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

John P. Flynn
Mark S. Moranville

Naval Postgraduate School
Monterey, California 93940

Approved for public release; distribution unlimited

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...
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.
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:

Ray A. Kibler
Thesis Advisor

Stephen J. Zelko
Second Reader

Chairman, Department of Computer Science

Dean of Information and Policy Sciences
ABSTRACT

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 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

A. HISTORY OF ALGOL

B. MICROCOMPUTER SOFTWARE

C. OBJECTIVES OF ALGOL-M

## II. ALGOL-M LANGUAGE DESCRIPTION

A. FEATURES OF THE ALGOL-M LANGUAGE
   1. Type Declarations
   2. Arithmetic Processing
   3. Control Structures
   4. Input/Output
   5. Disk Access

## III. IMPLEMENTATION

A. COMPILER IMPLEMENTATION
   1. Compiler Organization
   2. Scanner
   3. Symbol Table
   4. Parser
   5. Code Generation

B. INTERPRETER IMPLEMENTATION
   1. Building the ALGOL-M Pseudo Machine
   2. Overview of the Interpreter
   3. Allocation of Storage Space
      a. Integers
      b. Decimals
c. Strings.................................................. 31

d. Arrays.................................................. 34

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

5. Arithmetic Operations.............................. 37

b. String Operations................................. 38

7. Subroutines........................................... 39

a. Invocation............................................. 39

d. 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

6. 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. The purpose of the developers of ALGOL-60 was the establishment of a universal computer language specifically designed to allow for the formal and efficient program representation of algorithms. Additional versions and extensions of ALGOL-60 such as ALGOL-68 (115) and ALGOL-W (116) 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 \textsc{BASIC} compiler/interpreter (RASIC-F) \(^{(6)}\), and a \textsc{FORTRAN} compiler/interpreter (MICRO-COBOL) \(^{(2)}\). The majority of high level languages currently available for microcomputer based systems are extensions of the original Dartmouth \textsc{BASIC} and, although they allow for a reasonable level of programming sophistication, they are encumbered by the inherent limitations of the \textsc{BASIC} language constructs.

\section{Objective of \textsc{ALGOL-4}}

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. \textsc{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 \(^{(14)}\). \textsc{ALGOL-4} 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 \textsc{RUR} object code and which could be run on the \textsc{Vava} 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 \textsc{RUR} microprocessor and \textsc{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 \(-10,384\) and \(+10,384\). 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 microprocessor 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 ELSF, WHILE, CASE and GOTO constructs. Function and procedure calls are also used as control structures. ALGOL-M is a block structured 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 specified 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 identifier 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
ACCIUIM (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[141].

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

SYMBOL TABLE

HASH ARRAY

28 0 0 18

28 0 0 18

A B C

COLLISION FIELD

TWO ENTRIES WITH SAME HASH VALUES

FIGURE 1
The address field indicates the identifier's position in the PPI 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 FULLSLOOKUP which searches the current block and all outer blocks for an identifier. The second is NORMALSLOOKUP which simply checks the current block level. In most cases, FULLSLOOKUP is used to determine if an identifier being used has been declared and NORMALSLOOKUP is used to determine if an identifier being declared has been previously declared in the same block level.
4. Parser

The LALR parser is a derivative of the I-bar parsing technique. It receives tokens from the scanner and analyzes them to determine if they are part of the AlOL-LR 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.

4. 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 beginning 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 3 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 $\text{PA}$, while the next-to-top item is addressed by $\text{PB}$. The contents of the two bytes on top of the stack are referenced by the variable $\text{AHA}$ while the two byte contents of the next-to-top stack position are referenced by the variable $\text{ARB}$. 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" "RRC"

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

25
contents of the top and next-to-top stack locations are referenced by the variables PRA and RRA 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 $-10,324$ 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 [81]. 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
differentiation 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 runtime 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 runtime. 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

SIGN BIT

| 0 0 0 0 0 0 1 1 | 0 0 0 0 0 0 0 0 | +3 |

LOW ORDER BYTE  HIGH ORDER BYTE

SIGN BIT

| 0 0 0 0 0 0 1 0 | 1 0 0 0 0 0 0 0 | -2 |

INTEGER STORAGE

FIGURE 5

29
BEGIN
DECIMAL(12) X;
X := 123456.78;
END

<table>
<thead>
<tr>
<th>OFFSET TO BOTTOM OF DECIMAL STORAGE</th>
<th>SIGN (+)</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>1</td>
</tr>
<tr>
<td>56</td>
<td>78</td>
</tr>
<tr>
<td>12</td>
<td>34</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
</tr>
</tbody>
</table>

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 level 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 permitted 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 LEVEL 0 \textbf{BEGIN} \\
\textbf{BEGIN} \text{DECIMAL} A,B,C; \\
\text{STRING} X,Y,Z; \\
\textbf{BEGIN} \text{DECIMAL} C,D,E; \\
\text{END} \text{BLOCK LEVEL 2} \rightarrow \text{END}; \\
\text{END} \text{BLOCK LEVEL 1} \rightarrow \text{END}; \\
\text{END} \text{BLOCK LEVEL 0} \rightarrow \text{END}; \\

\begin{figure}
\centering
\includegraphics[width=0.7\textwidth]{figure7.png}
\caption{Block Levels and Stack Storage}
\end{figure}

32
BEGIN
STRING(9) TESTWORD;
TESTWORD := "HELLO";
END

<table>
<thead>
<tr>
<th>OFFSET TO BOTTOM OF STRING STORAGE</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;L&quot;</td>
<td>&quot;O&quot;</td>
</tr>
<tr>
<td>&quot;E&quot;</td>
<td>&quot;L&quot;</td>
</tr>
</tbody>
</table>
| ACTUAL NUMBER OF CHARACTERS STORED | 5  | "H"

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 \( (R) \), 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
yte 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-H: 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

BEFORE

 AFTER

<TOP OF MEMORY

<RETURN ADDR

<ADDR OF SUBR IN CODE AREA

<1st PARM TYPE

<PARAMETER

<2nd PARM TYPE

<PARAMETER

<1st PARM TYPE

<PARAMETER

<2nd PARM TYPE

<PARAMETER

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-E10. 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 reallocation 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 PCR 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

STACK BEORE

PRT

LOCAL VARIABLES

FORMAL

PARAMETER

POINTERS

<ALLOCATION

<LENGTH (FOR FUNC)

<SAVED BLOCK

<POINTER

<PTR FOR RETURN

VALUE

AFTER

TOP OF MEMORY

RETURN ADDRESS

ACTUAL PARM COUNT

1st PARM TYPE

PARAMETER

CALL INFORMATION (MOVED OFF

STACK) AREA USED TO SAVE

PCB FOR RECURSIVE CALLS

FIGURE 11

42
STACK CONFIGURATION FOR SV2 OPERATOR

FIGURE 12

STACK

<TOP OF MEMORY
<RETURN ADDR
<PARM COUNT
<1st PARM TYPE
<PARAMETER
<2nd PARM TYPE
<PARAMETER

THIS DATA IS COPIED INTO THE AlLOCATED AREA

ALLOCATED AREA

PRT

PCB
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].
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 APA and APB respectively. The contents of the low-order byte of AR and ARB is referenced by the variable BPA 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 LITLOC 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.

ATD: (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

ADD: (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.

SRD: (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 SRI 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.

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

CT2: (convert integer). The same action is taken as with CTI 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.

GEU: (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.

DGEO: (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.

SFGL: (string equal to). The same action is taken as in the NEQL 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

ACH: (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 Pb 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 clock level table is decreased by the value stored in the next two bytes in the code area. The stack pointer (RA) is moved below RA to the top of the second item on the stack.

BLI: (clock 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.

DLI: (clock 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

RAW: (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.

**n. 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.

**bPC**: (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 PRI address which is the top item on the stack. The PRI 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.

**SDU:** (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 SSD operator except the pointer to the
string allocated area and the string value are removed from
the stack.

j. Input/Output Operators

**DMP:** (dum). 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.

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

**ASC:** (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.

WSU: (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.

RFI: (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.

UPN: (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.

ENW: (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

PRU: (subroutine call). The two bytes of code following the PRU 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 PCR 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: (unsafe 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 REFLI and have been run with several versions of the BASIC programming language. The results (expressed as execution time in seconds) are summarized as follows:

<table>
<thead>
<tr>
<th>BENCHMARK NUMBER</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALGOL-M</td>
<td>1.8</td>
<td>1.3</td>
<td>3.5</td>
<td>7.3</td>
<td>9.1</td>
<td>14.2</td>
<td>16.4</td>
</tr>
<tr>
<td>INTEGER BASIC</td>
<td>1.3</td>
<td>3.1</td>
<td>7.2</td>
<td>7.2</td>
<td>8.8</td>
<td>18.5</td>
<td>28.0</td>
</tr>
<tr>
<td>STANDARD BASIC</td>
<td>1.7</td>
<td>7.5</td>
<td>20.9</td>
<td>20.9</td>
<td>22.1</td>
<td>36.2</td>
<td>51.8</td>
</tr>
</tbody>
</table>

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
600 A:=K/K*K+K-K
610 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

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

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

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

ERROR MESSAGES

AR 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.
D7 Division by zero, result set to 1.0.
EF Disk end of file, no action specified.
IO Integer overflow.
NI No TNT file found or directory.
UV 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.
SM Characters lost during string store.
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 ALCOL-M have an implied precedence which is used to determine the manner in which operators and operands are grouped. A-R/C causes the result of R divided by C to be subtracted from A. In this case R 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:
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:**

$$ (Y + Y) \times (Z + Y + X) \times ? $$

$$ x + y + z \times x \times y \times z / 5.456 + 1 $$
ARRAY declaration

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 19 digits. Integer lengths are not specified since storage adequate to represent all integer values between -10,384 and +10,384 is automatically allocated. Arrays are not automatically initialized to zero.

EXAMPLE:

INTEGER ARRAY x[0:5,0:5];
DECIMAL(10) ARRAY x,y[0:5,5:10];
STRING ARRAY WORDS[y+3:12];
assignment statement

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:

<table>
<thead>
<tr>
<th>expression</th>
<th>integer</th>
<th>decimal</th>
<th>string</th>
</tr>
</thead>
<tbody>
<tr>
<td>integer</td>
<td>yes</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>variable</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>decimal</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>string</td>
<td>no</td>
<td>no</td>
<td>yes</td>
</tr>
</tbody>
</table>

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

EXAMPLE:

x := y + z;

ELEMENr:
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 < R THEN A := 1 ELSE A := 2;

IF R = C THEN
BEGIN
   WRITE(R);
   R := R + 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-W 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 anywhere a simple statement can be used.

EXAMPLE:

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

IF X = Y THEN
  BEGIN
    Y := 5;
    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.
boolean expression

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:
\[x > y \text{ or } y < z;\]
\[(x = y) \text{ and } (y = 7 \text{ or } z = 10);\]
\[\text{if not } x = 1 \text{ then write("HELLO");}\]
bound pair list

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)
(3:6,0:v)
(v*3:z,1:12)
CASE statement

ELEMENT:

CASE statement

FORMAT:

CASE expression OF

BEGIN

statement; ...

END;

DESCRIPTION:

The CASE statement allows the programmer to choose 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;
CLOSE statement

ELEMENT:

CLOSE statement

FORMAT:

CLOSE identifier (identifier) ...

DESCRIPTION:

This statement allows the programmer to explicitly close the file specified. 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;
constant

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 -15,364 to +10,384. 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 characters 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.
FILE declaration

EXAMPLE:

FILE TAPE1, TAPE2(128);
FOR statement

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 + R;
    WRITE(A);
    END;
FUNCTIONS

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 := SUM(5.0);
c := FUNC * RND(2);
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:**

```plaintext
INTEGER FUNCTION VALUF(Y);
```
INTEGER X;
BEGIN
X := (X * 5) + (Y * 2);
VALUE := X;
END;
GOTO statement

ELEMENT:

GOTO statement

FORMAT:

label definition GOTO 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
IF statement

ELEMENT:

IF statement

DESCRIPTION:

See balanced statement or unbalanced statement.
Label definitions are optional on all balanced or unbalanced statements.
procedure call

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);
procedure declaration

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;
READ statement

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 action must be specified and the input device is the disk. A READ statement reads one or more variables at a time. The optional ONENDFILE section indicates action to be taken when the end of the specified file is reached.

