John Foy

engn
UMR0497

The Michigan Algorithm Decoder

(The MAD Manual)

Revised Edition - 1966

"Though this be madness, yet there is method in't."
Shakespeare: Hamlet

The Michigan Algorithm Decoder (MAD) is a computer program which translates statements describing algorithms into the equivalent machine instructions. This descriptive language--also called MAD--is explained in this manual. ALGOL 58, which was proposed at one time as a standard language for the description of algorithms, was used as a pattern for this language; the original translating program was written in 1959 for an IBM 704 computer with 8192 words of core storage. Translators were subsequently written for the IBM 709/90/94 machines by the University of Michigan Computing Center staff. Interested groups elsewhere have adapted the language for the IBM 7040, Philco 210-211, and Sperry-Rand 1107 machines. The translator was originally included in the University of Michigan Executive System (UMES), but its construction as a subroutine has permitted its inclusion in a number of different operating systems.

Over the years a number of useful extensions to the language have been incorporated. These were documented as addenda and minor revisions in the many printings of the first edition, which was written by Arden, Galler, and Graham. This edition is a major revision--done by Professor Elliott Organick of the University of Houston--which incorporates these addenda and corrects a number of shortcomings of the early version. Donald W. Boettner and Bruce J. Bolas of the University of Michigan assisted in this revision.

The MAD language has been widely used by the students and staff of the University of Michigan and it has been used at a number of other centers as well. There are a number of versions extant and some of these may not contain all of the features described herein.

Bruce W. Arden

TABLE OF CONTENTS

TABLE OF CONTENTS, continued

Chapter I

INTRODUCTION

"Begin at the beginning," the King said gravely, "and go on till you come to the end; then stop."

Lewis Carroll, Alice in Wonderland

In presenting a problem to a digital computer for solution, one transmits to the machine a procedure for solving it (usually called an algorithm for the solution of that problem), and the data for a particular case. The algorithm must be stated unambiguously and completely.

As a simple example, let us consider the problem of determining the largest number in a collection of $n + 1$ numbers $A = \{a_0, a_1, a_2, \ldots, a_n\}$ with $n \geq 1$. A verbal description of the procedure (algorithm) might be

(1) Pick up the first number.

(2) Compare it with the second number.

(3) If the first is larger or if they are equal, keep the first one.

(4) If the second is larger, keep the second one.

(5) Whichever one was saved from this comparison is now compared with the third number.

(6) Continue to repeat Steps 2 through 5 (each time moving down the list) until the $n + 1$st number has been included in the comparison.

(7) The number which has been finally saved is then the largest number in the collection A.

Unfortunately, this method of description is not very precise. Such words as "compare", "moving down the list", and "finally saved" should really be spelled out more exactly.

The following restatement of the procedure would probably be more suitable:

(1) Let $Z = a_0$.

(2) Let $j = 1$.

(3) If $j > n$, the problem is done,[†] go to Step 7; otherwise, go on.

(4) If $Z < a_j$, let $Z = a_j$; otherwise, go on.

(5) Let j increase by 1.

(6) Return to Step 3.

(7) Z is the answer.

Some further streamlining can be accomplished in the phrasing of a procedure statement without losing any clarity. This has to do with the way we denote the assigning of values

[†]This test is redundant the first time, but after n times through Steps 3 through 6 it will terminate the process for us. It is redundant the first time because if $n \geq 1$ as set forth in the problem statement, then for $j = 1$, $j > n$ will always be false.

to a variable of the problem.

Note that a statement like

"Let $Z = A_0$"

really means,

"Let Z acquire (or be assigned) the value a_0."

Now suppose we employ a special symbol, in particular, the left-pointing arrow ($\leftarrow$) to signify the process of assigning a value. Then

"$Z \leftarrow a_0$"

becomes a perfectly meaningful shorthand.

By the same reasoning, the statement

"Let j increase by 1"

can now be restated as $j \leftarrow j + 1$.

The value of $j + 1$ (the expression on the right of the arrow) is to be assigned to j (the variable on the left of the arrow). Another way of visualizing the step-by-step nature of a procedure is with the aid of a flow chart. (See Figure 1-1.)

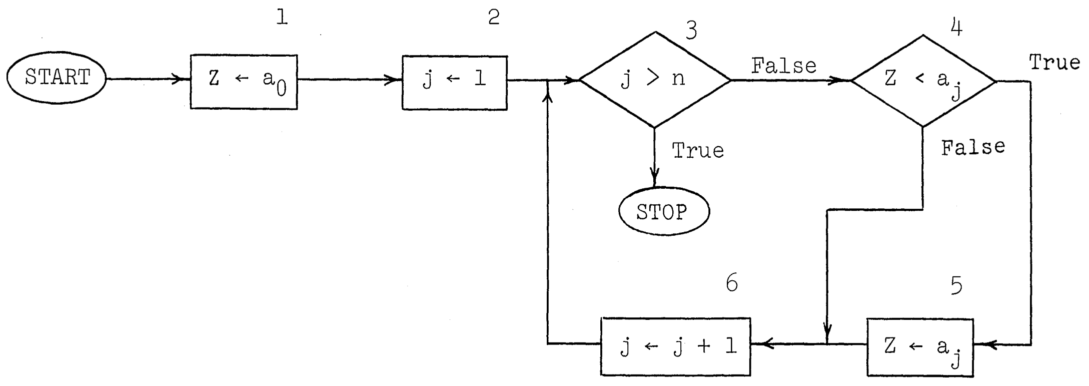
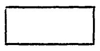
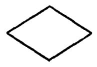
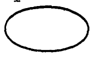


Figure 1-1

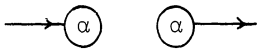
Note that the following conventions have been used here (or will be used later):

A. Computation occurs in rectangular boxes. 

B. Decisions occur in diamond-shaped boxes. 

C. The "terminals" of a procedure, such as entries and exits, are represented by the oval symbol: 

D. The direction of flow is represented by a "flow direction" line, which has an arrowhead wherever the "flow" enters another symbol.

E. If a flow direction line cannot conveniently be drawn between two points, we may use an internal (non-terminal) connector symbol, a circle containing a label: 

This algorithm, whether you study it as a sequence of statements or as a flow chart, exhibits an important concept, which occurs in a great many procedures; namely, it contains a loop. A loop has the following characteristic properties:

(a) It is repeated over and over until some condition is satisfied (occasionally this may be a very complex condition). In the example, the condition "$j > n$ is true".

(b) Before the first transit through the loop, i.e., the first "iteration", some variables are given initial values. In the example, $Z \leftarrow a_0$ and $j \leftarrow 1$.

(c) During each transit, usually just before it is completed, some variable is incremented, and the termination condition is tested again. In our example, j is increased by 1, and if $j \leq n$ is true, the "body" of the loop is computed again--employing the new value of j.

It is often convenient to take advantage of this standard structure of a loop, and use a special box in the flow chart called an "iteration box". The connections made by the box should make it easy to identify the "scope", i.e., the extent of the body of the loop. Moreover, the contents of the box itself should indicate the variable which is to be initialized, and later incremented, and the condition which will determine the number of iterations in the execution of the loop.

The iteration box as we shall use it, will have three parts or compartments as shown in Figure 1-2.

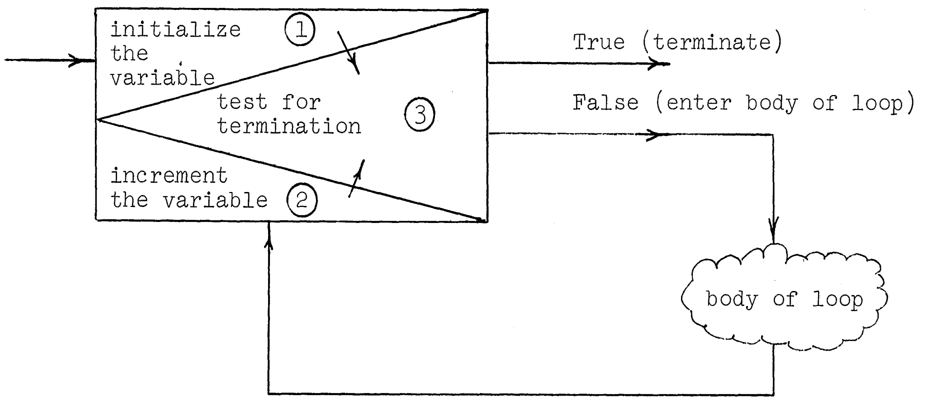


Figure 1-2. Main features of the iteration box

If we enter at compartment ① an initial value is given to the "control" or iteration variable. From this compartment the action moves to compartment ③ from whence either a True (T) or False (F) exit is then taken, depending on the condition that is tested there. Reentry to the iteration box is always at compartment ② where an incrementing (increase or decrease) of the control variable takes place. From compartment ② the action always shifts to compartment ③, from whence there is again a T or F exit.

If we study Figure 1-1, we see that Boxes 2, 6, and 3 serve the same functions as the three compartments of an iteration box as suggested in Figure 1-3.

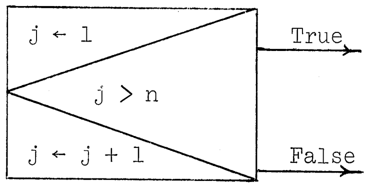


Figure 1-3

Figure 1-1 may therefore be streamlined in its structural appearance as seen in Figure 1-4 by taking advantage of the iteration box convention.

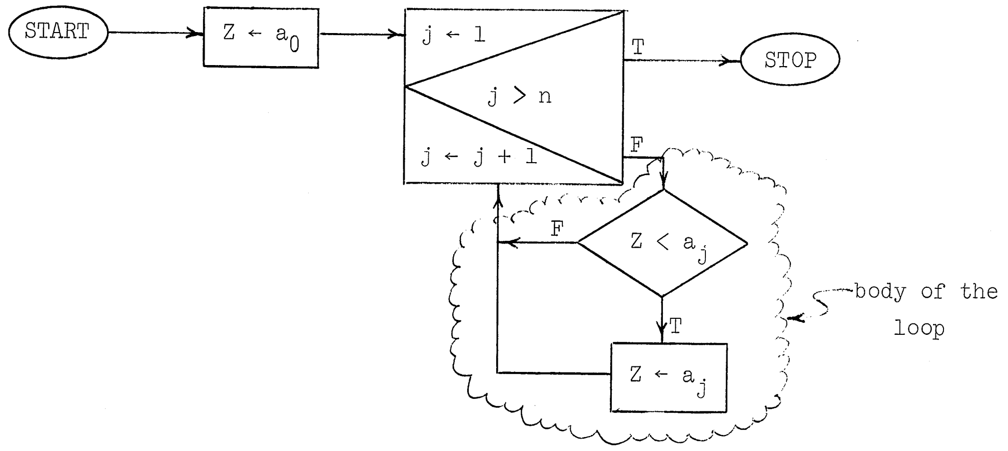


Figure 1-4

We consider one more illustration to suggest where and how the iteration box may be employed in diagramming a computational loop. Suppose we wish to repeat a certain computation starting with the variable α having the value 12, and increasing it by the amount 3 after each time through the body of the loop, terminating the repetition whenever $\alpha > 90$, or $|X + Y| \leq \epsilon$.

Figure 1-5 shows a suitable iteration box for this purpose.

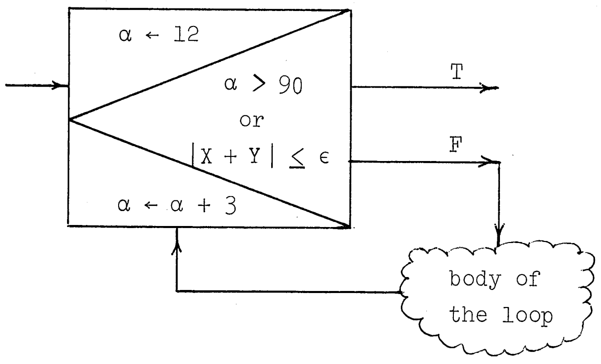


Figure 1-5. Another example

Communicating the algorithm to the computer. Once an algorithm has been stated as in a flow chart it should be presented to the computer directly in that form or as near to it as possible. For this, one needs a translator such as MAD which has the job of producing a translation from a flow diagram representation of the algorithm (or a direct equivalent of it) to a machine language representation of the same algorithm, i.e., into the basic code of the machine.

A flow chart for an algorithm may be re-expressed in the MAD language, which is then acceptable as input to the translator. Attention to the details of this conversion will be necessary, but aside from this step the user's work ends with the diagram itself. The full details of the MAD language are the subject of Chapter II. With this overview we wish to introduce some of the basic ideas or highlights that typify this "input language."

However, just to suggest what we mean by "details" of the language we show what the MAD language equivalent might be like for the iteration box of Figure 1-5:

```
THROUGH BACK, FOR ALPHA = 12, 3, ALPHA
        .G. 90 .OR. .ABS. (X+Y) .LE. EPSILN
```

Because MAD is a language built up out of a very limited character set (no more than 48 characters in all), certain symbols which we will use in our flow charts will not be available. Simple substitutions are needed such as ".G." for ">",

"="	in the absence of "←" ,
".G."	in the absence of ">" ,
".LE."	in the absence of "≤" ,
".ABS.(X+Y)"	in the absence of "|X+Y|" .

The term "THROUGH BACK" is to be read as "repeat the computation" in the body of the loop through and including the terminus of this body marked by the symbol "BACK".

To continue with out overview of the language, we shall next look at the MAD equivalent of the algorithm in Figure 1-4 which is presented in Figure 1-6.

```
          DIMENSION A(100)
          INTEGER J,N
          Z = A(0)
          THROUGH RETURN,FOR J=1,1,J.G.N
          WHENEVER Z .L. A(J), Z = A(J)
RETURN
          END OF PROGRAM
```

Figure 1-6

The DIMENSION statement assigns a block of storage in the computer that is large enough to handle a_0, a_1, ..., a_{100}, if necessary. The INTEGER statement declares the variables J and N to be integers. That is to say, only integer values will be assigned to these variables. By so declaring J and N, we can ensure that the arithmetic involving their values (subscript modification, counting, etc.) can be done more simply and efficiently, usually with less round-off error than would be the case if non-integer arithmetic were used.

Non-integer arithmetic is the arithmetic which the computer performs on representations of real numbers, i.e., on numbers that have both integral and fractional parts. Because of the finite precision with which such numbers can be represented in a computer, the reals are in general only approximated. Moreover the same finitude introduces round-off error that is characteristic of these non-integer arithmetic operations. The "approximated" reals are coded internally in a form called "floating point" representation. Unless otherwise declared, values for a variable in a MAD program are assumed to be represented in the floating point _mode_.

The WHENEVER statement in Figure 1-6 is to be interpreted in the sense: Whenever the following condition (in this case $Z < a_j$) is satisfied, do the specified action ($Z = a_j$), otherwise just go on.

It is interesting to ask just how complicated a condition can be used in making decisions. We have seen that such conditions may occur in iteration statements, and "WHENEVER" statements, etc., for the purpose of making binary (i.e., "yes" or "no") choices. An expression which can be labeled "True" or "False" is exactly what is needed here. Such expressions are called Boolean[†] expressions, and usually involve "and", "or", "not", and possibly other such words, connecting shorter expressions involving $<$, $\leq$, $=$, $\neq$, $>$, and $\geq$. For example, the following is a Boolean expression:

$$((x - 3)^3 < y \text{ and } i \leq j) \text{ or } x \geq 3$$

This will be "true" for some values of x, y, i, and j and "false" for others. It might then occur in statements such as:

```
WHENEVER ((X - 3) .P. 3 .L. Y .AND. I .LE. J) .OR. X .GE.3,
TRANSFER TO AGAIN
```

or in the iteration statement

```
THROUGH ALPHA, FOR BETA = 1, 1, ((X - 3) .P. 3 .L. Y .AND.
I .LE. J) .OR. X .GE. 3
```

where .P. denotes exponentiation (i.e., "to the power").

We now return to the MAD statements in Figure 1-5 to inspect the symbol "RETURN" which has been used twice. It is used in the THROUGH statement to identify the symbol used to designate or _label_ the terminating statement of the body of the loop. That is, each time the loop body is to be repeated, execution is to proceed down through and including the statement which has as its label the very same symbol, namely "RETURN". A MAD statement may be null or empty; in this example we have given an empty statement the label "RETURN" to clearly mark the terminus of the loop. Statement labels, like variables, are arbitrary symbols, though they must be unique. One chooses them in an essentially arbitrary fashion.

In our example problem on the largest of a set of numbers, we observe that no provision was made for obtaining the values of n, a_0, a_1, ..., a_n on which to perform our computation, nor was any provision made for producing an answer. Normally, each program would contain suitable input and output statements, such as will be described in Chapter II and illustrated in Chapter III. Let us assume instead that we are interested in making our little algorithm available for use in any other program, as a prepackaged "function", in the

[†]After the logician George Boole.

sense that, given n and $a_0, \ldots, a_n$, this function computes as its value the largest of $a_0, \ldots, a_n$. In this case we shall call our algorithm an EXTERNAL FUNCTION, and give it a name, say "MAX". It is an EXTERNAL FUNCTION because it will be written and translated externally with respect to the program which will later call upon it. The program for _defining_ MAX will now be written:

```
          EXTERNAL FUNCTION MAX.(N,A)
          INTEGER J, N
          Z = A(0)
          THROUGH BACK, FOR J = 1, 1, J .G. N
    BACK  WHENEVER Z .L. A(J), Z = A(J)
          FUNCTION RETURN Z
          END OF FUNCTION
```

The first statement specifies the inputs to the function to be N and A, the second statement indicates the point of entry, the FUNCTION RETURN statement specifies the value of Z as the desired value of the function, and the other statements are essentially as before. Any program using this function now need only call upon it by name, as in the statement:

```
LARGEQ = 1. + MAX.(6,Q)/3
```

Note that the set (in this use of MAX) whose largest element is desired is called Q, and N has the value 6. No DIMENSION statement is needed for A in the EXTERNAL FUNCTION definition program above, since A is there only as a "dummy variable" anyway. When used with the actual set Q, we would expect a DIMENSION statement for Q in the program that calls on MAX for a value.

For a second example consider the problem of solving the equation $f(x) = a^x + x = 0$ by Newton's method. The equation is to be solved repeatedly, each time taking a new value of a such that $a \geq 1$. The Newton formula is:

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)}$$

where the prime denotes the derivative with respect to x. This method involves the repeated evaluation of the (iterative) formula until $x_i + 1$ is a root, i.e., until $f(x_{i+1}) = 0$. Actually in the numerical solution of equations, where we deal with the finite-precision approximation of numbers, the loop termination condition becomes:

until $|f(x_{i+1})| < \epsilon$, where ϵ is a small positive number.

To evaluate the iterative formula the first time, it is necessary to have an initial approximation, x_0, to the desired root. The use of the index, i, as well as the initial subscript zero suggests that we will produce a sequence, $x_0, x_1, \ldots, x_n$, of approximations to the root. However, from the computational point of view we do not need all of these values simultaneously, since to evaluate the formula it is sufficient to know only the last value, x, produced. We can say

$$x \leftarrow x - \frac{f(x)}{f'(x)}$$

In other words each new value assigned to x will be the value of the expression (appearing on the right of the left arrow) which involves the current value of x. We are reminded of the fact that, in the actual MAD language statements produced for the computer, the "=" symbol is used to signify the left arrow. In MAD then, the use of the "=" is different from the usual use of the symbol in mathematics where it indicates a relation. When the "=" symbol is used as such a "replacement operator" the item on the left of the "=" is always the name of a variable. The variable may have a complicated subscript but nevertheless it is not an expression, but the name of a variable. The item on the right of the symbol is an expression involving one or more constants, variables, etc. The operation implied is simply that the value of the expression on the right becomes the value of the variable whose name appears on the left. This is referred to as assignment.

For our specific example, then, the evaluation of the iterative formula could be described as

$$x \leftarrow x - \frac{a^x + x}{\log_e a \cdot a^x + 1.0}$$

although we will, for convenience, break this into two statements. The entire computational procedure can be represented by the flow diagram in Figure 1-7.

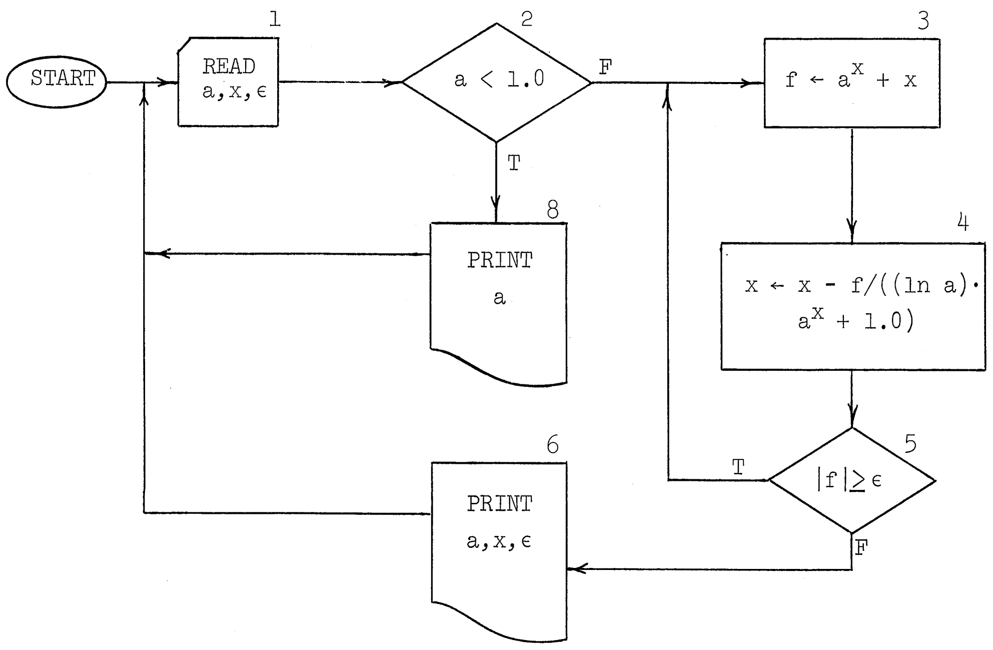


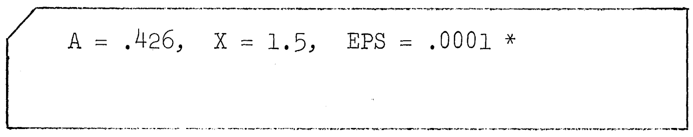
Figure 1-7. Finding the roots of $f(x) = a^x + x = 0$, using Newton's method.

The corresponding MAD statements are:

```
      ST  READ DATA
          WHENEVER A .L. 1.0, TRANSFER TO PRT
  REPEAT  F = A .P. X + X
          X = X - F/(ELOG.(A)*A.P.X + 1.0)
          WHENEVER .ABS.F .GE. EPS, TRANSFER TO REPEAT
          PRINT RESULTS A, X, EPS
          TRANSFER TO ST
     PRT  PRINT RESULTS A
          TRANSFER TO ST
          END OF PROGRAM
```

(1) The first statement, labeled ST, causes the data, which are values for a, x, and ϵ, to be read into computer storage.

The input for a typical data set for a, x, and ϵ can be supplied on a punch card as shown in Figure 1-8.



```
A = .426,  X = 1.5,  EPS = .0001 *
```

Figure 1-8. Typical data card which can be scanned as a result of executing a READ DATA statement.

When a READ DATA statement is executed, information on a data record (such as on the illustrated punch card) is scanned from left-to-right. Values read from the card are assigned to the appropriate variables, one after the other. The assignment process (input) terminates when an " * " character is encountered in the left-to-right scan.

(2) The second statement is a simple conditional which causes the statement labeled PRT to be the next one executed if $a < 1$. Otherwise, the next one in sequence (labeled REPEAT in this case) will be executed.

(3) The statement labeled REPEAT computes $a^x + x$ using the current value of x, and assigns this value to f, i.e., places the result of this computation in a storage location named F.

(4) The next statement divides the value F of the function by the derivative of the function: $(\log_e a \cdot a^x + 1.0)$, evaluated using the current value of x, subtracts this quotient from the current value of x and the resulting difference is stored as the current value of x. The name ELOG is the name used in MAD for the function $\log_e$ and the item in parentheses following this name indicates the variable whose natural logarithm is desired, i.e., the argument.

(5) The following statement is a simple conditional which causes the function and iterative formula to be evaluated again if $|f(x)| \geq \epsilon$. Otherwise, (i.e., $|f(x)| < \epsilon$) the next statement in sequence is executed.

(6) The PRINT RESULTS statement causes three numbers--the current values for a, x, and ϵ to be printed. When this statement is executed the names ("A", "X", and "EPS") will be printed along with the respective values as a simple and effective way of identifying each printed value.

(7) The TRANSFER statement that follows causes the statement labeled ST to be the next one performed.

(8) The statement labeled PRT is the one executed immediately after the first statement whenever $a < 1.0$. This statement causes the current value of a to be printed and labeled. (Here a must be < 1.0.) Thus, if a were 0.426 the phrase "A = 0.426000" would be printed.

(9) The final statement indicates the end of the statements and is the last statement executed when a definite termination to the problem is known. In our particular example, the computer never attempts to execute END OF PROGRAM because each time the TRANSFER statements return control to the READ DATA statement labeled ST. Instead, the computer would continue until it had exhausted the sets of given input values, a, x, ϵ.

