AD A047266

(2)

NAVAL POSTGRADUATE SCHOOL

Monterey, California

DDC
RECEIVED
DEC 5 1977
E

THESIS

ALGOL-M
An Implementation of a
High-level Block Structured Language
for a Microprocessor-based Computer System

by

John P. Flynn
and
Mark S. Moranville

September 1977

Thesis Advisor: G. A. Kildall

Approved for public release; distribution unlimited.

AD No.
DDC FILE COPY

COPY AVAILABLE TO DDC DOES NOT
PERMIT FULLY LEGIBLE PRODUCTION

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)

ALGOL-M
An Implementation of a High-level Block Structured Language for a Microprocessor-based Computer System

5. TYPE OF REPORT & PERIOD COVERED

Master's Thesis; September 1977

6. PERFORMING ORG. REPORT NUMBER

7. AUTHOR(s)

John P. Flynn
Mark S. Moranville

8. CONTRACT OR GRANT NUMBER(s)

9. PERFORMING ORGANIZATION NAME AND ADDRESS

Naval Postgraduate School
Monterey, California 93940

10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS

11. CONTROLLING OFFICE NAME AND ADDRESS

Naval Postgraduate School
Monterey, California 93940

12. REPORT DATE

Sep[redacted]977

13. NUMBER OF PAGES

188p.

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

Naval Postgraduate School
Monterey, California 93940

15. SECURITY CLASS. (of this report)

Unclassified

15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited

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

18. SUPPLEMENTARY NOTES

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

Microcomputer
Interpreter
ALGOL-M
Compiler

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

The design and implementation of the ALGOL-M programming language for use on a microprocessor-based system is described. The implementation is comprised of two subsystems, a compiler

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE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

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

251450

which generates code for a hypothetical zero-address machine and a run-time monitor which executes this code. The system was implemented in PL/M to run on an 8080 microcomputer in a diskette-based environment with at least 20K bytes of user storage.

BY
DISTRIBUTION/AVAILABILITY CODES
SPECIAL
A 23

Approved for public release; distribution unlimited

ALGOL-M
An Implementation of a High-level Block Structured Language for a Microprocessor-based Computer System

by

John P. Flynn
Lieutenant Commander, United States Navy
B.S., United States Naval Academy, 1965

and

Mark S. Moranville
Lieutenant, United States Navy
B.S., Oregon State University, 1972

Submitted in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
SEPTEMBER, 1977

Authors: John P. Flynn

Mark Moranville

Approved by: Gary A. Kildall
Thesis Advisor

Stephen T. Tobolt
Second Reader

Chairman, Department of Computer Science

D. Schrady
Dean of Information and Policy Sciences

ABSTRACT

The design and implementation of the ALGUL-M programming language for use on a microprocessor-based system is described. The implementation is comprised of two subsystems, a compiler which generates code for a hypothetical zero-address machine and a run-time monitor which executes this code. The system was implemented in PL/M to run on an 8080 microcomputer in a diskette-based environment with at least 20K bytes of user storage.

TABLE OF CONTENTS

I. INTRODUCTION..... 8

A. HISTORY OF ALGOL..... 8

B. MICROCOMPUTER SOFTWARE..... 8

C. OBJECTIVES OF ALGOL-M..... 9

II. ALGOL-M LANGUAGE DESCRIPTION..... 11

A. FEATURES OF THE ALGOL-M LANGUAGE..... 11

1. Type Declarations..... 11

2. Arithmetic Processing..... 12

3. Control Structures..... 12

4. Input/Output..... 13

5. Disk Access..... 14

III. IMPLEMENTATION..... 16

A. COMPILER IMPLEMENTATION..... 16

1. Compiler Organization..... 16

2. Scanner..... 16

3. Symbol Table..... 17

4. Parser..... 21

5. Code Generation..... 22

B. INTERPRETER IMPLEMENTATION..... 22

1. Building the ALGOL-M Pseudo Machine..... 22

2. Overview of the Interpreter..... 23

3. Allocation of Storage Space..... 26

a. Integers..... 26

b. Decimals..... 28

c. Strings..... 31

d. Arrays..... 34

4. Storage and Retrieval of Variables..... 36

5. Arithmetic Operations..... 37

6. String Operations..... 38

7. Subroutines..... 39

a. Invocation..... 39

b. Storage Allocation..... 39

c. Parameter Mapping..... 41

d. Function Return Value..... 41

e. External Functions..... 44

f. Forward References..... 44

g. Built-in Functions..... 44

8. Input-Output..... 45

9. ALGOL-M Pseudo Operators..... 46

a. Description of Interpreter Variables... 46

b. Literal Data References..... 47

c. Allocation Operators..... 48

d. Arithmetic Operators..... 49

e. String Operators..... 52

f. Stack Operators..... 53

g. Array Operators..... 54

h. Program Control Operators..... 55

i. Store Operators..... 55

j. Input/Output Operators..... 56

k. Subroutine Operators..... 59

IV. CONCLUSIONS..... 61

V. RECOMMENDATIONS..... 63

APPENDIX A - BENCHMARK PROGRAMS..... 64

APPENDIX B - COMPILER ERROR MESSAGES..... 67

APPENDIX C - INTERPRETER ERROR MESSAGES..... 69

APPENDIX D - ALGOL-M LANGUAGE MANUAL..... 70

APPENDIX E - ALGOL-M LANGUAGE STRUCTURE.....108

ALGOL-M PROGRAM LISTINGS.....120

LIST OF REFERENCES.....186

INITIAL DISTRIBUTION LIST.....188

I. INTRODUCTION

A. HISTORY OF ALGOL

The definition of the algorithmic language (ALGOL-60) was the result of the work of a committee of distinguished computer scientists and was originally published in 1960 [11]. The purpose of the developers of ALGOL-60 was the establishment of a universal computer language specifically designed to allow for the logical and efficient program representation of algorithms. Additional versions and extensions of ALGOL-60 such as ALGOL-68 [15] and ALGOL-W [16] have been developed and have found acceptance primarily in the academic communities and in Europe. The language ALGOL-E [10] is also based on ALGOL-60 and was developed as part of a complete system designed for teaching programming language concepts.

B. MICROCOMPUTER SOFTWARE

The rapid development of microcomputer hardware since 1975 has generally resulted in a considerable lag in the corresponding development of compatible software, particularly that of high level languages. The Intel 8080 microprocessor is one of the few microprocessors which has endured long enough to permit software development to advance beyond the assembly language level. High level

languages which have been developed for 8080 based systems by students at the Naval Postgraduate School include a macro assembler (ML-80) [12], a BASIC compiler/interpreter (BASIC-E) [6], and a COBOL compiler/interpreter (MICRO-COBOL) [2]. The majority of high level languages currently available for microcomputer based systems are extensions of the original Dartmouth BASIC and, although they allow for a reasonable level of programming sophistication, they are encumbered by the inherent limitations of the BASIC language constructs.

C. OBJECTIVES OF ALGOL-M

The major objective of this project was to develop a dynamic, block-structured, recursive high level language which would provide adequate programming power and flexibility for applications programming using microcomputer based systems. ALGOL constructs were chosen because of their simplicity and power and because it was possible to write the grammar in LALR(1) form for use with available compiler-compiler generated parse tables [14]. ALGOL-M was developed to run on 8080 based microcomputer systems because of the availability of a high level system development language (PL/M) [8] which produces 8080 object code and which could be run on the Naval Postgraduate School's IBM 360. The availability of an 8080 based disk operating system (CP/M) [13] simulator on the IBM 360 was also a strong factor in the choice of 8080 microprocessor and CP/M

operating system.

II. ALGOL-M LANGUAGE DESCRIPTION

A. FEATURES OF THE ALGOL-M LANGUAGE

Although ALGOL-M was modeled after ALGOL-60, no attempt was made to make it a formal subset of ALGOL-60. This was done intentionally in order to provide a language which would be best suited to the needs of applications programmers using microcomputer systems. However, the basic structure of ALGOL-M is similar enough to ALGOL-60 to allow simple conversion of programs from one language to the other. This was considered particularly important in view of the fact that the standard publication language is ALGOL-60. Therefore, there exists a large source of applications programs and library procedures which can be simply converted to execute under ALGOL-M.

1. Type Declarations

ALGOL-M supports three types of variables: integers, decimals, and strings. Integers may be any value between -16,384 and +16,384. Decimals may be declared with up to 18 digits of precision and strings may be declared as long as 255 characters. The default precision for decimals is ten digits and the default length for strings is ten characters. Decimal and string variable lengths may be integer variables which can be assigned actual values at run-time.

Another form of declaration in ALGOL-M is the array

declaration. Arrays may have up to 255 dimensions with each dimension ranging from -16,384 to +16,384. The maximum 8080 microprocesor address space of 64k bytes limits practical array sizes to something smaller than the maximum. Dimension bounds may be integer variables with the actual values assigned at run-time. Arrays may be of type integer, decimal or string.

2. Arithmetic Processing

Integer and binary coded decimal arithmetic are supported under ALGOL-M. Integers may be used in decimal expressions and will be converted to decimals at run-time. The integer and decimal comparisons of less-than (<), greater-than (>), equal-to (=), not-equal-to (<>), less-than-or-equal-to (<=), and greater-than-or-equal-to (>=) are provided. Additionally, the logical operators AND, OR and NOT are available.

3. Control Structures

ALGOL-M control structures consist of BEGIN, END, FOR, IF THEN, IF THEN ELSE, WHILE, CASE and GOTO constructs. Function and procedure calls are also used as control structures. ALGOL-M is a block stuctured language with a block normally bracketed by a BEGIN and an END. Blocks may be nested within other blocks to nine levels. Variables which are declared within a block can only be referenced within that block or a block nested within that block. Once program control proceeds outside of a block in which a variable has been declared, the variable may not be

referenced and, in fact, run-time storage space for that variable no longer exists.

Functions, when called, return an integer, decimal or string value depending on the type of the function. Procedures do not return a value when called. Both functions and procedures may have zero or more parameters which are call by value and both may be called recursively. Additionally, functions and procedures may be referenced before they are declared.

4. Input/Output

The ALGOL-M WRITE statement causes output to the console on a new line. The desired output is specified in a write list which is enclosed in parentheses. String constants may be used in a write list and are characterized by being enclosed in quotation marks. Any combination of integer, decimal and string variables or expressions may also be used in a write list. A WRITEON statement is also available which is essentially the same as the WRITE statement except that output continues on the same line as the output from a previous WRITE or WRITEON statement. When a total of 80 characters have been written to the console, a new line is started automatically. A TAB option may also be used in the write list which causes the following item in the write list to be spaced to the right by a specifed amount.

Console input is accomplished by the READ statement followed by a read list of any combination of integer,

decimal and string variables enclosed in parentheses. If embedded blanks are desired in the input for a string variable, the console input must be enclosed in quotation marks. A READ statement will result in a halt in program execution at run-time until the input values are typed at the console and a carriage return is sent. If the values typed at the console match the read list in number and type, program execution continues. If an error as to number or type of variables from the console occurs, program execution is again halted until values are re-entered on the console.

5. Disk Access

ALGOL-M programs may read data from, or write data to, one or more disk files which may be located on one or more disk drives. When file input or output is desired, the appropriate READ or WRITE statement is modified by placing a filename icentifier immediately after READ or WRITE. The actual name of the file may be assigned to the file name identifier when the program is written or it may be assigned at run-time. Various disk drives are referenced by the letters A through Z. A specific drive may be specified by prefixing the actual file name with the desired drive letter followed by a colon. Additionally, if random file access is desired, the file name identifier may be followed by an integer constant, variable or expression enclosed in parentheses. This integer value specifies the record within the file which is to be used for input/output.

Prior to the use of a file name identifier in a READ

or WRITE statement, the file name identifier must appear in a file declaration statement. The file name identifier can only be referenced within the same block (or a lower block) as the file declaration. Files are normally treated as unblocked sequential files. However, if blocked files are desired, the record length may optionally be specified in parentheses after the file name identifier in the file declaration statement.

III. IMPLEMENTATION

A. COMPILER IMPLEMENTATION

1. Compiler Organization

The compiler was designed to read source language statements from a diskette and to produce an intermediate language file with optional source listing at the console. A two pass approach was used to facilitate the implementation of GOTO statements, forward subroutines, and control statements. Pass one builds the symbol table and saves all branch locations for resolution during pass two. Pass one also computes the size of the program reference table (PRT) and writes this information out to the intermediate file. Pass two resolves all forward references and emits code to the intermediate file on disk.

2. Scanner

The scanner analyses the source program and sends a sequence of tokens to the parser. In addition, the scanner provides a listing of the source file (if requested), ignores remarks, and sets the compiler toggles. Analysis of the first non-blank character in the input file determines the general class of the next token. The rest of the token is then scanned as it is placed into the accumulator (ACCUM). The first byte of ACCUM contains the length of the token. In the case of constants that exceed the size of

ACCUM (32 bytes) a continuation flag is set. This permits the scanner and parser to continue as necessary to accept the entire constant.

When the scanner recognizes an identifier it searches the vocabulary table (VOCAB) to determine if the identifier is a reserved word. If found, the token number associated with the reserved word's position in the VOCAB table is returned. The reserved word COMMENT is a special case since it is not part of the grammar and is handled entirely by the scanner. The VOCAB table is one of the tables provided by the LALR(1) parse table generator[14].

Constants are passed unconverted from the scanner through the parser to the intermediate file. Although this procedure does not allow constant checking during compile time, it does save space in the compiler. The conversion routines must be in the run-time system for console input and their duplication in the compiler was not considered necessary.

3. Symbol Table

The symbol table stores attributes of program and compiler generated entities such as identifiers, procedures, and labels. The symbol table is constructed during pass one and the stored information is used by the compiler during pass two to verify that the program is semantically correct and to assist in code generation. Access to the symbol table is accomplished through various subroutines which operate on the symbol table through the use of based global

variables.

The symbol table is modeled after the BASIC-E symbol table [6]. It is an unordered linear list of entries which grows toward the top of memory. Individual entries are accessed via a chained hash addressing technique as illustrated in Figure 1. Each location in the hash table heads a linked list whose printnames all evaluate to the same hash address. If there is a zero in the hash table then there are no entries for that particular hash value. During references to the symbol table, the global variable PRINTNAME contains the address of a variable which contains the length of the variable name followed by the name itself. The variable SYMHASH contains the sum of the ASCII characters that make up the variable name, modulo 64. Entries which hash to the same value are chained so that the latest entry is the first one on the chain. They are, however, stored in the symbol table in the order in which they appear in the program.

Each entry in the symbol table contains the following information:

length of printname	1 byte
collision field	2 bytes
printname	variable length
type	1 byte
address	2 bytes
block level	1 byte
subtype	1 byte

SYMBOL TABLE STRUCTURE

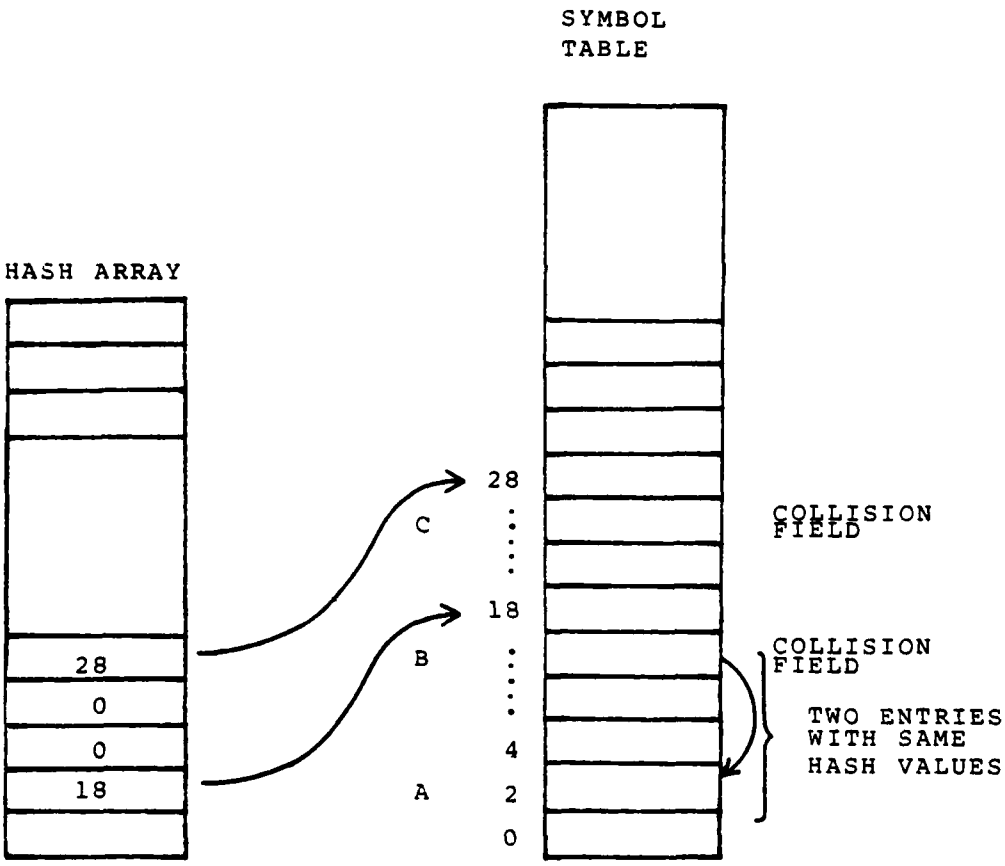


FIGURE 1

The address field indicates the identifier's position in the PRT unless the identifier is a label. For labels, it indicates the label's position in the code area. For subroutines, there are two extra symbol table entries: a parameter field which indicates the number of parameters associated with the subroutine (1 byte), and another address field which indicates the position of the subroutine in the code area.

Since ALGOL-M is completely block structured, there is a block number associated with each identifier in the symbol table. A "previous block level stack" was designed in order to retain the symbol table for debugging purposes during run-time. Each active block is used as an index into this stack which contains all blocks to which the active block is subordinate. When a block is deactivated (i.e., the corresponding block end is encountered), the block number is removed from the previous block stack and therefore any identifiers associated with that block become inaccessible.

Two different lookup routines were designed to facilitate symbol table lookup. The first is FULL$LOOKUP which searches the current block and all outer blocks for an identifier. The second is NORMAL$LOOKUP which simply checks the current block level. In most cases, FULL$LOOKUP is used to determine if an identifier being used has been declared and NORMAL$LOOKUP is used to determine if an identifier being declared has been previously declared in the same block level.

4. Parser

The LALR parser is modeled after that of the Harper parser [6], which is a table-driven pushdown automaton. It receives tokens from the scanner and analyzes them to determine if they are part of the ALGOL-M grammar. When the parser accepts a token, one of the following actions will be taken. It may save the token and continue to accept tokens in the lookahead state, or it may recognize the right part of one of the valid productions and apply the production state (cause a reduction to take place). Finally the parser may determine that the tokens received do not form a valid right part for a production in the grammar and cause a syntax error to be printed.

When an error is detected RECOVER is called and the parser backs up a state with an attempt to continue parsing from that state. If this fails, it continues to back up until the end of the currently pending reduction is reached. At that point the bad token is bypassed and an attempt to parse the following token is made until an acceptable token is found.

The major data structures in the parser are the LALR(1) parse tables and the parse stacks. The parse stacks consist of a state stack and six auxiliary stacks. These auxiliary stacks are parallel to the parse stack and are used to store information needed during code generation. The information stored in these stacks includes variable types, subtypes, and variable addresses.

5. Code Generation

The parser not only verifies the syntax of source statements, but also controls code generation by associating semantic actions with reductions. When a reduction takes place, the procedure SYNTHESIZE is called with the production number as a parameter. SYNTHESIZE copies the needed semantic information from the parse stacks into simple variables (to avoid extensive subscripting) and performs the appropriate semantic action. This is accomplished by the use of a large case statement with the production number as a key. The syntax of the language, along with the semantic actions taken, is listed in Appendix E.

B. INTERPRETER IMPLEMENTATION

1. Building the ALGOL-M Pseudo Machine

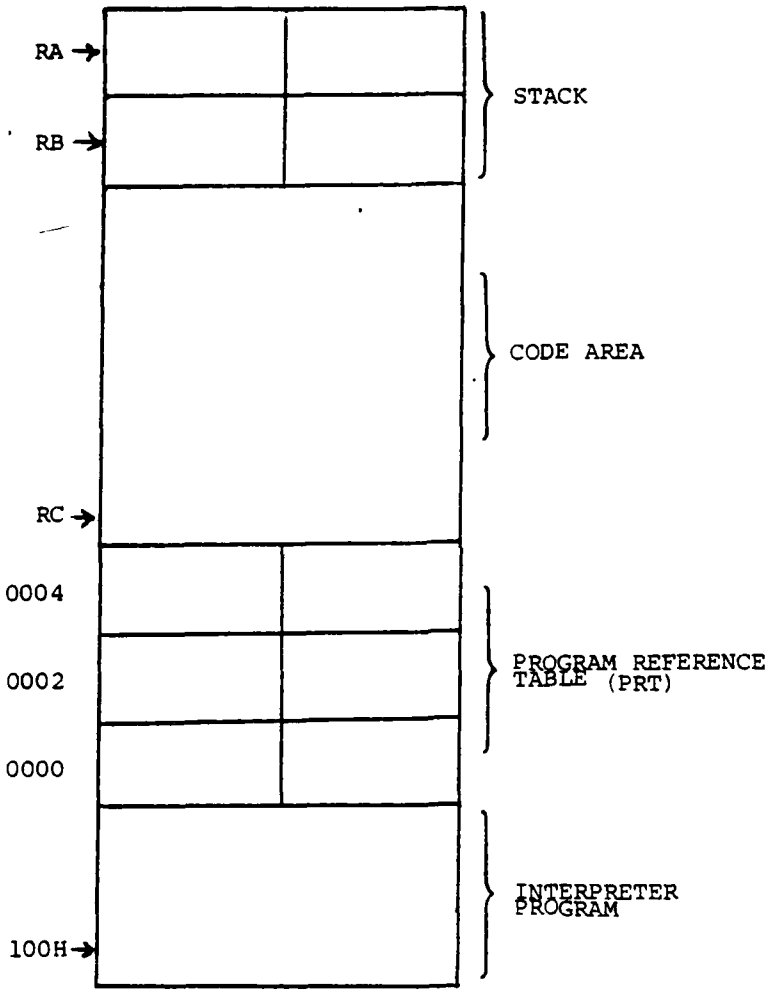
The ALGOL-M pseudo machine, as shown in Figure 2, is a software emulation of a stack-oriented CPU with an instruction set which is particularly well suited for execution of ALGOL-M programs. The ALGOL-M interpreter is loaded at address 100 hex (as are all executable programs under the CP/M operating system) and proceeds to read the ALGOL-M intermediate code from disk, constructing the pseudo machine begining at the first free memory location. The ALGOL-M intermediate code is read into a buffer in 128 byte segments. The first two bytes of the intermediate code represent an integer value equal to the number of bytes

to be used for the program reference table (PRT). Each PRT location is two bytes in length and is used to contain information relative to ALGOL-M program identifiers, arrays, and subroutines.

The remaining intermediate code is manipulated in accordance with the algorithm shown in Figure 5 in order to construct the pseudo machine code area.

2. Overview of the Interpreter

The ALGOL-M interpreter uses the pseudo machine code area as input data. Each pseudo machine operator is equated to an integer value which is evaluated in order to provide the correct entry point into a large case statement in the interpreter. Each entry in the case statement contains the necessary code to cause proper run-time execution of the specific ALGOL-M pseudo instruction. The case statement is executed continually until the ALGOL-M program has been completed, at which time control is passed back to the operating system. A run-time stack is used to facilitate the execution of ALGOL-M pseudo instructions. The stack can be viewed as being two bytes wide and expanding or contracting above the ALGOL-M machine code area as necessary. The top item on the stack is addressed by the variable PA, while the next-to-top item is addressed by PB. The contents of the two bytes on top of the stack are referenced by the variable APA while the two byte contents of the next-to-top stack position are referenced by the variable APB. The low order byte



ALGOL-M MACHINE
MEMORY ORGANIZATION

FIGURE 2

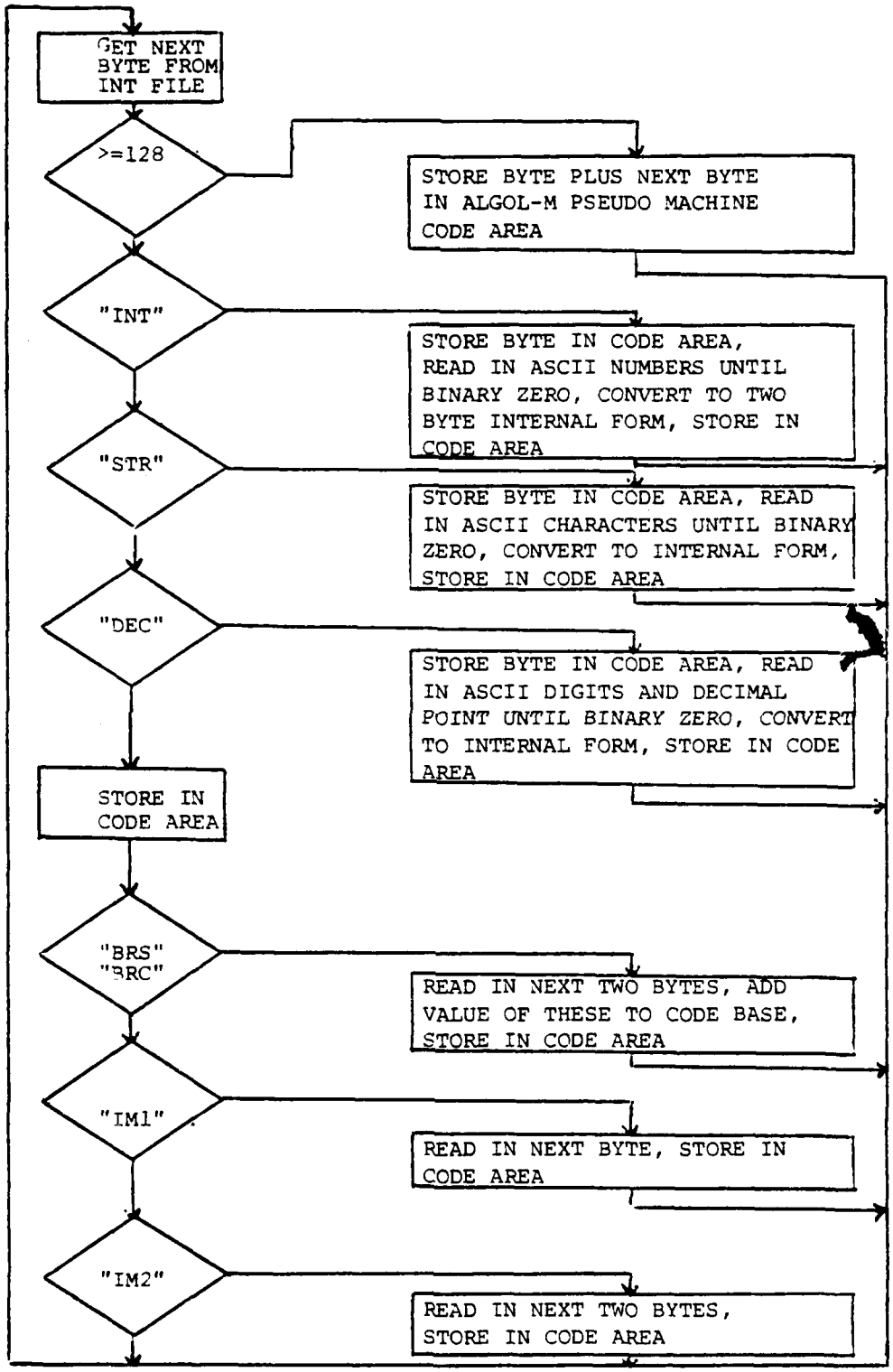

GET NEXT BYTE FROM INT FILE
>=128
STORE BYTE PLUS NEXT BYTE IN ALGOL-M PSEUDO MACHINE CODE AREA
"INT"
STORE BYTE IN CODE AREA, READ IN ASCII NUMBERS UNTIL BINARY ZERO, CONVERT TO TWO BYTE INTERNAL FORM, STORE IN CODE AREA
"STR"
STORE BYTE IN CODE AREA, READ IN ASCII CHARACTERS UNTIL BINARY ZERO, CONVERT TO INTERNAL FORM, STORE IN CODE AREA
"DEC"
STORE BYTE IN CODE AREA, READ IN ASCII DIGITS AND DECIMAL POINT UNTIL BINARY ZERO, CONVERT TO INTERNAL FORM, STORE IN CODE AREA
STORE IN CODE AREA
"BRS"
"BRC"
READ IN NEXT TWO BYTES, ADD VALUE OF THESE TO CODE BASE, STORE IN CODE AREA
"IM1"
READ IN NEXT BYTE, STORE IN CODE AREA
"IM2"
READ IN NEXT TWO BYTES, STORE IN CODE AREA

FIGURE 3

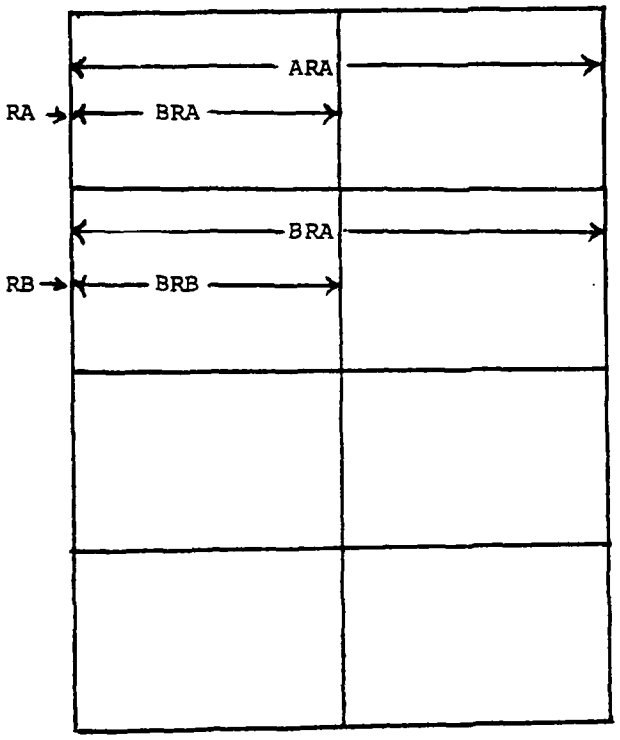
contents of the top and next-to-top stack locations are referenced by the variables BRA and BRB respectively. The various stack variables are depicted in Figure 4.