EXAMPLE:

READ(WURDONE, X, VALUE2);
READ FILE3 (WURDONE, X, VALUE2);

PROGRAMMING NOTE:

The ONENDFILE section is currently not implemented.
reserved word list

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

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

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.
simple declaration

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 10 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.
TAB expression

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]);
unbalanced statement

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]
WHILE statement

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 <> A DO
    BEGIN
        X := Y / 3;
        WRITE(X);
    END;

END;
WRITE statement

ELEMENT:

WRITE statement

FORMAT:

WRITE:WRITED (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 WRITEDE option will continue printing on the same line. If the form of the statement is WRITE() or WRITEDE(), 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", IAD s, 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>; x?T; (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
   **ALT

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

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

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

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

10
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>

22  <declaration type> ::= string
    *TMI (string default size);

23  ; string <size option>
    *<size option>

24  ; integer
    N/A

25  ; decimal
    *TMI (decimal default size);
26  *<size option>  
   : decimal <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  
   <balanced statement>  
   *<if clause>; <true part>; <balanced statement>

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

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

35  : !<if clause> <true part>  
36  else <unbalanced statement>  
   *<if clause>; <true part>; <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>
50 *<identifier>*
   ; <identifier>
51 *<assignment statement>* ::= <left part> <expression>
   *<left part>*; <expression>; ([LII/LII])
52   ; <left part>
52 *<assignment statement>*
   *<left part>*; <assignment statement>
53 *<left part>* ::= <variable> ::= 
   *<variable>*; ([LII/LIILOAD])
54 *<expression>* ::= <arithmetic expression>
   *<arithmetic expression>*
55   ; <if expression> <expression>
   N/A
56 *<arithmetic expression>* ::= <term>
   *<term>*
57   ; <arithmetic expression> +
57  \texttt{\textless term\textgreater ; \textless arithmetic expression\textgreater ; \textless term\textgreater ; [ADT/ADD]}

58  \texttt{\textless arithmetic expression\textgreater - \textless term\textgreater ; \textless arithmetic expression\textgreater ; \textless term\textgreater ; [SBI/SBN]}

59  \texttt{\textless arithmetic expression\textgreater ! \textless term\textgreater ; \textless arithmetic expression\textgreater ; \textless term\textgreater ; [CAT]}

60  \texttt{\textless term\textgreater ; \texttt{NEG} \textless term\textgreater ; \texttt{+ term\textgreater ;}}

61  \texttt{\textless term\textgreater ; \texttt{:= primary\textgreater ;}}

62  \texttt{\textless primary\textgreater ; \texttt{\textless primary\textgreater ; \textless primary\textgreater ; \texttt{mul primary\textgreater ; \texttt{div primary\textgreater ;}}}

63  \texttt{\textless term\textgreater ; \texttt{\textless primary\textgreater ; \texttt{\textless primary\textgreater ; \texttt{mul primary\textgreater ; \texttt{div primary\textgreater ;}}}

64  \texttt{\textless term\textgreater ; \texttt{\textless primary\textgreater ; \texttt{\textless primary\textgreater ; \texttt{mul primary\textgreater ; \texttt{div primary\textgreater ;}}}

65  \texttt{\textless primary\textgreater ; \texttt{\textless primary\textgreater ; \texttt{\textless primary\textgreater ; \texttt{mul primary\textgreater ; \texttt{div primary\textgreater ;}}}

66  \texttt{\textless primary\textgreater ; \texttt{\textless primary\textgreater ; \texttt{\textless primary\textgreater ; \texttt{mul primary\textgreater ; \texttt{div primary\textgreater ;}}}

67  \texttt{\textless primary\textgreater ; \texttt{:= variable\textgreater ; \texttt{LITLOAD \{address of simple variable\}/ N/A for \texttt{subscripted variables/ IM2 (subroutine \texttt{address in code area); PRU\}

68  \texttt{\textless constant\textgreater ;}}

69  \texttt{\textless procedure call\textgreater ;}

70  \texttt{\textless assignment statement\textgreater ;}

71  \texttt{\textless expression\textgreater ;}

72  \texttt{\textless constant\textgreater ; \texttt{:= integer\textgreater ; \texttt{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  \texttt{<array list>} ::= \texttt{<array head> <identifier>} \\
            \hspace{1em} \times \texttt{<array head>}

88  \texttt{<array head>} ::= \texttt{<declaration type> array} \\
            \hspace{1em} \times \texttt{<declaration type>}

89          \texttt{; \texttt{<array head> <identifier>},} \\
            \hspace{1em} \times \texttt{<array head>}

90  \texttt{<bound pair list>} ::= \texttt{<bound pair head>} \\
            \hspace{1em} \times \texttt{<bound pair> } \\
            \hspace{1em} \times \texttt{<bound pair head>; <bound pair>}

91  \texttt{<bound pair head>} ::= \{ \\
            \hspace{1em} \times \texttt{N/A}

92          \texttt{; \texttt{<bound pair head> <bound pair>},} \\
            \hspace{1em} \times \texttt{<bound pair head>}

93  \texttt{<bound pair>} ::= \texttt{<expression> : <expression>} \\
            \hspace{1em} \times \texttt{<expression> ; <expression>}

94  \texttt{<subscripted variable>} ::= \texttt{<subscript head>} \\
            \hspace{1em} \times \texttt{<subscript head> ; <expression>;} \\
            \hspace{1em} \times \texttt{LITLOAD <array address>; SUB}

95  \texttt{<subscript head>} ::= \texttt{<identifier> \{} \\
            \hspace{1em} \times \texttt{N/A}

96          \texttt{; \texttt{<subscript head> <expression>},} \\
            \hspace{1em} \times \texttt{<subscript head> ; <expression>}

97  \texttt{<go to statement>} ::= \texttt{<go to> <identifier>} \\
            \hspace{1em} \times \texttt{DCB (\# of blocks to decrement)/NOP;NOP} \\
            \hspace{1em} \times \texttt{followed by RHS (branch address)}

98          \texttt{; \texttt{<go to> <integer>}} \\
            \hspace{1em} \times \texttt{DCB (\# of blocks to decrement)/NOP;NOP} \\
            \hspace{1em} \times \texttt{followed by RHS (branch address)}

99  \texttt{<go to>} ::= \texttt{go to} \\
            \hspace{1em} \times \texttt{N/A}

100 \texttt{; goto} \\
            \hspace{1em} \times \texttt{N/A}

101 \texttt{<read statement>} ::= \texttt{<read head> <variable> )} \\
            \hspace{1em} \times \texttt{<read head>; LIT/LITLOAD (address of variable)}; \\
            \hspace{1em} \times \texttt{PDT/PCT; STD/SDD/SSD} \\
            \hspace{1em} \times \texttt{if console I/O then FCP}
102  \text{<read head> ::= read (}
   \text{*RCN}
\text{)}
103  \text{! read file option (}
   \text{*(not implemented)}
\text{)}
104  \text{! <read head> <variable>,}
   \text{<read head>;} \text{ same as production 101}
105  \text{<write statement> ::= <write head> <expression> )}
   \text{<write head>; <expression>; [WIC/WDC/WS/WID}
   \text{*WUN/WID]}
106  \text{! <write head> <tab expression> )}
   \text{*(not implemented)}
107  \text{! <write head> <pic definition> )}
   \text{*(not implemented)}
108  \text{<write head> ::= write (}
   \text{*DMP}
\text{)}
109  \text{! write file option (}
   \text{*(not implemented)}
\text{)}
110  \text{! writeon (}
   \text{N/A}
\text{)}
111  \text{! writeon file option (}
   \text{*(not implemented)}
\text{)}
112  \text{! <write head> <expression>,}
   \text{<write head>; <expression>; [LIT/LITLOAD]}
113  \text{! <write head> <tab expression>,}
   \text{*(not implemented)}
114  \text{! <write head> <pic definition>,}
   \text{*(not implemented)}
115  \text{file option ::= <identifier>}
   \text{N/A}
116  \text{! <identifier> <rec option>}
   \text{N/A}
117  \text{! <string>}
   \text{N/A}
118  \text{! <string> <rec option>}
   \text{N/A}
119  \text{<rec option> ::= , <identifier>}
   \text{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> ; RSC (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> ; RUR

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 \(<relation> ::= \sim\)
N/A

138 \(\sim <\)
N/A

139 \(\sim >\)
N/A

140 \(\sim <\comp>\)
N/A

141 \(<\comp> ::= < >\)
N/A

142 \(\sim < =\)
N/A

143 \(\sim > =\)
N/A

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

145 \(<\text{while clause}> ::= <\text{while} <\text{boolean expression}>\)
*<boolean expression>; RSC {branch address}

146 \(<\text{while}> ::= \text{while}\)
N/A

147 \(<\text{for statement}> ::= <\text{for clause}> <\text{step expression}>\)
<for clause>; <step expression>; <until clause>; <do statement>; RRS {branch address}

148 \(<\text{for clause}> ::= \text{for} <\text{assignment statement}>\)
*<assignment statement>; RRS (branch address); RRS (branch address); LIT (counter variable); LITLOAD (counter variable)

149 \(<\text{step expression}> ::= \text{step} <\text{expression}>\)
*<expression>; AOT; STD; LITLOAD (counter variable)

150 \(<\text{until clause}> ::= <\text{until non-term} <\text{expression}>\)
*<until non-term>; <expression>; LEO; BSC
*(branch address)

151 \(<\text{until non-term}> ::= \text{until}\)
*{1,1} {one}; AOT; STD; LITLOAD (counter variable)
N/A if there is a step expression

152 \(<\text{do statement}> ::= \text{do} <\text{simple statement}>\)
*<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> ; SV2\]
160 <procedure heading> ::= <paramless proc>\\
*<paramless proc>
161 \( <proc & params> ; SV2\]
162 <paramless function> ::= <declaration type>\\
162 function <identifier> ;
163 <function & params> ::= <function head>\\
163 <identifier> ) ;
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 PCR) ; SAV ; BLI\\
*if this is a non-integer function then\\
*LITLOAD (length of return value) ;\\
*LIT (PRT address of function) ; [ALD/ALS]
<proc & params> ::= <procedure head> <identifier> ) ;
  *<procedure head>; same as production 167

<procedure call> ::= <call heading> <expression> )
  *<call heading>; <expression>; IM2
  *(address of subroutine in code area); RNO

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

<call heading> ::= <identifier> )
  *<expression>

< declaration> ::= <declaration type> external

<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))

<case heading> ::= case <expression> of
  *<expression>; LIT {3}; WP1; IM2
  *(address of end of case block); S6R; BFA

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

<case block body> ::= <statement>
  *<statement>; BRS (address of end of case block)

<case block body> ::= <statement>
  *same as production 182
100h: /*load point for compiler*/

DECLARE false literally '0',
true literally '1',
lit literally '"',
startbdos address initial(5h),
max based startbdos address,
boot lit '0',
stacksize lit '48',
intsize lit '128',
dcl lit 'declare',
proc lit 'procedure',
fileof lit '1',
file lit '20',
identitize lit '32',
addr lit 'address',
forever lit 'while true',
varsize lit '100',
indexsize lit 'address',
statebitize lit 'address',
maxoncount lit '23',
err lit '3',
if lit '0',
stringdelim lit '22h',
questionmark lit '3fh',
tab lit '09h',
colin lit '3ah',
comment lit '9',
conbuffsize lit '82',
eolchar lit '0dh',
hashtblsize lit '64',
sourcecssize lit '128',
hashmask lit '63',
conchar lit '5ch',
eoffiller lit '1fh',
percent lit '25h',

declare maxrno literally '132',
maxlno literally '190',
maxpush count /
maxstate count /
start state /
number of productions /
semicolon /
colon /
string /
decimal /
integer /
procedure /
identifiers /
terminals /

DECLARE block address initial(5h),
sourcebuff based block sourcecssize) byte,
sourceptr byte initial(sourcecssize),
buffer byte initial(55),
errorcount address initial 0,
linebuff conbuffsize) byte,
lineptr byte initial(0).
When the following global variables are used by the scanner

```c
declare token byte; /* type of token just scanned */
hashcode byte; /* has value of current token */
nextchar byte; /* current character from geteliar */
accurimize byte; /* indicates accum was full, still more */
```
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,
    (a based a, d based b, 1) byte;
    do while (l := l - 1) <> 255;
        d := b := b + 1;
        a := a + 1;
    end;
end move;

fill: procedure (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 monl (10, toggle);
end read;

printchar: procedure (char);
    declare char byte;
    call monl (23, char);
end printchar;

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

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

open@sourcefile: procedure;
    call move (. 'alg'. rfchaddr + 16);
    rfch(32) := @;
    if mon2 (15, rfchaddr) = 253 then do;
        call print (. 'ns $' );
        go to boot;
    end;
end open@sourcefile;

close@sourcefile: procedure;
    /* close a file */
    if mon2 (16, rfch) = 253 then
call diskerr;
end close@int@file;

setup@int@file: procedure;
/* set up int file as new file */
if mothfile then /* only make file if this toggle is off */
return;
call move(.rfcb,.wfcb,9);
wfcb(32)=0;
call coml(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 reopen */
return;
end rewind@source@file;

read@source@file: proc byte;
declare cnt byte;
if(dnt:= mon2(rfile.rfcbaddr)) > fileeof then
call diskerr;
return cnt;
end read@source@file;

write@int@file: procedure;
if mothfile then
return;
call monl(26,.diskoutbuff);
if mon2(21,.wfcb) <> 0 then
call diskerr;
call nmonl(26,.0h);
/* reset dma address */
end write@int@file;

