH(HOLD);
          END;
        END;
  END;
ELSE IF PENDING$LITERAL=0 THEN RETURN;
IF (PENDING$LIT$ID<>ID$STACK$PTR) OR VALUE$FLAG
THEN RETURN;
IF PENDING$LITERAL <> 0 THEN
 CALL START$INITIALIZE(GET$ADDRESS,HOLD:=GET$S$LENGTH);
IF PENDING$LITERAL>2 THEN
  DO;
  IF PENDING$LITERAL=3 THEN CHAR='0';
  ELSE IF PENDING$LITERAL=4 THEN CHAR=' ';
  ELSE IF PENDING$LITERAL = 5 THEN CHAR = QUOTE;
  CALL FILL$STRING(HOLD,CHAR);
  END;
ELSE IF PENDING$LITERAL = 2 THEN
        DO;
        IF HOLD <= HOLD$LIT(0) THEN
           CALL STRING$OUT(.HOLD$LIT(1),HOLD);
        ELSE DO;
             CALL STRING$OUT(.HOLD$LIT(1),HOLD$LIT(0));
             CALL FILL$STRING(HOLD - HOLD$LIT(0),' ');
             END;
        END;
ELSE IF PENDING$LITERAL=1 THEN
      DO;
      /* THE NUMBER HANDELER */
      DECLARE (DEC,MINUS$SIGN,I,J,LIT$DEC,N$LENGTH,
           NUM$BEFORE,NUM$AFTER, TYPE) BYTE,
           ZONE LIT '10H';

      IF((TYPE:=GET$TYPE)<16) OR (TYPE>21) THEN
           CALL PRINT$ERROR('NV');
      N$LENGTH=GET$S$LENGTH;
      DEC=GET$DECIMAL;
      MINUS$SIGN=FALSE;
      IF HOLD$LIT(1) = '-' THEN
      DO;
           MINUS$SIGN=TRUE;
           J=1;
      END;
      ELSE IF HOLD$LIT(1) = '+' THEN J=1;
           ELSE J=0;
      LIT$DEC=0;
```

```
        DO I=1 TO HOLD$LIT(0);
             IF HOLD$LIT(I)='.' THEN LIT$DEC=I;
        END;
        IF HOLD$LIT(0) <> 0 THEN
          DO;
          IF LIT$DEC=0 THEN
             DO;
             NUM$BEFORE=HOLD$LIT(0)-J;
             NUM$AFTER=0;
             END;
          ELSE DO;
               NUM$BEFORE=LIT$DEC -J-1;
               NUM$AFTER=HOLD$LIT(0) - LIT$DEC;
               END;
          END;
        ELSE IF HOLD$LIT(0) = 0 THEN
               DO;
               NUM$BEFORE = 0;
               NUM$AFTER = 0;
               LIT$DEC = 0;
               END;
        IF (I:=N$LENGTH - DEC)<NUM$BEFORE THEN
             CALL PRINT$ERROR('SL');
        IF I>NUM$BEFORE THEN
             DO;
             I=I-NUM$BEFORE;
             IF MINUS$SIGN THEN
                  DO;
                  I=I-1;
                  CALL BYTE$OUT('0' + ZONE);
                  END;
             CALL FILL$STRING(I,'0');
             END;
        ELSE IF MINUS$SIGN THEN HOLD$LIT(J+1)
                 =HOLD$LIT(J+1)+ZONE;
        CALL STRING$OUT(.HOLD$LIT(1)+J,NUM$BEFORE);
        IF NUM$AFTER > DEC THEN NUM$AFTER = DEC;
        CALL STRING$OUT(.HOLD$LIT(1) + LIT$DEC, NUM$AFTER);
        IF (I:=DEC - NUM$AFTER)<>0 THEN
             CALL FILL$STRING(I,'0');
        END;
    IF NOT VALUE$FLAG THEN PENDING$LITERAL=0;
END REDEF$OR$VALUE;

REDUCE$STACK: PROCEDURE;
    DECLARE HOLD$LENGTH ADDRESS;
    CALL SET$CUR$SYM;
    CALL REDEF$OR$VALUE;
    HOLD$LENGTH=GET$S$LENGTH;
    IF GET$TYPE > 128 THEN
    DO;
         HOLD$LENGTH=HOLD$LENGTH * GET$OCCURS;
    END;
    ID$STACK$PTR=ID$STACK$PTR - 1;
```

```
    CALL SET$CUR$SYM;
    CALL SET$S$LENGTH(GET$S$LENGTH + HOLD$LENGTH);
    CALL SET$TYPE(GROUP);
END REDUCE$STACK;

END$OF$RECORD: PROCEDURE;
    DO WHILE ID$STACK$PTR <> 0;
       CALL SET$CUR$SYM;
       CALL REDEF$OR$VALUE;
       ID$STACK(ID$STACK$PTR)=0;
       ID$STACK$PTR=ID$STACK$PTR - 1;
    END;
    CALL SET$CUR$SYM;
    CALL REDEF$OR$VALUE;
    ID$STACK(0)=0;
    TEMP$HOLD=ALLOCATE(TEMP$TWO:=GET$S$LENGTH);
END  END$OF$RECORD;

CONVERT$INTEGER: PROCEDURE;
    DECLARE INTEGER ADDRESS;
    INTEGER=0;
    DO I = 1 TO VARC(0);
      IF NOT DIGIT(VARC(I)) THEN CALL PRINT$ERROR('NN');
            /* ERROR RECOVERY FOR AN 'O' WHICH SHOULD
               HAVE BEEN A ZERO--'0' */
      IF (VARC(I) = 'O') THEN VARC(I) = '0';
      INTEGER=SHL(INTEGER,3)+SHL(INTEGER,1)+(VARC(I)-'0');
    END;
    VALUE(SP)=INTEGER;
END CONVERT$INTEGER;

OR$VALUE: PROCEDURE(PTR,ATTRIB);
    DECLARE PTR BYTE, ATTRIB ADDRESS;
    VALUE(PTR)=VALUE(PTR) OR ATTRIB;
END OR$VALUE;

BUILD$FCB: PROCEDURE;
    DECLARE TEMP ADDRESS;
    DECLARE BUFFER(11) BYTE, (CHAR, I, J) BYTE;
    CALL FILL(.BUFFER,' ',11);
    J,I=0;
    DO WHILE (J < 11) AND (I< VARC(0));
         IF (CHAR:=VARC(I:=I+1))='.' THEN J=8;
         ELSE DO;
              BUFFER(J)=CHAR;
              J=J+1;
         END;
    END;
    CALL SET$ADDR2(TEMP:=ALLOCATE(165));
    CALL START$INITIALIZE(TEMP,37);
    CALL BYTE$OUT(0);
    CALL STRING$OUT(.BUFFER,11);
    CALL FILL$STRING(25,0);
    CALL OR$VALUE(SP-1,1);
```

```
END BUILD$FCB;

SET$SIGN: PROCEDURE(NUMB);
 DECLARE NUMB BYTE;
 IF GET$TYPE=17 THEN CALL SET$TYPE(VALUE(SP) + NUMB);
 ELSE CALL PRINT$ERROR('SG');
 IF VALUE(SP)<>Ø THEN CALL SET$S$LENGTH(GET$S$LENGTH + 1);
END SET$SIGN;

NUM$TRUNC: PROCEDURE;
    DECLARE I            BYTE,
            J            BYTE,
            TRUNC$TYPE   BYTE,
            TRUNC$ZERO   BYTE,
            SIGN$FLAG    BYTE,
            DEC$FLAG     BYTE;

    TRUNC$ZERO = TRUE;
    SIGN$FLAG = FALSE;
    DEC$FLAG = FALSE;
    HOLD$LIT(Ø) = Ø;
    J = 1;
    I = Ø;
 IF ((TRUNC$TYPE:=GET$TYPE)=16) OR (TRUNC$TYPE=17) OR
    (TRUNC$TYPE = 21) THEN
   DO WHILE J <= VARC(Ø);
     IF (VARC(J) <> '+') AND (VARC(J) <> '-') THEN
       DO;
       IF (VARC(J)='Ø') AND TRUNC$ZERO THEN J=J;
       ELSE IF ((VARC(J) >= 'Ø') AND (VARC(J)
             <='9')) OR
             (VARC(J) = '.') THEN
           DO;
           IF DEC$FLAG AND (VARC(J) = '.') THEN
              CALL PRINT$ERROR('MD');
           ELSE DO;
            HOLD$LIT(HOLD$LIT(Ø):=HOLD$LIT(Ø)+1)
                  =VARC(J);
                IF VARC(J) <> 'Ø' THEN TRUNC$ZERO = FALSE;
                IF VARC(J) = '.' THEN DEC$FLAG = TRUE;
                I = I + 1;
                END;
           END;
         ELSE IF ((VARC(J) < 'Ø') OR (VARC(J) > '9')) AND
            (VARC(J) <> '.') THEN CALL PRINT$ERROR('NN');
    END;
  ELSE IF SIGN$FLAG THEN CALL PRINT$ERROR('MS');
    ELSE IF (VARC(J) = '+') OR (VARC(J) = '-') THEN
      DO;
      IF TRUNC$TYPE = 16 THEN CALL PRINT$ERROR('SG');
      ELSE DO;
         HOLD$LIT(HOLD$LIT(Ø):=HOLD$LIT(Ø)+1)=VARC(J);
         SIGN$FLAG = TRUE;
         I = I + 1;
```

```
         END;
      END;
   J = J + 1;
   END;/* DO WHILE LOOP */
   HOLD$LIT(Ø) = I;
   IF ((HOLD$LIT(Ø) = 1) AND ((HOLD$LIT(1) = '+') OR
      (HOLD$LIT(1) = '-'))) OR (HOLD$LIT(Ø) = 'Ø') THEN
     DO;
     HOLD$LIT(Ø) = Ø;
     HOLD$LIT(1) = Ø;
     END;
END NUM$TRUNC;

PIC$ANALIZER: PROCEDURE;
    DECLARE   /* WORK AREAS AND VARIABLES */
    FLAG          BYTE,
    FIRST         BYTE,
    COUNT         ADDRESS,
    BUFFER (31) BYTE,
    SAVE          BYTE,
    REPITITIONS ADDRESS,
    J             ADDRESS,
    DEC$COUNT   BYTE,
    CHAR          BYTE,
    I             BYTE,
    TEMP          ADDRESS,
    TYPE          BYTE,
    DEC$FLAG    BYTE,
    K             BYTE,

    /* * * MASKS * * */
    ALPHA     LIT '1',
    A$EDIT    LIT '2',
    A$N       LIT '4',
    EDIT      LIT '8',
    NUM       LIT '16',
    NUM$EDIT LIT '32',
    DEC       LIT '64',
    SIGN      LIT '128',

    NUM$MASK        LIT       '10101111B',
    NUM$ED$MASK     LIT       '10000101B',
    S$NUM$MASK      LIT       '00101111B',
    ALPHA$MASK      LIT       '11111110B',
    A$E$MASK        LIT       '11111100B',
    A$N$MASK        LIT       '11101010B',
    A$N$E$MASK      LIT       '11100000B',

    /* TYPES */
    NETYPE LIT '80',
    NTYPE  LIT '16',
    SNTYPE LIT '17',
    ATYPE  LIT '8',
    AETYPE LIT '72',
```

```
ANTYPE LIT '9',
ANETYPE LIT '73';

INC$COUNT: PROCEDURE(SWITCH);
     DECLARE SWITCH BYTE;
     FLAG=FLAG OR SWITCH;
     IF (COUNT:=COUNT + 1) < 31 THEN BUFFER(COUNT)
     = CHAR;
END INC$COUNT;

CHECK: PROCEDURE (MASK) BYTE;
     /* THIS ROUTINE CHECKS A MASK AGINST THE
     FLAG BYTE AND RETURNS TRUE ID THE FLAG
     HAD NO BITS IN COMMON WITH THE MASK */
     DECLARE MASK BYTE;
     RETURN NOT ( (FLAG AND MASK) <> 0);
END CHECK;

PIC$ALLOCATE: PROCEDURE(AMT) ADDRESS;
     DECLARE AMT ADDRESS;
     IF (MAX$INT$MEM:=MAX$INT$MEM - AMT)
          < NEXT$AVAILABLE
          THEN CALL FATAL$ERROR ('MO');
     RETURN MAX$INT$MEM;
END PIC$ALLOCATE;

/* PROCEDURE EXECUTION STARTS HERE */

CUR$SYM = HOLD$SYM;
IF (GET$LEVEL = VALUE$LEVEL) THEN VALUE$FLAG = FALSE;
DEC$FLAG = FALSE;
COUNT,FLAG,DEC$COUNT=0;
/* CHECK FOR EXCESSIVE LENGTH */
IF VARC(0) > 30 THEN
DO;
     CALL PRINT$ERROR('PC');
     RETURN;
END;
/* SET FLAG BITS AND COUNT LENGTH */
I =1;
DO WHILE I<=VARC(0);
     IF (CHAR:=VARC(I))='A' THEN CALL INC$COUNT(ALPHA);
     ELSE IF CHAR ='B' THEN CALL INC$COUNT(A$EDIT);
     ELSE IF CHAR ='9' THEN CALL INC$COUNT(NUM);
     ELSE IF CHAR ='X' THEN CALL INC$COUNT(A$N);
     ELSE IF (CHAR='S') AND (COUNT=0) THEN
            FLAG=FLAG OR SIGN;
     ELSE IF (CHAR = 'V') AND (DEC$COUNT=0) THEN
            DO;
            DEC$COUNT = COUNT;
            DEC$FLAG = TRUE;
            END;
     ELSE IF(CHAR='/') OR (CHAR='0') THEN
          CALL INC$COUNT$(EDIT);
```

```
      ELSE IF
           (CHAR='Z') OR (CHAR=',') OR (CHAR='*') OR
           (CHAR='+') OR (CHAR='-') OR (CHAR='$') THEN
           CALL INC$COUNT(NUM$EDIT);
      ELSE IF (CHAR='.') AND (DEC$COUNT=0) THEN
             DO;
             CALL INC$COUNT(NUM$EDIT);
             DEC$COUNT=COUNT;
             DEC$FLAG = TRUE;
             END;
      ELSE IF ((CHAR='C') AND (VARC(I+1)='R')) OR
           ((CHAR='D') AND (VARC(I+1)='B')) THEN
             DO;
             CALL INC$COUNT(NUM$EDIT);
             CHAR=VARC(I:=I+1);
             CALL INC$COUNT(NUM$EDIT);
             END;
      ELSE IF (CHAR='(') AND (COUNT<>0) THEN
             DO;
             SAVE=VARC(I-1);
             REPITITIONS=0;
             DO WHILE(CHAR:=VARC(I:=I+1))<>')';
                REPITITIONS=SHL(REPITITIONS,3) +
                SHL(REPITITIONS,1) +(CHAR -'0');
             END;
             CHAR=SAVE;
             DO J=1 TO REPITITIONS-1;
                CALL INC$COUNT(0);
             END;
             END;
      ELSE DO;
           CALL PRINT$ERROR('PC');
           RETURN;
           END;
      I=I+1;
END; /* END OF DO WHILE I<= VARC */
/* AT THIS POINT THE TYPE CAN BE DETERMINED */
IF  NOT CHECK(NUM$EDIT) THEN
  DO;
     IF CHECK(NUM$ED$MASK) THEN TYPE=NETYPE;
  END;
ELSE IF CHECK(NUM$MASK) THEN TYPE=NTYPE;
ELSE IF CHECK(SNUM$MASK) THEN TYPE=SNTYPE;
ELSE IF CHECK(ALPHA$MASK) THEN TYPE = ATYPE;
ELSE IF CHECK(A$E$MASK) THEN TYPE = AETYPE;
ELSE IF CHECK(A$N$MASK) THEN TYPE=ANTYPE;
ELSE IF CHECK(A$N$E$MASK) THEN TYPE=ANETYPE;
IF TYPE=0 THEN CALL PRINT$ERROR('PC');
ELSE DO;
     IF (GET$TYPE=128) THEN CALL SET$TYPE(128+TYPE);
     ELSE CALL SET$TYPE(TYPE);
     CALL SET$SLENGTH(COUNT + GET$S$LENGTH);
     IF (TYPE AND 64) <> 0 THEN
       DO;
```

```
            CALL SET$ADDR2(TEMP:=PIC$ALLOCATE(COUNT));
            CALL START$INITIALIZE(TEMP,COUNT);
            CALL STRING$OUT(.BUFFER + 1,COUNT);
            END;
          IF (DEC$COUNT <> 0) OR DEC$FLAG THEN
            DO;
            IF (COUNT - DEC$COUNT) > 18 THEN
                  CALL PRINT$ERROR('DC');
            CALL SET$DECIMAL(COUNT - DEC$COUNT);
            END;
     END;
    IF (NOT TRUNC$FLAG) AND ((TYPE = 16) OR (TYPE = 17)) THEN
      DO;
      DO K = 0 TO HOLD$LIT(0);
        VARC(K) = HOLD$LIT(K);
      END;
      CALL NUM$TRUNC;
      TRUNC$FLAG = TRUE;
      END;
END PIC$ANALIZER;

SET$FILE$ATTRIB: PROCEDURE;
     DECLARE TEMP ADDRESS, TYPE BYTE;
     IF CUR$SYM<>VALUE(MPP1) THEN
     DO;
          TEMP=CUR$SYM;
          CUR$SYM=VALUE(MPP1);
          SYMBOL$ADDR(REL$ID)=TEMP;
     END;
     IF NOT (TEMP:=VALUE(SP-1)) THEN CALL PRINT$ERROR ('NF');
     ELSE  DO;
          IF TEMP=1 THEN TYPE=SEQUENTIAL;
          ELSE IF TEMP=15 THEN TYPE=RANDOM;
          ELSE IF (TEMP=5) OR (TEMP=13) THEN
               TYPE = SEQ$RELATIVE;
          ELSE DO;
               CALL PRINT$ERROR('IA');
                    TYPE=1;
          END;
          END;
     CALL SET$TYPE(TYPE);
END SET$FILE$ATTRIB;

LOAD$LITERAL: PROCEDURE(LIT$ONE);
     DECLARE I          BYTE,
             LIT$ONE    BYTE,
             LIT$TYPE   BYTE;

     LIT$TYPE = GET$TYPE;
     IF LIT$TYPE <> 0 THEN VALUE$FLAG = FALSE;
     ELSE DO;
          VALUE$FLAG = TRUE;
          VALUE$LEVEL = GET$LEVEL;
          END;
```

```
    IF PENDING$LITERAL <> 0 THEN CALL PRINT$ERROR ('LE');
    ELSE IF (LIT$ONE = 0) OR (LIT$TYPE = 0) THEN
           DO;
           DO I = 0 TO VARC(0);
             HOLD$LIT(I) = VARC(I);
           END;
           IF (LIT$ONE = 1) AND (LIT$TYPE = 0) THEN
                 TRUNC$FLAG = FALSE;
           END;
         ELSE IF (LIT$ONE = 1) AND ((LIT$TYPE = 16) OR
                (LIT$TYPE = 17) OR (LIT$TYPE = 21)) THEN
                   CALL NUM$TRUNC;
         ELSE IF (LIT$ONE = 1) AND ((LIT$TYPE <> 16) OR
                (LIT$TYPE <> 17) OR (LIT$TYPE <> 21)) AND
                (LIT$TYPE <> 0) THEN
               DO;
               CALL PRINT$ERROR('LV');
               DO I = 0 TO VARC(0);
                 HOLD$LIT(I) = VARC(I);
               END;
               PENDING$LITERAL = 2;
               END;
END LOAD$LITERAL;

REDEF$TEST:  PROCEDURE;
     DECLARE  SAVE$REDEF  BYTE,
              SAVE$REDEF$ONE  ADDRESS,
              SAVE$REDEF$TWO  ADDRESS;
     SAVE$REDEF$ONE = REDEF$ONE;
     SAVE$REDEF$TWO = REDEF$TWO;
     REDEF$ONE = CUR$SYM;
     CALL SET$CUR$SYM;
     REDEF$TWO = CUR$SYM;
     SAVE$REDEF = REDEF;
     REDEF = TRUE;
     CALL REDEF$OR$VALUE;
     ID$STACK(ID$STACK$PTR) = 0;
     ID$STACK$PTR = ID$STACK$PTR - 1;
     REDEF$ONE = SAVE$REDEF$ONE;
     REDEF$TWO = SAVE$REDEF$TWO;
     REDEF = SAVE$REDEF;
END REDEF$TEST;

CHECK$LVL$FILES:  PROCEDURE;
     DECLARE NEW$LEVEL  BYTE;
     HOLD$SYM,CUR$SYM=VALUE(MP-1);
     CALL SET$LEVEL(NEW$LEVEL:=VALUE(MP-2));
     IF NEW$LEVEL = 1 THEN
       DO;
       IF ID$STACK(0) <> 0 THEN
         DO;
           DO WHILE STACK$LEVEL > 1;
             CALL REDUCE$STACK;
           END;
```

```
            DO WHILE ID$STACK$PTR <> 0;
              CALL SET$CUR$SYM;
              CALL REDEF$OR$VALUE;
              ID$STACK(ID$STACK$PTR) = 0;
              ID$STACK$PTR = ID$STACK$PTR - 1;
            END;
            CUR$SYM = HOLD$SYM;
            CALL SET$REDEF(ID$STACK(0),VALUE(MP-1));
            VALUE(MP) = 1;/* SET REDEFINE FLAG */
          END;
        END;
      ELSE DO WHILE STACK$LEVEL >= NEW$LEVEL;
             CALL REDUCE$STACK;
           END;
END CHECK$LVL$FILES;

CHECK$LVL$WORK:  PROCEDURE;
      DECLARE NEW$LEVEL       BYTE,
              SAVE$SYM$LVL    BYTE,
              STACK$REDUCED   BYTE,
              SAVE$REDEF      BYTE,
              SAVE$SYM        ADDRESS;

SET$VALUE$CLAUSE: PROCEDURE;
      SAVE$REDEF = REDEF;
      REDEF = FALSE;
      CALL SET$CUR$SYM;
      CALL REDEF$OR$VALUE;
      REDEF = SAVE$REDEF;
      CUR$SYM = HOLD$SYM;
END SET$VALUE$CLAUSE;

TRUNC$FLAG = TRUE;
STACK$REDUCED = FALSE;
HOLD$SYM,CUR$SYM=VALUE(MP-1);
CALL SET$LEVEL(NEW$LEVEL:=VALUE(MP-2));
IF NEW$LEVEL = 1 THEN
DO;
DO WHILE STACK$LEVEL > 1 AND ID$STACK(ID$STACK$PTR) <>0;
SAVE$SYM,CUR$SYM=ID$STACK(ID$STACK$PTR - 1);
SAVE$SYM$LVL = GET$LEVEL;
IF SAVE$SYM$LVL = STACK$LEVEL THEN
 DO;
 CUR$SYM = SAVE$SYM;
 CALL REDEF$TEST;
 END;
ELSE IF STACK$LEVEL > 1 THEN
DO;
CALL REDUCE$STACK;
IF VALUE$FLAG AND (VALUE$LEVEL = STACK$LEVEL) THEN
DO;
VALUE$FLAG = FALSE;
CALL SET$VALUE$CLAUSE;
END;
```

```
END;
END;/* DO WHILE LOOP */
IF STACK$LEVEL = 1 AND ID$STACK$PTR <> 0 THEN
DO;
CUR$SYM = ID$STACK(ID$STACK$PTR - 1);
CALL REDEF$TEST;
END;
IF VALUE(MP) = 0 AND ID$STACK(ID$STACK$PTR) <> 0 THEN
DO;
CALL END$OF$RECORD;
REDEF = FALSE;
END;
IF (VALUE(MP) = 1) AND (ID$STACK(ID$STACK$PTR) = REDEF$ONE)
 THEN CALL SET$VALUE$CLAUSE;
CUR$SYM = HOLD$SYM;
END;
ELSE IF STACK$LEVEL >= NEW$LEVEL THEN
 DO;
IF (STACK$LEVEL = NEW$LEVEL) AND (VALUE(MP) = 1) AND
 (ID$STACK(ID$STACK$PTR) = REDEF$ONE) THEN
 CALL SET$VALUE$CLAUSE;
DO WHILE NOT STACK$REDUCED;
SAVE$SYM,CUR$SYM=ID$STACK(ID$STACK$PTR - 1);
SAVE$SYM$LVL = GET$LEVEL;
IF SAVE$SYM$LVL = STACK$LEVEL THEN
DO;
CUR$SYM = SAVE$SYM;
CALL REDEF$TEST;
END;
ELSE IF (STACK$LEVEL >= NEW$LEVEL) AND
 (VALUE(MP) = 0) THEN
   DO;
   DO WHILE STACK$LEVEL >= NEW$LEVEL;
 CALL REDUCE$STACK;
 IF VALUE$FLAG AND (VALUE$LEVEL=STACK$LEVEL)
   AND (VALUE$LEVEL = NEW$LEVEL) THEN
   DO;
   VALUE$FLAG = FALSE;
   CALL SET$VALUE$CLAUSE;
   END;
   END;/* DO WHILE LOOP */
   STACK$REDUCED = TRUE;
   END;
 ELSE IF (STACK$LEVEL >= NEW$LEVEL) AND
     (VALUE(MP) = 1) THEN
    DO;
    DO WHILE STACK$LEVEL > NEW$LEVEL;
      CALL REDUCE$STACK;
      IF VALUE$FLAG AND (VALUE$LEVEL = STACK$LEVEL)
        THEN DO;
             VALUE$FLAG = FALSE;
             CALL SET$VALUE$CLAUSE;
             END;
    END;/* DO WHILE LOOP */
```