3. Allocation of Storage Space

Run-time storage space is required for the values associated with ALGOL-M program identifiers which have been declared as integer, decimal, or string values, and for information needed to process arrays and subroutines. A sequential number is assigned to each new identifier as it is recognized by the compiler. This number is used to reference the PRT at run-time in order to store or retrieve the value associated with each identifier.

a. Integers

Integer values range from -16,384 to +16,384 and are stored directly in the two bytes allocated in the PRT for integer identifiers. A maximum length of two bytes for integer values was chosen because the resulting range of possible integer values was considered adequate for the primary use of integers as program control counters, such as array subscripts and loop boundaries. Additionally, two-byte values were the most convenient size to work with in the implementation language PL/M [8]. The high order bit of the integer representation is the sign bit, with zero indicating a positive value and one indicating a negative value. The second bit of the integer representation is always zero in order to permit



ALGOL-M STACK
VARIABLES

FIGURE 4

differentation between integers and other types on the stack. The ALGOL-M internal form for addresses is also a two byte value.

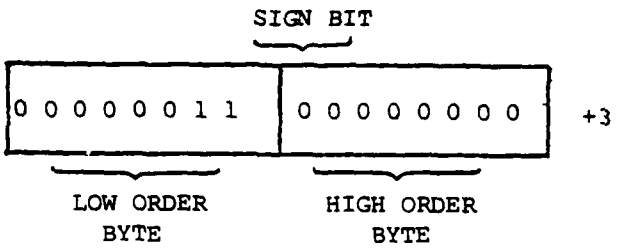
b. Decimals

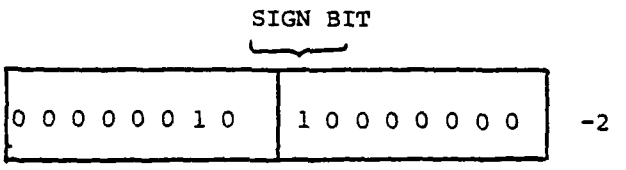
Decimal values up to 18 digits in length are permitted in the ALGOL-M language. Each decimal identifier is associated with a unique PRT entry. The value stored in the two byte PRT entry represents the run-time address of the location on the ALGOL-M stack where the actual decimal value is stored. The format for decimal storage is shown in Figure 6.

The next-to-last byte of the allocated space for decimal identifiers contains the number of bytes used for storage of that value. This value is a function of the size declared for the decimal by the programmer and may be delayed until run-time. A maximum of 18 digits of precision may be declared with the default precision being ten digits. The first byte of the decimal storage area contains a value representing the number of bytes used to hold the actual packed digits. This value may be less than the number of bytes which could be stored in the allocated area. In order to save storage space, the decimal values are packed two digits per byte of storage space.

ALGOL-M is a block structured language based upon a stack model for execution. Thus, it allows efficient allocation of storage for decimal identifiers. A block is normally bracketed by the ALGOL-M keywords

```
BEGIN
INTEGER A,B;
A := 3;
B := -2;
END
```

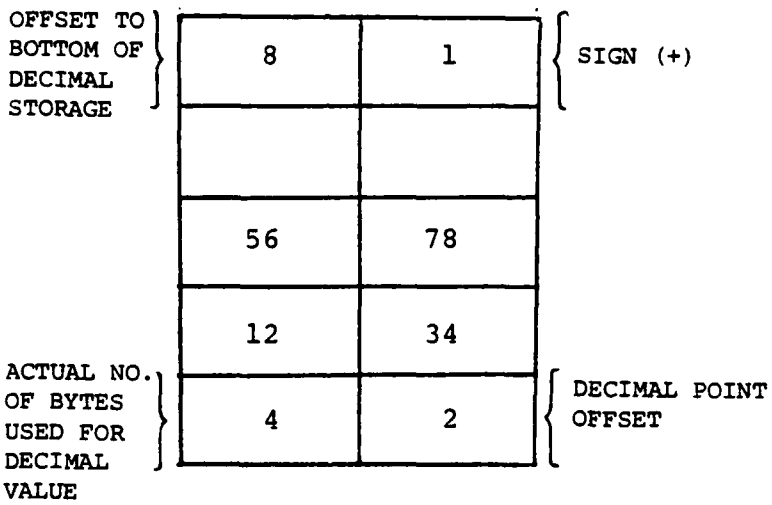




INTEGER STORAGE

FIGURE 5

```
BEGIN
DECIMAL(12) X;
X := 123456.78;
END
```



STACK

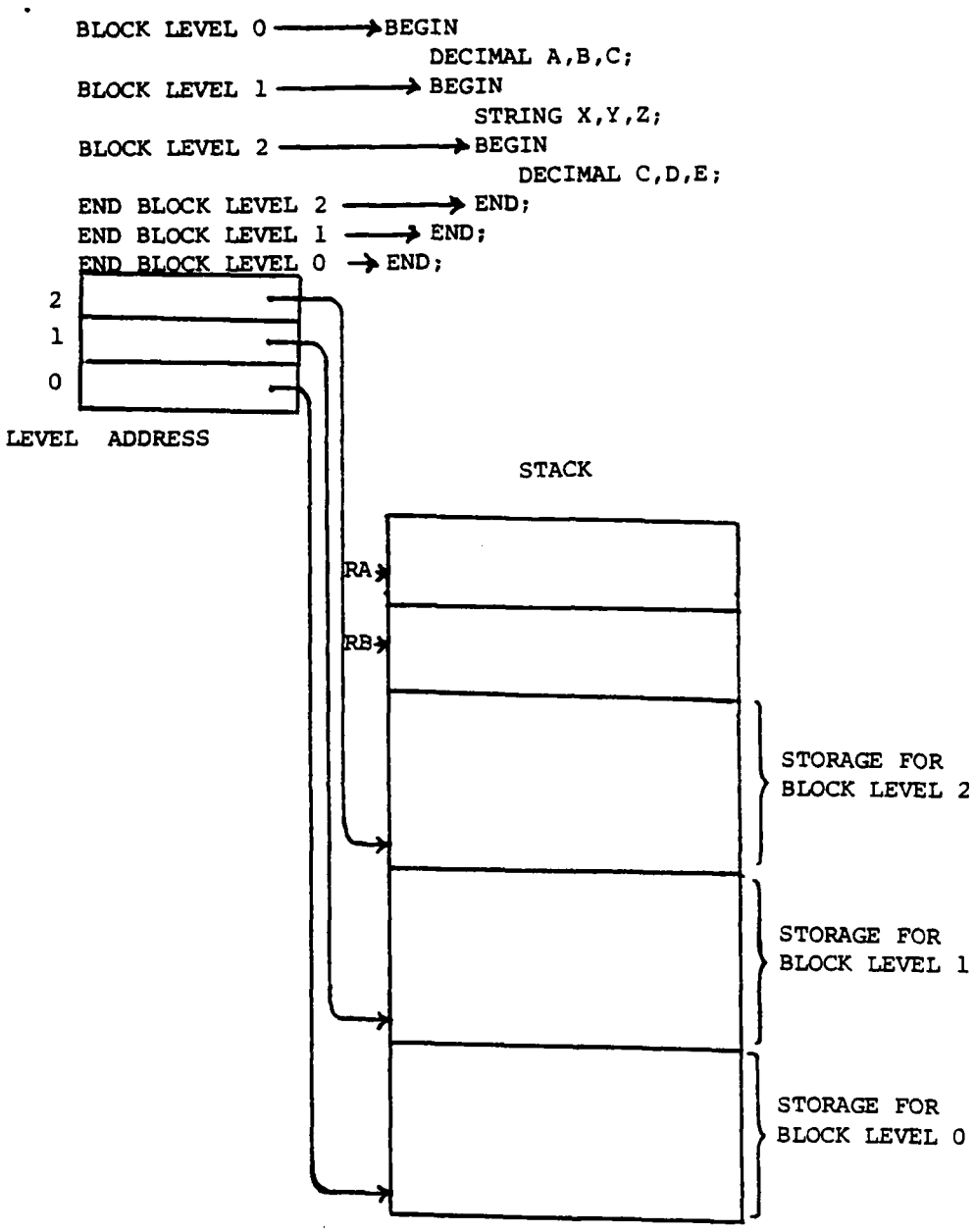
DECIMAL STORAGE

FIGURE 6

BEGIN and END. Blocks may be embedded within a higher level block as shown in Figure 7. Identifiers which are declared in a given block are considered local to that block and global to any lower lever block and therefore may be referenced from those blocks. However, once execution of an ALGOL-M program proceeds beyond a given block, the identifiers declared within that block may no longer be referenced and, in fact, the storage allocated for those identifiers is removed from the ALGOL-M machine stack.

c. Strings

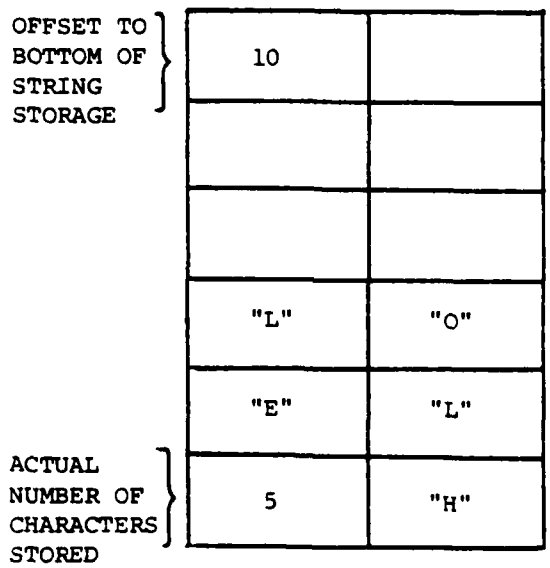
Strings of ASCII characters up to 255 bytes in length are permited in ALGOL-M. In the same manner as used for decimal storage, the string identifier is associated with a unique PRT entry. The corresponding PRT entry contains the address of the actual string storage space on the ALGOL-M stack. The format for string storage is shown in Figure 8. The next-to-last byte of the allocated storage space for a string identifier contains the value of the number of bytes actually allocated by the programmer up to a maximum of 255 bytes. The first byte of the allocated string storage area contains the value of the actual number of ASCII characters stored in the allocated area. The concept of storage allocation as related to block levels is the same as described for decimal identifiers.



BLOCK LEVELS AND STACK STORAGE

FIGURE 7

```
BEGIN
STRING(9) TESTWORD;
TESTWORD := "HELLO";
END
```



STACK

STRING STORAGE

FIGURE 8

d. Arrays

ALGOL-M arrays may be of type integer, decimal, or string. Arrays may consist of up to 255 dimensions with each dimension containing up to 16,384 elements. The upper and lower bounds of each dimension may consist of any positive integers, variables, or arithmetic expressions. Each array name is associated with a unique PRT entry. The PRT entry contains the address pointer to the actual array storage area on the ALGOL-M machine stack. The first part of the array storage area on the stack consists of the displacement vector and other information which is necessary to calculate the address of any specific array element at run-time. The format for array storage is shown in Figure 9. The allocated storage area for each array element is exactly like that used for integers, decimals, or strings which are declared as single identifiers. The algorithm used for calculating the displacement vector is expressed as:

$$D_I = \begin{cases} \text{IF } I = N \text{ THEN } 1, \text{ OTHERWISE} \\ (U_{I+1} - L_{I+1} + 1) \cdot D_{I+1} \end{cases}$$

where N is the number of dimensions, I is the respective dimension, and U is the upper bound and L is the lower bound of a dimension.

The offset vector, V, is calculated by:

$$V = -\sum_{I=1}^{N} L_I D_I$$

The offset vector represents the correction necessary for non-zero-origin subscripts. This approach to locating elements in dynamically declared arrays is essentially the same as that used in ALGOL-F [10].

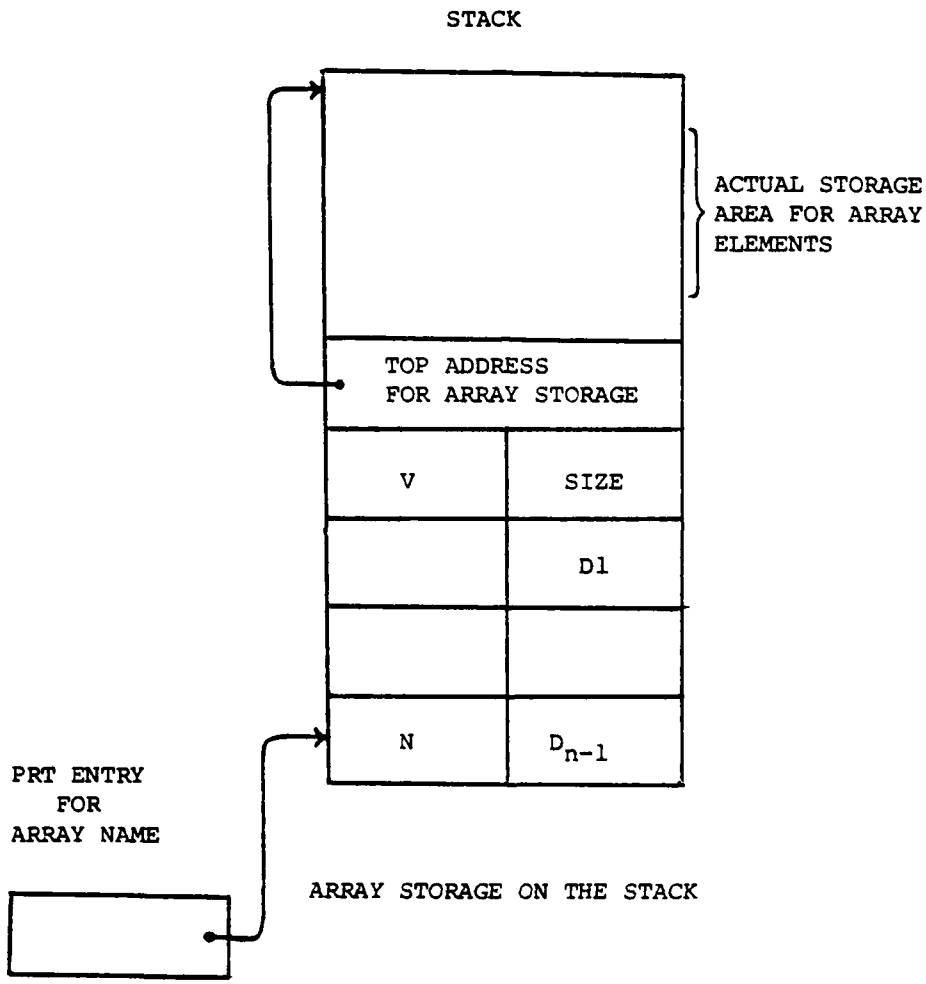

STACK
ACTUAL STORAGE AREA FOR ARRAY ELEMENTS
TOP ADDRESS FOR ARRAY STORAGE
V
SIZE
D1
N
D_{n-1}
PRT ENTRY FOR ARRAY NAME
ARRAY STORAGE ON THE STACK

FIGURE 9

4. Storage and Retrieval of Variables

Once allocation of storage for variables has been completed at run-time, storage and retrieval of actual values is relatively simple. A check is first made to insure that the number of bytes to be stored is less than or equal to the storage space allocated for a given variable. In the case of strings, if the destination storage space is not large enough to hold the entire string, as many characters as possible are stored and a run-time warning message is issued advising that string overflow has occurred. For decimal storage, if the total number of packed digits to be stored is larger than the available storage space, non-significant digits are deleted until the decimal value can be stored. If a significant digit must be deleted in order to store a decimal value, an error message is generated and the stored decimal value is arbitrarily set to 1.0 to allow continuation of program execution.

Array elements are stored exactly as the corresponding single element variables. However, the data located at the beginning of each array storage area (refer to Figure 9) is used to calculate the actual location of a specific array element. This is accomplished by initializing the offset variable to the value of the rightmost subscripted value. This value is then added to the product of the next rightmost subscripted value and the n-1 displacement vector value. This procedure is continued until the left-most subscripted value has been used in

the calculation. Next the offset vector, v , plus one, is subtracted from the offset variable and the result is multiplied by the size of the area allocated to each array element. The result of this calculation is the total offset, in bytes, from the beginning of the storage area for the array to the specific array element in question. A check is made to insure the calculated total offset is not greater than the offset which would result in access to the last array element. An error message is generated by the interpreter if a subscripted variable is referenced with subscripts that are not within the declared array dimensions.

5. Arithmetic Operations

Arithmetic operations for integer variables are straightforward because the implementation language, PL/M [8], provides all of the necessary two byte arithmetic operations. Therefore, the two integer values which are to be added, subtracted, multiplied, or divided are placed one above the other on the stack and the appropriate routine performs the necessary arithmetic operation, replacing the original two integer values with the result of that operation. The resulting value is then available to store into the space allocated for an integer variable or to be used as one of the integer values for continued arithmetic operations.

Decimal arithmetic is accomplished by manipulating packed decimal strings, each of which is loaded in a ten

byte register (two digits per byte). The result of the decimal arithmetic operation is stored in a third register. Decimal values are also stored in the packed decimal form, shown in Figure 6. Decimal strings are only unpacked when and if they are written to disk or console at run-time. Decimal addition is accomplished by adding the two registers, subtraction is done using nines complement arithmetic, multiplication is done through a shift and add algorithm, and division by a shift and subtract method. After the decimal arithmetic operation is completed, the result is placed on the top of the stack in preparation for a decimal store operation or use as a new value in a continuing algebraic expression.

6. String Operations

The ALGOL-M compiler is designed to handle strings up to 255 characters in length. The concatenation operator allows two or more strings to be combined to produce a new string consisting of all the characters contained in the original strings. The process of concatenation takes place on the stack with strings being combined repeatedly as necessary for multiple concatenations. The resulting string is then available for storage in the space allocated to a string identifier. If the result of concatenation produces a string which is longer than the allocated storage space, the string is truncated as necessary and an error message is issued by the interpreter.

7. Subroutines

There are two types of subroutines in ALGOL-M: functions and procedures. The only difference between the two is that a function returns a value to the top of the stack while a procedure does not. Subroutines are fully recursive and can be called prior to their declaration.

a. Invocation

A subroutine can be invoked with zero or more actual parameters. The actual parameters consist of integer, decimal, or string expressions which are evaluated and passed to the subroutine via the execution stack. In addition to parameter values, the only other information needed to call a procedure is the procedure address in the code area. The actual format of the stack at the point of a subroutine call is indicated in Figure 10.

b. Storage Allocation

Storage for variables and parameters declared within a subroutine is allocated on the stack at run-time. The actual parameters and the subroutine call information is also stored on the execution stack. To allow subroutine call by value and to save memory, it was necessary to move the actual parameter values and subroutine call information off the stack prior to allocating storage for the formal parameters and local variables. This was accomplished by moving them to the top of available memory. Storage can then be allocated for the formal parameters on top of the

STACK CONFIGURATION FOR
PRO OPERATOR

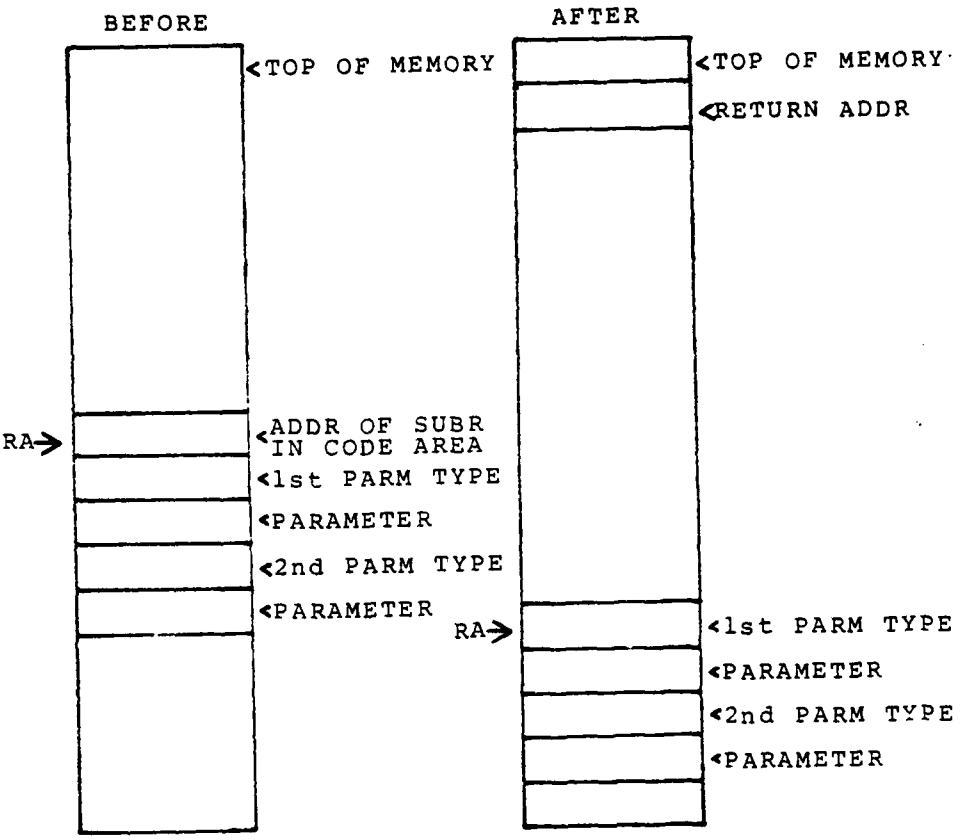


FIGURE 10

stack in the normal manner. This move was accomplished through the SAV operator as illustrated in Figure 11. The SAV operator also checks to determine if the call is a recursive call and if so, saves the procedure control block (PCB) on top of the stack. The PCB is similar in structure to that of ALGOL-E[10]. Although this procedure slows down the execution speed of the interpreter, it was considered more important to save memory space. Storing subroutine values on the stack as indicated above simplifies the de-allocation of memory at the end of the subroutine. This is accomplished by simply removing elements from the stack down to the appropriate level.

c. Parameter Mapping

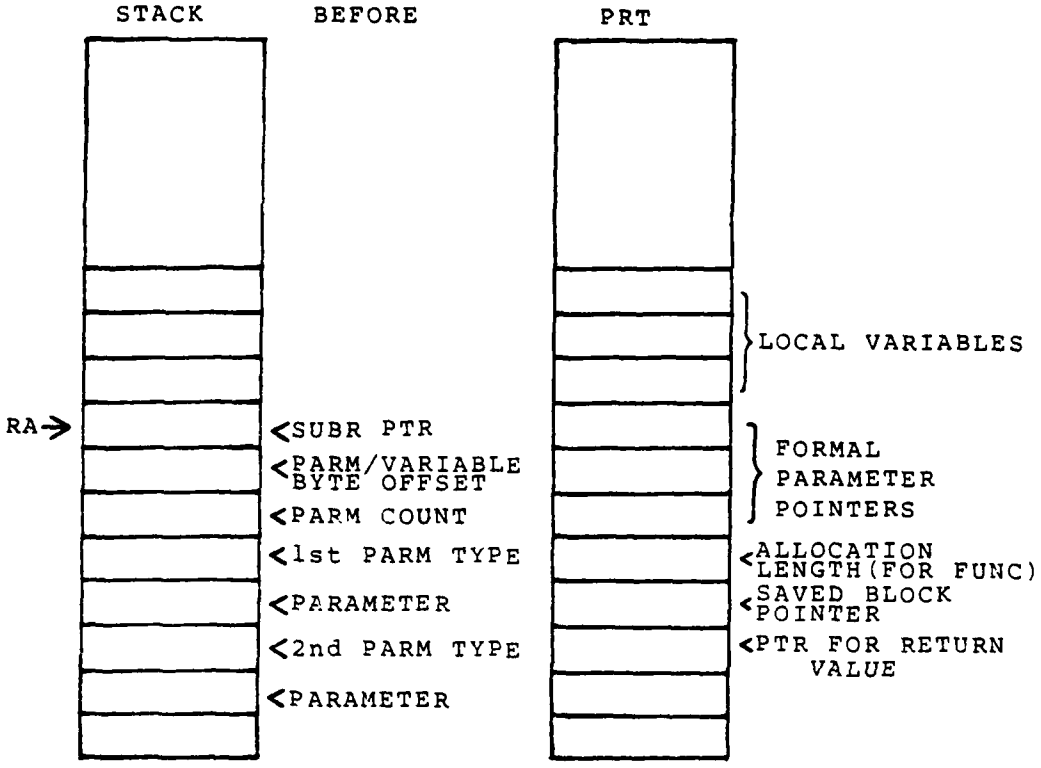
Parameter mapping is done through the use of the SV2 operator as illustrated in Figure 12. The operator copies the actual parameter information at the top of memory into the area allocated on top of the stack. The PCB which keeps track of subroutine variables is then set with pointers to the current parameter values.

d. Function Return Value

Included with the allocated area for parameters and local variables, there is an additional allocated area for the return value of functions. The function name is treated as a simple variable within the function and the return value is assigned to it. This value is copied to the top of the stack when the function returns.

STACK CONFIGURATION FOR

SAV OPERATOR



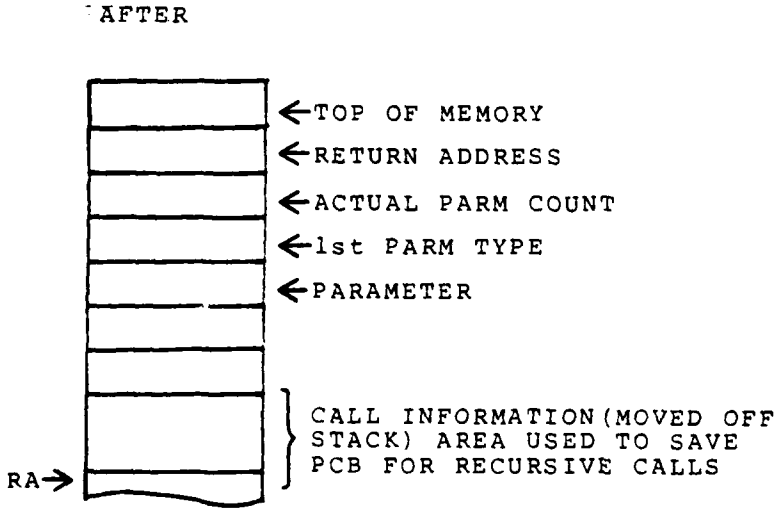


FIGURE 11

STACK CONFIGURATION FOR
SV2 OPERATOR

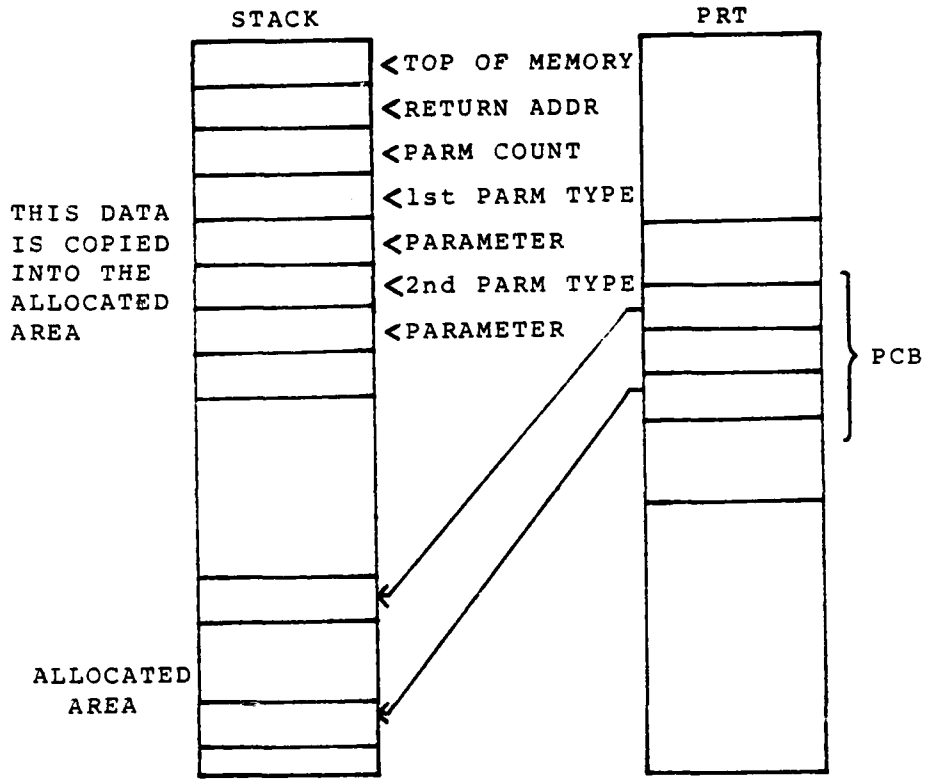


FIGURE 12

e. External Functions

The ALGOL-M grammar supports external function declarations and, although they have not been implemented, their design has been considered. External functions are conceived to be ALGOL-M intermediate files which could be declared and called from other ALGOL-M programs. The intermediate code for these files would be read from disk by the compiler and integrated into the program intermediate code as if the external subroutine had in fact been a normal program subroutine. The grammar would require modification in order to allow a subroutine to be compiled by itself.

f. Forward References

Forward subroutine references are supported by simply treating all undeclared identifiers as forward subroutine references during pass one of the compiler. If on pass two these references have not been resolved, then an undeclared variable error is generated by the compiler.

g. Built-in Functions

There are currently no built-in functions in ALGOL-M. However, their implementation has been considered. To implement built-in functions, a simple modification to the symbol table structure would be needed, allowing the built-in function names to be entered in the symbol table when it is initialized. There would be an operator associated with each function name in the symbol table which would indicate the run-time action to

be taken. This operator would be emitted each time the built-in function was referenced.