The final example before the description of the language in Chapter II is a more general and more elaborate illustration of Newton's Method. Here, we consider the solution of the equation $x^3 + ax^2 + bx + c = 0$, starting with some input value for x_0, using the iterative formula:

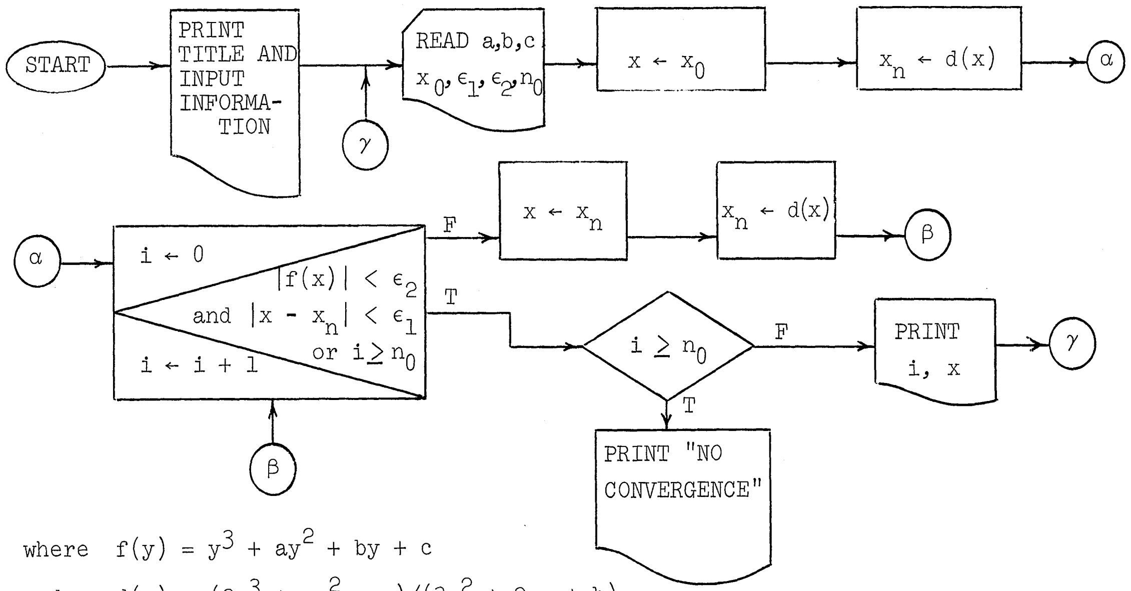
$$x_{i+1} = x_i - \frac{x_i^3 + ax_i^2 + bx_i + c}{3x_i^2 + 2ax_i + b}$$

$$= \frac{2x_i^3 + ax_i^2 - c}{3x_i^2 + 2ax_i + b}$$

which is obtained from the standard Newton's Method formula. We shall use the following condition as the criterion for halting the repeated evaluation of this formula:

$$(|x_{i+1} - x_i| < \epsilon_1 \text{ and } |f(x_i)| < \epsilon_2) \text{ or } i \geq n_0$$

where n_0 is some given upper limit to the number of iterations which we can tolerate. We are requiring that at x_i the value of the function $f(x_i)$ will lie between the upper and lower bounds ϵ_2 and $-\epsilon_2$. Also, the distance between x_{i+1} and x_i should be less than ϵ_1. The flow chart for a suitable algorithm is given in Figure 1-9. Observe that we are never concerned with more than two consecutive approximations to the root-- say x and x_n.



where $f(y) = y^3 + ay^2 + by + c$

and $d(z) = (2z^3 + az^2 - c)/(3z^2 + 2az + b)$

Figure 1-9. Flow chart for finding roots of a cubic equation by Newton's Method.

A corresponding MAD program is shown below:

```
              PRINT COMMENT $1SOLUTION OF CUBIC EQUATIONS$
GAMMA         READ AND PRINT DATA
             R  DATA CONSISTS OF NEW VALUES FOR
             R  A, B, C, XZERO, EPS1, EPS2, AND NZERO.
              X = XZERO
              NEXTX = D.(X)
              INTERNAL FUNCTION F.(Y) = Y .P.3 + A * Y.P.2
             1     +B*Y+C
              INTERNAL FUNCTION D.(Z) = (2.*Z.P.3 + A*Z.P.2
             1     -C)/(3.*Z .P.2 +2.*A*Z +B)
ALPHA         THROUGH BETA, FOR I = 0, 1, (.ABS.(NEXTX-X) .L.
             1     EPS1 .AND. .ABS. F.(X) .L. EPS2) .OR. I .GE.
             2     NZERO
              X = NEXTX
BETA          NEXTX = D.(X)
              WHENEVER I .GE. NZERO, PRINT COMMENT $ NO CONVERGENCE$
              PRINT COMMENT $ NO. OF ITERATIONS AND THE VALUE OF X$
              PRINT RESULTS I, X
              TRANSFER TO GAMMA
              INTEGER I, NZERO
              END OF PROGRAM
```

The first statement causes the program to print a comment or title consisting of the phrase

"SOLUTION OF CUBIC EQUATION"

which, except for the digit 1, is the string of characters between the dollar signs (MAD equivalent of quotation marks). The first character following the opening quote mark is used as a code to govern the positioning of the paper for printing. A code of "1" positions the paper at the top of a new page before printing. (A code of "□" (blank space) causes the paper to be advanced one line before printing.)

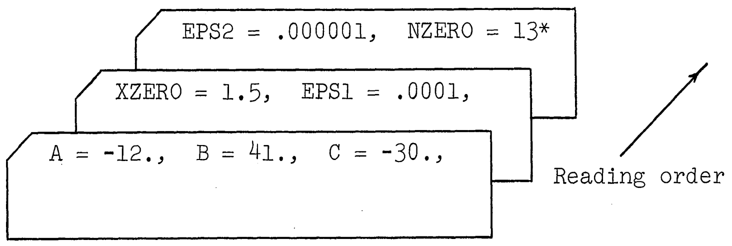

EPS2 = .000001, NZERO = 13*
XZERO = 1.5, EPS1 = .0001,
A = -12., B = 41., C = -30.,
Reading order

Figure 1-10. A set of data cards examined and copied as a result of executing the READ AND PRINT DATA statement.

The statement labeled GAMMA would cause the program to read from one or more data cards the numbers a, b, c, x_0, ϵ_1, ϵ_2, and n_0 and to print an exact copy of the information on the cards that are read. Cards are read until the terminating asterisk character is encountered as suggested in Figure 1-10.

Two functions are defined in this program, one for f(y) and the other for d(z). Each is designated as an INTERNAL FUNCTION.

These function definitions are a part of, but parenthetical to the main program. Such definitions may be placed anywhere in the program in which they are embedded.

The statement following that which is labeled BETA illustrates a conditional output statement. If the iteration is terminated because $i \geq n_0$, the comment NO CONVERGENCE is printed before the values of I and X are printed, otherwise that remark is not printed. The final transfer to GAMMA causes the program to start over with a new set of data, if additional data is present; otherwise, the computation is automatically terminated.

Chapter II

DESCRIPTION OF THE LANGUAGE

"There is a pleasure sure in being mad, which none but madmen know."

Dryden: The Spanish Friar

1. Constants, Variables, Statement labels, Functions, Operations, and Expressions

1.1 Constants

There are five classes of constants. Integer, floating point, alphabetic, Boolean, and octal.

1.1.1 Integer Constants

Integer constants must be less than or equal to 34359738367 (i.e., $2^{35} - 1$) in absolute value. (However, see Section 1.10 if integer to floating-point conversion is involved.) The decimal point is assumed to be immediately to the right of the rightmost digit, but is always omitted. Integers may be positive or negative, and while the "+" sign may be omitted, the "-" sign must be present if the number is negative (e.g., 2, -2, 0, +0, -0, 100 are all integers). Leading (but not following) zeros may be omitted (e.g., 5 and 005 represent the same integer, but 3 and 300 do not). An integer may not contain characters other than digits and an optional sign; e.g., commas are not allowed. Thus, "1,500" is an illegal form for "1500".

1.1.2 Floating Point Constants

Floating point constants may be written with or without exponents. If written without an exponent, the constant contains a decimal point ".", which must be written, but which may appear anywhere in the number. Thus, 0., 1.5, -0.05, +100.0, .1 and -4. are all floating point constants.

If the number is written with an exponent, it may be written with or without a decimal point, followed by the letter "E", followed by the exponent of the power of 10 that multiplies the number. (If the decimal point is omitted, it is assumed to be immediately to the left of the letter "E".) The exponent m consists of one or two digits preceded by a sign (although a "+" sign may be omitted), and must satisfy the condition $-38 \leq m \leq 38$. More specifically, the value of the number F must be 0 or else satisfy the condition

$$.1469368 \times 10^{-38} \leq |F| \leq .1701412 \times 10^{39}$$

Examples of floating point constants with exponent are: .05E -2(=$.05 \times 10^{-2}$), -.05E2(= $-.05 \times 10^{2}$), 5E02(= 5.0×10^{2}), 5.E2(= 5.0×10^{2}).

Negative floating point constants must be preceded by a "-" sign. Positive constants may be preceded by a "+" sign.

1.1.3 Alphabetic Constants

An alphabetic constant consists of from one to six admissible characters preceded and followed by the character " $ ". The admissible characters include all letters of the alphabet, the digits 0 through 9, the special characters +, - (minus sign), ' (prime), *, /, =,), (, ., the comma " , " and the blank space, which is to be represented here occasionally (but not punched on input cards), as the character "□ ". Thus the following are alphabetic constants: $ABCD$, $TO□BE$, $DEC.□4$, $5+3=8$. Note that blank spaces, while ignored elsewhere in the language, count as characters in alphabetic constants.

Within any alphabetic constant (i.e., between two $-signs), a pair of consecutive $-signs will be treated as the single character, $-sign. Any blanks between such a pair are completely ignored and are not counted as characters. Examples:

$A$$B$	represents the string	A$B
A B	also represents the string	A$B
$$$.56$	represents the string	$.56

with 3, 3, and 2 trailing blanks, respectively.

Each character of an alphabetic constant is stored in its internal representation as a unique integer code. An alphabetic constant, therefore, appears internally as an integer that results from the concatenation of the integer codes of six characters. Any alphabetic constant containing fewer than six characters will be extended to six characters by adding blanks on the right; thus $ABCD$ will appear internally as ABCD□□ .

1.1.4 Boolean Constants

There are two Boolean constants--"true," which is written 1B, and "false," which is written 0B.

1.1.5 Octal Constants

These constants are written as twelve digit octal integers followed by the letter K, except that leading zeros may be omitted. If one or more decimal digits follow the letter K, this is interpreted as an octal scale factor. Thus 127K2 would be the octal integer 000000012700, and 1K10 would produce the octal number 010000000000. Octal constants are assigned integer mode.

1.1.6 Constants of other Modes

Any constant (integer, floating-point, Boolean, alphabetic, or octal) may be declared to be of mode other than its normally assigned mode by following the constant by the letter M and the digit code for that mode. The standard digit codes are:

0	Floating-point
1	Integer
2	Boolean
3	Function Name
4	Statement Label

Modes 5, 6, and 7, which may be defined (see Appendix B), may also be used. The constant is converted <u>as usual</u>, but then is assigned the indicated mode, if the letter M is used.

Thus, if some new mode numbered 6 were defined for certain integers, then the appearance of the constant 32M6 would assign mode 6 to the decimal integer 32. The appearance of 32KM6 would assign mode 6 to the octal constant 32, and the appearance of ABM6 would assign mode 6 to the constant AB. Similarly, the appearance of 32.1E-1M6 would assign mode 6 to the number 3.21, which appears in storage in the usual floating-point form. (The mode digit may also be a parameter--see Section 3.8.)

1.2 Variables (integer, floating-point or Boolean)

The name of an integer, floating-point or Boolean variable consists of one to six letters or digits, the first of which must be a letter. If the variable is defined as an n-dimensional array variable (see Section 3.3) then the name of an element of the array consists of the variable name, (i.e., one to six letters or digits, starting with a letter), followed by the appropriate subscripts separated by commas and enclosed in parentheses. Thus the following are "single variables": X51, ALPHA6, LAMBDA, GROSS, while the following are elements of arrays: BETA(C1, C2, 6), X15(Y,Z1), J(6), J(Z1 + 5*Z2, 5). (See Section 1.11 for the description of subscripts.) Parentheses enclosing subscripts may not be omitted.

1.3 Statement Labels

A statement may be labeled or unlabeled. Labels are used to refer to a statement by other statements. A statement label consists of from one to six letters or digits, the first of which must be a letter, e.g., IN or BACK. A statement label may be an element of a statement label vector, in which case the vector name is followed by a constant integer subscript enclosed in parentheses, e.g., S(2) or LBL(3). A statement label appears in the label field (i.e., Columns 1-10) of the statement it identifies. When a statement extends to additional cards (i.e., cards identified by a digit punched in Column 11) the statement label need not be punched on the additional cards. (See also section 4.)

1.3.1 Statement label constants

A statement label appearing in the label field of a statement will be called a constant of statement label mode.

1.3.2 Statement label variables

A statement label which does not appear in the label field is a variable of statement label mode, provided it is so declared in a mode declaration (See Section 3.2).

1.4 Functions

The name of a function consists of one to six letters or digits, but the name must be followed by a period (.) so that the translator can recognize it as a function name. The first character of a function name must be a letter. If the function is single-valued, then the value of the function is represented by following the function name with the proper number of arguments (see Section 3.10 for the definition of function) separated by commas and enclosed in parentheses. Thus, ADD51., COS., POLY., and FUNCT3. are function names, while ADD51.(X,Y3,ADD.), POLY.(N,VJ,7) and COS.(X) are function values. A function name given explicitly in this form will be called a function name constant. (See also Sec. 2.8.) For additional explanation regarding the distinction between function name constants and function name variables, see Section 1.13. (See also section 4.)

1.5 Arithmetic Operations

The following arithmetic operations are available:

(a) Addition, written as "+", e.g., Z5 + D.

(b) Subtraction, written as "-", e.g., Z5 - D.

(c) Multiplication, written as "*", e.g., Z5*D. (Note that the "*" may not be omitted. It is illegal to write Z5D, since it would be impossible to distinguish such a product from the variable Z5D.)

(d) Division, written as "/"; e.g., Z5/D. If both Z5 and D are integers, the result is again an integer; e.g., the "fractional part" of the true quotient is truncated (not rounded). For example, if Z5 = 8, and D = 3, then Z5/D will have the value 2.

(e) Exponentiation, written as ".P.", e.g., Z5.P.D, and meaning $(Z5)^D$; i.e., Z5 raised to the power D.

(f) Absolute value, written ".ABS."; e.g., .ABS.Z5, meaning $|Z5|$, the absolute value of Z5, and .ABS.(Z5-D) meaning $|Z5 - D|$.

(g) Negation, written as "-"; e.g., -ALOHA, meaning the "negative of ALOHA." Thus -X.P.-.5 means $-(X^{-.5})$, the negative of the reciprocal of the square root of X.

(h) Full word bitwise negation, written .N.I, where I is an integer expression, and meaning the operation of negating each binary digit in the binary representation of the value of I. The result is again an integer.

Example: Let I = 6 which is represented internally as

```
(normally interpreted as the
 "sign bit": plus, in this case)
        \_____
              000...000110
                36 bits
```

Then .N.I would yield the value:

```
("sign bit": minus, in this case)
        \_____
              111...111001
```

(which can be interpreted as a large negative integer in this case).

(i) Full word bitwise logical operations and and or, written .A., .V., and .EV., respectively, meaning the bitwise and, or, and exclusive or of the full binary integer values of the operands. The result is again an integer.

Example: Let I = -17 which is represented internally as

```
1000...00010001
   36 bits
```

and let J = 9 which is represented internally as

```
0000...00001001
   36 bits
```

Then I .A. J would yield the value

$$\underbrace{0000\ldots00000001}_{36\text{ bits}}$$

which is the integer 1,

I .V. J would yield the value

$$\underbrace{1000\ldots00011001}_{36\text{ bits}}$$

which is the integer -25,

and I .EV. J would yield the value

$$\underbrace{1000\ldots00011000}_{36\text{ bits}}$$

which is the integer -24.

(j) Full word integer shifts, written .LS. and .RS., respectively; e.g., I .LS. J and I .RS. J, where I and J are integer expressions (see Section 1.10). I .LS. J means the value of I shifted <u>left</u> J binary places. The entire computer word, all 36 bits, is shifted. The sign of J is ignored; the absolute value of J is always used. Similarly with .RS.. Digits shifted off either end of the computer word are lost. Created blank positions are filled with zeros. The result is always again an integer.

Example: Let I = -30 which is represented internally as

$$\underbrace{100\ldots0011110}_{36\text{ bits}}$$

and let J = 4.

Then I .LS. J would yield a value which is represented as

$$\underbrace{000\ldots111100000}_{36\text{ bits}}$$

and I .RS. J would yield a value which is represented as

$$\underbrace{00001000\ldots0001}_{36\text{ bits}}$$

1.6 <u>Arithmetic Expressions</u>

Arithmetic expressions are defined inductively as follows:

(a) All integer, floating point, alphabetic and octal constants, integer and floating point individual variables, subscripted integer and floating point array variables, and integer and floating point values of functions are arithmetic expressions. A function value used in an expression must have at least one argument, even if the function is defined without dummy variables. (See Sections 2.8 and 3.10 for more information on functions.)

(b) If E and F are any arithmetic expressions, and I and J integer expressions, then the following are also arithmetic expressions: +E, -E, .ABS.E, E + F, E - F, E*F, E/F, E .P. F, (E), .N. I, I .A. J, I .V. J, I .EV. J, I .LS. J, and I .RS. J

(c) The only arithmetic expressions are those arising in (a) and (b).

1.7 Boolean Operations

The following Boolean, or logical, operations are available in the language (where M and P are Boolean expressions):

(a) .NOT.M has the value 1B if and only if M has the value 0B.

(b) (M) has the same value as M.

(c) M.OR.P has the value 0B if and only if both M and P have the value 0B.

(d) M.AND.P has the value 1B if and only if both M and P have the value 1B.

(e) M.THEN.P has the value 0B if and only if M has the value 1B and P has the value 0B.

(f) M.EXOR.P has the value 1B if and only if either M or P has the value 1B, but not both.

(g) M.EQV.P has the value 1B if and only if M and P have the same values.

Thus .NOT., .OR., .AND., .THEN., .EXOR., and .EQV. correspond to the usual logical operations, $\sim$, $\vee$, $\wedge$, $\supset$, "exclusive or," and $\equiv$.

1.8 Boolean Expressions

Boolean expressions are defined inductively as follows:

(a) Boolean constants, individual Boolean variables, subscripted Boolean array variables and Boolean-valued functions are Boolean expressions. (See Sections 1.1.4 and 3.2.)

(b) If H and F are arithmetic expressions: then H.L.F., H.LE.F, H.E.F, H.NE.F, H.G.F, H.GE.F, are Boolean expressions, where the meanings are $H < F$, $H \leq F$, $H = F$, $H \neq F$, $H > F$, and $H \geq F$, respectively.

(c) If M and P are Boolean expressions, then the following are also Boolean expressions: .NOT.M, (M), M.OR.P, M.AND.P, M.THEN.P, M.EXOR.P, and M.EQV.P.

(d) The only Boolean expressions are those that arise in (a), (b), and (c).

Examples of Boolean expressions are: (X .G. 3 .AND. Y .LE. 2) .OR. (GAMMA .L. EPSILN), (.ABS. (X1 - X2)/X1 .LE. EPSILN) .AND. (F.(X1) .L. EPSILN), and ((P .AND. Q) .THEN.Q) .EQV. (P .OR. .NOT. P), where P and Q are Boolean variables.

Boolean expressions of types (a) and (b) above are referred to as "atomic Boolean expressions." Object programs produced by the translator will skip the evaluation of the remaining terms of a disjunction (an "or" expression) as soon as one term has the value 1B, and a similar statement holds for conjunctions ("and" expressions). In order to obtain the maximum benefit from this skipping behavior, it is necessary to understand that the atomic Boolean expressions in a complex Boolean expression are evaluated from right to left, and the one most likely to be "true" in a disjunction, and the one most likely to be "false" in a conjunction, should be placed as far toward the right end of the expression as possible.

Thus, if one were testing for values of X between 0 and 2 and between 5 and 6, one might write

```
WHENEVER 0 .L. X .AND.X .L. 2 .OR. 5 .L. X .AND.X .L. 6
```

If one knew that for the data expected, the values of X would occur most often between +1 and 2, one would do better to write the above as follows:

```
WHENEVER X .L. 6 .AND. 5 .L. X .OR. 0 .L. X .AND. X .L. 2
```

1.9 Parentheses Conventions (Precedence of Operators)

Parentheses are used in the same way as in ordinary algebra and logic to specify the order of the computation. Also, certain conventions are used to allow deletion of parentheses. The conventions used here are the same as in ordinary algebra and logic, namely: Parentheses may be omitted, subject to the rules (A) and (B) below, but redundant parentheses are allowed.

(A) Within any expression the sequence of computation, unless otherwise indicated by parentheses, is:

.ABS., + (as unary operations), .N., .LS., .RS.
.A.
.V., .EV.
.P.
- (as a unary operator)
*, /
+, - (as binary operations, i.e., addition and subtraction)
.E., .NE., .G., .GE., .L., .LE.
.NOT.
.AND.
.OR., .EXOR.
.THEN.
.EQV.
, (as used to separate function arguments)
=

Two other operations occur by implication only; viz., the function call (see Sec. 2.8) and subscription (see Sec. 1.11). Thus the call for the function: SIN. (X + Y) implies that after the sum X + Y is computed, the operation of actually calling the function SIN must be performed. Similarly, the array element A(I + 3 × J) is determined by first evaluating the subscript I + 3 × J and then performing the implied subscription operation. These two implicit operations do not appear in the precedence list above, but may be considered to be together on a level just above .ABS., .N., etc.

Examples:

(1) .ABS.(B - C) means $|B - C|$, while .ABS.B -C means $|B| - C$.

(2) - B + C means (-B) + (C), while -(B + C) means the negation of the sum.

(3) B.P. - X + Y means $B^{-X} + Y$, while B.P.(-X + Y) means B^{-X+Y}.

(4) K2/Z - 3 means (K2/Z) - 3, while K2/(Z - 3) implies that Z - 3 is the denominator.

(5) A * B + C means (A * B) + C.

(6) A.P.3/J means $(A^3)/J$.

(7) X.L. Y + 3 means (X) .L. (Y + 3).

(8) P.AND..NOT.P .EQV.Q means (P.AND.(.NOT.P)).EQV.Q.

(9) Z = X + Y/QA means $Z \leftarrow (X + (Y/QA))$

(10) A = -B.P.X means $A \leftarrow -(B^X)$.

(B) Within an expression operations appearing on the same line of the list in (A) are to be performed from left to right, unless otherwise indicated by parentheses.

Examples:

(1) A + B - C + D - E means (((A + B) - C) + D) - E.

(2) X/Z * Y/R * S means (((X/Z) * Y)/R) * S.

1.10 Mode of Expressions

The kind of arithmetic performed on a constant, variable or function value is determined by its mode. There are five modes in MAD: floating point, integer, Boolean, statement label, and function name. Floating point, integer, and Boolean constants were described in Section 1.1. Alphabetic constants and octal constants are assumed to be of integer mode. Section 3.2 describes how the modes of variables and functions are specified.

If an expression consists entirely of one constant, one variable, or one functional value, the mode is that of the constant, variable, or functional value itself. If the expression is a compound expression; i.e., consists of two or more subexpressions joined by logical or arithmetic operations, the following rule applies:

If an expression is a Boolean expression as defined in Section 1.8, then its mode is Boolean. An arithmetic expression is considered to be in the floating point mode if any operand of any arithmetic operation in the expression is in the floating point mode. If all operands are integer (or alphabetic or octal), then the expression is considered to be in the integer mode. In this determination arguments, though not values, of functions are ignored.

Thus, if Y, Z, and W are floating point variables, while the function GCD. and the variables I and J are in the integer mode, then the expressions

```
Y + GCD.(I,J)
Y + Z - I
I + 1.
GCD.(I, J)/Z
```

are all floating point expressions while the expressions

```
I + GCD.(I, J)
(I + J)/3
I + 1
GCD.(I,J)/I
```

are all integer expressions.

If an expression has subexpressions of different modes, a conversion may be necessary before some of the operations can be performed. Thus, in the expression

$$Y + 3$$

if Y is in the floating point mode it cannot be added directly to the integer 3. But for one precaution the user need not be concerned with this since the instructions necessary for the conversion of the integer to floating point form before adding are automatically inserted by the translator during the translation process. The precaution is that if the integer being converted is greater than 134,217,727 (i.e., $2^{27} - 1$), then an improper conversion will take place.

In some cases, however, the user must understand the sequence in which the conversions will be made. Consider the expression

$$(Y + 7/3) + (I * J/K)$$

where Y is in the floating point mode, and I, J, and K are in the integer mode. According to the parenthesizing conventions, the computation will proceed in the following order (where the T's are temporary locations):

$$T_1 \leftarrow I \times J$$
$$T_2 \leftarrow T_1/K$$
$$T_3 \leftarrow 7/3$$
$$T_4 \leftarrow Y + T_3$$
$$T_5 \leftarrow T_4 + T_2$$

and T_5 will be the value of the expression.

Now, since both I and J are integers, the first multiplication will be integer multiplication, and T_1 will be an integer. Since the next step involves two integers, it will be integer division, and, if K happens to have a larger value than T_1 the quotient is 0. Similarly, T_3 will have the value 2 because of the division of two integers. In the computation of T_4, however, we have "mixed modes," since Y is floating point and T_3 is integer. Here T_3 will be automatically converted to floating point before adding. Likewise, in the next step, the integer T_2 will be converted to floating point before adding to the floating point number T_4.

In other words, although the mode of the expression is floating point because of the presence of the floating point variable Y, some of the computation (until Y is involved) is performed in integer arithmetic, and this may occasionally cause the final value to be different from the value one might expect from a different analysis.