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

printdec: procedure(value);
declare value address, l byte, count byte;
declare decl(4) address initial(1000,100,10,1);
declare flag byte;
flag = false;
count = 30h;
do i = 0 to 3;
do while value >= decl(i):
value = value - decl(i);
flag = true;
count = count + 1;
end;
if flag or (i = 3) then
call printchar(count); else
call printchar(' '); end;
call printchar(' ');
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 writeintfile:
      buffptr:=0;
      end;
  diskoutbuff(buffptr)=objcde;
  end emit;
end clear@line@buff:

listline: procedure (length);
  declare (length+1) byte;
  call printdec(linen0);
  call printdec(prevGtnex~l);
  call printchar(' ');
  do 1 = 0 to length;
      call printchar(linebuff+1));
  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),
  errset byte initial(false),
  compiling byte,
  codesize address, /* used to count size of code area */
  prtek address initial(0xffeh), /* used to count size of prtek */
  /x variables used during for loop code generation */
  forcount byte initial(0),
  randomfile byte,
  fileio byte initial(false);

getchar: procedure byte;
  declare addof data ('eof', enolchar, 1); /* add to end if left off */
  nextsourcechar: procedure byte;
    return sourcebuff(sourceptr);
end nextsourcechar;

checkfile: procedure byte;
  do forever;
    if (sourceptr<sourceptr+1) > sourcebuffsize then
      do:
        sourceptr:=0;
        if (readsourcefile=fileof then
          return true;
      end;
      if (nextchar>nextsourcechar) < if then
        return false;
  end;
end checkfile;

if checkfile or (nextchar = nofillchar) then
  do: /* eof reached */
    call move(.addof, chloc, 5):
sourceptr = 0;
nextchar = next$source$char;
end:
linebuff(lineptr) = lineptr + 2; nextchar = nextchar;
if nextchar = eolchar then
do:
line = line + 1;
if listsource then
call list(line(lineptr - 1));
lineptr = 0;
call clear(linebuff);
end;
if nextchar = tab then
nextchar = ' ';
return nextchar;
end getchar:

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

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

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

error: procedure(errorcode);
declare errorcode address.
  1 byte;
errorcount = errorcount + 1;
call print (. 'err $');
call printdec(errorcount);
call print (. 'error $');
call printchar('');
call printchar(high(errorcode));
call printchar(low(errorcode));
call crlf;
call printprod;
if token = eofc then
do:
call print$error$;
call mon3;
end:
end error:

initialize$scanner$: procedure:
declare count byte;
call open(sourcefile);
line, lineptr = 0;
call clear(linebuff);
sourceptr = 128;
call getnoblank;
do while nextchar = ' ';
call get$noblank$;
if count := (nextchar and $3fh) = 'a' (< 4 then
do case count;
end;
if pass then listsouce = true;
listprod $true$;
noinput $true$;
listtoken $true$;
debugin $true$;
end;
end of case x.
call get$noblank$;

126
end:
end initialize scanner:

scannerr: 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 getnolobank:
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 lowercase;
decimalpt: proc byte:
return nextchar = '1';
end decimalpt;
convtouppepr: proc:
if lowercase and lowercase then
nextchar = nextchar and 5fh;
end convtouppepr:
letter: procedure byte:
call convtouppepr;
return ((nextchar = 'a') <= 25) or lowercase;
end letter:
alpha: procedure byte:
return numeric or letter or decimalpt;
end alpha:
spoolnumeric: procedure:
do while numeric:
call putandchar:
end:
end spoolnumeric:

setupnextcall: procedure:
if nextchar = ' ' then a
getnolobank:
cont = false;
end setupnextcall:

lookup: procedure byte:
declare maxring lit '9';
declare vocab data(0., '<', '(', '+', '5bd.7ch. 'x' ')', ':', '-', '/', ' ', '
', ',', 'do', 'go', 'if', 'of', 'or', 'to', 'eof', 'end', '
', 'end', 'for', 'not', 'pic', 'tab', 'case', 'else', 'file', 'goto', 'end', '
', '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(0.1,16.24.31.38.45.53.54);
declare count data(0.14.7,6.5,0.4,1.9);
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 > maxwring 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=:
        return true:
      end:
      ptr=ptr+ accum:
    end:
  return false;
end lookup:

/*
 scanner main code */
/*
 do forever:
  accum. hashcode, token = 0;
do while nextchar=sochchar:
  call getmoblank:
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 putinaccum:
  end: /* of do forever */
end: /* of recognizing a string */
else if numeric or decimalpt then
  do: /* have digit */
    token = integer:
  do while nextchar='0': /*elim leading zeros*/
    nextchar=:
  end:
  call spoolnumeric;
  if decimalpt then

do:  
  token=decimal;  
  call putandchar;  
  call spoolnumeric;  
  end;  
  if accum@ then  
    hashcode, accum(accum : x) = '0';  
    call setup@next@call;  
    return;  
  end;  
else if letter then  
  do:  
    /% have a letter /%  
    do while alphanumeric:  
      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 setup@next@call;  
        return;  
      end;  
  end;  
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 = 'w' then  
              call putandget;  
            end;  
          end:  
        else if nextchar = 'x' then  
          do:  
            call putandchar;  
            if nextchar = 'x' then  
              call putandget;  
            end;  
          end:  
        else call putandget;  
        end:  
      end;  
    else call putandget;  
    end;  
  end;  
else if nextchar = 'w' then  
  do:  
    call putandchar;  
    if nextchar = 'w' then  
      call putandget;  
    end;  
else call putandget;  
  end;  
else call putandget;  
  if not lookup then  
    call error('w');  
    call setup@next@call;  
    return;  
end;  
end:  
else  
  do:  
    /% of recognizing special char /%  
  end;  
else  
  do forever  
end scanner;  
end of scanner.
/* procedures for synthesizer*/
initialize@sym tbl:proc:
   if pass 1 then
do:
      /* fill hashtable with 0's */
      call fill hashtable(0,shbl(hash tbl size,2));
      shbl = .memory;
   end:
/* initialize pointer to top of symbol table*/
shbl top = max - 2;
end initialize@sym tbl:

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

setblk@level:proc(level):
   declare level byte:
   call setaddrptr(6);
   byteptr = level;
end setblk@level:

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

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

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

hash tbl@off sym hash:proc address:
   return hashtable(sym hash):
end hash tbl@off sym hash:

limits:proc(count):
   /* check to see if additional shbl will overflow limits of
      memory. if so then punt else return */
   declare count byte: /*size being added is count */
   if shbltop <= (shbl + 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):
   addptr = loc:
end setaddr:

lookup@current @shbl:proc(chk@shbl byte:
   declare chk@shbl byte:
      len byte:
      a based printname byte:
bases $hmvhtb$

I do while base <> 0:
    call setaddrptr(6):
    if bytепtr < chkSblk then
        return false:
    if bytепtr = chkSblk then
        do:
            if (len:ptr) = n then
                do while (ptr(len + 2) = m(len)):
                    if (len + len - 1) = 0 then
                        return true:
                    end:
            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:
        else:
            return false:
        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 l byte:
    l based printname byte:
    call listadd(l:=n+8):
    base = sbtbl: /*base for new entry */
    call move(printname + l,sbtl + 3,(ptr := n)):
    call setaddrptr(3): /*set resolve bit to 0x*/
    bytепtr = 0:
    call setlink:
    addrptr = hash$tbl$off$symhash:
    hash$tbl$off$symhash = base:
    call set$lk$level(blk$level): s
    sbtl = sbtl + 1:
end enter:

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

gettype: proc byte: /*returns type of variable*/
    call setaddrptr(3):
    return bytепtr:
end gettype:

setsubtype: proc(stype): /*enter the subtype in $sbl$*/
    declare s type byte:
    call setaddrptr(7):
    bytепtr:=stype:
end setsubtype:

getparm: proc byte:
    call setaddrptr(10):
    return bytепtr:
end getparm:

getsubtype: proc byte: /*return the subtype*/
    call setaddrptr(7):
return byteptr;
end getsubtype;
settype:proc (type); /* set typefield = type */
declare type byte;
call setaddrptr (3);
byteptr = type;
end settype;

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', dbh lit '5', dmp lit '6', xit lit '7'.
add lit '8', als lit '9', alit '10', xim lit '11'.
sub lit '12', subd lit '13', subb lit '14', subb lit '15'.
mp lit '16', mpd lit '17', dvi lit '18', dvb lit '19'.
dneg lit '20', neg lit '21', c1 lit '22', c2 lit '23', c3 lit '24'.
dcl lit '25', pop lit '26', call lit '27', call lit '28'.

brc lit '34', bar lit '35', lss lit '36', diss lit '37'.
also lit '38', str lit '39', dgr lit '40', segr lit '41'.
eql lit '42', deq lit '43', seq lit '44', seq lit '45'.
dg lit '46', sgleq lit '47', geq lit '48', dgeq lit '49'.
sgeq lit '50', leg lit '51', sleq lit '52', seql lit '53'.
inot lit '54', dnot lit '55', snot lit '56', smdl lit '57'.
dand lit '58', sand lit '59', lor lit '60', dor lit '61'.
sor lit '62', wic lit '63', wic lit '64', wsc lit '65'.
wid lit '66', wdd lit '67', wdd lit '68', smr lit '69'.
bra lit '70', row lit '71', sub lit '72', rcl lit '73'.
rcd lit '74', ren lit '75', rdi lit '76', rdd lit '77'.
rds lit '78', renlit '79', erc lit '80', slt lit '81'.
sdi lit '82', smd lit '83', sid lit '84', smd lit '85'.
add lit '86', opn lit '87', csq lit '88', rdb lit '89'.
rdf lit '90', edr lit '91', edw lit '92', pro lit '93'.
mwv lit '94', sw2 lit '95', uns lit '96', rtn lit '97'.

declare state size;
statesize, statestack(2) statesize;
hash(2) statesize byte;
symbol(2) statesize address;
error(2) statesize address;
vartype(2) statesize byte;
type(2) statesize byte;
state(2) statesize byte;
var(2) statesize byte;
varindex byte;
(sp, mp, appl, molook) byte;
onstack(maxoncount) byte;
class byte initial(2);
class byte initial(23);
class byte;
strsiz byte initial(10);
decsize byte initial(10);
a byte;
num byte initial(10);
saveparm address;
saveparm based saveparm byte;
fpcount byte;
probsame address;
proc@type byte;
(ptest, l) byte;
proc byte initial(10);
type temp byte;
flag byte;
lpcount byte;
ident address;
initialize: procedure:
codeword, onstack, cable = 0:
prev$index = 255:
blks$cut = 0:
blks$level[z0]
end initialize: synthesize:

synthesize: proc:

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

declare s$ivar lit '0hh'.
s$ivar lit '99'.
pro lit '93'.
ex$nproc lit '03'.
b$nfunc lit '06'.
const lit '06'.
lab lit '09'.
integer lit '08'.
str lit '1'.
file lit '0ch'.
func lit '0dh'.
p$arm lit '10h'.

declare (typ$sp, typ$emp, typ$emp1) byte.
(b, temp) byte.
(stype$sp, stype$mp, stype$mp1) byte.
(hash$mp, hash$mp, hash$mp1) byte.
(s$iloc$sp, s$iloc$mp, s$iloc$mp1) address.
(s$iloc$sp, s$iloc$sp) address:

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

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

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

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

copy: procedure:
type$sp = type$sp;
typ$emp1 = type$mp1;
typ$emp = type$mp;
stype$sp = stype$sp;
stype$emp1 = stype$mp1;
stype$emp = stype$mp;
symloc$sp = symloc$sp;
symloc$mp1 = symloc$mp1;
symloc$mp = symloc$mp;
hash$mp = hash$mp;
hash$mp1 = hash$mp1;
hash$sp = hash$sp;
s$iloc$sp = s$iloc$sp;
s$iloc$mp = s$iloc$mp;
end copy:

setsymloc$sp: procedure(a):
declare s address;
symloc$sp = a;
end setsymloc$sp:

setsymloc$mp: procedure(a):
declare s address;
symloc$mp = a;
end setsymloc$mp:

settype$sp: procedure(b):
declare b byte;
type$sp = b;
end settype$sp:
settypeprop: procedure(b);
declare b byte;
type(sp) = b;
end settypeprop;