8. Input-Output

Two basic types of input-output (I/O) are implemented in the ALGOL-M language: console and disk. Console I/O refers to the device which is being used to provide commands to the system, typically a cathode-ray tube terminal or a teletype. Input is accomplished via ALGOL-M program READ statements and output via WRITE or WRITEON statements. Integer and decimal values, including signs and decimal points, are converted from their internal representation into ASCII characters which are provided to an operating system print routine for console output. String variables and constants are stored in the ALGOL-M pseudo-machine as ASCII strings and are sent character by character to the system print routine. Console input is accomplished via an operating system routine which reads one full console line into an ALGOL-M buffer. The interpreter examines the buffer and converts the ASCII characters in the buffer into the appropriate ALGOL-M internal decimal, integer or string format. The input value is then stored in the space allocated for its variable name.

The ALGOL-M READ or WRITE statement for disk I/O contains the name of the disk file to be used and, optionally, specifies the disk drive containing that file. The default drive is the currently logged drive [4]. An

additional option is that any specific record on the file may also be specified in the ALGOL-M READ or WRITE statement. A file declaration statement is used to associate file identifier names with a specific entry in the ALGOL-M PRT. At run-time, space is allocated on the ALGOL-M stack for file control block (FCB) information necessary to interface file operations with the operating system. In addition to the FCB, space is also allocated for a 128 byte I/O buffer for each declared file. The routines to convert packed decimal and integer numbers from internal form to ASCII form, and vice-versa, which are used for writing to and reading from disk files are the same as those used for console I/O. Any number of files may be open simultaneously and, as with all run-time storage, the space allocated on the stack for file operations is recovered when the block is exited.

9. ALGOL-M Pseudo Operators

a. Description of Interpreter Variables

The top item on the ALGOL-M stack is addressed by the variable RA while the next-to top item on the stack is addressed by the variable RB. The value contained in the first two bytes addressed by RA and RB are referenced by the variables ARA and ARB respectively. The contents of the low-order byte of ARA and ARB is referenced by the variable BRA or BRB respectively. The structure of the stack is shown in Figure 4. Decimal and string values may be represented on the stack by an address which yields the

actual storage area for the value, or by the actual value itself stored as one of the items on the stack. When a decimal or string value is stored on the stack, it is referred to as a temporary value. Temporary values are stored on the stack in preparation for storage into the area allocated to a specific identifier, or for use as a value in the evaluation of an expression.

The ALGOL-M compiler generates various pseudo operators which were chosen to allow effective run-time execution of the ALGOL-M pseudo machine. Following is a list of the pseudo operators and a brief description of the action taken at run-time when each operator is encountered in the ALGOL-M code area.

b. Literal Data References

An initial check is made of each one byte operator in the code area in order to determine if the high order bit of that byte is set to one. If the high order bit is set then the least significant 14 bits of that byte and the following byte are automatically added to the address of the beginning of the program reference table and placed on top of the stack. A check is then made of the second bit of the original byte and if it also is set to one the PRT address now on top of the stack is replaced by the contents of the two bytes pointed to by that address. These are referred to as LIT and LITLOD operators.

INT: (integer). The following two byte integer value is placed on the stack.

STR: (string). The program counter is incremented past the following string constant and the address of the next-to-last byte of the string constant is placed on the stack.

DECT: (decimal). The program counter is incremented past the following decimal constant and the address of the next-to-last byte of the decimal is placed on the stack. The next-to-last byte of the decimal constant contains the offset to the first byte of the constant which in turn contains the value of the number of bytes in which actual decimal digits are stored.

IM1: (load a one byte integer). The value of the following byte is converted to a two byte value and placed on the stack.

IM2: (load a two byte integer). The following two byte value is placed on the stack in reverse order.

c. Allocation Operators

ALD: (allocate decimal). Storage for a decimal variable is allocated on the stack and the address of the allocated area is placed in the PRT entry for the specific decimal variable.

ALS: (allocate string). Storage for a string variable is allocated on the stack and the address of the allocated area is placed in the PRT entry for the specific string variable.

AID: (allocate intermediate decimal). The same action is taken as in ALD except the declared decimal length

is left on top of the stack in preparation for the next allocation which is expected to immediately follow in the code area. This operator is used when several decimal identifiers are declared in a single declaration statement.

AIS: (allocate intermediate string). The same action is taken as in the AID operator with the exception that string allocation is used.

d. Arithmetic Operators

ADI: (add integer). The integer values of the top two items on the stack are replaced by their integer sum.

ADD: (add decimal). The decimal value of the top item on the stack is loaded into decimal arithmetic register zero, and the value of the second item on the stack is loaded into decimal arithmetic register one. The two arithmetic registers are added with the result placed in register two. The original decimal values on the stack are replaced by the result of the arithmetic operation.

SBI: (subtract integer). The second integer value on the stack is subtracted from the integer value on top of the stack, both values are removed from the stack and the result of the operation is placed on top of the stack.

SBD: (subtract decimal). The same action is taken as in the ADD operator except the second item is subtracted from the top item on the stack.

MPI: (multiply integer). The same action is taken as in the SBI operator except the top two items are

multiplied.

MPD: (multiply decimal). The same action is taken as in the ADD operator except the top two items are multiplied.

DVI: (divide integer). The same action is taken as in the SBI operator except the top item is divided by the second item.

DVD: (divide decimal). The same action is taken as in the ADD operator except the top item is divided by the second item.

NEG: (negative). The sign of the decimal or integer on top of the stack is changed.

CI1: (convert integer). The integer on top of the stack is replaced by its decimal equivalent.

CI2: (convert integer). The same action is taken as with CI1 except the integer which is the second item on the stack is converted to a decimal.

LSS: (integer less than). The integer value (ARB) is compared with the integer value (ARA). Both values are removed from the stack. If ARB was less than ARA, the value one is placed on the stack, otherwise the value zero is placed on the stack.

DLSS: (decimal less than). The decimal on top of the stack is compared to the decimal which is the second item on the stack. If the value on top of the stack is less than the second value on the stack, both values are removed from the stack and replaced by the value one, otherwise they are replaced by the value zero.

GTR: (integer greater than). The same action is taken as in the LSS operator except a one is placed on the stack if ARB is greater than ARA.

DGTR: (decimal greater than). The same action is taken as in the DLSS operator except a one is placed on the stack if the second decimal item is greater than the top decimal item.

EQL: (integer equal to). The same action is taken as in the GTR operator except a one is placed on the stack if ARB is equal to ARA.

DEQL: (decimal equal to). The same action is taken as in the DGTR operator except a one is placed on the stack if the second decimal item is equal to the top decimal item.

NEQ: (integer not equal to). The same action is taken as in the EQL operator except a one is placed on the stack if ARB is not equal to ARA.

DNEQ: (decimal not equal to). The same action is taken as in the DEQL operator except a one is placed on the stack if the second decimal item is not equal to the top decimal item.

GEQ: (integer greater than or equal to). The same action is taken as in the NEQ operator except a one is placed on the stack if ARB is greater than or equal to ARA.

DGEQ: (decimal greater than or equal to). The same action is taken as in the DNEQ operator except a one is placed on the stack if the second decimal item is greater than or equal to the top decimal item.

LEQ: (integer less than or equal to). The same action is taken as in the GEQ operator except a one is placed on the stack if ARB is less than or equal to ARA.

DLEQ: (decimal less than or equal to). The same action is taken as in the DGEQ operator except a one is placed on the stack if the second decimal value is less than or equal to the top decimal value.

NOT: (boolean not). This operator changes the result of any previous boolean operator by complementing the value of the one or zero which was placed on the stack by the previous operation.

AND: (boolean and). This operator checks the top two values left on the stack by any two previous boolean operations. If both values are one then both values are replaced with a one; otherwise both values are replaced with a zero.

OR: (boolean or). This operator checks the top two values left on the stack by any two previous boolean operations. If either value is a one then both values are replaced by a one; otherwise they are replaced by a zero.

e. String Operators

CAT: (concatenate). The two strings on top of the stack are combined to produce a new string consisting of the characters of the second string followed by the characters of the first string. The two original strings are popped from the stack and replaced by the resulting concatenated string.

SLSS: (string less than). The same action is taken as in the DLSS operator except that a character by character string comparison is made using the ASCII character collating sequence.

SGTR: (string greater than). The same action is taken as in the DGTR operator except a string comparison is made.

SEQL: (string equal to). The same action is taken as in the DEQL operator except a string comparison is made.

SNEQ: (string not equal to). The same action is taken as in the DNEQ operator except a string comparison is made.

SGEQ: (string greater than or equal to). The same action is taken as with DGEQ except a string comparison is made.

SLEQ: (string less than or equal to). The same action is taken as with DLEQ except a string comparison is made.

f. Stack Operators

XCH: (exchange). The value of the top two bytes on the stack (ARA) is exchanged with the value of the next-to-top two bytes on the stack(ARB).

POP: (pop the stack). The stack pointer (RA) is moved to the position of the stackpointer (RB) and RB is moved to point to the next item below its current position on the stack.

LOD: (load). The address value on the stack (ARA) is replaced by the two bytes pointed to by that address.

DCB: (decrement block by more than one level): The stack pointer (RA) is decremented to the address stored in the block level table and the index to the block level table is decreased by the value stored in the next two bytes in the code area. The stack pointer (RB) is moved below RA to the top of the second item on the stack.

BLI: (block level increment). The index to the block level array is increased by one and the address of the top item on the stack is stored in the block level array.

BLD: (block level decrement). The index to the block level array is decreased by one and the value of the stack pointer (RA) is changed to the address stored in the block level array.

SPR: (subtract stack values). This operator is used to subtract the second value on the stack from the top value on the stack using unsigned sixteen bit arithmetic. The two values are replaced by the result of the subtraction.

g. Array Operators

ROW: (allocate array storage). The number of array dimensions, the upper and lower bounds of each dimension, and the array type (integer, decimal, or string) are used to calculate the array displacement vector which is stored on the stack prior to allocation of storage for the

actual array elements.

SUB: (calculate the offset to a specific array element). The array subscript is used in conjunction with the displacement vector information stored at the beginning of the array storage area to calculate the address of the specific array element being referenced by the subscripted variable.

h. Program Control Operators

bRS: (branch absolute). The program counter is changed to an address one less than the address represented by the following two bytes in the code area.

BRC: (branch conditional). If the value on top of the stack is zero, the program counter is changed as in bRS; otherwise the program continues with the next operator in the code area.

BRA: (computed branch absolute). The program counter is changed relative to the start of the code area by the value on top of the stack.

XIT: (exit the interpreter). XIT caused return of control to the operating system.

i. Store Operators

STI: (store integer intermediate). The integer value which is the second item on the stack is stored in the PRT address which is the top item on the stack. The PRT address is then removed from the stack.

SDI: (store decimal intermediate). The same action is taken as with STI except a decimal value is stored

in the allocated area pointed to by the address on top of the stack.

SSI: (store string intermediate). The same action is taken as in the SDI operator except a string value is stored.

SID: (store integer destruct). The same action is taken as in the SII operator except both the PRT address and the integer value are removed from the stack.

SDD: (store decimal destruct). The same action is taken as in the SID operator except the pointer to the decimal allocated area and the decimal value are removed from the stack.

SSD: (store string destruct). The same action is taken as in the SDD operator except the pointer to the string allocated area and the string value are removed from the stack.

j. Input/Output Operators

DMP: (dump). DMP signifies the end of writing a line to the console. A carriage return and line feed are output to the console via the operating system.

WIC: (write integer to console). The integer value on top of the stack is converted to ASCII characters and printed on the console.

WDC: (write decimal to console). The decimal value on top of the stack is converted to ASCII characters and printed on the console.

WSC: (write string to console). The ASCII

string value on top of the stack is printed on the console.

WID: (write integer to disk). The integer value on top of the stack is converted to ASCII characters and stored in the disk buffer allocated for the file name specified in the source WRITE statement.

WDD: (write decimal to disk). The decimal value on top of the stack is converted to ASCII characters and stored in the disk buffer allocated for the file name specified in the source WRITE statement.

WSD: (write string to disk). The ASCII string characters on top of the stack are stored in the disk buffer allocated for the file name specified in the source WRITE statement.

RCI: (read console integer). The console read buffer is scanned for the ASCII representation of an integer which is converted into internal form and placed on top of the stack.

RCD: (read console decimal). The console read buffer is scanned for the ASCII representation of a decimal value which is converted into internal form and placed on top of the stack.

RCS: (read console string). The console read buffer is scanned for an ASCII string or for any series of characters delimited by quotation marks which is placed on top of the stack.

RDI: (read disk integer). The disk buffer allocated to the file name appearing in the source language READ statement is scanned for the ASCII representation of an

integer which is converted to internal form and stored on top of the stack.

RDD: (read disk decimal). The same action is taken as in the RDI operator except a decimal value is placed on top of the stack.

RDS: (read disk string). The same action is taken as in the RDI operator except a string value is placed on top of the stack.

RCN: (load console buffer). The current line on the console is dumped into the console read buffer and the current program counter is stored in preparation for the possibility of a console error and the subsequent need to recover for repeated console input.

ECR: (error in console read). If characters remain in the console read buffer after all console read operations have been completed, an error condition exists and the program counter is reset to the start of the console read routine allowing the console input to be entered again as necessary.

OPN: (disk open). The address on top of the stack points to the allocated area for the file control block and disk buffer associated with the disk file name specified in the source language file declaration statement. The file name which is stored in the file control block is passed to the operating system which in turn opens that specific file for disk input/output.

CLS: (disk close). The file name located in the file control block pointed to by the address of the top of

the stack is passed to the operating system which in turn closes that specific file to input/output.

RDB: (ready sequential block). The interpreter input/output routines are initialized to operate with the file control block and disk buffer area pointed to by the address on top of the stack. The address of the code to be executed upon reaching the end of a file is also initialized.

RDF: (ready random block). The same action is taken as with RDB with the addition that the specific record of the disk file is also taken from the top of the stack and the file control block is set up to conduct input/output from or to that specific record.

EDR: (end of record for read). At the end of a read statement the remainder of the record is skipped.

EDW: (end of record for write). At the end of a write statement the remainder of the record is filled with blanks and a line terminator is appended to the end of the record.

k. Subroutine Operators

PRO: (subroutine call). The two bytes of code following the PRO operator represent the address of the subroutine in the code area. This operator saves the return address at the top of memory, positions the stack pointer at the top of the first actual parameter (see Figure 10 for parameter format) and branches to the first statement in the subroutine.

SAV: (save actual parameters). The SAV operator expects the stack format illustrated in Figure 11. It copies the actual parameters to the top of memory and checks the PCB for a recursive call and if so it copies the PCB onto the top of the stack.

SV2: (copy actual parameters into formal parameters). The SV2 operator copies the actual parameters at the top of memory into allocated area for the formal parameters on top of the stack.

UNS: (unsave parameters). The UNS operator checks the PCB to see if it is the end of a recursive call. If this is the case, UNS restores the PCB from the previous call. UNS also returns the value associated with the name of the subroutine to the top of the stack. In the case of procedures the returned value is zero.

RTN: (return). The RTN operator changes the value of the program counter to the value of the return address.

IV. CONCLUSIONS

This project has resulted in the construction of a high-level, block-structured, applications oriented compiler for micro-computers with 20k bytes of memory or more. When compared to a fully dynamic scheme, the stack storage allocation and retrieval scheme presented here appears to enhance program execution speed, reduce memory requirements, and simplify compiler implementation.

Timing tests with several benchmark programs have been conducted. These test programs were obtained from REF [7] and have been run with several versions of the BASIC programming language. The results (expressed as execution time in seconds) are summarized as follows:

BENCHMARK NUMBER	1	2	3	4	5	6	7
ALGOL-M	1.8	1.3	3.5	3.3	5.1	14.2	16.4
INTEGER BASIC	1.3	3.1	7.2	7.2	8.8	18.5	28.0
STANDARD BASIC	1.7	7.5	20.6	20.9	22.1	36.2	51.8

Listings of the seven benchmark programs are contained in Appendix A. The reason ALGOL-M appears slower in the first benchmark is that the grammar requires at least one executable statement in FOR loops when compared to BASIC which has FOR loops which can do nothing. The resulting ALGOL-M program executes 1000 extra assignment statements which the BASIC programs did not. Otherwise, the results

clearly indicate that the ALGOL-M language is not only well-structured but also executes rapidly in comparison with the very best BASIC interpreters. It is believed that the major reason that the ALGOL-M versions performed so well is the fact that the language supports integer arithmetic as opposed to most BASIC implementations which convert all constants and variables to floating point format. This allows ALGOL-M loop control counters to be incremented extremely rapidly. Comparisons for decimal calculations have not been done, and thus no conclusions can be drawn concerning relative speeds of calculation-dependent programs.

V. RECOMMENDATIONS

There are several areas that could be enhanced in this implementation of ALGOL-M. Formatted I/O although defined in the grammar has not been implemented. The I/O definition is very similar to that of COBOL and implementation of this should not be too difficult. File I/O is implemented, but not tested. Debugging facilities in the run-time monitor are not currently implemented. However, the system is designed to provide the following information at run-time: the line number of the currently executing line and the value of each variable as it is altered.

The current version of ALGOL-M is designed to run on a system with at least 20k bytes of memory. A smaller system could be designed to run on a 16k system. This would involve deleting some of the more complicated sections of code such as dynamic arrays and recursive subroutines. The other features of the language which are not implemented are relatively minor, and are indicated in the program listings.

APPENDIX A - BENCHMARK PROGRAMS

Benchmark Program 1

```
300 PRINT "START"
400 FOR K=1 TO 1000
500 NEXT K
700 PRINT "END"
800 END
```

```
BEGIN
INTEGER A,K;
WRITE("START");
FOR K:=1 STEP 1 TO 1000 DO
   A:=0;
WRITE("END");
END
```

Benchmark Program 2

```
300 PRINT "START"
400 K=0
500 K=K+1
600 IF K<1000 THEN 500
700 PRINT "END"
800 END
```

```
BEGIN
INTEGER K;
WRITE("START");
K:=0;
WHILE K < 1000 DO
   K:=K+1;
WRITE("END");
END
```

Benchmark Program 3

```
300 PRINT "START"
400 K=0
500 K=K+1
510 A=K/K*K+K-K
600 IF K <1000 THEN 500
700 PRINT "END"
800 END
```

```
BEGIN
INTEGER A,K;
WRITE("START");
K:=0;
WHILE K<1000 DO
 BEGIN
  K:=K+1;
  A:=K/K*K+K-K;
 END;
WRITE("END");
END
```

Benchmark Program 4

```
300 PRINT "START"
400 K=0
500 K=K+1
510 A=K/2*3+4-5
600 IF K<1000 THEN 500
700 PRINT "END"
800 END
```

```
BEGIN
INTEGER A,K;
WRITE("START");
K:=0;
WHILE K<1000 DO
 BEGIN
  K:=K+1;
  A:=K/2*3+4-5;
 END;
WRITE("END");
END
```

Benchmark Program 5

```
300 PRINT"START"
400 K=0
500 K=K+1
510 A=K/2*3+4-5
520 GOSUB 820
600 IF K<1000 THEN 500
700 PRINT "END"
800 END
```

```
BEGIN
INTEGER A,K;
PROCEDURE DONOTHING;
 A:=0;
WRITE("START");
K:=0;
WHILE K<1000 DO
 BEGIN
  K:=K+1;
  A:=K/2*3+4-5;
  DONOTHING;
 END;
WRITE("END");
END
```

Benchmark Program 6

```
300 PRINT"START"
400 K=0
430 DIM M(5)
500 K=K+1;
510 A=K/2*3+4-5
520 GOSUB 820
530 FOR L-1 TO 5
540 NEXT L
600 IF K<1000 THEN 500
700 PRINT"END"
800 END
```

```
BEGIN
INTEGER A,K,L;
INTEGER ARRAY M[1:5];
PROCEDURE DONOTHING;
 BEGIN
  A:=0;
 END;
WRITE("START");
    K:=0;
WHILE K<1000 DO
 BEGIN
  K:=K+1;
  A:=K/2*3+4-5;
  DONOTHING;
  FOR L:=1 STEP 1 UNTIL 5 DO
   A:=0;
 END;
WRITE("END");
END
```

Benchmark Program 7

```
300 PRINT "START"
400 K=0
430 DIM M(5)
500 K=K+1
510 A=K/2*3+4-5
520 GOSUB 820
530 FOR L=1 TO 5
535 M(L)=A
540 NEXT L
600 IF K<1000 THEN 500
700 PRINT"END"
800 END
820 RETURN
```

```
BEGIN
INTEGER A,K,L;
INTEGER ARRAY M[1:5];
PROCEDURE DONOTHING;
 BEGIN
  A:=0;
 END;
WRITE("START");
K:=0;
WHILE K<1000 DO
BEGIN
 K:=K+1;
 A:=K/2*3+4-5;
 DONOTHING;
 FOR L:=1 STEP 1 UNTIL 5 DO
   M[L]:=A;
END;
WRITE("END");
END
```

APPENDIX B - COMPILER ERROR MESSAGES

AS	Function/Procedure on left hand side of assignment statement.
BP	Incorrect bound pair subtype (must be integer).
DE	Disk error; no corrective action can be taken in the program.
DD	Doubly declared identifier, label, variable etc.
IC	Invalid special character.
ID	Subtypes incompatible (decimal values can not be assigned to integer variables).
IO	Integer overflow.
IT	Identifier is not declared as a simple variable or function.
NG	No ALG file found.
NI	Subtype is not integer.
NP	No applicable production exists.
NS	Subtype is not string.
PC	Undeclared parameter.
SO	Stack overflow.
SI	Array subscript is not of subtype integer.
TD	Subtype has to be integer or decimal.
TM	Subtypes do not match or are incompatible.
TO	Symbol table overflow.
TS	Undeclared subscripted variable.
UD	Undeclared identifier.
UF	Undeclared file/function.
UP	Undeclared procedure.

V0 Varc table overflow. Possibly caused by too many long identifiers.

APPENDIX C - INTERPRETER ERROR MESSAGES

ERROR MESSAGES

AB	Array subscript out of bounds.
AZ	Attempt to allocate null decimal or string, default to 10 digits/characters.
CE	Disk file close error.
DC	Disk file create error.
DW	Disk file write error.
DZ	Division by zero, result set to 1.0.
EF	Disk end of file, no action specified.
IO	Integer overflow.
NI	No INT file found on directory.
OV	Overflow during decimal multiply.
RE	Attempt to read past end of record on blocked file.
RU	Attempt to random access a non-blocked file.
SL	Significant digits lost during decimal store, value set to 1.0.

WARNING MESSAGES

II	Invalid Console Input
IL	Non-significant digit lost during decimal store.
SO	Characters lost during string store.

APPENDIX D ALGOL-M LANGUAGE MANUAL

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

Braces {} indicate an optional entry.

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

Ellipses "..." indicate that the preceding item may be optionally repeated.

Reserved words are indicated by capital letters.

Reserved words and other special symbols must appear as shown.

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

ELEMENT:

arithmetic expression

FORMAT:

integer!decimal

variable

{(} arithmetic expression binary operator arithmetic expression {)}

{(} unary operator arithmetic expression {)}.

DESCRIPTION:

Operators in ALGOL-M have an implied precedence which is used to determine the manner in which operators and operands are grouped. A-B/C causes the result of B divided by C to be subtracted from A. In this case B is considered to be "bound" to the "/" operator instead of the "-" operator which causes the division to be performed first. The implied precedence binds operands to the adjacent operator of highest precedence. The implied precedence of operators is as follows:

unary -,+

**

*,/

+,-

Parentheses can be used to override the implied precedence in the same way as they are used in ordinary algebra. Thus the expression (A-B)/C will cause B to be subtracted from A and the result divided by C.

EXAMPLE:

```
(X + Y) * (Z * Y + X) ** 2
X + Y + Z * X * Y * Z / 5.456 + 1
```

ELEMENT:

ARRAY declaration

FORMAT:

INTEGER!DECIMAL!STRING {(expression)} ARRAY
identifier ... bound pair list {,identifier}

DESCRIPTION:

The array declaration dynamically allocates storage for arrays. The optional integer expression indicates the length of each array element. For strings, the maximum length is 255 characters and for decimals the maximum length is 18 digits. Integer lengths are not specified since storage adequate to represent all integer values between -16,384 and +16,384 is automatically allocated. Arrays are not automatically initialized to zero.

EXAMPLE:

```
INTEGER ARRAY X[0:5,0:5];
DECIMAL(10) ARRAY X,Y[3:6,5:10];
STRING ARRAY WORDS[Y+3:12];
```

ELEMENT:

assignment statement

FORMAT:

variable := {variable :=} ... expression

DESCRIPTION:

The expression is evaluated and stored into the variable. The types of permissible assignments are indicated by the following table:

		expression		
		integer	decimal	string
variable	integer	yes	no	no
	decimal	yes	yes	no
	string	no	no	yes

Multiple assignments are allowed with the expression being assigned to all of the listed variables.

EXAMPLE:

```
X := Y + Z;
Y[1] := Y[2] := 50;
```

ELEMENT:

balanced statement

FORMAT:

{label definition} simple statement

{label definition} IF boolean expression THEN balanced statement ELSE balanced statement

DESCRIPTION:

If the boolean expression is true, the balanced statement to the left of the ELSE is executed. If the boolean expression is false, the balanced statement to the right of the ELSE is executed.

EXAMPLE:

```
IF A < B THEN A := 1 ELSE A := 2;

IF B = C THEN
   BEGIN
     WRITE(B);
     B := B + 1;
   END
ELSE
   BEGIN
     WRITE(C);
     C := C + 1;
   END;
```

PROGRAMMING NOTE:

A semicolon is not allowed after the statement

immediately preceding an ELSE.

ELEMENT:

block

FORMAT:

BEGIN {declaration;} ... statement; ... END;

DESCRIPTION:

The block is the foundation of the ALGOL-M language. Each time a new block is entered new variables may be declared. These variables are unique in the sense that a variable X declared in two different blocks represents two different variables. All storage within a block is dynamic and allocated when the block is entered and de-allocated when the block is departed. A block can be used any place a simple statement can be used.

EXAMPLE:

```
BEGIN
X := 1;
Y := 2;
END;

IF X = Y THEN
  BEGIN
  X := 3;
  Y := 4;
  END;
```

PROGRAMMING NOTE:

Declarations may not appear in case blocks. The final END, which matches the initial program BEGIN, is not followed by a semicolon.