In the example above, the divisions would be performed in the floating point mode if the expression were written:

$$(Y + 7./3) + (I * J)/(K + 0.)$$

Of course, many times the expression will be written as originally stated just to achieve the "truncation" effect.

A single constant, variable, or function value of statement label (function name) mode is an expression of statement label (function name) mode. There are no other expressions of statement label (function name) mode. If A and B are both statement label (function name) expressions then A.E.B and A.NE.B are Boolean expressions except when A or B are elements of a vector which has been preset by a VECTOR VALUES statement (see Section 3.7). In this exceptional case A.E.B and A.NE.B are not expressions.

1.11 Subscript Expressions

Any arithmetic expression may be used as a subscript expression for an element of a linear or two-dimensional array. If the value of the expression is in the floating point mode (see Section 1.10), it is truncated to integer form before being used as a subscript.

(Since any mode conversion takes extra storage and time, one usually tries to avoid the use of floating point subscripts. To help in this, the MAD translator now reports to the programmer the first occurrence of such a conversion, as a warning that this has perhaps inadvertently been incorrectly written. This is not considered an error, and translation continues.)

The expressions for subscript elements of an array whose dimension is three or greater must be of integer mode. Moreover, for arrays of dimension three or greater, the use of subscripts having other than integer mode will not be caught as an error. Subscript expressions may contain variables with subscripts, etc.

Examples of subscripted variables: J(3), K10(Z + 5 * XY/T), MP(A,B + 6 * J, I * 6/TDX), T(I(J)), MA(K(Z + 5) + T(1) + 6).

1.12 Block Notation

Input-output lists (see Section 2.14), VECTOR VALUES statements (see Section 3.7), and some subroutines (see Section 3, Chapter IV) allow the use of block notation. This has the form

A,...,B or A...B

which is usually interpreted as the entire region from A to B, inclusive. The most common use is in terms of a single array; e.g., A(1),...,A(N), B(I,J),...,B(M,N). These would be interpreted as the regions: A(1), A(2), A(3), ..., A(N) and B(I,J), B(I,J+1), B(I,J+2), ..., B(I+1,J), B(I+1,J+1), ..., B(M,N). If the commas are omitted, both must be omitted.

1.13 Function Name Constants and Variables

(a) The notation for function evaluations, i.e., function references, always requires a period after the name of the function.

(b) Function name constants are the names which designate entries in actual definitions, e.g., SIN., COS., F. --assuming F is defined as either an internal or external function. Function name constants are never subscripted and never appear without a period; they may not stand alone on the left side of an assignment statement.

(c) Function name variables, i.e., variables of function name mode where the mode is either declared or implicit from a VECTOR VALUES declaration, should not have a period when used as variables. Use on the left side of an assignment statement or as an argument of a function are examples. The following two restrictions apply to the use of function name variables when they are used as function names in function evaluations.

(1) Single function name variables, i.e., variables not normally considered to be an element of an array, must be written with a zero subscript preceding the period. For example, if G is a single function name variable its use in a function evaluation would appear G(0).(A,B) where A and B are the arguments. As with any single variable used only with a zero subscript, it does not need to be dimensioned.

(2) Such function evaluations may not be embedded in a larger expression. They may appear by themselves or as the right side of an assignment statement.

Thus if G and H are function name <u>variables</u>, then

```
        H(0).(A,B,T)
or      T = G(0).(A,B)
```

would be acceptable statements, but not

```
T = G(0).(A,B) + C*D
```

No checking of mode (or conversion) will be made as a result of such an assignment.

2. <u>Statements</u> (Executable) (See Appendix A for admissible abbreviations)

2.1 <u>Assignment Statement</u>

This statement has the form illustrated by

```
ALPHA = Y + Z + F.(X, Y, Z)
```

That is, the left side of the "=" sign consists of the name of a variable (either an individual variable or a subscripted array variable), and the right side consists of any expression of the same mode. The only exceptions to this mode requirement are the cases: (1) If the variable on the left is of integer mode then the value of a floating point expression on the right will be converted to integer mode. (2) If the variable on the left is of floating point mode then the value of an integer expression on the right will be converted to floating point mode.

The assignment statement (sometimes referred to as the <u>substitution statement</u>) is to be interpreted as: "(1) Compute the value of the expression on the right, (2) convert it, if necessary, to the mode of the variable on the left of the "=" sign, and (3) give the variable on the left the value which results from Steps (1) and (2)." (See Section 1.10 for mode of expressions.)

Thus, if Y is a floating point variable, then the statement

```
Y = 1
```

will cause the integer 1 to be converted to floating point and then stored in the location called "Y"; i.e., Y will now have the value 1. (as a floating point number). If the statement were written

```
Y = 1.
```

then the floating point number 1. would be stored in the location "Y"; i.e., Y would again have the floating point value 1., but in this case the conversion of the integer is unnecessary, thus speeding up the computation.

When a floating point number is to be converted to an integer, it is first expressed as a number with both an integer part and a fractional part, and then the fractional part is truncated. Thus, the following floating point numbers:

3E5, .3E0, .34568127E2, -.345681E10

would convert to the following integers, respectively:

300000, 0, 34, -3456810000.

Examples of assignment statements in other modes are:

(1) Assuming B and C have been declared Boolean

C = B .AND. D .L. 10.

(2) Assuming BILL(3) is a statement label constant and HARRY is a statement label variable

HARRY = BILL(3)

(3) Assuming FUN is a function name variable

FUN = COS.

2.2 Transfer Statement

This statement has the form: TRANSFER TO $\mathcal{S}$

Here $\mathcal{S}$ may be any statement label or any expression in statement label mode which labels an executable statement. Execution of this statement causes the computation to continue from the statement whose label is the value of $\mathcal{S}$. Examples: TRANSFER TO SUMX, TRANSFER TO SWTCH (K + 2) (if K = 4 then the value of SWTCH(K + 2) is SWTCH(6)).

2.3 Conditional Statements

There are two types of conditional statements.

(a) Simple Conditional

WHENEVER B,Q

Here B is a Boolean expression and Q any executable statement except the following: END OF PROGRAM, another conditional, iteration, and function entry. Q may, however, be an iterated statement[†]. If at the time of execution of this statement, the expression B has the value 1B, i.e., _true_, the statement Q is executed. If, however, B has the value 0B, i.e., _false_, then Q is skipped and computation continues from the next statement in sequence. The comma in this statement, as in other statements containing punctuation, must be written.

Examples: WHENEVER XM.LE.1, TRANSFER TO END
WHENEVER I.GE.N.AND.J.NE.I-1, I = 0

(b) Compound Conditional

$\mathcal{S}_1$ WHENEVER B_1

$$\left.\begin{matrix}\cdots\\\cdots\\\cdots\end{matrix}\right\}\theta_1$$

[†](See sec. 2.17.2).

$$\begin{array}{ll}
\mathscr{S}_2 & \text{OR WHENEVER } B_2 \\
 & \left.\begin{array}{l}\ldots\\ \ldots\end{array}\right\} \theta_2 \\
 & \ldots \\
 & \ldots \\
\mathscr{S}_k & \text{OR WHENEVER } B_k \\
 & \left.\begin{array}{l}\ldots\\ \ldots\end{array}\right\} \theta_k \\
\mathscr{S}_{k+1} & \text{END OF CONDITIONAL}
\end{array}$$

Often the last condition B_k expressed is one for which the condition is always true. This may be expressed by the statement

OR WHENEVER 1B

or alternately the statement

OTHERWISE

The $\mathscr{S}_i$ are statement labels which need not be used unless desired. k may equal one (if no "OR WHENEVER ..." statements are used). Here $B_1, \ldots, B_k$ are Boolean expressions, and the execution of this block of statements is as follows. θ_j is a sequence (possibly empty) of statements. These may include conditional statements, of either form.

Each B_i is tested in turn, starting with B_1. If B_1 has the value 0B, then B_2 is tested, etc. As soon as some expression, say B_j, has the value 1B, then the statements between (but not including) $\mathscr{S}_j$ and $\mathscr{S}_{j+1}$ (i.e., θ_j) are executed. At this point the execution of the entire block is considered ended, and computation continues from the first statement after the END OF CONDITIONAL statement which, in this illustration, we have chosen to label $\mathscr{S}_{k+1}$. In other words, no more than one of the alternative computations is performed; e.g., that one which immediately follows the first expression B_i which has the value 1B. If none of the B_i has the value 1B, none of the computation in the scope of these statements is performed.

Example: The evaluation of the function whose graph is shown in Figure 2-1

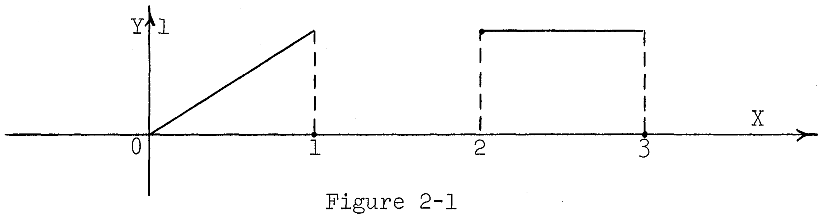


Figure 2-1

might be given by the section of the program:

```
WHENEVER X .LE. 0. .OR. 1..LE. X .AND. X .L. 2. .OR. X .GE. 3
    Y = 0
OR WHENEVER 0. .LE. X .AND. X.L.1.
    Y = X
OR WHENEVER 2. .LE. X .AND. X .L. 3.
    Y = 1.
END OF CONDITIONAL
```

This section of program could be rewritten in another way.

```
WHENEVER 0..LE. X .AND. X .L.  1.
     Y = X
OR WHENEVER 2..LE. X .AND. X .L. 3.
     Y = 1.
OTHERWISE
     Y = 0.
END OF CONDITIONAL
```

The indentation of the assignment statements between the conditional statements is not required but contributes to the readability.

2.4 CONTINUE Statement

This statement has the simple form:

```
CONTINUE
```

When labeled, it serves as a junction point in the program, but causes no computation to be performed by its presence. It is merely a dummy or "do-nothing" statement. Its chief use is to indicate the scope of an iteration statement (see example in Section 2.5). A statement which is blank in Columns 11 - 72, but which has a statement label, is treated as a CONTINUE statement.

2.5 Iteration Statement (THROUGH Statement)

Figure 2-2 is a program segment which illustrates the use of this statement.

```
     .
     .
     .
     K = 1
     L = 1
     A = D(1,1)
     THROUGH ST1, FOR I = 1,1,I .G. 10
     THROUGH ST1, FOR J = 1,1,J .G. 10
     WHENEVER A .GE. D(I,J), TRANSFER TO ST1
     K = I
     L = J
ST1  CONTINUE
     .
     .
     .
```

Figure 2-2. A program segment to determine the largest element (A) in a 10 row by 10 column array called D and to record its location (K,L). The search proceeds left to right, row by row. If the largest value appears more than once in the array, only the location of the first such element is recorded.

A THROUGH statement causes the block of statements which follows immediately afterwards to be repeatedly executed, each time varying the value of some variable, until the specified list of values for that variable is exhausted, or until some specified condition is satisfied. The THROUGH statement may take either of the following two forms. (The second form is used more frequently than the first.)

(a) THROUGH $\mathscr{S}$, FOR VALUES OF $V = E_1, E_2, \ldots, E_m$

Here $\mathscr{S}$ is the statement label of the last executable statement in the block to be repeated. The block of statements following (and not including) the THROUGH statement, up to <u>and including</u> the statement whose label is $\mathscr{S}$, will be called the "scope" of the THROUGH statement. Following the word OF appears the name of the iteration variable (in the illustration: V), which may be either an individual variable or subscripted array variable of any mode. To the right of the " = " sign may appear any number, i.e., a list, of expressions $E_1, \ldots, E_m$. The modes of the E_i bear the same relationship to the mode of V as they would in the statement $V = E_i$ (see Section 2.1). Thus, if V is an integer or a floating point variable, then each of the E_i must be an integer or floating point expression. Similarly, if V is Boolean, then each of the E_i must be a Boolean expression.

The execution of this statement causes the statements within its "scope" to be executed, first with $V = E_1$, then again with $V = E_2$, and so on, until the list of expressions is exhausted. Computation then proceeds with the statement immediately following statement $\mathscr{S}$. At this time the iteration variable will have the value of the expression E_m unless its value was changed during the final iteration. Should a transfer be made to another part of the program at any time during the iteration, V will have its current value. An example of this type of statement is:

```
  THROUGH A, FOR VALUES OF BETA = 3, 4, X5, Y(6 + I) + 3
  J = 5 * BETA + 6
  J1 = J .P..5 - 1.
A X(BETA) = J1 * COS.(2. * THETA)
```

(b) THROUGH $\mathscr{S}$, FOR $V = E_1, E_2$, B

Here $\mathscr{S}$ is a statement which defines the "scope" exactly as in (a) above (with the exception: if $\mathscr{S}$ is the label of the THROUGH statement itself, the iteration will proceed as described below, but the scope will be empty, and the iteration will consist only of the incrementing of V by E_2. and the testing). Following the word FOR is the name of the iteration variable (in the illustration: V), which may be an individual variable or subscripted array variable of any mode, e.g., V may be an integer or a floating point variable and E_1 and E_2 may be integer or floating point expressions. In fact V, E_1, and E_2 may be of any modes such that $V = E_1$ and $V = V + E_2$ are defined. B is a Boolean expression.

The execution of the statement proceeds as follows: The variable V is set equal to the value of E_1. If the value of B = 1B, the scope of the THROUGH statement is not executed. If the value of B = 0B, the scope is executed. V is then incremented by the value of E_2, and B is tested again. In general, as soon as B = 1B, the scope is not executed, and the computation proceeds from the statement immediately following statement $\mathscr{S}$. Each time B = 0B, the statements in the scope are executed, then V is incremented by E_2, and B is tested again. Thus, when the iteration is finished and B = 1B, V has the value used during the last computation of the scope, incremented by E_2. The scope will not have been executed for this value of V. (The value of V will be E_1, of course, i B = 1B before the scope is executed at all.) If, at any time, the computation transfers out of the iteration to another part of the program, the value of V will be the current value at the time the transfer was made.

In all cases, every reference to an expression E_i will involve its current value at the time of reference. Moreover, the variable V may have its value changed at any time during the execution of the scope. In a statement of the form (a), the value of V will be reset by the value of the next E_i for the next computation of the scope. In a statement of form (b), the current value of V will be used for incrementing, testing, etc.

Examples:

(1) To evaluate the polynomial $c_nx^n + c_{n-1}x^{n-1} + \ldots + c_1x + c_0$ using the formula $(\ldots((c_nx + c_{n-1})x + c_{n-2})x + \ldots + c_1)x + c_0$ (nested multiplication), we may write the program:

```
          INTEGER J,N
          Y = 0.
          THROUGH POLY, FOR J = N, -1, J .L. 0
POLY      Y = X * Y + C(J)
```

(For the meaning of the statement INTEGER J,N, see Section 3.2.)

(2) A Newton's Method solution $(x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)})$ of the equation $f(x) = \cos x - x = 0$ could be written as a single statement, using the criterion "$|f(x)| < \epsilon$ and $|x_k - x_{k+1}| = \left|\frac{f(x_k)}{f'(x_k)}\right| < \epsilon$" for stopping the iteration:

```
NEW       THROUGH NEW, FOR X = X0, (COS.(X)-X)/(SIN.(X) + 1.),
   1          .ABS.(COS.(X) - X) .L. EPSILN .AND.
   2          .ABS.(COS.(X)  - X)/(SIN.(X) + 1.)) .L. EPSILN
```

where X0 is the initial guess.

(3) If the value of the iteration variable is to be altered within the scope of the iteration, one may use a zero increment. E.g., suppose J is an integer variable, and the scope of the iteration is to be performed for those values of J which are multiples of 2, but not multiples of 5, and at the same time are less than the value of the integer K. One might write the iteration as follows:

```
          THROUGH END, FOR J = 2, 0, J .GE. K
          ...
          ...
          J = J + 2
END       WHENEVER J .E. (J/5) * 5, J = J + 2
```

(4) A table-look-up procedure using an iteration statement. Suppose that a string of alphabetic (or numeric) characters (i.e., a "sentence") has been decomposed into single characters stored in C(1), C(2), ..., C(K), where K is the length of the string. Then the first occurrence of a comma could be found as follows:

```
LOOK      THROUGH LOOK, FOR I=1, 1, C(I) .E. $,$  .OR. I .G. K
          WHENEVER  I .G. K, TRANSFER TO NOCOMA
```

(c) As an alternative to the THROUGH statement and its scope, there are certain special constructions called the <u>iterated expressions</u> and <u>iterated statements</u> which are available. For details see Section 2.17.

2.6 Nested Iteration Statements

As indicated in Section 2.5, the "scope" of an iteration statement is the block of statements designated for repeated execution:

```
                    THROUGH END, FOR V = E1, E2, B
scope {                 ...
                        ...
                 END    ...
```

Some of the statements within the scope of an iteration may themselves be iteration statements. However, if iteration statement <u>b</u> is in the scope of iteration statement <u>a</u>, then the scope of <u>b</u> must be entirely within the scope of <u>a</u>. Figures 2-3 and 2-4 show valid configurations:

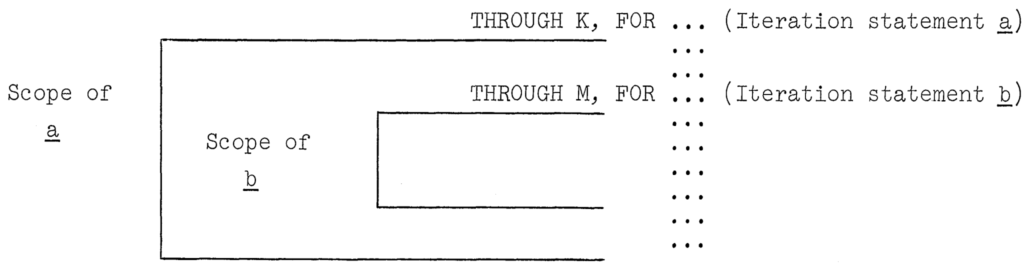

THROUGH K, FOR ... (Iteration statement a)
THROUGH M, FOR ... (Iteration statement b)
Scope of a
Scope of b

Figure 2-3

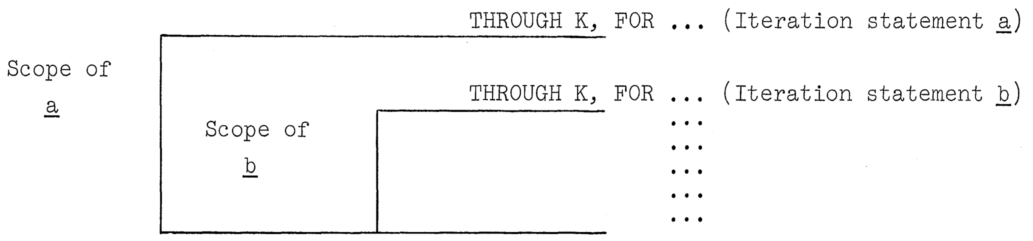

THROUGH K, FOR ... (Iteration statement a)
THROUGH K, FOR ... (Iteration statement b)
Scope of a
Scope of b

Figure 2-4

In Figure 2-4, although the scopes of <u>a</u> and <u>b</u> both end on the statement labeled K, iteration <u>b</u> is incremented and tested first. Therefore, iteration <u>b</u> is completed before iteration <u>a</u> is incremented. Figure 2-5 shows another example of a valid configuration.

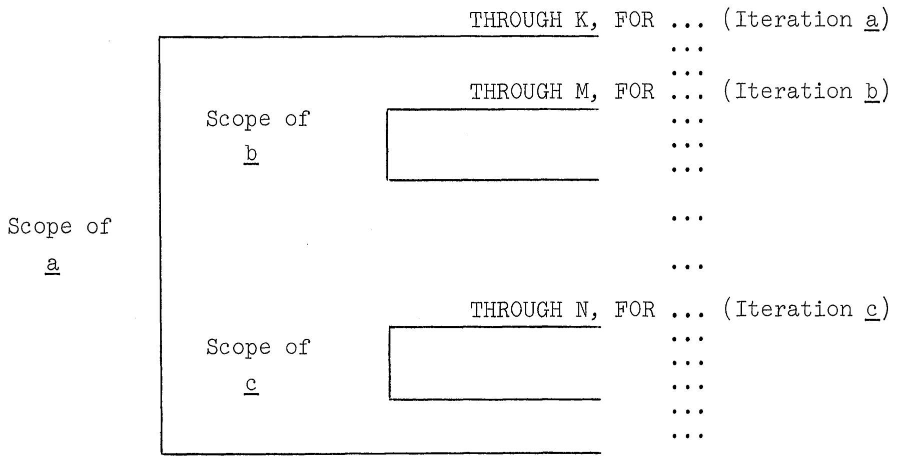

THROUGH K, FOR ... (Iteration a)
THROUGH M, FOR ... (Iteration b)
Scope of b
Scope of a
THROUGH N, FOR ... (Iteration c)
Scope of c

Figure 2-5

Figure 2-6 represents an invalid configuration:

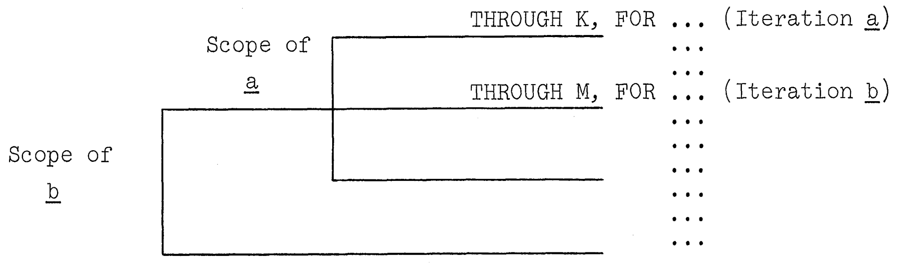

THROUGH K, FOR ... (Iteration a)
Scope of a
THROUGH M, FOR ... (Iteration b)
Scope of b

Figure 2-6

When iteration statements occur in the scope of other iteration statements, they are said to be "nested." The "nesting depth" of a statement is the number of iteration statements in whose scope it appears. The nesting depth of an iteration may not exceed 50.

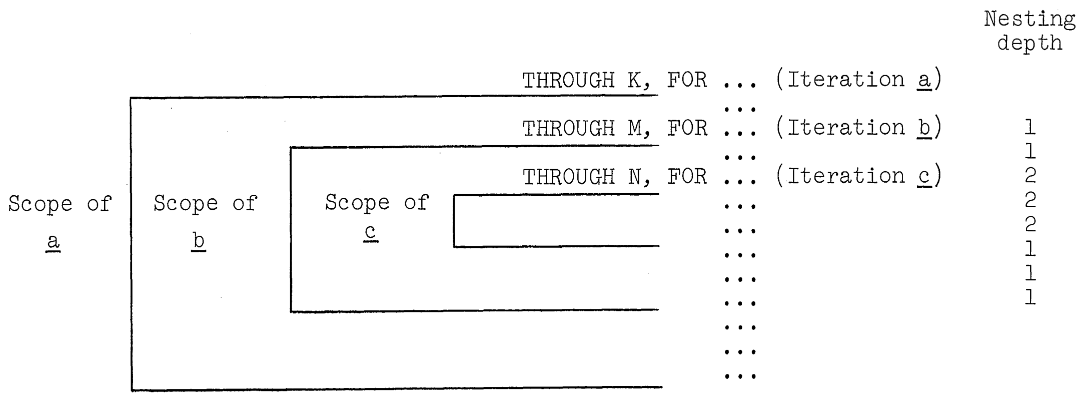

Nesting depth
THROUGH K, FOR ... (Iteration a)
THROUGH M, FOR ... (Iteration b) 1
1
THROUGH N, FOR ... (Iteration c) 2
2
2
1
1
1
Scope of a
Scope of b
Scope of c

Figure 2-7

In Figure 2-7 iteration <u>a</u> has a nesting depth 0, iteration <u>b</u> has nesting depth 1, and iteration <u>c</u> has nesting depth 2. In Figure 2-5 both <u>b</u> and <u>c</u> have nesting depths of 1

A form of nesting which often leads to confusion, although the compiler will accept it, is shown in Figure 2-8. This is the case of a partial overlap in the scopes of an iteration and of a compound conditional. Such overlap should be avoided.

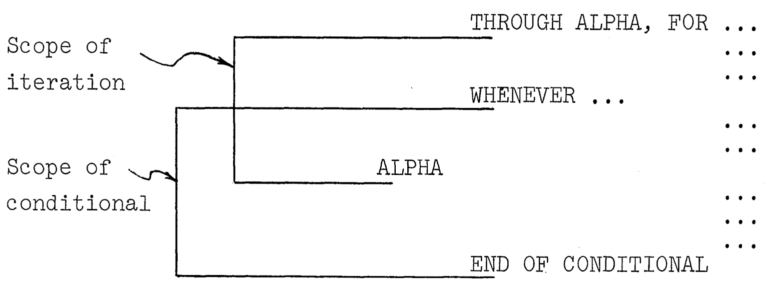


Figure 2-8

There are no restrictions on jumping into or out of the statements in the scope of an iteration.

Automatic indication of nesting depth

At the right side of the listing of the MAD source program produced by the MAD compiler there appear two numbers on occasion. The first of these indicates the nesting depth within compound conditionals of the statement, and the second indicates the iteration nesting depth as suggested in Figure 2-7. If either of these numbers is zero, it is not printed. This is especially useful in cases where either an END OF CONDITIONAL statement or the statement ending a THROUGH loop is omitted.