```
    STACK$REDUCED = TRUE;
    END;
END;/* DO WHILE LOOP */
 END;
 CUR$SYM = HOLD$SYM;
END CHECK$LVL$WORK;

CODE$GEN: PROCEDURE(PRODUCTION);
    DECLARE PRODUCTION BYTE,
            LIT$TYPE   BYTE;
    IF PRINT$PROD THEN
    DO;
         CALL CRLF;
         CALL PRINTCHAR(POUND);
         CALL PRINT$NUMBER(PRODUCTION);
    END;

    DO CASE PRODUCTION;

  /*  P R O D U C T I O N S */

 /* CASE Ø NOT USED      */
       ;
/* 1 <PROGRAM> ::= <ID-DIV> <E-DIV> <D-DIV> PROCEDURE */
 DO;
 COMPILING=FALSE;
 DISPLAY(DISPLAY(Ø)+1)='$';
 CALL PRINT(.DISPLAY(1));
 END;
/* 2 <ID-DIV> ::= IDENTIFICATION DIVISION . PROGRAM-ID .*/
/* 2    <COMMENT> . <AUTH> <DATE> <SEC>  */
 ; /* NO ACTION REQUIRED */
/* 3 <AUTH> ::= AUTHOR . <COMMENT> .      */
 ; /* NO ACTION REQUIRED */
/* 4   \! <EMPTY>          */
 ; /* NO ACTION REQUIRED */
/* 5 <DATE> ::= DATE-WRITTEN . <COMMENT> .    */
 ; /* NO ACTION REQUIRED */
/* 6   \! <EMPTY>          */
 ; /* NO ACTION REQUIRED */
/* 7 <SEC> ::= SECURITY . <COMMENT> .     */
 ; /* NO ACTION REQUIRED */
/* 8   \! <EMPTY>          */
 ; /* NO ACTION REQUIRED */
/* 9 <COMMENT> ::= <INPUT>        */
 ; /* NO ACTION REQUIRED */
/*1Ø    \! <COMMENT> <INPUT>      */
 ; /* NO ACTION REQUIRED */
/*11 <E-DIV> ::= ENVIRONMENT DIVISION . CONFIGURATION */
/*11    SECTION . <SRC-OBJ> <I-O>    */
 ; /* NO ACTION REQUIRED */
/*12 <SRC-OBJ> ::= SOURCE-COMPUTER . <COMMENT> <DEBUG> .*/
/*12     OBJECT-COMPUTER . <COMMENT> .   */
 ; /* NO ACTION REQUIRED */
```

```
/*13 <DEBUG> ::= DEBUGGING MODE        */
 DEBUGGING=TRUE; /* SETS A SCANNER TOGGLE */
/*14     \! <EMPTY>              */
 ; /* NO ACTION REQUIRED */
/*15 <I-O> ::= INPUT-OUTPUT SECTION . FILE-CONTROL .  */
/*15     <FILE-CONTROL-LIST> <IC>      */
 ; /* NO ACTION REQUIRED */
/*16    \! <EMPTY>             */
 ; /* NO ACTION REQUIRED */
/*17 <FILE-CONTROL-LIST> ::= <FILE-CONTROL-ENTRY>  */
 ; /* NO ACTION REQUIRED */
/*18        \! <FILE-CONTROL-LIST>    */
/*18        <FILE-CONTROL-ENTRY>  */
 ; /* NO ACTION REQUIRED */
/*19 <FILE-CONTROL-ENTRY> ::= SELECT <ID>
                              <ATTRIBUTE-LIST> . */
 CALL SET$FILE$ATTRIB;
/*20 <ATTRIBUTE-LIST> ::= <ONE-ATTRIB>      */
 ; /* NO ACTION REQUIRED */
/*21       \! <ATTRIBUTE-LIST> <ONE-ATTRIB> */
 VALUE(MP)=VALUE(SP) OR VALUE(MP);
/*22 <ONE-ATTRIB> ::= ORGANIZATION <ORG-TYPE>    */
 VALUE(MP)=VALUE(SP);
/*23      \! ACCESS <ACC-TYPE> <RELATIVE>  */
 VALUE(MP)=VALUE(MPP1) OR VALUE(SP);
/*24      \! ASSIGN <INPUT>       */
 CALL BUILD$FCB;
/*25 <ORG-TYPE> ::= SEQUENTIAL        */
 ; /* NO ACTION REQUIRED - DEFAULT */
/*26     \! RELATIVE          */
 CALL OR$VALUE(SP,4);
/*27 <ACC-TYPE> ::= SEQUENTIAL        */
 ; /* NO ACTION REQUIRED - DEFAULT */
/*28     \! RANDOM          */
 CALL OR$VALUE(SP,2);
/*29 <RELATIVE> ::= RELATIVE <ID>       */
 CALL OR$VALUE(MP,8);
/*30     \! <EMPTY>           */
 ; /* NO ACTION REQUIRED - DEFAULT */
/*31 <IC> ::= I-O-CONTROL . <SAME-LIST>      */
 ;
/*32    \! <EMPTY>           */
 ;
/*33 <SAME-LIST> ::= <SAME-ELEMENT>       */
 ;
/*34      \! <SAME-LIST> <SAME-ELEMENT>    */
 ;
/*35 <SAME-ELEMENT> ::= SAME <ID-STRING> .     */
 ;
/*36 <ID-STRING> ::= <ID>          */
 ;
/*37      \! <ID-STRING> <ID>      */
 ;
```

```
/*38 <D-DIV> ::= DATA DIVISION . <FILE-SECTION> <WORK> */
/*38     <LINK>          */
 ; /* NO ACTION REQUIRED */
/*39 <FILE-SECTION> ::= FILE SECTION . <FILE-LIST>  */
 FILE$SEC$END = TRUE;
/*40      \! <EMPTY>        */
 FILE$SEC$END=TRUE;
/*41 <FILE-LIST> ::= <FILES>          */
 ; /* NO ACTION REQUIRED */
/*42      \! <FILE-LIST> <FILES>      */
 ; /* NO ACTION REQUIRED */
/*43 <FILES> ::= FD <ID> <FILE-CONTROL> .     */
/*43     <RECORD-DESCRIPTION>      */
 DO;
  DO WHILE STACK$LEVEL > 1;
   CALL REDUCE$STACK;
  END;
  CALL END$OF$RECORD;
  REDEF=FALSE;
 END;
/*44 <FILE-CONTROL> ::= <FILE-LIST>       */
 CALL SET$IO$ADDRS;
/*45      \! <EMPTY>        */
 CALL SET$IO$ADDRS;
/*46 <FILE-LIST> ::= <FILE-ELEMENT>       */
 ; /* NO ACTION REQUIRED */
/*47      \! <FILE-LIST> <FILE-ELEMENT>    */
 ; /* NO ACTION REQUIRED */
/*48 <FILE-ELEMENT> ::= BLOCK <INTEGER> RECORDS    */
 ; /* NO ACTION REQUIRED - FILES NEVER BLOCKED */
/*49      \! RECORD <REC-COUNT>     */
 CALL SET$SLENGTH(VALUE(SP));
/*50      \! LABEL RECORDS STANDARD    */
 ; /* NO ACTION REQUIRED */
/*51      \! LABEL RECORDS OMITTED    */
 ; /* NO ACTION REQUIRED */
/*52      \! VALUE OF <ID-STRING>     */
 ; /* NO ACTION REQUIRED */
/*53 <REC-COUNT> ::= <INTEGER>         */
 ; /* NO ACTION REQUIRED - VALUE(SP) CORRECT */
/*54      \! <INTEGER> TO <INTEGER>     */
 DO;
 VALUE(MP)=VALUE(SP); /* VARIABLE LENGTH */
 CALL SET$TYPE(4); /* SET TO VARIABLE */
 END;
/*55 <WORK> ::= WORKING-STORAGE SECTION .     */
/*55     <RECORD-DESCRIPTION>       */
 DO;
  DO WHILE STACK$LEVEL > 1;
  CUR$SYM = ID$STACK(ID$STACK$PTR - 1);
  IF GET$LEVEL = STACK$LEVEL THEN
   CALL REDEF$TEST;
  ELSE IF STACK$LEVEL > 1 THEN
    CALL REDUCE$STACK;
```

```
  END;
  IF STACK$LEVEL = 1 AND ID$STACK$PTR <> 0 THEN
  DO;
  CUR$SYM = ID$STACK(ID$STACK$PTR - 1);
  IF REDEF THEN CALL REDEF$TEST;
  END;
  CALL END$OF$RECORD;
 END;
/*56   \! <EMPTY>          */
 ; /* NO ACTION REQUIRED */
/*57 <LINK> ::= LINKAGE SECTION . <RECORD-DESCRIPTION> */
 CALL PRINT$ERROR('NI'); /* INTER PROG COMM */
/*58   \! <EMPTY>          */
 ; /* NO ACTION REQUIRED */
/*59 <RECORD-DESCRIPTION> ::= <LEVEL-ENTRY>    */
 ; /* NO ACTION REQUIRED */
/*60       \! <RECORD-DESCRIPTION>  */
/*60       <LEVEL-ENTRY>    */
 ; /* NO ACTION REQUIRED */
/*61 <LEVEL-ENTRY> ::= <INTEGER> <DATA-ID> <REDEFINES> */
/*61      <DATA-TYPE> .       */
 DO;
   CALL LOAD$LEVEL;
   IF (PENDING$LITERAL <> 0) AND (NOT VALUE$FLAG) THEN
   PENDING$LIT$ID = ID$STACK$PTR;
 END;
/*62 <DATA-ID> ::= <ID>          */
 ; /* NO ACTION REQUIRED */
/*63    \! FILLER         */
 DO;
   CUR$SYM, VALUE(SP)=NEXT$SYM;
   CALL BUILD$SYMBOL(0);
 END;
/*64 <REDEFINES> ::= REDEFINES <ID>       */
 DO;
   CALL SET$REDEF(VALUE(SP),VALUE(SP-2));
   VALUE(MP)=1; /* SET REDEFINE FLAG ON */
   IF NOT FILE$SEC$END THEN
   CALL PRINT$ERROR('R3');
   CALL CHECK$LVL$WORK;
 END;
/*65     \! <EMPTY>         */
 DO;
 IF NOT FILE$SEC$END THEN
  CALL CHECK$LVL$FILES;
 ELSE CALL CHECK$LVL$WORK;
 END;
/*66 <DATA-TYPE> ::= <PROP-LIST>        */
 ; /* NO ACTION REQUIRED */
/*67     \! <EMPTY>         */
 ; /* NO ACTION REQUIRED */
/*68 <PROP-LIST> ::= <DATA-ELEMENT>       */
 ; /* NO ACTION REQUIRED */
/*69     \! <PROP-LIST> <DATA-ELEMENT>    */
```

```
 ; /* NO ACTION REQUIRED */
/*70 <DATA-ELEMENT> ::= PIC <INPUT>       */
 CALL PIC$ANALIZER;
/*71      \! USAGE COMP       */
 CALL SET$TYPE(COMP);
/*72      \! USAGE DISPLAY      */
 ; /* NO ACTION REQUIRED - DEFAULT */
/*73      \! SIGN LEADING <SEPARATE>    */
 CALL SET$SIGN(17);
/*74      \! SIGN TRAILING <SEPARATE>    */
 CALL SET$SIGN(18);
/*75      \! OCCURS <INTEGER>      */
 DO;
   CALL OR$TYPE(128);
   CALL SET$OCCURS(VALUE(SP));
 END;
/*76      \! SYNC <DIRECTION>      */
 ; /* NO ACTION REQUIRED - BYTE MACHINE */
/*77      \! VALUE <LITERAL>      */
 DO;
   IF NOT FILE$SEC$END THEN
   DO;
    CALL PRINT$ERROR('VE');
    PENDING$LITERAL=0;
   END;
 END;
/*78 <DIRECTION> ::= LEFT         */
 ; /* NO ACTION REQUIRED */
/*79      \! RIGHT         */
 ; /* NO ACTION REQUIRED */
/*80      \! <EMPTY>      */
 ; /* NO ACTION REQUIRED */
/*81 <SEPARATE> ::= SEPARATE         */
 VALUE(SP)=2;
/*82     \! <EMPTY>         */
 ; /* NO ACTION REQUIRED */
/*83 <LITERAL> ::= <INPUT>         */
 DO;
 IF ((LIT$TYPE:=GET$TYPE) <> 16) AND
  (LIT$TYPE <> 17) AND (LIT$TYPE <> 21) THEN
  DO;
  CALL PRINT$ERROR('NV');
  CALL LOAD$LITERAL(0);
  PENDING$LITERAL = 2;
  END;
 ELSE DO;
   CALL LOAD$LITERAL(1);
   PENDING$LITERAL = 1;
   END;
 END;
/*84     \! <LIT>          */
 DO;
   CALL LOAD$LITERAL(0);
   PENDING$LITERAL=2;
```

```
 END;
/*85    \! ZERC          */
 PENDING$LITERAL=3;
/*86    \! SPACE         */
 PENDING$LITERAL=4;
/*87    \! QUOTE         */
 PENDING$LITERAL=5;
/*88 <INTEGER> ::= <INPUT>        */
 CALL CONVERT$INTEGER;
/*89 <ID> ::= <INPUT>          */
 VALUE(SP)=MATCH; /* STORE SYMBOL TABLE POINTERS */


 END; /* END OF CASE STATEMENT */
END CODE$GEN;

GETIN1: PROCEDURE BYTE;
 RETURN INDEX1(STATE);
END GETIN1;

GETIN2: PROCEDURE BYTE;
 RETURN INDEX2(STATE);
END GETIN2;

INCSP: PROCEDURE;
 SP=SP + 1;
 IF SP >= PSTACKSIZE THEN CALL FATAL$ERROR('SO');
 VALUE(SP)=0; /* CLEAR VALUE STACK */
END INCSP;


DUP$IDEN$FLAG: PROCEDURE;
  IF TOKEN = 02 THEN FILE$DESC$FLAG = TRUE;
  IF TOKEN = 47 THEN REDEF$FLAG = TRUE;
END DUP$IDEN$FLAG;

LOOKAHEAD: PROCEDURE;
 IF NCLOOK THEN
 DO;
   CALL SCANNER;
   CALL DUP$IDEN$FLAG;
   NOLOOK=FALSE;
   IF PRINT$TOKEN THEN
    DO;
    CALL CRLF;
    CALL PRINT$NUMBER(TOKEN);
    CALL PRINT$CHAR(' ');
    CALL PRINT$ACCUM;
   END;
 END;
END LOOKAHEAD;

NO$CONFLICT: PROCEDURE (CSTATE) BYTE;
 DECLARE (CSTATE,I,J,K) BYTE;
```

```
 J=INDEX1(CSTATE);
 K=J + INDEX2(CSTATE) - 1;
 DO I=J TO K;
   IF READ1(I)=TOKEN THEN RETURN TRUE;
 END;
RETURN FALSE;
END NO$CONFLICT;

RECOVER: PROCEDURE BYTE;
 DECLARE (TSP, RSTATE) BYTE;
 DO FOREVER;
   TSP=SP;
   DO WHILE TSP <> 255;
    IF NO$CONFLICT(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; /* TRY ANOTHER TOKEN */
 END;
END RECOVER;

END$PASS: PROCEDURE;
 /* THIS PROCEDURE STORES THE INFORMATION REQUIRED BY
 PART2 IN LOCATIONS ABOVE THE SYMBOL TABLE. THE FOLLOWING
 INFORMATION IS STORED:
   OUTPUT FILE CONTROL BLOCK
   COMPILER TOGGLES
   INPUT BUFFER POINTER
 THE OUTPUT BUFFER IS ALSO FILLED SO THE CURRENT RECORD.
 IS WRITTEN */

 CALL BYTE$OUT(SCD);
 CALL ADDR$OUT(NEXT$AVAILABLE);
 DO WHILE OUTPUT$PTR<>.OUTPUT$BUFF;
   CALL BYTE$OUT(0FFH);
 END;

 CALL MOVE(.OUTPUT$FCB,MAX$MEMORY-PASS1$LEN,PASS1$LEN);
 L: GO TO L; /* PATCH TO 'JMP 3100H' */
END END$PASS;

 /* * * * * PROGRAM EXECUTION STARTS HERE * * */

CALL MOVE(INITIAL$POS,MAX$MEMORY,RDR$LENGTH);
CALL INIT$SCANNER;
CALL INIT$SYMBOL;


 /* * * * * * * PARSER * * * * * * */

DO WHILE COMPILING;
```

```
IF STATE <= MAXRNO THEN  /* READ STATE */
DO;
  CALL INCSP;
  STATESTACK(SP) = STATE; /* SAVE CURRENT STATE */
  CALL LOOKAHEAD;
  I=GETIN1;
  J = I + GETIN2 - 1;
  DO I=I TO J;
   IF READ1(I) = TOKEN THEN
   DO;
   /* COPY THE ACCUMULATOR IF IT IS AN INPUT
   STRING. IF IT IS A RESERVED WORD IT DOES
   NOT NEED TO BE COPIED. */
    IF (TOKEN=INPUT$STR) OR (TOKEN=LITERAL) THEN
     DO K=Ø TO ACCUM(Ø);
       VARC(K)=ACCUM(K);
     END;
    STATE=READ2(I);
    NOLOOK=TRUE;
    I=J;
   END;
   ELSE
   IF I=J THEN
   DO;
    CALL PRINT$ERROR('NP');
    CALL PRINT(.(' ERROR NEAR $'));
    CALL PRINT$ACCUM;
    IF (STATE:=RECOVER)=Ø THEN COMPILING=FALSE;
   END;
  END;
END; /* END OF READ STATE */
ELSE
IF STATE>MAXPNO THEN  /* APPLY PRODUCTION STATE */
DO;
  MP=SP - GETIN2;
  MPP1=MP + 1;
  CALL CODE$GEN(STATE - MAXPNO);
  SP=MP;
  I=GETIN1;
  J=STATESTACK(SP);
  DO WHILE (K:=APPLY1(I)) <> Ø AND J<>K;
   I=I + 1;
  END;
  IF (K:=APPLY2(I))=Ø THEN COMPILING=FALSE;
  STATE=K;
END;
ELSE
IF STATE<=MAXLNO THEN /*LOOKAHEAD STATE*/
DO;
I=GETIN1;
  CALL LOOKAHEAD;
  DO WHILE (K:=LOOK1(I))<>Ø AND TOKEN <>K;
   I=I+1;
  END;
```

```
 STATE=LOOK2(I);
 END;
 ELSE
 DO;   /*PUSH STATES*/
   CALL INCSP;
   STATESTACK(SP)=GETIN2;
   STATE=GETIN1;
 END;
END; /* OF WHILE COMPILING */
CALL END$PASS;
END;
```

```
PART2: /* MODULE NAME */
   DO;

  /* COBOL COMPILER - PART 2 */

/* 100H = MODULE LOAD POINT */

    /* GLOBAL DECLARATIONS AND LITERALS */

DECLARE LIT LITERALLY 'LITERALLY'; DECLARE
  PASS1$LEN LIT '48',
  MAX$MEMORY LIT '0D100H',
  PASS1$TOP LIT '0D000H',
  CR LIT '13',
  LF LIT '10',
  QUOTE LIT '27H',
  POUND LIT '23H',
  TRUE LIT '1',
  FALSE LIT '0',
  FOREVER LIT 'WHILE TRUE',
  ALPHA$LIT$FLAG BYTE INITIAL(FALSE),
  IF$FLAG  BYTE  INITIAL(FALSE);  DECLARE  MAXRNO  LITERALLY
'82',/* MAX READ COUNT */
    MAXLNO LITERALLY '105',/* MAX LOOK COUNT */
    MAXPNO LITERALLY '120',/* MAX PUSH COUNT */
    MAXSNO LITERALLY '218',/* MAX STATE COUNT */
    STARTS LITERALLY '1';/* START STATE */
DECLARE READ1(*) BYTE
 DATA(0,63,5,6,9,14,16,20,22,24,26,31,32,41,42,44,45  ,49,53
,54,58,60,48,28,48,29,28,29,36,37,48,59,11,35,46
,34,13,28,29,36,37    ,48,3,1,40,23,48,57,1,56,2,30,43,27,19
,33,50,52,64,18,4,38,28,39,48     ,61,55,1,15,7,12,10,51,5,9
,14,16,20,22,24,26,31,41,42,44,45,49,53,54
,58,60,51,7,17,1,1
,5,9,14,16,20,21,22,24,26,31,41,42,44,45,49,53,54
,54,58,60,48,62,8,48,25,0,0);
DECLARE LOOK1(*) BYTE
 DATA(0,48,0,40,0,2,0,40,0,1,15,0,48,0,30,43,0,2,0,27,0,7
,0,17,0,1,15,0,55,0,55,0,55,0,55,0,1,15,0,12,0,1,0,51,0
,48,0);,0,25,0,8,48
DECLARE APPLY1(*) BYTE
 DATA(0,0,22,0,6,0,0,77,0,0,81,0,11,66,68,74,79,0,0,3,81,0
,3,81,0,25,0,0,0,0,57,58,59,0,0,0,0,0,0,0,69,0,0,0,0,0,0
,5,7,8,13,14       ,44,0,0,2,5,6,7,8,12,13,14,18,21,23,24,26
,27,28,29,33,34,40,44,75,76     ,77,80,0,9,30,37,38,49,52,54
,0,5,7,8,13,14,28,44,0,52,0,20,0,0,15,    ,32,63,65,0,0,0,3,
81,0,0);
DECLARE READ2(*) BYTE
 DATA(0,41,6,218,9,10,83,15,17,18,20,23,24,27,28,29,30,32
,33,34,37,38,31,201,85,84,201,205,207,206,85,178,194,192
,193,185             ,172,210,205,207,206,209,202,129,26,191
,197,86,3,35,4,189,188,21,167          ,168,166,161,162,14,5
,181,201,25,85,39,169,2,11,7,164,174,184,6,9,10          ,83
,15,17,18,20,23,27,28,29,30,32,33,34,37,38,184,8,13,130
```

```
,131,6,9,10                    ,83,15,16,17,18,20,23,27,28,29,30
,32,33,34,37,38,19,8,40,121,198,19,0,0);
DECLARE LOOK2(*) BYTE
 DATA(0,12,106,22,107,198,199,36,108,142,142,124,44,109,45
,45,110,46,196,47,111,112,49,113,52,114,114,54,56,115,57
,116,58        ,117,59,118,119,119,63,64,120,147,67,69,139,75
,122,78,136,128,128,81);
DECLARE APPLY2(*) BYTE
 DATA(0,0,137,60,76,103,77,127,126,105,73,72,151,150,152
,177,149,132,133,104,104,136,102,102,139,182,74,160,48
,65,155,153               ,156,154,148,68,134,61,94,146,66,173
,79,159,55,186,80,96,144,97,98          ,95,175,135,190,42,90
,87,90,90,215,90,90,217,179,138,88,124,89,90          ,157,91
,158,143,90,125,125,42,145,43,92,50,51,93,203,203,53,211
,195,195,195,195,195,195,195,200,71,70,208,212,171,62
,99,213,163,100 ,140,141,101,101,147,82);
DECLARE INDEX1(*) BYTE
 DATA(0,1,115,2,22,115,115,115,115,23,25,73,115,115,115,
,26,31,32,115,35,36,115,44,115,115,26,115,115,115,115
,23,42,26,115                         ,115,43,44,23,23,45,115,47
,48,50,115,51,50,53,54,23,59,60,23,61,62,65,        ,66,66,66
,66,67,68,69,26,70,26,73,71,73,91,92,93,94,95,96,115,115,117
,119,73,115,2,26,1,3,5,7,9,12,14,17,19,21,23,25,28,30,32
,34,36,39,                          ,41,43,45,47,49,216,123,123,176
,187,180,204,204,183,170,170,170,170              ,214,165,1,2
,2,4,4,6,6,7,7,9,9,10,10,10,12,12,12,12,12,12,12,12,12,12,12,
 ,12,12,12,12,18,18,18,18,19,19,19,19,22,22,22,25,27,27,27
,28,28,29,29              ,29,30,30,34,34,35,35,36,36,37,37,38
,38,39,39,39,40,42,43,43,44,44              ,45,45,46,46,46,47
,47,54,55,80,80,80,88,96,96,98,98,98,100,100,100
 ,101,101,106,106,107,107,108,111);
DECLARE INDEX2(*) BYTE
 DATA(0,1,1,20,1,1,1,1,1,2,1,18,1,1,1,5,1,3,1,1,6,1,1,1,
,5,1,1,1,1,2,1,5,1,1,1,1,2,2,2,1,1,2,1,1,2,1,1,5,2,1,1,2
,1,3,,1,1,1              ,1,1,1,1,1,5,1,5,18,2,18,1,1,1,1,1,19
,1,2,2,1,18,1,20,5,2,2,2,2,3,2,                    ,3,2,2,2,2,3,2
,2,2,2,3,2,2,2,2,2,3,12,22,36,44,45,47,49,52,54,56,57,
 ,58,59,63,64,5,1,0,0,1,0,1,2,2,1,2,0,0,2,1,0,2,1,0,2,1,1
,3,3,2,3,0,1,                          ,2,2,4,2,5,4,4,5,1,1,2,2,0
,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,0,1,1
 ,0,0,1,2,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,3,0,0,0,0,0,0,0,0
,0,0,0,0,0,0 ,1,0);