ELEMENT:

boolean expression

FORMAT:

NOT boolean expression

boolean expression OR boolean expression

boolean expression AND boolean expression

{(} expression =|<|>|>=|<=|<> expression {)}

DESCRIPTION:

Integer-integer, decimal-integer, decimal-decimal, integer-decimal, and string-string comparisons are allowed in ALGOL-M. For integer-decimal and decimal-integer comparisons the integer value is converted to a decimal value prior to comparison. The result of a comparison of numerical values is based on the size of the numbers. The result of a comparison of string values depends on a character-by-character comparison where the first instance of a non-equal character establishes the boolean result. The collating sequence of the ASCII character set is used for string comparisons. Generally, numbers are followed by upper case letters which are followed by lower case letters.

EXAMPLE:

```
X > Y OR Y < Z;

(X = Y) AND (Y = Z OR Z = 10);

IF NOT X = 1 THEN WRITE("HELLO");
```

ELEMENT:

bound pair list

FORMAT:

[expression : expression{,expression : expression} ...]

DESCRIPTION:

Expressions in the bound pair list must be of type integer and greater than or equal to zero. There can be no more than 255 dimensions.

EXAMPLE:

[1:7,0:5]

[3:6,x:y]

[y*3:z,1:12]

ELEMENT:

CASE statement

FORMAT:

```
CASE expression OF
     BEGIN
     statement; ...
     END;
```

DESCRIPTION:

The CASE statement allows the programmer to chose one of several statements to be executed. The statement chosen depends on the value of the integer expression. The first statement is executed if the expression evaluates to zero. If the value of the expression is greater than the number of statements in the case block, the resulting action is undefined.

EXAMPLE:

```
CASE X + Y OF
     BEGIN
     WRITE("CASE 0");
     WRITE("CASE 1");
     END;
```

ELEMENT:

CLOSE statement

FORMAT:

CLOSE identifier {,identifier} ...

DESCRIPTION:

This statement allows the programmer to explicitly close the file indicated. Closing a file results in the file being rewound (i.e., if it is reopened the file begins at the first record). Any number of files may be open at any one time. All files are implicitly closed at the end of the program.

EXAMPLE:

```
CLOSE FILE1, FILE2;
```

ELEMENT:

constant

FORMAT:

integer!decimal!string

DESCRIPTION:

A constant may be either an integer, decimal, or string constant. Integer constants are numbers with no decimal point ranging from -16,384 to +16,384. Decimal constants are numbers with a decimal point and may not exceed 18 digits in length. String constants may be composed of any combination of alphanumeric and special charaters and may be up to 255 characters in length. Strings entered from the console or disk may be either enclosed in quotation marks or delimited with blanks. Strings used as constants in the program must be enclosed in quotation marks.

EXAMPLE:

10

10.5678

"EXAMPLE ONE"

ELEMENT:

declaration

DESCRIPTION:

See FILE declaration, ARRAY declaration, simple declaration, procedure declaration, and function declaration.

ELEMENT:

expression

FORMAT:

boolean expression|arithmetic expression

DESCRIPTION:

See arithmetic expression and boolean expression.

ELEMENT:

FILE declaration

FORMAT:

FILE identifier {(expression)} {,identifier {(expression)}} ...

DESCRIPTION:

The identifiers used in the FILE declaration are file identifiers which reference actual file names. The actual file names may be assigned at compile-time or at run-time. The optional integer expression following the file identifier is used to specify the record length in bytes for blocked records.

EXAMPLE:

```
FILE TAPE1, TAPE2(128);
```

ELEMENT:

FOR statement

FORMAT:

{label definition} FOR assignment statement
{STEP expression} UNTIL expression DO simple statement

DESCRIPTION:

Execution of all statements within the simple statement are repeated until the indexing variable is greater than or equal to the value of the UNTIL expression. The indexing variable is incremented by the amount specified in the STEP expression and must be incremented by a positive amount. The UNTIL and STEP expressions are evaluated on each loop. If the optional STEP expression is omitted, a default value of one is used.

EXAMPLE:

```
FOR I := 1 STEP 1 UNTIL 10 DO
    X := Y;

FOR INDEX := X + Y UNTIL X * Y DO
    BEGIN
    A := A + B;
    WRITE(A);
    END;
```

ELEMENT:

function call

FORMAT:

identifier {(expression {,expression} ...)}

DESCRIPTION:

Functions may appear as primary elements in arithmetic or boolean expressions. Parameter passing is by value. Functions may be called recursively with no limit to the number of recursive calls allowed.

EXAMPLE:

```
X := RAND;
Y := SQRT(5.6);
C := FUNC * RND(2);
```

ELEMENT:

function declaration

FORMAT:

INTEGER!DECIMAL!STRING FUNCTION identifier
{(identifier {,identifier} ...)}
{declaration; ...} simple statement;

DESCRIPTION:

A function declaration may or may not include parameters. If parameters are included they must be declared before the simple statement which represents the body of the function. Parameters are passed by value and may be of type integer, decimal, or string. Functions return a value to the point of call. The value to be returned is assigned to the function name (which is used as a simple variable within the function) prior to the end of the function. Functions may be called recursively with no limit set as to the number of recursive calls which can be made. Variables may be declared within functions and are considered local to the function.

EXAMPLE:

INTEGER FUNCTION VALUE(X);

```
INTEGER X;
BEGIN
X := (X * 5) + (X * 2 );
VALUE := X;
END;
```

ELEMENT:

GOTO statement

FORMAT:

label definition GO TO identifier|integer

label definition GOTO identifier|integer

DESCRIPTION:

Execution continues at the statement labeled with the identifier or integer following the GOTO or GO TO statement.

EXAMPLE:

```
NEXT: GO TO 100;
100: GOTO NEXT;
```

PROGRAMMING NOTE:

GOTO statements can only be used to branch within the current block or to an outer block.

ELEMENT:

identifier

FORMAT:

letter {letter|number} ...

DESCRIPTION:

Identifiers begin with a letter and are continued with any alphanumeric characters. Although identifiers up to 255 characters may be used, only the first 31 characters are actually used to distinguish the identifiers.

EXAMPLE:

A

NAME

COUNTER1

ELEMENT:

IF statement

DESCRIPTION:

See balanced statement or unbalanced statement.

ELEMENT:

label definition

FORMAT:

identifier|integer :

DESCRIPTION:

Label definitions are optional on all balanced or unbalanced statements.

EXAMPLE:

```
FINISH:
100:
```

ELEMENT:

procedure call

FORMAT:

identifier {(expression {,expression} ...)}

DESCRIPTION:

Procedures can be called with or without parameters. Parameter passing is by value. Procedures can be called recursively with no limit set as to the number of recursive calls.

EXAMPLE:

```
COMPUTE;
COMPARE("AAA", WORD);
COUNT(1, 2, 3);
```

ELEMENT:

procedure declaration

FORMAT:

PROCEDURE identifier {(identifier {,identifier} ...)}
{declaration; ...} simple statement

DESCRIPTION:

A procedure declaration may or may not include parameters. If parameters are included they must be declared before the simple statement which represents the body of the procedure. Parameters are passed by value and may be of type integer, decimal or string. Procedures do not return a value to the point of call. Procedures can be called recursively. Procedures are considered separate blocks within which local variables may be declared.

EXAMPLE:

```
PROCEDURE OUTPUT;
   WRITE ("HELLO");

PROCEDURE COMPARE(X,Y);
   INTEGER X,Y;
   BEGIN
   WRITE("THE LARGEST INTEGER IS ");
   IF X > Y THEN
      WRITE(X);
   ELSE
      WRITE(Y);
   END;
```

ELEMENT:

READ statement

FORMAT:

READ (variable {,variable} ...) {ONENDFILE block}

DESCRIPTION:

If the form of the READ statement is READ(, then the input device is the console. Otherwise a file option must be specified and the input device is the disk. A READ statement reads one or more variables at a time. The optional ONENDFILE s[illegible] indicates action to be taken when the end of the specified file is reached.

EXAMPLE:

```
READ(WORDONE, X, VALUE2);
READ FILE3 (WORDONE, X, VALUE2);
```

PROGRAMMING NOTE:

The ONENDFILE section is curently not implemented.

ELEMENT:

reserved word list

FORMAT:

letter {letter} ...

DESCRIPTION:

The following words are reserved by ALGOL-M and may not be used as identifiers:

AND	ARRAY	BEGIN	CASE
CLOSE	DECIMAL	DO	ELSE
END	EXTERNAL	FILE	FUNCTION
GO	GOTO	IF	INITIAL
INTEGER	NOT	OF	ONENDFILE
OR	PIC	PROCEDURE	READ
STEP	STRING	TAB	THEN
TO	UNTIL	WHILE	WRITE
WRITEON			

Reserved words must be preceeded and followed by either a special character or a space. Spaces may not be embedded within reserved words.

ELEMENT:

simple statement

FORMAT:

block|assignment statement|for statement|
case statement|close statement|goto statement|
while statement|read statement|write statement|
procedure call|identifier

DESCRIPTION:

All ALGOL-M statements are free form and must be separated by semicolons.

ELEMENT:

simple declaration

FORMAT:

INTEGER!DECIMAL!STRING {(identifier!integer)}
identifier {,identifier} ...

DESCRIPTION:

Simple integer variables may be any value between -16,384 and +16,384. Simple decimal variables can be specified as any length from one to 18 digits with a default length of 10 digits. Simple string variables can be specified as any length from one to 255 characters with a default length of 10 characters.

EXAMPLE:

```
INTEGER X;
DECIMAL(15) X,Y;
STRING(33) WORDONE, WORDTWO, WORDTHREE;
```

ELEMENT:

special characters

DESCRIPTION:

The following special characters are used by ALGOL-M:

(	open parenthesis
)	close parenthesis
*	asterisk
+	plus
-	minus
:	colon
;	semicolon
<	less-than
>	greater-than
=	equal
,	comma
[	open bracket
]	close bracket
:=	assigned equal
**	exponentiation
%	percentage

Any special character in the ASCII character set may appear in a string. Special characters, other than those listed above, will cause an error condition if used outside of a string.

ELEMENT:

TAB expression

FORMAT:

TAB expression

DESCRIPTION:

TAB is optionally used in a WRITE statement to cause spacing on the output line. The amount of spacing is specified by the integer expression following TAB.

EXAMPLE:

```
WRITE("NEXT NAME", TAB 5, NAME[I]);
```

ELEMENT:

unbalanced statement

FORMAT:

{label definition} IF boolean expression THEN statement

{label definition} IF boolean expression THEN balanced statement ELSE unbalanced statement

DESCRIPTION:

Unlike the balanced statement that will always have a balanced statement on either side of the ELSE in an IF THEN ELSE structure, an unbalanced statement may not even include the ELSE portion of the statement.

EXAMPLE:

```
IF S > Y THEN WRITE(X);

IF X < Y THEN
   IF Z > Y THEN
      WRITE(Z)
   ELSE
      WRITE(X);
```

PROGRAMMING NOTE:

A semicolon is not allowed after the statement immediately preceding an ELSE.

ELEMENT:

variable

FORMAT:

identifier {[bound pair list]}

DESCRIPTION:

A variable in ALGOL-M may be simple or subscripted and of type INTEGER, DECIMAL, or STRING.

EXAMPLE:

```
X
VALUE[2]
Z[1,X * Y]
```

ELEMENT:

WHILE statement

FORMAT:

WHILE boolean expression DO simple statement

DESCRIPTION:

WHILE statements continue executing the simple statement following the DO for as long as the boolean expression is true.

EXAMPLE:

```
WHILE I > 0 DO
      I := I - 1;

WHILE X > 5 AND Y <> 8 DO
      BEGIN
        X := X / 3;
        WRITE(X);
      END;
```

ELEMENT:

WRITE statement

FORMAT:

WRITE|WRITON (file option)
(expression|tab expression|pic definition|string
{,expression|tab expression|pic definition|string} ...)

DESCRIPTION:

The WRITE option indicates the output will start printing on a new line, while the WRITEON option will continue printing on the same line. If the form of the statement is WRITE(or WRITEON(, the output device is the console. Otherwise, a file option must be specified and the output device is the disk.

EXAMPLE:

```
WRITE(X);
WRITE("THE NUMBER IS",X + Y);
WRITE("ANSWER", TAB 5, X * Y);
```

PROGRAMMING NOTE:

The PIC definition is not currently implemented.

APPENDIX E ALGOL-M LANGUAGE STRUCTURE

In the following sections, the syntax of the language will be listed in BNF notation followed by the semantic actions (offset with asterisks) associated with that production. The description will be given in terms of compiler data structures and the ALGOL-M machine code generated. Items enclosed in brackets and separated by slants are alternative semantic actions. N/A indicates no action. This notation is similar to that used in REF 6 and REF 10.

```
1 <program> ::= <block> -'-
    *<block>; XIT; {eof indicator}

2 <block> ::= <block head> <block end>
    *<block head>; <block end>

3 <block head> ::= <block head> <declaration> ;
    *<block head>; <declaration>

4                 | <begin>
    *<block>

5 <begin> ::= begin
    *BLT

6 <block end> ::= <block body> ; end
    *BLD

7 <block body> ::= <statement>
    *<statement>

8 <block body> ; <statement>
    *<block body>; <statement>

9 <declaration> ::= <file declaration>
    *<file declaration>
```

```
10              ; <simple declaration>
    *<simple declaration>

11              ; <simple declaration> <initial option>
    *<simple declaration>; <initial option>

12              ; <array declaration>
    *<array declaration>

13              ; <array declaration> <initial option>
    *<array declaration>; <initial option>

14              ; <subprogram declaration>
    *<subprogram declaration>

15              ; <external declaration>
    *{Not implemented}

16 <simple declaration> ::= <declaration head>
                            <identifier>
    *<declaration head>: LIT {identifier address}
    *[ALD/ALS/ N/A for integers]

17 <initial option> ::= <initial head> <constant> )
    *<initial head>; <constant>

18 <initial head> ::= initial (
    N/A


19                      ; <initial head> <constant> ,
    *<initial head>; <constant>


20   <declaration head> ::= <declaration type>
    *<declaration type>

21                    ! <declaration head> <identifier>
21                          ,
    *<declaration head>; LIT {identifier address};
    *[ALD/ALS/ N/A for integers]


22   <declaration type> ::= string
    *[M1 {string default size}]

23                    ! string <size option>
    *<size option>

24                    ! integer
    N/A

25                    ! decimal
    *[M1 {decimal default size}]
```

```
26                    ! decimal <size option>
    *<size option>

27   <size option> ::= ( <variable> )
    *LITLOAD {variable address};

28                    ! ( <integer> )
    *<integer>

29   <statement> ::= <balanced statement>
    *<balanced statement>

30                  ! <unbalanced statement>
    *<unbalanced statement>

31   <balanced statement> ::= <simple statement>
    *<simple statement>

32                      ! <if clause> <true part> else
32                              <balanced statement>
    *<if clause>; <true part>; <balanced statement>

33                      ! <label definition>
33                              <balanced statement>
    *<label definition>; <balanced statement>

34 <unbalanced statement> ::= <if clause> <statement>
    *<if clause>; <statement>

35                        ! <if clause> <true part>
35                         else <unbalanced statement>
    *<if clause>; <true part>; <unbalanced statement>

36                              ! <label definition>
36                                <unbalanced statement>
    *<label definition>; <unbalanced statement>

37   <true part> ::= <balanced statement>
    *<balanced statement>; BRS {address of end of
    *current statement}

38   <label definition> ::= <identifier> :
    N/A

39                          ! <integer> :
    N/A

40   <simple statement> ::= <block>
    *<block>
```

```
41                          | <assignment statement>
    *<assignment statement>

42                          | <for statement>
    *<for statement>

43                          | <while statement>
    *<while statement>

44                          | <read statement>
    *<read statement>

45                          | <write statement>
    *<write statement>

46                          | <case statement>
    *<case statement>

47                          | <go to statement>
    *<go to statement>

48                          | <close statement>
    *<close statement>

49                          | <procedure call>
    *<procedure call>; PRO;

50                          | <identifier>
    *IM2 {subroutine address}; PRO; POP;


51 <assignment statement> ::= <left part> <expression>
    *<left part>; <expression>; [SIT/SID]

52                              | <left part>
52                                <assignment statement>
    *<left part>; <assignment statement>


53  <left part> ::= <variable> :=
    *<variable>; [LIT/LITLOAD]


54  <expression> ::= <arithmetic expression>
    *<arithmetic expression>

55                  | <if expression> <expression>
    N/A


56  <arithmetic expression> ::= <term>
    *<term>


57                          | <arithmetic expression> +
```

```
57                                          <term>
    *<arithmetic expression>; <term>; [ADI/ADD]

58                          ! <arithmetic expression> -
58                                          <term>
    *<arithmetic expression>; <term>; [SBI/SBD]

59                          ! <arithmetic expression>
59                                          ! ! <term>
    *<arithmetic expression>; <term>; CAT;

60                          ! - <term>
    *<term>; NEG

61                          ! + <term>
    *<term>


62   <term> ::= <primary>
    *<primary>

63             ! <term> * <primary>
    *<term>; <primary>; [MLI/MLD]

64             ! <term> / <primary>
    *<term>; <primary>; [DVI/DVD]

65   <primary> ::= <primary element>
    *<primary element>

66                 ! <primary> ** <primary element>
    *<primary>; <primary element>; IXP


67   <primary element> ::= <variable>
    *LITLOAD {address of simple variable}/ N/A for
    *subscripted variables/ IM2 {subrountine
    *address in code area}; PRO

68                          ! <constant>
    *<constant>

69                          ! <procedure call>
    *<procedure call>

70                          ! ( <assignment statement> )
    *<assignment statement>

71                          ! ( <expression> )
    *<expression>


72   <constant> ::= <integer>
    *INT; {constant}
```

```
73                  ¦ <decimal>
    *DEC; {constant}

74                  ¦ <string>
    *STR; {constant}


75   <variable> ::= <identifier>
    N/A

76                  ¦ <subscripted variable>
    *<subscripted variable>


77   <file declaration> ::= <file head> <file name>
    *{not implemented}


78   <file head> ::= file
    *{not implemented}

79                   ¦ <file head> <file name> ,
    *{not implemented}


80   <file name> ::= <string> <length option>
    *{not implemented}

81                   ¦ <string>
    *{not implemented}

82                   ¦ <identifier> <length option>
    *{not implemented}

83                   ¦ <identifier>
    *{not implemented}


84   <length option> ::= [ <identifier> ]
    *{not implemented}

85                       ¦ [ <integer> ]
    *{not implemented}


86   <array declaration> ::= <array list>
86                           <bound pair list>
    *<array list>; <bound pair list>;
    *LIT {# of arrays(m)}; LIT {type of
    *array};
    *followed by:
              (1) LIT {array location}
              (2) m=m-1 if m=0 then halt else step(1)
```

```
87   <array list> ::= <array head> <identifier>
    *<array head>

88   <array head> ::= <declaration type> array
    *<declaration type>

89                    | <array head> <identifier> ,
    *<array head>


90   <bound pair list> ::= <bound pair head>
90                          <bound pair> ]
    *<bound pair head>; <bound pair>


91   <bound pair head> ::= [
    N/A

92                    | <bound pair head> <bound pair> ,
    *<bound pair head>; <bound pair>

93   <bound pair> ::= <expression> : <expression>
    *<expression>; <expression>

94   <subscripted variable> ::= <subscript head>
94                              <expression> ]
    *<subscript head>; <expression>;
    *LITLOAD <array address>; SUB

95   <subscript head> ::= <identifier> [
    N/A

96                    | <subscript head> <expression> ,
     *<subscript head>; <expression>

97   <go to statement> ::= <go to> <identifier>
    *DCB {# of blocks to decrement}/NOP;NOP
    *followed by BRS {branch address}

98                          | <go to> <integer>
    *DCB {# of blocks to decrement}/NOP;NOP
    *followed by BRS {branch address}

99   <go to> ::= go to
    N/A

100                | goto
    N/A

101  <read statement> ::= <read head> <variable> )
    *<read head>; LIT/LITLOAD {address of variable};
    *RDT/RCT; STD/SDD/SSD
    *if console I/O then ECP
```

```
102   <read head> ::= read (
     *RCN

103                   ! read <file option> ( ·
     *{not implemented}

104                   ! <read head> <variable> ,
     *<read head>; same as production 101

105   <write statement> ::= <write head> <expression> )
     *<write head>; <expression>; [WIC/WDC/WSC/WID
     *WDD/WID]

106                    ! <write head> <tab expression> )
     *{not implemented}

107                    ! <write head> <pic definition> )
     *{not implemented}

108   <write head> ::= write (
     *DMP

109                    ! write <file option> (
     *{not implemented}

110                    ! writeon (
     N/A

111                    ! writeon <file option> (
     *{not implemented}

112                    ! <write head> <expression> ,
     *<write head>; <expression>; [LIT/LITLOAD]

113                    ! <write head> <tab expression> ,
     *{not implemented}

114                    ! <write head> <pic definition> ,
     *{not implemented}

115   <file option> ::= <identifier>
     N/A

116                     ! <identifier> <rec option>
     N/A

117                     ! <string>
     N/A

118                     ! <string> <rec option>
     N/A

119   <rec option> ::= , <identifier>
     N/A
```

```
120                    | , <integer>
     N/A

121   <pic definition> ::= <pic head> <pic list> )
     *{not implemented}

122   <pic head> ::= pic <string>
     *{not implemented}

123                  | pic <identifier>
     *{not implemented}

124   <pic list> ::= ( <expression>
     *{not implemented}

125                  | <pic list> , <expression>
     *{not implemented}

126   <tab expression> ::= tab <expression>
     *{not implemented}

127   <if clause> ::= if <boolean expression> then
     *<boolean expression>; BSC {branch address}

128   <if expression> ::= <if clause> <expression> else
     *{not implemented}

129   <boolean expression> ::= <boolean term>
     *<boolean term>

130                            | <boolean expression> or
130                              <boolean term>
     *<boolean expression>; <boolean term>; BOR

131   <boolean term> ::= <boolean primary>
     *<boolean primary>

132                      | not <boolean primary>
     *<boolean primary>; NOTO

133                      | <boolean term> and
133                        <boolean primary>
     *<boolean term>; <boolean primary> ANDO

134   <boolean primary> ::= <logical expression>
     *<logical expression>

135                        | ( <boolean expression> )
     *<boolean expression>

136   <logical expression> ::= <expression> <relation>
136                              <expression>
     *<expression>; <expression>; [string, integer,
     *or decimal relational operator]
```

```
137   <relation> ::= =
     N/A

138                  | <
     N/A

139                  | >
     N/A

140                  | <comp>
     N/A

141   <comp> ::= < >
     N/A

142              | < =
     N/A

143              | > =
     N/A

144 <while statement> ::= <while clause> <do statement>
     *<while clause>; <do statement>; BRS
     *{branch location}

145   <while clause> ::= <while> <boolean expression>
     *<boolean expression>; BSC {branch address}

146   <while> ::= while
     N/A

147  <for statement> ::= <for clause> <step expression>
147                      <until clause> <do statement>
     *<for clause>; <step expression>; <until clause>;
     *<do statement>; BRS {branch address}

148   <for clause> ::= for <assignment statement>
     *<assignment statement>; BRS {branch address};
     *LIT {counter variable}; LITLOAD {counter variable}

149   <step expression> ::= step <expression>
     *<expression>; ADT; STD; LITLOAD {counter variable}

150   <until clause> ::= <until non-term> <expression>
     *<until non-term>; <expression>; LEQ; BSC
     *{branch address}

151   <until non-term> ::= until
     *[1M1 {one}; ADT; STD; LITLOAD {counter variable}/
       N/A if there is a step expression]

152   <do statement> ::= do <simple statement>
     *<simple statement>
```

```
153   <close statement> ::= close <identifier>
     *{not implemented}

154                     | <close statement> , <identifier>
     *{not implemented}

155   <subprogram declaration> ::= <subprogram heading>
155                                <simple statement>
     *<subprogram heading>; <simple statement>; LIT
     *{subroutine PRT address}; UNS; RTN; LIT(SBP);
     *LIT 0; SID;

156   <subprogram heading> ::= <function heading>
     *<function heading>

157                            | <procedure heading>
     *<procedure heading>

158   <function heading> ::= <paramless function>
     *<paramless function>

159                          | <function & params>
     *<function params>; SV2

160   <procedure heading> ::= <paramless proc>
     *<paramless proc>

161                           | <proc & params>
     *<proc & params>; SV2

162   <paramless function> ::= <declaration type>
162                              function <identifier> ;
     *<declaration type>

163   <function & params> ::= <function head>
163                             <identifier> ) ;
     *<function head>

164                           | <function & params>
164                             <declaration> ;
     *<function params>; <declaration>

165   <function head> ::= <declaration type> function
165                         <identifier> (
     *<declaration type>

166                  | <function head> <identifier> ,
     *<function head>

167   <paramless proc> ::= procedure <identifier> ;
     *IM1 {parameter count}; IM1 {local variable &
     *parameter offset}; LIT {base of PCB}; SAV; BLI
     *if this is a non-integer function then
     *LITLOAD {length of return value};
     *LIT {PRT address of function}; [ALD/ALS]
```

```
168   <proc & params> ::= <procedure head> <identifier>
168                         ) ;
     *<procedure head>; same as production 167

169                       | <proc & params> <declaration>
169                         ;
     *<proc params>; <declaration>

170   <procedure head> ::= procedure <identifier> (
     *BLI;

171                    | <procedure head> <identifier> ,
     N/A

172 <procedure call> ::= <call heading> <expression> )
     *<call heading>; <expression>; IM2
     *{address of subroutine in code area}; PRO

173   <call heading> ::= <identifier> (
     N/A

174                      | <call heading> <expression> ,
     *<expression>

175   < declaration> ::= <declaration type> external
175                        function <external list>
     *{not implemented}

176                               | external procedure
176                                 <external list>
     *{not implemented}

177   <external list> ::= <identifier>
     *{not implemented}

178                    | <external list> , <identifier>
     *{not implemented}

179   <case statement> ::= <case heading> <case block>
     *<case heading>; <case block>;
     *do n times(n=number of statements in case block)
       BRS {address of case statement(n)}

180   <case heading> ::= case <expression> of
     *<expression; LIT {3}; MPI; IM2
     *{address of end of case block}; SBR; BFA

181   <case block> ::= begin <case block body> ; end
     N/A

182   <case block body> ::= <statement>
     *<statement>; BRS {address of end of case block}
183                      | <case block body> ; <statement>
     *same as production 182
```

ALGOL-M PROGRAM LISTINGS

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

    /*   ***************************************************************   */
    /*   **********          system literals          **********          */
    /*   ***************************************************************   */

declare false          literally '0',
        true           literally '1',
        lit            literally 'literally',
        bdos  lit '5h',      /* entry point to disk operating system */
        startbdos address initial(6h), /*addr of ptr to top of bdos */
        max based startbdos address,
        boot           lit '0',          /* exit to return to operating system */
        pstacksize     lit '48',                /* stack sizes for parser */
        intrecsize     lit  '128',
        dcl            lit 'declare',
        proc           lit         'procedure',
        fileeof        lit  '1',
        rfile          lit  '20',
        identsize      lit  '32',
        addr           lit         'address',
        forever        lit 'while true',
        varcsize       lit '100',
        indexsize      lit 'address',
        statesize      lit 'address',
        maxoncount     lit '25',
        cr             lit '13',
        lf             lit '0ah',
        stringdelim    lit  '22h',
        questionmark   lit  '3fh',
        tab            lit  '09h',
        colin          lit  '3ah',
        comment        lit  '0',
        conbuffsize    lit  '82',
        eolchar        lit '0dh',
        hashtblsize    lit  '64',
        sourcerecsize  lit  '128',
        hashmask       lit  '63',
        contchar       lit  '5ch',
        eoffiller      lit '1ah',
        percent        lit '25h';


declare maxrno    literally '132',/* max read count */
        maxlno    literally '190',/* max look count */
        maxpno    literally '190',/* max push count */
        maxsno    literally '373',/* max state count */
        starts    literally '1',/* start state */
        prodno    literally '183',/* number of productions */
        semic     literally '8',/* semicolon */
        colonc    literally '13',/* colon */
        doc       literally '18',/* do */
        eofc      literally '24',/* eof */
        endc      literally '26',/* end */
        string    literally '49',/* string */
        decimal   literally '52',/* decimal */
        integerc  literally '53',/* integer */
        procc     literally '54',/* procedure */
        identifier lit '55',/*identifier*/
        termno    literally '55';/* terminal count */

declare sbloc              address initial(80h),
        sourcebuff based sbloc(sourcerecsize) byte,
        sourceptr          byte    initial(sourcerecsize),
        buffptr            byte    initial(255),
        errorcount         address initial(0),
        linebuff(conbuffsize) byte,
        lineptr            byte    initial(0),
```

```
        lineno              address,
        pass1               byte     initial(true),
        pass2               byte     initial(false),
        nointfile           byte     initial(false),
        rfcbaddr            address initial(5ch),
        rfcb  based         rfcbaddr(33) byte,
        wfcb(33)            byte initial(0,'        ','ain',0,0,0,0),
        cursourcerecsize    byte initial(sourcerecsize),
        nolook              byte,
        production          byte,
        arr$loc(5)          address,
        arr$num             byte,
        sub$proc$loc        address,
        sub$proc$var$num    byte,
        arr$dim             byte,
        diskoutbuff(intrecsize) byte;