2.7 Pause Statement

PAUSE NO. n

This statement indicates a breakpoint in the program, and it causes the computer to stop in such a way that the operator can manually start it and automatically go on to the next statement in the program. The number n is an octal integer (written without the K) containing up to 5 digits, which will be displayed on the computer console for the operator to note when the computer stops, thus indicating the point in the program at which the stop occurred.

2.8 Function Statement

Normally, the value of a function will occur as part of an expression, as in the statement:

```
Z = COS.(X)/SIN.(X + 2.)
```

Certain functions, however, may stand alone as separate statements, as a statement calling for the sorting of a list, etc. This would appear as:

```
(a)     EXECUTE LSORT.(ARRAY, MAP, N)
        or alternatively as:
(b)     LSORT.(ARRAY, MAP, N)
```

(See Section 3.8 for definition of functions.) Here ARRAY, MAP, N are the "arguments" of the function (subroutine) LSORT. A function called as in (a) or (b) above need not be followed by a list of arguments.

2.9 Error Return Statement

Provision is made for including an error return in function definition programs (see Section 3.10). The form of the statement is simply:

ERROR RETURN

(An example of its use is shown near the end of subsection 3.10.3.)

In order to use it in a function evaluation, the last (right-most) argument of the function must be the label of a statement or a variable in statement label mode, whose value is the label of a statement to which a transfer is made in case the ERROR RETURN statement is executed.

Even though an error return has been provided in the definition of a function, in the use of the function the last argument may be omitted. If it is omitted, execution of the ERROR RETURN statement will cause control to be transferred to the operating system in which the translated program is embedded, with an error indication. Note that the extra argument used in a call for the function does not appear at all in the function definition.

Specifically, suppose the function F has been defined with n dummy variables, and suppose that in this definition an ERROR RETURN statement can be executed. Then a call on F may employ either n or n + 1 arguments. If n + 1 arguments are used then the n + 1st of these must be a statement label which will be referred to as a result of executing the ERROR RETURN. If n arguments are used in the call, then execution of ERROR RETURN will cause a return to the operating system. See the example in Section 3.10.3.

2.10 End of Program Statement

This executable statement has the form: END OF PROGRAM

This statement must be physically the last statement in the program (i.e., the last card of the program being compiled). It may also be the last step in the sequence of computation. Execution of this statement will transfer control to the operating system in which the translated program is embedded. An alternate way of terminating a program - i.e., returning to the operating system - is to attempt to execute an input statement when the data has been exhausted.

2.11 Function return

FUNCTION RETURN $\mathcal{E}$

This statement is used in a function definition to indicate a return to the calling program. $\mathcal{E}$ is an expression whose value is to be the output of this function when the function is considered as a single valued function. $\mathcal{E}$ need not be specified if output variables are specified in the argument list (see Section 3.8), or if no value is to be returned.

2.12 Entry to a Function

Entry points to a function being defined, particularly alternate entry points, are indicated by

ENTRY TO η.

where η is the name of this entry (see Section 3.10).

2.13 List Manipulation Statements

These statements facilitate the writing of recursive internal and external functions and other algorithms which employ push down lists or "stacks". (See Section 3.10.) They cause the designation and use of a vector for the temporary storage of data and actual transfer instructions (i.e., function returns).

(a) SET LIST TO $\mathcal{U}, \mathcal{E}$

$\mathcal{U}$ is the name of an array element designated as the initial location to be used for temporary storage. This statement is an executable statement; thus different arrays may become the currently-designated LIST during execution. The value of the expression is checked as an upper limit on the length of the list. If $\mathcal{E}$ (with the comma) is omitted, no checking is done. The zeroth element of the list contains the index of the most recently added element. Since V(0)≡V the "top" element on a list V is V(V). The programmer should be sure to assign V(0) an initial value. The usual case is handled by the assignment statement V=0.

Note: The index in V(0) must always have the form of a MAD integer. If the array V is not of integer mode, statements such as V = V - 1 and V(V) = A will not give the desired action. In this case, the EQUIVALENCE declaration (Section 3.4) can be used to permit reference to V(0) by some other name which is of integer mode.

(b) SAVE DATA $\mathcal{L}$

This statement causes the current values of the elements of the list $\mathcal{L}$ to be stored in the order of their appearance in the list $\mathcal{L}$ as consecutive elements of the currently designated temporary storage vector--starting with the first currently available element. $\mathcal{L}$ has the form of an output list as defined in Section 2.14.

Example:

Let V be an integer array which is the currently designated temporary storage vector as a result of executing

```
SET LIST TO V
```

Further, suppose that V(0) currently has the value 6. Then execution of

```
SAVE DATA A, C, F
```

will cause the following action:

(1) the values of A, C, F, presumably of integer mode also, will be assigned to V(7), V(8) and V(9) respectively, i.e., as if the statements

V(7) = A, V(8) = C, and V(9) = F

had been executed.

(2) as a result of these assignments, the value of V(0) will be increased to 9.

(c) SAVE RETURN

This statement will appear only in the scope of the definition of a function (see Section 3.10). Its execution causes the reentry point to the calling program to be stored as the next available element of the currently designated temporary storage vector.

(d) RESTORE DATA $\mathcal{L}$

$\mathcal{L}$ is an *input list* as defined in Section 2.14. If the list designates *n* storage assignments (not necessarily *n* items in the list) the values of the last *n* elements of the currently designated temporary storage vector are assigned as the values of the list variables. The order of this value assignment, assuming *k* used locations in temporary storage vector, (left to right in the list $\mathcal{L}$) correspond to the k, k - 1, ..., k - n + 1 elements of the temporary storage vector. The last *n* elements of the temporary storage vector are then made available for use by successive SAVE statements.

Example (1):

Let V be the currently designated LIST and let the value of V(0) be 15. The execution of

```
RESTORE DATA B(1) ... B(3)
```

will cause the following action:

(1) values stored in V(15), V(14) and V(13) will be assigned respectively to B(1), B(2) and B(3).

(2) the value of V(0), which designates the "most recently added element" is decreased (by 3) to 12.

Example (2):

If the statement

```
SAVE DATA A, B(1) ... B(3), C
```

is used, the numbers A, B(1), B(2), B(3), C will go onto the push-down list, in that order. If one wishes to return these values to their original locations, the statement

```
RESTORE DATA C, B(3) ... B(1),A
```

is used, since the value of C, being the last on the list, is the first to be restored.

(e) RESTORE RETURN

This statement will appear *only* in the scope of a function definition and causes the current last element in the currently designated temporary storage vector to be used as the reentry point to the program calling the function. The last element of temporary storage is then made available for use by successive SAVE statements.

2.14 Input-Output Statements

In this section the following definitions will hold.

$\mathcal{F}$ denotes the format specification "vector" (see Section 2.15). $\mathcal{F}$ may be:

(a) a single integer variable whose value is the format specification, or

(b) an integer array element which is the first element of the format vector, or

(c) the vector itself, written as an alphabetic constant (which in this case may be more than six characters in length). See Example (1) in Chapter III.

In (c), format information will be grouped six characters per word and preset automatically (along with dimension information produced by "simple dimensioning") in an internally created and dimensioned vector called .MODE1, which will appear in the symbol table. (See Appendix B.)

$\mathcal{N}$ is an integer expression whose value is a tape number.

$\mathcal{L}$ is an input-output list. Elements of $\mathcal{L}$ may be:

(1) single variable names or array names with subscripts,

(2) blocks of the form $A(i_1, \ldots, i_n) \ldots A(j_1, \ldots, j_n)$ or $A(i_1, \ldots, i_n), \ldots, A(j_1, \ldots, j_n)$ where the i's and j's may be any subscript expression and $n \geq 1$,

(3) iteration elements of the form

$$(\mathcal{V} = \epsilon_1, \epsilon_2, \mathcal{B}, \mathcal{L})$$

where $\mathcal{V}$ is the name of a variable, ϵ_1 and ϵ_2 are arithmetic expressions, $\mathcal{B}$ is a Boolean expression, and $\mathcal{L}$ is a non-empty input-output list of the type being presently defined. (This list element may not be used in lists which are to be transmitted to or from binary tapes.) The interpretation of such an element is exactly analogous to the execution of an interation statement in that the values designated on the list $\mathcal{L}$ are transmitted (as input or output) until $\mathcal{B}$ has the value 1B, with $\mathcal{V}$ being initialized to ϵ_1 and being incremented by ϵ_2 after each transmission of the list $\mathcal{L}$. These iterations may be nested (just as iteration statements) to a depth of 50 - s, where s is the current nesting depth of iteration statements in whose scope this element occurs. In addition to these, when $\mathcal{L}$ designates an output list, the elements of $\mathcal{L}$ may be:

(4) constants, or

(5) expressions including iterated expressions (see Section 2.17.1.) (There is one small limitation on the use of expressions. If a function call occurs in the list, such as: X + F.(X, Y), the function F may use only a limited number (currently 20) of the first locations of erasable storage (see Section 3.5).) Moreover, the function F must not itself contain input or output statements which use the same subroutines as the output statement in progress.

Elements of a list are separated by commas. Example of an output list: AB, D, 2.5, MTX(1) ... MTX(N), P(14), J(I, K). Example of a list which may be used either for input or output: A, B, K(3), J(25*I-L), A(K+1)...A(L*2). To completely understand the operation of statements which make an explicit reference to "TAPE," it is advisable to read the sections of the IBM 7090 Reference Manual which describes the operation of these units. (See also Restrictions in Section 4 below.)

(a) PRINT FORMAT $\mathcal{F}, \mathcal{L}$
Prints $\mathcal{L}$ according to format $\mathcal{F}$ on the off-line printer.

(b) PRINT ON LINE FORMAT $\mathcal{F}, \mathcal{L}$
Prints $\mathcal{L}$ according to format $\mathcal{F}$ on the on-line printer. This statement is used for comments to the operator. After $\mathcal{L}$ has been printed a skip of 1/6 page is automatically produced, allowing the operator to read the comment.

In the following statement definitions, the phrase "Prints $\mathcal{L}$" (or "Punches $\mathcal{L}$" or "Transmits $\mathcal{L}$") should be interpreted as "prints (or punches or transmits) the value of each element of the list $\mathcal{L}$".

(c) PUNCH FORMAT $\mathcal{F}$, $\mathcal{L}$
Punches $\mathcal{L}$ on cards according to format $\mathcal{F}$.

(d) READ FORMAT $\mathcal{F}$, $\mathcal{L}$
Reads cards into $\mathcal{L}$ according to format $\mathcal{F}$.

(e) READ BCD TAPE $\mathcal{N}$, $\mathcal{F}$, $\mathcal{L}$
WRITE BCD TAPE $\mathcal{N}$, $\mathcal{F}$, $\mathcal{L}$
Transmits $\mathcal{L}$ to or from tape $\mathcal{N}$ in BCD form according to format $\mathcal{F}$.
Note that when these statements are executed, the input-output conversion subroutine will enforce the usual restriction on the length of a "line". Therefore, the format should not specify more than 132 characters per record.

(f) READ BINARY TAPE $\mathcal{N}$, $\mathcal{L}$
Transmits to $\mathcal{L}$ from tape $\mathcal{N}$ until $\mathcal{L}$ is exhausted or until a complete tape record has been read. If there are more elements on the list than words in the record being read, reading will continue into the next record.

WRITE BINARY TAPE $\mathcal{N}$, $\mathcal{L}$
Transmits $\mathcal{L}$ to tape $\mathcal{N}$ in binary form as one record.

(g) LOOK AT FORMAT $\mathcal{F}$, $\mathcal{L}$
Transmits the information on the next card on the input tape, according to format $\mathcal{F}$, into $\mathcal{L}$ without going past the card. Hence the next time a READ FORMAT or LOOK AT FORMAT statement is processed, this same card will again be transmitted. Warning: If more than one card is specified (via one or more slashes in the format), each instruction to get a new card merely causes the same card to be rescanned.

(h) Simplified Input-Output
Statements from this group were illustrated in Chapter I. They are defined separately in Section 2.16. The symbol "List" which appears in the PRINT RESULTS statements is essentially similar to "$\mathcal{L}$" as defined above, but differs in details.

READ DATA
READ AND PRINT DATA
PRINT COMMENT
PRINT RESULTS List
PRINT BCD RESULTS List
PRINT OCTAL RESULTS List

(i) END OF FILE TAPE $\mathcal{N}$
Writes an end of file mark on tape $\mathcal{N}$.

(j) BACKSPACE RECORD OF TAPE $\mathcal{N}$, IF LOAD POINT TRANSFER TO $\mathcal{S}$
Move tape $\mathcal{N}$ back to beginning of last record. If the tape encounters the load point, the program goes to the executable statement labeled $\mathcal{S}$ (where $\mathcal{S}$ is a statement label or variable of statement label mode). If the phrase ", IF LOAD POINT TRANSFER TO $\mathcal{S}$" (including the comma) is omitted no transfer is made even if at load point.

(k) BACKSPACE FILE OF TAPE $\mathcal{N}$, IF LOAD POINT TRANSFER TO $\mathcal{S}$
Tape $\mathcal{N}$ is moved backward until an end-of-file mark, the load point gap, or the load point is encountered. If an end-of-file mark has been written previously, executing this statement backspaces over this end-of-file mark and stops just in front of it (since the end-of-file mark must be passed over to be recognized). If the tape encounters the load point, the program goes to the executable statement labeled $\mathcal{S}$ (where $\mathcal{S}$ is a statement label or variable of statement label mode). If the phrase ",IF LOAD POINT TRANSFER TO $\mathcal{S}$" (including the comma) is omitted no transfer is made even if at load point.

(m) SET LOW DENSITY TAPE $\mathcal{N}$
Causes tape $\mathcal{N}$ to be set to <u>low</u> density.

(n) SET HIGH DENSITY TAPE $\mathcal{N}$
Causes tape $\mathcal{N}$ to be set to <u>high</u> density.

(o) UNLOAD TAPE $\mathcal{N}$
Causes tape $\mathcal{N}$ to rewind and unload.

(p) REWIND TAPE $\mathcal{N}$
Rewinds tape $\mathcal{N}$.

2.14.1 <u>End of File and End of Tape</u>

If an error (improperly formed format specification, invalid data, a tape check, etc.) occurs during any input-output statement, the subroutine ERROR is automatically entered. The subroutine ERROR sets an error flag and returns control to the system in which the translated program is embedded.

An end-of-file on the input tape (tape 7) signifies the end of the data. Normally the job is terminated and the comment,

"**** ALL INPUT DATA HAVE BEEN PROCESSED"

is printed. An end-of-file on other tapes terminates the job with the comment

"****END OF FILE ON SCRATCH TAPE $\underbrace{\text{XX}}_{\text{tape number}}$"

This procedure can be altered by executing the subroutine SETEOF prior to the reading action. The calling sequence is

SETEOF.(S, T) (T is optional)

S is a label which designates where to go when an end-of-file is encountered and T, an integer variable, designates where to store the number of the tape on which the end-of-file was found. In this case the above comments are not printed when the end-of-file occurs. The end-of-file procedure can be reset to normal procedure by executing SETEOF with S = 0, i.e., SETEOF.(0).

When the end-of-tape condition is detected on the output tape (tape 6) the tape is terminated with an end-of-file mark and replaced with a blank tape by the operator. When writing on tapes other than the output tape, the job is terminated and control is returned to the executive system. This procedure can be altered (for tapes other than tape 6) by executing the subroutine SETETT prior to the writing action. The calling sequence is

SETETT.(S, T) (T is optional)

S is a label which designates where to go when the end-of-tape condition occurs and T designates where to store the number of the tape that caused the end-of-tape. In this case the tape is not terminated and replaced. The end-of-tape procedure can be reset to the normal procedure by executing SETETT with S = 0, i.e., SETETT.(0).

The subroutines SETEOF and SETETT may be executed as many times as desired. Only one setting is in effect for end-of-file (that specified by the latest execution of SETEOF) and end-of-tape (that specified by the latest execution of SETETT), i.e., each setting cancels the previous one.

2.15 FORMAT SPECIFICATIONS

When information is read from (or punched in) a card into (or from) a computer, it is necessary to know how this information has been allocated among the available columns of the card. Similarly, whenever information is to be printed by a printer (either on-line or off-line), it is necessary to know how this information has been allocated among the columns available on the printer. A description of each allocation is called a format specification. Usually, but not always, such a specification is accompanied by a list of variables whose values are to be printed or punched or whose values are to be read. (Occasionally, the information is contained entirely in the format specification, so the list may be empty.) A format specification is a string of alphabetic characters (as described below) terminated by an "*". This string is stored six characters per computer word in a vector of integer mode. This vector may be preset (see Section 3.7), computed, or read in as data. (Format features in addition to those described below may be found in the University of Michigan Executive System Manual.)

2.15.1 Carriage Control

Every format specification consists of a description of the allocation of available columns, and the form in which the particular information is to appear. Specifications for reading or punching cards and printing of information are identical, with the following exceptions:

(a) A card has 80 available columns,[†] while a line of print has 132 available columns. Any attempt to allocate more than the available number of columns will cause an error return (see Section 2.14.1).

(b) The first column of a card has no special significance. The first (left-most) character of a line of print is treated differently. This character governs printer carriage control, such as skipping to a new page, double spacing, etc., and should not contain information to be printed. The user has effectively 131 available columns on a line of print, but he in addition must always include as the first character a code to control the vertical spacing of that line.

[†]Actually, 80 columns may be used, but it is highly recommended that only 72 columns be used, with the last eight columns, 73 - 80, used for card identification information.

For example, the specification

S16, 6HBETA = I2*

will indicate that the line (or card) starts with a skip of 16 columns and then prints (or reads or punches) the characters "BETA =" followed by an integer field of two columns. The effect of the "carriage control" character (in this example, the "blank") may be described as follows: Before printing the line, the carriage is "positioned" according to the carriage control character as indicated in the table below. Then the legal carriage control character is replaced by a "blank", and the line is printed. If the first character is not a legal carriage control code, a single space is given before printing, and the entire line is printed.

Legal Carriage Control Codes

Character	Before Printing
Blank	Single space before printing.
+	No space - overprint the previous line.
0	Double space before printing.
-	Triple space before printing.
1	Skip to top of next page before printing.
2	Skip to next half-page.
4	Skip to next quarter-page.
6	Skip to next sixth-page.
8	Skip to next sixth-page.
9	Single space before printing; suppress overflow. I.e., print across page boundaries, without skipping.

Carriage control characters apply only to lines that are to be printed off-line, and not at all to cards or to on-line printing. Normally, when the printed lines come within 3 or 4 lines from the bottom of the page, there is an automatic skip to the next page, called the "overflow" skip. The carriage control character 9 suppresses this skip so printing will continue across page boundaries. This is rarely used.

As another example, the specification

6H1PHI□=F6.3*

"□" signifies a blank space

would cause a skip to the next page (because the first character is a "1"), and cause "PHI□=" to be printed, followed by a fixed point number. As will be explained below, the "6H" that appears in the specification indicates that 6 Hollerith (or BCD) characters follow, e.g., "1PHI□=". Note that blanks are counted as characters here.

2.15.2 The Basic Field Description

Each specification describes successive fields across the available columns of the line (or card) record starting from the left. If the specification describes fewer than the total number of available columns, the line (or card) will automatically be filled in with blanks. But, as indicated above, if more than the total number of available columns is included in the specification, an error return (see Section 2.14.1) will result.

The basic field description consists of a letter followed by an integer. The letter (except for T) indicates the form of the information in the external medium as follows:

S	Skip columns
I	Integer
F	Fixed point number (Internally floating point)
E	Floating point number
K	Octal number
C	Characters
T	(Causes a transfer of the format scanner)

(For the purposes of input-output the Boolean values 0B, 1B are considered as integers and will be punched in cards and appear in print as 0,1.)

The integer indicates the size of the field; i.e., the number of available columns to be used. For example, K3 indicates a 3-column octal field, C23 indicates a 23-column character field, and S31 indicates a skip of 31 columns. The T field is explained in Section 2.15.10 below.

For F and E fields, i.e., fixed and floating point numbers, there is always the question of the placement of the decimal point, the form of the numbers, etc., in addition to the field size.

For this reason, the basic field description for E and F fields requires an additional integer giving the number of places after the decimal point that are to be rounded and printed (or read, or punched). The decimal point in the specification itself must always be present for E and F fields, except that if no fractional part is desired, the .0 may be omitted (i.e., F6 is the same as F6.0).

Within the numeric type fields, E, F, I and K, "+" signs are not printed or punched, nor are they necessary on input cards. However, "-" signs may not be omitted on input, and are always printed and punched. If the field description gives a field size larger than the number requires, the number is positioned for printing or punching as far right as possible. If the field size given in a specification is too small, an error indication is given (see Section 2.15.12). It is important, therefore, to be sure to provide a large enough field to handle the information expected. In fact, some spacing can be achieved by giving large field sizes, since blanks automatically occur to the left of a number pushed to the right end of an over-sized field.

2.15.3 The Form of the Number

Integers (I fields) and octal numbers (K fields) are printed, punched, etc., directly, without any decimal point. Numbers printed or punched in E fields have the form (if 5 decimal places are requested, for example):

$$\pm .x_1x_2x_3x_4x_5E\pm n_1n_2$$

where n_1n_2 is the exponent. On input cards numbers in E fields must have an exponent of the form

$$E\pm n_1 n_2$$

although if the sign is punched, the "E" may be omitted. Similarly, if the "E" is punched, a "+" may be omitted, as well as a leading zero in the exponent. The exponent must be counted in the field size (four columns on output).

Numbers to be printed or punched in F fields have the form (if 5 decimal digits are requested, for example):

$$\pm x_1 x_2 x_3 . x_4 x_5 x_6 x_7 x_8$$

although "+" signs are not printed. On input cards "+" signs may be omitted, but not "-" signs. If no digits are printed after the decimal point, the point itself is not printed.

On both E and F data any number of digits may be used, but only 8 digits of accuracy are retained. Moreover, E and F input data need not have the decimal point punched, and either E or F type data may be used in any E or F field on input data. If the decimal point is not punched, the field specification determines its position. For example, the punched number +9032 described by the specification F10.2 would be understood to be the number +90.32 because the 2 in the specification indicates 2 decimal places to the right of the point. Similarly, the punched number +9032E3 described by the specification E10.4 would be understood to be the number +.9032E3. If the decimal point is punched in the card field, however, it completely overrides the setting of the point by the specification. The entire specification must be present, however. In K, I, E, and F fields, blanks are ignored on input cards, i.e., to the left, to the right, or even within the number. A completely blank input I, E, or F field is interpreted as minus zero, while a completely blank input K field is interpreted as plus zero.

2.15.4 Permissible Conversion

It should be understood that there must be a relationship between the form of a number inside the computer and its external form. In other words, a number described by an I field specification is assumed to be integer in storage. A number described by E or F specification is assumed to be in floating point form in storage. Similarly a number governed by a K specification will be handled by direct binary-octal conversion. Information described by a C specification is assumed to be in alphabetic (BCD) form both inside the computer and outside.

2.15.5 Repetition of Basic Field Specifications

If several consecutive fields can be described by the same basic specification, repetition may be avoided by prefixing the basic specification by its multiplicity. For example, the specification

```
3F10.3,  E18.4,  2E9.1,  3I2*
```

is a short way of writing

```
F10.3,  F10.3,  F10.3,  E18.4,  E9.1,  E9.1,  I2,  I2,  I2*.
```

Either specification may be used, of course. A group of basic specifications may be repeated by enclosing the group in parentheses and preceding the left parenthesis by the multiplicity. Thus

3E10.3, 2(I2,3F10.1), 2C5*

would be equivalent to the following:

E10.3, E10.3, E10.3, I2, F10.1, F10.1, F10.1, I2, F10.1, F10.1, F10.1, C5, C5*.

A multiplicity of zero in front of a field specification or left parenthesis means that that specification (or group of specifications) is to be entirely skipped, without any effect on the line (or card) or on the list. This is useful in conjunction with the use of Boolean format variables (see Section 2.15.11).

Nested parentheses are allowed; information about parenthesis nesting is retained in a list in the erasable section of storage.

2.15.6 Modifiers

It has been shown that field specifications have the standard form nXw.d, where n is the multiplicity, X the control letter, w is the field width, and d is the number of characters after the decimal point. (The .d may be missing.) Several modifiers may be prefixed to this field specification to cause special effects. Some modifiers (B, P) require integers immediately in front of them, while others (D, L, R, Z) do not. The order in which modifiers occur is immaterial. Examples of modified specifications are:

-3P4F19.6, LK6, D2E25.12

The modifiers have the following effect:

B - Normally, conversion is performed from (to) a decimal external form to (from) a binary internal form. The B modifier allows the use of other external number systems for I conversions. Thus, the modifier 16B causes the external form to be in hexadecimal (base 16) notation. The base thus specified may not exceed 19. For bases greater than 10, the additional characters needed are taken from the beginning of the alphabet. (For base 16, A = 10, B = 11, C = 12, D = 13, E = 14, and F = 15.) Note that K12 is not exactly equal to 8BI12, since the K12 conversion uses the left-most bit as part of the number, while the 8BI12 uses this bit as the sign of the integer. The numbers occurring in the format specification itself are always interpreted as being in the decimal system.

D - Double Precision - If the character D occurs in an E or F specification, it indicates that this conversion is to be performed on a double-precision number (with each half of the number containing its own characteristic and fractional parts). Both halves of the number must be specified on the "list," the high order half first, followed immediately by the low order half.

L - Left justified - For input, this affects only the octal (K) specification. Octal numbers are usually right-adjusted, with leading zeros inserted. The L modifier causes the number to be left-adjusted (i.e., shifted left to eliminate all leading zeros), with trailing zeros supplied. For output, this causes numbers printed with I, E, or F specifications to be positioned in their fields as far to the left as possible (instead of to the right, as is normal).