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

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

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

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

setrlocomp: procedure(a);
declare a address;
setloc(sp) = a;
end setrlocomp;

setrlocomp: proc(a);
declare a byte;
setloc(mp) = a;
end setrlocomp;

getrloc: proc byte;
call = taddrptr(8);
return addptr;
end getrloc;

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;

genintiv: proc(a):
declare a byte;
call generate(int1);
call generate(a);
end genintiv;

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(bll);
end decr@blk$level;

calc@var: procedure(b) address;
declare b byte;
return var(b) + .varc;
end calc@var;
setlookup: procedure(a);  
declare a byte;
printname = value(a);
symhash = label(a);
end setlookup;

lookuponly: procedure(a) byte;
declare a byte;
call setlookup(a);
if lookup&current(b1k(b1k#level)) then
  return true;
base&flag := true;
return false;
end lookuponly:

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

normallookup: procedure(a) byte;
declare a byte;
if lookuponly(a) then
  return true;
call enter;
return false;
end normallookup:

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

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

literals: proc(a);
declare a address;
call genexpr or $0000h;
end literals:

setname: proc;
printname := aliasing;
symhash := table and hashmask;
end setname:

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

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

compiler$label: proc:
call set$compiler$label:
call gentwo(getaddr):
end compiler$label:

set$center: proc(a):
  declare (a,b) byte:
  call settypep(clable):
call compiler$label:
call gentwo(getaddr):
call compiler$label:
end set$center:

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

process$case$label: proc(a):
  declare (a,b,x) byte:
  b=clable:
  clable=a:
  call generate(brs):
call setname:
if pass$2 then
  x=lookup:
call gentwo(getaddr):
call compiler$label:
end process$case$label:

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

lit$load: proc(a):
  declare a address:
  call gentwo(a or 0c00h):
end lit$load:

step$gen: proc:
  call generate(adj):
  call generate(add):
  call compiler$label:
call lit$load(adjident):
end step$gen:

chk$typ: proc(a): /* check mpop to see if they are both decimal */
declare a byte: /* both integer, one of each, or neither */
if (stypemp=Int) and (styps$int) then
call generate(a):
else
  if (stypemp=dec) and (styps$dec) then
    call generate(a):
    call generate(adj):
  else
    if (stypemp=dec) and (styps$int) then
do:
  else
    if (stypemp=Int) and (styps$int) then
do:
  else
    if typemp fun or pass$2 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;

chktyp4: proc byte;
if (stypesp<>Int) then
  return true;
call error('mn');
return false;
end chktyp4;

chktyp5: proc byte;
if (stypesp<>Int) then
  return true;
call error('mf');
return false;
end chktyp5;

chktyp6: proc;
if pass2 then
  do;
    if (stypesp=Int) and (stypesp=Dec) then
      call error('id');
    else if (stypesp<>Int) then
      return false;
    end;
  end;
end: end chktyp6;

gencst: proc(subtype);
declare (i, subtype) byte;
gen@accum: proc;
if pass2 then
  do i=1 to accum;
    call emit(accump(i));
  end;
end gen@accum;
call generate(subtype);
call setstypemp(const);
call setstypemp(subtype);
if subtype=Int then
  if accum<>0 then
    call error('lo');
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(accump(i));
    end;
  else
do;
    call gen@accum;
codesize=codesize+accum(i)/2;
end;
if cont then
call scanner:
else
do:
    if subtype = dec then
call gen two(0);
    else
codesize = codesize + 2;
call generate(0);
return;
end:
end gencon;

processstore: proc(a):
declare a byte:
if chktyp5 then
call generate(a+2);
else
call chktype(a);
end processstore:

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

get$field: proc:
gen$read: proc(s):
declare a byte:
if stypempi = int then
do:
call generate(a):
call generate(s):;
end:
else
    if stypempi = dec then
do:
call generate(a+1):
call generate(sadd);
end:
else
do:
call generate(a+2):
call generate(sadd);
end:
end gen$read:
call gen$loc(stypempi,symlocmpl):
if filelo then
call gen$read(rd):
else
call gen$read(rc):
end get$field:

put$field: proc(s):
declare a byte:
if filelo then
    a = 3:
    if stypempi = int then
do:
        if typempi = subvar then
call generate(lod):
call generate($n):
end:
else
    if stypempi = dec then
call generate(a+1):
else
call generate(a+2):
end put$field:
process proc: proc(a): declare a byte; call settypeemp(a); if pass2 then do: call seterroremp(srlcemp:geterrloc); call settypeemp(getsubtype); parmbase=base+ptr+1; end; end process proc:

process ident: proc(a) byte; declare a byte; if fulllookup(a) then return gettype; 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=processident(a)) subvar) and pass2 then do: if b<>0 then call error('ln'); end; else do: call settypeemp(getsubtype); call setsymlocemp(symlocemp:getaddr); call settypeemp(subvar); end; end process array:

process ident dcl: proc(a,b,c); dcl(a,b,c) byte; pnum=pnum+1; if not normallookup(a) then do: call settype(b); if subvar then call setaddr(conntptr); call setsubtype(c); end; else if pass1 then do: if gettype=parm then do: call settype(b); call setsubtype(c): ptest=ptest-1; end; else call error('dd'); end; end process ident dcl:

process sav: proc; call genint@('pcount'); call genint@('pnum+31*2); call literal(symlocemp); call generate(sav); call generate(b11); if procstypefunc then do: if stypeemp(func) int then
do:
call lit load(symlocmp+4):
call literal (symlocmp):
if stypelscmp=doc then
  call generate(sid):
else
  call generate(ais):
end:
end proc$parm:

check$parm: proc(a):
declare a byte:
  b address:
if pass2 then
  do:
    b=base:
    base+para base:
    if a 'get subtype then
      call error('pm'):
    para base+base+ptr+8:
    base+:
    end:
    call generate(imj):
call generate(a):
end check$parm:

proc$pro: proc:
call generate(im2):
call gen$two(snilcmp):
call generate(pro):
end proc$pro:

process$proc$decl: proc(a,b,c):
declare (a,b,c) byte:
call process$ident$decl(a,b,c):
plist; count=0:
call get$symlocmp(symlocmp:=get addr):
if pass1 then
  prct=prct+4:
if (proc$type:=b) = func then
  do:
    call literal(symlocmp+4):
call generate(xch):
call generate(sid):
  end:
call branch$clause(hrs):
if pass1 then
  do:
call set$addrptr(b):
  addrptr+codesize:
  stb1=stb1+3:
savparm=stb1+1:
  end:
call incr$blk$level:
end process$proc$decl:

process$var: proc(a):
declare (a,b) byte:
if (b = process$ident(a)) = simvar then
  do:
    call set$symlocmp(symlocmp:=get addr):
call set$typesimvar:
call set$typesimvar (get subtype):
  end:
else
  if b!=func then
    call process$proc$func:
else
  if b>0 then
    call error('ip'):
end process$var:
process$simvar$decl: proc(a, b);
declare (a, b) byte;
call process$simident$decl(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$simvar$decl;

process$label: proc;
if pass1 then
  do:
    if full$lookup(mp) then
      call error('dd');
    else
      do:
        call enter;
        call setaddr(cod$size);
        call settype(lab);
        call setblk$level(blk$level);
      end;
    end;
end process$label;

resolve$label: proc;
declare (chk$blk, tindex) byte;
if pass2 then
  do:
    if ((not full$lookup(sp)) or (getype <> lab)) then
      call error('ul');
    else
      do:
        call enter;
        call setaddr(cod$size);
        call settype(lab);
        call setblk$level(blk$level);
      end;
    end;
  end;
else
  call gen$two(nop);
end resolve$label;

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

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

assign$stmt: proc;
call chktyp;
if (tpcount = tpcount-$1) then /* test multiple assign stmt */
call process$store($1);
else
call processstore(sld):
end assignstmt:

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

if listprod then
  call print$prod:
call copy:
do case production: /* call to synthesize 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 writeInt$file:
call closeInt$file:
call mon3:
  end:
/* 2 <block> ::= <block head> <block end>
*/

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

/* 4 <begin> ::= begin
  call incr$blk$level:
*/

/* 5 <block end> ::= <block body> ; end
call decr$blk$level:
*/

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

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

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

/* 10 <simple declaration>
*/

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

/* 12 <array declaration>
*/

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

/* 14 <subprogram declaration>
*/

/* 15 <external declaration>
*/
(simple declaration) ::= (declaration head) (identifier)

call process$simvar$decl(sp.aid);

(initial option) ::= (initial head) (constant)

(initial head) ::= initial (constant)

(declaration head) ::= (declaration type)

(declaration type) ::= string
do:
call settypesp(str);
call gen$intv(str*size);
end:

(string size option)
do:
call settypesp(str);
end:

(integer)
call settypesp(int);

(decimal size option)
do:
call settypesp(dec);
call gen$intv(dec*size);
end:

decimal

call settypesp(dec);

(size option) ::= (variable)
if settypesp(int) then
call litload(symblocmpl);
else
call error("si");

(integer)
do:
call move(calcvarc(mppl), accum7);
call gen$cont;
end:

(balanced statement)

(unbalanced statement)

(balanced statement) ::= (simple statement)

(if clause) <true part> else <balanced statement>

(label definition)

(balanced statement)

(unbalanced statement) ::= (if clause) <statement>'
call set$enter(tempsl);  

(balanced statement)

(unbalanced statement) ::= (if clause) <true part>
call set$enter(tempsl);  

(balanced statement)

(unbalanced statement) ::= (if clause) <true part> else <unbalanced statement>
call set$enter(tempsl);  

(label definition)

(unbalanced statement)

(true part) ::= <balanced statement>
do:
call branch$clause(brp)
call set$enter(typepsl-1));
end:

(label definition) ::= <identifier>
call process$label;