   /* the following global variables are used by the scanner   */

declare token       byte,        /* type of token just scanned */
        hashcode    byte,        /* has value of current token */
        nextchar    byte,        /* current character from getchar */
        accum(identsize)     byte,     /* holds current token */
        cont        byte;        /* indicates accum was full, still more */


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

declare base        address, /* base of current entry */
        hashtable(hashtblsize) address,
        sbtbltop    address, /*current top of symbol table*/
        sbtbl       address,
        ptr based base byte, /*first byte of entry */
        aptraddr    address, /*utility variable to access table*/
        addrptr based aptraddr address,
        byteptr based aptraddr byte,
        printname   address, /*set prior to lookup or enter*/
        symhash     byte,
        prev$blk$level(12) byte,
        prev$index  byte initial(255),
        step$flag   byte,
        blk$cnt     byte initial(0),
        blk$level   byte initial(1);

declare read1 data(0,39,12,15,53,55,2,49,52,53,53,5,8,8,19,20,26,27,31
     ,34,35,39,40,42,43,48,53,55,19,20,26,27,31,34,35,39,40,42,43,48,53
     ,55,55,53,55,55,15,33,55,23,2,3,9,20,28,49,52,53,55,55,2,3,9,20,49
     ,52,53,55,49,55,2,3,9,20,49,52,53,55,2,49,55,55,2,49,55,2,2,2,49,55
     ,11,14,54,51,55,13,7,4,55,55,2,13,14,2,14,14,2,8,11,7,11,7,11,11,2,8
     ,4,6,10,24,53,55,16,7,11,2,7,11,17,7,11,7,49,55,11,19,20,27,31,34,35
     ,39,40,42,43,48,53,55,32,35,14,8,19,20,27,31,33,34,35,39,40,42,43,44
     ,45,47,48,50,53,54,55,4,11,1,12,15,21,7,11,7,11,4,11,1,7,12,15,32,7
     ,13,36,2,3,9,20,29,30,49,52,53,55,8,8,8,2,2,2,25,39,49,52,53,18,11
     ,55,33,44,45,47,50,54,7,11,55,7,11,8,11,41,55,38,50,51,46,22,37,7,22
     ,19,27,31,34,35,39,40,42,43,48,55,7,3,5,9,0);

declare look1 data(0,12,15,0,15,0,2,0,2,0,11,0,14,0,8,17,32,0,2,14,0,14
     ,0,11,0,14,0,11,0,11,0,2,14,17,0,6,10,0,6,10,0,6,10,0,6,10,0,6,10,0
     ,6,10,0,16,0,16,0,16,0,17,0,11,0,25,0,25,0,11,0,11,0,33,44,45,47,50
     ,54,0,11,0,55,0,46,0,33,44,45,47,50,54,0,32,0,22,0,46,0,3,5,9,0);

declare apply1 data(0,0,1,0,0,0,0,0,131,145,0,0,149,0,0,0,0,44,0,14,15
     ,40,93,0,37,93,143,0,37,143,0,0,0,27,150,0,3,4,32,95,0,0,3,4,6,22,24
     ,35,36,39,86,94,95,112,122,130,134,136,137,0,0,7,8,10,16,17,0,11,18
     ,0,26,0,124,0,3,4,5,14,15,27,32,37,40,93,95,96,101,143,150,0,0,0,0
     ,63,0,0,0,0,0,0,0,0,0,0,0,0,0,42,46,0,61,0,0,0,0,0,14,15,37,40
     ,93,101,143,0,0,3,79,0,30,0,31,33,0,0,0,0,0,0,0,0,0,0,0,0,125,0,0,0
     ,0,0,0,0,0,0,0,0,0,14,15,27,37,40,93,101,143,150,0,0,0,52,0,0,0,0,0
     );

declare read2(254) address initial
```

```
            (0,195,331,332,54,161,4,264,263,262,160,10,353,358,28
    ,29,196,32,36,290,38,195,41,336,42,46,53,159,28,29,371,32,36,290,38
    ,195,41,336,42,46,53,159,344,310,309,368,333,55,70,289,3,7,16,29,33
    ,264,263,262,160,161,3,7,16,29,264,263,262,160,312,313,4,7,16,29,264
    ,263,262,160,292,157,162,343,298,157,162,3,298,300,157,162,20,25,57
    ,52,69,229,218,275,62,367,363,228,285,363,285,285,360,357,279,12,356
    ,13,361,211,355,352,274,11,18,191,288,287,26,207,209,6,311,22,243
    ,291,294,217,158,163,269,28,29,32,36,290,38,195,41,336,42,46,53,159
    ,37,164,281,14,28,29,32,36,268,290,38,195,41,336,42,155,156,214,46
    ,49,53,56,159,280,282,153,134,327,370,295,302,362,364,284,286,153
    ,261,154,327,318,261,24,39,4,7,16,29,34,35,264,263,262,160,193,359
    ,354,293,299,301,31,40,264,263,262,27,21,65,268,155,156,214,49,56
    ,297,304,66,296,303,13,19,341,163,278,50,51,45,30,317,325,30,28,32
    ,36,290,38,195,41,336,42,46,106,260,8,9,17,0);

declare look2(101) address initial
                (0,2,2,328,23,329,43,212,44,215,47,307,48,271,240,265
    ,240,58,59,59,265,60,265,61,305,63,273,64,277,67,206,68,68,265,240
    ,71,71,246,72,72,251,73,73,250,74,74,247,75,75,248,76,76,249,80,252
    ,81,253,82,254,88,257,92,267,120,319,121,320,127,366,128,365,131,131
    ,131,131,131,131,351,139,238,210,142,144,202,145,145,145,145,145,145
    ,349,227,219,147,333,149,290,152,152,152,244);

declare apply2(177) address initial
                (0,0,77,230,101,194,192,100,115,116,114,189,201,203
    ,124,141,184,216,219,198,373,372,224,197,222,187,223,219,225,226,220
    ,97,143,342,345,221,151,151,338,242,231,95,108,110,314,315,283,316
    ,104,339,326,109,241,105,106,245,107,340,111,103,190,168,170,172,169
    ,171,167,174,175,173,256,255,83,258,176,176,90,87,87,87,87,87,87,87
    ,176,89,87,87,87,257,199,91,177,272,270,185,99,98,276,137,192,266
    ,134,237,78,234,96,235,113,118,119,117,306,308,132,84,85,135,93,93
    ,93,93,93,93,93,94,130,148,188,146,179,178,323,322,321,324,86,330
    ,233,126,79,232,112,140,125,136,337,334,183,204,150,346,347,348,186
    ,129,350,182,133,239,239,239,239,239,239,239,239,239,259,122,205,181
    ,180,236,123,369,138);

declare index1 data(0,1,2,30,70,4,70,6,6,11,6,6,12,13,14,28,6,6,6,42,43
    ,45,70,46,70,47,6,238,49,50,50,60,59,60,68,70,70,135,78,70,135,81,82
    ,85,85,86,87,90,91,92,93,94,99,95,96,97,98,99,100,103,105,90,106,91
    ,108,109,111,113,103,114,116,117,117,117,117,117,117,119,120,50,122
    ,122,122,123,125,126,70,128,128,129,131,132,134,135,70,70,59,148,149
    ,150,151,152,171,173,176,177,179,181,183,187,188,189,190,191,201,202
    ,203,204,205,206,207,207,70,208,209,212,212,213,213,214,70,215,221
    ,223,70,224,70,70,226,227,228,229,230,135,233,215,234,234,236,233
    ,238,249,250,1,4,6,8,10,12,14,18,21,23,25,27,29,31,35,38,41,44,47,50
    ,53,55,57,59,61,63,65,67,69,71,78,80,82,84,91,93,95,97,1,2,4,4,5,6,7
    ,7,8,8,8,8,8,8,8,11,12,14,14,13,15,16,16,16,16,16,17,17,19,19,24,24
    ,24,28,28,28,31,32,32,33,33,33,33,33,33,33,33,33,33,33,36,36,41,42
    ,42,60,60,60,60,60,60,61,61,61,67,67,70,70,70,70,70,72,72,72,74,74
    ,90,91,91,92,92,92,92,93,93,95,96,97,97,98,99,99,100,101,102,102,103
    ,103,104,104,105,106,106,106,107,107,107,108,108,108,108,108,108,108
    ,109,109,109,109,112,112,114,115,115,116,116,117,118,126,127,127,130
    ,130,130,132,132,135,136,136,136,136,137,137,137,138,139,140,141,142
    ,143,144,145,146,148,148,149,150,150,151,151,152,152,153,154,154,155
    ,155,156,157,157,158,158,159,169,169,170,170,171,171,173,174,175,176
    ,176);

declare index2 data(0,1,2,9,8,2,8,5,5,1,5,5,1,1,14,14,5,5,5,1,2,1,8,1,8
    ,2,5,11,1,9,9,8,1,8,2,8,8,13,3,8,13,1,3,1,1,1,3,1,1,1,1,1,1,1,1,1,1
    ,1,3,2,1,1,2,1,1,2,2,1,2,2,1,2,2,2,2,2,2,1,2,9,1,1,1,2,1,2,8,1,1,2,1
    ,2,1,13,8,8,1,1,1,1,1,19,2,3,1,2,2,2,4,1,1,1,1,10,1,1,1,1,1,1,1,1,8
    ,1,3,1,1,1,1,1,8,6,2,1,8,2,8,8,1,1,1,1,3,13,1,6,2,1,2,1,11,1,3,3,2,2
    ,2,2,2,4,3,2,2,2,2,2,4,3,3,3,3,3,3,2,2,2,2,2,2,2,2,2,7,2,2,2,7,2,2,2
    ,4,1,1,2,0,0,2,0,2,0,0,1,0,1,0,0,1,2,1,2,0,2,0,1,0,0,1,2,2,0,0,0,3,1
    ,1,3,1,0,1,1,0,0,0,0,0,0,0,0,0,0,1,1,1,0,1,0,2,2,3,1,1,0,2,2,0,2,0
    ,0,0,2,2,0,0,0,0,0,1,0,2,1,0,1,0,2,2,1,1,1,2,2,0,2,2,2,1,2,1,1,1,0,2
    ,1,2,2,2,2,2,1,2,1,2,2,2,2,0,1,0,1,1,1,2,1,1,1,2,1,2,2,0,2,0,1,2,0,2
    ,2,0,0,0,0,1,1,1,1,1,0,3,1,1,1,0,1,1,2,1,0,0,0,0,0,0,3,3,2,3,2,2,3,2
    ,2,2,2,1,2,3,2,0,2,1,2,3,0,2);
```

```
/*****************************************************/
/*                  global procedures                */
/*****************************************************/

mon1: procedure(f,a);
     declare     f byte,
     a address;
     go to bdos;
end mon1;

mon2: procedure (f,a) byte;
     declare f byte, a address;
     go to bdos;
     end mon2;

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

move: procedure (a,b,l);
     /* moves from a to b for l bytes (l < 255) */
     declare (a,b) address,
     (s based a, d based b,l) byte;
     do while (l:=l - 1) <> 255;
          d=s;  b=b + 1;  a =a + 1;
     end;
     end move;

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

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


printchar: procedure(char);
     declare char byte;
     call mon1(2,char);
end printchar;

print: procedure(a);
      declare a address;
      call mon1(9,a);
end print;

diskerr: procedure;
    call print(.'de     $');
    goto boot;
end diskerr;

open@sourcefile: procedure;
    call move(.'alg',rfcbaddr+9,3);
    rfcb(32) = 0;
    if mon2(15,rfcbaddr) = 255 then
         do;
         call print(.'ns $');
         go to boot;
         end;
end open@sourcefile;

close@int@file: procedure ;
     /* closes a file */
    if mon2(16,.wfcb) = 255 then
```

```
        call diskerr;
end close$int$file;

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

rewind$source$file:proc;
    /*cp/m does not require any
      action prior to reopening*/
     return;
end rewind$source$file;

read$source$file:proc byte;
     declare dcnt byte;
     if(dcnt:=mon2(rfile,rfcbaddr)) > fileeof then
     call diskerr;
     return dcnt;
end read$source$file;

write$int$file: procedure;
    if nointfile then
       return;
    call mon1(26,.diskoutbuff);
    if mon2(21,.wfcb) <> 0 then
        call diskerr;
    call mon1(26,80h); /* reset dma address */
end write$int$file;

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

printdec: procedure(value);
    declare value address, i byte, count byte;
    declare dec1(4) address initial(1000,100,10,1);
    declare flag byte;
    flag = false;
    count = 30h;
    do i = 0 to 3;
      do while value >= dec1(i);
           value = value - dec1(i);
           flag= true;
           count = count + 1;
       end;
       if flag or (i>= 3) then
           call printchar(count);
       else
           call printchar(' ');
     end;
     return;
end printdec;

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

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

```
emit:proc(objcode);
   declare objcode byte;
   if(buffptr:=buffptr+1) >= intrecsize then /*write to disk*/
     do;
       call write$int$file;
       buffptr=0;
     end;
   diskoutbuff(buffptr)=objcode;
end emit;

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

listline: procedure(length);
    declare (length,i) byte;
    call print$dec(lineno);
    call print$dec(prev$index+1);
    call print$char(' ');
    do i = 0 to length;
       call printchar(linebuff(i));
    end;
    call crlf;
end listline;

    /*************************************************************/
    /*   the following variables are used by the parser          */
    /*************************************************************/

declare listprod     byte initial(false),
        lowertoupper byte initial(true),
        listsource   byte initial(false),
        debugin      byte initial(false),
        listtoken    byte initial(false),
        errset       byte initial(false),
        compiling    byte,
        codesize     address,  /* used to count size of code area */
        prtct        address initial(0fffeh),  /* used to count size of prt */

         /* variables used during for loop code generation */

        forcount      byte initial(0),
        randomfile    byte,
        fileio        byte initial(false);


   /*************************************************************/
   /*              scanner procedures                           */
   /*************************************************************/

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

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

     checkfile: procedure byte;
      do forever;
        if (sourceptr:=sourceptr+1)=cursourcerecsize then
          do;
             sourceptr=0;
             if read$source$file=fileeof then
                  return true;
          end;
        if(nextchar:=next$source$char)<>lf then
          return false;
      end;
     end checkfile;

    if checkfile or (nextchar = eoffiller) then
              do; /* eof reached */
              call move(.addeof,sbloc,5);
```

```
                sourceptr = 0;
                nextchar=next$source$char;
                end;
        linebuff(lineptr:=lineptr + 1)=nextchar; /*output line*/
        if nextchar  = eolchar  then
                do;
                  lineno = lineno + 1;
                  if listsource  then
                       call listline(lineptr-1);
                  lineptr = 0;
                  call clearlinebuff;
                end;
        if nextchar = tab then
            nextchar = ' ';
        return nextchar;
end getchar;

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


title:procedure;
      call print(.'algol-m vers 1.0$');
      call crlf;
end title;

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

error: procedure(errcode);
       declare errcode address.
                 1        byte;
       errorcount=errorcount+1;
       call print(.'***$');
       call print$dec(lineno);
       call print(.'  error $');
       call printchar(' ');
       call printchar(high(errcode));
       call printchar(low(errcode));
       call crlf;
       call print$prod;
       if token=eofc then
         do;
          call print$error;
          call mon3;
         end;
end error;


initialize$scanner: procedure:
    declare count byte:
    call open$sourcefile;
    lineno.lineptr = 0;
    call clear$line$buff;
    sourceptr = 128;
    call getnoblank;
      do while nextchar = '$';
          call get$no$blank;
          if(count := (nextchar and 5fh) - 'a') <= 4 then
                do case count;
                      if pass1 then listsource = true;
                      listprod = true;
                      nointfile = true;
                      listtoken = true;
                      debugin = true;
                end; /* of case */
      call getnoblank;
```

```
      end;
end initialize$scanner;

   /*****************************************************/
   /*                scanner                            */
   /*****************************************************/

scanner: procedure;

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

    putandget: procedure;
         call putinaccum;
         call getnoblank;
    end putandget;

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

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

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

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

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

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

    alphanum: procedure byte;
         return numeric or letter or decimalpt;
    end alphanum;

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


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


lookup: procedure byte;

declare maxrwing lit  '9';
```

```
declare vocab data(0,'<','(','+',5dh,7ch,'*',')',':','-','/',',','>'
    ,':',5bh,'=','**',':=','do','go','if','of','or','to','eof','and'
    ,'end','for','not','pic','tab','case','else','file','goto','rend'
    ,'step','then','array','begin','close','until','while','write'
    ,'string','decimal','initial','integer','writeon'
    ,'comment','external','function','procedure');
declare vloc data(0,1,16,32,53,81,111,117,152,168,177);
declare vnum data(0,1,16,24,31,38,44,45,50,54);
declare count data(0,14,7,6,6,5,0,4,1,0);
declare ptr address, (field based ptr) (9) byte;
declare i byte;

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

        if accum > maxrwing then
             return false;
        ptr=vloc(accum)+.vocab;
        do i=vnum(accum) to (vnum(accum)+count(accum));
            if compare then
                do;
                  if i=50 then
                    token=comment;
                  else
                    token=i;
                  return true;
                end;
            ptr=ptr+accum;
        end;
        return false;
    end lookup;


/*****************************************************/
/*              scanner main code                    */
/*****************************************************/


    do forever;
      accum, hashcode, token = 0;
      do while nextchar=eolchar;
          call getnoblank;
          end;
      if(nextchar = stringdelim) or cont then
         do; /* found string */
              token = string;
              cont = false;
              do forever;
                   do while getchar <> stringdelim;
                        call putinaccum;
                        if cont then return;
                        end;
                   call getnoblank;
                   if nextchar <> stringdelim then
                        return;
                   call put$in$accum;
              end; /* of do forever */
         end; /* of recognizing a string */


   else if numeric or decimalpt then
         do; /* have digit */
              token = integerc;
              do while nextchar='0'; /*elim leading zeros*/
                 nextchar=getchar;
              end;
              call spoolnumeric;
              if decimalpt then
```

```
                do;
                token=decimal;
                call putandchar;
                call spoolnumeric;
                end;
            if accum=0 then
                hashcode, accum(accum := 1) = '0';
            call setup$next$call;
            return;
        end; /* of recognizing numeric constant */


    else if letter then
        do; /* have a letter */
            do while alphanum;
                call putandchar;
                end;
            if not lookup then
            do;
                token = identifier;
                call setup$next$call;
                return;
            end;
            else /* is a rw but if comment skip */
                if token = comment then
                   do;
                     do while nextchar <> ';';
                          nextchar = getchar;
                          end;
                     call get$no$blank;
                   end;
                else
                    do;
                      call set$up$next$call;
                      return;
                    end;
        end; /* of recognizing rw or ident */


    else
     do; /* special character */
       if nextchar = 25h then
          do;
             nextchar=getchar;
             do while nextchar <> 25h;
                nextchar = getchar;
             end;
             call get$no$blank;
          end;
       else
         do;
           if nextchar = ':' then
             do;
               call putandchar;
               if nextchar = '=' then
               call putandget;
             end;
          else
            if nextchar = '*' then
             do;
               call putandchar;
               if nextchar = '*' then
               call putandget;
             end;
            else  call putandget;
            if not lookup then
              call error('lc');
            call setup$next$call;
            return;
          end;
     end; /* of recognizing special char */
    end;  /* of do forever */
end scanner; /* end of scanner */
```

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


initialize$symtbl:proc;
    if pass1 then
      do;
   /* fill hashtable with 0's */
      call fill(.hashtable,0,shl(hashtblsize,2));
      sbtbl = .memory;
      end;
   /*initialize pointer to top of symbol table*/
    sbtbltop = max - 2;
end initialize$symtbl;

setaddrptr:proc(offset); /*set ptr for addr reference*/
    declare offset byte;
    aptraddr = base + ptr + offset;/*position for addr reference*/
end setaddrptr;

set$blk$level:proc(level);
   declare level byte;
   call setaddrptr(6);
   byteptr = level;
end set$blk$level;

gethash:proc byte;
   declare hash byte,
           i    byte;
    hash = 0;
    aptraddr = base + 2;
    do i = 1 to ptr;
       hash = (hash + byteptr(i)) and hashmask;
    end;
    return hash;
end gethash;

nextentry:proc;
   base = base + ptr + 8;
end nextentry;

setlink:proc;
   aptraddr = base + 1;
end setlink;

hashtbl$of$symhash:proc address;
   return hashtable(symhash);
end hashtbl$of$symhash;

limits:proc(count);
     /*check to see if additional sbtbl will overflow limits of
      memory. if so then punt else return */
      declare count byte;  /*size being added is count */
      if sbtbltop <= (sbtbl + count) then
        do;
          call error('to');
          call mon3;
        end;
end limits;

setaddr:proc(loc);
     /*set the address field and resolved bit*/
     declare loc address;
     call setaddrptr (4);
     addrptr = loc;
end setaddr;

lookup$current$blk:proc(chk$blk) byte;
   declare chk$blk byte,
       len byte,
        n based printname byte;
```

```
    base=hashtbl$of$symhash;
    do while base <> 0;
       call setaddrptr(6);
       if byteptr < chk$blk then
            return false;
       if byteptr = chk$blk then
          do;
            if (len:=ptr) = n then
              do while (ptr(len + 2) = n(len));
                if (len := len - 1) = 0 then
                   return true;
              end;
      end;
      call setlink;
      base = addrptr;
   end;
   return false;
end lookup$current$blk;

lookup:proc byte;
 declare test$blk byte,
         test$index byte;
 test$index = prev$index+1;
 test$blk = blk$level;
 do while (test$index := test$index - 1) <> 255;
    if lookup$current$blk(test$blk) then
         return true;
    test$blk = prev$blk$level(test$index);
 end;
 return false;
end lookup;

enter:proc;
   /*enter token reference by printname and symhash
     into next available location in the symbol table.
     set base to beginning of this entry and increment
     sbtbl.  also check for symbol table full.  */
   declare i byte,
           n based printname byte;
   call limits(i:=n+8);
   base = sbtbl; /*base for new entry */
   call move(printname + 1,sbtbl + 3,(ptr := n));
   call setaddrptr(3);/*set resolve bit to 0*/
   byteptr = 0;
   call setlink;
   addrptr = hashtbl$of$symhash;
   hashtable(symhash) = base;
   call set$blk$level(blk$level);
   sbtbl = sbtbl + i;
end enter;

getlen:proc byte; /*return length of the p/n */
   return ptr;
end getlen;

getype:proc byte; /*returns type of variable*/
   call setaddrptr (3);
   return byteptr;
end getype;

setsubtype:proc(stype);/*enter the subtype in sbtbl*/
     declare stype byte;
     call setaddrptr(7);
     byteptr=stype;
end setsubtype;

get$parm: proc byte;
     call setaddrptr(10);
     return byteptr;
end getparm;

getsubtype:proc byte;/*return the subtype*/
     call setaddrptr(7);
```

```
     return byteptr;
end getsubtype;
setype:proc (type); /*set typefield = type*/
   declare type byte;
   call setaddrptr (3);
   byteptr =  type;
end setype;

getaddr:proc address;
   call setaddrptr(4);
   return addrptr;
end getaddr;


do;   /* block for parser */

      /* pneumonics for ALGOL-M machine  */

declare nop lit  '0',  str lit  '1',  int lit  '2',  xch lit  '3',
        lod lit  '4',  dcb lit  '5',  dmp lit  '6',  xit lit  '7',
        sld lit  '8',  als lit  '9',  aid lit '10',  ais lit '11',
        adl lit '12',  add lit '13',  sbi lit '14',  sbd lit '15',
        mpi lit '16',  mpd lit '17',  dvi lit '18',  dvd lit '19',
       dneg lit '21',  neg lit '22',  cil lit '23',  ci2 lit '24',
       decl lit '25',  pop lit '26',  iml lit '27',  im2 lit '28',
                                      cat lit '31',  bli lit '32',
        brs lit '34',  bsc lit '35',  lss lit '36', dlss lit '37',
       slss lit '38',  gtr lit '39', dgtr lit '40', sgtr lit '41',
        eql lit '42', deql lit '43', seql lit '44',  neq lit '45',
       dneq lit '46', sneq lit '47',  geq lit '48', dgeq lit '49',
       sgeq lit '50',  leq lit '51', dleq lit '52', sleq lit '53',
       inot lit '54', dnot lit '55', snot lit '56', land lit '57',
       dand lit '58', sand lit '59',  lor lit '60',  dor lit '61',
        sor lit '62',  wic lit '63',  wdc lit '64',  wsc lit '65',
        wid lit '66',  wdd lit '67',  wsd lit '68',  sbr lit '69',
        bra lit '70',  row lit '71',  sub lit '72',  rcl lit '73',
        rcd lit '74',  rcs lit '75',  rdi lit '76',  rdd lit '77',
        rds lit '78',  rcn lit '79',  ecr lit '80',  sli lit '81',
        sdi lit '82',  ssi lit '83',  sid lit '84',  sdd lit '85',
        ssd lit '86',  opn lit '87',  cls lit '88',  rdb lit '89',
        rdf lit '90',  edr lit '91',  edw lit '92',  pro lit '93',
        sav lit '94',  sv2 lit '95',  uns lit '96',  rtn lit '97';

declare state                  statesize,
        statestack(pstacksize) statesize,
        hash(pstacksize)         byte,
        symloc(pstacksize)       address,
        srloc(pstacksize)        address,
        var(pstacksize)          byte,
        type(pstacksize)         byte,
        stype(pstacksize)        byte,
        varc(varcsize)           byte,
        varindex               byte,
        (sp,mp,mppl,nolook)      byte,
        onstack(maxoncount)      byte,
        clabing                byte initial(2),
        clab2                  byte initial(23),
        clable                 byte,
        str$size               byte initial(10),
        dec$size               byte initial(9),
        n                      byte,
        pvnum                  byte initial(0),
        saveparm               address,
        parm based saveparm    byte,
        fpcount                byte,
        parmbase               address,
        proc$type              byte,
       (ptest,i)               byte,
        pcount                 byte initial(0),
        typetemp               byte,
        fflag                  byte,
        lpcount                byte,
        sident                 address;
```

```
initialize$synthesize:procedure;
    codesize,onstack,clable = 0;
    prev$index = 255;
    blk$cnt = 0;
    blk$level=0;
end initialize$synthesize;


synthesize: proc;


  /* **** synthesize local declarations  **** */


declare simvar          lit '0bh',
        subvar          lit '99',
        pro             lit  '93',
        ext$proc        lit '03',
        bit$in$func     lit '05',
        const           lit '06',
        lab             lit '07',
        integer         lit '08',
        str             lit '1',
        file1           lit '0ch',
        func            lit '0dh',
        parm            lit '10h';

declare (typesp,typemp,typempl)          byte,
        (b,temp)                         byte,
        (stypesp,stypemp,stypempl)       byte,
        (hashsp,hashmp,hashmpl)          byte,
        (symlocsp,symlocmp, symlocmpl)   address,
        (srlocsp,srlocmp)                address;

    /*   ***************************************************************   */
    /*   **********      code generation proc's          **********   */
    /*   ***************************************************************   */

     copy:procedure;
          typesp = type(sp);
          typempl = type(mppl);
          typemp = type(mp);
          stypesp = stype(sp);
          stypempl = stype(mppl);
          stypemp = stype(mp);
          symlocsp = symloc(sp);
          symlocmpl = symloc(mppl);
          symlocmp = symloc(mp);
          hashmp = hash(mp);
          hashmpl = hash(mppl);
          hashsp = hash(sp);
          srlocsp = srloc(sp);
          srlocmp = srloc(mp);
     end copy;


    setsymlocsp: procedure(a);
         declare a address;
         symloc(sp) = a;
    end setsymlocsp;

     setsymlocmp: procedure(a);
       declare a address;
       symloc(mp) = a;
     end setsymlocmp;

    settypesp: procedure(b);
         declare b byte;
         type(sp) = b;
    end settypesp;
```

```
     setstypesp: procedure(b);
          declare b byte;
          stype(sp) = b;
     end setstypesp;

     setstypemp: procedure(b);
          declare b byte;
          stype(mp) = b;
     end setstypemp;

     settypemp: procedure(b);
          declare b byte;
          type(mp) = b;
     end settypemp;


     sethashmp: procedure(b);
          declare b byte;
          hash(mp) = b;
     end sethashmp;

     sethashsp: procedure(b);
          declare b byte;
          hash(sp) = b;
     end sethashsp;

     setsrlocsp: procedure(a);
          declare a address;
          srloc(sp) = a;
     end setsrlocsp;
    setsrlocmp: proc(a);
        declare a byte;
        srloc(mp)=a;
    end setsrlocmp;

    getsrloc: proc byte;
        call setaddrptr(8);
        return addrptr;
    end  getsrloc;

      generate:proc(objcode);
        /*writes generated code and counts size
        of code area.  */
        declare objcode byte;
        codesize = codesize + 1;
        if not pass1 then
          call emit(objcode);
      end generate;

      gen$int$v: proc(a);
          declare a byte;
          call generate(iml);
          call generate(a);
      end gen$int$v;

incr$blk$level: proc;
     prev$blk$level(prev$index := prev$index+1) = blk$level;
     blk$level = blk$cnt + 1;
     blk$cnt = blk$cnt+1;
     call generate(bll);
end incr$blk$level;

decr$blk$level: proc;
     blk$level = prev$blk$level(prev$index);
     prev$index = prev$index-1;
     call generate(bld);
end decr$blk$level;

calc$varc: procedure(b) address.
  declare b byte;
  return var(b) + .varc;
end calc$varc;
```