P - A "scale factor" may be applied to an F number according to the formula

$$\text{Printed number} = \text{Internal number} \times 10^{\text{Scale factor}}$$

(where the scaling is accomplished before the conversion is done). The "scale factor," followed by the letter "P" is prefixed to the field specification, as in the example

-2P2F7.3, F7.3*

Thus, three numbers which would print □□0.522□-1.567□93.671 according to the specification 3F7.3* would print instead □□0.005□-0.016□93.671 if the specification -2P2F7.3, F7.3* were used. It must be noted that for F fields this scale factor actually changes the values of the numbers to which it applies. It affects only those numbers to which it is directly applied, however. For E fields, the "scale factor" causes the number itself to be modified, but the exponent is correspondingly modified so the true value of the number remains unchanged. Thus, the number 0.9321E-3 would print as □□0.9321E-03 according to the specification E12.4, but it would print as □93.2100E-05 according to the specification 2PE12.4. Unlike an F number, the value is the same in either case.

R - This is used with characters brought in by a C specification. Character information is usually left-adjusted, with trailing blanks inserted. The R modifier causes the characters to be right-adjusted (i.e., shifted right to eliminate trailing blanks), with leading blanks supplied.

Z - This forces leading or trailing blanks to be replaced by zeros on a C specification.

2.15.7 Multiple Specifications

Several specifications may be condensed into one larger one by the use of the character "/". Each appearance of "/" (except as part of a Hollerith field--see Section 2.15.8) indicates that the specification of a new line (or card) record is to be started with what follows. A pair "//" causes a blank line (or card), three "///" cause two blank lines (or cards), etc. Thus the specification

3F10.3/I2, K18, 2C5/7I3*

implies that one line (or card) is described by the specification 3F10.3, and the next line (or card) is described by the specification I2, K18, 2C5, and the next by 7I3. It must be noted that each line (if printing is being described) is described individually here, and must include its own carriage control (see Section 2.15.1).

2.15.8 Hollerith fields

Although the C specification is available for transmitting characters (Hollerith information) to and from storage, it is often convenient to include strings of Hollerith characters directly in the format specification. This is done by means of a basic Hollerith field specification consisting of the string of characters to be transmitted, preceded by a count of these characters and the letter "H". Thus, if the specification

1H1 F10.3, 6HBETA = E10.2*

were used in printing, one would obtain a new page skip, because of the one-column Hollerith field containing the character "1." Then a ten-column F number would print, followed by the six characters "BETA□=" and a ten-column floating point field. Note that blanks are completely ignored throughout all format specifications except when they occur as characters in a Hollerith string. Note also, that while every field specification of types S, I, E, F, K and C must be followed by commas, the comma may be omitted after a Hollerith string of the type described here. The comma may be used, however, if desired.

If a Hollerith field is specified without a count in front of the H, then the first character following the H is taken as a "break character," and all characters in the format between it and the next appearance of it are used as the Hollerith field. As an example, the specification

T20, H*AND/OR*, I5*

will cause the six characters AND/OR to appear beginning in position 20 of the record, followed by a five-column integer. Warning: There is a danger here when writing several such Hollerith strings without counts. By not paying attention to the length of these strings, the maximum number of columns allowed for a line (or card) could easily be exceeded.

2.15.9 Character fields

It has been stated above that a number appearing in an oversize field is positioned as far as possible to the right. In the case of C information (characters), any information occurring in an oversize field is pushed to the left, while in either case, unused columns of the field are filled with blanks. Similarly, in case the specification describes too small a field, characters are taken from the left end of the field until the field is exhausted.

Thus, if a card contains the characters: ABCDEFGHIJK in column 1 through 11, and it is read according to the specification 2C3*, the two six characters words that are read into the computer are:

ABC□□□
DEF□□□

(where "□" denotes "blank"), while the specification C6* would cause a single word to be read:

ABCDEF

and C7,C3* would cause the words:

ABCDEF and HIJ□□□

to be read (since at most six characters can go into one word of storage).

2.15.10 Relationship between the List and the Specification

The "list" consists of a set of names of variables to or from which information is to flow. Except for Hollerith strings embedded in the format specification itself (see Section 2.15.8) and the S fields, each field in the specification refers to one item on the list. For this purpose, a block entry on the list, such as A(6) ... A(8), counts as several names of variables (in this case, the three variables A(6), A(7), and A(8)), During the transmission of information, the input or output subroutine scans both the list and the specification simultaneously, correlating corresponding entries, and associating a field size, a type of conversion, etc., to each variable. If a Hollerith string is encountered in the specification, it is immediately transmitted, and it is not associated with any item on the "list." A T field causes the next specification to refer to a specified column in the line (or card) image being processed; e.g., T35, 4HABCD* would cause the characters ABCD to be put into columns 35 - 38.

For example, if the list consisted of

A, B(1,1), I, K

where I and K were integers, and the others floating point, and the specification was

1H1, F11.3, -2PE14.4, S13, 3HM = I3, S9, 3HJ = I3*

we might find a printed line something like the following (at the top of the next page because of the 1H1):

```
□□□□456.010□□-16.1251E+02□□□□□□□□□□□□□M□=□50□□□□□□□□□J□=□17
```

The same list would look the following way with the specification 1H1, 2F11.3, S16, 3HM = I3, S9, 3HJ = I3*:

```
□□□□456.010□□-1612.510□□□□□□□□□□□□□□□□M□=□50□□□□□□□□□J□=□17
```

As stated above, a specification may not account for more than 80 columns on a card. It may happen, however, that a list calls for more information than can appear on a single card. Or perhaps only a certain part of each card is to be read. The determining factor in every case is whether or not the entire list has been accounted for. After each card is read according to the format specification, the list is consulted; if not yet satisfied, another card is read, and so on. It is important to realize that the specification is not necessarily scanned from the beginning when an asterisk (*) is encountered and the list is not satisfied. The specification scanner moves to the left from the end of the specification (the *) until it hits a left parenthesis belonging to a pair of parentheses which is not nested inside any other pair of parentheses, and which is not in an H field. (If there is no such left parenthesis, it will move to the beginning of the specification.) It then examines the characters just to the left of this left parenthesis to see if they are a multiplicity indication (see Section 2.15.5). From now until the list is satisfied the effective portion of the format specification begins at the selected left parenthesis (together with the multiplicity, if any) and ends at the end of the specification. A similar statement may be made for printed or punched output.

Thus, in the specification:

3F10.3, /4(F10.3, 6HBETA = I2)*

the first line printed (or read) would have three fixed-point numbers, while all subsequent lines would all be printed (or read) according to the specification

4(F10.3, 6HBETA = I2)*

As an example, one might have an integer on a first data card, followed by many cards, each with six floating point numbers. The specification might then be:

I6/(6E10.5)*

Only the first six columns would be read on the first card, and only 60 columns would be read on subsequent cards. The remaining columns are ignored and may contain any legitimate Hollerith characters.

If a specification contains a Hollerith string of the form $nHa_1a_2...a_n$, certain conventions are observed. (1) If the list is satisfied, but the next field specification is a Hollerith string, the string is transmitted, anyway. (2) On input, i.e., reading from cards, when a Hollerith string is encountered in the specification, the information in the corresponding columns of the input card will be brought in and will replace the BCD string itself within the format specification. This can then be used as a specification for output. For example, this is useful for labeling a set of data and causing the label to appear on the output along with a date, etc.

Thus, a card punched as follows:

```
        1  DATA SET NO. 3-A          JULY 31, 1959          J. DOE
```

might be read in with a format specification

72H□□ ←————————— (72 blank spaces) —————————→ □□*

Later, this specification could be used to print the same information as a heading for the results. Note the "1" provided for carriage control for the printing.

WARNING: The specifications S72* and 72(1H)*, while indicating 72 blank spaces, do not allow the reading in of an entire card, as indicated above, since they do not really provide a storage region of 72 characters in length into which the information on the card may be read and stored until needed.

2.15.11 Use of Format Variables

A format variable (see Section 3.9) may be used at any point in a format specification at which an integer constant is normally found; i.e., within any field specification or as part of any modifier. Each time a format is interpreted (i.e., an input or output statement referring to it is executed), the current value of each format variable encountered is used in the format in place of the name of the format variable. Primes are used to delimit format variables when used in format information. Format variable occurrences may have one of three possible forms:

$$'V', \quad 'V(I)', \quad \text{or} \quad 'V(I_1,I_2)'$$

where V is a format variable and I, I_1, and I_2 are either integer constants or format variables. Also, format variables V, I, I_1, and I_2 may be of either floating point, integer, or Boolean mode. Boolean and floating point values will be converted to integers before being used. 1B and 0B will be converted to 1 and 0, respectively, and floating point values will be truncated as usual. Thus, in the format 'A'F3.1, F10.6*, where A is a Boolean format variable, the specification F3.1 will be used if A has the value 1B, and not otherwise. All format variables must be names that exist in the program in which the input or output statement using the format occurs (i.e., they may not be arguments to a function which uses them as format variables), and they must be declared to be format variables using a FORMAT VARIABLE declaration (see Section 3.9).

2.15.12 Input-Output Error Procedures

The normal procedure after detection of an error is to print a description of the error along with other pertinent information and then terminate execution of the program. This procedure can be altered by executing the subroutine SETERR prior to reading or writing. The calling sequence for SETERR is:

```
SETERR.(S,E,T)
```

T is the location where the logical tape number will be stored, if an error occurs. E and T are optional arguments. S is a label which designates where to go when one of the standard errors occurs and E designates where to store the error number. The comment is still printed. The error number is increased by 100 if the error occurs during output (e.g., error number = 1 if bad format occurs during input and error number = 101 if bad format occurs during output).†

2.16 Simplified Input-Output Statements

(a) READ DATA

This statement causes information to be read from cards; no list of variable names or format specification is necessary. The values to be read and the variable names are punched in the data cards in a sequence of fields of the form:

$$V_1 = n_1, \quad V_2 = n_2, \quad V_3 = n_3, \quad \ldots, \quad V_k = n_k*$$

The $V_1, \ldots, V_k$ are the variable names and $n_1, \ldots, n_k$ are the corresponding values. Reading is continued from card to card until the terminating mark * is encountered. Only the first 72 columns of a card may be used for data; as in the other cards the last eight columns are reserved for identification. Fields cannot be divided between cards, so the last character in a card not terminated by an asterisk would normally be a comma. However, as a convenience, the end of the card is treated as an implied comma and hence this final comma may be omitted. The variable names may designate single variables or elements of linear and two dimensional arrays. The subscripts on the array variables must be integer constants. The values may be floating point, integer, octal, Boolean, or alphabetic with the forms described for constants of corresponding mode (see Section 1.1).

For convenience in entering values of array elements it is possible to designate only one variable name and have successive numbers, written without names, interpreted as the consecutive values of the array, i.e.,

$$V(j) = n_1, n_2, n_3, \ldots, n_k$$

would be the same as

$$V(j) = n_1, \quad V(j + 1) = n_2, \quad \ldots, \quad V(j + k - 1) = n_k$$

For 2-dimensional arrays successive numbers will be entered in succeeding columns of the designated row until the row - as determined from the current value of the dimension vector - is filled, and then the next row will be started. (The dimension vector is discussed in Section 3.3.)

Zeros must be punched; adjacent commas (,,) are simply skipped. Blanks are ignored throughout except between dollar signs (which are used only to delimit a string of Hollerith characters).

Six or less Hollerith characters - delimited by dollar signs - may be values of single integer variables. Longer strings of Hollerith characters may be entered as elements of arrays. Such strings are divided into six character groups for storage.

As an example illustrating many of the features described herein consider the data card set:

†For a list of current error numbers and their interpretation, see Appendix I to "Reference Manual for I/O (with conversion)" in the University of Michigan Executive System (UMES) Manual.

```
X1 = 1.2,  Y1 = -6.8,  INDEX = 4,  A(4) = 3.1,  -10.93,
12.6,  MATRIX (2,1) = 25E-2, 1.8E-10, 3.14E-8,
STRING (1) = $ END OF PROBLEM $ *
```

It is important to remember that, since such cards are data cards, they should not be included as an integral part of the MAD statements but handled as ordinary numeric data.

Strings of Hollerith characters may extend over more than one data card.[†] The character "$" may be represented within an input string by writing two dollar signs with no spaces between them. The pair of dollar signs "$$" will be replaced by a single dollar sign when read in. There is one exception. If the first of a pair of dollar signs is in column 72 of a card and the other in column 1 of the next card, they will not be interpreted as one dollar sign but instead will be interpreted as the termination of one Hollerith string and the beginning of the next.

(b) READ AND PRINT DATA

This has the same effect as READ DATA, except that after a card is read it is also printed as part of the output of the program. It is printed out exactly as the image of the entire card which was read.

(c) PRINT COMMENT $$\mathscr{S}$$

Here $\mathscr{S}$ designates a string of no more than 132 Hollerith characters. The dollar sign is indicated in this string by the occurrence of two contiguous dollar signs. Blanks are valid characters. The string, delimited by dollar signs as indicated, will be printed, except for the first character which will be interpreted as a carriage control code and will not be printed if it is a legal carriage control. (See Section 2.15.1.) An example statement is:

PRINT COMMENT $1 JOHN PUBLIC, PROBLEM 1$

(d) PRINT RESULTS *List*

Here *List* designates a list of variable names, block designations, or expressions, but not iteration elements. The printed output is analogous to the input in that values of variables are preceded by the appropriate variable name and an equal sign; e.g., "X = -12.4". Blocks are labeled as such and printed using a block format. Elements of three and higher dimensional arrays will be labeled with the equivalent linear subscript. If dummy variables (in a function definition or expression) are included in the list the specific values assigned to such variables or expressions during execution will not be labeled but simply preceded by three dots (...).

An example statement is:

PRINT RESULTS X1, Y1, Z(1) ... Z(N + 1), MTX(1,1) ... MTX(M,N)

(e) PRINT BCD RESULTS *List*

(f) PRINT OCTAL RESULTS *List*

These statements have the same effect as PRINT RESULTS *List* except that the value for each *List* element is treated as BCD (or OCTAL) information, and printed accordingly.

The subscripts which are printed for two-dimensional arrays are based on the assumption that the lower limit of both subscript variables is one. If this is not the case, the subscript values which are printed may not be those the programmer might have expected; however, the result which is printed will be the value of the correct element of the array.

[†]Column 1 of the next card immediately follows column 72.

2.17 Iterated expression and iterated statement

Two constructions are available which resemble the iteration element of an input-output list $\mathscr{L}'$. These are the iterated expression and the iterated statement.

2.17.1 The iterated expression has one of the forms:

$$(V, I = E_1, E_2, B, E_3, E_4, \ldots, E_n)$$

$$\text{or} \quad (V = E_0, I = E_1, E_2, B, E_3, E_4, \ldots, E_n)$$

The interpretation of this element is as follows: The variable V (possibly subscripted) is given the value of the expression E_0, if this is present. The iteration is as in the THROUGH statement, in that I is given the value of the expression E_1, then B is tested, etc., with the expression E_2 used as an increment. Each time B is false, each of the expressions $E_3, E_4, \ldots, E_n$ is evaluated, and each is immediately substituted for V. Thus, the action is similar, in the second form, for example, to the sequence

```
    V = E0
    THROUGH A, FOR I = E1, E2, B
    V = E3
    V = E4
      .
      .
      .
A   V = En
```

except that the final value of V is considered to be the value of the iterated expression, and so an iterated expression may be used in any context in the MAD language in which an expression having the mode of V is legal. Mode conversion (such as floating point - integer) will be performed as usual on the value of V after the iteration, if necessary. Note that any of the expressions $E_0, E_1, \ldots, E_n$ may involve V and/or I. In addition, one may write, instead of some E_i $(i = 3, \ldots, n)$, an assignment statement with some other variable than V on the left. In this case, that assignment is made, and no assignment is made for V at that point. Subsequent assignments are made for V, of course. For example, to carry out the computation

$$y = a_0 + \sum_{i=1}^{M} a_i \frac{\sin^2(x+t_i)}{\cos(t_i)}$$

one might write (using MAD notation):

```
Y = A(0) + (S=0., I=1, 1, I.G.M, S+A(I)*
    SIN.(X + T(I)).P.2/COS.(T(I)))
```

or alternatively:

```
Y=A(0)+(S=0., I=1, 1, I.G.M, Q=SIN.(X+T(I)), S+A(I)*Q*Q/COS.(T(I)))
```

As another example, the first element of a vector C which is greater than the value of B (assuming that there is such an element in C) is

```
C((J,J=1,1,C(J).G.B))
```

Note that here the "scope" of the iteration is empty; i.e., there are no expressions after the Boolean expression.

2.17.2 The <u>iterated statement</u> has the form

$$(I = E_1, E_2, B, S_1, S_2, \ldots, S_n)$$

where S_1 S_2, ..., S_n are assignment statements, iterated statements, or function calls (i.e., EXECUTE statements without the word EXECUTE). Iterated statements are executable.

As an example, one may compute the product C of two arrays A and B (Chapter III, Example 6). If A has dimensions $m \times n$, and B has dimensions $n \times p$, then

$$C_{ij} = \sum_{k=1}^{n} A_{ik} B_{kj} \qquad i = 1, \ldots, m; \quad j = 1, \ldots, p$$

This may be written in MAD as follows:

$$\texttt{(I=1,1,I.G.M, (J=1,1,J.G.P, C(I,J) = }\underbrace{\texttt{(S=0.,K=1,1,K.G.N,S+A(I,K) * B(K,J)))}}_{\text{iterated expression}}\texttt{)}$$

Note the use of the <u>iterated expression</u> here.

The iterated statement and the iterated expression should be recognized as different from the <u>iteration element of an input-output list</u>. The distinguishing characteristics of each are as follows:

(1) The <u>iterated expression</u> has a variable V explicitly written before the iteration structure and the others do not. A single value results from the iteration, viz., the final value of V.

(2) The <u>iterated statement</u> is not embedded within any other statement, except as the second half of a simple conditional, or within larger iterated statements. No <u>value</u> results from executing an iterated statement, except for assignments made during the execution.

(3) The <u>iterated input-output list element</u> occurs only in certain input or output lists. It does not have the variable V written explicitly before the iteration structure, and it has only expressions in its scope. Each value of each expression is used in the input or output transmission, subject to format specifications, as usual. An iterated expression may occur in an input or output list, but it always produces only one value, the value of V.

3. <u>Declarations</u> (Non-executable statements) (See Appendix A for allowable abbreviations).

Declarations are non-executable statements, and, except for function declarations, they may occur anywhere in the program. Their purpose is to furnish information to the translator program or to the reader of the program. Declarations may have statement labels, but names in the label field are ignored by the translator, and may not be referred to in other statements.

3.1 Remark Declaration

A remark declaration consists of any string of characters acceptable to the computer. This statement is completely ignored by the translator, and furnishes information to the reader of the program. Every card of the remark declaration must have an "R" in column 11. A card which is blank in columns 1 - 72 is treated as a remark card. A remark card may occur before or after any other card in a MAD program, i.e., before or after END OF PROGRAM, between continuation cards, etc.

3.2 Mode Declaration

All variables and function values are assumed to have the normal mode unless declared otherwise. The normal mode is floating point unless stated otherwise. Any of the other modes may be specified as the normal mode by writing the following declaration:

NORMAL MODE IS $\mathcal{M}$

where $\mathcal{M}$ is one of the following phrases: INTEGER, BOOLEAN, STATEMENT LABEL, FUNCTION NAME, FLOATING POINT, or MODE NUMBER $\underline{n}$, where $\underline{n}$ is one of the digits 0 through 7, or a parameter. The meaning of each of these digits when used in a mode declaration is discussed in Appendix C. Only one such declaration may appear in a program and it is in effect for the whole program, no matter where it occurs in the program. If a variable or function value is to have a mode different from the normal mode then its mode must be declared in a declaration of the form:

$\mathcal{M}$V, L, ..., F., ..., BFN.

where $\mathcal{M}$ is one of the phrases: INTEGER, BOOLEAN, STATEMENT LABEL, FUNCTION NAME, FLOATING POINT, or MODE NUMBER $\underline{n}$, and where $\underline{n}$ is as just defined; for example

BOOLEAN P, Q, DIGIT., TRUE

Following $\mathcal{M}$ is the list of variables and functions whose values are to be of mode $\mathcal{M}$. A program may contain any number of such mode declarations but a name may not be declared to be of two different modes.

In the statement MODE NUMBER $\underline{n}\mathcal{L}$ where $\underline{n}$ = 5, 6, or 7, and $\mathcal{L}$ is a list of variable names and/or function names, a comma is allowed between $\underline{n}$ and $\mathcal{L}$ if $\underline{n}$ is one of the digits 5, 6, or 7, and a comma is required if $\underline{n}$ is a symbol which has been given a value of 5, 6, or 7 in a previous PARAMETER declaration. (See Section 3.8.)

3.2.1 Automatic mode assignment

All constants are automatically assigned modes by the translator (see Section 1.1). Other automatic assignments of modes are:

(a) A name appearing in the statement label field is assigned statement label mode (see Section 1.3).

(b) A function name constant is assigned function name mode (see Section 1.4).

(c) A vector appearing as the dimension vector of some array in a dimension declaration is assigned integer mode (see Section 3.3).

(d) A vector which is preset by a vector values declaration is assigned a mode consistent with the first assigned value (see Section 3.7).

3.3 The DIMENSION Declaration

In order to be sure that consecutive elements of a vector or array are stored in order in the computer, it is necessary to declare the ranges of the subscripts to be used in referring to elements of the array. If only one subscript is used (i.e., one is referring to the elements of a vector), it is understood that the lowest value a subscript may have is zero, so one declares the highest value the subscript may assume at any time during the computation:

```
DIMENSION V(50)
```

In this case, consecutive storage locations will be assigned for 51 elements, e.g., V(0), V(1), V(2), ..., V(50). Negative subscripts may not be used with vectors. If the name V is used without any subscript, it is exactly the same as if V(0) had been written.

For arrays with two or more dimensions (i.e., each reference to an element requires two or more subscripts), one declares the range of each subscript. Thus, if the array B is two-dimensional, and if the first subscript used with B is expected to take on values between -5 and 10 inclusive, while the second subscript will vary between 1 and 15 inclusive, one would write:

```
DIMENSION B((-5...10)*(1...15))
```

Since many arrays have subscripts with ranges starting with 1, if 1 is the lower bound for a subscript, the one, the three dots, and the parentheses may be omitted, so that the last declaration above would more likely be written:

```
DIMENSION B((-5...10)*15)
```

In this case, storage would be allocated to B(-5,1), B(-5,2), ..., B(-5,15), B(-4,1), ..., B(10,15). There are $10 - (-5) + 1 = 16$ rows and 15 columns in this array, so 241 storage locations would be assigned to the array B. ($16 \times 15 + 1 = 241$; the "first" element of the array has linear subscript 1 - see below.)

Each array is always considered to have storage assigned to it as if it were a vector, regardless of the 2-dimensional (or higher-dimensional) structure declared for it as described here. References to elements of an array may therefore be made by using the appropriate number of subscripts, or by using a single subscript. The "first" element of any array (of dimension two or higher) is automatically set to correspond to the single subscript 1, so that in the example above, B(-5,1) could also be referred to as B(1) if desired. The single subscript is often called the "linear subscript", and the relationship between the subscripts i and j in B(i,j) and the corresponding linear subscript r in B(r) is (for two-dimensional arrays):

$$r = n(i - 1) + (j - 1) + b$$

where n is the number of columns and b is chosen so that the "first" element has linear subscript $r = 1$. For the example above, the first element is B(-5,1). Substituting, we have

$$1 = 15(-5 - 1) + (1 - 1) + b$$
$$b = 91$$

For the array B, then, the relationsip is

$$r = 15(i - 1) + (j - 1) + 91$$

Since B is a vector, the symbol B itself is the same as B(0), which is not considered part of the array B (since the "first" element corresponds to B(1)). Thus, with care, the symbol B may be used as a variable which is quite separate from the array B to which reference is made with subscripts, but is necessarily of the same mode.

Declarations may occur anywhere in the program in any order, and they may be combined into a single statement, so a typical declaration might be

```
DIMENSION P(20), Q(10*20), R((-5...10)*10*20), B((-5...10)*15)
```

Elements of arrays are assigned storage in the order determined by varying the last subscript first, then the next to last, etc., as indicated for the array B above. Thus, if one writes B(0,12), ..., B(1,3) in an output list, for example, with B dimensioned as above

```
B(0,12), B(0,13), B(0,14), B(0,15), B(1,1), B(1,2), B(1,3)
```

would be printed, because the declared ranges of the subscripts would be used. (These are kept for use during execution of the program.) In this way it will correctly happen that the successor to B(0,15) is B(1,1), rather than B(0,16).

3.3.1 Modifying the declared range of array subscripts during execution (use of SETDIM)

It may be that the declared range for a subscript should be modified during execution of the program to reflect the storage requirements of different sets of data. In other words, the need can arise to keep the dimension current.

For example, a program may be written which deals with an $M \times N$ array called D, and the largest values which M and N may have are 30 and 20, respectively. Suppose we employ the declaration

```
DIMENSION D(30*20)
```

For a particular set of data, it might happen that $M = 6$ and $N = 4$. Unless this were reflected in the "dimension information", the output list element D(1,1), ..., D(M,N) would cause the values of D(1,1), ..., D(1,20), D(2,1), ..., D(2,20), ..., D(5,1), ..., D(5,20), D(6,1), ..., D(6,4) to be printed, and many of these values would be meaningless. A library subroutine (i.e., external function) is available called SETDIM, which will update dimension information when executed. The arguments to SETDIM are the name of the array, followed by the new ranges of all the subscripts, in order. In the example used here, one would write (after the values of M and N have been read as data):

```
SETDIM.(D,M,N)
```

The arguments giving the ranges of the subscripts may be any integer valued expressions. The block notation must be used if the lower limit is not 1, so that one might write

```
SETDIM.(D,3...N,M)
```

Expressions of integer mode may be written as part of a block designation. If a subscript is to range from N to 2*N, one would write:

```
SETDIM.(D,N...2*N,M)
```

Additional flexibility is available for more advanced applications, and this is described in Appendix B.