  /* END OF TABLES */
DECLARE
  /* JOINT DECLARATIONS */
  /* THE FOLLOWING ITEMS ARE DECLARED TOGETHER IN THIS
  GROUP IN ORDER TO FACILITATE THEIR BEING PASSED FROM
  THE FIRST PART OF THE COMPILER.
  */
  OUTPUT$FCB   (33) BYTE,
  DEBUGGING   BYTE,
  PRINT$PROD   BYTE,
  PRINT$TOKEN  BYTE,
```

```
  LIST$INPUT   BYTE,
  SEQ$NUM    BYTE,
  NEXT$SYM    ADDRESS,
  POINTER    ADDRESS, /* POINTS TO THE NEXT BYTE
                         TO BE READ */
  NEXT$AVAILABLE ADDRESS,
  MAX$INT$MEM  ADDRESS,
  HASH$TAB$ADDR  ADDRESS,   /* ADDRESS OF THE BOTTOM OF
                    THE TABLES FROM PART1 */

  /* I O BUFFERS AND GLOBALS */
  IN$ADDR ADDRESS INITIAL (5CH),
  INPUTFCB BASED INADDR (33) BYTE,
  OUTPUT$BUFF  (128)     BYTE,
  OUTPUT$PTR     ADDRESS,
  OUTPUT$END     ADDRESS,
  OUTPUT$CHAR  BASED OUTPUT$PTR BYTE;

  /* MESSAGES FOR OUTPUT */

DECLARE
  ERROR$NEAR$$ (*) BYTE DATA (' ERROR NEAR $'),
  END$OF$PART$2(*) BYTE DATA (' END OF COMPILATION $');

  /* GLOBAL COUNTERS */
DECLARE
  CTR BYTE,
  A$CTR ADDRESS,
  BASE ADDRESS,
  B$BYTE BASED BASE BYTE,
  B$ADDR BASED BASE ADDRESS;

MON1: PROCEDURE (F,A) EXTERNAL;
  DECLARE F BYTE, A ADDRESS;
END MON1;

MON2: PROCEDURE (F,A) BYTE EXTERNAL;
  DECLARE F BYTE, A ADDRESS;
END MON2;

BOOT: PROCEDURE EXTERNAL;
   END BOOT;

PRINTCHAR: PROCEDURE (CHAR);
  DECLARE CHAR BYTE;
  CALL MON1 (2,CHAR);
END PRINTCHAR;

CRLF: PROCEDURE;
  CALL PRINTCHAR(CR);
  CALL PRINTCHAR(LF);
END CRLF;

PRINT: PROCEDURE (A);
```

```
  DECLARE A ADDRESS;
  CALL MON1 (9,A);
END PRINT;

PRINT$ERROR: PROCEDURE (CODE);
  /* THIS PROCIDURE IS USED TO PRINT COMPILER ERRORS TO
    THE CONSOL */
  DECLARE CODE   ADDRESS,
      I      BYTE,
      CODE1(6) ADDRESS;
  IF CODE = FALSE THEN
  DO;
    DO I = Ø TO 5;
      CODE1(I) = Ø;
    END;
    I = Ø;
  END;
  ELSE
  IF CODE = TRUE THEN
  DO;
    I = Ø;
    DO WHILE((I<>6) AND (CODE1(I) <> Ø));
      CALL CRLF;
      CALL PRINTCHAR(HIGH(CODE1(I)));
      CALL PRINTCHAR (LOW(CODE1(I)));
      CODE1(I) = Ø;
      I = I + 1;
    END;
    I = Ø;
  END;
  ELSE
  IF (CODE = 'NP') OR (CODE = 'NV') OR (CODE = 'SL') THEN
  DO;
    CALL CRLF;
    CALL PRINTCHAR(HIGH(CODE));
    CALL PRINTCHAR( LOW(CODE));
  END;
  ELSE
  DO;
     IF I <> 6 THEN
     DO;
       CODE1(I) = CODE;
       I = I + 1;
     END;
  END;
END PRINT$ERROR;

FATAL$ERROR: PROCEDURE(REASON);
  DECLARE REASON ADDRESS;
  CALL PRINT$ERROR(REASON);
  CALL PRINT$ERROR(TRUE);
  CALL TIME(10);
  CALL BOOT;
END FATAL$ERROR;
```

```
CLOSE: PROCEDURE;
  IF MON2(16,.OUTPUT$FCB)=255 THEN CALL FATAL$ERROR('CL');
END CLOSE;

MORE$INPUT: PROCEDURE BYTE;
  /* READS THE INPUT FILE AND RETURNS TRUE IF A RECORD
     WAS READ. FALSE IMPLIES END OF FILE */
  DECLARE DCNT BYTE;
  IF (DCNT:=MON2(20,.INPUT$FCB))>1 THEN
      CALL FATAL$ERROR('BR');
  RETURN NOT(DCNT);
END MORE$INPUT;

WRITE$OUTPUT: PROCEDURE (LOCATION);
  /* WRITES OUT A 128 BYTE BUFFER FROM LOCATION*/
  DECLARE LOCATION ADDRESS;
  CALL MON1(26,LOCATION); /* SET DMA */
  IF MON2(21,.OUTPUT$FCB)<>0 THEN CALL FATAL$ERROR('WR');
  CALL MON1(26,80H); /*RESET DMA */
END WRITE$OUTPUT;

MOVE: PROCEDURE(SOURCE, DESTINATION, COUNT);
  /* MOVES FOR THE NUMBER OF BYTES SPECIFIED BY COUNT */
  DECLARE (SOURCE,DESTINATION) ADDRESS,
  (S$BYTE BASED SOURCE, D$BYTE BASED DESTINATION, COUNT)
   BYTE;
  DO WHILE (COUNT:=COUNT - 1) <> 255;
     D$BYTE=S$BYTE;
     SOURCE=SOURCE +1;
     DESTINATION = DESTINATION + 1;
  END;
END MOVE;

FILL: PROCEDURE(ADDR,CHAR,COUNT);
  /* MOVES CHAR INTO ADDR FOR COUNT BYTES */
  DECLARE ADDR ADDRESS,
  (CHAR,COUNT,DEST BASED ADDR) BYTE;
  DO WHILE (COUNT:=COUNT -1)<>255;
     DEST=CHAR;
     ADDR=ADDR + 1;
  END;
END FILL;

  /*  * * * * * SCANNER LITS * * * * */
DECLARE
  LITERAL    LIT    '29',
  INPUT$STR   LIT    '48',
  PERIOD     LIT    '1',
  RPARIN     LIT    '3',
  LPARIN     LIT    '2',
  INVALID    LIT    '0';

  /* * * * * SCANNER TABLES * * * *  */
```

```
DECLARE TOKEN$TABLE (*) BYTE DATA
  /* CONTAINS THE TOKEN NUMBER ONE LESS THAN THE FIRST
  RESERVED WORD FOR EACH LENGTH OF WORD */
  (0,0,3,7,13,29,41,48,56,60,63),

TABLE (*) BYTE DATA('BY','GO','IF','TO','EOF','ADD','END'
,'I-O','NOT','RUN','CALL','ELSE','EXIT','FROM','INTO'
,'LESS','MOVE','NEXT','OPEN','PAGE','READ','SIZE','STOP'
,'THRU','ZERO','AFTER','CLOSE','ENTER','EQUAL','ERROR'
,'INPUT','QUOTE','SPACE','TIMES','UNTIL','USING','WRITE'
,'ACCEPT','BEFORE','DELETE','DIVIDE','OUTPUT','DISPLAY'
,'GREATER','INVALID','NUMERIC','PERFORM','REWRITE'
,'ROUNDED','SECTION','DIVISION','MULTIPLY','SENTENCE'
,'SUBTRACT','ADVANCING','DEPENDING','PROCEDURE'
,'ALPHABETIC'),
OFFSET (11) ADDRESS INITIAL
/* NUMBER OF BYTES TO INDEX INTO THE TABLE FOR
   EACH LENGTH */
(0,0,0,8,26,86,146,176,232,264,291).

WORD$COUNT (*) BYTE DATA
  /* NUMBER OF WORDS OF EACH SIZE */
  (0,0,4,6,15,12,5,8,4,3,1),

  MAX$ID$LEN   LIT    '12',
  MAX$LEN    LIT    '10',
  ADD$END    (*) BYTE DATA ('EOF '),
  LOOKED     BYTE INITIAL (0),
  HOLD       BYTE,
  EOFFILLER   LIT    '1AH',
  BUFFER$END   ADDRESS  INITIAL  (100H),
  NEXT       BASED POINTER    BYTE,
  INBUFF     LIT    '80H',
  CHAR       BYTE   INITIAL(' '),
  ACCUM (82)   BYTE,
  DISPLAY (82)  BYTE  INITIAL (0),
  TOKEN      BYTE;      /*RETURNED FROM SCANNER */

  /* PROCEDURES USED BY THE SCANNER */

NEXT$CHAR: PROCEDURE BYTE;
  IF LOOKED THEN
  DO;
     LOOKED=FALSE;
     RETURN (CHAR:=HOLD);
  END;
  IF (POINTER:=POINTER + 1) >= BUFFER$END THEN
  DO;
    IF NOT MORE$INPUT THEN
     DO;
       BUFFER$END=.MEMORY;
       POINTER=.ADD$END;
     END;
     ELSE POINTER=INBUFF;
```

```
  END;
  IF NEXT = EOFFILLER THEN
  DO;
    BUFFER$END = .MEMORY;
    POINTER = .ADD$END;
  END;
  RETURN (CHAR:=NEXT);
END NEXT$CHAR;

GET$CHAR: PROCEDURE;
  /* THIS PROCEDURE IS CALLED WHEN A NEW CHAR IS NEEDED
  WITHOUT THE DIRECT RETURN OF THE CHARACTER*/
  CHAR=NEXT$CHAR;
END GET$CHAR;

DISPLAY$LINE: PROCEDURE;
  IF NOT LIST$INPUT THEN RETURN;
  DISPLAY(DISPLAY(0) + 1) = '$';
  CALL PRINT(.DISPLAY(1));
  DISPLAY(0)=0;
END DISPLAY$LINE;

LOAD$DISPLAY: PROCEDURE;
  IF DISPLAY(0)<80 THEN
     DISPLAY(DISPLAY(0):=DISPLAY(0)+1)=CHAR;
  CALL GET$CHAR;
END LOAD$DISPLAY;

PUT: PROCEDURE;
  IF ACCUM(0) < 80 THEN
  ACCUM(ACCUM(0):=ACCUM(0)+1)=CHAR;
  CALL LOAD$DISPLAY;
END PUT;

EAT$LINE: PROCEDURE;
  DO WHILE CHAR<>CR;
     CALL LOAD$DISPLAY;
  END;
END EAT$LINE;

GET$NO$BLANK: PROCEDURE;
  DECLARE (N,I) BYTE;
  DO FOREVER;
     IF CHAR = ' ' THEN CALL LOAD$DISPLAY;
     ELSE
     IF CHAR=CR THEN
     DO;
       CALL DISPLAY$LINE;
       CALL PRINT$ERROR(TRUE);
       IF SEC$NUM THEN N=8; ELSE N=2;
       DO I = 1 TO N;
          CALL LOAD$DISPLAY;
       END;
       IF CHAR = '*' THEN CALL EAT$LINE;
```

```
      END;
        ELSE
        IF CHAR = ':' THEN
        DO;
           IF NOT DEBUGGING THEN CALL EAT$LINE;
           ELSE
           CALL LOAD$DISPLAY;
        END;
      ELSE
      RETURN;
  END;  /* END OF DO FOREVER */
END GET$NO$BLANK;

SPACE: PROCEDURE BYTE;
  RETURN (CHAR=' ') OR (CHAR=CR);
END SPACE;

LEFT$PARIN: PROCEDURE BYTE;
  RETURN CHAR = '(';
END LEFT$PARIN;

RIGHT$PARIN: PROCEDURE BYTE;
  RETURN CHAR = ')';
END RIGHT$PARIN;

DELIMITER: PROCEDURE BYTE;
  /* CHECKS FOR A PERIOD FOLLOWED BY A SPACE OR CR*/
  IF CHAR <> '.' THEN RETURN FALSE;
  HOLD=NEXT$CHAR;
  LOOKED=TRUE;
  IF SPACE THEN
  DO;
     CHAR = '.';
     RETURN TRUE;
  END;
  CHAR='.';
  RETURN FALSE;
END DELIMITER;

END$OF$TOKEN: PROCEDURE BYTE;
  RETURN SPACE OR DELIMITER OR LEFT$PARIN OR RIGHT$PARIN;
END END$OF$TOKEN;

GET$LITERAL: PROCEDURE BYTE;
  CALL LOAD$DISPLAY;
  DO FOREVER;
    IF CHAR = QUOTE THEN
     DO;
      CALL LOAD$DISPLAY:
      RETURN LITERAL;
     END;
    CALL PUT;
  END;
END GET$LITERAL;
```

```
LOOK$UP: PROCEDURE BYTE;
  DECLARE POINT ADDRESS,
  HERE BASED POINT (1) BYTE, I BYTE;

  MATCH: PROCEDURE BYTE;
     DECLARE J BYTE;
     DO J=1 TO ACCUM(0);
       IF HERE(J - 1) <> ACCUM(J) THEN RETURN FALSE;
     END;
     RETURN TRUE;
  END MATCH;

  POINT=OFFSET(ACCUM(0))+ .TABLE;
  DO I=1 TO WORD$COUNT(ACCUM(0));
     IF MATCH THEN RETURN I;
     POINT = POINT + ACCUM(0);
  END;
  RETURN FALSE;
END LOOK$UP;

RESERVED$WORD: PROCEDURE BYTE;
  /* RETURNS THE TOKEN NUMBER OF A RESERVED WORD IF THE
  CONTENTS OF THE ACCUMULATOR IS A RESERVED WORD,
  OTHERWISE RETURNS ZERO */
  DECLARE VALUE BYTE;
  DECLARE NUMB BYTE;
  IF ACCUM(0) <= MAX$LEN THEN
  DO;
     IF (NUMB:=TOKEN$TABLE(ACCUM(0)))<>0 THEN
     DO;
       IF (VALUE:=LOOK$UP) <> 0 THEN
          NUMB=NUMB + VALUE;
       ELSE NUMB=0;
     END;
  END;
  ELSE NUMB=0;
  RETURN NUMB;
END RESERVED$WORD;

GET$TOKEN: PROCEDURE BYTE;
  ACCUM(0)=0;
  CALL GET$NO$BLANK;
  IF CHAR=QUOTE THEN RETURN GET$LITERAL;
  IF DELIMITER THEN
  DO;
     CALL PUT;
     RETURN PEPIOD;
  END;
  IF LEFT$PARIN THEN
  DO;
     CALL PUT;
     RETURN LPARIN;
  END;
```

```
  IF RIGHT$PARIN THEN
  DO;
     CALL PUT;
     RETURN RPARIN;
  END;
  DO FOREVER;
     CALL PUT;
     IF END$OF$TOKEN THEN RETURN INPUT$STR;
  END; /* OF DO FOREVER */
END GET$TOKEN;
  /*  END OF SCANNER ROUTINES */
  /*  SCANNER EXEC */
SCANNER: PROCEDURE;
  IF(TOKEN:=GET$TOKEN) = INPUT$STR THEN
     IF (CTR:=RESERVED$WORD) <> 0 THEN TOKEN=CTR;
END SCANNER;
PRINT$ACCUM: PROCEDURE;
  ACCUM(ACCUM(0)+1)='$';
  CALL PRINT(.ACCUM(1));
END PRINT$ACCUM;
PRINT$NUMBER: PROCEDURE(NUMB);
  DECLARE(NUMB,I,CNT,K) BYTE, J (*) BYTE DATA(100,10);
  DO I=0 TO 1;
     CNT=0;
     DO WHILE NUMB >= (K:=J(I));
       NUMB=NUMB - K;
       CNT=CNT + 1;
     END;
     CALL PRINTCHAR('0' + CNT);
  END;
  CALL PRINTCHAR('0' + NUMB);
END PRINT$NUMBER;
  /* * * * END OF SCANNER PROCEDURES * * * */
  /* * * * * SYMBOL TABLE DECLARATIONS * * * */
DECLARE
CUR$SYM       ADDRESS,    /*SYMBOL BEING ACCESSED*/
SYMBOL        BASED CUR$SYM (1) BYTE,
SYMBOL$ADDR     BASED CUR$SYM (1) ADDRESS,
NEXT$SYM$ENTRY   BASED NEXT$SYM   ADDRESS,
HASH$MASK      LIT     '3FH',
S$TYPE        LIT       '2',
DISPLACEMENT     LIT     '13',
OCCURS        LIT     '12',
P$LENGTH       LIT       '3',
FLD$LENGTH      LIT       '3',
LEVEL        LIT       '10',
DECIMAL       LIT       '11',
REL$ID        LIT       '5',
LOCATION       LIT       '2',
START$NAME      LIT       '12', /*1 LESS*/
FCB$ADDR       LIT     '4',
   /* * * * * * * SYMBOL TYPE LITERALS * * * * * * * */
UNRESOLVED      LIT     '255',
LABEL$TYPE      LIT     '32',
```

```
MULT$OCCURS    LIT    '128',
GROUP        LIT    '6',
NON$NUMERIC$LIT  LIT    '7',
ALPHA        LIT    '8',
ALPHA$NUM      LIT    '9',
LIT$SPACE      LIT    '10',
LIT$QUOTE      LIT    '11',
LIT$ZERO       LIT    '12',
NUMERIC$LITERAL  LIT    '15',
NUMERIC      LIT    '16',
COMP         LIT    '21',
A$ED         LIT    '72',
A$N$ED         LIT    '73',
NUM$ED         LIT    '80';
  /* * * * SYMBOL TABLE ROUTINES * * * */
SET$ADDRESS: PROCEDURE(ADDR);
DECLARE ADDR ADDRESS;
  SYMBOL$ADDR(LOCATION)=ADDR;
END SET$ADDRESS;
GET$ADDRESS: PROCEDURE ADDRESS;
  RETURN SYMBOL$ADDR(LOCATION);
END GET$ADDRESS;
GET$FCB$ADDR: PROCEDURE ADDRESS;
  RETURN SYMBOL$ADDR(FCB$ADDR);
END GET$FCB$ADDR;
GET$TYPE: PROCEDURE BYTE;
  RETURN SYMBOL(S$TYPE);
END GET$TYPE;
SET$TYPE: PROCEDURE(TYPE);
  DECLARE TYPE BYTE;
  SYMBOL(S$TYPE)=TYPE;
END SET$TYPE;
GET$LENGTH: PROCEDURE ADDRESS;
  RETURN SYMBOL$ADDR(FLD$LENGTH);
END GET$LENGTH;
GET$LEVEL: PROCEDURE BYTE;
  RETURN SYMBOL(LEVEL);
END GET$LEVEL;
GET$DECIMAL: PROCEDURE BYTE;
  RETURN SYMBOL(DECIMAL);
END GET$DECIMAL;
GET$P$LENGTH: PROCEDURE BYTE;
RETURN SYMBOL(P$LENGTH);
END GET$P$LENGTH;
BUILD$SYMBOL: PROCEDURE(LEN);
  DECLARE LEN BYTE, TEMP ADDRESS;
  TEMP=NEXT$SYM;
  IF (NEXT$SYM:=.SYMBOL(LEN:=LEN + DISPLACEMENT))
     > MAX$MEMORY THEN CALL FATAL$ERROR('ST');
  CALL FILL (TEMP,0,LEN);
END BUILD$SYMBOL;
AND$OUT$OCCURS: PROCEDURE (TYPE$IN) BYTE;
  DECLARE TYPE$IN BYTE;
  RETURN TYPE$IN AND 127;
```

```
END AND$OUT$OCCURS;
  /* * * * PARSER DECLARATIONS * * * * */
DECLARE
PSTACKSIZE  LIT     '30',    /* SIZE OF PARSE STACKS*/
VALUE     (PSTACKSIZE)  ADDRESS, /* TEMP VALUES */
STATESTACK  (PSTACKSIZE)  BYTE, /* SAVED STATES */
VALUE2    (PSTACKSIZE)  ADDRESS,   /* VALUE2 STACK*/
VARC        (100)     BYTE,   /*TEMP CHAR STORE*/
ID$STACK   (20)   ADDRESS,
ID$PTR    BYTE,
MAX$BYTE   BASED     MAX$INT$MEM     BYTE,
SUB$IND    BYTE   INITIAL  (0),
COND$TYPE   BYTE,
HOLD$SECTION ADDRESS,
HOLD$SEC$ADDR ADDRESS,
SECTION$FLAG BYTE   INITIAL (0),
L$ADDR    ADDRESS,
DISPLAY$FLAG BYTE   INITIAL (FALSE),
L$LENGTH   ADDRESS,
L$TYPE    BYTE,
L$DEC     BYTE,
CON$LENGTH  BYTE,
COMPILING   BYTE  INITIAL(TRUE),
SP       BYTE  INITIAL (255),
MP       BYTE,
MPP1      BYTE,
NOLOOK    BYTE  INITIAL(FALSE),
(I,J,K)    BYTE,    /*INDICIES FOR THE PARSER*/
STATE     BYTE  INITIAL(STARTS),
  /* * * * * * * * CODE LITERALS * * * * * * * * * * */
  /* THE CODE LITERALS ARE BROKEN INTO GROUPS DEPENDING
  ON THE TOTAL LENGTH OF CODE PRODUCED FOR THAT ACTION */
  /* LENGTH ONE */
ADD LIT '1',  /* ADD REGISTER 1 TO REGISTER 0 */
SUB LIT '2',  /* SUBTRACT REGISTER 1 FROM REGISTER 0 */
MUL LIT '3',  /* MULTIPLY REGISTER 0 BY REGISTER 1 */
DIV LIT '4',  /* DIVIDE REGISTER 0 BY REGISTER 1
                 (NO REMAINDER) */
NEG LIT '5',  /* NOT OPERATOR */
STP LIT '6',  /* STOP PROGRAM */
STI LIT '7',  /* STORE REGISTER 2 INTO REGISTER 0 */
   /* LENGTH TWO */
RND LIT '8',  /* ROUND CONTENTS OF REGISTER 2 */
   /* LENGTH THREE */
RET LIT '9',  /* RETURN */
CLS LIT '10', /* CLOSE */
SER LIT '11', /* BRANCH ON SIZE ERROR */
BRN LIT '12', /* BRANCH */
OPN LIT '13', /* OPEN A FILE FOR INPUT */
OP1 LIT '14', /* OPEN A FILE FOR OUTPUT */
OP2 LIT '15', /* OPEN A FILE FOR BOTH INPUT AND OUTPUT */
RGT LIT '16', /* REGISTER GREATER THAN */
RLT LIT '17', /* REGISTER LESS THAN */
REQ LIT '18', /* REGISTER EQUAL */
```

```
INV LIT '19', /*BRANCH IF INVALID-FILE-ACTION FLAG TRUE*/
EOR LIT '20', /* BRANCH ON END-OF-RECORDS FLAG */
   /* LENGTH FOUR */
ACC LIT '21', /* ACCEPT */
STD LIT '22', /* STOP WITH DISPLAY */
LDI LIT '23', /* LOAD A CODE ADDRESS DIRECT */
   /* LENGTH FIVE */
DIS LIT '24', /* DISPLAY */
DEC LIT '25', /* DECREMENT COUNT AND BRANCH IF ZERO */
STO LIT '26', /* STORE NUMERIC */
ST1 LIT '27', /* STORE SIGNED NUMERIC LEADING */
ST2 LIT '28', /* STORE SIGNED NUMERIC TRAILING */
ST3 LIT '29', /* STORE SEPARATE SIGN LEADING */
ST4 LIT '30', /* STORE SEPARATE SIGN TRAILING */
ST5 LIT '31', /* STORE A PACKED NUMERIC FIELD */
   /* LENGTH SIX */
LOD LIT '32', /* LOAD NUMERIC LITERAL */
LD1 LIT '33', /* LOAD NUMERIC */
LD2 LIT '34', /* LOAD SIGNED NUMERIC LEADING */
LD3 LIT '35', /* LOAD SIGNED NUMERIC TRAILING */
LD4 LIT '36', /* LOAD SEPARATE SIGN LEADING */
LD5 LIT '37', /* LOAD SEPARATE SIGN TRAILING */
LD6 LIT '38', /* LOAD A PACKED NUMERIC FIELD */
   /* LENGTH SEVEN */
PER LIT '39', /* PERFORM */
CNU LIT '40', /* COMPARE NUMERIC UNSIGNED */
CNS LIT '41', /* COMPARE NUMERIC SIGNED */
CAL LIT '42', /* COMPARE ALPHABETIC */
RWS LIT '43', /* REWRITE SEQUENTIAL */
DLS LIT '44', /* DELETE SEQUENTIAL */
RDF LIT '45', /* READ A SEQUENTIAL FILE */
WTF LIT '46', /* WRITE A RECORD TO A SEQUENTIAL FILE */
RVL LIT '47', /* READ A VARIABLE LENGTH FILE */
WVL LIT '48', /* WRITE A VARIABLE LENGTH RECORD */
   /* LENGTH NINE */
SCR LIT '49', /* CALCULATE A SUBSCRIPT */
SGT LIT '50', /* STRING GREATER THAN */
SLT LIT '51', /* STRING LESS THAN */
SEQ LIT '52', /* STRING EQUAL */
MOV LIT '53', /* MOVE */
   /* LENGTH TEN */
RRS LIT '54', /* READ RELATIVE SEQUENTIAL */
WRS LIT '55', /* WRITE RELATIVE SEQUENTIAL */
RRR LIT '56', /* READ RELATIVE RANDOM */
WRR LIT '57', /* WRITE RELATIVE RANDOM */
RWR LIT '58', /* REWRITE RELATIVE */
DLR LIT '59', /* DELETE RELATIVE */
   /* LENGTH ELEVEN */
MED LIT '60', /*MOVE INTO AN ALPHANUMERIC EDITED FIELD*/
MNE LIT '61', /* MOVE INTO A NUMERIC EDITED FIELD */
   /* VARIABLE LENGTH */
GDP LIT '62', /* GO TO - DEPENDING ON */
   /* BUILD DIRECTING ONLY */
INT LIT '63', /* INITIALIZE MEMORY */
```