```
setlookup: procedure(a);
    declare a byte;
    printname = calc$varc(a);
    symhash = hash(a);
end setlookup;

lookup$only: procedure(a) byte;
    declare a byte;
    call setlookup(a);
    if lookup$current$blk(blk$level) then
         return true;
    base$flag = true;
    return false;
end lookup$only;

full$lookup:proc(a) byte;
    declare a byte;
    call setlookup(a);
    if lookup then
      return true;
    return false;
end full$lookup;

normal$lookup: procedure(a) byte;
    declare a byte;
    if lookup$only(a) then
         return true;
    call enter;
    return false;
end normal$lookup;

countprt:proc address;
/*counts the size of the prt */
   return (prtct := prtct + 2);
end countprt;

gentwo:proc(a);
/* writes two bytes of object code on disk for literals */
  declare a address;
  call generate(high(a));
  call generate(low(a));
end gentwo;


literal:proc(a);
   declare a address;
   call gentwo(a or 8000h);
end literal;

setcname:proc;
     printname=.clabing;
     symhash=clable and hashmask;
end setcname;

enter$compiler$label:proc(b);
     declare b byte;
     if pass1 then
       do;
         call setcname;
         call enter;
         call setaddr(codesize + b);
       end;
end enter$compiler$label;

set$compiler$label:proc;
    declare x byte;
    clable = clable + 1;
    call setcname;
    if pass2 then
      x=lookup;
end set$compiler$label;

compiler$label:proc;
```

```
    call set$compiler$label;
    call gen$two(getaddr);
end compiler$label;

set$enter: proc(a);
     declare (a,b) byte;
     b=clable;
     clable=a;
     call enter$compiler$label(0);
     clable=b;
end set$enter;


branch$clause: proc(a);
     declare a byte;
     call generate(a);
     call compiler$label;
     call settypemp(clable);
end branch$clause;

process$case$label: proc(a);
   declare (a,b,x) byte;
   b=clable;
   clable=a;
   call generate(bra);
   call setcname;
   if pass=2 then
     x=lookup;
   call gen$two(getaddr);
   clable=b;
end process$case$label;

case$state: proc;
   call process$case$label(type(mp-2));
   n=n+1;
   clable=clable+1;
   call enter$compiler$label(0);
end case$state;

litload:proc(a);
    declare a address;
    call gentwo(a or 0c000h);
 end litload;

step$gen: proc;
     call generate(adi);
     call generate(sid);
     call enter$compiler$label(0);
     call litload(sident);
end step$gen;

chktypl: proc(a); /* check mp,sp to see if they are both decimal */
    declare a byte;   /* both integer, one of each, or niether */
    if (stypemp=int) and (stypesp=int) then
       call generate(a);
    else
       if (stypemp=dec) and (stypesp=dec) then
          call generate(a+1); /* generate decimal operator */
       else
          if (stypemp=dec) and (stypesp=int) then
             do;
               call generate(cil); /* convert integer to decimal */
               call generate(a+1);
             end;
          else
             if (stypemp=int) and (stypesp=dec) then
                do;
                  call generate(cil2);
                  call generate(a+1);
                  call setstypemp(dec);
                end;
             else
                if typemp<>func or pass2 then
```

```
                call error('mf');
end chktyp1;

chktyp2: proc byte;
        if stypesp <> stypemp then
          do;
            call error('mm');
            return false;
          end;
        return true;
end chktyp2;

chktyp3: proc byte;
      call setstypemp(stypesp);
      if (stypesp=int) or (stypesp=dec) then
          return true;
      call error ('mf');
      return false;
end chktyp3;

chktyp5: proc byte;
     if (stypemp<>str) or (stypesp<>str) then
        return false;
     else
        return true;
end chktyp5;

chktyp6: proc;
     if pass2 then
       do;
       if (stypemp=int) and (stypesp=dec) then
          call error('id');
       if (typemp<>simvar) and (typemp<>subvar) and (typemp<>filel) then
          do;
            if (typemp<>func) or (not fflag)  then
               call error('as');
          end;
       end;
end chktyp5;

gencon: proc(subtype);
      declare (i,subtype) byte;
      gen$accum: proc;
      if pass2 then
         do i=1 to accum;
           call emit(accum(i));
         end;
      end gen$accum;
      call generate(subtype);
      call settypemp(const);
      call setstypemp(subtype);
      if subtype=int then
       do;
         if accum>5 then
           call error('io');
         call gen$accum;
         if pass2 then
           call emit(0);
         codesize=codesize+2;
       end;
      else
        do forever;
           if subtype=str then
               do i = 1 to accum;
                  call generate(accum(i));
               end;
           else
             do;
              call gen$accum;
              codesize=codesize+(accum+1)/2;
             end;
               if cont then
                  call scanner;
```

```
                else
                   do;
                    if subtype=dec then
                      call gentwo(0);
                    else
                      codesize = codesize + 2;
                     call generate(0);
                     return;
                   end;
            end;
end gencon;

process$store: proc(a);
     declare a byte;
     if chktyp5 then
       call generate(a+2);
     else
       call chktyp1(a);
end process$store;

     gen$loc: proc(a,b);
       declare a byte,
            b address;
       if a =int then
         call literal(b);
       else
         call litload(b);
     end gen$loc;

get$field: proc;
     gen$read: proc(a);
         declare a byte;
         if stypemp1=int then
           do;
             call generate(a);
             call generate(sld);
           end;
         else
           if stypemp1=dec then
              do;
                call generate(a+1);
                call generate(sdd);
              end;
            else
              do;
                call generate(a+2);
                call generate(sad);
              end;
      end gen$read;

      call gen$loc(stypemp1,symlocmp1);
      if fileio then
         call gen$read(rdi);
      else
         call gen$read(rci);
end get$field;

put$field: proc(a);
     declare a byte;
     if fileio then
      a=a+3;
      if stypemp1=int then
        do;
          if typemp1=subvar then
            call generate(iod);
            call generate(a);
        end;
     else
       if stypemp1=dec then
          call generate(a+1);
       else
         call generate(a+2);
end put$field;
```

```
process$proc: proc(a) ;
    declare a byte;
    call settypemp(a);
    if pass2 then
         do;
           call setsrlocmp(srlocmp:=getsrloc);
           call setstypemp(getsubtype);
           parmbase=base+ptr+11;
         end;
end process$proc;

process$ident:proc(a) byte;
    declare a byte;
    if full$lookup(a) then
       return getype;
    else
       if pass2 then
         do;
          call error('ud');
          return false;
         end;
       else
         return func;
end process$ident;

process$array:proc(a);
   declare (a,b) byte;
   if ((b:=process$ident(a)) <> subvar) and pass2 then
     do;
       if b<>0 then
         call error('iu');
     end;
   else
     do;
       call setstypemp(getsubtype);
       call setsymlocmp(symlocmp:=getaddr);
       call settypemp(subvar);
     end;
end process$array;

process$ident$dcl:proc(a,b,c);
   dcl(a,b,c) byte;
   pvnum=pvnum+1;
   if not normal$lookup(a) then
     do;
       call setype(b);
       call setaddr(countprt);
       call setsubtype(c);
     end;
   else
     if pass1 then
        do;
          if getype=parm then
           do:
             call setype(b);
             call setsubtype(c);
             ptest=ptest-1;
           end;
          else
           call error('dd');
        end;
end process$ident$dcl;

proc$sav: proc;
     call gen$int$v(pcount);
     call gen$int$v((pvnum+3)*2);
     call literal(symlocmp);
     call generate(sav);
     call generate(bll);
     if proc$type=func then
        do;
          if stypemp<> int then
```

```
                do;
                   call litload(symlocmp+4);
                   call literal (symlocmp);
                   if stypcmp=dec then
                     call generate(ild);
                   else
                     call generate(als);
                end;
            end;
 end proc$sav;

 check$parm: proc(a);
      declare a byte,
          b address;
      if pass2 then
        do;
          b=base;
          base=parmbase;
          if a <> getsubtype then
            call error('pm');
          parmbase=base+ptr+8;
          base=b;
        end;
        call generate(iml);
        call generate(a);
 end check$parm;

 proc$pro: proc;
      call generate(lm2);
      call gen$two(srlocmp);
      call generate(pro);
 end proc$pro;

 process$proc$dcl: proc(a,b,c);
     declare (a,b,c) byte;
     call process$ident$dcl(a,b,c);
     pvnum,pcount=0;
     call setsymlocmp(symlocmp:=getaddr);
     if pass1 then
       prtct=prtct+4;
     if (proc$type:=b) = func then
       do;
         call literal(symlocmp+4);
         call generate(xch);
         call generate(sid);
       end;
     call branch$clause(brs);
     if pass1 then
      do;
        call setaddrptr(8);
        addrptr=codesize;
        sbtbl=sbtbl+3;
        saveparm=sbtbl-1;
      end;
     call incr$blk$level;
end process$proc$dcl;

process$var: proc(a);
     declare (a,b) byte;
     if (b:=process$ident(a)) = simvar then
        do;
         call setsymlocsp(symlocsp:=getaddr);
         call settypesp(simvar);
         call setstypesp(getsubtype);
        end;
     else
         if b=func then
            call process$proc('func');
         else
            if b<> 0 then
               call error('ip');
end process$var;
```

```
process$simvar$dcl: proc(a,b);
     declare (a,b) byte;
     call process$ident$dcl(a,simvar,stypemp);
     if stypemp<>int then
       do;
         call literal(getaddr);
         if stypemp=dec then
            call generate(b);
         else
            call generate(b+1);
       end;
end process$simvar$dcl;

process$label:proc;
   if pass1 then
      do;
        if full$look$up(mp) then
           call error('dd');
        else
           do;
             call enter;
             call setaddr(codesize);
             call setype(lab);
             call set$blk$level(blk$level);
           end;
      end;
end process$label;

resolve$label:proc;
    declare (chk$blk,tindex) byte;
    if pass2 then
       do;
         if ((not full$lookup(sp)) or (getype <> lab)) then
              call error('ul');
         call setaddrptr(6);
         chk$blk=byteptr;
         tindex=prev$index;
         if blk$level<>chk$blk then
          do;
            do while prev$blk$level(tindex) > chk$blk;
              tindex=tindex-1;
            end;
            call generate(dcb);
            call generate(prev$index+1-tindex);
          end;
        else
          call gen$two(nop);
      end;
    else
      call gen$two(nop);
    call generate(brs);
    call gen$two(getaddr);
end resolve$label;


  process$array$dcl: proc(a);
      declare a byte;
      call process$ident$dcl(a,subvar,stypemp);
      arr$loc(arr$num)=getaddr;
      arr$num=arr$num+1;
  end process$array$dcl;

close$file: proc(a);
    declare a byte;
    if process$ident(a) then; /* not implemented */
    else call error('uf');
end close$file;

assign$stmt: proc;
     call chktyp6;
     if (ipcount:=ipcount-1)<>0 then /* tests multiple assign stmts */
       call process$store(sil);
     else
```

```
      call process$store(sid);
end assign$stmt;

/*  execution of synthesize begins here----      */

  if listprod then
     call print$prod;
   call copy;
   do case production; /* call to systhesize handles one prod */


/*case 0 not used */  ;

/*      1   <program> ::= <block> _|_                                  */

      if pass1 then
         do;
           pass1 = false;
           if errorcount > 0 then
            do;
               call print$error;
               call mon3;
            end;
           call rewind$source$file;
           call gentwo(codesize+1); /* plus one to include the xit */
           call gentwo(countprt);
         end;
       else
         do;
           call print$error;
           call generate(xit);
           call generate(7fh);
           call write$int$file;
           call close$int$file;
           call mon3;
         end;

/*      2   <block> ::= <block head> <block end>                       */
          ;

/*      3   <block head> ::= <block head> <declaration> ;              */
          ;
/*      4                  | <begin>                                 */
          ;

/*      5   <begin> ::= begin                                          */
         call incr$blk$level;

/*      6   <block end> ::= <block body> ;  end                        */
         call decr$blk$level;

/*      7   <block body> ::= <statement>                               */
          ;
/*      8                  | <block body> ;  <statement>              */
          ;

/*      9   <declaration> ::= <file declaration>                       */
          ;
/*     10                   | <simple declaration>                   */
          ;
/*     11                   | <simple declaration>                   */
/*     11                      initial option>                        */
          ;
/*     12                   | <array declaration>                    */
          ;
/*     13                   | <array declaration> <initial option>   */
          ;
/*     14                   | <subprogram declaration>               */
          ;
/*     15                   | <external declaration>                 */
          ;
```

```
/*      16   <simple declaration> ::= <declaration head>                  */
/*      16                            <identifier>                        */

              call process$simvar$dcl(sp,aid);
/*      17   <initial option> ::= <initial head> <constant> )             */
            ;
/*      18   <initial head> ::= initial (                                 */
            ;
/*      19                    | <initial head> <constant> ,               */
            ;
/*      20   <declaration head> ::= <declaration type>                    */
            ;
/*      21                        | <declaration head> <identifier>       */
/*      21                                                                */
          call process$simvar$dcl(mpp1,aid);

/*      22   <declaration type> ::= string                                */
            do;
              call setstypesp(str);
              call gen$int$v(str$size);
            end;
/*      23                        | string <size option>                  */
            do;
              call setstypemp(str);
            end;
/*      24                        | integer                               */
              call setstypesp(int);
/*      25                        | decimal                               */
            do;
              call setstypesp(dec);
              call gen$int$v(dec$size);
            end;
/*      26                        | decimal <size option>                 */
             call setstypemp(dec);
/*      27   <size option> ::= ( <variable> )                             */
          if stypemp1=int then
              call litload(symlocmp1);
          else
              call error('si');
/*      28                    | ( <integer> )                             */
          do;
             call move(calc$varc(mpp1),.accum,7);
             call gencon(int);
          end;
/*      29   <statement> ::= <balanced statement>                         */
            ;
/*      30                  | <unbalanced statement>                      */
            ;
/*      31   <balanced statement> ::= <simple statement>                  */
            ;
/*      32                          | <if clause> <true part> else        */
/*      32                            <balanced statement>                */
          call set$enter(typemp1);
/*      33                          | <label definition>                  */
/*      33                            <balanced statement>                */
            ;
/*      34   <unbalanced statement> ::= <if clause> <statement>          */
            call set$enter(typemp);
/*      35                            | <if clause> <true part>           */
/*      35                              else <unbalanced statement>       */
            call set$enter(typemp1);
/*      36                            | <label definition>                */
/*      36                              <unbalanced statement>            */
              ;
/*      37   <true part> ::= <balanced statement>                         */
            do;
              call branch$clause(brs)
              call set$enter(type(sp-1));
            end;
/*      38   <label definition> ::= <identifier> :                        */
              call process$label;
/*      39                        | <integer> :                           */
            call process$label;
```

```
/*      40   <simple statement> ::= <block>                                  */
          ;
/*      41                          | <assignment statement>                 */
            ;
/*      42                          | <for statement>                        */
          ;
/*      43                          | <while statement>                      */
          ;
/*      44                          | <read statement>                       */
          ;
/*      45                          | <write statement>                      */
          ;
/*      46                          | <case statement>                       */
          ;
/*      47                          | <go to statement>                      */
           ;
/*      48                          | <close statement>                      */
           ;
/*      49                          | <procedure call>                       */
          call generate(pop);
/*      50                          | <identifier>                           */
                do;
                  if ((process$ident(sp) <> pro) and pass2) then
                          call error('nf');
                  call process$proc(pro);
                  call proc$pro;
                  call generate(pop);
                end;
/*      51   <assignment statement> ::= <left part> <expression     */
         call assign$stmt;
/*   52                          <left part>                          */
/*                             <assignment statement>        */
        call assign$stmt;
/*      53      <left part>  <variable> :=                          */
          do;
            if typemp<>subvar then
              call gen$loc(stypemp,symlocmp);
            lpcount=lpcount+1;
          end;
/*      54   <expression> ::= <arithmetic expression>                        */
          ;
/*      55                  |  if expression> <expression>                   */
          ;
/*      56   <arithmetic expression> ::= <term>                              */
          ;
/*      57                              | <arithmetic expression>  +          */
/*      57                                <term>                              */
        call chktyp1(adi);
/*      58                              | <arithmetic expression>  -          */
/*      58                                <term>                              */
         call chktyp1(sbi);
/*      59                              | <arithmetic expression>  !          */
/*      59                                ! <term>                            */
           if chktyp5 then
              call generate(cat);
/*      60                              | - <term>                            */
        if chktyp3 then
            call generate(neg);
/*      61                              | + <term>                            */
        if chktyp3 then ;  /* no action required */
/*      62   <term> ::= <primary>                                            */
          ;
/*      63              | <term> * <primary>                                 */
          call chktyp1(mpi);
/*      64              | <term> / <primary>                                 */
        call chktyp1(dvi);
/*      65   <primary> ::= <primary element>                                 */
          ;
/*      66                | <primary> ** <primary element>                   */
         call chktyp1(ixp);
/*      67   <primary element> ::= <variable>                                */
          if typesp=simvar then
              call litload(symlocsp);
```

```
        else
            if typesp<> subvar then
              call proc$pro;
/*    68                          | <constant>                          */
        ;
/*    69                          | <procedure call>                    */
        ;
/*    70                          | ( <assignment statement> )          */
        call setstypemp(stypemp:=stypemp1);
/*    71                          | ( <expression> )                    */
        do;
          call setstypemp(stypemp1);
          call settypemp(typemp1);
        end;
/*    72   <constant> ::= <integer>                                     */
        call gencon(int);
/*    73                  | <decimal>                                   */
          call gencon(dec);
/*    74                  | <string>                                    */
          call gencon(str);
/*    75   <variable> ::= <identifier>                                  */
          call process$var(sp);
/*    76                  | <subscripted variable>                      */
        ;
/*    77   <file declaration> ::= <file head> <file name>               */
        ;
/*    78   <file head> ::= file                                         */
        ;
/*    79                  | <file head> <file name> ,                   */
        ;
/*    80   <file name> ::= <string> <length option>                     */
        ;
/*    81                  | <string>                                    */
        ;
/*    82                  | <identifier> <length option>                */
          call process$ident$dcl(mp,file1,0);
/*    83                  | <identifier>                                */
          call process$ident$dcl(sp,file1,0);
/*    84   <length option> ::= ' <identifier> &                         */
          if process$ident(mp)=int then;
/*    85                      | ' <integer> &                           */
        call gencon(int);
/*    86   <array declaration> ::= <array list> <bound pair list>       */
        do;
          call gen$int$v(arr$dim);
          call gen$int$v(arr$num);
          call gen$int$v(stypemp);
          call generate(row);
          do while (arr$num := arr$num-1)<>255;
            call literal(arr$loc(arr$num));
          end;
        end;
/*    87   <array list> ::= <array head> <identifier>                   */
        call process$array$dcl(sp);
/*    88   <array head> ::= <declaration type> array                    */
          arr$num=0;
/*    89                  | <array head> <identifier> ,                 */
        call process$array$dcl(mpp1);
/*    90   <bound pair list> ::= <bound pair head> <bound pair> &       */
        arr$dim=arr$dim+1;
/*    91   <bound pair head> ::= '                                      */
        arr$dim=0;
/*    92                        | <bound pair head> <bound pair> ,      */
        arr$dim=arr$dim+1;
/*    93   <bound pair> ::= <expression> : <expression>                 */
          if (stypemp<>int) or (stypesp<>int) then
              call error('bp');
/*    94   <subscripted variable> ::= <subscript head>                  */
/*    94                                <expression> &                  */
          do;
            if stypemp1<>int then
                call error('sh');
            call litload(symlocmp);
```

```
            call generate(subo);
          end;
/*     95   <subscript head> ::= <identifier> '                          */
          call process$array(mp);
/*     96                        | <subscript head> <expression> ,        */
             if stypempl <> int then
                  call error('sh');
/*     97   <go to statement> ::= <go to> <identifier>                   */
           call resolve$label;
/*     98                        | <go to> <integer>                     */
            call resolve$label;
/*     99   <go to> ::= go to                                            */
           ;
/*    100              | goto                                            */
           ;
/*    101   <read statement> ::= <read head> <variable> )                */
        do;
          call get$field;
          if fileio then
            do;
              fileio=false;
            end;

          else
            call generate(ecr);
        end;
/*    102   <read head> ::= read (                                        */
          call generate(rcn);
/*    103                  | read <file option> (                        */
           ;
/*    104                  | <read head> <variable> ,                    */
           call get$field;
/*    105   <write statement> ::= <write head> <expression> )            */
         call put$field(wic);
/*    106                        | <write head> <tab expression> )        */
          ;
/*    107                        | <write head> <pic definition> )        */
           ;
/*    108   <write head> ::= write (                                      */
          call generate(dmp);
/*    109                  | write <file option> (                       */
          ;
/*    110                  | writeon (                                    */
          ;
/*    111                  | writeon <file option> (                      */
           ;
/*    112                  | <write head> <expression> ,                  */
          call put$field(wic);
/*    113                  | <write head> <tab expression> ,              */
          ;
/*    114                  | <write head> <pic definition> ,              */
          ;
/*    115   <file option> ::= <identifier>                                */
          if process$ident(sp) = file1 then;
/*    116                  | <identifier> <rec option>                   */
           if process$ident(mp) = file1 then;
             else call error('nf');
/*    117                  | <string>                                     */
           ;
/*    118                  | <string> <rec option>                        */
           ;
/*    119   <rec option> ::= , <identifier>                               */
        if ((process$ident(sp) = simvar) and (getsubtype = int)) then;
          else call error ('nl');
/*    120                  | , <integer>                                  */
         call gencon(int);
/*    121   <pic definition> ::= <pic head> <pic list> )                  */
           ;
/*    122   <pic head> ::= pic <string>                                   */
           ;
/*    123                  | pic <identifier                             */
          if (process$ident(sp) = simvar and getsubtype = str  then;
          else call error('nl');
```

```
/*    124   <pic list> ::= ( <expression>                                  */
          ;
/*    125                  | <pic list> , <expression>                    */
          ;
/*    126   <tab expression> ::= tab <expression>                         */
          ;
/*    127   <if clause> ::= if <boolean expression> then                  */
       call branch$clause(bsc);
/*    128   <if expression> ::= <if clause> <expression> else             */
          ;
/*    129   <boolean expression> ::= <boolean term>                       */
          ;
/*    130                          | <boolean expression> or               */
/*    130                            <boolean term>                       */
        call chktyp1(bor);
/*    131   <boolean term> ::= <boolean primary>                          */
          ;
/*    132                      | not <boolean primary>                    */
      if chktyp3 then
          call generate(noto);
/*    133                      | <boolean term> and                       */
/*    133                        <boolean primary>                        */
         call chktyp1(ando);
/*    134   <boolean primary> ::= <logical expression>                    */
          ;
/*    135                        | ( <boolean expression> )               */
          call setstypemp(stypemp1);
/*    136   <logical expression> ::= <expression> <relation>              */
/*    136                            <expression>                         */
         if (stypemp=str) and (stypesp=str) then
          call generate(typemp1+2);
         else
           call chktyp1(typemp1);
/*    137   <relation> ::= =                                              */
       call settypesp(eql);
/*    138                  | <                                            */
        call settypesp(lss);
/*    139                  | >                                            */
       call settypesp(gtr);
/*    140                  | <comp>                                       */
          ;
/*    141   <comp> ::= < >                                                */
       call settypemp(neq);
/*    142              | < =                                              */
       call settypemp(leq);
/*    143              | > =                                              */
        call settypemp(geq);
/*    144   <while statement> ::= <while clause> <do statement>          */
            do;
              call generate(brs);
              call gen$two(symlocmp);
              call set$enter(typemp);
            end;
/*    145   <while clause> ::= <while> <boolean expression>              */
          call branch$clause(bsc);
/*    146   <while> ::= while                                             */
          call setsymlocmp(codesize);
/*    147   <for statement> ::= <for clause> <step expression>           */
/*    147                       <until clause> <do statement>            */
          do;
            call generate(brs);
            call gen$two(symlocmp);
            call set$enter(type(sp-1));
          end;
/*    148   <for clause> ::= for assignment statement>                   */
          do;
            if (typesp<>simvar) or (stypesp<>int) then
               call error('ni');
            call generate(brs);
            call compiler$label;
            call setsymlocmp(codesize);
            call literal(symlocsp);
            call litload(sident:=symlocsp);
```

```
          end;
/*    149   <step expression> ::= step <expression>                    */
         do;
           if stypesp<>int then
              call error('ni');
           step$flag=true;
           call step$gen;
         end;
/*    150   <until clause> ::= <until non-term> <expression>           */
        do;
          call generate(leq);
          call branch$clause(bsc);
        end;
/*    151   <until non-term> ::= until                                 */
       if not step$flag then
        do;
          call gen$int$v(1);
          call step$gen;
        end;
       else
        step$flag=false;
/*    152   <do statement> ::= do  simple statement>                   */
         ;
/*    153   <close statement> ::= close <identifier>                   */
          call close$file(sp);
/*    154                         | <close statement> , <identifier>   */
        call close$file(sp);
/*    155   <subprogram declaration> ::= <subprogram heading>          */
/*    155                                <simple statement>            */
      do;
        call literal(symlocmp);
        call generate(uns);
        call generate(rtn);
        call decr$blk$level;
        call set$enter(typemp);
        call literal(symlocmp+2);
        call gen$int$v(0);
        call generate(sid);
        pvnum,pcount=0;
        fflag=false;
      end;
/*    156   <subprogram heading> ::= <function heading>                */
        fflag=true;
/*    157                             <procedure heading>              */
         ;
/*    158   <function heading> ::= <paramless function>                */
         ;
/*    159                           <function & params>                */
        do;
          call generate(sv2);
           if ptest<>0 and pass1 then
             call error('pd');
          end;
/*    160   <procedure heading> ::= <paramless proc>                   */
         ;
/*    161                           | <proc & params>                  */
        do;
          call generate(sv2);
          if ptest<>0 and pass1 then call error('pd');
        end;
/*    162   <paramless function> ::= <declaration type> function       */
/*    162                             <identifier>      ;              */
          call process$proc$dcl(sp-1,func,stypemp);
/*    163   <function & params> ::= <function head> <identifier> ) ; */   */
        do;
          call process$ident$dcl(mppl,parm,0);
          parme,ptest=(pcount,=pcount+1);
          call proc$sav;
        end;
/*    164                           | <function & params>              */
/*    164                             <declaration> ;                  */
         ;
/*    165   <function head> ::= <declaration type> function            */
```

```
/*    163                          <identifier> (                          */
        call process$proc$dcl(sp-1,func,stypemp);
/*    166                          | <function head> <identifier> ,          */
      do;
        call process$ident$dcl(mppl,parm,0);
        pcount=pcount+1;
      end;
/*    167   <paramless proc> ::= procedure <identifier>     ;       */
      do;
        call process$proc$dcl(mppl,pro,0);
        call proc$sav;
      end;
/*    168   <proc & params> ::= <procedure head> <identifier> )   ;    */
        do;
        call process$ident$dcl(mppl,parm,0);
        parme,ptest=(pcount:=pcount+1);
        call proc$sav;
        end;
/*    169                          | <proc & params> <declaration> ;        */
        ;
/*    170   <procedure head> ::= procedure <identifier> (                 */
        call process$proc$dcl(mppl,pro,0);
/*    171                          | <procedure head> <identifier> ,        */
      do;
        pcount=pcount+1;
        call process$ident$dcl(mppl,parm,0);
      end;
/*    172   <procedure call> ::= <call heading> <expression> )            */
        do;
          pcount=pcount+1;
          call checkparm(stypempl);
          call proc$pro;
          if fpcount<>pcount and pass2 then
            call error('pe');
         end;
/*    173   <call heading> ::= <identifier> (                             */
          do;
          if ((b:=process$ident(mp))<>func) and (b<>pro) then
                call error('up');
          call process$proc(b);
          fpcount=getparm;
          pcount=0;
          end;
/*    174                          | <call heading> <expression> ,          */
        do;
        pcount=pcount+1;
        call checkparm(stypempl);
        end;
/*    175   <external declaration> ::= <declaration type> external       */
/*    175                                function <external list          */
        ;
/*    176                                | external procedure               */
/*    176                                  <external list>                  */
        ;
/*    177   <external list> ::= <identifier>                              */
        ;
/*    178                          | <external list> , <identifier>         */
        ;
/*    179   <case statement> ::= <case heading> <case block>              */
      do;
        do i=1 to n;
          call process$case$label(clable-i);
        end;
        call set$enter(typemp);
      end;
/*    180   <case heading> ::= case <expression> of                       */
      do;
        if stypempl<>int then
          call error('ni');
        call gen$int$v(3);
        call generate(mpi);
        call set$compiler$[illegible]
        call settypemp(clable[illegible]
```