3.3.2 Including dimension information in other declarations

The word DIMENSION may also be replaced by any of the following: PROGRAM COMMON, ERASABLE, INTEGER, BOOLEAN, FLOATING POINT, FUNCTION NAME, STATEMENT LABEL, FORMAT VARIABLE, MODE NUMBER $\underline{n}$ (where $\underline{n}$ is a legitimate integer constant or parameter), with the effect determined by the specific declaration used. In any of these other cases, dimension information is not required for a name on the list. If given, as described above, the dimensioning is in addition to the declared effect. (For PROGRAM COMMON see Section 3.6. For ERASABLE see Section 3.5)

Example:

```
INTEGER A(10), N, P, Q(30*3)
```

3.3.3 Duplicate (or multiple) dimensioning

Several variables with the same dimension information may be grouped in a declaration. Any form of the dimension declaration may be used (see Appendix B). Variables so grouped will actually refer to the same dimension vector, and any change to the dimension information, such as a call for SETDIM, will be a change for all arrays in the group.

Examples:

```
(1)  ERASABLE (A,B,C)(10),(D,E,F,G)(10*15)
(2)  DIMENSION (U,S,P)(25)
```

3.3.4 Automatic dimensioning

Dimensioning is automatic in two situations:

(a) In case a statement label vector, say L, is used (see Section 1.3) and $\underline{n}$ is the highest subscript used on L in the statement label field (i.e., if L($\underline{m}$) appears explicitly in the program, it is assumed that $\underline{m} \leq \underline{n}$), then n + 1 locations are reserved for L. Of course, L may also appear in a dimension declaration, in which case the highest subscript is used.

(b) If part or all of a vector is set by a VECTOR VALUES declaration (see Section 3.7), the vector need not appear in a dimension statement unless the maximum subscript implied by the initial values is not sufficiently high.

3.4 Equivalence Declaration

This declaration has the form

$$\text{EQUIVALENCE } (a_1, a_2, \ldots, a_m),\ (b_1, b_2, \ldots, b_n),\ \ldots$$

where a_i and b_j are individual variables or variables shown with constant linear subscripts.

Example:

```
EQUIVALENCE (A,B), (MATRIX, XARRAY), (C, D(3))
```

and implies that the variables A and B are to represent the same storage location throug out the program, that MATRIX and XARRAY are to represent the same storage location through the program, etc. (Two variables which represent the same location always have the same value at any given time.) Thus, any number of equivalences may be established by one EQUIVALENCE declaration, and any number of such declarations may occur (at any place) in a program.

Variables whose names appear within the same set of parentheses need not have the same mode. The mode must be established by the appropriate MODE declaration for each of the variables. Occurrences within an EQUIVALENCE declaration do not establish mode.

A nonsubscripted array variable name in an EQUIVALENCE declaration represents that element of the array (considered as a one-dimensional vector) whose subscript is zero. Reference in an EQUIVALENCE declaration to an array element of any number of dimensions may be made by linear-subscript only (i.e., as an element of a vector). Note that occurrence of any elements from any two arrays in the same parentheses implies equating the entire arrays accordingly.

3.5 ERASABLE Declarations

This declaration has the form

ERASABLE $\mathcal{a}, \mathcal{b}, \mathcal{c}, \ldots$

where $\mathcal{a}, \mathcal{b}, \mathcal{c}, \ldots$ is a list of one or more variables or array names.

Example:

ERASABLE MATRIX, XARRAY, YARRAY

and implies that the arrays and individual variables listed after the word ERASABLE (which need not have the same number of dimensions) are disjoint (i.e., non-overlapping in storage), but are assigned (in the order listed, from left to right) to a special section of storage which is separate from the usual storage of variables and arrays. Each ERASABLE declaration eliminates the effect of previous assignments to this special section of storage, thus allowing several arrays to occupy the same storage at different times. It should be understood that this storage is accessible to, and may be used by, subroutines. Dimension information may be included, if desired in this declaration.

3.6 PROGRAM COMMON Declaration

This declaration has the form

PROGRAM COMMON $\mathcal{a}, \mathcal{b}, \mathcal{c}, \ldots$

where $\mathcal{a}, \mathcal{b}, \mathcal{c}, \ldots$ is a list of one or more variables or array names.

Example:

PROGRAM COMMON MATRIX, X, Y1, BC

and implies that the arrays and individual variables listed after the words PROGRAM COMMON are non-overlapping in storage and are assigned (in the order in which they occur, from left to right) to a special section of storage which is separate from the usual storage of variables and arrays, and separate from ERASABLE storage (see Section 3.5). Dimension information may be included, if desired.

One use of this statement provides for several sections of a program to refer to variables and arrays by the same names, while being translated and checked out separately. A program divided up this way would have the form of a main program and several external function programs, with the main program being used primarily to call on each of the external functions in turn. Although variables and arrays to be used jointly by several external functions can be communicated as arguments to the functions, assigning them to PROGRAM COMMON makes them available to all sections which declare them as such.

The storage reserved for program common is reserved separately, and is not a part of any program (main program or external function program). Every program which refers to a variable in program common must have the same address assignment for that variable. This is usually done by including identical PROGRAM COMMON and DIMENSION declarations in all the programs which refer to program common.

Another use for this PROGRAM COMMON assignment is in the case of a segmented program, i.e., a program so large that it is written in blocks which will occupy the same section of storage and be brought in one at a time. If one block of a program is to use the results of the previous block's computation, the variables involved should be specified as being in the PROGRAM COMMON region. Then they will not be destroyed in the process of bringing in the new block of program. The PROGRAM COMMON and DIMENSION declarations which set up this storage allocation must be identical in all blocks in which these arrays and variables are to be used, except that later segments may add additional names at the end of the list.

PROGRAM COMMON declarations do not affect variables and arrays which have already been assigned to this common section of storage. If another such declaration occurs in a program, the arrays and variables listed therein are considered appended to the previous list of PROGRAM COMMON arrays and variables. The amount of storage actually reserved for program common is determined solely by the <u>first</u> program to be loaded into the computer. This may be either a main program or an external function program.

3.7 <u>Presetting Vectors</u>

Any vector or portion of a vector (or array when considered as a vector, i.e., using linear subscripts) may be preset (up to 200 locations) by declarations of one of the following two forms:

(a) VECTOR VALUES $A(\underline{n}) = C_0, C_1, \ldots, C_r$

Here $\underline{n}$ is an integer constant, and $A(\underline{n})$ may be written simply as A if $\underline{n} = 0$. The entries $C_0, \ldots, C_r$ may be any constants (not necessarily all of the same mode), and in addition, C_i may be an alphabetic constant with more than six characters between dollar signs. In the latter case, the alphabetic constant C_i is treated as if it were broken up into six-character groups from the left, with any partial group filled with blanks at the right. (This must be an explicit list of constants; block notation may not be used here.) If there are $\underline{s}$ constants $d_0, \ldots, d_{s-1}$ in the list after breaking up long alphabetic constants, then the elements $A(\underline{n}), A(\underline{n}+1), \ldots, A(\underline{n}+\underline{s}-1)$ are preset (in order) to the values $d_0, \ldots, d_{s-1}$. A is automatically set to have the same mode as d_0; and A is automatically given a storage reservation of $\underline{n}+\underline{s}$ locations, which is the same as writing DIMENSION $A(\underline{n}+\underline{s}-1)$. **The number $\underline{s}$ must not exceed 200 in any one declaration.**

A may appear in a mode declaration as well, provided it is consistent with the mode of d_0. If A appears in other VECTOR VALUES or DIMENSION declarations the maximum length given or implied for A is used for storage assignment.

(b) VECTOR VALUES A(m) ... A(n) = k

Here m and n are integer constants, with $m \leq n$, and k is any constant (not a sequence of constants). If k is an alphabetic constant, it may not have more than six characters between dollar signs. This statement is treated exactly as in part (a), except that A(m), A(m + 1), ..., A(n - 1), A(n) all are preset with the value k. The storage reservation for A is equivalent to DIMENSION A(n) in this case, and A is set to the mode of k. **Here, also, n - m + 1 must not exceed 200.**

These declarations are useful for presetting dimension vectors and format descriptions. The presetting is done at the time of translation. The constants c_i (or k) are loaded (as part of the translated program) into A. These declarations produce no computation at execution time. However, the values of A may be modified later by other statements in the program during execution.

Vectors which have been assigned to ERASABLE storage (see Section 3.5) may not be preset by a VECTOR VALUES statement. Vectors assigned to PROGRAM COMMON may be preset with constants of any mode; but if statement label or function name values are preset in a multiple-segment ("overlay" or "ping-pong") program, the PROGRAM COMMON region should be identical for all sections.

3.8 PARAMETER declaration

The following declaration may be used to assign values to symbols (called translation parameters) at the time of translation.

PARAMETER $A_1(B_1)$, $A_2(B_2)$, ..., $A_n(B_n)$

Here A_i consists of 1 to 6 letters or digits, the first of which is a letter. B_i is any numeric, alphabetic or Boolean constant, or a symbol consisting of 1 to 6 letters or digits, the first of which is a letter (possibly a translation parameter previously declared).

The effect of the declaration is to permit the replacement of any (subsequent) occurrence of A_i by B_i in the program. For example, A_i may occur in place of the integer constant normally required in a DIMENSION declaration if B_i is an integer constant. Similarly, A_i may occur in front of a single period as a function name if B_i is a symbol. However, A_i may not occur as part of a constant name, so that 3.AB is illegal even if AB is a translation parameter, and A_i may not occur as a defined operator. (Defined operators are discussed in Appendix C.)

Such a declaration is effective at the point at which it occurs in the program, so that normally it would precede any use of A_i. Subsequent PARAMETER declarations supersede declarations of the same translation parameters, so that, in particular,

PARAMETER X(X)

cancels the effect of any previous PARAMETER declaration on X. No PARAMETER substitutions are ever made within any PARAMETER declaration itself. If B_i is itself a parameter, an additional substitution is not performed, allowing, for example, the interchange of two symbols, as in:

PARAMETER B(A), A(B)

Examples are given below of possible uses of translation parameters as declared in the statement:

```
                    PARAMETER A1(77), PI(3.1416), FCN(SIN)
```

```
(a) DIMENSION B(A1*6), C(A1), V(A1)
(b) RADIUS = CIRCUM/2.*PI
(c) Y = FCN.(X)
(d) J = K + A1*B
(e) PAUSE NO. A1
```

These would have exactly the same effect as if the following statements had been used.

```
(a) DIMENSION B(77*6), C(77), V(77)
(b) RADIUS = CIRCUM/2.*3.1416
(c) Y = SIN.(X)
(d) J = K + 77*B
(e) PAUSE NO. 77
```

No mode, dimension, or other effects are implied by a PARAMETER declaration except those implied by the form of B_i as an integer, floating point, or other constant.

A parameter may occur as part of a constant name if it is enclosed in parentheses and used, after the letter M, as a mode designation. The value substituted for the parameter must be one of the integers 0 through 7. Parentheses may also be used with integer constant mode numbers. Thus, the following are legal uses of parameters:

```
PARAMETER N(6), F(0)
B = X .EQ. $ABC$M(N)
VECTOR VALUES A = 1.2M3, 1.5M(5), 4M(N), 1M(F)
```

3.9 Format Variable Declaration

This declaration has the form

$$\text{FORMAT VARIABLE } \mathscr{X}$$

where $\mathscr{X}$ is a list of unsubscripted variable names. If this declaration is embedded within a function definition (see Section 3.10), then none of the names in $\mathscr{X}$ may be dummy variables. All names that will be used as format variables in formats (see Section 2.15.11) must be declared to be format variables in this way. This declaration does not imply anything at all about arithmetic or Boolean mode or about dimension. There may be any number of such declarations, anywhere in the program.

3.10 Function Definitions

There are two main types of functions: the internal function and the external function. Since these are quite similar in many ways, that part of the description which is specific to external functions will be given in subsection 3.10.1, that which is specific to internal functions in subsection 3.10.2, and that which is common to both types will follow in subsection 3.10.3. Throughout this section a single-valued function will be called a "function", and a function with multiple outputs and/or multiple exits will be called a "procedure". For the purposes of this manual a recursive function is one whose definition contains calls for the function being defined--or calls for functions which ultimately call the one being defined. Recursive functions have the same structure as other functions although, in general, they will include statements of the type described in Section 2.13, and certain considerations should be borne in mind when constructing such functions. These considerations arise from the fact that names, instead of values, are used as function arguments. The problem is considered in more detail in the examples of Chapter III.

3.10.1 External Function Definitions

These statements define functions not yet available in the language or as standard package programs ("subroutines"). The designation "EXTERNAL" implies that the statements which follow are to be translated independently of the main program in which they are to be used. (Because of this independence this block of statements is to be considered an entirely separate program, and must have its own DIMENSION and MODE declaration, etc. Names of variables, functions and labels which denote (or "represent") arguments of the function being defined are designated "dummy variables" (or "bound variables"). The modes of these dummy variables (if other than the normal mode), must be declared in the usual way (see Section 3.2), but arrays which are dummy variables must not be dimensioned. The word "EXTERNAL" also implies that names chosen for variables and functions in the current function definition program have no relation whatever with similarly named variables and functions in the main program (or other definition programs), and that no difficulties will be encountered because of the use of similar names. (See also section 4.)

3.10.2 Internal Function Definitions

These statements define functions not yet available in the language or as standard package programs ("subroutines"). The designation "INTERNAL" implies that the definition program which follows is to be translated as part of the main program. The word "INTERNAL" also implies that any variables or functions not listed as dummy variables in the definition of the function (but used in its evaluation), are understood to be the same as elsewhere in the main program, and the current values of these variables and functions will be used. Names of variables, functions, and labels which denote arguments of the function being defined are designated "dummy variables" (or "bound variables"). They must be distinct from those appearing elsewhere in the program. The modes of dummy variables (if other than normal mode) must be declared in the usual way, and arrays which are arguments must not be dimensioned. (See also section 4.)

One-sentence definition:

One form of internal function definition (not available as an external function definition because the latter must be a complete, independent program) is the one-sentence definition, which has the form:

INTERNAL FUNCTION *Name*.(*List*) = *E*

where *Name* is the name of the function being defined and *E* is an expression (arithmetic or Boolean) involving the variables in the *List* of dummy variables.

Example:

```
INTERNAL FUNCTION SUMSQ.(X, Y, Z) = X*X + Y*Y + Z*Z - T*T
```

As indicated above, X, Y, and Z, as they occur here, are dummy variables, and (X, Y, Z) is the dummy variable list. The current value of T, however, will be obtained and used each time the value of the function is needed. An example of the use of the function so defined would be:

```
A = 1.-SUMSQ.(U, V + 3, W) .P..5
```

In the one-sentence internal function definition, at least one dummy variable must be indicated, even if the function does not use arguments. For example:

```
INTERNAL FUNCTION F. = 2*Y + 1
```

is <u>not</u> a legal definition, but

```
INTERNAL FUNCTION F.(X) = 2*Y + 1
```

is.

Only nonsubscripted names of variables (either individual or array) or names of functions (without arguments) may appear in the dummy variable list. In the use of the function in an expression, the arguments may be any expressions that agree in mode with the corresponding dummy variable in the declaration.

The modes of dummy variables and "actual" arguments must correspond. Thus, in the example definition

```
INTERNAL FUNCTION POLY. (N, X, FN.) = FN.(J*X).P.N - X/XBAR
```

which might be used in the statement

```
BETA    ZQ = POLY. (M + 1, Y, SIN.) + POLY.(M - 1, Z, COS.)
```

it is understood that if N is in the integer mode, then so is M, and if X is in the floating point mode, then so are Y and Z. It is, of course, presumed that both M and N have been declared to be in the integer mode. Similarly, the values of SIN. and COS. must be the same mode as the values of FN. Moreover, in the use of functions this mode correspondence cannot be checked by the translator.

The function POLY has as one of its arguments the name of a function. In the statement BETA the function used in the first term to the right of the "=" sign is SIN and in the second term COS . Hence statement BETA is then equivalent to:

```
BETA    ZQ = SIN.(J*Y).P.(M+1) - Y/XBAR + COS.(J*Z).P.(M-1) - Z/XBAR
```

3.10.3 <u>Internal and External Functions (Things they have in common)</u>

Each function definition (except one-sentence definitions described in 3.10.2) may define any number of functions and/or any number of procedures. However, within one function definition all functions and procedures defined must use exactly the same set of dummy variables. In other words, the functions CPADD.(X,Y,A,B) and CPMPY.(X,Y,A,B) may be defined, if desired, by one definition, but the functions SIN.(X) and ARCTAN.(X,Y) would require separate definitions. Similarly, the procedures RKSUB and ADAMS could be defined by the same definition if they have the same outputs, say Z and W, and the same input parameters, say X, Y, T, and N. However, if they did not have exactly the same outputs and the same inputs, they would require separate declarations.

In the use of a function (i.e., the call for it) the arguments may be constants, variables, function names, labels, or expressions. However, if one of the arguments appears to the left of an " = " sign in an assignment statement in the defining program it is not meaningful to use a constant or an expression for that argument in the call. As mentioned earlier, the arguments cannot be checked for correspondence in mode and number to dummy variables.

An example function definition program is as follows:

```
INTERNAL FUNCTION COS.(X)      or alternatively,
...                              INTERNAL FUNCTION (X)
...                              ENTRY TO COS.
...
ENTRY TO SIN.
...
...
...
FUNCTION RETURN ALPHA + J - 3.
ENTRY TO TAN.
...
...
...
FUNCTION RETURN BETA/K5 - 4.* D
END OF FUNCTION
```

The first statement (INTERNAL FUNCTION COS.(X)) is a function declaration (i.e., declares that the following statements define a function called COS whose entry point is here). The opening declaration and entry point may alternatively be split into two statements as shown. If this is defining an external function, the declaration would be EXTERNAL FUNCTION COS.(X). Following the words INTERNAL (or EXTERNAL) FUNCTION is the dummy variable list ((X) in this example). The END OF FUNCTION declaration is the last statement in the function definition program. (In an EXTERNAL function definition this is also the last statement in the program.) An entry must be provided for each function being defined, but several functions may share any number of FUNCTION RETURN statements. An entry statement merely marks a point of entry, and does not affect the sequence of computation in any way. The expression after the phrase "FUNCTION RETURN" indicates that on this return the value of the function is to be the value of that expression. This expression must agree in mode with the function whose value it supplies, i.e., it must agree with the expected mode of the function value being called for in the calling program. This agreement is not checked. The definitions of functions whose calls are intended to be included in expressions must have an expression following the FUNCTION RETURN statement. If the calls for a function are to appear in an EXECUTE statement (generally such functions have multiple outputs) the FUNCTION RETURN statement may appear without an accompanying expression.

Example of a procedure (called PROC) definition

```
EXTERNAL FUNCTION PROC. (M,N,I,P,Q)
STATEMENT LABEL N,I
...
...
...
FUNCTION RETURN
...
...
...
WHENEVER B, TRANSFER TO I
...
...
...
TRANSFER TO N
END OF FUNCTION
```

In this example, N and I are actually alternate exits, and B represents some Boolean expression.

It is important to note that internal function definitions of any kind whatever (including the single statement definition of subsection 3.10.2) may occur anywhere in the program, except within another internal function definition. Internal function definitions may occur within external function definitions, however. External function definitions may not occur within any other programs, not even within other external function definitions. Each external function definition must be a complete, self-contained program.

Example of a function definition (called INVSF)

The following is an example of a function whose value is $1/x$ if $0 < x \leq 1$ and $1/x^2$ if $x > 1$. If $x \leq 0$, one obtains an error return (see Section 2.9).

```
A   EXTERNAL FUNCTION (X)
J   ENTRY TO INVSF.
G   WHENEVER X.G.0. .AND. X .LE. 1.
C      FUNCTION RETURN X .P. -1
H   OR WHENEVER X .G. 1.
D      FUNCTION RETURN X .P. -2
I   OTHERWISE
E      ERROR RETURN
K   END OF CONDITIONAL
B   END OF FUNCTION
```

(Here the statements are all labeled only for reference in what follows.)

The list of dummy variables in the opening declaration (statement A in the preceding paragraph) may contain only nonsubscripted variable names (either individual or array) or function names (without arguments). Within the definition program itself (the statements between statement A and statement B), a function name will usually occur with arguments, and an array variable will usually occur with subscripts.

A few comments about the last example: This definition program defines a single-valued function of X, called INVSF . Since no mode declaration is given it is assumed by the translator that X is floating point. The value of INVSF.(X) is computed by the use of a compound conditional. If $0 < X \leq 1$, (statement G) then statement C is executed, causing a return to the calling program with the value $\frac{1}{X}$. If the condition $0 < X \leq 1$ is not true, then the condition $X > 1$ is tested (statement H). If $X > 1$, statement D is executed. Finally, if neither of the conditions $0 < X \leq 1$ or $X > 1$ is true, then statement I finds that $X \leq 0$ and statement E (the ERROR RETURN) is executed.

Suppose

```
A    = B - D
X    = T(I) + INVSF.(Y) * T(I-1)
Y(I) = Z + R(J) * 2.5
```

is part of a program which calls on INVSF , and suppose the error return statement is executed during the evaluation of INVSF.(Y) (i.e., $Y \leq 0$). Then control is returned to the system in which the translated program is embedded, with an error flag set. However, suppose instead these statements:

```
    A    = B - D
F   Z    = T(I) + INVSF.(Y,ER) * T(I-1)
S   Y(I) = Z + R(J) * 2.5
    ...
    ...
    ...
ER  Z    = 0
 L  Y(I) = 1.
```

are part of the calling program and $Y \leq 0$. When the ERROR RETURN statement is executed, control transfers to statement ER (then goes on to L), instead of finishing the execution of statement F (and then going on to S). Note that the END OF FUNCTION statement will never be executed, but must be present in the definition. Indeed, the only type of function definition in which the statement END OF FUNCTION is not required (in fact, is not allowed!) is the one-sentence internal function (Section 3.10.2).

There is a table kept by the MAD translator in which are recorded all occurrences of dummy variables (or parameters) within an external or internal function definition. Since all such occurrences must be initialized on each entry to the subroutine, it is very desirable that the number of entries in this table be kept as small as possible. This is especially important when a diagnostic comment is generated saying that this table has in fact overflowed! (The diagnostic comment produced in this case is PARAMETER USE TABLE EXCEEDED.)

There are several ways to cut down the entries in this table and thus speed up the execution of the subroutine and shorten its length as well. (1) If X is an input to the subroutine and its value is used several times, start off the subroutine's computation with the assignment statement Y = X, and use Y everywhere as an auxiliary variable instead of X. Only this assignment statement will then need to be initialized with the address of X. (2) If the address of a variable is needed, as in the case of an output argument, or an argument which is an array name, one cannot use the method just given in (1). One can instead put the variable or array in PROGRAM COMMON by means of (identical) PROGRAM COMMON declarations in both the main program and the subroutine. Then the variable or array should not be used as an argument at all, and the initialization is thus avoided. Note that in this case (identical) DIMENSION declarations are necessary for such arrays in both the main program and the subroutine.

3.10.4 TRANSMIT statements (General familiarity with the contents of the IBM 7090 reference manual is assumed)

A "calling sequence" (see Chapter IV, Section 3) is a process of establishing and transmitting (i.e., making available to the called program) the values of input argument expressions and/or the addresses of input or output arguments. These sequences (instructions in machine code) are generated by the MAD translator in the calling program as a consequence of each call on a function or procedure.

Calling sequences for input-output subroutines ordinarily differ from calling sequences of other subroutines, in that the former use the STR instruction (IOP and FMT in UMAP) while the latter use the TXH and TIX instructions (PAR and BLK, respectively, in UMAP). The use of the STR allows computation by the called subroutine after the processing (evaluation) of one argument and before the next is processed. This occurs in the role of N in the example below:

```
READ FORMAT $I4*$, N, A(I,J), ..., A(I,N)
```

where the new value of N is used to determine A(I,N). Four types of TRANSMIT statements allow other subroutines (i.e., EXTERNAL functions) to have the input-output type of calling sequence:

TRANSMIT LIST $\mathcal{N}$., $\mathcal{L}$

TRANSMIT TAPE LIST $\mathcal{N}$., $\mathcal{T}$, $\mathcal{L}$

TRANSMIT FORMAT LIST $\mathcal{N}$., $\mathcal{F}$, $\mathcal{L}$

TRANSMIT TAPE FORMAT LIST $\mathcal{N}$., $\mathcal{T}$, $\mathcal{F}$, $\mathcal{L}$

where $\mathcal{N}$ is the name of an INTERNAL or EXTERNAL FUNCTION. $\mathcal{L}$ is any input or output list, $\mathcal{T}$ is a logical tape number, and $\mathcal{F}$ is any format designation, such as the name of a vector preset to format information, or format information itself. Note that since the STR instruction causes a transfer of control to location 00002, the subroutine SET2 in the UM Executive System Library will probably be useful. The action of each of the above statements is to call the function named as $\mathcal{N}$ with an input-output calling sequence based on $\mathcal{T}$, $\mathcal{F}$, and/or $\mathcal{L}$. (See Chapter IV, section 3.)