```
BST LIT '64', /* BACK STUFF */
TER LIT '65', /* TERMINATE BUILD */
SCD LIT '66'; /* START CODE */
  /* * * * PARSER ROUTINES * * * * */
DIGIT: PROCEDURE (CHAR) BYTE;
  DECLARE CHAR BYTE;
  RETURN (CHAR<='9') AND (CHAR>='0');
END DIGIT;
LETTER: PROCEDURE (CHAR) BYTE;
  DECLARE CHAR BYTE;
  RETURN (CHAR>='A') AND (CHAR<='Z');
END LETTER;
INVALID$TYPE: PROCEDURE;
  CALL PRINT$ERROR('IT');
END INVALID$TYPE;
BYTE$OUT: PROCEDURE(ONE$BYTE);
  DECLARE ONE$BYTE BYTE;
  IF (OUTPUT$PTR:=OUTPUT$PTR + 1) > OUTPUT$END THEN
  DO;
     CALL WRITE$OUTPUT(.OUTPUT$BUFF);
     OUTPUT$PTR=.OUTPUT$BUFF;
  END;
  OUTPUT$CHAR=ONE$BYTE;
END BYTE$OUT;
ADDR$OUT: PROCEDURE (ADDR);
  DECLARE ADDR ADDRESS;
  CALL BYTE$OUT(LOW(ADDR));
  CALL BYTE$OUT(HIGH (ADDR));
END ADDR$OUT;
INC$COUNT: PROCEDURE(CNT);
  DECLARE CNT BYTE;
  IF(NEXT$AVAILABLE:=NEXT$AVAILABLE + CNT)
     >MAX$INT$MEM THEN CALL FATAL$ERROR('MO');
END INC$COUNT;
ONE$ADDR$OPP: PROCEDURE(CODE,ADDR);
  DECLARE CODE BYTE, ADDR ADDRESS;
  CALL BYTE$OUT(CODE);
  CALL ADDR$OUT(ADDR);
  CALL INC$COUNT(3);
END ONE$ADDR$OPP;
NOT$IMPLIMENTED: PROCEDURE;
  CALL PRINT$ERROR ('NI');
END NOT$IMPLIMENTED;
MATCH: PROCEDURE ADDRESS;
  /* CHECKS AN IDENTIFIER TO SEE IF IT IS IN THE SYMBOL
  TABLE. IF IT IS PRESENT, CUR$SYM IS SET FOR ACCESS,
  OTHERWISE THE POINTERS ARE SET FOR ENTRY*/
  DECLARE POINT ADDRESS, COLLISION BASED POINT ADDRESS,
          (HOLD,I) BYTE;
  IF VARC(0)>MAX$ID$LEN THEN VARC(0)=MAX$ID$LEN;
  HOLD=0;
  DO I=1 TO VARC(0);
     HOLD=HOLD+VARC(I);
  END;
```

```
  POINT=HASH$TAB$ADDR + SHL((HOLD AND HASH$MASK),1);
  DO FOREVER;
    IF COLLISION=0 THEN
     DO;
       CUR$SYM,COLLISION=NEXT$SYM;
       CALL BUILD$SYMBOL(VARC(0));
       SYMBOL(P$LENGTH)=VARC(0);
       DO I=1 TO VARC(0);
          SYMBOL(START$NAME+I)=VARC(I);
       END;
       CALL SET$TYPE(UNRESOLVED); /* UNRESOLVED LABEL */
       RETURN CUR$SYM;
     END;
     ELSE
     DO;
       CUR$SYM=COLLISION;
       IF (HOLD:=GET$P$LENGTH)=VARC(0) THEN
       DO;
         I=1;
         DO WHILE SYMBOL(START$NAME + I)= VARC(I);
         IF (I:=I+1)>HOLD THEN RETURN(CUR$SYM:=COLLISION);
         END;
       END;
    END;
    POINT=COLLISION;
  END;
END MATCH;
SET$VALUE: PROCEDURE(NUMB);
  DECLARE NUMB ADDRESS;
  VALUE(MP)=NUMB;
END SET$VALUE;
SET$VALUE2: PROCEDURE(ADDR);
  DECLARE ADDR ADDRESS;
  VALUE2(MP)=ADDR;
END SET$VALUE2;
SUB$CNT: PROCEDURE BYTE;
  IF (SUB$IND:=SUB$IND + 1)>8 THEN
     SUB$IND=1;
  RETURN SUB$IND;
END SUB$CNT;
CODE$BYTE: PROCEDURE (CODE);
  DECLARE CODE BYTE;
  CALL BYTE$OUT(CODE);
  CALL INC$COUNT(1);
END CODE$BYTE;
CODE$ADDRESS: PROCEDURE (CODE);
  DECLARE CODE ADDRESS;
  CALL ADDR$OUT(CODE);
  CALL INC$COUNT(2);
END CODE$ADDRESS;
INPUT$NUMERIC: PROCEDURE BYTE;
  DO CTR=1 TO VARC(0);
     IF NOT DIGIT(VARC(CTR)) THEN RETURN FALSE;
  END;
```

```
  RETURN TRUE;
END INPUT$NUMERIC;
CONVERT$INTEGER: PROCEDURE ADDRESS;
  ACTR=0;
  DO CTR=1 TO VARC(0);
     IF NOT DIGIT(VARC(CTR)) THEN CALL PRINT$ERROR('NN');
     A$CTR=SHL(ACTR,3)+SHL(ACTR,1) + VARC(CTR) - '0';
  END;
  RETURN ACTR;
END CONVERT$INTEGER;
BACKSTUFF: PROCEDURE (ADD1,ADD2);
  DECLARE (ADD1,ADD2) ADDRESS;
  CALL BYTE$OUT(BST);
  CALL ADDR$OUT(ADD1);
  CALL ADDR$OUT(ADD2);
END BACK$STUFF;
UNRESOLVED$BRANCH: PROCEDURE;
  CALL SET$VALUE(NEXT$AVAILABLE + 1);
  CALL ONE$ADDR$OPP(BRN,0);
  CALL SET$VALUE2(NEXT$AVAILABLE);
END UNRESOLVED$BRANCH;
BACK$COND: PROCEDURE;
  CALL BACKSTUFF(VALUE(SP-1),NEXT$AVAILABLE);
END BACK$COND;
SET$BRANCH: PROCEDURE;
  CALL SET$VALUE(NEXT$AVAILABLE);
  CALL CODE$ADDRESS(0);
END SET$BRANCH;
KEEP$VALUES: PROCEDURE;
  CALL SET$VALUE(VALUE(SP));
  CALL SET$VALUE2(VALUE2(SP));
END KEEP$VALUES;
GET$REC$ADDRESS: PROCEDURE(RECORD$ADDR) ADDRESS;
  DECLARE (RECORD$ADDR, HOLD$ADDR) ADDRESS;
  CUR$SYM=RECORD$ADDR;
  HOLD$ADDR=GET$ADDRESS;
  CUR$SYM=GET$FCB$ADDR;
  RETURN HOLD$ADDR;
END GET$REC$ADDRESS;
GET$REC$LEN: PROCEDURE(RECORD$ADDR) ADDRESS;
  DECLARE (RECORD$ADDR, HOLD$LENGTH) ADDRESS;
  CUR$SYM=RECORD$ADDR;
  HOLD$LENGTH=GET$LENGTH;
  CUR$SYM=GET$FCB$ADDR;
  RETURN HOLD$LENGTH;
END GET$REC$LEN;
STD$ATTRIBUTES: PROCEDURE(TYPE);
   DECLARE TYPE BYTE;
   CALL CODE$ADDRESS(GET$FCB$ADDR);
   CALL CODE$ADDRESS(GET$REC$ADDRESS(GET$ADDRESS));
   CALL CODE$ADDRESS(GET$REC$LEN(GET$ADDRESS));
   IF TYPE=0 THEN RETURN;
   CUR$SYM=SYMBOL$ADDR(REL$ID);
   CALL CODE$ADDRESS(GET$ADDRESS);
```

```
   CALL CODE$BYTE(GET$LENGTH);
END STD$ATTRIBUTES;
WRITE$A$RECORD: PROCEDURE;
  IF GET$LEVEL<>1 THEN CALL PRINT$ERROR('WL');
  ELSE DO;
    CUR$SYM=GET$FCB$ADDR;
    IF (CTR:=GET$TYPE)=1 THEN
    DO;
      CALL CODE$BYTE(WTF);
      CALL STD$ATTRIBUTES(0);
    END;
    ELSE IF CTR=2 THEN
    DO;
      CALL CODE$BYTE(WRS);
      CALL STD$ATTRIBUTES(1);
    END;
    ELSE IF CTR=3 THEN
    DO;
      CALL CODE$BYTE(WRR);
      CALL STD$ATTRIBUTES(1);
    END;
    ELSE IF CTR=4 THEN
    DO;
      CALL CODE$BYTE(WVL);
      CALL STD$ATTRIBUTES(0);
    END;
    ELSE CALL PRINT$ERROR('FT');
  END;
END WRITE$A$RECORD;
READ$A$FILE: PROCEDURE;
  IF (CTR:=GET$TYPE)=1 THEN
  DO;
    CALL CODE$BYTE(RDF);
    CALL STD$ATTRIBUTES(0);
  END;
  ELSE IF CTR=2 THEN
  DO;
    CALL CODE$BYTE(RRS);
    CALL STD$ATTRIBUTES(1);
  END;
  ELSE IF CTR=3 THEN
  DO;
    CALL CODE$BYTE(RRR);
    CALL STD$ATTRIBUTES(1);
  END;
  ELSE IF CTR=4 THEN
  DO;
    CALL CODE$BYTE(RVL);
    CALL STD$ATTRIBUTES(0);
  END;
  ELSE CALL PRINT$ERROR('FT');
END READ$A$FILE;
ARITHMETIC$TYPE: PROCEDURE BYTE;
  IF ((L$TYPE:=AND$OUT$OCCURS(L$TYPE))>=NUMERIC$LITERAL)
```

```
  AND (L$TYPE<=COMP) THEN RETURN L$TYPE - NUMERIC$LITERAL;
  CALL INVALID$TYPE;
  RETURN 0;
END ARITHMETIC$TYPE;
DELETE$A$FILE: PROCEDURE;
  IF (CTR:=GET$TYPE)=3 THEN
  DO;
    CALL CODE$BYTE(DLR);
    CALL STD$ATTRIBUTES(1);
  END;
  ELSE IF CTR=2 THEN
  DO;
    CALL CODE$BYTE(DLS);
    CALL STD$ATTRIBUTES(0);
  END;
  ELSE CALL PRINT$ERROR('IT');
END DELETE$A$FILE;
REWRITE$A$RECORD: PROCEDURE;
  IF GET$LEVEL<>1 THEN CALL PRINT$ERROR('WL');
  ELSE DO;
    CUR$SYM=GET$FCB$ADDR;
    IF (CTR:=GET$TYPE)=3 THEN
    DO;
      CALL CODE$BYTE(RWR);
      CALL STD$ATTRIBUTES(1);
    END;
    ELSE IF CTR=2 THEN
    DO;
      CALL CODE$BYTE(RWS);
      CALL STD$ATTRIBUTES(0);
    END;
    ELSE CALL PRINT$ERROR('IT');
  END;
END REWRITE$A$RECORD;
ATTRIBUTES: PROCEDURE;
  CALL CODE$ADDRESS(L$ADDR);
  CALL CODE$BYTE(L$LENGTH);
  CALL CODE$BYTE(L$DEC);
END ATTRIBUTES;
LOAD$L$ID: PROCEDURE(S$PTR);
  DECLARE S$PTR BYTE;
  IF ((A$CTR := VALUE(S$PTR)) <= NON$NUMERIC$LIT) OR
     (ACTR = NUMERIC$LITERAL) THEN
  DO;
     L$ADDR=VALUE2(SPTR);
     L$LENGTH=CON$LENGTH;
     L$TYPE=A$CTR;
     RETURN;
  END;
  IF A$CTR<=LIT$ZERO THEN
  DO;
     L$TYPE,L$ADDR=A$CTR;
     L$LENGTH=1;
     RETURN;
```

```
  END;
  CUR$SYM=VALUE(S$PTR);
  L$TYPE=GET$TYPE;
  L$LENGTH=GET$LENGTH;
  L$DEC=GET$DECIMAL;
  IF(L$ADDR:=VALUE2(S$PTR))=0 THEN L$ADDR=GET$ADDRESS;
END LOAD$L$ID;
LOAD$REG: PROCEDURE(REG$NO,PTR);
  DECLARE (REG$NO,PTR) BYTE;
  CALL LOAD$L$ID(PTR);
  CALL CODE$BYTE(LOD+ARITHMETIC$TYPE);
  CALL ATTRIBUTES;
  CALL CODE$BYTE(REG$NO);
END LOAD$REG;
STORE$REG: PROCEDURE(PTR);
  DECLARE PTR BYTE;
  CALL LOAD$L$ID(PTR);
  CALL CODE$BYTE(STO + ARITHMETIC$TYPE -1);
  CALL ATTRIBUTES;
END STORE$REG;
STORE$CONSTANT: PROCEDURE ADDRESS;
  IF(MAX$INT$MEM:=MAX$INT$MEM - VARC(0))<NEXT$AVAILABLE
     THEN CALL FATAL$ERROR('MO');
  CALL BYTE$OUT(INT);
  CALL ADDR$OUT(MAX$INT$MEM);
  CALL ADDR$OUT(CON$LENGTH:=VARC(0));
  DO CTR = 1 TO CON$LENGTH;
     CALL BYTE$OUT(VARC(CTR));
  END;
  RETURN MAX$INT$MEM;
END STORE$CONSTANT;
NUMERIC$LIT: PROCEDURE BYTE;
  DECLARE CHAR BYTE;
  DO CTR=1 TO VARC(0);
     IF NOT( DIGIT(CHAR:=VARC(CTR))
       OR (CHAR='-') OR (CHAR='+')
       OR (CHAR='.')) THEN RETURN FALSE;
  END;
  RETURN TRUE;
END NUMERIC$LIT;
ALPHA$LIT: PROCEDURE BYTE;
  DO CTR=1 TO VARC(0);
     IF NOT(LETTER(VARC(CTR))) THEN RETURN FALSE;
  END;
  RETURN TRUE;
END ALPHA$LIT;
ROUND$STORE: PROCEDURE;
  IF VALUE(SP)<>0 THEN
  DO;
     CALL CODE$BYTE(RND);
     CALL CODE$BYTE(L$DEC);
  END;
  CALL STORE$REG(SP-1);
END ROUND$STORE;
```

```
ADD$SUB: PROCEDURE (INDEX);
   DECLARE INDEX BYTE;
  CALL LOAD$REG(0,MPP1);
  IF VALUE(SP-3)<>0 THEN
  DO;
     CALL LOAD$REG(1,SP-3);
     CALL CODE$BYTE(ADD);
     CALL CODE$BYTE(STI);
  END;
  CALL LOAD$REG(1,SP-1);
  CALL CODE$BYTE(ADD + INDEX);
  CALL ROUND$STORE;
END ADD$SUB;
MULT$DIV: PROCEDURE(INDEX);
  DECLARE INDEX BYTE;
  CALL LOAD$REG(0,MPP1);
  CALL LOAD$REG(1,SP-1);
  CALL CODE$BYTE(MUL + INDEX);
  CALL ROUND$STORE;
END MULT$DIV;
CHECK$SUBSCRIPT: PROCEDURE;
  CUR$SYM=VALUE(MP);
  IF GET$TYPE<MULT$OCCURS THEN
  DO;
     CALL PRINT$ERROR('IS');
     RETURN;
  END;
  IF INPUT$NUMERIC THEN
  DO;
     CALL SET$VALUE2(GET$ADDRESS + (GET$LENGTH *
          CONVERT$INTEGER));
     RETURN;
  END;
  CUR$SYM=MATCH;
  IF ((CTR:=GET$TYPE)<NUMERIC) OR (CTR>COMP) THEN
     CALL PRINT$ERROR('TE');
  CALL ONE$ADDR$OPP(SCR,GET$ADDRESS);
  CALL CODE$BYTE(SUB$CNT);
  CALL CODE$BYTE(GET$LENGTH);
  CALL SET$VALUE2(SUB$IND);
END CHECK$SUBSCRIPT;
LOAD$LABEL: PROCEDURE;
  CUR$SYM=VALUE(MP);
  IF (A$CTR:=GET$ADDRESS)<>0 THEN
     CALL BACK$STUFF(A$CTR,VALUE2(MP));
  CALL SET$ADDRESS(VALUE2(MP));
  CALL SET$TYPE(LABEL$TYPE);
  IF (A$CTR:=GET$FCB$ADDR)<>0 THEN
       CALL BACK$STUFF(A$CTR,NEXT$AVAILABLE);
  SYMBOL$ADDR(FCB$ADDR)=NEXT$AVAILABLE;
  CALL ONE$ADDR$OPP(RET,0);
END LOAD$LABEL;
LOAD$SEC$LABEL: PROCEDURE;
  A$CTR=VALUE(MP);
```

```
  CALL SET$VALUE(HOLD$SECTION);
  HOLD$SECTION=A$CTR;
  A$CTR=VALUE2(MP);
  CALL SET$VALUE2(HOLD$SEC$ADDR);
  HOLD$SEC$ADDR = A$CTR;
  CALL LOAD$LABEL;
END LOAD$SEC$LABEL;
LABEL$ADDR$OFFSET: PROCEDURE (ADDR, HOLD, OFFSET) ADDRESS;
  DECLARE ADDR ADDRESS;
  DECLARE (HOLD, OFFSET, CTR) BYTE;
  CUR$SYM=ADDR;
  IF(CTR:=GET$TYPE)=LABEL$TYPE THEN
  DO;
     IF HOLD THEN RETURN GET$ADDRESS;
     RETURN GET$FCB$ADDR;
  END;
  IF CTR<>UNRESOLVED THEN CALL INVALID$TYPE;
  IF HOLD THEN
  DO;
     A$CTR=GET$ADDRESS;
     CALL SET$ADDRESS(NEXT$AVAILABLE + OFFSET);
     RETURN A$CTR;
  END;
  A$CTR=GET$FCB$ADDR;
  SYMBOL$ADDR(FCB$ADDR)=NEXT$AVAILABLE + OFFSET;
  RETURN A$CTR;
END LABEL$ADDR$OFFSET;
LABEL$ADDR: PROCEDURE (ADDR, HOLD) ADDRESS;
  DECLARE ADDR ADDRESS,
      HOLD BYTE;
  RETURN LABEL$ADDR$OFFSET (ADDR, HOLD, 1);
END LABEL$ADDR;
CODE$FOR$DISPLAY: PROCEDURE (POINT);
  DECLARE POINT BYTE;
  CALL LOAD$L$ID(POINT);
  CALL ONE$ADDR$OPP(DIS,L$ADDR);
  CALL CODE$BYTE(L$LENGTH);
  IF DISPLAY$FLAG THEN CALL CODE$BYTE(1);
  ELSE CALL CODE$BYTE(0);
DISPLAY$FLAG=FALSE;
END CODE$FOR$DISPLAY;
A$AN$TYPE: PROCEDURE BYTE;
 RETURN (L$TYPE=ALPHA) OR (L$TYPE=ALPHA$NUM);
END A$AN$TYPE;
NOT$INTEGER: PROCEDURE BYTE;
  RETURN L$DEC<>0;
END NOT$INTEGER;
NUMERIC$TYPE: PROCEDURE BYTE;
  RETURN (L$TYPE>=NUMERIC$LITERAL) AND (L$TYPE<=COMP);
END NUMERIC$TYPE;
GEN$COMPARE: PROCEDURE;
  DECLARE (H$TYPE,H$DEC) BYTE,
     (H$ADDR,H$LENGTH) ADDRESS;
  CALL LOAD$L$ID(MP);
```

```
  L$TYPE=AND$OUT$OCCURS(L$TYPE);
  IF COND$TYPE=3 THEN /* COMPARE FOR NUMERIC */
  DO;
     IF A$AN$TYPE OR (L$TYPE>COMP) THEN CALL INVALID$TYPE;
     CALL SET$VALUE2(NEXT$AVAILABLE);
     IF L$TYPE=NUMERIC THEN CALL CODE$BYTE(CNU);
     ELSE CALL CODE$BYTE(CNS);
     CALL CODE$ADDRESS(L$ADDR);
     CALL CODE$ADDRESS(L$LENGTH);
     CALL SET$BRANCH;
   END;
  ELSE IF COND$TYPE=4 THEN
  DO;
     IF NUMERIC$TYPE THEN CALL INVALID$TYPE;
     CALL SET$VALUE2(NEXT$AVAILABLE);
     CALL CODE$BYTE(CAL);
     CALL CODE$ADDRESS(L$ADDR);
     CALL CODE$ADDRESS(L$LENGTH);
     CALL SET$BRANCH;
  END;
  ELSE DO;
     IF NUMERIC$TYPE THEN CTR=1;
     ELSE CTR=0;
     H$TYPE=L$TYPE;
     H$DEC=L$DEC;
     H$ADDR=L$ADDR;
     H$LENGTH=L$LENGTH;
     CALL LOAD$L$ID(SP);
     IF NUMERIC$TYPE THEN CTR=CTR+1;
     IF CTR=2 THEN  /* NUMERIC COMPARE */
     DO;
       CALL LOAD$REG(0,MP);
       CALL SET$VALUE2(NEXT$AVAILABLE-6);
       CALL LOAD$REG(1,SP);
       CALL CODE$BYTE(SUB);
       CALL CODE$BYTE(RGT + COND$TYPE);
       CALL SET$BRANCH;
     END;
     ELSE DO;
       /* ALPHA NUMERIC COMPARE */
       IF (H$DEC<>0) OR (H$TYPE=COMP)
          OR (I$DEC<>0) OR (L$TYPE=COMP)
          OR (H$LENGTH<>L$LENGTH) THEN CALL INVALID$TYPE;
       CALL SET$VALUE2(NEXT$AVAILABLE);
       CALL CODE$BYTE(SGT+COND$TYPE);
       CALL CODE$ADDRESS(H$ADDR);
       CALL CODE$ADDRESS(L$ADDR);
       CALL CODE$ADDRESS(H$LENGTH);
       CALL SET$BRANCH;
     END;
  END;
END GEN$COMPARE;
MOVE$TYPE: PROCEDURE BYTE;
  DECLARE
```

```
  HOLD$TYPE BYTE,
  ALPHA$NUM$MOVE   LIT '0',
  A$N$ED$MOVE      LIT '1',
  NUMERIC$MOVE     LIT '2',
  N$ED$MOVE        LIT '3';
  L$TYPE=AND$OUT$OCCURS(L$TYPE);
  IF((HOLD$TYPE:=AND$OUT$OCCURS(GET$TYPE))=GROUP) OR
    (L$TYPE=GROUP)
     THEN RETURN ALPHA$NUM$MOVE;
  IF HOLD$TYPE=ALPHA THEN
   IF A$AN$TYPE OR (L$TYPE=A$ED) OR (L$TYPE=A$N$ED)
     OR ((ALPHA$LIT$FLAG) AND (L$TYPE = NON$NUMERIC$LIT))
     THEN RETURN ALPHA$NUM$MOVE;
  IF HOLD$TYPE=ALPHA$NUM THEN
  DO;
     IF NOT$INTEGER THEN CALL INVALID$TYPE;
     RETURN ALPHA$NUM$MOVE;
  END;
  IF (HOLD$TYPE>=NUMERIC) AND (HOLD$TYPE<=COMP) THEN
  DO;
     IF (L$TYPE=ALPHA) OR (L$TYPE>COMP) THEN
        CALL INVALID$TYPE;
     RETURN NUMERIC$MOVE;
  END;
  IF HOLD$TYPE=A$N$ED THEN
  DO;
     IF NOT$INTEGER THEN CALL INVALID$TYPE;
     RETURN A$N$ED$MOVE;
  END;
  IF HOLD$TYPE=A$ED THEN
     IF A$AN$TYPE OR (L$TYPE>COMP) OR (L$TYPE
        = NON$NUMERIC$LIT)
     THEN RETURN A$N$ED$MOVE;
  IF HOLD$TYPE=NUM$ED THEN
     IF NUMERIC$TYPE OR (L$TYPE=ALPHA$NUM) THEN
       RETURN N$ED$MOVE;
  CALL INVALID$TYPE;
  RETURN 0;
END MOVE$TYPE;
GEN$MOVE:PROCEDURE;
  DECLARE
  LENGTH1 ADDRESS,
  ADDR1 ADDRESS,
  EXTRA ADDRESS;
  ADD$ADD$LEN: PROCEDURE;
     CALL CODE$ADDRESS(ADDR1);
     CALL CODE$ADDRESS(L$ADDR);
     CALL CODE$ADDRESS(L$LENGTH);
  END ADD$ADD$LEN;
  CODE$FOR$EDIT: PROCEDURE;
     CALL ADD$ADD$LEN;
     CALL CODE$ADDRESS(GET$FCB$ADDR);
     CALL CODE$ADDRESS(LENGTH1);
  END CODE$FOR$EDIT;
```