```
          call generate(i-2);
          call gen$two(getaddr-4);
          call generate(sbr);
          call generate(bra);
          clable=clable+1;
          call enter$compiler$label(0);
          n=0;
        end;
 /*     181   <case block> ::= begin <case block body> end              */
          ;
 /*     182   <case block body> ::= <statement>                          */
      call case$state;
 /*     183                        | <case block body> ; <statement>     */
        call case$state;
end;  /*of case statement*/

end synthesize;


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

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

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


    /*   ***************************************************************   */
    /*   **********          lair parser routines          ***********   */
    /*   ***************************************************************   */


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

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


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


    getin2: procedure indexsize;
         return index2(state);
    end getin2;
```

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


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


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


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

do forever;
 do while pass1 or pass2;
     /* initialize variables */
     compiling,nolook=true;  state=starts;
     sp=255;
    varindex,var = 0;

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

```
                j=statestack(sp);
                     do while (k:=apply1(i)) <> 0 and j <> k;
                     i=i+1;
                     end;
                if(state:= apply2(i))=0 then compiling = false;
                end;  else
            if state<= maxino 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
            /* push state  */
                do;  call incsp;
                statestack(sp)= getin2;
                state=getin1;
                end;
            end;   /* of while compiling   */
        end;  /*of while pass1 or pass2*/
           listsource = false;
           listprod = false;
           listtoken = false;
           call initialize;

           pass2 = true;
       end;  /*of do forever*/

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


    eof
```

```
100h: /*load point for interp program*/

declare /*global literals*/
        lit          literally      'literally',
        true         lit            '1',
        false        lit            '0',
        forever      lit            'while true',
        cr           lit            '0dh',
        lf           lit            '0ah';

declare /*op codes for algol-m machine instructions*/
        decl         lit            '25',
        str          lit            '1',
        int          lit            '2',
        dcb          lit            '5',
        brs          lit            '34',
        bsc          lit            '35',
        im1          lit            '27',
        im2          lit            '28';

declare /*interface points for cp/m and interp*/
        bdos         lit            '05h',
        boot         lit            '0h',
        diskbuffloc  lit            '80h',
        fcbloc       lit            '5ch',
        diskbuffend  lit            '100h',
        bdosbegin    address        initial(06h),
        max          based          bdosbegin  address,
        buff         address        initial(diskbuffend),
        char         based          buff  byte,
        filename     address        initial(fcbloc),
        fnp          based          filename  byte;

declare /*build variables*/
        prtbase      address,
        prt$addr     address,
        prt$entry    based        prt$addr   address,
        codebase     address,
        codeptr      address,
        stackbase    address,
        curchar      byte,
        switch       byte         initial(false),
        bld$flag     byte,
        b            based        codeptr    byte,
        a            based        codeptr    address,
        temp1        address,
        t1           based        temp1      byte,
        temp2        address,
        t2           based        temp2      byte;


 /* declarations for interpreter */

declare contz        lit          '1ah',
        quote        lit          '22h',
        what         lit          '63';


declare eolchar      lit          '0dh',
        eoffiller    lit          '1ah',
        intrecsize   lit          '128',
        diskrecsize  lit          '128',
        stringdelim  lit          '22h',
        conbuffsize  lit          '80',
        console      lit          '0',
        nrstack      lit          '48', /*stack size times 4*/
        max$blk$level lit         '10',
        negative     lit          '0',
        positive     lit          '1';

declare ra             address,
        rb             address,
        rc             address,
```

```
        c               based       rc        byte,
        twobyteoprand   based       rc        address,
        sb              address,
        st              address,
        bra             based       ra        byte,
        ara             based       ra        address,
        arb             based       rb        address,
        brb             based       rb        byte,
        mpr             address,
        mcd             address;

declare inputbuffer                 byte initial(conbuffsize),
        buff$space(conbuffsize)     byte,
        inputindex                  byte,
        conbuffptr                  address,
        con$char                    based    conbuffptr  byte,
        inputptr                    address,
        input$char   based          inputptr    byte,
        num$read                    byte,
        printbufflength             lit         '71',
        printbufferloc              lit         '80h',
        tabpos1                     lit         '142',
        tabpos2                     lit         '156',
        tabpos3                     lit         '170',
        tabpos4                     lit         '184',
        printbuffer                 address      initial(printbufferloc),
        printpos                    based        printbuffer  byte,
        printbuffend                lit          '0c7h',
        rereadaddr                  address,
        inputtype                   byte,
        field$length                byte,
        sign                        byte;

declare fileaddr                    address,
        fcb                         based        fileaddr  byte,
        fcbadd                      based        fileaddr  address,
        eofaddr                     address,
        buffer$end                  address,
        record$pointer              address,
        buffer                      address,
        nextdiskchar                based        record$pointer byte,
        reg$length                  lit          '128',
        blocksize                   address,
        blk$level                   byte         initial(255),
        error$flag                  byte         initial(false),
        blk(max$blk$level)          address,
        bytes$written               address,
        firstfield                  byte,
        eofra                       address,
        eofrb                       address;

declare (r0,r1,r2) (11)             byte,
        (sign0,sign1,sign2)         byte,
        (dec$pt0,decpt1,dec$pt2)    byte,
        ctr                         address,
        no$shift                    byte,
        base                        address,
        b$byte  based base          byte,
        b$addr  based base          address,
        hold                        address,
        h$byte  based hold          byte,
        h$addr  based hold          address,
        ptr$one                     address,
        ptr$two                     address,
        p$one   based ptr$one       byte,
        p$two   based ptr$two       byte,
        stacktop                    address,
        ret$addr based stacktop     address,
        pcb$ptr                     address,
        pcb$value based pcb$ptr     address,
        counter                     byte,
        move$cnt                    address,
        ret$value                   address,
```

```
        testvalue                   address,
        tad1                        address,
        tad2                        address,
        tad3                        address,
        r$ptr                       byte,
        overflow                    byte,
        signlf$no                   byte,
        e$flag                      byte,
        s$flag                      byte,
        zero$result                 byte;


 /*cp/m interface routines*/

mon1:procedure(function,parameter);
   declare function byte,
           parameter address;
   goto bdos;
end mon1;

mon2:procedure(function,parameter) byte;
   declare function byte,
           parameter address;
    goto bdos;
end mon2;

mon3:procedure;
   goto boot;
end mon3;

printchar:procedure(char);
   declare char byte;
   call mon1(2,char);
end printchar;

print:procedure(buffer);
   declare buffer address;
   call mon1(9,buffer);
end print;


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

 /* procedures for build */

open$int$file:procedure;
   fnp(9) = 'a';
   fnp(10) = 'l';
   fnp(11) = 'n';
   fnp(32) = 0;
   if mon2(15,filename) = 255 then
      do;
        call print(.'nl    $');
        call crlf;
        call mon3;
      end;
end open$int$file;

read$int$file:procedure byte;
   return mon2(20,filename);
end read$int$file;


 /*global procedures*/

incbuf:procedure;
   if(buff := buff+1) >= diskbuffend then
     do;
       buff = diskbuffloc;
```

```
        if read$int$file <> 0 then
           char = 7fh;
     end;
end incbuf;

sto$char$inc:procedure;
   b = char;
   codeptr = codeptr+1;
end sto$char$inc;

next$char:procedure byte;
   call incbuf;
   return curchar := char;
end nextchar;

get$two$bytes:procedure;
   b(1) = next$char;
   b = next$char;
end get$two$bytes;

inc$codeptr$two:procedure;
   codeptr = codeptr + 1 + 1;
end inc$codeptr$two;

getparm:procedure address;
   return shl(double(nextchar),8) + nextchar;
end getparm;

pack$decimal:procedure(loc$one,loc$two);
   declare switch byte,
           loc$one address,
           loc$two address;

   pack:procedure;
   if (switch := not switch) then
      p$two = shl(p$one-30h,4); /* odd */
   else
      do;
        p$two = p$two or (p$one-30h); /* even */
        ptr$two = ptr$two + 1;
      end;
   end pack;

   ptr$one = loc$one;
   ptr$two = loc$two;
   switch = false;
   temp1 = ptr$two;
   p$two = 0;
   ptr$two = ptr$two + 1;
   temp2 = ptr$two;
   p$two = 0;
   ptr$two = ptr$two + 1;
   do while p$one <> 0;
     if (p$one >= '0') and (p$one <= '9') then
       do;
         call pack;
         t1 = t1 + 1;
       end;
     else
       if p$one = '.' then
        t2 = t1; /* left offset to decpt */
     else
       do;
         error$flag = true;
         return;
       end;
   if ((ptr$one := ptr$one+1) >= diskbuffend) and bld$flag then
      do;
        ptr$one = diskbufflo;
        if read$int$file <> 0 then
          do;
            p$one(2) = 7fh;
            return;
```

```
            end;
         end;
      end;
      if switch then
        do;
          t1 = t1 + 1;
          ptr@two = ptr@two + 1;
        end;
      t2 = t1 - t2; /* right offset to decpt */
      t1 = t1 / 2;
      p@two = t1 + 2;
      ptr@two = ptr@two + 1;
      p@two = positive; /* this field used for dec sign */
end pack@decimal;

 /* procedures for interp */

readchar:procedure byte;
         return mon2(1,0);
end readchar;

read:procedure(a);
     declare a address;
     call mon1(10,a);
end read;

open:procedure byte;
     return mon2(16,fileaddr);
end open;

close:procedure byte;
      return mon2(16,fileaddr);
end close;

diskread:procedure byte;
         return mon2(20,fileaddr);
end diskread;

diskwrite:procedure byte;
          return mon2(21,fileaddr);
end diskwrite;

make:procedure byte;
     return mon2(22,fileaddr);
end make;

delete:procedure;
       call mon1(19,fileaddr);
end delete;

setdma:procedure;
       call mon1(26,buffer);
end setdma;

select:procedure(drive);
   declare drive byte;
   call mon1(14,drive);
end select;

mask:procedure(location) address;
   declare location address,
           l based location address;
   return l and 0bfffh;
end mask;

check@int@sign:procedure(value) byte;
   declare value address;
   if rol(high(value),1) [illegible]
      return negative;
   else
      return positive;
end check@int@sign;
```

```
check$int:procedure(stack$loc) byte;
   declare stack$loc address;
   if not(rol(high(stack$loc),2)) then
      return true;
   else return false;
end check$int;

check$temp:procedure(stack$loc) byte;
   declare stack$loc address;
    if (rol (high(stack$loc),1)) and
          (rol(high(stack$loc),2)) then
            return true;
      else return false;
end check$temp;

set$up$neg:procedure;
   if not check$int$sign(ara) then
      ara = ara or 4000h;
   if not check$int$sign(arb) then
      arb = arb or 4000h;
end set$up$neg;

check$neg:procedure;
   if not check$int$sign(arb) then
      arb = arb and 0bfffh;
end check$neg;

pop$stack:procedure;
           declare num byte;
           ra = rb;
           if check$temp(arb) then
              rb = rb - (bri-2);
           else rb = rb - 2;
end pop$stack;

push$stack:procedure(num);
           declare num byte;
           rb = ra;
           ra = ra + num;
end push$stack;

move:procedure(source,dest,count);
     declare source address,
             dest   address,
             count  byte,
             schar based source byte,
             dchar based dest byte;
     do while(count := count - 1) <> 255;
        dchar = schar;
        source = source + 1;
        dest = dest + 1;
     end;
end move;

fill:procedure(dest,char,n);
    /*fill locations starting at dest with char for n bytes*/
     declare dest address,
             n    byte,
             d    based  dest  byte,
             char byte;
     do while (n:=n-1) <> 0ffh;
        d = char;
        dest = dest + 1;
     end;
end fill;

error$msg:procedure(msg);
   declare msg address;
   call print$char(' ');
   call print$char(high(msg));
   call print$char(low(msg));
end error$msg;
```

```
warning:procedure(warncode);
   declare warncode address;
   call crlf;
   call print(. 'warning $');
   call error$msg(warncode);
end warning;


error:procedure(errcode);
   declare errcode address;
   call crlf;
   error$flag = true;
   call print(. 'error $');
   call error$msg(errcode);
end error;
 /*****************************************************************/
 /*    file processing routines for use with cp/m                 */
 /*****************************************************************/

initialize$disk$buffer:procedure;
   call fill(buffer,eoffiller,128);
end initialize$disk$buffer;

buffer$status$byte:procedure byte;
   return fcb(33);
end buffer$status$byte;

set$buffer$status$byte:procedure(status);
   declare status byte;
   fcb(33) = status;
end set$buffer$status$byte;

write$mark:procedure byte;
   return buffer$status$byte;
end write$mark;

set$write$mark:procedure;
   call set$buffer$status$byte(buffer$status$byte or 01h);
end set$write$mark;

clear$write$mark:procedure;
   call set$buffer$status$byte(buffer$status$byte and 0feh);
end clear$write$mark;

active$buffer:procedure byte;
   return shr(buffer$status$byte,1);
end active$buffer;

set$buffer$inactive:procedure;
   call set$buffer$status$byte(buffer$status$byte and 0fdh);
end set$buffer$inactive;

set$buffer$active:procedure;
   call set$buffer$status$byte(buffer$status$byte or 02h);
end set$buffer$active;

set$random$mode:procedure;
   call set$buffer$status$byte(buffer$status$byte or 80h);
end set$random$mode;

random$mode:procedure byte;
   return rol(buffer$status$byte,1);
end random$mode;

disk$eof:procedure;
   if eofaddr = 0 then
      call error('ef');
   rc = eofaddr + 1;
   ra = eofra;
   rb = eofrb;
   goto eofexit;
end disk$eof;
```

```
fill$file$buffer:procedure:
   if diskread = 0 then
      do:
        call set$buffer$active:
        return:
      end:
   if not random$mode then
     do:
        call disk$eof:
        return:
     end:
   call initialize$disk$buffer:
   call set$buffer$active:
   return:
end fill$file$buffer:

write$disk$if$req:procedure:
   if write$mark then
   do:
     if diskwrite <> 0 then
        call error('dw'):
     call clear$write$mark:
     if random$mode then
        call set$buffer$inactive:
     else
        call initialize$disk$buffer;
   end:
   record$pointer = buffer;
end write$disk$if$req:

at$end$disk$buffer:procedure byte:
   return (record$pointer := record$pointer + 1) >= buffer$end.
end at$end$disk$buffer:

var$block$size:procedure byte:
   return blocksize <> 0:
end var$blocksize;

store$rec$ptr:procedure;
   fcbadd(18) = record$pointer:
end store$rec$ptr;

write$a$byte:procedure(char):
   declare char byte:
   if var$block$size and (byteswritten := byteswritten+1)
         > blocksize then
      call error('er');
   if at$end$disk$buffer then
       call write$disk$if$req:
   if not active$buffer and random$mode then
      do:
        call fill$file$buffer:
        fcb(32) = fcb(32) - 1; /* reset record no */
      end:
   nextdiskchar = char:
   call set$write$mark:
end write$a$byte:

set$file$addr:procedure:
   prt$addr = mask(ra):
   fileaddr = prt$entry:
   eofaddr = fcbadd(19):
   call pop$stack:
end set$file$addr:

set$file$pointers:procedure:
   buffer$end = (buffer := fileaddr + 33) + diskrecsize:
   recordpointer = fcbadd(18):
   blocksize = fcbadd(17):
   call setdma:
end set$file$pointers:

setup$file$extent:procedure.
```

```
    if open = 255 then
       do:
          if make = 255 then
            call error('me');
       end;
end setup$file$extent;

disk$open:procedure;
    declare (filename,buff,blk$size) address,
            (i,j) byte,
            char based buff byte;
    inc$j:procedure byte;
         return (j := j + 1);
    end inc$j;

    prt$addr = mask(ra);
    call pop$stack;
    blk$size = ara;
    call pop$stack;
    buff = mask(ra);
    call pop$stack;
    buff = buff - char;
    fileaddr,prt$entry = ra;
    buffer = ra + 0;
    call fill ((filename := ra + 1),' ',11);
    if char(2) = ':' then
      do;
        bra = (char(1) and 0fh) - 1;
        i = char - 2;
        buff = buff + 2;
      end;
    else
      i = char;
    if i > 12 then
      i = 12;
    buff = buff + 1;
    j = 255;
    do while (char(inc$j) <> '.') and (j < 8);
       end;
    call move (buff,filename,j);
    if i > inc$j then
      call move (.char(j),filename+8,i-j);
    call setup$file$extent;
    ara(19) = 0;
    ara(18) = ra + 256;
    ara(17) = blk$size;
    ra = ra + 168;
end disk$open;

set$eof$stack:procedure;
    eofra = ra;
    eofrb = rb;
end set$eof$stack;

setup$disk$io:procedure;
    call set$file$addr;
    call set$file$pointers;
    bytes$written = 0;
    firstfield = true;
end setup$disk$io;

random$setup:procedure;
    declare bytecount address,
            record address,
            extent byte;
    if not var$block$size then
       call error('ru');
    ara = ara - 1;
    call set$random$mode;
    call set$buffer$inactive;
    call write$disk$if$req;
    bytecount = blocksize * ara;
    recordpointer = (bytecount and 7fh) + buffer - 1;
```

```
   call store$rec$ptr;
   record = shr(bytecount,7);
   extent = shr(record,7);
   if extent <> fcb(12) then
     do;
        if close = 255 then
           call error('ce');
        fcb(12) = extent;
        call setup$file$extent;
     end;
   fcb(32) = low(record) and 7fh;
   call pop$stack;
end random$setup;

get$disk$char:procedure byte;
   if at$end$disk$buffer then
     do;
       call write$disk$if$req;
       call fill$file$buffer;
     end;
   if not active$buffer then
     call fill$file$buffer;
   if nextdiskchar = eoffill$c then
     call disk$eof;
   return nextdiskchar;
end get$disk$char;

disk$close:procedure;
   call set$file$pointers;
   call write$disk$if$req;
   prt$addr = mask(ra);
   if close = 255 then
     call error('ce');
   prt$entry = 0;
end disk$close;

clear$print$buff:procedure;
   call fill((printbuffer := printbufferloc),' ',72);
end clear$print$buff;

dump$print$buff:procedure;

     declare temp address,
             char based temp byte;
     temp = printbuffend;
     do while char = ' ';
        temp = temp - 1;
        end;
    call crlf;
    do printbuffer = printbufferloc to temp;
       call printchar(printpos);
       end;
    call clear$print$buff;
end dump$print$buff;

load$print$buff:procedure(char);
   declare char byte;
   printpos = char;
   if(printbuffer := printbuffer +1) >
      printbuffend then
      call dump$print$buff;
end load$print$buff;

output:procedure(dest,char);
   declare dest byte,
           char byte;
   if dest = console then
      call print$char(char);   /* to console */
   else
      call write$a$byte(char);  /* to disk */
end output;

write$doc:procedure(source,dest);
```

```
    declare source address,
            num byte,
            num$digits byte,
            char byte,
            index byte,
            dest byte,
            count byte;
    store$one:procedure;
       if (switch := not switch) then
         do:
           char = shr(h$byte,4) or '0';
           if (count=0) and (char='0') then;
           else
           call output(dest,char);
         end;
       else
         do:
            call output(dest,(h$byte and 0fh) or '0');
            hold = hold + 1;
         end;
      count = count + 1;
    end store$one;

   hold = source;
   sign = h$byte(1);
    if sign or 00h then
       call output(dest,' ');
    else call output(dest,'-');
    count = 0;
    hold = hold - h$byte;
    num$digits = h$byte * 2;
    hold = hold + 1;
    num = num$digits - h$byte;
    switch = false;
    hold = hold + 1;
    do index = 1 to num$digits+1;
       if count = num then
          do;
          call output(dest,'.');
          count = 100;
          end;
       else
          call store$one;
    end;
end write$dec;

write$str:procedure(hold,dest);
     declare hold address,
                h based hold byte,
                dest byte,
                index byte;
    hold = hold - h;
       do index = 1 to h;
          call output(dest,h(index));
       end;
end write$str;

write$int:procedure(value,dest);
   declare value address, i byte, count byte;
   declare declut(5) address initial(10000,1000,100,10,1);
   declare (flag,dest) byte;
   sign = check$int$sign(value);
  if sign = negative then
  do;
     value = -value and 0bfffh; /* mask off next to left bit */
     call output(dest,'-');
  end;
  else call output(dest,' ');
  flag = false;
  do i = 0 to 4;
      count = 30h;
      do while value >= declut(i);
         value = value - declut(i);
```

```
            flag=true;
            count=count+1;
         end;
         if flag or (i >= 4) then
            call output(dest,count);
         else
            call output(dest,' ');
      end;
   end write$int;

   write$to$disk:procedure(type);
      /* type 0-integer,type 1-decimal,type 2-string */
      declare type byte;
      if type = 0 then
        call write$int(ara,1);

      if type = 1 then
        call write$dec(ara,1);

      if not firstfield then /* separate fields with commas */
        call write$a$byte(',');
      else firstfield = false;

      if type = 2 then
        do;
          call write$a$byte(quote); /* strings put in quotes */
          call write$str(ara,1);
          call write$a$byte(quote); /* add trailing quotes */
        end;
      call pop$stack;
   end write$to$disk;

   concatenate:procedure;
      declare (size1,size2,t$size) byte;
      temp1 = mask(ra);
      call pop$stack;
      if not check$temp(ara) then
        do;
          temp2 = mask(ra);
          temp2 = temp2 - t2;
          size2 = t2 + 1;
          call move(temp2,ra,size2);
        end;
      else
        do;
           size2 = bra;
           ra = ra - bra;
         end;
      temp1 = temp1 - t1;
      size1 = t1;
      call move(temp1+1,ra+size2,size1);
      t$size = size1 + size2;
      bra = t$size - 1;
      ra = ra + t$size;
      bra = t$size;
      bra(1) = 0c0h; /* set temp bits */
   end concatenate;

   convert$to$int:procedure(loc,size) address;
      declare decint(5) address initial (10000,1000,100,10,1),
              loc address,
              hold address,
              h based hold byte,
              num address,
              (i,j,size) byte;
      num = 0;
      j = 5;
      hold = loc + size - 1;
      if size > 5 then
         call error('io');
      else
        do;
          do i = 1 to size;
```

```
            num = num + (h - 30h) * decint(j:=j-1);
            hold = hold - 1;
         end;
      if num <= 16383 then
         return num;
      else
         call error('lo');
       end;
end convert$to$int;

one$left:procedure;
   declare ctr byte;
   if shr(b$byte(1),4) = 0 then
        do:
          do ctr = 1 to 9;
          b$byte(ctr) = shl(b$byte(ctr),4) or shr(b$byte(ctr+1),4);
          end;
        end;
        else no$shift = true;
end one$left;

one$right:procedure;
   declare ctr byte,
           index byte;
   ctr = 11;
   do index = 1 to 10;
      ctr = ctr-1;
      b$byte(ctr) = shr(b$byte(ctr),4) or shl(b$byte(ctr-1),4);
   end;
end one$right;

shift$right:procedure(count);
   declare(count,ctr) byte;
   do ctr = 1 to count;
      call one$right;
   end;
end shift$right;

shift$left:procedure(count);
   declare count byte;
   noshift = false;
   do ctr = 1 to count;
      call one$left;
      if noshift then
            return;
   end;
end shift$left;

leading$zeroes:procedure(addr) byte;
   declare count byte,
           ctr byte,
           addr address;
   count = 0;
   base = addr;
   do ctr = 1 to 9;
     if (b$byte(ctr) and 0f0h) <> 0 then
       return count;
     count = count + 1;
     if (b$byte(ctr) and 0fh) <> 0 then
       return count;
     count = count + 1;
   end;
   if base = .r0 then
     do;
       call error('dz');
       b$byte(9) = 10h;
       decpt0 = 1;
       return 16;
     end;
   return count;
end leading$zeroes;

r1$greater:procedure byte;
```

```
   declare (i,ctr) byte;
   do ctr = 1 to 9;
     if r1(ctr) > (1 := (99h-r0(ctr))) then
       return true;
     if r1(ctr) < 1 then
       return false;
   end;
   e$flag = true;
   return true;
end r1$greater;

allign:procedure;
   declare(x,y) byte;

   right$op:procedure(addr);
      declare addr address;
      if noshift  then
         do;
           base = addr;
           call shift$right(y := x - ctr);
         end;
   end right$op;

   y = 0;
   if dec$pt0 > dec$pt1 then
     do;
       base = .r1;
       call shift$left(x := decpt0 - decpt1);
       decpt1 = decpt1 + ctr-1;
       call right$op(.r0);
       decpt0 = decpt0 - y;
    end;
    else
      do;
        base = .r0;
        call shift$left(x := decpt1 - dec$pt0);
        decpt0 = decpt0 + ctr-1;
        call right$op(.r1);
        decpt1 = decpt1- y;
      end;
end allign;

add$r0:procedure(second,dest);
   declare (second, dest) address, (index,cy , a , b, i ) byte;
   hold = second;
   base = dest;
   cy = 0;
   ctr = 10;
   do index = 1 to 11;
     a = r0(ctr);
     b = h$byte(ctr);
     i = dec(a + cy);
     cy = carry and 1;
     i = dec(i + b);
     cy = (cy or carry) and 1;
     b$byte(ctr) = i;
     ctr = ctr - 1;
   end;
   if cy then
   do;
     ctr = 10;
     do index = 1 to 11;
     i = b$byte(ctr);
     i = dec(i + cy);
     cy = carry and 1;
     b$byte(ctr) = i;
     ctr = ctr - 1;
     end;
   end;
end add$r0;

compliment:procedure(numb);
   declare numb byte;
```

```
    do case numb;
       hold = .r0;
       hold = .r1;
       hold = .r2;
    end;
    if sign$0(numb) then sign0(numb) = negative;
    else sign$0(numb) = positive;
    do ctr = 0 to 10;
       h$byte(ctr) = 99h - h$byte(ctr);
    end;
end compliment;

right$justify:procedure(numb);
    declare (numb,i) byte;
    do case numb;
      base = .r0;
      base = .r1;
      base = .r2;
    end;
    i = 0;
    do while (((i:=i+2) < decpt0(numb)) and (b$byte(9)=0));
       call shift$right(2);
    end;
    decpt0(numb) = decpt0(numb) - (i-2);
end right$justify;

set$mult$div:procedure;
    noshift = false;
    if (sign0 and sign1) or
       (not sign0 and not sign1) then
       sign2 = positive;
    else sign2 = negative;
    call fill (.r2,0,10);
end set$mult$div;

add$series:procedure(count);
    declare (i,count) byte;
    do i = 1 to count;
      call add$r0(.r2,.r2);
    end;
end add$series;

multiply:procedure(value);
    declare value byte;
    if value <> 0 then
      do;
        if noshift then
            call error('ov');
       call add$series(value);
      end;
      base = .r0;
    call one$left;
end multiply;

divide:procedure;
    declare (i,j,k,x,lz0,lz1) byte;
    call set$mult$div;
    s$flag = true;
    e$flag = false;
    if (lz0 := leading$zeroes(.r0)) <>
       (lz1 := leading$zeroes(.r1)) then
       do;
         if lz0 > lz1 then
           do;
              base=.r0;
             call shift$left(i:=lz0-lz1);
             decpt0 = decpt0 + i;
             x = lz1;
           end;
        else
          do;
            base = .r1;
            call shift$left(i:=lz1-lz0);
```

```
            dec$pt1 = decpt1 + 1;
            x = lz0;
          end;
        end;
    else x = lz1;
        decpt2 = 19 - x + decpt1 - decpt0;
        call compliment(0);
        do i = x to 19;
           j = 0;
           do while r1$greater and s$flag;
             call add$r0(.r1,.r1);
             j = j + 1;
             if e$flag = true then s$flag = false;
           end;
           k = shr(i,1);
           if 1 then
             r2(k) = r2(k) or j;
           else r2(k) = r2(k) or shl(j,4);
           base = .r0;
           call one$right;
        end;
 end divide;

 check$result:procedure;
    if r2 = 99h then
       do;
          call compliment(2);
          sign2 =  sign2 xor 1;
       end;
       else
          do;
          if r2 <>0 then
             call error('ov');
          if not s$flag then
             sign2=positive;
 end;
 end check$result;

 check$sign:procedure;
    s$flag=false;
    if sign0 and sign1 then
    do;
      sign2 = positive;
      return;
    end;
    sign2 = negative;
    if not sign0 and not sign1 then
        do;
           s$flag=true;
           return;
         end;
    if sign0 then call compliment(1);
    else call compliment(0);
end check$sign;

add:procedure;
   call check$sign;
   call add$r0(.r1,.r2);
   call check$result;
   decpt2 = decpt0;
end add;

cpy$reg2$onstack:procedure;
   declare count byte,
            l    byte;
    call right$justify(2);
  call pop$stack;
   count = 0;
   base = .r2;
    l = 10 - (decpt2+1)/2;
   do while (b$byte = 0) and (count < l);
       base = base + 1;
      count = count+ 1;
```