call process$label;
/* 40 simple statement ::= (block) */
/* 41 | (assignment statement) */
/* 42 | (for statement) */
/* 43 | (while statement) */
/* 44 | (read statement) */
/* 45 | (write statement) */
/* 46 | (case statement) */
/* 47 | (go to statement) */
/* 48 | (case statement) */
/* 49 call generate(pop): */
/* 50 | (identifier) */
/* do; */
/* if ((process#ident(sp) <= pro) and pass2) then */
/* call error("nf"): */
/* call process#proc(pro); */
/* call proc#pro; */
/* call generate(pop): */
/* end; */
/* 51 (assignment statement) ::= (left part) (expression) */
/* call assignStmt: */
/* 52 | (left part) */
/* 53 | (assignment statement) */
/* do; */
/* if typemp<subvar then */
/* call gen$loc(stypem#symlocmp): */
/* ipcount=ipcount+1: */
/* end; */
/* 54 (expression) ::= (arithmetic expression) */
/* | if expression (expression) */
/* 55 | (arithmetic expression) ::= (term) */
/* 56 | (term) */
/* 57 | (arithmetic expression) ::= <term> */
/* 58 | (primary) */
/* 59 call chktmpl(sdi): */
/* 60 call chktmpl(sbi): */
/* if chktyp3 then */
/* call generate(neg): */
/* if chktyp3 then */
/* call generate(not): */
/* if chktyp3 then */
/* | (term) ::= (primary) */
/* 61 | (term) ::= (primary) */
/* 62 call chktmpl(mpi): */
/* 63 call chktmpl(dvi): */
/* 64 call chktmpl(lpi): */
/* 65 call chktmpl(expr): */
/* 66 | (primary) ::= (primary element) */
/* 67 | (primary element) ::= (variable) */
/* if typemp<subvar then */
/* call litt$loc(stypem#symlocmp): */
If typesp() subvar then
  call proc@pro:
    (constant)
/x 68
    (procedure call)
/x 69
    (assignment statement)
/x 70
    call set@typep(stypep:=stypep):
/x 71
do:
  call set@typep(stypep):
  call set@typep(stypep):
end:
/x 72
  (constant) := (integer)
  call gencon(int):
/x 73
  (decimal) := (string)
  call gencon(str):
/x 74
  (variable) := (identifier)
  call process$var(sp):
/x 75
  (subscripted variable)
/x 76
  (file declaration) := (file head) <file name>
/x 77
  (file head) := file
/x 78
  (file head) := file
/x 79
  (file name) := (string) <length option>
/x 80
  (string)
/x 81
  (identifier) <length option>
/x 82
  call process$ident$dc(mp,filel.0):
/x 83
  (identifier)
  call process$ident$dc(sp,filel.0):
/x 84
  (length option) := (identifier)
  if process$ident$mp then:
/x 85
  (integer) 3
/call gencon(int):
/x 86
  (array declaration) := (array list) <bound pair list>
/x 87
do:
  call gen$int$av(arr$dim):
  call gen$int$av(arr$num):
  call gen$int$av(stypep):
  call generate(row):
  do while (arr$num := arr$num-1) > 255;
  call literal(arr$loc(arr$num)):
end:
/x 88
  (array list) := (array head) <identifier>
  call process$array$dc(sp):
/x 89
  (array head) := (declaration type) array
  arr$num:=0:
/x 90
  (array head) <identifier>
  call process$array$dc(mp):
/x 91
  (bound pair list) := (bound pair head) <bound pair> 0
/x 92
  (bound pair head) := '
  arr$dim:=0:
/x 93
  arr$dim:=arr$dim+1:
/x 94
  (bound pair head) <bound pair> 0
/x 95
  (expression) <expression>
  if (stypep(:int) or (stypep(:int)) then
    call error("bp"):
/x 96
  (expression) 0
/x 97
do:
  if stypep(:int then
    call error("<int"):
    call literal(stypep):
call generate(subt):
end:
/* 93 <subscript head> ::= <identifier> */
call processArray(tmp):
/* 96 <subscript head> <expression> */
if stypeemp int then
  call error("int"): /* 97 'go to statement' ::= <go to> <identifier> */
call resolveLabel:
/* 98 <go to> <integer> */
call resolveLabel:
/* 99 'go to' ::= <go to> */
/* 100 'goto' */
/* 101 'read statement' ::= <read head> <variable> */
do:
call getFeld:
  if filelo then
    filelo=false:
  end:
else
  call generate(ecn):
end:
/* 102 'read head' ::= read (*/
call generate(ecn):
/* 103 'read <file option> */
: /* 104 'read head' <variable> */
call getFeld:
/* 105 'write statement' ::= <write head> <expression> */
call putFeld(wlc):
/* 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 'writeon head' <expression> */
call putFeld(wlc):
/* 113 'write head' <tab expression> */
/* 114 'write head' <pic definition> */
/* 115 'file option' ::= <identifier> */
if process$Identemp $filel then:
  /* 116 'identifier' */
if process$Identemp $filel then:
  else call error("if"): /* 117 'string' */
else /* 118 'string' */
  : /* 119 'rec option' ::= 'identifier' */
  if ((process$Identemp $sимвar) and (getSubtype = int)) then:
    else call error ("int"): /* 120 'integer' */
call gencon(int):
/* 121 'pic definition' ::= <pic head> <pic list> */
/* 122 'pic head' ::= <pic string> */
/* 123 'pic string' */
if (process$Identemp $sимвar and getSubtype = string) then:
else call error("string"): /* 124 'string' */
if (process$Identemp $sимвar and getSubtype = string) then:
else call error("string"): /* 124 'string' */
/*
124  <pic list> ::= <expression>
125     |
126  <tab expression> ::= <tab <expression>]
127  <if clause> ::= if <boolean expression> then
128     call branchclause(brc);
129  <if expression> ::= <if clause> <expression> else
130  <boolean expression> ::= <boolean expression> or
131     <boolean expression>
132  call chktyp(bor);
133  <boolean term> ::= <boolean primary>
134  if chktyp3 then
135     call generate(nolo);
136     <boolean term> and
137     <boolean primary>
138  call chktyp(ando);
139  <boolean primary> ::= <logical expression>
140  if (<boolean expression> )
141     call settypep(stypep1):
142  <logical expression> ::= <expression> <relation>
143  if (stypep2a) and (stypep2a) then
144     call generate(typer3+2):
145  else
146  call chktyp(typer3);
147  <relation> ::= =
148     call settypep(eq1);
149     <true expression>
150     call settypep(eq2);
151     call settypep(eq3);
152  <comp> ::= <>
153     call settypep(neq):
154     =
155     call settypep(eq):
156     !=
157     call settypep(neq):
158  <while statement> ::= <while clause> <do statement>
159     do:
160     call generate(brc);
161     call gen@two(symlocmp);
162     call setenter(typer3):
163  end:
164  <while clause> ::= <while> <boolean expression>
165     call branchclause(brc);
166  <while> ::= while
167     call setsymlocmp(codesize):
168  <for statement> ::= <for clause> <step expression>
169  <until clause> <do statement>
170     do:
171     call generate(brc):
172     call gen@two(symlocmp);
173     call setenter(typer3+1));
174  end:
175  <for clause> ::= for assignment statement
176     do:
177     if (stypep'stvar') or (stypep'int) then
178     call error('ni');
179     call generate(brc);
180     call compiler@label;
181     call setsymlocmp(codesize);
182     call literal(symlocmp);
183     call literal(label: symlocmp):
184     call literal(loca: symlocmp):
185 */
/x 149 do:
step expression ::= step (expression)
end:
do:
if stepflag = true:
call error("nil");
stepflag = false:
call stepgen;
end:
/* 150 until clause ::= until non-term (expression)
do:
call generate(sleq);
call branchclause(last);
end:
/* 131 until non-term ::= until
if not stepflag then
do:
call generate(intv(1));
call stepgen;
end:
else
stepflag = false:
/* 152 do statement ::= do simple statement
/* 153 close statement ::= close identifier
call closefile(sp):
/* 154 close statement, identifier
call closefile(slp):
/* 155 subprogram declaration ::= (subprogram heading)
/* 156 (simple statement)
do:
call literal(symlocmp):
call generate(uns):
call generate(rtn):
call declbkllevel:
call setenter(symemp):
call literal(symlocnum-2):
call genintv(0):
call generate(sid):
mnum, mcount = 0:
if flag = false:
end:
/* 157 (function heading)
/* 158 procedure heading, Proc>
/* 159 function & param
/* 160 (paramless proc)
/* 161 (proc & param
/* 162 paramless function ::= (declaration type) function
/* 163 function & param
/* 164 function head
/* 165 declaration
/* 166
call generate(i=2);
call gen@io(getaddr-4);  
call generate(bra):
call generate(bir):
call enter@compiler$shb-1(0):
if0:
end:
/* 181  <case block> ::= begin (case block body) end */
/* 182  (case block body) ::= (statement) */
end:
/* 183  (case block body) ::= (statement) */
end of case statement/
end synthesize:

/recovery error routine
module:
  noconflict: proc (estate: byte):
    declare estate state(size: (i,j,k) index:size:)
    k = j * index2(estate) - 1;
    do i = j to k;
      if read(i) = token then return true:
    end:
    return false:
  end noconflict:
  recover: proc (state: size:)
    declare state stack: byte: state stack: size:;
    do forever:
      sp = sp - 1;
      do while (sp < 255):
        if noconflict(state: stack: sp) then
          if state will read token then
            if sp <> sp then sp = sp - 1:
            return state:
          end:
          sp = sp - 1:
        end:
      end:
    end:
    call scanner:
    end:
  end recover:

module: parser routines
module:
  do: /*block for declaration of*/
  declare (i,j,k) index:size: index byte:
  initialize: procedure:
    call title:
    call initialize@scanner:
    call initialize@synthesizer:
    end initialize:

getin1: procedure index:size:  
return index1(state:
end getin1:

getin2: procedure index:size:  
return index2(state:
end getin2:
incsp: procedure;
  if (sp := sp + 1) = length(states) then
    call error('too');
end incsp:

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

set$var$1: procedure(i:
  declare 1 byte;
  set varc: and increment varindex x/
  varc(varindex)$i:
  if(varindex: varindex$1) > length(varc) then
    call error('var');
  end set$var$1:

/* initialize for input - output operations */
call move$rch..wch.999: /* put filename in write fcb */
call setup$int$filename: /* 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 sta$=maxno then /* read state */
      do:
        call incsp: st$:states(sp)=state;
        i=get$11: call lookahead:
        j=i+get$12-1:
      do i = 1 to j:
        if read(token) then /* save token */
          do: var(sp)=varindex:
            /* copy to proper position */
            do index := 0 to accum:
              call set$var$1(accum(index));
            end:
            hash(sp)=hashcode:
            /* save relative table location */
            state=starts; end:
      end:
      else
        if i: then
          do:
            call error('sp');
            if(state < recover10 then compiling: false:
          end:
        end:
        else
          if state=maxno then /* apply production */
            do:
              sp:=sp-get$11: set$11:
              production = state-maxno:
              call synthetize:
              sp:=sp-get$11:
              varindex=starts:
        end:
      end:
  end:
end:
j = statestack(en);
   do while k = apply1(i) <> 0 and j <> k:
      j = j+1;
   end;
   if state = apply2(i) = 0 then compiling = false:
      end; else
   if state = maxino then "lookahead state */
do:
   s = getini; call lookahead:
      do while k = look1(i) = 0 and token (1) k:
         t = t+1;
      end;
   state = look2(1):
   end; else
/* push state */
do; call incsp:
   statestack(sp) = sp+2;
   state = getini; 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*/
oef
190h: /*load point for interp program*/

declare /*global literals*/
  lit literally 'literally'.
  true lit '1'
  false lit '0'
  forever lit 'while true'.
  cr lit '0dh'
  if lit '0ah'.

declare /*op codes for algol-n machine instructions*/
  decl lit '25'.
  str lit '1'.
  int lit '2'.
  db lit '3'.
  bci lit '4'.
  boc lit '5'.
  iml lit '27'.
  im2 lit '28'.

declare /*interface points for c/r/m and interp*/
  bdos lit '05h'.
  hoot lit '0h'.
  diskbuff loc lit '80h'.
  fc bloc lit '5ch'.
  diskbuffend lit '100h'.
  bdosbegin address initial(000h).
  buf address initial(diskbuffend).
  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.
  templ address.
  t1 based templ byte.
  temp2 address.
  t2 based temp2 byte.

/* declarations for interpreter */

declare contz lit '0ah'.
  quote lit '22h'.
  what lit '63'.

declare eolchar lit '0dh'.
  eoffiller lit '0ah'.
  intsize lit '128h'.
  disksize lit '128h'.
  stringdelim lit '22h'.
  conbuffsize lit '80h'.
  consote lit '0h'.
  usstock lit '48h'.
  max$shk$level lit '10h'.
  negative lit '0'.
  positive lit '1'.

declare ea address.
  rb address.
  rc address.
c        based       rc       byte.
twobyteoperand based       rc       address.
sb       address.
st       address.
br       based       ra       byte.
ara       based       ra       address.
rb       based       rb       byte.
mpn       address.
med       address.
declare inputbuffer byte initial(conbuffsize).
bufspace(conbuffsize) byte.
inputindex byte.
conbuffptr address.
con@char based       conbuffptr byte.
inputptr address.
input@char based       inputptr byte.
umread 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 '067h'.
rereadaddr address.
inputtype byte.
field@length byte.
sign byte.
declare fieldaddr address.
fbp            address.
packed            address.
buffsend address.
record@pointer address.
buffer address.
nexttick@char based       record@pointer byte.
reg@length lit.
blocksize lit.
error@flag byte.
bk(max@blocksize) address.
byte@written address.
firstfield byte.
eofnode address.
eofrb address.
declare (r0,r1,r2) (11)
(sign0,sign1,sign2) byte.
decep0,decpt1,decpt2 byte.
cte 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.
pcl@value based pcl@ptr address.
counter byte.
mov@nt address.
ret@value address.