```
  CALL LOAD$L$ID(MPP1);
  CUR$SYM=VALUE(SP);
  IF (ADDR1:=VALUE2(SP))=0 THEN ADDR1=GET$ADDRESS;
  LENGTH1=GET$LENGTH;
  DO CASE MOVE$TYPE;

     /* ALPHA NUMERIC MOVE */

     DO;
       IF LENGTH1>L$LENGTH THEN EXTRA=LENGTH1-L$LENGTH;
       ELSE DO;
          EXTRA=0;
          L$LENGTH=LENGTH1;
       END;
       CALL CODE$BYTE(MOV);
       CALL ADD$ADD$LEN;
       CALL CODE$ADDRESS(EXTRA);
     END;
     /* ALPHA NUMERIC EDITED */
     DO;
       CALL CODE$BYTE(MED);
       CALL CODE$FOR$EDIT;
     END;
     /* NUMERIC MOVE */
     DO;
       CALL LOAD$REG(2,MPP1);
       CALL STORE$REG(SP);
     END;
     /* NUMERIC EDITED MOVE */

     DO;
       CALL CODE$BYTE(MNE);
       CALL CODE$FOR$EDIT;
       CALL CODE$BYTE(L$DEC);
       CALL CODE$BYTE(GET$DECIMAL);
     END;
  END;
END GEN$MOVE;
CODE$GEN: PROCEDURE(PRODUCTION);
  DECLARE PRODUCTION BYTE;
  IF PRINT$PROD THEN
  DO;
     CALL CRLF;
     CALL PRINTCHAR(POUND);
     CALL PRINT$NUMBER(PRODUCTION);
  END;
  DO CASE PRODUCTION;
/* P R O D U C T I O N S */
 /* CASE 0 NOT USED */
      ;
/*  1 <P-DIV> ::= PROCEDURE DIVISION <USING> .
                  <PROC-BODY> */
 DO;
    COMPILING = FALSE;
```

```
    IF SECTION$FLAG THEN CALL LOAD$SEC$LABEL;
 END;
/*   2  <USING> ::= USING <ID-STRING>                    */
 CALL NOT$IMPLIMENTED;  /* INTER PROG COMM */
/*   3       \! <EMPTY>                                  */
 ;  /* NO ACTION REQUIRED */
/*   4  <ID-STRING> ::= <ID>                             */
 ID$STACK(ID$PTR:=0)=VALUE(SP);
/*   5         \! <ID-STRING> <ID>                       */
 DO;
    IF(ID$PTR:=IDPTR+1)=20 THEN
    DO;
      CALL PRINT$ERROR('ID');
      ID$PTR=19;
    END;
    ID$STACK(ID$PTR)=VALUE(SP);
 END;
/*   6  <PROC-BODY> ::= <PARAGRAPH>                      */
 ;  /* NO ACTION REQUIRED */
/*   7         \! <PROC-BODY> <PARAGRAPH>                */
 ;  /* NO ACTION REQUIRED */
/*   8  <PARAGRAPH> ::= <ID> . <SENTENCE-LIST>           */
 DO;
    IF SECTION$FLAG=0 THEN SECTION$FLAG=2;
    CALL LOAD$LABEL;
 END;
/*   9         \! <ID> SECTION .                         */
 DO;
    IF SECTION$FLAG<>1 THEN
    DO;
      IF SECTION$FLAG=2 THEN CALL PRINT$ERROR('PF');
      SECTION$FLAG=1;
      HOLD$SECTION=VALUE(MP);
      HOLD$SEC$ADDR=VALUE2(MP);
    END;
    ELSE CALL LOAD$SEC$LABEL;
 END;
/*  10  <SENTENCE-LIST> ::= <SENTENCE> .                 */
 ;  /* NO ACTION REQUIRED */
/*  11          \! <SENTENCE-LIST> <SENTENCE> .          */
 ;  /* NO ACTION REQUIRED */
/*  12  <SENTENCE> ::= <IMPERATIVE>                      */
 ;  /* NO ACTION REQUIRED */
/*  13        \! <CONDITIONAL>                           */
 ;  /* NO ACTION REQUIRED */
/*  14        \! ENTER <ID> <OPT-ID>                     */
 CALL NOT$IMPLIMENTED;  /* LANGUAGE CHANGE */
/*  15  <IMPERATIVE> ::= ACCEPT <SUBID>                  */
 DO;
   CALL LOAD$L$ID(SP);
   CALL ONE$ADDR$OPP(ACC,L$ADDR);
   CALL CODE$BYTE(L$LENGTH);
 END;
/*  16         \! <ARITHMETIC>                           */
```

```
 ;  /* NO ACTION REQUIRED */
/*   17          \! CALL <LIT> <USING>              */
 CALL NOT$IMPLIMENTED;  /* INTER PROG COMM */
/*   18          \! CLOSE <ID>                     */
 DO;
   DECLARE TYPE BYTE;
   TYPE=GET$TYPE;
   IF (TYPE>0) AND (TYPE<5) THEN
     CALL ONE$ADDR$OPP(CLS,GET$FCB$ADDR);
   ELSE CALL PRINT$ERROR('CE');
 END;
/*   19          \! <FILE-ACT>                     */
 ;  /* NO ACTION REQUIRED */
/*   20          \! DISPLAY <LIT/ID> <OPT-LIT/ID>      */
 DO;
    CALL CODE$FOR$DISPLAY(MPP1);
    IF VALUE(SP)<>0 THEN
    DO;
      DISPLAY$FLAG=TRUE;
      CALL CODE$FOR$DISPLAY(SP);
    END;
 END;
/*   21          \! EXIT <PROGRAM-ID>              */
 ;  /* NO ACTION REQUIRED */
/*   22          \! GO <ID>                    */
 CALL ONE$ADDR$OPP(BRN,LABEL$ADDR(VALUE(SP),1));
/*   23          \! GO <ID-STRING> DEPENDING <ID>      */
DO;
CALL CODE$BYTE(GDP);
CALL CODE$BYTE(ID$PTR);
CUR$SYM=VALUE(SP);
CALL CODE$BYTE(GET$LENGTH);
CALL CODE$ADDRESS(GET$ADDRESS);
DO CTR=0 TO ID$PTR;
CALL
 CODE$ADDRESS(LABEL$ADDR$OFFSET(ID$STACK(ID$PTR),1,0));
END;
END;
/*   24          \! MOVE <LIT/ID> TO <SUBID>       */
 CALL GEN$MOVE;
/*   25          \! OPEN <TYPE-ACTION> <ID>        */
 DO;
   DECLARE TYPE BYTE;
    TYPE=GET$TYPE;
    IF (TYPE=1 OR TYPE=4) AND (VALUE(MPP1)<>2)
     THEN CALL ONE$ADDR$OPP(OPN+VALUE(MPP1),GET$FCB$ADDR);
    ELSE
      IF (TYPE=2 OR TYPE=3) THEN
        CALL ONE$ADDR$OPP(OPN+VALUE(MPP1),GET$FCB$ADDR);
      ELSE CALL PRINT$ERROR('OE');
 END;
/*   26          \! PERFORM <ID> <THRU> <FINISH>   */
 DO;
    DECLARE (ADDR2,ADDR3) ADDRESS;
```

```
    IF VALUE(SP-1)=0
    THEN ADDR2=LABEL$ADDR$OFFSET(VALUE(MPP1),0,3);
    ELSE ADDR2=LABEL$ADDR$OFFSET(VALUE(SP-1),0,3);
    IF (ADDR3:=VALUE2(SP))=0 THEN ADDR3=NEXT$AVAILABLE
       + 7;
    ELSE CALL BACKSTUFF(VALUE(SP),NEXT$AVAILABLE + 7);
    CALL ONE$ADDR$OPP(PEK,LABEL$ADDR(VALUE(MPP1),1));
    CALL CODE$ADDRESS(ADDR2);
    CALL CODE$ADDRESS(ADDR3);
 END;
/*   27          \! <READ-ID>                    */
 CALL NOT$IMPLIMENTED;  /* GRAMMAR ERROR */
/*   28          \! STOP <TERMINATE>             */
 DO;
    IF VALUE(SP)=0 THEN CALL CODE$BYTE(STP);
        ELSE DO;
                CALLONE$ADDR$OPP(STD,VALUE2(SP));
                CALLCODE$BYTE(CON$LENGTH);
        END;
 END;
/*   29  <CONDITIONAL> ::= <ARITHMETIC> <SIZE-ERROR>   */
/*   29           <IMPERATIVE>                 */
 CALL BACK$COND;
/*   30          \! <FILE-ACT> <INVALID> <IMPERATIVE> */
 CALL BACK$COND;
/*   31          \! <IF-NONTERMINAL> <CONDITION>
                   <ACTION> ELSE */
/*   31           <IMPERATIVE>                 */
 DO;
    CALL BACKSTUFF(VALUE(MPP1),VALUE2(SP-2));
    CALL BACKSTUFF(VALUE(SP-2),NEXT$AVAILABLE);
 END;
/*   32          \! <READ-ID> <SPECIAL> <IMPERATIVE>  */
 CALL BACK$COND;
/*   33  <ARITHMETIC> ::= ADD <L/ID> <OPT-L/ID> TO
                          <SUBID> */
/*   33           <ROUND>                      */
 CALL ADD$SUB(0);
/*   34          \! DIVIDE <L/ID> INTO <SUBID> <ROUND>   */
 CALL MULT$DIV(1);
/*   35          \! MULTIPLY <L/ID> BY <SUBID> <ROUND>   */
 CALL MULT$DIV(0);
/*   36          \! SUBTRACT <L/ID> <OPT-L/ID> FROM      */
/*   36           <SUBID> <ROUND>                */
 CALL ADD$SUB(1);
/*   37  <FILE-ACT> ::= DELETE <ID>                   */
 CALL DELETE$A$FILE;
/*   38         \! REWRITE <ID>                    */
 CALL REWRITE$A$RECORD;
/*   39         \! WRITE <ID> <SPECIAL-ACT>         */
 CALL WRITE$A$RECORD;
/*   40  <CONDITION> ::= <LIT/ID> <NOT> <COND-TYPE>    */
DO;
 IF IF$FLAG THEN
```

```
  DO;
   IF$FLAG=NOT IF$FLAG;              /* RESET IF$FLAG */
   CALL CODE$BYTE(NEG);
  END;
 CALL GEN$COMPARE;
END;
/*   41  <COND-TYPE> ::= NUMERIC                    */
 COND$TYPE=3;
/*   42          \! ALPHABETIC                     */
 COND$TYPE=4;
/*   43          \! <COMPARE> <LIT/ID>             */
 CALL KEEP$VALUES;
/*   44  <NOT> ::= NOT                             */
 IF NOT IF$FLAG THEN
 CALL CODE$BYTE(NEG);
 ELSE IF$FLAG=NOT IF$FLAG;  /* RESET IF$FLAG */
/*   45       \! <EMPTY>                           */
 ;  /* NO ACTION REQUIRED */
/*   46  <COMPARE> ::= GREATER                     */
 COND$TYPE=0;
/*   47         \! LESS                            */
 COND$TYPE=1;
/*   48         \! EQUAL                           */
 COND$TYPE=2;
/*   49  <ROUND> ::= ROUNDED                       */
 CALL SET$VALUE(1);
/*   50        \! <EMPTY>                          */
 ;  /* NO ACTION REQUIRED */
/*   51  <TERMINATE> ::= <LITERAL>                 */
 ;  /* NO ACTION REQUIRED */
/*   52          \! RUN                            */
 ;  /* NO ACTION REQUIRED - VALUE(SP) ALREADY ZERO */
/*   53  <SPECIAL> ::= <INVALID>                   */
 ;  /* NO ACTION REQUIRED */
/*   54         \! END                             */
 DO;
    CALL SET$VALUE(2);
    CALL CODE$BYTE(EOR);
    CALL SET$BRANCH;
 END;
/*   55  <OPT-ID> ::= <SUBID>                      */
 ;  /* VALUE AND VALUE2 ALREADY SET */
/*   56        \!                                  */
 ;  /* VALUE ALREADY ZERO */
/*   57  <ACTION> ::= <IMPERATIVE>                 */
 CALL UNRESOLVED$BRANCH;
/*   58        \! NEXT SENTENCE                    */
 CALL UNRESOLVED$BRANCH;
/*   59  <THRU> ::= THRU <ID>                      */
 CALL KEEP$VALUES;
/*   60      \!                                    */
 ;  /* NO ACTION REQUIRED */
/*   61  <FINISH> ::= <L/ID> TIMES                 */
 DO;
```

```
    CALL LOAD$L$ID(MP);
    CALL ONE$ADDR$OPP(LDI,L$ADDR);
    CALL CODE$BYTE(L$LENGTH);
    CALL SET$VALUE2(NEXT$AVAILABLE);
    CALL ONE$ADDR$OPP(DEC,0);
    CALL SET$VALUE(NEXT$AVAILABLE);
        CALLCODE$ADDRESS(0);  END;
/*   62        \! UNTIL <CONDITION>                */
 CALL KEEP$VALUES;
/*   63        \!                          */
 ;  /* NO ACTION REQUIRED */
/*   64  <INVALID> ::= INVALID                     */
 DO;
    CALL SET$VALUE(1);
    CALL CODE$BYTE(INV);
    CALL SET$BRANCH;
 END;
/*   65  <SIZE-ERROR> ::= SIZE ERROR                */
 DO;
    CALL CODE$BYTE(SER);
    CALL UNRESOLVED$BRANCH;
 END;
/*  66 <SPECIAL-ACT> ::= <WHEN> ADVANCING <HOW-MANY> */
 CALL NOT$IMPLIMENTED;  /* CARRAGE CONTROL */
/*   67          \!                          */
 ;  /* NO ACTION REQUIRED */
/*   68  <WHEN> ::= BEFORE                         */
 CALL NOT$IMPLIMENTED;  /* CARRAGE CONTROL */
/*   69       \! AFTER                           */
 CALL NOT$IMPLIMENTED;  /* CARRAGE CONTROL */
/*   70  <HOW-MANY> ::= <INTEGER>                    */
 CALL NOT$IMPLIMENTED;  /* CARRAGE CONTROL */
/*   71        \! PAGE                          */
 CALL NOT$IMPLIMENTED;  /* CARRAGE CONTROL */
/*   72  <TYPE-ACTION> ::= INPUT                    */
 ;  /* NO ACTION REQUIRED - VALUE(SP) ALREADY ZERO */
/*   73          \! OUTPUT                        */
 CALL SET$VALUE(1);
/*   74          \! I-O                         */
 CALL SET$VALUE(2);
/*   75  <SUBID> ::= <SUBSCRIPT>                    */
 ;  /* VALUE AND VALUE2 ALREADY SET */
/*   76        \! <ID>                          */
 ;  /* NO ACTION REQUIRED */
/*   77  <INTEGER> ::= <INPUT>                      */
 CALL SET$VALUE(CONVERT$INTEGER);
/*   78  <ID> ::= <INPUT>                         */
 DO;
    CALL SET$VALUE(MATCH);
    IF GET$TYPE=UNRESOLVED THEN
       CALL SET$VALUE2(NEXT$AVAILABLE);
 END;
/*   79  <L/ID> ::= <INPUT>                       */
 DO;
```

```
    IF NUMERIC$LIT THEN
    DO;
      CALL SET$VALUE(NUMERIC$LITERAL);
      CALL SET$VALUE2(STORE$CONSTANT);
    END;
    ELSE CALL SET$VALUE(MATCH);
 END;
/*   80       \! <SUBSCRIPT>                    */
 ;  /* NO ACTION REQUIRED */
/*   81       \! ZERO                          */
 CALL SET$VALUE(LIT$ZERO);
/*   82  <SUBSCRIPT> ::= <ID> ( <INPUT> )      */
 CALL CHECK$SUBSCRIPT;
/*   83  <OPT-L/ID> ::= <L/ID>                 */
 ;  /* NO ACTION REQUIRED */
/*   84         \! <EMPTY>                     */
 ;  /* VALUE ALREADY SET */
/*   85  <NN-LIT> ::= <LIT>                    */
 DO;
    ALPHA$LIT$FLAG = ALPHA$LIT;
    CALL SET$VALUE(NON$NUMERIC$LIT);
    CALL SET$VALUE2(STORE$CONSTANT);
 END;
/*   86       \! SPACE                         */
 CALL SET$VALUE(LIT$SPACE);
/*   87       \! QUOTE                         */
 CALL SET$VALUE(LIT$QUOTE);
/*   88  <LITERAL> ::= <NN-LIT>                */
 ;  /* NO ACTION REQUIRED */
/*   89         \! <INPUT>                     */
 DO;
    IF NOT NUMERIC$LIT THEN CALL INVALID$TYPE;
    CALL SET$VALUE(NUMERIC$LITERAL);
    CALL SET$VALUE2(STORE$CONSTANT);
 END;
/*   90         \! ZERO                        */
 CALL SET$VALUE(LIT$ZERO);
/*   91  <LIT/ID> ::= <L/ID>                   */
 ;  /* NO ACTION REQUIRED */
/*   92       \! <NN-LIT>                      */
 ;  /* NO ACTION REQUIRED */
/*   93  <OPT-LIT/ID> ::= <LIT/ID>             */
 ;  /* NO ACTION REQUIRED */
/*   94         \! <EMPTY>                     */
 ;  /* NO ACTION REQUIRED */
/*   95  <PROGRAM-ID> ::= <ID>                 */
 CALL NOT$IMPLIMENTED;  /* INTER PROG COMM */
/*   96         \!                             */
 ;  /* NO ACTION REQUIRED */
/*   97  <READ-ID> ::= READ <ID>               */
 CALL READ$A$FILE;
/*   98  <IF-NONTERMINAL> ::= IF               */
 IF$FLAG = TRUE;                  /* SET IF$FLAG */
 END;  /* END OF CASE STATEMENT */
```

```
ENDCODE$GEN;
GETIN1:PROCEDURE BYTE;
 RETURN INDEX1(STATE);
ENDGETIN1;
GETIN2:PROCEDURE BYTE;
 RETURN INDEX2(STATE);
ENDGETIN2;
INCSP:PROCEDURE;
 VALUE(SP:=SP + 1)=0;  /* CLEAR THE STACK WHILE
                          INCREMENTING */
 VALUE2(SP)=0;
 IF SP >= PSTACKSIZE THEN CALL FATAL$ERROR('SO');
ENDINCSP;
LOOKAHEAD:PROCEDURE;
 IF NOLOOK THEN
 DO;
    CALL SCANNER;
    NOLOOK=FALSE;
    IF PRINT$TOKEN THEN
    DO;
      CALL CRLF;
      CALL PRINT$NUMBER(TOKEN);
      CALL PRINT$CHAR(' ');
      CALL PRINT$ACCUM;
    END;
 END;
ENDLOOKAHEAD;
NO$CONFLICT:PROCEDURE (CSTATE) BYTE;
 DECLARE (CSTATE,I,J,K) BYTE;
 J=INDEX1(CSTATE);
 K=J + INDEX2(CSTATE) - 1;
 DO I=J TO K;
    IF READ1(I)=TOKEN THEN RETURN TRUE;
 END;
RETURNFALSE;
ENDNO$CONFLICT;
RECOVER:PROCEDURE BYTE;
 DECLARE TSP BYTE, RSTATE BYTE;
 DO FOREVER;
    TSP=SP;
    DO WHILE TSP <> 255;
      IF NO$CONFLICT(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; /* TRY ANOTHER TOKEN */
 END;
ENDRECOVER;
 /* * * * * PROGRAM EXECUTION STARTS HERE * * */
 /* INITIALIZATION */
TOKEN=63;  /* PRIME THE SCANNER WITH -PROCEDURE- */
```

```
CALLMOVE(PASS1$TOP-PASS1$LEN,.OUTPUT$FCB,PASS1$LEN);
 /* THIS SETS
    OUTPUT FILE CONTROL BLOCK
    TOGGLES
    READ POINTER
    NEXT SYMBOL TABLE POINTER
 */
OUTPUT$END=(OUTPUT$PTR:=.OUTPUT$BUFF-1)+128;
CALLPRINT$ERROR(FALSE); /* INITIALIZE ERROR MSG OUTPUT */
 /* * * * * * * PARSER * * * * * * */
DO WHILE COMPILING;
 IF STATE <= MAXRNO THEN     /* READ STATE */
 DO;
    CALL INCSP;
    STATESTACK(SP) = STATE; /* SAVE CURRENT STATE */
    CALL LOOKAHEAD;
    I=GETIN1;
    J = I + GETIN2 - 1;
    DO I=I TO J;
      IF READ1(I) = TOKEN THEN
      DO;
      /* COPY THE ACCUMULATOR IF IT IS AN INPUT
      STRING. IF IT IS A RESERVED WORD IT DOES
      NOT NEED TO BE COPIED. */
         IF (TOKEN=INPUT$STR) OR (TOKEN=LITERAL) THEN
           DO K=Ø TO ACCUM(Ø);
              VARC(K)=ACCUM(K);
           END;
         STATE=READ2(I);
         NOLOOK=TRUE;
         I=J;
      END;
      ELSE
      IF I=J THEN
      DO;
         CALL PRINT$ERROR('NP');
         CALL PRINT(.ERROR$NEAR$$);
         CALL PRINT$ACCUM;
         IF (STATE:=RECOVER)=Ø THEN COMPILING=FALSE;
      END;
    END;
 END;  /* END OF READ STATE */
 ELSE
 IF STATE>MAXPNO THEN   /* APPLY PRODUCTION STATE */
 DO;
    MP=SP - GETIN2;
    MPP1=MP + 1;
    CALL CODE$GEN(STATE - MAXPNO);
    SP=MP;
    I=GETIN1;
    J=STATESTACK(SP);
    DO WHILE (K:=APPLY1(I)) <> Ø AND J<>K;
      I=I + 1;
    END;
```

```
    IF (K:=APPLY2(I))=0 THEN COMPILING=FALSE;
    STATE=K;
 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;
 ELSE
 DO;     /*PUSH STATES*/
    CALL INCSP;
    STATESTACK(SP)=GETIN2;
    STATE=GETIN1;
 END;
END;/* OF WHILE COMPILING */
CALLBYTE$OUT(TER);
DOWHILE OUTPUT$PTR<>.OUTPUT$BUFF;
 CALL BYTE$OUT(TER);
END;
CALLCLOSE;
CALLCRLF;
CALLPRINT(.END$OF$PART$2);
CALLBOOT;
END;
```

```
INTERP:     /*  MODULE " I N T E R P "    */
       DO;

    /*          COBOL INTERPRETER      */

 /*      NORMALLY ORG'ED TO X'100'      */

    /* GLOBAL DECLARATIONS AND LITERALS   */

DECLARE

LIT      LITERALLY      'LITERALLY',
BDOS          LIT       '5H',     /* ENTRY TO OPERATING
                                     STSTEM */
BOOT          LIT       '0',
CR            LIT       '13',
LF            LIT       '10',
TRUE          LIT       '1',
FALSE         LIT       '0',
FOREVER       LIT       'WHILE TRUE';

    /* UTILITY VARIABLES */

DECLARE

BOOTER        ADDRESS                 INITIAL (0000H),
INDEX         BYTE,
A$CTR         ADDRESS,
CTR           BYTE,
CTR1          BYTE,
BASE          ADDRESS,
B$BYTE        BASED BASE (1)          BYTE,
B$ADDR        BASED BASE (1)          ADDRESS,
HOLD          ADDRESS,
H$BYTE        BASED HOLD (1)          BYTE,
H$ADDR        BASED HOLD (1)          ADDRESS,


    /* CODE POINTERS */

CODE$START          LIT       '3000H',
PROGRAM$COUNTER     ADDRESS,
C$BYTE              BASED PROGRAM$COUNTER (1)   BYTE,
C$ADDR              BASED PROGRAM$COUNTER (1)   ADDRESS;


/* * * * *   GLOBAL INPUT AND OUTPUT ROUTINES * * * * * */


DECLARE
CURRENT$FCB ADDRESS,
START$OFFSET        LIT       '37';

MON1: PROCEDURE (F,A) EXTERNAL;
```

```
    DECLARE F BYTE, A ADDRESS;
END MON1;

MON2: PROCEDURE (F,A) BYTE EXTERNAL;
    DECLARE F BYTE, A ADDRESS;
END MON2;

PRINT$CHAR: PROCEDURE (CHAR);
    DECLARE CHAR BYTE;
    CALL MON1 (2,CHAR);
END PRINT$CHAR;

CRLF: PROCEDURE;
    CALL PRINT$CHAR(CR);
    CALL PRINT$CHAR(LF);
END CRLF;

PRINT: PROCEDURE (A);
    DECLARE A ADDRESS;
    CALL CRLF;
    CALL MON1(9,A);
END PRINT;


READ: PROCEDURE(A);
    DECLARE A ADDRESS;
    CALL MON1(10,A);
END READ;