```
   end;
  ra = rb + 2;
   bra(0) = (count := 10 - count);
   bra(1) = dec$pt2;
   call move(base, ra+2,count);
   bra(count+2) = count+2;
   bra (count+3) = sign2 or 0c0h; /* set sign and temp bits */
    ra = ra + count + 2;
end cpy$reg2$onstack;

load$reg:procedure(source,reg$num);
   declare source address,
           reg$num byte,
           count   byte;
   hold = source;
   if not check$temp(h$addr) then
        hold = mask(hold);
   sign0(reg$num) = h$byte(1);
   hold = hold - h$byte;
    count = h$byte;
   do case reg$num;
      base = .r0;
      base = .r1;
      base = .r2;
   end;
   call fill(base,0,11);
   hold = hold + 1;
   dec$pt0(reg$num) = h$byte;
   hold = hold + 1;
   call move(hold,base+10-count,count);
end load$reg;

set$up$regs:procedure;
   noshift = false;
   call load$reg(ra,0);
   call load$reg(rb,1);
   call right$justify(0);
   call right$justify(1);
end set$up$regs;

step$ins$cnt:procedure(num);
              declare num byte;
              rc=rc+num;
end step$ins$cnt;


branch$absolute:procedure;
   call step$ins$cnt(1);
   rc=twobyteoprand - 1;
end branch$absolute;


get$code$addr:procedure(offset) address;
   declare offset address;
   return codebase + offset;
end get$code$addr;

get$prt$addr:procedure(offset) address;
   declare offset address;
   return prtbase + offset;
end get$prt$addr;

load:procedure(addr);
    declare addr address,
            a based addr address;
      prt$addr = mask(addr);
      a = prt$entry;
end load;

store$dec:procedure(source,dest);
   declare (source,dest,dest$sign) address,
           (amt$sto,avail$sto,sig$bytes) byte,
           s based source address,
```

```
            d based dest address,
            decpt byte,
            sign based dest$sign byte;
    if check$temp(s) then
       temp1 = source;
    else temp1 = mask(source);
    temp2=mask(dest);
    avail$sto = t2 - 2;
    dest$sign = temp2+1;
    t2(1) = t1(1) and 01h; /*mask off temp bits */
    source= (temp1:=temp1-t1);
    dest=temp2-t2;
    amt$sto = t1;
    decpt = t1(1);
          sig$bytes = ((amt$sto * 2 - decpt) + 1) / 2;
    if amt$sto <= avail$sto then
       do;
         call fill(dest,00h,t2);
         call move(source,dest,t1+2);
       end;
      else if sig$bytes <= avail$sto then
             do;
               call move (source,dest,t2);
               temp2 = dest;
               t2 = avail$sto;
               t2(1) = (avail$sto - sig$bytes) * 2;
                if  decpt then
                    t2(1) = t2(1) + 1;
               call warning('il');
             end;
     else
        do;
          hold = dest;
          h$addr = 0101h;
          h$byte(2) = 10h;
          sign = positive;
          call error('sl');
        end;
end store$dec;

store$int:procedure(dest,value);
   declare (dest,value) address;
   prt$addr=mask(dest);
   prt$entry=value;
end store$int;

store$str:procedure(source,dest);
    declare (dest,source) address,
            s based source address,
            d based dest address;
   if check$temp(s) then
     temp1 = source;
   else temp1 = mask(source);
   temp2 = mask(dest);
   dest = temp2 - t2;
   source = temp1 - t1;
   if t1 <= t2 then
     do;
       call fill(dest,20h,t2);
       call move(source,dest,t1);
     end;
   else
     do;
       call move(source,dest,t2);
       call warning('so');
     end;
end store$str;

allocate$str:procedure;
   if brs = 0 then
      do;
        call error('nz');
        brs = 10;
```

```
      end;
   call push$stack(bra:=bra+1);
   bra = brb;
end allocate$str;

allocate$dec:procedure;
    declare store byte;
   if bra = 0 then
      do;
        call error('az');
        bra = 9;
      end;
   ctr = bra;
   store = (bra+1)/2+2;
   call push$stack(store);
   bra = store;
   ara = ara or 0100h;
end allocate$dec;

set$up$alloc:procedure;
   prt$addr = mask(ra);
   call pop$stack;
end set$up$alloc;

find$rb:procedure;
    if check$temp(ara) then
        rb = ra - (bra + 2);
    else rb = ra - 2;
end find$rb;

sav$pcb:procedure;
   if move$cnt = 6 then
      return;
   call push$stack(2);
   call move(pcbptr,ra,move$cnt);
   pcb$value(1) = ra;
   ra = ra + move$cnt;
   ara = move$cnt;
end sav$pcb;

unsave:procedure;
   call move(temp1:=ra-ara,hold,ara);
   ra = temp1;
end unsave;

calc$row:procedure;
   declare (index,num,v,a$size,type) byte,
           (num$arrays,num$dim,alloc$len,count,i) byte,
           d (10) byte;
   type = bra; /* 1-int,2-dec,3-str */
   call pop$stack;
   num$arrays = bra;
   call pop$stack;
   num$dim = bra;
   call pop$stack;
   v = 0;
   a$size = 1;
   d(num$dim) = 1;
   if num$dim = 1 then
      do;
        a$size = bra - brb + 1;
        v = brb;
        call pop$stack;
        call pop$stack;
      end;
   else
   do index = 1 to num$dim;
      i = num$dim - index;
      a$size = a$size * (num := bra - brb + 1);
      d(i) = num * d(i+1);
      v = v + brb * d(i+1);
      call pop$stack;
      call pop$stack;
```

```
    end;
     if type = int then
         call push$stack(2);
     else
       alloc$len = bra;
    do count = 1 to num$arrays;
      call step$ins$cnt(1);
      bra = c(1);
      bra(1) = c and 3fh;
      prt$addr = ara + prt$base;
      prt$entry = ra or 4000h; /* set addr bits */
      bra = num$dim;
      if num$dim <> 1 then
      do index = 1 to num$dim -1;
         i = num$dim - index;
         ra = ra + 1;
         bra = d(i);
      end;
      ra = ra + 1;
      bra = v;
      ra = ra + 1;
      bra = alloc$len;
      if type = int then
         do;
           bra = 2;
           ra = ra + 1;
           hold = ra;
           ra = ra + a$size * 2;
           h$addr = ra;
         end;
      else
         if type = decl then
            do;
              bra = (bra+1)/2 + 4;
              ra = (hold := ra+1) + 2;
              bra = alloc$len;
              do index = 1 to a$size;
                 call allocate$dec;
                 call push$stack(2);
                 bra = ctr; /* reset to allocated length */
              end;
              call pop$stack;
              h$addr = ra;
            end;
       else
          if type = str then
             do;
               bra = bra + 3;
               ra = (hold := ra + 1) + 2;
               bra = alloc$len;
               do index = 1 to a$size;
                  call allocate$str;
                  call push$stack(2);
                  ara = arb;
               end;
               call pop$stack;
               h$addr = ra;
             end;
      call step$ins$cnt(1);
    end; /*count*/
end calc$row;

calc$sub:procedure;
   declare array$addr address,
           location   address,
           a$byte based array$addr byte,
           a$addr based array$addr address,
           (i,num$dim) byte,
           offset address;
   array$addr = mask(ra);
   call pop$stack;
   offset = ara;
   num$dim = a$byte;
```

```
   do i = 2 to num$dim;
      call pop$stack;
      array$addr = array$addr + 1;
      offset = ara * a$byte + offset;
   end;
   array$addr = array$addr + 1;
   offset = (offset - a$byte+1) * a$byte(1);
   array$addr = array$addr + 2;
   if (location:=array$addr+offset) > a$addr then
      call error('ab');
   ara = location or 4000h;
end calc$sub;

decrement$blk:procedure(num);
   declare num byte;
   ra = blk((blk$level:=blk$level-num) + 1);
   call find$rb;
end decrement$blk;

add$int:procedure(int1,int2) address;
   declare (int1,int2) address;
   return int1 + int2;
end add$int;

sub$int:procedure(int1,int2) address;
   declare (int1,int2) address;
   return int1 - int2;
end sub$int;

mul$int:procedure(int1,int2) address;
   declare (int1,int2) address;
   return int1 * int2;
end mul$int;

div$int:procedure(int1,int2) address;
   declare (int1,int2) address;
   return int1 / int2;
end div$int;


exit$interp:procedure;
   call crlf;
   call mon3;
end exit$interp;

console$read:procedure;
   call crlf;
   call printchar('-');
   call printchar('>');
   call printchar(' ');
   call read(.inputbuffer);
   if buff$space(1) = contz then
      call exit$interp;
      num$read = buff$space;
   conbuffptr = .buff$space;
   buff$space(buff$space+1) = eolchar;
end console$read;

more$con$input:procedure byte;
   return conbuffptr < .buff$space( num$read);
end more$con$input;

console$input$error:procedure;
   rc = rereadaddr; /* reset program counter */
   call warning ('il');
   goto error$exit; /* return to outer level */
end console$input$error;

get$con$char:procedure byte;
   conbuffptr = conbuffptr + 1;
   return con$char;
end get$con$char;
```

```
next$input$char:procedure byte;
   if inputtype = 0 then /* read from disk */
     /*  do forever;
         if (buff$space(inputindex) := getdisk$char) = 1f then
           do;
             if var$blocksize then
               call error('re');
           end;
         else
           return next$disk$char;
       end  */;
   if input$type = 1 then /* input frpm console */
      return get$con$char;
end next$input$char;

get$field:procedure;
   declare hold byte,
           delim byte;
   field$length = 0;
   do while (hold := next$input$char) = ' ';
      end;
   if inputtype = 0 then
      inputptr = .buff$space;
   if inputtype = 1 then
      inputptr = conbuffptr;
   if hold <> quote then
      delim = ' ';
   else
    do;
      delim = quote;
      hold = next$input$char;
    end;
   do while (hold <> delim) and (hold <> eolchar);
      field$length = field$length + 1;
      hold = next$input$char;
   end;
end get$field;

get$int$field:procedure;
   declare sign byte;
   call get$field;
   if input$char = '-' then
      do;
        sign = 1;
        inputptr = inputptr + 1;
        field$length = field$length - 1;
      end;
   else
   if input$char = '+' then
      do;
        sign = 0;
        inputptr = inputptr + 1;
        field$length = field$length - 1;
      end;
   else
     sign = 0;
   call push$stack(2);
   ara = convert$to$int(inputptr,field$length);
   if error$flag then
      call console$input$error;
   if sign then
     ara = -ara and 0bfffh; /* set neg bit and conv to neg int
end get$int$field;

get$str$field:procedure;
   call get$field;
   con$char = field$length+1;
   input$ptr = conbuffptr - con$char;
   input$char = field$length;
   call push$stack(2);
   ara = conbuffptr or 4000h;
end get$str$field;
```

```
get$dec$field:procedure:
   call get$field;
   call push$stack(2);
   con$char = 0; /* set binary 0 as end of dec */
   if input$char = '+' then
      do:
         sign = 1;
         inputptr = inputptr + 1;
      end;
    else
   if input$char = '-' then
      do;
         sign = 0;
         inputptr = inputptr + 1;
       end;
   else
         sign = 1;
   call pack$decimal(inputptr,ra);
   if error$flag then
      call console$input$error;
   ra = ptr$two - 1;
   bra(1) = sign or 0c0h; /* set sign and temp bits */
end get$dec$field;

initialize$execute:procedure:
  stacktop=3330h;
  mcd, rc = codebase;
  mpr = prtbase;
  st, sb = stackbase;
  ra = (rb := sb) + 2;
  bld$flag = false;
end initialize$execute;

 /*set up machine*/

call print('algol-m interpreter-vers 1.0$');
call crlf;
call open$int$file;
bld$flag = true;
call incbuf; call incbuf; /* skip codesize */
prtbase = .memory;
codebase = getparm + prtbase;
codeptr=codebase;

 /*load machine*/

do while next$char <> 7fh;
if curchar >= 128  then
   do;
     call sto$char$inc;
     call incbuf;
     call sto$char$inc;
   end;
else
   if curchar = str then
      do;
        call sto$char$inc;
        temp1 = codeptr;
        char = 0; /* set initial length to zero */
        call sto$char$inc;
        do while next$char <> 0;
           call sto$char$inc;
           t1 = t1+1;
        end;
        char = t1+1;
        call sto$char$inc;
        char = 0; /*must make str lenght an addr quantity */
        call sto$char$inc;
      end;
else
    if curchar = int then
     do;
       call sto$char$inc;
```

```
        hold = buff + 1;
        field$length = 0;
        do while next$char <> 0;
           field$length = field$length + 1;
        end;
        a = convert$to$int(hold,field$length);
        codeptr = codeptr + 2;
      end;

else
      if curchar = deci then
      do;
        call sto$char$inc;
        call incbuf;
        call pack$decimal(buff,codeptr);
        codeptr = ptr$two + 1;
        buff = ptr$one + 2;
   end;
   else
      do;
         call sto$char$inc;
         if (curchar = brs) or (curchar = bsc) then
              do;
                call get$two$bytes;
                a = a + codebase;
                call inc$codeptr$two;
             end;
      else
         if (curchar = iml) or (curchar = dcb) then
          do;
            call incbuf;
            call sto$char$inc;
          end;
    else
       if curchar = im2 then
          do;
            call incbuf;
            call sto$char$inc;
            call incbuf;
            call sto$char$inc;
          end;
      end;
end;

stackbase = codeptr;

 /* start of interp */

execute:procedure;
    do forever;
        if rol(c,1) then /* must be lit or lit-lod */
            do;
              call push$stack(2);
              bra = c(1); /* load in reverse order */
              bra(1) = c and 3fh; /* mask   bits  10  */
              ara = (ara + prt$base) or 4000h; /* set 01 addr bits */
              if rol(c,2) then call load(ra);
              call step$ins$cnt(1);
            end;
        else
do case c;

 /* 0 case 0 not used*/
            ;

 /* 1 str */
          do;
            call push$stack(2);
            call step$ins$cnt(1);
            rc = rc + c + 1;
            ara = rc or 4000h;
            call step$ins$cnt(1);
          end;
```

```
/* 2 int  */
        do;
          call push$stack(2);
          call step$ins$cnt(1);
          ara = two$byte$oprand;
          call step$ins$cnt(1);
        end;

/* 3 xch  */
        do;
          hold = ara;
          ara = arb;
          arb = hold;
        end;

/* 4 lod  */
      call load(ra);

/* 5 dcb  */
        do;
          call step$ins$cnt(1);
          call decrement$blk(c);
        end;

/* 6 dmp */
        call crlf;

 /* 7 xit */
        return;

/* 8 ald */
      do;
        call set$up$alloc;
        call allocate$dec;
        prt$entry=ra or 4000h;
      end;

/* 9 als */
      do;
        call set$up$alloc;
        call allocate$str;
        prt$entry=ra or 4000h;
      end;

/*10 ald */
        do;
        call set$up$alloc;
          call allocate$dec;
          prt$entry=ra or 4000h;
          call push$stack(2);
          bra = ctr;
        end;

/*11 als */
        do;
          call set$up$alloc;
          call allocate$str;
          prt$entry=ra or 4000h;
          call push$stack(2);
          ara = arb;
        end;

/*12 adi */
        do;
          call set$up$neg;
          arb = add$int(arb,ara);
          call check$neg;
          call pop$stack;
        end;

/*13 add */
        do;
```

```
          call set$up$regs:   /* puts values of top two items */
                              /* in reg0 and reg1 respectively*/
          call allign:
          call add;
          call cpy$reg2$onstack:
        end;

/*14 sbi */
        do;
          call set$up$neg:
          arb = sub$int(arb,ara):
          call check$neg:
          call pop$stack:
        end;

/* 15 sbd */
        do;
          call set$up$regs:   /* puts values of top two items */
                              /* in reg0 and reg1 respectively*/
          call allign:
          call compliment(0):
          if sign0 then sign0 = negative:
          else sign0 = positive;
          call add;
          call cpy$reg2$onstack:
        end:

/*16 mpi */
        do:
          call set$up$neg:
          arb = mul$int(arb,ara):
          call check$neg:
          call pop$stack:
        end:

/*17 mpd */
        do:
         call set$up$regs:   /* puts values of top two items */
                             /* in reg0 and reg1 respectively*/
         declare (i,index) byte;
         call set$mult$div:
         decpt2 = decpt0 + decpt1;
         i = 10;
         do index = 1 to 10;
            call multiply(r1(i:=i-1) and 0fh);
            call multiply(shr(r1(i),4));
         end;
         call cpy$reg2$onstack:
        end;

/*18 dvi */
        do:
          call set$up$neg:
          arb = div$int(arb,ara):
          call check$neg:
          call pop$stack:
        end;

/*19 dvd */
        do:
          call set$up$regs:   /* puts values of top two items */
                              /* in reg0 and reg1 respectively*/
          call divide:
          call cpy$reg2$onstack:
        end;

/*20      */
/* not used */
        :

/*21      */
/* not used */
        :
```

```
/*22 neg */
      do;
        if check$int(ara) then
           ara = -ara and 0bf'fb;
        else if check$temp(ara) then
            ara = ara xor 0100h; /*change sign bit*/
        else
            do;
              hold = mask(ra);
              h$addr = h$addr xor 0100h;
            end;
      end;

/*23 cll */
/* not implemented */
       ;

/*24 cl2 */
/* not implemented */
       ;

/*25 deci */
       do;
         call push$stack(2);
         call step$ins$cnt(1);
         rc=rc + c + 2;
         ara = rc or 4000h;
         call step$ins$cnt(1);
       end;

/*26 pop */
       call pop$stack;

/*27 lm1 */
      do;
        call push$stack(2);
        call step$ins$cnt(1);
        ara=c;
      end;

/*28 lm2 */
      do;
        call push$stack(2);
        call step$ins$cnt(1);
        bra=c(1);  /* load in reverse */
        bra(1)=c;
        call step$ins$cnt(1);
      end;

/*29  */
/* not used */
      ;

/*30  */
/* not used */
      ;

/*31 cat */
       call concatenate;

/*32 bll */
       blk(blk$level:=blk$level+1) = ra;

/*33 bld */
       call decrement$blk(1);

/*34 bra */
       call branch$absolute;

/*35 bec */
       if bra = 0 then
        call branch$absolute;
```

```
        else
        do;
         call step$ins$cnt(2);
         call pop$stack;
        end;

/*36 lss*/
        do;
          if brb < bra then
            brb=1;
          else brb=0;
          call pop$stack;
        end;

/*37 dlss */
/* not implemented */
        :
/*38 slss */
/* not implemented */
        :
/*39 gtr */
        do;
          if brb > bra then
             brb=1;
          else brb=0;
          call pop$stack;
        end;

/*40 dgtr */
/* not implemented */
        :
/*41 sgtr */
/* not implemented */
        :
/*42 eql */
        do;
          if brb = bra then
             brb=1;
          else brb=0;
          call pop$stack;
        end;
/*43 deql */
/* not implemented */
        :
/*44 seql */
/* not implemented */
        :

/*45 neg */
        do;
          if brb <> bra then
             brb = 1;
          else brb = 0;
          call pop$stack;
        end;
/*46 dneg */
/* not implemented */
        :
/*47 dneg */
/* not implemented */
        :
/*48 geq */
        do;
          if brb >= bra then
             brb = 1;
          else brb = 0;
          call pop$stack;
        end;
/*49 dgeq */
/* not implemented */
        :
/*50 sgeq */
/* not implemented */
```

```
        ;
/*51 leq */
       do;
         if brb <= bra then
             brb = 1;
          else brb = 0;
          call pop$stack;
       end;
/*52 dleq */
/* not implemented */
        ;
/*53 sleq */
/* not implemented */
        ;
/*54 inot */
        do;
          if bra = 0 then
             bra = 1;
          else bra = 0;
        end;
/*55 dnot */
/* not implemented */
        ;
/*56 snot */
/* not implemented */
        ;
/*57 iand */
        do;
          if (bra=0) or (brb=0) then
               brb=1;
           else brb=0;
           call pop$stack;
        end;
/*58 dand */
/* not implemented */
        ;
/*59 sand */
/* not implemented */
        ;
/*60 ior */
       do;
         if (bra=0) and (brb=0) then
             brb=1;
          else brb=0;
          call pop$stack;
       end;
/*61 dor */
/* not implemented */
        ;
/*62 sor */
/* not implemented */
        ;
/*63 wic */
       do;
         call write$int(ara,0);
         call pop$stack;
       end;

/*64 wdc */
      do;
        if check$temp(ara) then
           call write$dec(ra,0);
        else
           call write$dec(mask(ra),0);
        call pop$stack;
      end;

/*65 wsc */
       do;
         if check$temp(ara) then
            call write$str(ra,0);
         else
            call write$str(mask(ra),0);
```

```
          call pop$stack;
        end;

/*66 wid */
        call write$to$disk(0);

/*67 wdd */
        call write$to$disk(1);

/*68 wsd */
        call write$to$disk(2);

/*69 sbr */
        do;
          arb = ara - arb;
          call pop$stack;
        end;

/*70 bra */
        do;
          rc = code$base + ara;
          call pop$stack;
        end;

/*71 row */
        call calc$row;

/*72 sub */
        call calc$sub;

/*73 rci */
        call get$int$field;

/*74 rcd */
        call get$dec$field;

/*75 rcs */
        call get$str$field;

/*76 rdi */
        do;
          inputtype = 0;
          call get$int$field;
        end;

/*77 rdd */
        do;
          inputtype = 0;
          call get$dec$field;
        end;

/*78 rds */
        do;
          inputtype = 0;
          call get$str$field;
        end;

/*79 rcn */
        do;
          inputtype = 1;
          rereadaddr = rc;
          call console$read;
        end;

/*80 ecr */
        if more$con$input then
           call console$input$error;

/*81 sii */
        do;
          call store$int(rb,ara);
          rb = rb - 2;
        end;
```

```
/*82 sdi */
        do;
          call store$dec(ra,rb);
          rb = rb - 2;
        end;

/*83 ssi */
        do;
          call store$str(ra,rb);
          rb = rb - 2;
        end;

/*84 sid */
        do;
          call store$int(rb,ara);
          call pop$stack;
          call pop$stack;
        end;

/*85 sdd */
        do;
          call store$dec(ra,rb);
          call pop$stack;
          call pop$stack;
        end;

/*86 ssd */
        do;
          call store$str(ra,rb);
          call pop$stack;
          call pop$stack;
        end;

/*87 opn */
          call disk$open;

/*88 cls */
        do;
          call set$file$addr;
          call disk$close;
          call pop$stack;
        end;

/*89 rdb */
/* ready sequential block */
        do;
          call setup$disk$io;
          call set$eof$stack;
        end;

/*90 rdf */
/* ready random block */
        do;
          call setup$disk$io;
          call random$setup;
          call set$eof$stack;
        end;

/*91 edr*/
/* end of record for read */
/* advances to next line feed */
        do;
          if var$block$size then
            do while get$disk$char <> lf;
              end;
          call store$rec$ptr;
        end;

/*92 edw */
/* end of record for write */
        do;
          if var$block$size then
```

```
            do while bytes$written < (blocksize - 2);
            call write$a$byte(' ');
            end;
        call write$a$byte(cr);
        call write$a$byte(lf);
        call store$rec$ptr;
      end;

/*93 pro */
      do;
        stacktop = stacktop - 2;
        ret$addr = rc ;
        rc = ara + codebase;
        call pop$stack;
      end;

/*94 sav */
      do;
        declare (i,num) byte;
        pcbptr = mask(ra);
        call pop$stack;
        move$cnt = ara;
        call pop$stack;
        if ara <> 0 then
         do;
           hold = ra;
           counter = 2 * ara + 1;
           do i = 1 to counter;
               call pop$stack;
           end;
           num = hold - ra ;
           if (temp1 := stacktop - num) <= ra then
             call error('mo');
           call move(ra+2,temp1,num);
           call fill(temp1-2,00h,2);
         end;
        else
          call pop$stack;
          blk(blklevel:=blklevel+1)=ra;
          if pcb$value(i) = 0 then
            pcb$value(i) = 1;
          else
            call sav$pcb;
        end;

/*95 sv2 */
      do;
        declare i byte,
                parm$count address;
        tad1 = ra;
        tad2 = rb;
        tad3,ra = stacktop - 2;
        rb = ra - 2;
        parm$count = ara;
        call pop$stack;
        pcbptr = pcbptr + 4 + parm$count * 2;
        do i = 1 to parm$count;
           testvalue = ara;
           call pop$stack;
           if testvalue = int then
              call store$int(i,pcbptr,ara);
           else
             if testvalue = dec1 then
                call store$dec(ra,pcbptr);
             else
               call store$str(ra,pcbptr);
           call pop$stack;
           pcbptr = pcbptr - 2;
        end;
        ra = tad1;
        rb = tad2;
        tad3 = 0;
      end;
```

```
/*96 uns */
        do;
          hold = mask(ra);
          ret$value = h$addr;
          call decrement$blk(1);
          if h$addr(1) <> 1 then
             call unsave;
          else
             h$addr(1) = 0;
          call decrement$blk(1);
          call push$stack(2);
          if check$int(ret$value) then
             ara = ret$value;
          else
            do;
              temp1 = mask(.ret$value);
              call move(temp1 - t1,ra,t1+2);
              ra = ra + t1;
              bra(1) = bra(1) or 0c0h;
            end;
          end;

 /*97 rtn */
        do;
           rc = ret$addr;
           stacktop = stacktop + 2;
        end;

end;  /*end case*/

call step$ins$cnt(1);
error$flag = false;
end;  /* of do for ever */

end execute;

mainline:
   call crlf;
   call initialize$execute;

eofexit:  /* on end of file of current disk file come here */

errorexit:  /* regroup on console input error */
   call execute;
   call exit$interp;

eof
```

LIST OF REFERENCES

1. Aho, A. V. and S. C. Johnson, LR Parsing, Computing Surveys, Vol. 6 No. 2, June 1974.

2. Craig, Alan S. MICRO-COBOL An Implementation of Navy Standard Hypo-Cobol for a Microprocessor-Based Computer System, Masters Thesis, Naval Postgraduate School, March 1977.

3. DeMorgan, R. M. and others Modified Report on the Algorithmic Language ALGOL 60.

4. Digital Research, An Introduction to CP/M Features and Facilities, 1976, Box 579, Pacific Grove, CA., 93959.

5. Digital Research, CP/M Interface Guide, 1976.

6. Eubanks, Gordon E. Jr. A Microprocessor Implementation of Extended Basic, Masters Thesis, Naval Postgraduate School, December 1976.

7. Feldman, P. and Rugg, T., "BASIC Timing Comparisons", Kilobaud, Issue #6, p. 66-69, June 1977.

8. Intel Corporation, 8008 and 8080 PL/M Programming Manual, 1975, 3065 Bowers Ave, Santa Clara, CA., 95051.

9. Intel Corporation, 8080 Simulator Software Package, 1974

10. Naval Postgraduate School Report NPS-53KD72 11A, ALGOL-E: An experimental Approach to the study of programming languages, by Gary A.Kildall, 7 January 1972

11. Naur, P. (ed.), "Report on the Algorithmic Language ALGOL 60", Comm. ACM, Vol. 3, No. 5, May 1960, pp.

299-314.

12. PEDROSO, L. R. B., ML80: A Structured Machine-Oriented Microcomputer Programming Language, Masters Thesis, Naval Postgraduate School, Monterey, CA, December 1975.

13. Strutynski, Kathryn B. Information on the CP/M Interface Simulator, internally distributed technical note.

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

15. van Wijngaarden, A., B. J. Mallioux, J. E. L. Peck, C. H. A. Koster, "Report on the Algorithmic Language ALGOL 68", Numer. Math. Vol.14, 1969.

16. Wirth, N., C. A. R. Hoare, "A Contribution to the Developments of ALGOL", CASM, Vol.9, No. 5, June 1966, pp. 413-432.

INITIAL DISTRIBUTION LIST

		No. Copies
1.	Defense Documentation Center Cameron Station Alexandria, Virginia 22314	2
2.	Library, Code 0212 Naval Postgraduate School Monterey, California 93940	2
3.	Department Chairman, Code 52 Department of Computer Science Naval Postgraduate School Monterey, California 93940	4
4.	Assoc Professor Gary A. Kildall, Code 52Kd Department of Computer Science Naval Postgraduate School Monterey, California 93940	1
5.	LCDR Stephen T. Hell, USN, Code 52Hl Department of Computer Science Naval Postgraduate School Monterey, California 93940	1
6.	Microcomputer Laboratory, Code 52ec Department of Computer Science Naval Postgraduate School Monterey, California 93940	2
7.	LCDR John P. Flynn, USN Fleet Air Reconnaissance Squadron One NAS Agana, Guam FPO San Francisco 96601	1
8.	LT Mark S. Moranville, USN 19109 Creekside Place Salinas, California 93908	1