/*cp/m interface routines*/

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

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

mon3: procedure:
  goto boot;
end mon3;

printchar: procedure(char):
  declare char byte;
  call mon2(char);
end printchar;

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

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

/* procedures for build */

openintofile: procedure:
  fnp(9) = 'a':
  fnp(10) = 't':
  fnp(11) = 'l':
  fnp(32) = 0:
  if mon2(15, filename) = 255 then
    do:
      call printchar('at
    call crlf;
    call mon3;
    end:
  end openintofile:

readintofile: procedure byte:
  return mon2(20, filename):
end readintofile:

/*global procedures*/

inbuf: procedure:
  if (buff := buff+1) = diskbuff then
    do:
      buff = diskbuff;
If read<int>file \( < \) 0 then

char = 7fh;
end
end inclbuf;

storechar<inc>procedure:
  b = char;
  codeptr = codeptr+1;
end storechar<inc>:

nextchar<procedure byte>:
  call inclbuf;
  return curchar := char;
end nextchar<procedure byte>:

get<two>bytes<procedure>:
  h(i) = nextchar;
  b = nextchar;
end get<two>bytes<procedure>:

inc<codeptr>two<procedure>:
  codeptr = codeptr + 1 + 1;
end inc<codeptr>two<procedure>:

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

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 */
        p<two> = p<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;
  ptr1<two> = ptr<two> + 1;
  do while p<one> <> 0:
    if (p<one> = '0') and (p<one> = '9') then do:
      call pack;
      ti = ti + 1;
    end:
  else
    if p<one> = '.' then do:
      12 = ti; /* left offset to decpt */
    else
      do:
        error$flag = true;
        return;
      end:
    if (ptr<one> := ptr<one>+11 = diskbuffend) and hidd$flag then do:
      p<one> = diskbuffend;
      if read<int>file \( < \) 0 then do:
        p<one>(2) = 7fh;
        return;
end:
end:

if switch then
do:
t1 = t1 + 1;
p1two = p1two + 1;
end:
t2 = ti - t2; /* right offset to fetch */
t1 = ti / 2;
p1two = t1 + 2;
p1two = p1two * 1;
p1two = positive; /* this field used for design */
end pack
/* 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, file addr):
end open:

close: procedure byte:
return mon2(16, file addr):
end close:

diskread: procedure byte:
return mon2(20, file addr):
end diskread:

diskwrite: procedure byte:
return mon2(21, file addr):
end diskwrite:

make: procedure byte:
return mon2(22, file addr):
end make:

delete: procedure:
return mon2(22, file addr):
end delete:

setdata: procedure:
call mon1(19, file addr):
end setdata:

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

mask: procedure(location) address:
declare location address,
1 based location address:
return mon 1 and 0bfff:
end mask:

checkIntSign: procedure value byte:
declare value address:
if not high value:
return negative:
else
return positive:
end checkIntSign:
checkSint: procedure(stack$loc): byte;
declare stack$loc address;
if not (rol(high(stack$loc),2)) then return true;
else return false;
end checkSint;

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

setUp$neg: procedure;
if not checkSint(sign(ara)) then
ara = ara or 4000h;
if not checkSint(sign(arb)) then
arb = arb or 4000h;
end setUp$neg;

check$neg: procedure;
if not checkSint(sign(arb)) then
arb = arb and #bffh;
end check$neg;

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

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

move: procedure(source,dest,count);
declare source address;
dest address;
count byte;
schar based source byte;
dchar based dest byte;
do while(count := 1;

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;
char byte;
d based dest byte;
do while (n := n-1) ": #0ffh;
d = char;
dest = dest + 1;
end;
end fill;

error$neg: procedure(msg):
declare msg address;
call print$char(" ");
call print$char(high(msg));
call print$char(low(msg));
call print$char(low(msg));
end error$neg;
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$buffer.fill$(20);
end initialize$disk$buffer;

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

set$buffer$status$byte: procedure(byte);
    declare status byte;
    fch(33) = status;
end set$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$byte or 00h);
end set$random$mode;

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

disk$eof: procedure;
    if eof$addr = 0 then
        call error("eof");
        re = eof$addr + 1;
        ra = eof$ra;
        rb = eof$rb;
        goto eof$exit;
end disk$eof;
fill@filebuffer: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@diskbuffer:
call set@buffer@active:
return:
end fill@filebuffer:
write@disk@ifreq: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@diskbuffer:
      end:
    record@pointer := buffer:
    end write@disk@ifreq:
atSend@diskbuffer:procedure byte:
  return (record@pointer := record@pointer + 1) := buffer:
  end atSend@diskbuffer:
var@block@size:procedure byte:
  return blocksize <> 0:
  end var@block@size:
store@rec@ptr:procedure:
  fcbadd(12) := record@pointer:
  end store@rec@ptr:
write@@byte:procedure(char):
declare char byte:
if var@block@size and (bytes@written := bytes@written + 1) > blocksize then
  call error('er'):
  if atSend@diskbuffer then
    call write@disk@ifreq:
  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@@byte:
set@file@addr:procedure:
  prt@addr := mask(19):
  file@addr := prt@entry:
  sof@addr := fcbadd(19):
  call pop@stack:
  end set@file@addr:
set@file@pointers:procedure:
  buffer@end := (buffer := file@addr + 30) + disk@size:
  record@pointer := fcbadd(17):
  blocksize := fcbadd(14):
  call set@index:
  end set@file@pointers:
setup@file@entry:procedure,
if open = 255 then
do:
  if make = 255 then
    call error("me");
end:
end setupfileextent;

diskopen: procedure;
declare (filename, buff, blksize) address,
  (i, j) byte,
  char based buff byte;
incj: procedure byte;
  return (j := j + 1);
end incj;

printaddr = mask(ra);
call popstack;
blksize = ara;
call popstack;
buff = mask(ra);
call popstack;
buff = buff - char;
fileaddr, ptrentry = ra;
buffer = ra + 0;
call fill((filename = ra + 1),",", 11);
if char(2) = ':' then
do:
  bra = (char(1) and $ff) - 1;
  l = char - 2;
  buff = buff + 2;
end:
else
  l = char;
if l > 12 then
  l = 12;
  buff = buff + 1;
  j = 255;
do while (char(incl) <> ".") and (j < 8):
end;
call move (buff, filename, j);
if l > incl then
call move (.char(j), filename=8, l-j);
call setupfileextent:
ara(19) = 0;
ara(18) = ra + 256;
ara(17) = blksize;
ra = ra + 168;
end diskopen;

setoffstack: procedure:
  enra = ra;
  eoffb = rb;
end setoffstack;

setupdisklo: procedure:
call set(fileaddr);
call set(file pointers);
byteswritten = 0;
firstfield = true;
end setupdisklo;

randomsetup: procedure;
declare bytecount address,
  record address,
  extent byte;
if not varblocksize then
  call error("run");
arra = ara - 1;
call setrandommode;
call setbufferinactive;
call write disksize;
bytecount = blocksize * ara;
recordpointer = (bytecount + 1) / 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 set@file@extent;
  end;
  fcb(32) = low(record) and 7fh;
call pop@stack;
end random@setup;

get@disk@char: procedure byte;
if attend@disk@buffer then
do:
  call write@disk@ifree;
  call fill@file@buffer;
  end:
if not active@buffer then
  call fill@file@buffer;
if next@disk@char = eof@ll then
  call disk@eof;
return next@disk@char;
end get@disk@char:

disk@close: procedure;
call set@file@pointers;
call write@disk@ifree;
print@addr = mask(link);
if close = 255 then
  call error('ce');
print@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 = printbuffer;
do while char = ' ';
temp = temp - 1;
end:
call chr;
do printbuffer := printbufferloc to temp;
call printchar(printpos);
end:
call clear@print@buff;
end dump@print@buff:
load@print@buff: procedure(chr):
declare char @byte;
printpos := char;
if (printbuffer := printbuffer +1)
  printbuffer := 0 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(chr):
  " to console ";
else
call write@byte(chr):
  " to disk ";
end output:
write@byte: procedure(source, dest):

Declare source address.
num byte,
num digits byte,
char byte,
index byte,
dest byte,
count byte;
storeOne: procedure;
if (switch := not switch) then
  do:
    char = shr(hbyte,4) or '0';
  else
    call output(dest, char);
  end;
else
  do:
    call output(dest, hbyte and 0fh or '0');
    hold = hold - 1;
  end;
  count = count + 1;
end storeOne;

hold = source;
sign = hbyte(1);
if sign or 00h then
  call output(dest, ' ');
else call output(dest, '-');
count = 0;
hold = hold - hbyte;
numdigits = hbyte * 2;
hold = hold + 1;
um = numdigits - hbyte;
switch = false;
hold = hold + 1;
do index = 1 to numdigits+1:
  if count = num then
    do:
      call output(dest, ' ');
      count = 100;
    end;
  else
    call storeOne;
  end;
end writeStr;

writeStr: 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 writeStr;

writeInt: procedure(value, dest):
declare value address, i byte, count byte;
declare declint(5) address initial(1000), (100, 100, 10, 1);
declare (flag, dest) byte;
sign = checkIntSign(value):
if sign = negative then
  do:
    value = -value and 0ffh; /* mask off next to left bit */
call output(dest, '-');
  end;
else call output(dest, ' '):
  flag = false;
do l = 0 to 4:
count = 30h:
do while value = 0:
  value = value - dec(l);
  do:
    if flag then
      call output(dest, ' ');
      value = value - dec(l);
    else:
      call output(dest, ' ');
    end;
count = count + 1;
  end:
call output(dest, ' ');
end writeInt;
flag := true;
count := count + 1;
end:
if flag or (1 := 4) then
call output(dest, count);
else
call output(dest, 1);
end;
end write4int;

write4to4disk: procedure (type);
/* type 0-integer, type 1-decimal, type 2-string */
declare type byte;
if type = 0 then
call write4int(ara, l);
if type = 1 then
call write4int(ara, l);
if not firstfield then /* separate fields with commas */
call write4byte(", ");
else firstfield := false;
if type = 2 then
do:
call write4byte(quote); /* strings put in quotes */
call write4str(ara, l);
call write4byte(quote); /* add trailing quotes */
end;
call pop4stack;
end write4to4disk;

concatenate: procedure;
declare (size1, size2, tsize) byte;
temp1 := mask(ra);
call pop4stack;
if not check4(temp(ara)) then
do:
temp2 := mask(ra);
temp2 := temp2 * t2;
tsize2 := t2 + 1;
call move(temp2, ra, size2);
end;
else
do:
tsize2 := bra;
ra := ra - bra;
end;
temp1 := temp1 - t1;
tsize := t1;
call move(temp1 + 1, ra + size2, size1);
tsize := size1 + size2;
bra := tsize - 1;
ra := ra + tsize;
bra := tsize;
bra(i) := 0:
/* set temp bits */
end concatenate;

convert4to4int: procedure (loc, size) address;
declare decint(5) address, initial (i1000, i000, 100, 10, 1),
loc address,
hold address,
h based hold byte,
num address,
(i1, j, size) byte;
num := 0;
j := 3;
hold := loc + size - 1;
if size < 3 then
call error("too");
else
do i := 1 to size:

num = num + (h - 30) * decimal(y + z - 1);
hold = hold - 1;
end;
if num <= 10233 then
return num;
else
error('io');
end;
end convert$to$stat:

ones$left$:
procedure:
declare ctr byte;
if shrink$byte.(ctr:4) = 0 then
do:
do ctr = 1 to 9;
shr$byte(ctr) = shr$byte(ctr).4 or shr$byte(ctr + 1).4
end:
end:
else no$shift = true;
end ones$left:

ones$right$:
procedure:
declare ctr byte;
ctr = 11;
do index = 1 to 9;
ctr = ctr - 1;
shr$byte(ctr) = shr$byte(ctr).4 or shr$byte(ctr + 1).4
end:
end ones$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;
no$shift$ = false;
do ctr = 1 to count;
call one$left$;
if no$shift$ then
return;
end;
end shift$left$:

leading$zeros$:
procedure(addr byte):
declare count byte,
ctr byte,
addr address:
count = 0;
base = addr;
do ctr = 1 to 9;
if (shr$byte(ctr) and 0fh) = 0 then
return count;
end:
end:
if base = 0 then
do:
call error('dz');
shr$byte(9) = 10h;
depth = 1;
return 16;
end;
return count;
end leading$zeros$:

right$greater$:
procedure byte:
declar: (i. ctr) byte;
do ctr = 1 to 9;
  if ri ctr) > 128 then (99h - ri ctr)) then
    return true;
  if ri ctr) < 1 then
    return false;
end:
elfflag = true:
return true:
end ri greater:
align: procedure:
  declart (x, y) byte:
rightop: procedure (addr);
  declare addr address:
  if no shift then
    do:
      base = addr:
      call shift right(y := x - ctr);
    end:
  end rightop:
y = 0;
  if decp0 > decptl then
    do:
      base = .rl;
      call shift left(x := decp0 - decpt1);
      decptl = decpt1 + ctr - 1;
      call rightop (.rl);
      decpt0 = decpt0 - y;
    end:
  else
    do:
      base = .rl;
      call shift left(x := decptl - decp0);
      decp0 = decp0 + ctr - 1;
      call rightop (.rl);
      decptl = decptl - y;
    end:
  end align:
addor: procedure (second, dest);
  declare (second, dest) address, (index, cy, a, b, l) byte:
  hold = second:
  base = dest:
  cy = 0:
  ctr = 10;
do index = 1 to 11:
    a = ri ctr);
    b = hbyte (ctr);
    i = dec (a + cy);
    cy = carried and 1;
    i = dec (1 + b);
    cy = (cy or carry) and 1;
    hbyte (ctr) = i;
    ctr = ctr - 1;
end:
if cy then
  do:
    ctr = 10;
do index = 1 to 11:
    i = hbyte (ctr);
    i = dec (1 + cy);
    cy = carried and 1;
    hbyte (ctr) = i;
    ctr = ctr - 1;
end:
end addor:
compliment: procedure (numb);
  declare numb byte:
do case numb:
    hold = .r0;
    hold = .r1;
    hold = .r2;
end;
if sign0(numb) then sign0(numb) = negative:
else sign0(numb) = positive:
do ctr = 0 to 10:
    h$byte(ctr) = 99h - h$byte(ctr);
end;
eend compliment;
right$justify:procedure(numb):
declare (numb,i) byte:
do case numb:
    base = .r0;
    base = .r1;
    base = .r2;
end:
l = 0:
do while (((i:=l+2) < dec$byte(numb)) and (h$byte(9)=0)):
call shift$right(2):
end:
dept0(numb) = decpt0(numb) - (l-2):
end right$justify;
set$mul$div:procedure:
noshift = false:
if (sign0 and sign1) or
  (not sign0 and not sign1) then
sign2 = positive:
else sign2 = negative:
call fill (.r2,0,10):
end set$mul$div;
add$series:procedure(count):
declare (i,count) byte:
do i = 1 to count:
call add$series(.r2,.r2):
end;
ed add$series:
multiply:procedure(value):
declare value byte:
if value ":" 0 then
do:
    if noshift then
      call error('ov');
    call add$series(value):
end:
bases = .r0:
call ones$left;
end multiply:
divide:procedure:
declare (i,j,k,x,1z0,1zl) byte:
call set$mul$div;
if flag then false:
if (1z0 "= leading$zeros(.r0))
  (1zl "= leading$zeros(.r1)) then
do:
  if 1z0 > 1zl then
    do:
      base = .r0;
      call shift$left(i:=(z0-1zl):
      decpt0 = decpt0 + i:
      x = 1zl:
    end:
else
do:
  base = .r1:
  call shift$left(i:=(z0-1z0):
end;
decpt2 = decpt1 + 1;
 x = 1x0;
end;
else x = 1x1;
 decpt2 = 19 - x + decpt1 - decpt0;
call compliment(0);
do l = x to 19:
j = 0;
do while l is greater and sflag:
call addreg(r1,rl):
j = J + 1;
if sflag = true then sflag = false;
k = shr(I,1);
if I then
 r2(k) = r2(k) or j:
else r2(k) = r2(k) or shl(j,4):
base = .r0;
call one@right;
end:
divide:
checkresult: 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 sflag then
sign2 = positive:
end:
decr checkresult:
checksign: procedure:
sflag = false;
if sign0 and sign1 then
do:
sign2 = positive;
return;
end:
sign2 = negative;
if not sign0 and not sign1 then
do:
sflag = true;
return;
end:
if sign0 then call compliment(1);
else call compliment(0);
decr checksign:
add: procedure:
call checksign:
call addreg(r1,r2):
call checkresult:
decpt2 = decpt0;
end add:
cpy@reg2@on@stack: procedure:
declare count byte;
 1 byte:
call right@justify(2):
call pop@stack:
count = 0;
base = .r2;
l = 16 - (decpt2+1)/2;
do while (l is byte = 0 and count < 1):
  base = base + 1;
  count = count + 1;

end;
rs = rb + 2;
brs(0) = (count = 10 - count);
brs(1) = decSp(2);
call move(base, ra + 2, count);
brs(count + 2) = count + 2;
brs (count + 3) = sign2 or 000;  // set sign and temp bits r
rs = ra + count + 2;
end

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$num:
base = .r0;
base = .r1;
base = .r2;
end:
call fill(base, 0, 11);
hold = hold + 1;
deSp(0)(reg$num) = h$byte;
hold = hold + 1;
call move(hold, base + 10 - count, count);
end load$reg:

setSup$regs: procedure;
noShift = false;
call load$reg(rs, 0);
call load$reg(rb, 1);
call right$justify(0);
call right$justify(1);
end setSup$regs:

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

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

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

g$ptr$addr: procedure(offset address);
declare offset address;
return ptr$base + offset;
end g$ptr$addr:

load: procedure(addr);
declare addr address.
ptr$addr = mask(addr);
ptr$entry;
end load:

store$dec: procedure(source, dest);
declare (source, dest, dest$entry)
(adj$to, avail$to, h$byte) byte,
a based source address.
d based dest address.
decr byte.
sign based dest & sign byte:
if check@temp(s) then
templ = source;
else templ = mask(source);
temp2 = mask(dest);
availsto = t2 - 2;
dest@sign = temp2 + 1;
t2(1) = t1(1) and 0th // mask off temp bits
source = (templ * templ - 1);
dest = temp2 - t2;
availsto = t1;
decpt = t1(1);
sigbytes = ((availsito + 2 - decpt) + 1) / 2;
if availsito <= availsto then
do:
call fill(dest, 00h, t2);
call move(source, dest, t1+2);
end:
else if sigbytes < availsto then
do:
call move(source, dest, t2);
t2 = dest;
t2 = availsto;
t2(1) = (availsto - sigbytes) * 2;
if decpt then
t2(1) = t2(1) + 1;
call warning('11');
end:
else
do:
hold = dest;
hradr = 0101h;
hr@byte(2) = 10h;
sign = positive;
call error('11');
end:
end store@dec:
store@int: procedure(dest, value);
declare (dest, value) address;
prnt@adr = mask(dest);
prnt@entry = value;
end store@int;
store@str: procedure(source, dest);
declare (dest, source) addresses.
da based source address.
d based dest address:
if check@temp(s) then
templ = source;
else templ = mask(source);
temp2 = mask(dest);
dest = temp2 - t2;
source = templ - t1;
if t1 <= t2 then
do:
call fill(dest, 20h, t2);
call move(source, dest, t1);
end:
else
do:
call move(source, dest, t1);
call warning('so');
end:
end store@str:
allocate@str: procedure:
if brs = 0 then
do:
call error('az');
brs = 10;
end
  call push#stack(brn:=brn+1);
  brn = brb;
end allocate#str;

allocate$loc:procedure:
declare store byte;
  if brn = 0 then
do:
    call error('az');
    brn = 9;
  end:
  ctr = brn;
  store = (brn+1)/2+2;
  call push#stack(store);
  brn = store;
  ars = ars or 0100h;
end allocate$loc;

set$up$allocation:procedure:
  pushd ir = mask(ra);
  call pop$stack;
end set$up$allocation;

find$orb:procedure:
  if check$temp(ars) then
    rb = ra - (brn + 2);
  else rb = ra - 2;
end find$orb;

save$pchb:procedure:
  if move$cnt = 0 then
    return;
  call push#stack(2);
  call move(pchptr,ra,move$cnt);
  pch$value(1) := ra;
  ra = ra + move$cnt;
  ars = move$cnt;
end save$pchb;

unsave:procedure:
  call move({expl:=ra-ars,hold,ars})
  ra = {expl}
end unsave;

calc$row:procedure:
declare (index,num,v,size:type) byte,
  (numarrays,numdim,allocation,counts) byte,
  d (10) byte;
  type = brn: .x 1-int.2-dec.3-str */
call pop$stack;
  numarrays = brn;
call pop$stack;
  numdim = brn;
call pop$stack;
  v := 0;
  isize = 1:
  d(numdim) := 1;
  if numdim = 1 then
do:
    isize = isize + brb + 1;
    v := brb;
    call pop$stack;
    call pop$stack;
end:
else
  do index = 1 to num$dim:
    = numdim - index;
  isize = isize + num := brn - brb + 1:
  d(index) := num := d(index+1);
  v := v + brb := d(index+1);
  call push$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-insent(1):
  bra = c(1);
  bra(1) = c and 3fh;
  prt-addr = arr + ptr-base:
  ptr-entry = ra or 4000h: /* set addr bits */
  bra = num-dim:
  if num-dim <> 1 then
    do index = 1 to num-dim - 1:
      ra = ra + 1;
      bra = alloc-len;
    end:
  end:
  ra = ra + 1;
  bra = 0;
  bra = alloc-len;
  if type = int then
    do:
      bra = 2;
      ra = ra + 1;
      hold = ra;
      ra = ra + absise * 2;
      h-addr = ra;
    end;
  else
    if type = decI then
      do:
        bra = (bra+1)/2 + 4;
        ra = (hold := ra+1) + 2;
        bra = alloc-len;
        do index = 1 to absise:
          call allocate-dec:
          call push-stack(2):
          bra = c(tr; /* 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 absise:
            call allocate-str:
            call push-stack(2);
            arr = arr;
          end:
          call pop-stack:
          h-addr = ra;
        end:
      end:
    end:
  call step-insent(1):
end counts;
end calc-row:

calc-row: procedure:
declare array-addr address:
location address:
abs-byte based array-str byte:
abs-addr based array-addr address:
(1 num-dim) byte:
offset address:
array-addr = mask(rn):
call pop-stack:
offset = arr:
num-dim = abs-byte:
do 1 = 2 to numaddr;
call pop@stack;
arrayaddr = arrayaddr + 1;
offset = arr * a$byte + offset;
end;
arrayaddr = arrayaddr + 1;
offset = (offset - a$byte + 1) = a$byte();
arrayaddr = arrayaddr + 2;
if (location = arrayaddr + offset) a$addr then
call error('ab');
end calc@sub;

decrement@blk: procedure(num);
declare num byte;
ra = 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 printfchar('-');
call printfchar('');
call printfchar('');
call read(), inputbuffer);
if buff$space(1) = cntz then
call exit@interp;
numRead = buff$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$err: procedure;
re = readdr; /* reset program counter */
call warning ('11');
goto error@exit; /* return to outer level */
end console@input$err;

get@con@char: procedure byte;
conbuffptr = conbuffptr + 1;
return con@char;
end get@con@char;
next@input@char: procedure byte:
  if input$type = 0 then /* read from disk */
  /* do forever:
    if (buffer$space (input$index) := get disk$char) = if then
    do:
      if var$blocksize then
        call error'(set'
    end:
    else
      return next@disk$char:
    end
    if input$type = 1 then /* input from console */
      return get$console$char:
    end
  next@input$char:
get$field: procedure:
  declare hold byte, delim byte;
  field$length = 0;
  do while (hold := next@input$char) = ' ':
    end:
  if input$type = 0 then
    input$ptr = buffer$space:
  if input$type = 1 then
    input$ptr = console$buffer$ptr:
  if hold <> quote then
    delim = ' ':
  else:
    delim = quote:
    hold = next@input$char:
  end:
  do while (hold <> delim) and (hold <> eof$char):
    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:
      input$ptr = input$ptr + 1:
      field$length = field$length - 1:
    end:
  else
    if input$char = '+' then
      do:
        sign = 0:
        input$ptr = input$ptr + 1:
        field$length = field$length - 1:
      end:
    else
      sign = 0:
      call push$stack(2):
      are = convert$to$int(input$ptr,field$length):
      if error$flag then
        call console$input$error:
        if sign then
          are = -are and 0xff#h: /* set neg bit and conv to neg int */
        end
        get$int$field:
    end
get$str$field: procedure:
  call get$field:
  con$char = field$length + 1:
  input$ptr = console$buffer$ptr + con$char:
  input$char = field$length:
  call push$stack(2):
  are = console$buffer$ptr or 4000#h:
  end get$str$field:
get#dec#field: procedure:
call get#field:
call push#stack(2):
cond#char = 0; /* set binary 0 as end of dec */
if input#char = '0' then
do:
   sign = 1;
   input#ptr = input#ptr + 1;
end:
else
if input#char = '=' then
do:
   sign = 0;
   input#ptr = input#ptr + 1;
end:
else
   sign = 1;
call pack#decimal(input#ptr, rm);
if error#flag then
call console#input#error:
   rm = ptr两点 = 1;
   brs(i) = sign or 0@8h; /* set sign and temp bits */
end get#dec#field:
initialize#execute: procedure:
stack-top=3050h:
med, rc = code#base:
opr = pr#base:
str, sb = stack#base:
rm = (rb := sb) + 2:
begin#flag := false:
end initialize#execute:
/* set up machine */
call print("algol-m interpreter-vers 1.08");
call crlf:
call open#int#file:
begin#flag := true:
call incbuf; call incbuf; /* skip codesize */
pr#base = .memory:
code#base = get#parm + pr#base:
code#ptr = code#base:
/* load machine */
do while next#char <> 7fh:
   if curchar = 128 then
do:
call sto#char#inc:
call incbuf:
call sto#char#inc:
end:
else
   if curchar = str then
do:
call sto#char#inc:
   incr = code#ptr:
   char = 0; /* set initial length to zero */
call sto#char#inc:
do while next#char = 0:
call sto#char#inc:
t = t + 1;
end:
char = t + 1:
call sto#char#inc:
char = 0; /* must set str length an odd quantity */
call sto#char#inc:
end:
else
   if curchar = int then
do:
call sto#char#inc:
hold = buff + 4:
fieldlength = 0;
do while nextchar <> 0:
    fieldlength = fieldlength + 1;
end;
d = convert(Stoint(hold, fieldlength);
codeptr = codeptr + 2;
end;
else
    if curchar = dell then
do;
call stochar$inc;
call incbuf;
call pack$decimal(buff, codeptr);
codeptr = codeptr + 2;
buff = ptr$one + 2;
end;
else
    do:
call stochar$inc;
    if (curchar = brs' or 'curchar = bsc) then
do;
call get$two$byte$es;
    a = a + codebase;
call inc$codeptr$two;
end;
else
    if (curchar = i1) or (curchar = deb) then
do;
call incbuf;
call stochar$inc;
end;
else
    if curchar = im2 then
do;
call incbuf;
call stochar$inc;
call incbuf;
call stochar$inc;
call incbuf;
call stochar$inc;
end;
end;
end;
stackbase = codeptr;

execute: procedure:
do forever:
    if rol(c.1) then /* must be lit or lit+lit */
do:
call push$stack(2);
brs = c(1); /* load in reverse order */
brs(1) = c and 3fl; /* mask bits 10 */
arm = ptr(arm + ptr$base) or 4000h; /* set @i addr bits */
if rol(c.2) then call load$main:
call step$ins$nent(1);
end:
else
    do case c:
/* @ case @ not used*/
/* @ lit */
do:
call push$stack(2);
call step$ins$nent(1);
rc = rc + c + 1;
arm = rc or 4000h;
call step$ins$nent(1);
end:
/* 2 int */
do;
call push@stack(2);
call step@instcnt(1);
ara = two@byte@opnand;
call step@instcnt(1);
end;