PRINT$ERROR: PROCEDURE (CODE);
    DECLARE CODE ADDRESS;
    CALL CRLF;
    CALL PRINT$CHAR(HIGH(CODE));
    CALL PRINT$CHAR(LOW(CODE));
END PRINT$ERROR;


FATAL$ERROR: PROCEDURE(CODE);
    DECLARE CODE ADDRESS;
    CALL PRINT$ERROR(CODE);
    CALL BOOTER;
END FATAL$ERROR;


SET$DMA: PROCEDURE;
    CALL MON1 (26, CURRENT$FCB + START$OFFSET);
END SET$DMA;


OPEN: PROCEDURE (ADDR) BYTE;
    DECLARE ADDR ADDRESS;
    CALL SET$DMA;    /* INSURE DIRECTORY READ WON'T
                        CLOBBER CORE */
```

```
    RETURN MON2(15,ADDR);
END OPEN;


CLOSE: PROCEDURE (ADDR);
    DECLARE ADDR ADDRESS;
    IF MON2(16,ADDR)=255 THEN CALL FATAL$EPROR('CL');
END CLOSE;


DELETE: PROCEDURE;
    CALL MON1(19,CURRENT$FCB);
END DELETE;


MAKE: PROCEDURE (ADDR);
    DECLARE ADDR ADDRESS;
    IF MON2(22,ADDR)=255 THEN CALL FATAL$ERROR('ME');
END MAKE;


DISK$READ: PROCEDURE BYTE;
    RETURN MON2(20,CURRENT$FCB);
END DISK$READ;


DISK$WRITE: PROCEDURE BYTE;
    RETURN MON2(21,CURRENT$FCB);
END DISK$WRITE;


/* * * * * UTILITY PROCEDURES * * * * * * * */


DECLARE
SUBSCRIPT           (8)       ADDRESS;


RES: PROCEDURE(ADDR) ADDRESS;
    /* THIS PROCEDURE RESOLVES THE ADDRESS OF A
    SUBSCRIPTED IDENTIFIER OR A LITERAL CONSTANT */

    DECLARE ADDR ADDRESS;
    IF ADDR > 32 THEN RETURN ADDR;
    IF ADDR < 9 THEN RETURN SUBSCRIPT(ADDR);
    DO CASE ADDR - 9;
        RETURN .('0');
        RETURN .(' ');
        RETURN .(' ');
    END;
    RETURN 0;
END RES;
```

```
MOVE: PROCEDURE(FROM,DESTINATION,COUNT);
    DECLARE (FROM,DESTINATION,COUNT) ADDRESS,
         (F BASED FROM, D BASED DESTINATION) BYTE;
    DO WHILE (COUNT:=COUNT - 1) <> 0FFFFH;
         D=F;
         FROM=FROM + 1;
         DESTINATION=DESTINATION + 1;
    END;
END MOVE;


FILL: PROCEDURE(DESTINATION,COUNT,CHAR);
    DECLARE (DESTINATION,COUNT) ADDRESS,

         (CHAR,D BASED DESTINATION) BYTE;
    DO WHILE (COUNT:=COUNT - 1)<> 0FFFFH;
         D=CHAR;
         DESTINATION=DESTINATION + 1;
    END;
END FILL;


CONVERT$TO$HEX: PROCEDURE(POINTER,COUNT) ADDRESS;
  DECLARE POINTER ADDRESS, COUNT BYTE;
  A$CTR=0;
  BASE=POINTER;
  DO CTR = 0 TO COUNT-1;
   A$CTR=SHL(A$CTR,3) + SHL(A$CTR,1) + B$BYTE(CTR) - '0';
  END;
  RETURN A$CTR;
END CONVERT$TO$HEX;


/* * * * * CODE CONTROL PROCEDURES * * * * * */

DECLARE

BRANCH$FLAG         BYTE       INITIAL(FALSE);

INC$PTR: PROCEDURE (COUNT);
    DECLARE COUNT BYTE;
    PROGRAM$COUNTER=PROGRAM$COUNTER + COUNT;
END INC$PTR;


GET$OP$CODE: PROCEDURE BYTE;
    CTR=C$BYTE(0);
    CALL INC$PTR(1);
    RETURN CTR;
END GET$OP$CODE;


COND$BRANCH: PROCEDURE(COUNT);
    /* THIS PROCEDURE CONTROLS BRANCHING INSTRUCTIONS */
```

```
    DECLARE COUNT BYTE;
    IF  BRANCH$FLAG THEN
    DO;
         BRANCH$FLAG=FALSE;
         PROGRAM$COUNTER=C$ADDR(COUNT);
    END;
    ELSE CALL INC$PTR(SHL(COUNT,1)+2);
END COND$BRANCH;


INCR$OR$BRANCH: PROCEDURE(MARK);
    DECLARE MARK BYTE;
    IF MARK THEN CALL INC$PTR(2);
    ELSE PROGRAM$COUNTER=C$ADDR(0);
END INCR$OR$BRANCH;

/* * * * * *COMPARISONS * * * * * * * * */



CHAR$COMPARE: PROCEDURE BYTE;
    BASE=C$ADDR(0);
    HOLD=C$ADDR(1);
    DO A$CTR=0 TO C$ADDR(2) - 1;
         IF B$BYTE(A$CTR) > H$BYTE(A$CTR) THEN RETURN 1;
         IF B$BYTE(A$CTR) < H$BYTE(A$CTR) THEN RETURN 0;
    END;
    RETURN 2;
END CHAR$COMPARE;


STRING$COMPARE: PROCEDURE(PIVOT);
  DECLARE PIVOT BYTE;
  IF CHAR$COMPARE=PIVOT THEN BRANCH$FLAG=NOT BRANCH$FLAG;
  CALL COND$BRANCH(3);
END STRING$COMPARE;


NUMERIC: PROCEDURE(CHAR) BYTE;
    DECLARE CHAR BYTE;
    RETURN (CHAR >='0') AND (CHAR <='9');
END NUMERIC;


LETTER: PROCEDURE(CHAR) BYTE;
    DECLARE CHAR BYTE;
    RETURN (CHAR >='A') AND (CHAR <='Z');
END LETTER;


SIGN: PROCEDURE(CHAR) BYTE;
    DECLARE CHAR BYTE;
    RETURN (CHAR='+') OR (CHAR='-');
```

```
END SIGN;


COMP$NUM$UNSIGNED: PROCEDURE;
    BASE=C$ADDR(0);
    DO A$CTR=0 TO C$ADDR(2)-1;
         IF NOT NUMERIC(B$BYTE(A$CTR)) THEN
         DO;
              BRANCH$FLAG=NOT BRANCH$FLAG;
              RETURN;
         END;
    END;
    CALL COND$BRANCH(2);
END COMP$NUM$UNSIGNED;


COMP$NUM$SIGN: PROCEDURE;
    BASE=C$ADDR(0);
    DO A$CTR=0 TO C$ADDR(2)-1;
         IF NOT(NUMERIC(CTR:=B$BYTE(A$CTR))
              OR SIGN(CTR)) THEN
         DO;
              BRANCH$FLAG=NOT BRANCH$FLAG;
              RETURN;
         END;
    END;
    CALL COND$BRANCH(2);
END COMP$NUM$SIGN;


COMP$ALPHA: PROCEDURE;
    BASE=C$ADDR(0);
    DO A$CTR=0 TO C$ADDR(2)-1;
         IF NOT LETTER(B$BYTE(A$CTR)) THEN
         DO;
              BRANCH$FLAG=NOT BRANCH$FLAG;
              RETURN;
         END;
    END;
    CALL COND$BRANCH(2);
END COMP$ALPHA;



/* * * * * *NUMERIC OPERATIONS * * * * * * */


DECLARE

(R0,R1,R2)          (10)      BYTE, /* REGISTERS */
SIGN0(3)           BYTE,
(DEC$PT0,DEC$PT1,DEC$PT2)     BYTE,
DEC$PTA (3)         BYTE AT (.DEC$PT0),
OVERFLOW            BYTE,
```

```
R$PTR                   BYTE,
SWITCH                  BYTE,
SIGNIF$NO               BYTE,
ZONE                    LIT         '10H',
POSITIVE                LIT         '1',
NEGITIVE                LIT         '0';


CHECK$FOR$SIGN: PROCEDURE(CHAR) BYTE;
    DECLARE CHAR BYTE;
    IF NUMERIC(CHAR) THEN RETURN POSITIVE;
    IF NUMERIC(CHAR - ZONE) THEN RETURN NEGITIVE;
    CALL PRINT$ERROR('SI');
    RETURN POSITIVE;
END CHECK$FOR$SIGN;


STORE$IMMEDIATE: PROCEDURE;
    DO CTR=0 TO 9;
         R0(CTR)=R2(CTR);
    END;
    DEC$PT0=DEC$PT2;
    SIGN0(0)=SIGN0(2);
END STORE$IMMEDIATE;


ONE$LEFT: PROCEDURE;
  DECLARE (CTR, FLAG) BYTE;
  IF ((FLAG:=SHR(B$BYTE(0),4))=0) OR (FLAG=9) THEN
  DO;
   DO CTR=0 TO 8;
    B$BYTE(CTR)=SHL(B$BYTE(CTR),4) OR
     SHR(B$BYTE(CTR+1),4);
   END;
   B$BYTE(9)=SHL(B$BYTE(9),4) OR FLAG;
  END;
  ELSE OVERFLOW=TRUE;
END ONE$LEFT;


ONE$RIGHT: PROCEDURE;
    DECLARE CTR BYTE;
    CTR=10;
    DO INDEX=1 TO 9;
         CTR=CTR-1;
         B$BYTE(CTR)=SHR(B$BYTE(CTR),4) OR
          SHL(B$BYTE(CTR-1),4);
    END;
    B$BYTE(0)=SHR(B$BYTE(0),4);
    IF B$BYTE(0) = 09H THEN
        B$BYTE(0) = 99H;
END ONE$RIGHT;
```

```
SHIFT$RIGHT: PROCEDURE(COUNT);
    DECLARE COUNT BYTE;
    DO CTR=1 TO COUNT;
         CALL ONE$RIGHT;
    END;
END SHIFT$RIGHT;


SHIFT$LEFT: PROCEDURE (COUNT);
    DECLARE COUNT BYTE;
    OVERFLOW=FALSE;
    DO CTR=1 TO COUNT;
         CALL ONE$LEFT;
         IF OVEPFLOW THEN RETURN;
    END;
END SHIFT$LEFT;


ALLIGN: PROCEDURE;
    BASE=.RØ;
    IF DEC$PTØ > DEC$PT1 THEN
    CALL SHIFT$RIGHT(DEC$PTØ-DEC$PT1);
    ELSE CALL SHIFT$LEFT(DEC$PT1-DEC$PTØ);
END ALLIGN;


ADD$RØ: PROCEDURE(SECOND, DEST);
    DECLARE (SECOND, DEST) ADDRESS, (CY,A,B,I,J) BYTE;
    HOLD= SECOND;
    BASE = DEST;
    CY=Ø;
    CTR=9;
    DO J=1 TO 1Ø;
         A=RØ(CTR);
         B=H$BYTE(CTR);
         I=DEC(A+CY);
         CY=CARRY;
         I=DEC(I + B);
         CY=(CY OR CARRY) AND 1;
         B$BYTE(CTR)=I;
         CTR=CTR-1;
    END;
    IF CY THEN
    DO;
         CTR=9;
         DO J = 1 TO 1Ø;
              I=B$BYTE(CTR);
              I=DEC(I+CY);
              CY=CARRY AND 1;
              B$BYTE(CTR)=I;
              CTR=CTR-1;
         END;
    END;
END ADD$RØ;
```

```
COMPLIMENT: PROCEDURE(NUMB);
    DECLARE NUMB BYTE;

    SIGNØ(NUMB) = SIGNØ(NUMB) XOR 1;  /* COMPLIMENT
                                         SIGN */
    DO CASE NUMB;
         HOLD=.RØ;
         HOLD=.R1;
         HOLD=.R2;
    END;

    DO CTR=Ø TO 9;
         H$BYTE(CTR)=99H - H$BYTE(CTR);
    END;

END COMPLIMENT;



R2$ZERO: PROCEDURE BYTE;
   DECLARE I BYTE;
   IF (SHL(R2(Ø),4)<>Ø) OR (SHR(R2(9),4)<>Ø)
   THEN RETURN FALSE;
   ELSE DO I=1 TO 8;
          IF R2(I)<>Ø THEN RETURN FALSE;
        END;
   RETURN TRUE;
END R2$ZERO;

CHECK$RESULT: PROCEDURE;
    IF SHR(R2(Ø),4)=9 THEN CALL COMPLIMENT(2);
    IF SHR(R2(Ø),4)<>Ø THEN OVERFLOW=TRUE;
END CHECK$RESULT;


CHECK$SIGN: PROCEDURE;
    IF SIGNØ(Ø) AND SIGNØ(1) THEN
    DO;
         SIGNØ(2)=POSITIVE;
         RETURN;
    END;
    SIGNØ(2)=NEGITIVE;
    IF NOT SIGNØ(Ø) AND NOT SIGNØ(1) THEN RETURN;
    IF SIGNØ(Ø) THEN CALL COMPLIMENT(1);
    ELSE CALL COMPLIMENT(Ø);
END CHECK$SIGN;


LEADING$ZEROES: PROCEDURE (ADDR) BYTE;
    DECLARE COUNT BYTE, ADDR ADDRESS;
    COUNT=Ø;
```

```
    BASE=ADDR;
    DO CTR=0 TO 9;
         IF (B$BYTE(CTR) AND 0F0H) <> 0 THEN RETURN COUNT;
         COUNT=COUNT + 1;
         IF (B$BYTE(CTR) AND 0FH) <> 0 THEN RETURN COUNT;
         COUNT=COUNT + 1;
    END;
    RETURN COUNT;
END LEADING$ZEROES;


CHECK$DECIMAL: PROCEDURE;
 IF DEC$PT2<>(CTR:=C$BYTE(3)) THEN
 DO;
      BASE=.R2;
      IF DEC$PT2 > CTR THEN CALL SHIFT$RIGHT(DEC$PT2-CTR);
      ELSE CALL SHIFT$LEFT(CTR-DEC$PT2);
 END;
 IF LEADING$ZEROES(.R2) < 19 - C$BYTE(2) THEN OVERFLOW
   = TRUE;
END CHECK$DECIMAL;


ADD: PROCEDURE;
    OVERFLOW=FALSE;
    CALL ALLIGN;
    CALL CHECK$SIGN;
    CALL ADDR0(.R1,.R2);
    CALL CHECK$RESULT;
END ADD;


ADD$SERIES: PROCEDURE(COUNT);
    DECLARE (I,COUNT) BYTE;
    DO I=1 TO COUNT;
         CALL ADD$R0(.R2,.R2);
    END;
END ADD$SERIES;


SET$MULT$DIV: PROCEDURE;
    OVERFLOW=FALSE;
    SIGN0(2) = (NOT (SIGN0(0) XOR SIGN0(1))) AND 01H;
    CALL FILL(.R2,10,0);
END SET$MULT$DIV;


R1$GREATER: PROCEDURE BYTE;
    DECLARE I BYTE;
    DO CTR=0 TO 9;
         IF R1(CTR)>(I:=99H-R0(CTR)) THEN RETURN TRUE;
         IF R1(CTR)<I THEN RETURN FALSE;
    END;
    RETURN TRUE;
```

```
END R1$GREATER;


MULTIPLY: PROCEDURE(VALUE);
    DECLARE VALUE BYTE;
    IF VALUE<>0 THEN CALL ADD$SERIES(VALUE);
    BASE=.R0;
    CALL ONE$LEFT;
END MULTIPLY;


DIVIDE: PROCEDURE;
    DECLARE (I, J, K, LZ0, LZ1, X) BYTE;
    CALL SET$MULT$DIV;
    IF   (LZ0:=LEADING$ZEROES(.R0))<>
         (X := (LZ1 := LEADING$ZEROES(.R1))) THEN
     DO;
         IF LZ0>LZ1 THEN
         DO;
              BASE = .R0;
              CALL SHIFT$LEFT(I := LZ0-LZ1);
              DEC$PT0=DEC$PT0 + I;
              X = LZ1;
         END;
         ELSE DO;
              BASE = .R1;
              CALL SHIFT$LEFT (I:=LZ1-LZ0);
              DEC$PT1=DECPT1 + I;
              X = LZ0;
         END;
    END;
    DECPT2= 18 - X + DECPT1 - DECPT0;
    CALL COMPLIMENT(0);
    DO I = X TO 19;
         J=0;
         DO WHILE R1$GREATER;
              CALL ADD$R0(.R1,.R1);
              IF R1(0) = 99H THEN
                      CALL COMPLIMENT (1);
              J=J+1;
         END;
         K=SHR(I,1);
         IF I THEN R2(K)=R2(K) OR J;
         ELSE R2(K)=R2(K) OR SHL(J,4);
         BASE=.R0;
         CALL ONE$RIGHT;
    END;
END DIVIDE;



LOAD$A$CHAR: PROCEDURE(CHAR);
    DECLARE CHAR BYTE;
```

```
    IF (SWITCH:=NOT SWITCH) THEN
         B$BYTE(R$PTR)=B$BYTE(R$PTR) OR SHL(CHAR - 30H,4);
    ELSE B$BYTE(R$PTR:=R$PTR-1)=CHAR - 30H;
END LOAD$A$CHAR;


LOAD$NUMBERS: PROCEDURE(ADDP,CNT);
    DECLARE ADDR ADDRESS, (I,CNT)BYTE;
    HOLD=RES(ADDR);
    CTR=CNT;
    DO INDEX = 1 TO CNT;
         CTR=CTR-1;
         CALL LOAD$A$CHAR(H$BYTE(CTR));
    END;
    CALL INC$PTR(5);
END LOAD$NUMBERS;


SET$LOAD: PROCEDURE (SIGN$IN);
  DECLARE SIGN$IN BYTE;
    DO CASE (CTR:=C$BYTE(4));
         BASE=.R0;
         BASE=.R1;
         BASE=.R2;
    END;
    DEC$PTA(CTR)=C$BYTE(3);
    SIGN0(CTR)=SIGN$IN;
    CALL FILL (BASE,10,0);
    R$PTR=9;
    SWITCH=FALSE;
END SET$LOAD;


LOAD$NUMERIC: PROCEDURE;
    CALL SET$LOAD(1);
    CALL LOAD$NUMBERS(C$ADDR(0),C$BYTE(2));
END LOAD$NUMERIC;


LOAD$NUM$LIT: PROCEDURE;
    DECLARE(LIT$SIZE,FLAG) BYTE;

    CHAR$SIGN: PROCEDURE;
         LIT$SIZE=LIT$SIZE - 1;
         HOLD=HOLD + 1;
    END CHAR$SIGN;

    LIT$SIZE=C$BYTE(2);
    HOLD=C$ADDR(0);
    IF H$BYTE(0)='-' THEN
    DO;
         CALL CHAR$SIGN;
         CALL SET$LOAD(NEGITIVE);
    END;
```

```
    ELSE DO;
         IF H$BYTE(0)='+' THEN CALL CHAR$SIGN;
         CALL SET$LOAD(POSITIVE);
    END;
    FLAG=0;
    CTR=LIT$SIZE;
    DO INDEX=1 TO LIT$SIZE;
         CTR=CTR-1;
         IF H$BYTE(CTR)='.' THEN FLAG=LIT$SIZE - (CTR+1);
         ELSE CALL LOAD$A$CHAR(H$BYTE(CTR));
    END;
    DEC$PTA(C$BYTE(4))= FLAG;
    CALL INC$PTR(5);
END LOAD$NUM$LIT;


STORE$ONE: PROCEDURE;
    IF(SWITCH:=NOT SWITCH) THEN
         B$BYTE(0)=SHR(H$BYTE(0),4) OR '0';
    ELSE DO;
         HOLD=HOLD-1;
         B$BYTE(0)=(H$BYTE(0) AND 0FH) OR '0';
    END;
    BASE=BASE-1;
END STORE$ONE;


STORE$AS$CHAR: PROCEDURE(COUNT);
    DECLARE COUNT BYTE;
    SWITCH=FALSE;
    HOLD=.R2 + 9;
    DO CTR=1 TO COUNT;
         CALL STORE$ONE;
    END;
END STORE$AS$CHAR;


SET$ZONE: PROCEDURE (ADDR);
    DECLARE ADDR ADDRESS;
    IF NOT SIGN0(2) THEN
    DO;
         BASE=ADDR;
         B$BYTE(0)=B$BYTE(0) OR ZONE;
    END;
    CALL INC$PTR(4);
END SET$ZONE;


SET$SIGN$SEP: PROCEDURE (ADDR);
    DECLARE ADDR ADDRESS;
    BASE=ADDR;
    IF SIGN0(2) THEN B$BYTE(0)='+';
    ELSE B$BYTE(0)='-';
    CALL INC$PTR(4);
```

```
END SET$SIGN$SEP;


STORE$NUMERIC: PROCEDURE;
    CALL CHECK$DECIMAL;
    BASE=C$ADDR(0) + C$BYTE(2) -1;
    CALL STORE$AS$CHAR(C$BYTE(2));
END STORE$NUMERIC;




/* * * * * INPUT-OUTPUT ACTIONS * * * * * */


DECLARE

EOF$FLAG$OFFSET     LIT         '36',
FLAG$OFFSET         LIT         '33',
EXTENT$OFFSET       LIT         '12',
REC$NO              LIT         '32',
PTR$OFFSET          LIT         '17',
BUFF$LENGTH         LIT         '128',
VAR$END             LIT         'CR',
TERMINATOR          LIT         '1AH',
HIGH$VALUE          LIT         '0FFH',
INVALID             BYTE,
REWRITE$FLAG        BYTE    INITIAL (0H),
RANDOM$FILE         BYTE,
CURRENT$FLAG        BYTE,
FCB$BYTE            BASED CURRENT$FCB         BYTE,
FCB$ADDR            BASED CURRENT$FCB         ADDRESS,
FCB$BYTE$A          BASED CURRENT$FCB (1) BYTE,
FCB$ADDR$A          BASED CURRENT$FCB (1) ADDRESS,
BUFF$PTR            ADDRESS,
BUFF$END            ADDRESS,
BUFFSTART           ADDRESS,
BUFF$BYTE           BASED BUFF$PTR       BYTE,
CON$BUFF            ADDRESS    INITIAL (80H),
CON$BYTE            BASED CON$BUFF       BYTE,
CON$INPUT           ADDRESS    INITIAL (82H);


ACCEPT: PROCEDURE;
    CALL CRLF;
    CALL PRINT$CHAR(3FH);
/*  CALL CRLF;  */
    CALL FILL(CON$INPUT,(CON$BYTE:=C$BYTE(2)),' ');
    CALL READ(CON$BUFF);
    CALL MOVE(CON$INPUT,RES(C$ADDR(0)),CON$BYTE);
    CALL INC$PTR(3);
END ACCEPT;
```

```
DISPLAY: PROCEDURE;
    DECLARE  B$CNT BYTE;
    BASE=C$ADDR(0);
    IF NOT C$BYTE(3) THEN CALL CRLF;
    B$CNT = C$BYTE(2);
    DO CTR = 0 TO B$CNT - 1;
        CALL PRINT$CHAR(B$BYTE(CTR));
    END;
    CALL INC$PTR(4);
END DISPLAY;


GET$FILE$TYPE: PROCEDURE BYTE;
    BASE=C$ADDR(0);
    RETURN B$BYTE(FLAG$OFFSET);
END GET$FILE$TYPE;


SET$FILE$TYPE: PROCEDURE(TYPE);
    DECLARE TYPE BYTE;
    BASE=C$ADDR(0);
    IF GET$FILE$TYPE<>0 THEN CALL FATAL$ERROR('OE');
    B$BYTE(FLAG$OFFSET)=TYPE;
END SET$FILE$TYPE;


SET$I$O: PROCEDURE;
    INVALID=FALSE;
    IF C$ADDR(0)=CURRENT$FCB THEN RETURN;
    /* STORE CURRENT POINTERS AND SET INTERNAL
       WRITE MARK */
    BASE=CURRENT$FCB;
    FCB$ADDR$A(PTR$OFFSET)=BUFF$PTR;
    FCB$BYTE$A(FLAG$OFFSET)=CURRENT$FLAG;
    /* LOAD NEW VALUES */
    BUFF$END=(BUFF$START:=(CURRENT$FCB:=C$ADDR(0))
     + START$OFFSET) + BUFF$LENGTH;
    CURRENT$FLAG=FCB$BYTE$A(FLAG$OFFSET);
    BUFF$PTR=FCB$ADDR$A(PTR$OFFSET);
END SET$I$O;


OPEN$FILE: PROCEDURE(TYPE);
    DECLARE TYPE BYTE;
    CALL SET$FILE$TYPE(TYPE);
    CTR=OPEN(CURRENT$FCB:=C$ADDR(0));
    DO CASE TYPE-1;
        /* INPUT */
        DO;
            IF CTR=255 THEN CALL FATAL$ERROR('NF');
        END;
        /* OUTPUT */
        DO;
```