3.11 SYMBOL TABLE statements

The MAD translator produces a "symbol table" as part of the object program whenever certain statements occur in the program. The symbol table contains symbols used in the source program, and associates with each symbol an "information word", described below. This table makes it possible, for example, for a symbol appearing on a data card (and which is read by a READ DATA statement) to be assigned a storage location and a mode (i.e., integer, floating point, etc.) at the time the data card is read. The information word has the form:

A	0	0	DV	M	Add
S	1	2	3 17	18 20	21 35

where A is a 1 if the associated symbol is an array (i.e., it was dimensioned) and 0 otherwise; DV is 0 if the symbol has no dimension vector (i.e., is not an array), and the relocatable address of the dimension vector if there is one; M is the mode of the symbol, such as 0 for floating point, 1 for integer, etc.; and Add is the (relocatable) address of the symbol. The concept of the "dimension vector" is developed in Appendix B.

The table is stored as an ordinary MAD vector, i.e., the top of the table would correspond to V(0) if the vector were named V. (Actually the symbol .SYMTB is used internally for this table, and this will appear in the symbol table). Assuming such a name, the table has the form:

V(0) = integer N giving highest subscript used in the table;
i.e., if there were 15 symbols, or 30 words, then N = 30.

V(1) = information word for symbol in V(2).
V(2) = symbol associated with word in V(1).

⋮

V(N-1) = information word for symbol in V(N).
V(N) = symbol associated with word in V(N-1).

All symbols are in the form of left-justified alphabetic constants (with trailing blanks).

The symbol table produced by the translator does not always contain every symbol used in the program. If any of the (simple I/O) statements

READ DATA

READ AND PRINT DATA

PRINT RESULTS $\mathcal{L}$

PRINT BCD RESULTS $\mathcal{L}$

PRINT OCTAL RESULTS $\mathcal{L}$

occurs anywhere in the program, then the symbol table will be "full", i.e., it will contain all symbols in the program (plus two additional symbols, as described below). If none of these statements occurs, but a FORMAT VARIABLE declaration occurred, then a "partial" symbol table is produced containing only those variables declared to be format variables (plus the two additional symbols). If neither a full nor partial table is produced, then a trivial one-word table is generated, with V(0) = 0.

A non-trivial table (V(0) ≠ 0) has two word entries whose format is described above, and in addition has two entries appended at the high-index (i.e., V(N-3), ..., V(N)) end of the symbol table vector. Both of these entries have BCD symbols of "□ ... □□", i.e., in V(N-2) and V(N). The address in V(N-3) is the address of the first location above the transfer vector in the MAD object program. This location is the one assigned to all variables which have a single occurrence in programs not using simple I/O. In the other added entry at location V(N-1), there is the address which is the first one below the ERASABLE region that has been assigned in the object program. The entire table is not ordered in any directly usable way and may be re-ordered at execution without affecting input-output operations.

In order to allow interrogation and/or modification of the symbol table during the execution phase of a program two declarations are provided.

SYMBOL TABLE VECTOR $\mathcal{V}$

FULL SYMBOL TABLE VECTOR $\mathcal{V}$

where $\mathcal{V}$ is a <u>non-subscripted</u> variable. The first treats the symbol table as preset values of $\mathcal{V}$, gives $\mathcal{V}$ itself integer mode, and dimensions $\mathcal{V}$ large enough to take the particular kind of table (i.e., full, partial, or trivial) that is being produced. The second declaration does all of this and forces a full table in addition. The variable $\mathcal{V}$ may be declared to be in PROGRAM COMMON, and it may be dimensioned larger than necessary, if desired. It should <u>not</u> be declared to be in ERASABLE.

3.12 <u>LISTING ON and LISTING OFF declarations</u>

The form of these declarations is as follows:

LISTING ON

LISTING OFF

At the beginning of each MAD translation, LISTING ON will be automatically in effect. If LISTING OFF occurs, that statement will be printed; but statements that come after it, up to and including any LISTING ON declaration, will not print. They will print with the object program if $PRINT OBJECT is requested. Note that UMAP-like code used in DEFINE sequences (see Appendix C) does not print with the object code in any case.

3.13 REFERENCES ON and REFERENCES OFF declarations

The form of these declarations is as follows:

REFERENCES ON

REFERENCES OFF

At the beginning of every MAD translation an implicit REFERENCES OFF condition will be in effect. After an occurrence of REFERENCES ON, references to symbols which are encountered in the program will be collected until an occurrence of REFERENCES OFF. These declarations may be used as often as necessary. If any references have been collected, a list of references to (a) symbols, (b) function names, (c) dummy variables of INTERNAL FUNCTIONS, (d) dummy variables of EXTERNAL FUNCTIONS, and (e) PARAMETERS will be printed after part I of a MAD translation (even if there are errors).

References are given in terms of the * number assigned to each statement by the translator. In case a statement contains a part I error,† some references for that statement may be included, but some may not be included.

Each variable listed as occurring only once will be associated at that time with an number to make it easier to find. Since variables are often dimensioned (and/or declared to be in PROGRAM COMMON or ERASABLE or equivalent to something which is) to create specific storage assignments and are not used again in the program, no symbol is listed as occurring only once which appears in a PROGRAM COMMON, ERASABLE, or EQUIVALENCE declaration, or which is followed by dimension information in any declaration. The list of variables occurring only once does not appear if there are any part I errors.

4. Restrictions

Variable names appearing as dummy variables in any function declaration may not appear in a PROGRAM COMMON, ERASABLE, or EQUIVALENCE declaration.

The name of a function with the period deleted must not be used as the name of a variable or as a statement label. A statement label must not be identical with any variable name.

The object program produced by the translator automatically calls on the subroutines SYSTEM and ERROR , hence these names may not be used as variable names or statement labels in a MAD program. They may be used as function names only when referring to these subroutines. (These subroutines will be described in Chapter IV.)

If a block designation $A(i_1, \ldots, i_n) \ldots A(j_1, \ldots, j_n)$ is used in an input-output list $\mathcal{L}$ for reading or writing binary tape, the linear subscript corresponding to $i_1, \ldots, i_n$ may not be greater than the linear subscript corresponding to $j_1, \ldots, j_n$. While the list for a binary tape statement is transmitted in the sequence written, a block $A(i_1, \ldots, i_n) \ldots A(j_1, \ldots, j_n)$ within the list is actually transmitted in reverse order, i.e., in the sequence $A(j_1, \ldots, j_n)$ to $A(i_1, \ldots, i_n)$.

†See Chapter IV, section 2.

Chapter III

EXAMPLES

"He prepares to go mad with fixed rule and method."

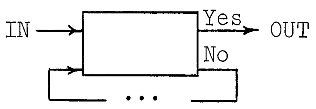
Horace: Satires

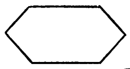
Note: The following illustrates how some programs may be written in the MAD language. Since they were written to illustrate as many features of the language as possible, they are not necessarily the most efficient or elegant programs which could have been written. They have all been tested on the computer, however; and they are correct.

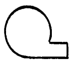
The flow chart notation used here may vary somewhat from the notation used by others. This variation reflects present computing practice where many individualistic forms are encountered and, moreover, causes no difficulty due to the essentially graphic nature of charts.

In addition to the conventions given in Chapter I, the following are used in this chapter:

(a) THROUGH $\mathcal{S}$, FOR VALUES OF $\mathcal{V} = \mathcal{L}$ is given by a box with two outlets: One used when the list is not satisfied, and one used when the list is satisfied.



(b) A pause is represented by a hexagon. 

(c) Operations involving tapes are indicated by: 

1. Scientific Examples

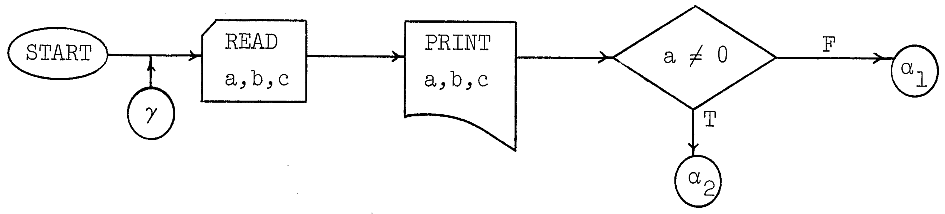
Example 1

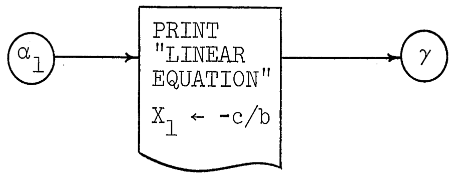
Problem: To solve the quadratic equation $ax^2 + bx + c = 0$ for various sets of coefficients a, b, and c.

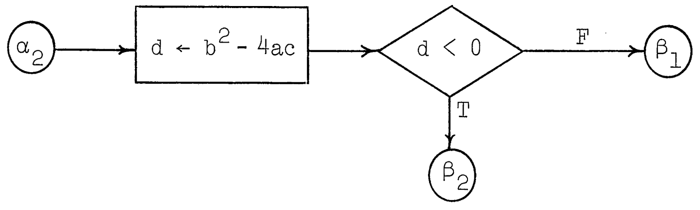
Analysis: Let x_1 and x_2 be the two roots of the equation. Then their values are found by the formulas,

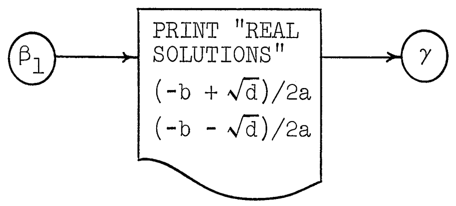
$$x_1 = \frac{-b + \sqrt{b^2 - 4ac}}{2a} \qquad x_2 = \frac{-b - \sqrt{b^2 - 4ac}}{2a}$$

whenever $a \neq 0$. The single root x_1 of the equation when $a = 0$ is $x_1 = -c/b$. The input values of a, b, and c are printed immediately after they are brought in to help in finding trouble spots during the development of the program (not as necessary here as in longer problems, but a good idea!). Assume $b \neq 0$ if $a = 0$.

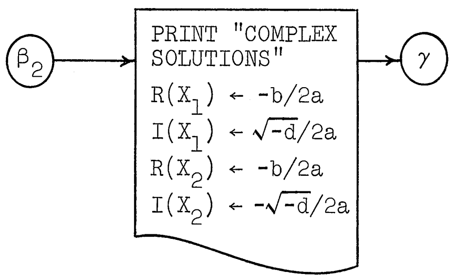








Note: $R(X_1)$ and $I(X_1)$ are the real and imaginary parts of X_1, and similarly, $R(X_2)$ and $I(X_2)$ are the real and imaginary parts of X_2.


β_2
PRINT "COMPLEX SOLUTIONS"
$R(X_1) \leftarrow -b/2a$
$I(X_1) \leftarrow \sqrt{-d}/2a$
$R(X_2) \leftarrow -b/2a$
$I(X_2) \leftarrow -\sqrt{-d}/2a$
γ

The Program:

```
            R
            R MAIN PROGRAM
            R
GAMMA        READ FORMAT $3F10.4*$, A, B, C
             PRINT RESULTS A, B, C
             WHENEVER A .NE. 0,TRANSFER TO ALPHA2
ALPHA1       PRINT FORMAT LINEAR,-C/B
             TRANSFER TO GAMMA
ALPHA2       D = B .P.2 -4.*A*C
             WHENEVER D .L. 0.,TRANSFER TO BETA2
BETA1        PRINT FORMAT REAL,(-B+SQRT.(D))/(2.*A),(-B-SQRT.(D))/(2.*A)
             TRANSFER TO GAMMA
BETA2        PRINT FORMAT COMPLX,-B/(2.*A),SQRT.(-D)/(2.*A),
            1 -B/(2.*A),-SQRT.(-D)/(2.*A)
             TRANSFER TO GAMMA
            R
            R FORMAT SPECIFICATIONS
            R
             VECTOR VALUES LINEAR = $21H0LINEAR EQUATION, X = F10.4*$
             VECTOR VALUES REAL = $21H0REAL SOLUTIONS, X1 =
            1 F10.4,S8,4HX2 = F10.4*$
             VECTOR VALUES COMPLX = $19H0COMPLEX SOLUTIONS,
            1 S4,7HR(X1) = F10.4,S8,7HI(X1) = F10.4,S8,
            2 7HR(X2) = F10.4,S8,7HI(X2) = F10.4*$
             END OF PROGRAM
$DATA
          4.        -8.         4.
          0.         5.        10.
          1.         1.         1.
```

Example 2

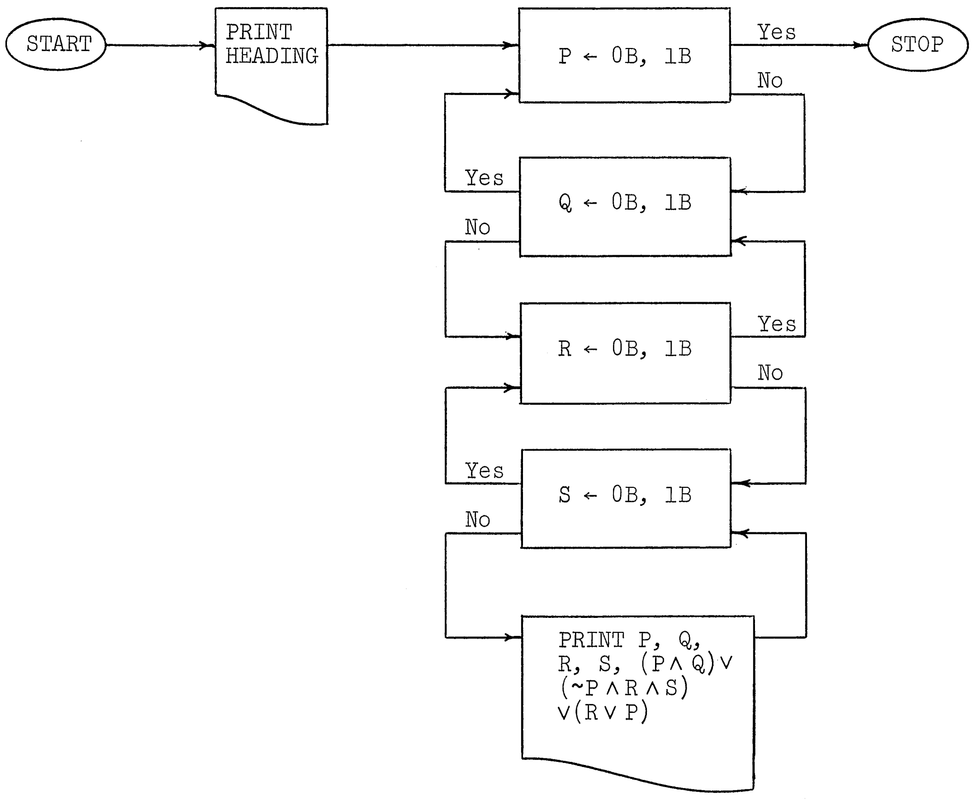
Problem: A logical (Boolean) expression such as

T = (P .AND. Q) .OR. (.NOT. P .AND. R .AND. S) .OR. (R .OR. P)

will have a value TRUE or FALSE (represented here by 1B and 0B, respectively) depending on the "input values" of the variables involved: P, Q, R, S. Thus, if P = 1B, Q = R = S = 0B, then the total expression T will have the value 1B. The entire table of outputs for all possible inputs would be as follows:

P	Q	R	S	T
0B	0B	0B	0B	0B
0B	0B	0B	1B	0B
0B	0B	1B	0B	1B
0B	0B	1B	1B	1B
0B	1B	0B	0B	0B
0B	1B	0B	1B	0B
0B	1B	1B	0B	1B
0B	1B	1B	1B	1B
1B	0B	0B	0B	1B
1B	0B	0B	1B	1B
1B	0B	1B	0B	1B
1B	0B	1B	1B	1B
1B	1B	0B	0B	1B
1B	1B	0B	1B	1B
1B	1B	1B	0B	1B
1B	1B	1B	1B	1B

The problem is to write a program to generate the entire "truth table" for the given expression T.



<u>The Program</u>:

```
    PARAMETER TRUE(0B), FALSE(1B)
    PRINT COMMENT $1TRUTH TABLE FOR THE FUNCTION$
    PRINT COMMENT$0(P.AND.Q).OR.(.NOT.P.AND.R.AND.S).OR.(R.OR.P)$
    PRINT FORMAT HEADER
    BOOLEAN P,Q,R,S
    THROUGH A,FOR VALUES OF P = FALSE, TRUE
    THROUGH A,FOR VALUES OF Q = FALSE, 1B
    THROUGH A,FOR VALUES OF R = 0B,TRUE
    THROUGH A,FOR VALUES OF S = 0B,1B
    PRINT FORMAT TABLE,P,Q,R,S,(P .AND. Q) .OR.(.NOT. P .AND. R
   1 .AND.S).OR.(R .OR.P)
    VECTOR VALUES HEADER = $1H1,S10,1HP,S10,1HQ,S10,1HR,S10,1HS,
   1 S15,1HT*$
    VECTOR VALUES TABLE = $1H0,4(S10,I1),S15,I1*$
    END OF PROGRAM
```

<u>Note</u>: Although it would have meant only a slight change in the format information, no attempt was made here to label the "0" and "1" that print as values in the table as Boolean, i.e., "0B" and "1B". This points up the fact that internally 0B and 1B are stored as 0 and 1, respectively. Also, the statement

NORMAL MODE IS BOOLEAN

could have been used as the fifth statement of this program instead of the BOOLEAN declaration.

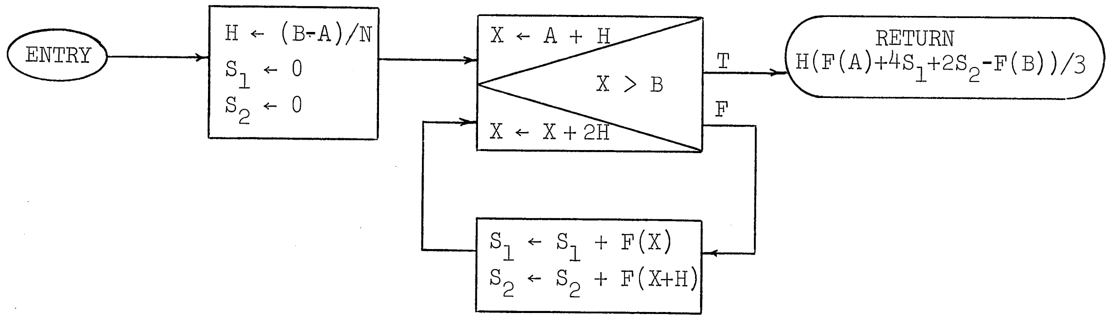
Example 3

Problem: To approximate $\int_a^b f(x)$ by Simpson's Rule, for an arbitrary interval $[a, b]$ using N equal subintervals (where N is an arbitrary even integer and $a < b$).

Analysis: By Simpson's Rule, $\int_a^b f(x)dx \approx \frac{b-a}{3N}(y_0 + 4y_1 + 2y_2 + 4y_3 + \dots + 4y_{N-1} + y_N)$, where $y_i = f(x_i)$, and $a = x_0, x_1, \dots, x_N = b$ are the partition points of the interval $[a, b]$.

Method: We shall write the program in the form of an external function, so that it could be used with any other program. The evaluation of $f(x)$ may be accomplished by another external function or an internal function.

Flow Diagram:



The Program:

```
              EXTERNAL FUNCTION SIMPS. (A,B,N,F.)
              INTEGER N
              H = (B-A)/N
              S1 = 0.
              S2 = 0.
              THROUGH ALPHA,FOR X = A+H, 2.*H, X .G. B
              S1 = S1 + F.(X)
ALPHA         S2 = S2 + F.(X+H)
              FUNCTION RETURN H*(F.(A)+4.*S1+2.*S2-F.(B))/3.
              END OF FUNCTION
```

If, for some reason, the integral of $\sin 3x - \cos(rx + 1)$ were needed if $0 \le r \le 3$, and the integral of $\sin 3x - \cos rx$ otherwise, the program might then be as follows:

The Program:

```
READ      READ FORMAT $2F12.4,I6,F12.4*$,A,B,N,R
          INTEGER N
          WHENEVER 0. .LE. R .AND. R .LE. 3.
          PRINT FORMAT RESULT,A,B,N,R,SIMPS.(A,B,N,F1.)
          OTHERWISE
          PRINT FORMAT RESULT,A,B,N,R,SIMPS.(A,B,N,F2.)
          END OF CONDITIONAL
          TRANSFER TO READ
         R
         R DEFINITION OF FUNCTIONS
         R
          INTERNAL FUNCTION F1.(X) = SIN.(3.*X)-COS.(R*X+1.)
          INTERNAL FUNCTION F2.(X) = SIN.(3.*X)-COS.(R*X)
         R
         R FORMAT SPECIFICATIONS
         R
          VECTOR VALUES RESULT = $23H1 FOR THE INTERVAL FROM
         1 F12.4,3H TO F12.4,5H WITH I6,38H EQUAL SUB-INTERVALS AND
         2 PARAMETER F12.4/ H*0THE VALUE OF THE INTEGRAL IS F12.4**$
          END OF PROGRAM
```

External Function:

```
          EXTERNAL FUNCTION (A,B,N,F.)
          INTEGER N
          ENTRY TO SIMPS.
          H = (B-A)/N
          S1 = 0.
          S2 = 0.
          THROUGH ALPHA, FOR X = A+H,2.*H, X .G. B
          S1 = S1+F.(X)
ALPHA     S2 = S2+F.(X+H)
          FUNCTION RETURN H*(F.(A)+4.*S1+2.*S2-F.(B))/3.
          END OF FUNCTION
```

Data:

```
          0           2.     10          10.
```

An alternate way to write the first eight lines of this program, illustrating one use of the FUNCTION NAME mode, and using full abbreviations (see Appendix A), would be:

```
          I'R N
READ      R'T $2F12.4,I6,F12.4*$,A,B,N,R
          W'R 0. .LE. R .AND. R .LE. 3.
             S=F1.
          O'E
             S=F2.
          E'L
          P'T RESULT, A, B, N, R, SIMPS.(A,B,N,S)
          T'O READ
          FUNCTION NAME S
```

Example 4

Problem: To find one real solution (if it exists) of the equation $f(x) = 0$ (where f is a continuous function) on an arbitrary interval $[a, b]$, provided the roots (if there are more than one) are at least ϵ apart.

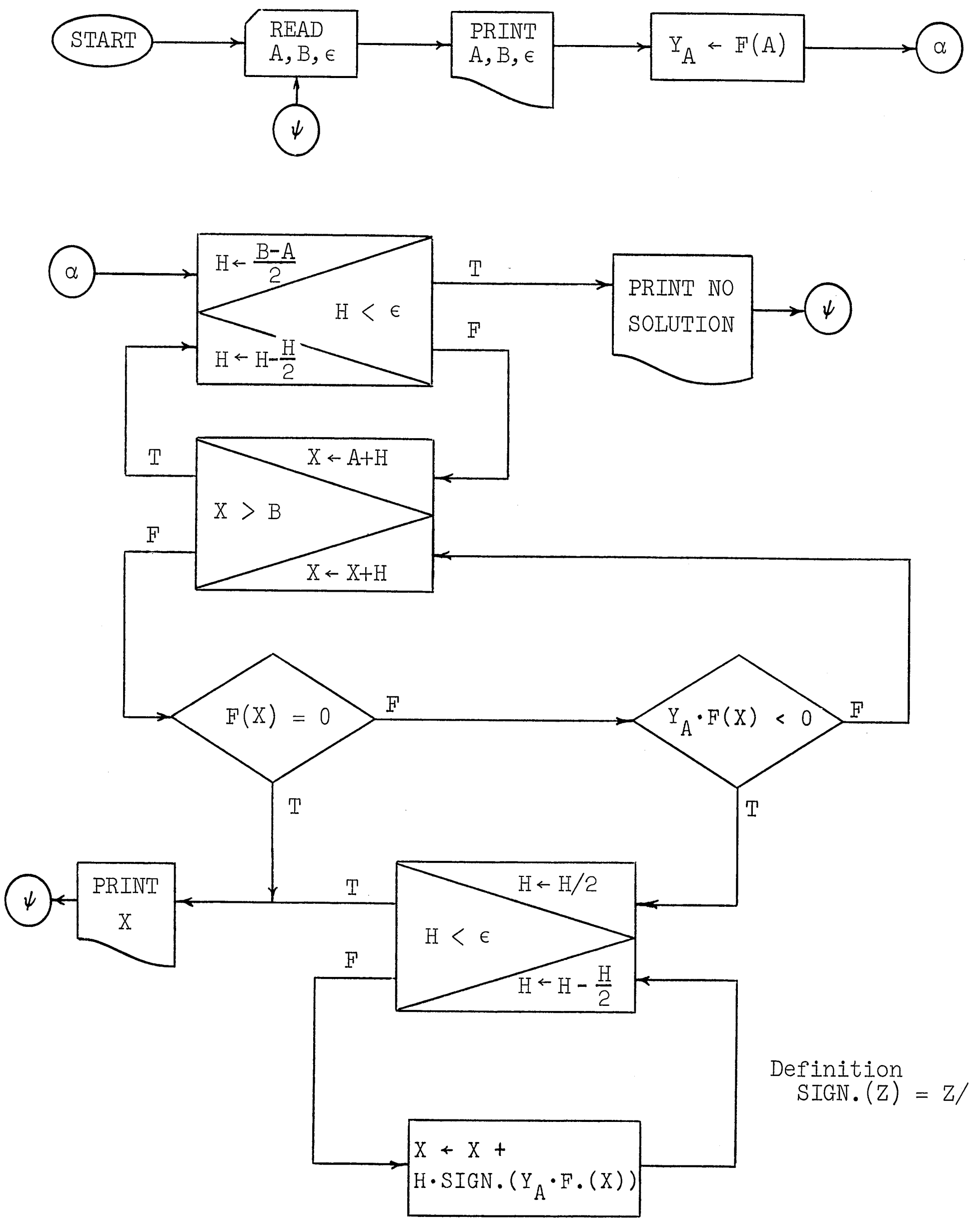
Analysis: We specify a, b, and ϵ as parameters. The method used will be "half-interval convergence," in which the function is evaluated at $x = a$, and then the interval is scanned for a change of sign[†] in the value of $f(x)$. If no change of sign is found, the scanning is repeated with a step size for searching equal to one-half the previous step size. If the step size becomes smaller than ϵ, and no change of sign is found, the process is terminated, and comment is printed: "NO SOLUTION".