/* 3 xch */
do;
hold = ara;
ara = arb;
 arb = hold;
end;

/* 4 lod */
call load(ra);

/* 5 dch */
do;
call step@instcnt(1);
call decrement@bkcr;
end;

/* 6 dmp */
call cr1f;

/* 7 xit */
return;

/* 8 ald */
do;
call set@up@alloc;
call allocate@dec;
prt$entry=ra or 4000h;
end;

/* 9 ald */
do;
call set@up@alloc;
call allocate@str;
prt$entry=ra or 4000h;
end;

/* 10 ald */
do;
call set@up@alloc;
call allocate@str;
prt$entry=ra or 4000h;
call push@stack(2);
bra = str;
end;

/* 11 ald */
do;
call set@up@alloc;
call allocate@str;
prt$entry=ra or 4000h;
call push@stack(2);
ara = arb;
end;

/* 12 ald */
do;
call set@up@neg;
arb = add$int(arb,ara);
call check$neg;
call pop@stack;
end;

/* 13 add */
do;
call setSupRegs: /* puts values of top two items in reg0 and reg1 respectively */
call align;
call add;
call cpyReg2SonStack;
end;

/*14 sbi x*/
do:
call setSupNeg:
    arb = subInt(arb,arn);
call checkNeg;
call popStack;
end;

/*15 sbd x*/
do:
call setSupRegs: /* puts values of top two items in reg0 and reg1 respectively */
call align;
call compliment(0);
if sign0 then sign0 = negative;
else sign0 = positive;
call add;
call cpyReg2SonStack;
end;

/*16 mp1 x*/
do:
call setSupNeg:
    arb = multInt(arb,arn);
call checkNeg;
call popStack;
end;

/*17 mpd x*/
do:
call setSupRegs: /* puts values of top two items in reg0 and reg1 respectively */
declare (1, index) byte;
call setMulDiv;
    decPt2 = decPt0 + decPt1;
i = 10;
do index = 1 to 10;
    call multiply(r1(i) = i and 0fh);
    call multiply(shrl(r1(i),4)));
end:
call cpyReg2SonStack;
end;

/*18 dji x*/
do:
call setSupNeg:
    arb = divInt(arb,arn);
call checkNeg;
call popStack;
end;

/*19 djd x*/
do:
call setSupRegs: /* puts values of top two items in reg0 and reg1 respectively */
call divide;
call cpyReg2SonStack;
end;

/ *20 x*/
/* not used x*/

/ *21 x*/
/* not used x*/

170
/*22 neg */
do:
if check#int(ars) then
ars = ~ars and 0h
else if check#temp(ars) then
ars = ars xor 0100; /*change sign bit*/
else
dot
hold = mask(ars);
hr#addr = h#addr xor 0100h.
end:

/*23 call */
/* not implemented */

/*24 cl2 */
/* not implemented */

/*25 decl */
do:
call push#stack(2):
call step#ins#cnt(1):
rec = rec + c + 2;
ars = rec or 4000h:
call step#ins#cnt(1):
end:

/*26 pop */
call pop#stack:

/*27 im1 */
do:
call push#stack(2):
call step#ins#cnt(1):
ars = c1:
end:

/*28 im2 */
do:
call push#stack(2):
call step#ins#cnt(1):
br = c(1); /*load in reverse*/
br(1) = c1:
call step#ins#cnt(1):
end:

/*29 */
/* not used */

/*30 */
/* not used */

/*31 cat */
call concatenate:

/*32 bli */
hik #hik#level + 1 = ran:

/*33 bid */
call decrement #hik(1):

/*34 brz */
call branch#absolute:

/*35 bsc */
if brz = 0 then
call branch#absolute;
else
   do:
      call stepsinsent(2);
      call popstack:
   end;
/*36 lsw */
   do:
      if brb < bra then
         brb=1;
      else brb=0;
      call popstack:
   end;
/*37 lsw */
/* not implemented */
/*38 lsw */
/* not implemented */
/*39 str */
   do:
      if brb > bra then
         brb=1;
      else brb=0;
      call popstack:
   end;
/*40 dgtr */
/* not implemented */
/*41 sgtr */
/* not implemented */
/*42 eq1 */
   do:
      if brb = bra then
         brb=1;
      else brb=0;
      call popstack:
   end;
/*43 dsql */
/* not implemented */
/*44 sq1 */
/* not implemented */
/*45 neg */
   do:
      if brb <> bra then
         brb=1;
      else brb=0;
      call popstack:
   end;
/*46 dneg */
/* not implemented */
/*47 deq */
/* not implemented */
/*48 geq */
   do:
      if brb > bra then
         brb=1;
      else brb=0;
      call popstack:
   end;
/*49 dgeq */
/* not implemented */
/*50 ageq */
/* not implemented */
/* St. log */
do;
  if brb < bra then
    brb = 1;
  else brb = 0;
  call pop@stack;
end;
/* not Implemented */
/* u-aleq */
/* not Implemented */
if bra = 0 then
  bra = 1;
else bra = 0;
end;
/* not Implemented */
/* not Implemented */
/* not Implemented */
do;
  if (bras=0) or (brb=0) then
    brb=1;
  else brb=0;
  call pop@stack;
end;
/* not Implemented */
/* not Implemented */
/* not Implemented */
do;
  if (bras=0) and (brb=0) then
    brb=1;
  else brb=0;
  call pop@stack;
end;
/* not Implemented */
/* not Implemented */
dcall write@int(ars,0);
call pop@stack;
end;
/* wdc */
dcall write@temp(ars) then
call write@dec(ra,0);
else
call write@dec(mask(rn),0);
call pop@stack;
end;
/* wdc */
dcall write@temp(ars) then
call write@email(ars,0);
else
call write@email(mask(ra),0);
call pop@stack:
end;
/*66 wid z/
call write@to@disk(0):
/*67 wid z/
call write@to@disk(1):
/*68 wid z/
call write@to@disk(2):
/*69 sbr z/
do:
   arb = ara = arb;
call pop@stack;
end;
/*70 bra z/
do:
   rc = codebase + ara;
call pop@stack;
end;
/*71 row z/
call calc@row:
/*72 sub z/
call calc@sub:
/*73 rec z/
call get@int@field:
/*74 rec z/
call get@dec@field:
/*75 rec z/
call get@str@field:
/*76 rdi z/
do:
   inputtype = 0;
call get@int@field:
end;
/*77 rdd z/
do:
   inputtype = 0;
call get@dec@field:
end;
/*78 rds z/
do:
   inputtype = 0;
call get@str@field:
end;
/*79 rem z/
do:
   inputtype = 1;
   rc = addr = rc;
call console@read;
end;
/*80 scr z/
if more@con@input then
   call console@input@error;
/*81 stil z/
do:
   call store@int(rb, arr);
   rb = rb - 2;
end;
call store#dec(ra, rb);  
r
rb = rb - 2;
end;

call store#str(ra, rb);  
r
rb = rb - 2;
end;

call store#int(rb, ara);  
call pop#stack;

call pop#stack;
end;

call store#dec(ra, rb);  
call pop#stack;

call pop#stack;
end;

call store#str(ra, rb);  
call pop#stack;

call pop#stack;
end;

call disk#open;

call disk#close;

call pop#stack;
end;

rdb =

ready sequential block x/
do:
call setup#disk#10;

call set#of#stack;
end;

ready random block x/
do:
call setup#disk#10;

call random#setup;

call set#of#stack;
end;

end of record for read x/
advances to next line feed x/
if var#block#size then
   do while get#disk#char () if;
      end:
call store#rec#ptr;
end;

end of record for write x/
do;
if var#block#size then
do while byteswritten < (blocksize - 2);
call writeaabyte(' ');
end;
call writeaabyte(cr);
call writeaabyte(lf);
call storearecsptr;
end;
/*93 pro.x/
do:
stack = stacktop - 2;
retaddr = rec;
rc = area + codebase;
call pop@stack;
end;
/*94 sav.x/
do:
declare (lm, num) byte;
pcbptr = mask(ra);
call pop@stack;
movcnt = area;
call pop@stack;
if area <> 0 then
do:
hold = ra;
counter = 2 * area + 1;
do l = 1 to counter:
call pop@stack;
end;
nm = hold - ra:
if (temp := stacktop - num) <> ra then
call error('mo');
call movcra+2,temp1,num);
call fill(temp1-2,0h,2);
end;
else
call pop@stack;
blk = blklevel+1 = ra;
if pcbAvalue(l) = 0 then
pcbAvalue(l) = 1;
else
call sav@pcb;
end;
/*95 sav2 x/
do:
declare 1 byte,
parm@count address;
tad1 = ra;
tad2 = rb;
tad3 = stacktop - 2;
rba = ra - 2;
parm@count = area;
call pop@stack;
pcbptr = pcbptr + 4 + parm@count * 2;
do l = 1 to parm@count:
testvalue = area;
call pop@stack;
if testvalue = int then

call store@int(pcbptr, area);
else
if testvalue = deci then

call store@deci(ra, pcbptr);
else

call store@str(ra, pcbptr);
call pop@stack;
pcbptr = pcbptr - 2;
end;
rm = tad1;
rb = tad2;
tad3 = 0;
end
/*96 uns */
do:
  haddr = mask(rs);
  ret$value = haddr;
  call decrement$hlk(1);
  if haddr(1) = 1 then
    call unsave;
  else
    haddr(1) = 0;
    call decrement$hlk(1);
    call push$stack(2);
    if check$int(ret$value) then
      ara = ret$value;
    else
      do:
        templ = mask(ret$value);
        call move(templ = tl.ru, tl+2);
        ra = ra + tl;
        bra(1) = bra(1) or 0e0h;
      end;
    end:
end:
/*97 rtn */
do:
  rc = ret$addr;
  stacktop = stacktop + 2;
end:
/*end case*/
call step$line$cnt(1);
error$flag = false:
end: /* if do for ever */
end execute;
mainline:
call crlf:
call initialize$execute;
eofexit: /* on end of file of current disk file come here */
error$exit: /* regroup on console input error */
call execute:
call exit$interp:
eof

LIST OF REFERENCES


13. Strutvnski, Kathryn B. Information on the CP/M Interface Simulator, internally distributed technical note.


<table>
<thead>
<tr>
<th>No.</th>
<th>Copies</th>
<th>Name and Details</th>
</tr>
</thead>
</table>
| 1.  | 2      | Defense Documentation Center  
      Cameron Station  
      Alexandria, Virginia 22314 |
| 2.  | 2      | Library, Code O212  
      Naval Postgraduate School  
      Monterey, California 93940 |
| 3.  | 4      | Department Chairman, Code 52  
      Department of Computer Science  
      Naval Postgraduate School  
      Monterey, California 93940 |
| 4.  | 1      | Assoc Professor Barry A. Kildall, Code 52Kn  
      Department of Computer Science  
      Naval Postgraduate School  
      Monterey, California 93940 |
| 5.  | 1      | LCDR Stephen T. Holl, USN, Code 52H1  
      Department of Computer Science  
      Naval Postgraduate School  
      Monterey, California 93940 |
| 6.  | 2      | Microcomputer Laboratory, Code 52ec  
      Department of Computer Science  
      Naval Postgraduate School  
      Monterey, California 93940 |
| 7.  | 1      | LCDR John P. Flynn, USN  
      Fleet Air Reconnaissance Squadron One  
      NAS Agana, Guam  
      FPO San Francisco 96601 |
| 8.  | 1      | LT Mark S. Moranville, USN  
      19109 Creekside Place  
      Salinas, California 93908 |