```
            CALL DELETE;
            CALL MAKE(C$ADDR(0));
        END;
        ; /* CASE 2 NOT USED */
        /* I-O */
        DO;
            IF CTR=255 THEN CALL FATAL$ERROR('NF');
        END;
    END;
    FCB$BYTE$A(EXTENT$OFFSET)=0;   /* SET THE EXTENT FIELD
                                      IN FCB */
    FCB$BYTE$A(REC$NO)=0;         /* SET THE RECORD NUMBER
                                     IN FCB */
    FCB$BYTE$A(ECF$FLAG$OFFSET)=FALSE;
                          /* SET THE EOF INDICATOR OFF */
    BUFF$END=(BUFF$START:=(CURRENT$FCB + START$OFFSET))
    + BUFF$LENGTH;
    CURRENT$FLAG=FCB$BYTE$A(FLAG$OFFSET);
    BUFF$PTR,FCB$ADDR$A(PTR$OFFSET)=BUFF$START-1;
    CALL INC$PTR(2);
END OPEN$FILE;


WRITE$MARK: PROCEDURE BYTE;
    RETURN ROL(CURRENT$FLAG,1);
END WRITE$MARK;


SET$WRITE$MARK: PROCEDURE;
    CURRENT$FLAG=CURRENT$FLAG OR 80H;
END SET$WRITE$MARK;


WRITE$RECORD: PROCEDURE;
    CALL SET$DMA;
    CURRENT$FLAG=CURRENT$FLAG AND 0FH;
    IF (CTR:=DISK$WRITE) =0 THEN RETURN;
    CALL PRINT$ERRCR('W3');
    INVALID=TRUE;
END WRITE$RECORD;


READ$RECORD: PROCEDURE;
    CALL SET$DMA;
    IF WRITE$MARK THEN CALL WRITE$RECORD;
    IF (CTR:=DISK$READ)=0 THEN RETURN;
    IF CTR=1 THEN FCB$BYTE$A(ECF$FLAG$OFFSET)=TRUE;
    INVALID=TRUE;
END READ$RECORD;


READ$BYTE: PROCEDURE BYTE;
    IF (BUFF$PTR:=BUFF$PTR + 1) >= BUFFEND THEN
    DO;
```

```
          CALL READ$RECORD;
          IF FCB$BYTE$A(EOF$FLAG$OFFSET) THEN
            RETURN TERMINATOR;
          BUFF$PTR=BUFF$START;
     END;
     RETURN BUFF$BYTE;
END READ$BYTE;


WRITE$BYTE: PROCEDURE (CHAR);
     DECLARE CHAR BYTE;
     IF (BUFF$PTR:=BUFF$PTR+1) >= BUFF$END THEN
     DO;
          CALL WRITE$RECORD;
          BUFF$PTR=BUFF$START;
          IF REWRITE$FLAG THEN
             DO;
                CALL READ$RECORD;
                FCB$BYTE$A(REC$NO)=FCB$BYTE$A(REC$NO)-1;
             END;
     END;
     CALL SET$WRITE$MARK;
     BUFF$BYTE=CHAR;
END WRITE$BYTE;


WRITE$END$MARK: PROCEDURE;
     CALL WRITE$BYTE(CR);
     CALL WRITE$BYTE(LF);
END WRITE$END$MARK;


READ$END$MARK: PROCEDURE;
     IF READ$BYTE<>CR THEN CALL PRINT$ERROR('EM');
     IF READ$BYTE<>LF THEN CALL PRINT$ERROR('EM');
END READ$END$MARK;


READ$VARIABLE:PROCEDURE;
     CALL SET$I$O;
     BASE=C$ADDR(1);
     DO A$CTR=0 TO C$ADDR(2)-1;
       IF (CTR:=(B$BYTE(A$CTR):=READ$BYTE)) = VAR$END THEN
       DO;
            CTR=READ$BYTE;
            RETURN;
       END;
       IF CTR=TERMINATOR THEN
       DO;
            FCB$BYTE$A(EOF$FLAG$OFFSET)=TRUE;
            RETURN;
       END;
  END;
  CALL READ$END$MARK;
```

```
END READ$VARIABLE;


WRITE$VARIABLE: PROCEDURE;
    DECLARE COUNT ADDRESS;
    CALL SET$ISO;
    BASE=C$ADDR(1);
    COUNT=C$ADDR(2);
    DO WHILE(B$BYTE(COUNT:=COUNT-1)<>' ')AND (COUNT<>0);
    END;
    DO A$CTR=0 TO COUNT;
        CALL WRITE$BYTE(B$BYTE(A$CTR));
    END;
    CALL WRITE$END$MARK;
END WRITE$VARIABLE;


READ$TO$MEMORY: PROCEDURE;
    BASE=C$ADDR(1);
    DO A$CTR=0 TO C$ADDR(2)-1;
        IF (B$BYTE(A$CTR):=READ$BYTE)=TERMINATOR THEN
        DO;
            FCB$BYTE$A(EOF$FLAG$OFFSET)=TRUE;
            RETURN;
        END;
    END;
    CALL READ$END$MARK;
END READ$TO$MEMORY;


WRITE$FROM$MEMORY: PROCEDURE;
    BASE=C$ADDR(1);
    DO A$CTR=0 TO C$ADDR(2)-1;
        CALL WRITE$BYTE(B$BYTE(A$CTR));
    END;
    CALL WRITE$END$MARK;
END WRITE$FROM$MEMORY;


/* * * * * RANDOM I-O PROCEDURES * * * * */


SET$RANDOM$POINTER: PROCEDURE;
    /*
    THIS PROCEDURE READS THE RANDOM KEY AND COMPUTES
    WHICH RECORD NEEDS TO BE AVAILABLE IN THE BUFFER
    THAT RECORD IS MADE AVAILABLE AND THE POINTERS
    SET FOR INPUT OR OUTPUT
    */
    DECLARE (BYTE$COUNT,RECORD) ADDRESS,
        EXTENT BYTE;
    IF WRITE$MARK THEN CALL WRITE$RECORD;
    BYTE$COUNT=(C$ADDR(2)+2)*(CONVERT$TO$HEX(C$ADDR(3)
    ,C$BYTE(8))-1);
```

```
    RECORD=SHR(BYTE$COUNT,7);
    EXTENT=SHR(RECORD,7);
    IF EXTENT<>FCB$BYTE$A(EXTENT$OFFSET) THEN
    DO;
      CALL CLOSE(C$ADDR(0));
      FCB$BYTE$A(EXTENT$OFFSET)=EXTENT;
      IF OPEN(C$ADDR(0))<>0 THEN
      DO;
       IF SHR(CURRENT$FLAG,1) THEN CALL MAKE(C$ADDR(0));
       ELSE INVALID=TRUE;
      END;
    END;
    BUFF$PTR=(BYTE$COUNT AND 7FH) + BUFF$START -1;
    FCB$BYTE$A(32)=LOW(RECORD) AND 7FH;
    CALL READ$RECORD;
END SET$RANDOM$POINTER;

GET$REC$NUMBER: PROCEDURE ADDRESS;
    DECLARE (RECORD,LOGICAL$REC$NUM,BYTE$COUNT) ADDRESS;
    RECORD=SHL(FCB$BYTE$A(EXTENT$OFFSET),7)
    + FCB$BYTE$A(REC$NO);
    IF NOT SHR(CURRENT$FLAG,1) THEN RECORD=RECORD-1;
    BYTE$COUNT=SHL(RECORD,7) + ((BUFF$PTR+1)-BUFF$START);
    LOGICAL$REC$NUM=(BYTE$COUNT/(C$ADDR(2)+2))+1;
    RETURN LOGICAL$REC$NUM;
END GET$REC$NUMBER;

SET$RELATIVE$KEY: PROCEDURE;
    DECLARE (REC$NUM, K) ADDRESS,
         (I,CNT) BYTE,
         J(4) ADDRESS DATA (10000,1000,100,10),
         BUFF(5) BYTE;
    REC$NUM=GET$REC$NUMBER;
    DO I=0 TO 3;
    CNT=0;
         DO WHILE REC$NUM>=(K:=J(I));
              REC$NUM=REC$NUM - K;
              CNT=CNT + 1;
         END;
         BUFF(I)=CNT + '0';
    END;
    BUFF(4)=REC$NUM+'0';
    IF (I:=C$BYTE(8))<=5 THEN
    CALL MOVE(.BUFF+5-I,C$ADDR(3),I);
    ELSE DO;
    CALL FILL(C$ADDR(0),I-5,' ');
         CALL MOVE(.BUFF,C$ADDR(3)+I-6, 5);
    END;
END SET$RELATIVE$KEY;

WRITE$EMPTY$RECORD: PROCEDURE;
    DO A$CTR=1 TO C$ADDR(2);
         CALL WRITE$BYTE(HIGH$VALUE);
    END;
```

```
    CALL WRITE$END$MARK;
END WRITE$EMPTY$RECORD;

WRITE$DUMMY$RECORDS: PROCEDURE(DIFFERENCE);
    DECLARE DIFFERENCE ADDRESS, COUNT BYTE;
    DO COUNT=1 TO DIFFERENCE;
       CALL WRITE$EMPTY$RECORD;
    END;
END WRITE$DUMMY$RECORDS;

BACK$ONE$EXTENT: PROCEDURE;
    CALL CLOSE(C$ADDR(0));
    IF FCB$BYTE$A(EXTENT$OFFSET):=
     FCB$BYTE$A(EXTENT$OFFSET)-1=255 THEN
           CALL FATAL$ERROR('W?');
    IF OPEN(C$ADDR(0))<>0 THEN
       DO;
          CALL PRINT$ERROR('OP');
          INVALID=TRUE;
          RETURN;
       END;
    FCB$BYTE$A(REC$NO)=127;
END BACK$ONE$EXTENT;

BACK$ONE$RECORD: PROCEDURE;
  IF(BUFF$PTR:=BUFF$PTR-(C$ADDR(2)+2))>=BUFF$START-1 THEN
       DO;
           FCB$BYTE$A(REC$NO)=FCB$BYTE$A(REC$NO)-1;
           RETURN;
       END;
     BUFF$PTR=BUFF$END-(BUFF$START-BUFF$PTR);
     IF FCB$BYTE$A(REC$NO)=0 THEN
        DO;
           CALL BACK$ONE$EXTENT;
           IF INVALID THEN RETURN;
           CALL READ$RECORD;
           CALL BACK$ONE$EXTENT;
        END;
     ELSE
        DO;
           FCB$BYTE$A(REC$NO)=FCB$BYTE$A(REC$NO)-2;
           CALL READ$RECORD;
           FCB$BYTE$A(REC$NO)=FCB$BYTE$A(REC$NO)-1;
        END;
END BACK$ONE$RECORD;

REWRITE$SEQ: PROCEDURE(FLAG);
     DECLARE FLAG BYTE;
     CALL BACK$ONE$RECORD;
     REWRITE$FLAG=TRUE;
     IF FLAG THEN CALL WRITE$FROM$MEMORY;
                              /* THIS IS A REWRITE */
      ELSE CALL WRITE$EMPTY$RECORD;  /* THIS IS
                                        A DELETE */
```

```
     CALL WRITE$RECORD;
     FCB$BYTE$A(REC$NO)=FCB$BYTE$A(REC$NO)-1;
     REWRITE$FLAG=FALSE;
     CALL READ$RECORD;
END REWRITE$SEQ;

CHECK$DIFFERENCE: PROCEDURE;
 DECLARE (DIFFERENCE,NEXT$RECORD,NEXT$KEY) ADDRESS;
   NEXT$RECORD=GET$REC$NUMBER;
   NEXT$KEY=CONVERT$TO$HEX(C$ADDR(3),C$BYTE(2));
   IF NEXT$RECORD > NEXT$KEY THEN CALL FATAL$ERROR('W2');
   DIFFERENCE=NEXT$KEY-NEXT$RECORD;
   IF DIFFERENCE > 0 THEN
   CALL WRITE$DUMMY$RECORDS(DIFFERENCE);
END CHECK$DIFFERENCE;

/* * * * * * * MOVES * * * * * * * */


INC$HOLD: PROCEDURE;
    HOLD=HOLD + 1;
    CTR=CTR + 1;
END INC$HOLD;


LOAD$INC: PROCEDURE;
    H$BYTE(0)=B$BYTE(0);
    BASE=BASE+1;
    CTR1=CTR1 + 1;
    CALL INC$HOLD;
END LOAD$INC;


CHECK$EDIT: PROCEDURE(CHAR);
    DECLARE CHAR BYTE;
    IF (CHAR='0') OR (CHAR='/') THEN CALL INC$HOLD;
    ELSE IF CHAR='B' THEN
    DO;
         H$BYTE(0)=' ';
         CALL INC$HOLD;
    END;
    ELSE IF CHAR='A' THEN
    DO;
     IF NOT LETTER(B$BYTE(0)) THEN CALL PRINT$ERROR('IC');
     CALL LOAD$INC;
    END;
    ELSE IF CHAR='9' THEN
    DO;
     IF NOT NUMERIC (B$BYTE(0)) THEN
     CALL PRINT$ERROR('IC');
     CALL LOAD$INC;
    END;
    ELSE CALL LOAD$INC;
END CHECK$EDIT;
```

```
/* * * * * * MACHINE ACTIONS *  * * * * * */


STOP: PROCEDURE;
    CALL PRINT(.('END OF JOB $'));
    CALL BOOTER;
END STOP;


/* * * * * * * * * * * * * * * * * * *

  THE PROCEDURE BELOW CONTROLS THE EXECUTION OF THE CODE.
  IT DECODES EACH OP-CODE AND PERFORMS THE ACTIONS

* * * * * * * * * * * * * * * * * * * */


EXECUTE: PROCEDURE;
 DO FOREVER;
      DO CASE GET$OP$CODE;

           ;     /* CASE ZERO NOT USED */

 /* 01: ADD */

           CALL ADD;

 /* 02: SUB */

           DO;
                CALL COMPLIMENT(0);
                IF SIGN0(0) THEN SIGN0(0)=NEGITIVE;
                ELSE SIGN0(0)=POSITIVE;
                CALL ADD;
           END;

 /* 03: MUL */

           DO;
                DECLARE I BYTE;
                CALL SET$MULT$DIV;
                DECPT1,DECPT2=DECPT1 + DECPT0;
                CALL ALLIGN;
                CALL MULTIPLY(SHR(R1(I:=9),4));
                DO INDEX=1 TO 9;
                     CALL MULTIPLY(R1(I:=I-1) AND 0FH);
                     CALL MULTIPLY(SHR(R1(I),4));
                END;
           END;

 /* 04: DIV */

           CALL DIVIDE;
```

```
/* 05: NEG */

          BRANCH$FLAG=NOT BRANCH$FLAG;

/* 06: STP */

          CALL STOP;

/* 07: STI */

          CALL STORE$IMMEDIATE;


/* 08: RND */

          DO;
               CALL STORE$IMMEDIATE;
               CALL FILL(.R2,10,0);
               R2(9)=1;
               CALL ADD;
          END;

/* 09: RET */

          DO;
               IF C$ADDR(0)<>0 THEN
               DO;
                    A$CTR=C$ADDR(0);
                    C$ADDR(0)=0;
                    PROGRAM$COUNTER=A$CTR;
               END;
               ELSE CALL INC$PTR(2);
          END;

/* 10: CLS */

          DO;
               CALL SET$I$O;
               IF WRITE$MARK THEN
                   DO;
                     IF NOT SHR(CURRENT$FLAG,2) THEN
                         CALL WRITE$BYTE(TERMINATOR);
                     CALL WRITE$RECORD;
                    END;
               ELSE
                    CALL SET$DMA;
               CALL CLOSE(C$ADDR(0));
               FCB$BYTE$A(FLAG$OFFSET)=0;
               CALL INC$PTR(2);
          END;

/* 11: SER */

          DO;
```

```
            IF OVERFLOW  THEN PROGRAM$COUNTER
               = C$ADDR(0);
            ELSE CALL INC$PTR(2);
         END;
/* 12: BRN */

         PROGRAM$COUNTER=C$ADDR(0);

/* 13: OPN */

        DO;
          CALL OPEN$FILE(1);
          CALL READ$RECORD;
        END;

/* 14: OP1 */

          CALL OPEN$FILE(2);

/* 15: OP2 */

        DO;
          /* 4 IS USED SO EACH TYPE SETS ONLY
             ONE BIT IN CURRENT$FLAG */
          CALL OPEN$FILE(4);
          CALL READ$RECORD;
        END;

/* 16: RGT */

          DO;
               IF NOT SIGN0(2) THEN
                    BRANCH$FLAG=NOT BRANCH$FLAG;
               CALL COND$BRANCH(0);
          END;

/* 17: RLT */

          DO;
               IF SIGN0(2) THEN
                    BRANCH$FLAG=NOT BRANCH$FLAG;
               CALL COND$BRANCH(0);
          END;

/* 18: REQ */

          DO;
               IF R2$ZERO THEN
                    BRANCH$FLAG=NOT BRANCH$FLAG;
               CALL COND$BRANCH(0);
          END;

/* 19: INV */
```

```
          CALL INCR$OR$BRANCH(INVALID);

/* 20: EOR */

    CALL INCR$OR$BRANCH(FCB$BYTE$A(EOF$FLAG$OFFSET));

/* 21: ACC */

          CALL ACCEPT;

/* 22: STD */

          DO;
               C$BYTE(3)=0;
               CALL DISPLAY;
               CALL STOP;
          END;

/* 23: LDI */

          DO;
               C$ADDR(2)=CONVERT$TO$HEX(C$ADDR(0)
                         ,C$BYTE(2))+1;
               CALL INC$PTR(3);
          END;

/* 24: DIS */

          CALL DISPLAY;

/* 25: DEC */

          DO;
               IF C$ADDR(0)<>0 THEN C$ADDR(0)
                  = C$ADDR(0)-1;
               IF C$ADDR(0)=0 THEN
                  PROGRAM$COUNTER = C$ADDR(1);
               ELSE CALL INC$PTR(4);
          END;

/* 26: STO */

          DO;
               CALL STORE$NUMERIC;
               CALL INC$PTR(4);
          END;

/* 27: ST1 */

          DO;
               CALL STORE$NUMERIC;
               CALL SET$ZONE(C$ADDR(0));
          END;
```

```
/* 28: ST2 */

          DO;
               CALL STORE$NUMERIC;
               CALL SET$ZONE(C$ADDR(0)+C$BYTE(2)-1);
          END;

/* 29: ST3 */

          DO;
               CALL CHECK$DECIMAL;
               BASE=C$ADDR(0) + C$BYTE(2);
               CALL STORE$AS$CHAR(C$BYTE(2) - 1);
               CALL SET$SIGN$SEP(C$ADDR(0));
          END;

/* 30: ST4 */

          DO;
               CALL CHECK$DECIMAL;
               BASE=C$ADDR(0) + C$BYTE(2) -1;
               CALL STORE$AS$CHAR(C$BYTE(2)-1);
              CALLSET$SIGN$SEP(C$ADDR(0)+C$BYTE(2)-1);
          END;

/* 31: ST5 */

          DO;
               CALL CHECK$DECIMAL;
               R0(9)=R2(9) OR SIGN0(2);
               CALL MOVE(.R2 + 9 - C$BYTE(2),C$ADDR(0)
                    ,C$BYTE(2));
               CALL INC$PTR(4);
          END;

/* 32: LOD */

          CALL LOAD$NUM$LIT;

/* 33: LD1 */

          CALL LOAD$NUMERIC;

/* 34: LD2 */

         DO;
              HOLD=C$ADDR(0);
              IF CHECK$FOR$SIGN(H$BYTE(0)) THEN
              DO;
                CALL SET$LOAD(POSITIVE);
                CALL LOAD$NUMBERS(C$ADDR(0),C$BYTE(2));
              END;
              ELSE DO;
                   CALL SET$LOAD(NEGITIVE);
```

```
                CALL LOAD$NUMBERS(C$ADDR(0)+1
                     ,C$BYTE(2)-1);
                CALL LOAD$A$CHAR(H$BYTE(0)-ZONE);
            END;
        END;

/* 35: LD3 */

        DO;
            DECLARE I BYTE;
            HOLD=C$ADDR(0);
            IF CHECK$FOR$SIGN(CTR:=H$BYTE(I:=
               C$BYTE(2)-1)) THEN
            DO;
                CALL SET$LOAD(POSITIVE);
                I=I+1;
            END;
            ELSE DO;
                CALL SET$LOAD(NEGITIVE);
                CALL LOAD$A$CHAR(CTR-ZONE);
            END;
            CALL LOAD$NUMBERS(C$ADDR(0),I);
        END;

/* 36: LD4 */

        DO;
            HOLD=C$ADDR(0);
            IF(H$BYTE(0)='+') THEN CALL SET$LOAD(1);
            ELSE CALL SET$LOAD(0);
            CALL LOAD$NUMBERS(C$ADDR(0),C$BYTE(2) -1);
        END;

/* 37: LD5 */

        DO;
            HOLD=C$ADDR(0);
            IF H$BYTE(C$BYTE(2) - 1) = '+' THEN
                CALL SET$LOAD(1);
            ELSE CALL SET$LOAD(0);
            CALL LOAD$NUMBERS(C$ADDR(0),C$BYTE(2)-1);
        END;

/* 38: LD6 */

        DO;
            DECLARE I BYTE;
            HOLD=C$ADDR(0);
            CALL SET$LOAD(H$BYTE(I:=C$BYTE(2)-1));
            BASE=BASE + 9 - I;
            DO CTR = 0 TO I;
                B$BYTE(CTR)=H$BYTE(CTR);
            END;
            B$BYTE(CTR)=B$BYTE(CTR) AND 0F0H;
```

```
            CALL INC$PTR(5);
        END;

/* 39: PER */

        DO;
            BASE=C$ADDR(1)+1;
            B$ADDR(0)=C$ADDR(2);
            PROGRAM$COUNTER=C$ADDR(0);
        END;

/* 40: CNU */

        CALL COMP$NUM$UNSIGNED;

/* 41: CNS */

        CALL COMP$NUM$SIGN;

/* 42: CAL */

        CALL COMP$ALPHA;

/* 43: RWS */

        DO;
            CALL SET$I$O;
            IF NOT SHR(CURRENT$FLAG,2) THEN
              CALL FATAL$ERROR('W6');
            IF NOT FCB$BYTE$A(EOF$FLAG$OFFSET) THEN
                  CALL REWRITE$SEQ(1);
            CALL INC$PTR(6);
        END;

/* 44: DLS */

        DO;
          CALL SET$I$O;
          IF NOT SHR(CURRENT$FLAG,2) THEN
             CALL FATAL$ERROR('W6');
          IF NOT FCB$BYTE$A(EOF$FLAG$OFFSET) THEN
               CALL REWRITE$SEQ(0);
          CALL INC$PTR(6);
        END;

/* 45: RDF */

        DO;
            CALL SET$I$O;
            IF NOT CURRENT$FLAG THEN
               CALL FATAL$ERROR('W5');
            IF NOT FCB$BYTE$A(EOF$FLAG$OFFSET) THEN
                  CALL READ$TO$MEMORY;
            CALL INC$PTR(6);
```

```
          END;

/* 46: WTF */

          DO;
               CALL SET$I$0;
               IF NOT SHR(CURRENT$FLAG,1) THEN
                  CALL FATAL$ERROR('W3');
               CALL WRITE$FROM$MEMORY;
               CALL INC$PTR(6);
          END;

/* 47: RVL */

          CALL READ$VARIABLE;

/* 48: WVL */

          CALL WRITE$VARIABLE;

/* 49: SCR */

          DO;
               SUBSCRIPT(C$BYTE(2))=
                    CONVERT$TO$HEX(C$ADDR(0),C$BYTE(3));
               CALL INC$PTR(4);
          END;

/* 50: SGT */

          CALL STRING$COMPARE(1);

/* 51: SLT */

          CALL STRING$COMPARE(0);

/* 52: SEQ */

          CALL STRING$COMPARE(2);

/* 53: MOV */

          DO;
           CALL MOVE(RES(C$ADDR(1)),RES(C$ADDR(0))
                ,C$ADDR(2));
           IF C$ADDR(3)<>0 THEN CALL
                FILL(RES(C$ADDR(0)) + C$ADDR(2)
                ,C$ADDR(3),' ');
           CALL INC$PTR(8);
          END;

/* 54: RES */

          DO;
```

```
                CALL SET$I$0;
                IF  SHR(CURRENT$FLAG,1) THEN
                    CALL FATAL$ERROR('W5');
                IF NOT FCB$BYTE$A(EOF$FLAG$OFFSET) THEN
                  DO;
                       CALL SET$RELATIVE$KEY;
                       CALL READ$TO$MEMORY;
                  END;
                CALL INC$PTR(9);
           END;

/* 55: WRS */

        DO;
        CALL SET$I$0;
        IF NOT SHR(CURRENT$FLAG,1) THEN
        CALL FATAL$ERROR('W1');
        CALL CHECK$DIFFERENCE;
        CALL SET$RELATIVE$KEY;
        CALL WRITE$FROM$MEMORY;
        CALL INC$PTR(9);
        END;
/* 56: RRR */

           DO;
                CALL SET$I$0;
                IF SHR(CURRENT$FLAG,1) THEN
                   CALL FATAL$ERROR('W5');
                CALL SET$RANDOM$POINTER;
                IF NOT INVALID THEN CALL READ$TO$MEMORY;
                IF VALID THEN
                FCB$BYTE$A(EOF$FLAG$OFFSET)=FALSE;
                CALL INC$PTR(9);
           END;
/* 57: WRR */

        DO;
           DECLARE DIFFERENCE ADDRESS;
           CALL SET$I$0;
           IF  SHR(CURRENT$FLAG,1) THEN
          DO;
             CALL CHECK$DIFFERENCE;
             CALL SET$RELATIVE$KEY;
             CALL WRITE$FROM$MEMORY;
          END;
        ELSE
          DO;
            IF SHR(CURRENT$FLAG,2) THEN
               DO;
                  CALL SET$RANDOM$POINTER;
                  IF NOT INVALID THEN
                    DO;
                      IF (.BUFF$PTR+1)=HIGH$VALUE THEN
                         DO;
```

```
                        REWRITE$FLAG=TRUE;
                        CALL WRITE$FROM$MEMORY;
                        REWRITE$FLAG=FALSE;
                     END;
                  ELSE
                     CALL PRINT$ERROR('W4');
               END;
            ELSE
                 CALL FATAL$ERROR('W3');
       END;
      END;
     CALL INC$PTR(9);
     END;

/* 58: RWR */

     DO;
        CALL SET$I$0;
        IF NOT SHR(CURRENT$FLAG,2) THEN
           CALL FATAL$ERROR('W6');
        REWRITE$FLAG=TRUE;
        CALL BACK$ONE$RECORD;
        IF NOT INVALID THEN CALL WRITE$FROM$MEMORY;
        REWRITE$FLAG=FALSE;
        CALL INC$PTR(9);
     END;

/* 59: DLR */

       DO;
            CALL SET$I$0;
            IF NOT SHR(CURRENT$FLAG,2) THEN
               CALL FATAL$ERROR('W6');
            CALL SET$RANDOM$POINTER;
            REWRITE$FLAG=TRUE;
            IF NOT INVALID THEN
               CALL WRITE$EMPTY$RECORD;
            REWRITE$FLAG=FALSE;
            CALL INC$PTR(9);
       END;

/* 60: MED */

       DO;
            CALL MOVE(C$ADDR(3),RES(C$ADDR(0))
                 ,C$ADDR(4));
            BASE=RES(C$ADDR(1));
            HOLD=RES(C$ADDR(0));
            CTR=0;
            CTR1=0;
            DO WHILE (CTR<C$ADDR(2))AND(CTR
                      < C$ADDR(4));
                 CALL CHECK$EDIT(H$BYTE(0));
            END;
```

```
                IF CTR < C$ADDR(4) THEN
                     CALL FILL(HOLD,C$ADDR(4)-CTR,' ');
                CALL INC$PTR(10);
           END;

 /* 61: MNE */

 ;      /*   NULL CASE   */


 /* 62: GDP */

           DO;
             DECLARE OFFSET BYTE;
             OFFSET=CONVERT$TO$HEX(C$ADDR(1),C$BYTE(1)-1);
             IF OFFSET > C$BYTE(0) + 1 THEN
             DO;
                  CALL PRINT$ERROR('GD');
                  CALL INC$PTR(SHL(C$BYTE(0),1) + 5);
             END;
             ELSE PROGRAM$COUNTER=C$ADDR(OFFSET + 2);
           END;

      END; /* END OF CASE STATEMENT */
 END; /* END OF DO FOREVER */
END EXECUTE;

/* * * * * * PROGRAM EXECUTION STARTS HERE * * * */

BASE=CODE$START;
PROGRAM$COUNTER=B$ADDR(0);
CALL EXECUTE;
END;
```

```
READER:
       DO;

    /* COBOL COMPILER - PART 2 READER */

    /* THIS PROGRAM IS LOADED IN WITH THE PART 1 PROGRAM
    AND IS CALLED WHEN PART 1 IS FINISHED.  THIS PROGRAM
    OPENS THE PART2.COM FILE THAT CONTAINS THE CODE FOR
    PART 2 OF THE COMPILER, AND READS IT INTO CORE. AT
    THE END OF THE READ OPERATION, CONTROL IS PASSED TO
    THE SECOND PART PROGRAM.                          */


/*          3100H:    LOAD POINT */

DECLARE

START    LITERALLY '100H',
              /* STARTING LOCATION FOR PART2 */
ADR      ADDRESS INITIAL(START),
FCB (33) BYTE
INITIAL(0,'PART2   COM',0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0)
INITIAL(0,'PART2   COM',0,0,0,0,0,0,0,0,0,0,0,0,0
          ,0,0,0,0,0,0,0,0),
I        ADDRESS;

MON1: PROCEDURE(F,A) EXTERNAL;
    DECLARE F BYTE, A ADDRESS;
END MON1;

MON2: PROCEDURE(F,A)BYTE EXTERNAL;
    DECLARE F BYTE, A ADDRESS;
END MON2;

BOOT: PROCEDURE EXTERNAL;
    END;

OPEN: PROCEDURE (FCB) BYTE;
      DECLARE FCB ADDRESS;
      RETURN MON2 (15, FCB);
      END;

READ: PROCEDURE (ADDR) BYTE;
      DECLARE ADDR ADDRESS;
      CALL MON1 (26, ADDR);  /* SET DMA ADDRESS */
      RETURN MON2 (20, .FCB);  /*  READ, AND RETURN
                                   ERROR CODE */
      END;

ERROR: PROCEDURE(CODE);
    DECLARE CODE ADDRESS;
    CALL MON1(2,(HIGH(CODE)));
    CALL MON1(2,(LOW(CODE)));
    CALL TIME(10);
```

```
    CALL BOOT;
END ERROR;
    CALL MON1 (26, 0100H);

    /* OPEN PASS2.COM */
IF OPEN(.FCB)=255 THEN CALL ERROR('02');
    /* READ IN FILE */

      I = 0100H;   /* INITIAL ADDRESS  */
      DO WHILE READ(I) = 0;      /* READ 1 SECTOR */
          I = I + 0080H;        /* BUMP DMA ADDRESS  */
      END;

     CALL MON1 (26, 0080H);   /*  RESET DMA ADDRESS  */
      CALL ADR;

  END;
```

```
BUILD:
DO;
/* NORMALLY ORG'ED AT 100H */

/* THIS PROGRAM TAKES THE CODE OUTPUT FROM THE COBOL
COMPILER AND BUILDS THE ENVIRONMENT FOR THE COBOL
INTERPRETER */


DECLARE

LIT             LITERALLY       'LITERALLY',
BOOT            LIT             '0',
BDOS            LIT             '5',
TRUE            LIT             '1',
FALSE           LIT             '0',
FOREVER         LIT             'WHILE TRUE',
FCB             ADDRESS         INITIAL (5CH),
FCB$BYTE        BASED     FCB   BYTE,
FCB$BYTE$A      BASED FCB (33)  BYTE,
I               BYTE,
ADDR            ADDRESS         INITIAL (100H),
CHAR            BASED     ADDR  BYTE,
BUFF$END        LIT             '100H',
INTERP$FCB      (33)      BYTE
                          INITIAL(0,'CINTERP COM',0,0,0,0),
CODE$NOT$SET    BYTE      INITIAL (TRUE),
READER$LOCATION     LIT         '1C80H',
INTERP$ADDRESS      ADDRESS     INITIAL(2000H),
INTERP$CONTENT      BASED       INTERP$ADDRESS ADDRESS,
I$BYTE              BASED       INTERP$ADDRESS (2) BYTE,
CODE$CTR            ADDRESS,
C$BYTE              BASED       CODE$CTR  BYTE,
BASE                ADDRESS,
B$ADDR              BASED        BASE     ADDRESS,
B$BYTE              BASED        BASE (4) BYTE;

MON1: PROCEDURE (F,A) EXTERNAL;
    DECLARE F BYTE, A ADDRESS;
END MON1;


MON2: PROCEDURE (F,A) BYTE EXTERNAL;
    DECLARE F BYTE, A ADDRESS;
END MON2;


PRINT$CHAR: PROCEDURE(CHAR);
    DECLARE CHAR BYTE;
    CALL MON1(2,CHAR);
END PRINT$CHAR;


CRLF: PROCEDURE;
```

```
    CALL PRINT$CHAR(13);
    CALL PRINT$CHAR(10);
END CRLF;


PRINT: PROCEDURE(A);
    DECLARE A ADDRESS;
    CALL CRLF;
    CALL MON1(9,A);
END PRINT;


OPEN: PROCEDURE (A) BYTE;
    DECLARE A ADDRESS;
    RETURN MON2(15,A);
END OPEN;

REBOOT: PROCEDURE;
    ADDR = BOOT; CALL ADDR;
    END REBOOT;


MOVE: PROCEDURE(FROM, DEST, COUNT);
    DECLARE (FROM, DEST, COUNT) ADDRESS,
        (F BASED FROM, D BASED DEST) BYTE;
    DO WHILE(COUNT:=COUNT-1)<>0FFFFH;
        D=F;
        FROM=FROM+1;
        DEST=DEST+1;
    END;
END MOVE;


GET$CHAR: PROCEDURE BYTE;
    IF (ADDR:=ADDR + 1)>=BUFF$END THEN
    DO;
        IF MON2(20,FCB)<>0 THEN
        DO;
            CALL PRINT(.('END OF INPUT   $'));
            CALL REBOOT;
        END;
        ADDR=80H;
    END;
    RETURN CHAR;
END GET$CHAR;


NEXT$CHAR: PROCEDURE;
    CHAR=GET$CHAR;
END NEXT$CHAR;


STORE: PROCEDURE(COUNT);
    DECLARE COUNT BYTE;
```

```
    IF CODE$NOT$SET THEN
    DO;
         CALL PRINT(.('CODE ERROR$'));
         CALL NEXT$CHAR;
         RETURN;
    END;
    DO I=1 TO COUNT;
         C$BYTE=CHAR;
         CALL NEXT$CHAR;
         CODE$CTR=CODE$CTR+1;
    END;
END STORE;


BACK$STUFF: PROCEDURE;
    DECLARE (HOLD,STUFF) ADDRESS;
    BASE=.HOLD;
    DO I=0 TO 3;
         B$BYTE(I)=GET$CHAR;
    END;
    DO FOREVER;
         BASE=HOLD;
         HOLD=B$ADDR;
         B$ADDR=STUFF;
         IF HOLD=0 THEN
         DO;
              CALL NEXT$CHAR;
              RETURN;
         END;
    END;
END BACK$STUFF;


START$CODE: PROCEDURE;
    CODE$NOT$SET=FALSE;
    I$BYTE(0)=GET$CHAR;
    I$BYTE(1)=GET$CHAR;
    CODE$CTR=INTERP$CONTENT;
    CALL NEXT$CHAR;
END START$CODE;


GO$DEPENDING: PROCEDURE;
    CALL STORE(1);
    CALL STORE(SHL(CHAR,1) + 4);
END GO$DEPENDING;


INITIALIZE: PROCEDURE;
    DECLARE (COUNT,WHERE,HOW$MANY) ADDRESS;
    BASE=.WHERE;
    DO I=0 TO 3;
         B$BYTE(I)=GET$CHAR;
    END;
```

```
    BASE=WHERE - 1;
    DO COUNT = 1 TO HOW$MANY;
         B$BYTE(COUNT)=GET$CHAR;
    END;
    CALL NEXT$CHAR;
END INITIALIZE;


BUILD: PROCEDURE;
    DECLARE
    F2   LIT  '8',
    F3   LIT  '9',
    F4   LIT  '21',
    F5   LIT  '24',
    F6   LIT  '32',
    F7   LIT  '39',
    F9   LIT  '49',
    F10  LIT  '54',
    F11  LIT  '60',
    F13  LIT  '61',
    GDP  LIT  '62',
    INT  LIT  '63',
    BST  LIT  '64',
    TER  LIT  '65',
    STP  LIT  '06',
    SCD  LIT  '66';

    DO FOREVER;
         IF CHAR < F2 THEN CALL STORE(1);
         ELSE IF CHAR < F3 THEN CALL STORE(2);
         ELSE IF CHAR < F4 THEN CALL STORE(3);
         ELSE IF CHAR < F5 THEN CALL STORE(4);
         ELSE IF CHAR < F6 THEN CALL STORE(5);
         ELSE IF CHAR < F7 THEN CALL STORE(6);
         ELSE IF CHAR < F9 THEN CALL STORE(7);
         ELSE IF CHAR < F10 THEN CALL STORE(9);
         ELSE IF CHAR < F11 THEN CALL STORE(10);
         ELSE IF CHAR < F13 THEN CALL STORE(11);
         ELSE IF CHAR < GDP THEN CALL STORE(13);
         ELSE IF CHAR = GDP THEN CALL GO$DEPENDING;
         ELSE IF CHAR = BST THEN CALL BACK$STUFF;
         ELSE IF CHAR = INT THEN CALL INITIALIZE;
         ELSE IF CHAR = TER THEN
         DO;
              C$BYTE = STP;
              CALL PRINT(.('LOAD FINISHED$'));
              RETURN;
         END;
         ELSE IF CHAR = SCD THEN CALL START$CODE;
         ELSE DO;
              IF CHAR <> 0FFH THEN
                        CALL PRINT(.('LOAD ERROR$'));
              CALL NEXT$CHAR;
         END;
```

```
    END;
END BUILD;


    /* PROGRAM EXECUTION STARTS HERE */

FCB$BYTE$A(32),FCB$BYTE=0;
CALL MOVE(.('CIN',0,0,0,0),FCB + 9,7);
IF OPEN(FCB)=255 THEN
DO;
    CALL PRINT(.('FILE NOT FOUND   $'));
    CALL REBOOT;
END;
CALL NEXT$CHAR;
CALL BUILD;
CALL MOVE(.INTERP$FCB,FCB,33);
FCB$BYTE$A(32) = 0;
IF OPEN(FCB)=255 THEN
DO;
    CALL PRINT(.('INTERPRETER NOT FOUND   $'));
    CALL REBOOT;
END;
CALL MOVE(READER$LOCATION, 80H, 80H);
ADDR = 80H; CALL ADDR; /* BRANCH TO 80H */
END;
```

```
INTRDR:        /*  NAME OF MODULE  */
       DO;

    /* COBOL COMPILER - INTERP READER */

    /* THIS PROGRAM IS CALLED BY THE BUILD PROGRAM AFTER
    CINTERP.COM HAS BEEN OPENED, AND READS THE CODE INTO
    MEMORY */


    /*   80H  -  LOAD POINT     */

DECLARE

START    LITERALLY '100H',  /* STARTING LOCATION FOR
                               PART2 */
INTERP   ADDRESS  INITIAL(START),
I   ADDRESS  INITIAL (0080H);

MONA: PROCEDURE(F,A);
    DECLARE F BYTE, A ADDRESS;
  L:GO TO L;   /*   PATCH TO ->  'JMP  BDOS'   */
END MONA;

MONB: PROCEDURE(F,A)BYTE;
    DECLARE F BYTE, A ADDRESS;
  L:GO TO L;   /*   PATCH TO ->  "JMP  BDOS"   */
    RETURN 0;      /*   ZAP -> "NO-OP"    */
END MONB;

    DO WHILE 1;
      CALL MONA (26, (I:=I+0080H)); /* SET DMA ADDRESS */
      IF   MONB (20, 5CH) <> 0 THEN
                                    CALL INTERP;
    END;
END;
```

```
DECODE:  DO;

/* THIS PROGRAM TAKES THE CODE OUTPUT FROM THE COBOL
COMPILER AND CONVERTS IT INTO A READABLE OUTPUT TO
FACILITATE DEBUGGING */

/* * * 100H:       LOAD POINT */

DECLARE

LIT             LITERALLY       'LITERALLY',
BOOT            LIT             '0',
BDOS            LIT             '5',
FCB             ADDRESS         INITIAL (5CH),
FCB$BYTE        BASED     FCB (1)  BYTE,
I               BYTE,
ADDR            ADDRESS         INITIAL (100H),
BYTE$COUNT      ADDRESS         INITIAL (0),
BYTE$LOW        BYTE,
BYTE$HI         BYTE,
CHAR            BASED      ADDR BYTE,
C$ADDR          BASED ADDR           ADDRESS,
BUFF$END        LIT             '0FFH',
FILE$TYPE (*) BYTE     DATA ('C','I','N');

MON1: PROCEDURE (F,A);
    DECLARE F BYTE, A ADDRESS;
    L:  GO TO L;   /*    PATCH TO JMP 5   */
END MON1;


MON2: PROCEDURE (F,A) BYTE;
    DECLARE F BYTE, A ADDRESS;
  L:GO TO L;      /* * *   PATCH TO    " JMP 5 "   * * */
    RETURN 0;
END MON2;


PRINT$CHAR: PROCEDURE(CHAR);
    DECLARE CHAR BYTE;
    CALL MON1(2,CHAR);
END PRINT$CHAR;


CRLF: PROCEDURE;
    CALL PRINT$CHAR(13);
    CALL PRINT$CHAR(10);
END CRLF;


P: PROCEDURE(ADD1);
    DECLARE ADD1 ADDRESS, C BASED ADD1 (1) BYTE;
    CALL CRLF;
    DO I=0 TO 2;
```

```
          CALL PPINT$CHAR(C(I));
     END;
     CALL PRINT$CHAR(' ');
END P;

GET$CHAR: PROCEDURE BYTE;
     IF (ADDR:=ADDR + 1)>BUFF$END THEN
     DO;
          IF MON2(20,FCB)<>0 THEN
          DO;
               CALL P(.('END'));
               CALL TIME(10);
             L:  GO TO L;  /*  PATCH TO  "JMP 0000"  */
          END;
          ADDR=80H;
     END;
     RETURN CHAR;
END GET$CHAR;


D$CHAR: PROCEDURE (OUTPUT$BYTE);
     DECLARE OUTPUT$BYTE BYTE;
     IF OUTPUT$BYTE<10 THEN
     CALL PRINT$CHAR(OUTPUT$BYTE + 30H);
     ELSE CALL PRINT$CHAR(OUTPUT$BYTE + 37H);
END D$CHAR;


D: PROCEDURE (COUNT);
     DECLARE(COUNT,J) ADDRESS;
     DO J=1 TO COUNT;
          CALL D$CHAR(SHR(GET$CHAR,4));
          CALL D$CHAR(CHAR AND 0FH);
          CALL PRINT$CHAR(' ');
     END;
END D;


PRINT$REST: PROCEDURE;
     DECLARE
     F2   LIT  '8',
     F3   LIT  '9',
     F4   LIT  '21',
     F5   LIT  '23',
     F6   LIT  '32',
     F7   LIT  '39',
     F9   LIT  '49',
     F10  LIT  '54',
     F11  LIT  '60',
     F13  LIT  '61',
     GDP  LIT  '62',
     INT  LIT  '63',
     BST  LIT  '64',
     TER  LIT  '65',
```

```
    SCD  LIT  '66';

    IF CHAR < F2 THEN RETURN;
    IF CHAR < F3 THEN DO; CALL D(1); RETURN; END;
    IF CHAR < F4 THEN DO; CALL D(2); RETURN; END;
    IF CHAR < F5 THEN DO; CALL D(3); RETURN; END;
    IF CHAR < F6 THEN DO; CALL D(4); RETURN; END;
    IF CHAR < F7 THEN DO; CALL D(5); RETURN; END;
    IF CHAR < F9 THEN DO; CALL D(6); RETURN; END;
    IF CHAR < F10 THEN DO; CALL D(8); RETURN; END;
    IF CHAR < F11 THEN DO; CALL D(9); RETURN; END;
    IF CHAR < F13 THEN DO; CALL D(10); RETURN; END;
    IF CHAR < GDP THEN DO; CALL D(12); RETURN; END;
    IF CHAR=GDP THEN DO;
 CALL D(1); CALL D(SHL(CHAR,1)+5); RETURN; END;
    IF CHAR = INT THEN
    DO;
       BYTE$COUNT = 0;
       CALL D(3);
       BYTE$LOW = CHAR;
       CALL D(1);
       BYTE$HI = CHAR;
       BYTE$COUNT = BYTE$HI;
       BYTE$COUNT = SHL(BYTE$COUNT,8) + BYTE$LOW;
       CALL D(BYTE$COUNT);
       RETURN;
    END;
    IF CHAR=BST THEN DO; CALL D(4); RETURN; END;
    IF CHAR=TER THEN DO; CALL P(.('END'));
   L: GO TO L;   /* PATCH TO  'JMP  0'  * * */ END;
    IF CHAR=SCD THEN DO; CALL D(2); RETURN; END;
    IF CHAR <> 0FFH THEN CALL P(.('XXX'));
END PRINT$REST;


    /* PROGRAM EXECUTION STARTS HERE */

FCB$BYTE(32), FCB$BYTE(0) = 0;
DO I=0 TO 2;
    FCB$BYTE(I+9)=FILE$TYPE(I);
END;

IF MON2(15,FCB)=255 THEN DO; CALL P(.('ZZZ'));
                         L: GO TO L; END;
               /* * * * PATCH TO  "JMP BOOT" * * * */

DO WHILE 1;
    IF GET$CHAR <= 66 THEN DO CASE CHAR;
    ;    /* CASE 0 NOT USED */
        CALL P(.('ADD'));
        CALL P(.('SUB'));
        CALL P(.('MUL'));
        CALL P(.('DIV'));
        CALL P(.('NEG'));
```

```
CALL P(.('STP'));
CALL P(.('STI'));
CALL P(.('RND'));
CALL P(.('RET'));
CALL P(.('CLS'));
CALL P(.('SER'));
CALL P(.('BRN'));
CALL P(.('OPN'));
CALL P(.('OP1'));
CALL P(.('OP2'));
CALL P(.('RGT'));
CALL P(.('RLT'));
CALL P(.('REQ'));
CALL P(.('INV'));
CALL P(.('EOR'));
CALL P(.('ACC'));
CALL P(.('STD'));
CALL P(.('LDI'));
CALL P(.('DIS'));
CALL P(.('DEC'));
CALL P(.('STO'));
CALL P(.('ST1'));
CALL P(.('ST2'));
CALL P(.('ST3'));
CALL P(.('ST4'));
CALL P(.('ST5'));
CALL P(.('LOD'));
CALL P(.('LD1'));
CALL P(.('LD2'));
CALL P(.('LD3'));
CALL P(.('LD4'));
CALL P(.('LD4'));
CALL P(.('LD6'));
CALL P(.('PER'));
CALL P(.('CNU'));
CALL P(.('CNS'));
CALL P(.('CAL'));
CALL P(.('RWS'));
CALL P(.('DLS'));
CALL P(.('RDF'));
CALL P(.('WTF'));
CALL P(.('RVL'));
CALL P(.('WVL'));
CALL P(.('SCR'));
CALL P(.('SGT'));
CALL P(.('SLT'));
CALL P(.('SEQ'));
CALL P(.('MOT'));
CALL P(.('RRS'));
CALL P(.('WRS'));
CALL P(.('RRR'));
CALL P(.('WRR'));
CALL P(.('RWR'));
CALL P(.('DLR'));
```

```
        CALL P(.('MED'));
        CALL P(.('MNE'));
        CALL P(.('GDP'));
        CALL P(.('INT'));
        CALL P(.('BST'));
        CALL P(.('TER'));
        CALL P(.('SCD'));
    END; /* OF CASE STATEMENT */
    CALL PRINT$REST;
END; /* END OF DO WHILE */
END;
```

LIST OF REFERENCES

1. Aho, A. V. and Ullman, J. D., Principles of Compiler Design, Addison-Wesley, 1977.

2. Craig, A. S., MICRO-COBOL An Implementation of Navy Standard HYPO-COBOL for a Micro-processor based Computer System, Master's Thesis, Naval Postgraduate School, Monterey, California, 1977.

3. Department of the Navy Automated Data Processing Equipment Selection Office, HYPO-COBOL Validation System (HCCVS), April 1975.

4. Digital Research, An Introduction to CP/M Features and Facilites, 1976.

5. Digital Research, CP/M Interface Guide, 1976.

6. Digital Research, Symbolic Instruction Debugger User's Guide, 1978.

7. Digital Research, CP/M Dynamic Debugging Tool (DDT) User's Guide, 1976.

8. Intel Corporation, PL/M-80 Programming Manual, 1976.

9. Intel Corporation, ISIS-II User's Guide, 1976.

10. Intel Corporation, ISIS-II PL/M-80 Compiler Operator's Manual, 1976.

11. Intel Corporation, 8008 and 8080 PL/M Programming Manual, 1975.

12. Kiefer, R. and Perry, C., MICRO-COBOL Validation System, paper presented as a term project for CS3020, Naval Postgraduate School, Monterey, California, Fall Term, 1978.

13. McCracken, D. M., A Simplified Guide to Structured COBOL Programming, Wiley, 1976.

14. Myers, G. J., Software Reliability--Principles and Practices, Wiley, 1976.

15. Mylet, P. R., MICRO-COBOL A Subset of Navy Standard HYPO-COBOL for Micro-computers, Master's Thesis, Naval Postgraduate School, Monterey, California, 1978.

16. Polya, G., How to Solve It, Princeton N.J.: Princeton University Press, 1971.

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

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	3
4.	Asst. Professor L. A. Cox Jr., Code 52Cl Department of Computer Science Naval Postgraduate School Monterey, California 93940	1
5.	Lt. M. S. Moranville, Code 52Mi Department of Computer Science Naval Postgraduate School Monterey, California 93940	2
6.	ADPE Selection Office Department of the Navy Washington, D. C. 20376	1
7.	Capt. J. Farlee Jr. 393-B Ricketts Rd. Monterey, California 93940	2
8.	Capt. M. L. Rice 12477 Fallingleaf St. Garden Grove, California 92640	2
9.	Rayner K. Rosich U. S. Department of Commerce National Telecommunications and Information Administration Boulder, Colorado 80303	1

Thesis
F2255
c.1

Farlee

183371

NPS MICRO-COBOL: an implementation of Navy standard HYPO-COBOL for a microprocessor-based computer system.