If a change of sign is found between x_L and x_R, the value of f is computed at $x_M = \frac{x_L + x_R}{2}$, i.e., the midpoint of the interval of uncertainty $[x_L, x_R]$. We then determine which of the intervals $[x_L, x_M]$, $[x_M, x_R]$ now contains a change in sign. We then compute the value of f at midpoint of that smaller interval, etc., until the interval being considered finally has length less than ϵ, at which time either end may be taken as the solution with an error less than ϵ.

The method used here to handle the x_M computation is perhaps not the most obvious one. It consists of a simple loop in which the value of x is adjusted by $h' = \frac{h}{2}$, then $h'' = \frac{h'}{2} = \frac{h}{4}$, etc., until h is small enough. The adjustment of x is either to the left or right, depending on the occurrence or non-occurrence, respectively, of a change of sign between $f(a)$ and $f(x)$.

It should be understood that this method may not find a root which is one of a pair of roots which either coincide or are less than ϵ apart.

[†]A change of sign is detected when the numbers involved have a negative product.



Definition
SIGN.(Z) = Z/|Z|

Program: (It is assumed here that f (referred to as F in the program) will be defined as an internal function.)

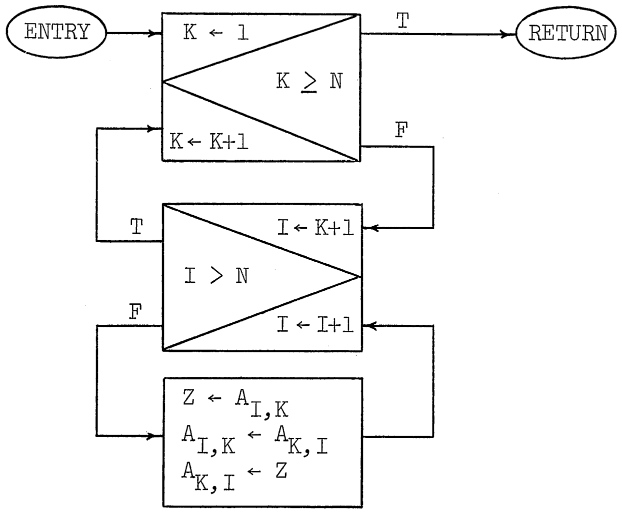
```
               REFERENCES ON

               INTERNAL FUNCTION F.(Z)= Z .P. 2 - 2.
               PRINT COMMENT$ ECHO CHECK OF VALUES FOR A,B, AND EPS$
PSI            READ AND PRINT DATA
               YA = F.(A)
               THROUGH ALPHA,FOR H = (B-A)/2.,-H/2.,H .L. EPS
               THROUGH ALPHA,FOR X = A+H,H, X .G. B
               WHENEVER F.(X) .E. 0.,TRANSFER TO ETA
ALPHA          WHENEVER YA*F.(X) .L. 0.,TRANSFER TO DELTA
               PRINT COMMENT $0 NO SOLUTIONS$
               TRANSFER TO PSI
              R
              R THE NEXT SECTION IS ENTERED WHEN A CHANGE
              R OF SIGN IS FOUND
              R
DELTA          THROUGH SIGMA,FOR H=H/2.,-H/2.,H .L.EPS
SIGMA          X = X+SIGN.(YA*F.(X))*H
ETA            PRINT COMMENT $0 SOLUTION$
               PRINT RESULTS X
               TRANSFER TO PSI
              R
              R DEFINITION OF SIGN. FUNCTION
              R
               INTERNAL FUNCTION SIGN.(Z) = Z/.ABS.Z
               END OF PROGRAM
$DATA
A = 1., B = 2., EPS = .01        *
```

Example 5

Problem: Find the transpose A' of an n × n matrix $A = (a_{ij})$.

Analysis: If we write $A' = (b_{ij})$, then $b_{ij} = a_{ji}$. We shall interchange symmetrically placed pairs of elements, leaving untouched elements on the main diagonal. The program will be in the form of an external function.

Flow Diagram:



```
           EXTERNAL FUNCTION TRANS. (A, N)
           (K= 1,1,K.GE.N,(I=K+1,1, I .G. N,
          1 Z = A(I,K), A(I,K) = A(K,I), A(K,I) = Z))
          R  THE ABOVE IS AN ITERATED STATEMENT. SEE SECTION 2.17
           FUNCTION RETURN
           INTEGER N,K,I
           END OF FUNCTION
```

Note: No dimension information is given for A, since it is an argument in a function definition program. This function would be called in a statement of the form
TRANS. (A,N).

```
           DIMENSION A(25*25)
           INTEGER N
           FORMAT VARIABLE N
          RTHE FOLLOWING STATEMENT READS IN THE
          RVALUES OF N AND THE MATRIX A, PROVIDING
          RTHEY ARE SO IDENTIFIED ON THE DATA CARDS,
 READ      READ DATA
          R  THE FOLLOWING STATEMENT RESETS THE DIMENSION
          R  INFORMATION FOR THE  A ARRAY.
                   SETDIM. (A,N,N)
                   TRANS.(A,N)
           PRINT FORMAT MATRIX, A(1,1)...A(N,N)
           TRANSFER TO READ
           VECTOR VALUES MATRIX = $1HO, 'N' F12.6*$
           END OF PROGRAM
```

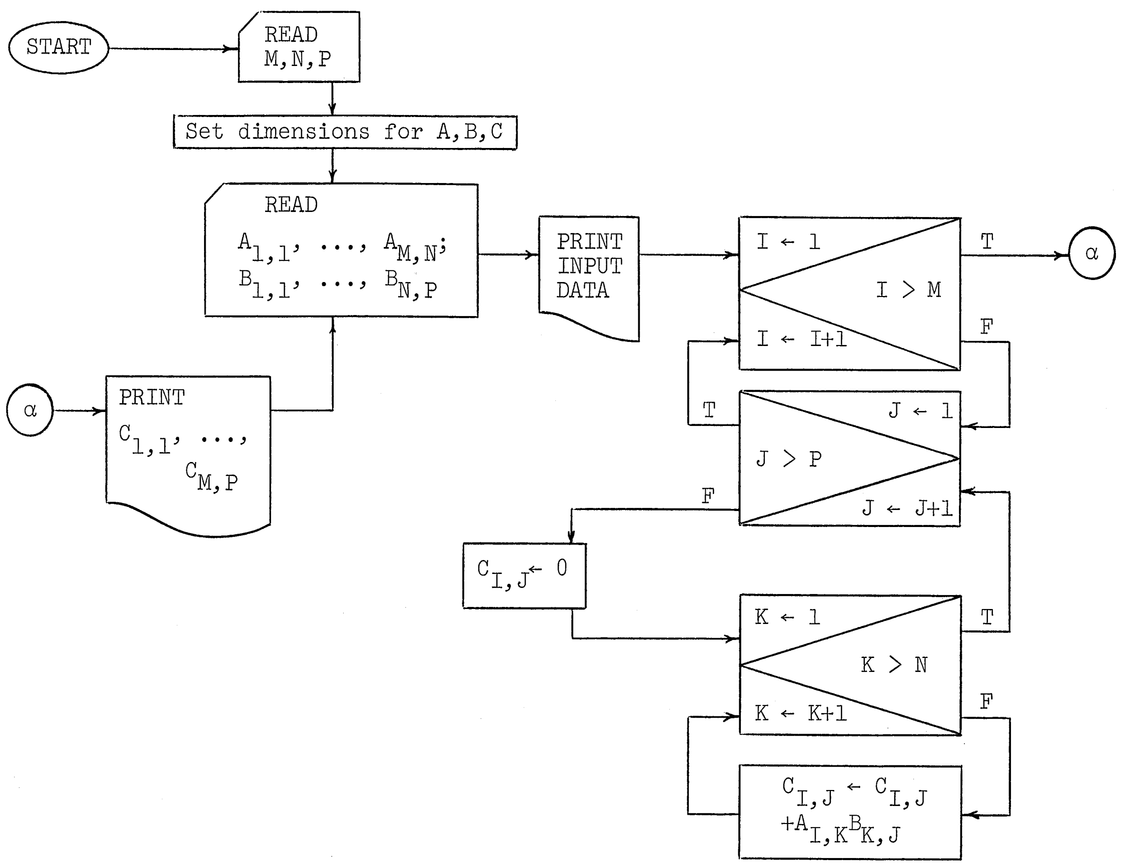
Note that the use of the format variable N (see Section 2.15.11) allows the printing of the matrix by rows, with each row having the correct number of columns. If the value of N exceeds 10, however, a format error will occur, since the format will be trying to specify more than 132 characters per line.

Example 6

Problem: Multiply the matrix $A = (a_{ij})$ by the matrix $B = (b_{ij})$ to produce the matrix $C = (c_{ij})$, i.e., $C = A \cdot B$. Assume that A has dimensions $m \times n$ with $m \cdot n \leq 1500$, B has dimensions $n \times p$, with $n \cdot p \leq 1500$, and C has dimensions $m \times p$, with $m \cdot p \leq 1500$.

Analysis: An element c_{ij} of C is computed by the formula

$$c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}$$


START
READ M,N,P
Set dimensions for A,B,C
READ $A_{1,1}, \ldots, A_{M,N}$; $B_{1,1}, \ldots, B_{N,P}$
PRINT INPUT DATA
I ← 1
I > M
I ← I+1
T
F
α
PRINT $C_{1,1}, \ldots, C_{M,P}$
J ← 1
J > P
J ← J+1
$C_{I,J} \leftarrow 0$
K ← 1
K > N
K ← K+1
$C_{I,J} \leftarrow C_{I,J} + A_{I,K}B_{K,J}$

```
               DIMENSION A(40*40), B(40*40), C(40*40)
               INTEGER I,J,K, M, N, P
              R  THE NEXT STATEMENT INPUTS VALUES FOR M,N, AND/OR P
READ           READ AND PRINT DATA
               SETDIM.(A,M,N)
               SETDIM.(B,N,P)
               SETDIM.(C,M,P)
               READ FORMAT $6F12.4*$, A(1,1)...A(M,N), B(1,1)...B(N,P)
               PRINT FORMAT $1H0,8F13.4*$,A(1,1)...A(M,N),B(1,1)...B(N,P)
               THROUGH Q, FOR I = 1,1, I .G. M
               THROUGH Q, FOR J = 1,1, J .G. P
Q              C(I,J)=(S=0.,K=1,1,K.G.N,S+A(I,K)*B(K,J))
              R NOTE. THE PRECEDING 3 STATEMENTS CAN BE REPLACED BY A
              R SINGLE ITERATED STATEMENT.  SEE EXAMPLE IN SECTION 2.17
               PRINT FORMAT RESULT,C(1,1)...C(M,P)
               TRANSFER TO READ
              R
              R FORMAT SPECIFICATIONS
              R
               VECTOR VALUES RESULT = $9H1C MATRIX//(1H0,8F13.4)*$
               END OF PROGRAM
$DATA
M = 2, N = 3, P = 3         *
              1.          2.          3.          4.          5.          6.
              7.          8.          9.         10.         11.         12.
             13.         14.         15.
```

Example 7

Problem: Solve a system of $n \leq 20$ simultaneous linear equations in n unknowns, assuming that one does not encounter a zero on the main diagonal of the coefficient matrix during the solution process.

Analysis: We shall use a Jordan Elimination Method, in which each diagonal coefficient is used to "clear" all other coefficients in its column to zero by appropriate multiplications and subtractions. Since we shall divide the "clearing row" by the diagonal element in that row before clearing the column, we shall finish the process with only a diagonal of ones and the solution to the problem as the resulting right hand side of the equations.

We denote the system of equations to be solved by:

$$\begin{array}{c} a_{11}x_1 + a_{12}x_2 + \dots + a_{1n}x_n = a_{1,n+1} \\ a_{21}x_1 + a_{22}x_2 + \dots + a_{2n}x_n = a_{2,n+1} \\ \vdots \qquad \vdots \qquad\qquad \vdots \qquad \vdots \\ a_{n1}x_1 + a_{n2}x_2 + \dots + a_{nn}x_n = a_{n,n+1} \end{array} \tag{1}$$

We divide the first row by its diagonal element a_{11}. Then to clear a_{21} to zero we subtract a_{21} times the first row from the second row, and so on. In general, to clear a_{ik} to zero (after row k has been divided by a_{kk}), we subtract a_{ik} times row k from row i ($i \neq k$). A typical element a_{ij} is thus transformed each time by the formulas:

$$a_{kj} = a_{kj}/a_{kk} \tag{2}$$

$$a_{ij} = a_{ij} - a_{ik}a_{kj} \qquad (i \neq k) \tag{3}$$

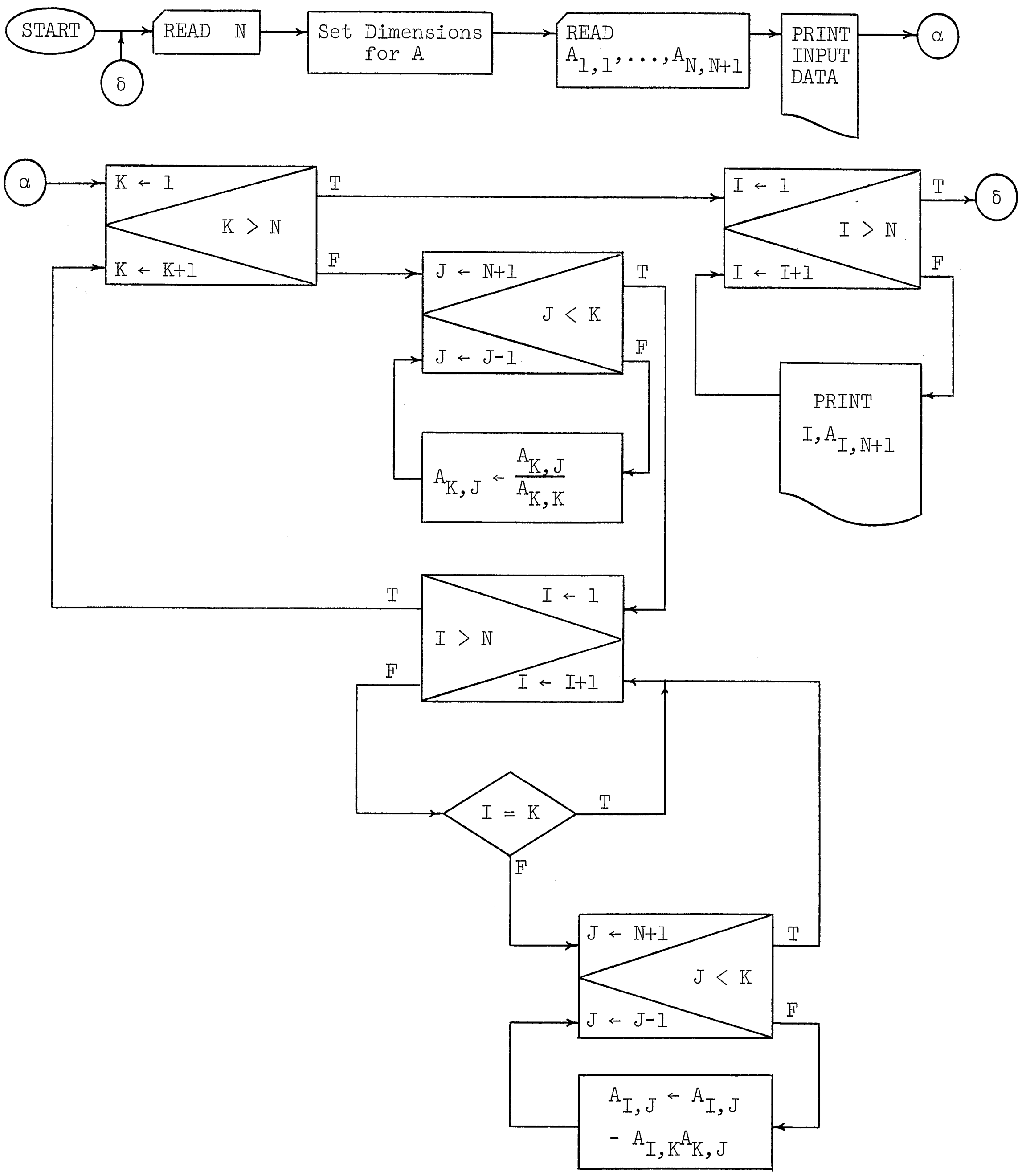
where the value of a_{kj} in (3) is the result of (2). These transformations are performed for $k = 1,2, \dots, n$. For each (fixed) k, we will let $i = 1,2, \dots, k-1,k+1, \dots, n$, so as to operate on all rows except $i = k$. While transforming each row we will cycle on j from right to left; i.e., $j = n+1, n, n-1, \dots, k$, and we stop at $j = k$ since for $j < k$ there is no change in the matrix (since for $j < k$, $a_{kj} = 0$, except that we don't actually store the 0).

The array

$$A = (a_{ij}) = \begin{bmatrix} a_{11}a_{12} & \cdots & a_{1,n+1} \\ \vdots & & \\ a_{n1}a_{n2} & \cdots & a_{n,n+1} \end{bmatrix}$$

is called the "matrix of coefficients" of the system (1).

It should be understood that this method, involving the assumption of no zeros on the diagonal and not searching for the largest element of a row to use as a divisor (to minimize round-off error), is not satisfactory from a mathematical point of view. It could serve as a basis for a larger, more complete program, however, and serves here only as an example problem.



START
δ
READ N
Set Dimensions for A
READ $A_{1,1},\ldots,A_{N,N+1}$
PRINT INPUT DATA
α
α
K ← 1
K > N
K ← K+1
T
F
J ← N+1
J < K
J ← J-1
T
F
$A_{K,J} \leftarrow \frac{A_{K,J}}{A_{K,K}}$
I ← 1
I > N
I ← I+1
T
F
δ
PRINT
$I, A_{I,N+1}$
I ← 1
I > N
I ← I+1
T
F
I = K
T
F
J ← N+1
J < K
J ← J-1
T
F
$A_{I,J} \leftarrow A_{I,J} - A_{I,K}A_{K,J}$

The program:

```
          DIMENSION A(20*21)
DELTA     READ DATA (VALUE OF N)
          SETDIM.(A,N,N+1)
          READ FORMAT$6F12.4*$, A(1,1)...A(N,N+1)
          PRINT FORMAT INVAL,N,A(1,1)...A(N,N+1)
          THROUGH B, FOR K = 1,1,K .G. N
          (J = N+1,-1,J.L.K, A(K,J) = A(K,J) / A(K,K)
          THROUGH B, FOR I = 1,1,I .G. N
          WHENEVER I .E. K, TRANSFER TO B
          (J = N+1, -1, J .L. K,
         1A(I,J) = A(I,J)-A(I,K)*A(K,J)   )
B         CONTINUE
          THROUGH E, FOR I = 1,1,I .G. N
E         PRINT FORMAT RESULT,I,A(I,N+1)
          TRANSFER TO DELTA
          INTEGER I,J,K,N
         R
         RFORMAT SPECIFICATIONS
         R
          VECTOR VALUES INVAL = $7H1 INPUT/ 4H0N = I4/
         1 7H0MATRIX//(1H0,8F12.4)*$
          VECTOR VALUES RESULT = $ 1H0,T2,2HA(,I2,3H) = F12.4*$
          END OF PROGRAM
$DATA
   N = 3 *
         1.          1.          1.          6.          -1.          0
         0.         -1.         -1.         -2.          -9.        -32.
```

2. <u>Business Data Processing Examples</u>

<u>Example 1</u>

<u>Problem</u>: Compute the social security deduction and accumulated gross pay. The program should read a card containing: (a) the employee's name, (b) his payroll number, (c) his gross pay for the current week, and (d) his accumulated gross pay for the current year (but not including item (c)). For each card read, the program should print (a) and (b) from the card, and, in addition, print (e) the updated gross pay, (f) the social security deduction for the current week, and (g) the net pay for the current week, taking into account only the social security deduction.

<u>Analysis</u>: The social security deduction is currently 3%[†] of the gross pay until the accumulated gross pay for the year exceeds $4800.00.[†] The updated gross pay can be computed from the formula, (e) = (c) + (d). The social security deduction has already been made on (d). There are thus three cases to consider:

(1) (d) $\geq$ 4800.00, in this case (f) = 0.

(2) (d) < 4800.00 and (c) + (d) > 4800.00, in this case (f) = 3% of 4800.00 - (d).

(3) (c) + (d) $\leq$ 4800.00, in this case (f) = 3% of (c).

The information on the cards to be read will be in the following format:

[†]Admittedly, slightly out of date. Maybe it should be a parameter.

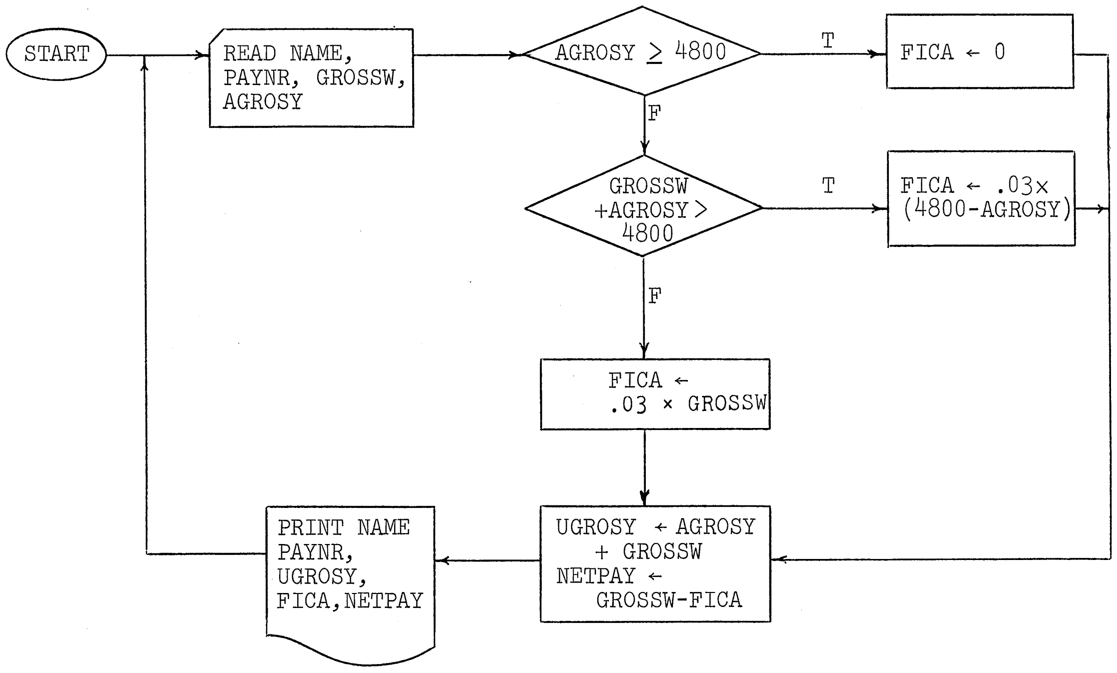
Card Columns		Information
1 - 30	(a)	employee's name
31 - 38	(b)	payroll number
39 - 44	(c)	gross pay for the current week in the form XXX.XX
45 - 52	(d)	accumulated gross pay for the current year in the form XXXXX.XX

The printed output will be in the following format:

Line Columns		Information
1		Carriage control for printer
2 - 31	(a)	employee's name
32 - 34		Blank
35 - 42	(b)	payroll number
43 - 45		Blank
46 - 53	(e)	updated gross pay for current year in the form XXXXX.XX
54 - 56		Blank
57 - 61	(f)	social security deduction for current week in the form XX.XX
62 - 64		Blank
65 - 70	(g)	net pay for current week in the form XXX.XX

Flow chart: We will use the following abbreviations:

NAME	for employee's name (a).
PAYNR	for payroll number (b).
GROSSW	gross pay for current week (c).
AGROSY	accumulated gross pay for current year (d).
UGROSY	updated gross pay for current year (e).
FICA	social security deduction for current week (f).
NET PAY	net pay for current week (g).


START
READ NAME, PAYNR, GROSSW, AGROSY
AGROSY ≥ 4800
T
FICA ← 0
F
GROSSW +AGROSY > 4800
T
FICA ← .03× (4800-AGROSY)
F
FICA ← .03 × GROSSW
UGROSY ← AGROSY + GROSSW NETPAY ← GROSSW-FICA
PRINT NAME PAYNR, UGROSY, FICA, NETPAY

The Program:

```
START      READ FORMAT $5C6,I8,F6.2,F8.2*$,
          1  NAME(1)...NAME(5), PAYNR, GROSSW, AGROSY
           DIMENSION NAME(5)
           INTEGER PAYNR, NAME
           WHENEVER AGROSY .GE. 4800.
                FICA=0.
           OR WHENEVER GROSSW+AGROSY .G. 4800.
                FICA=.03*(4800.-AGROSY)
           OTHERWISE
                FICA=.03*GROSSW
           END OF CONDITIONAL
           UGROSY = AGROSY+GROSSW
           NETPAY = GROSSW-FICA
           PRINT FORMAT $1H0,5C6,S3,I8,S3,F8.2,S3,F5.2,S3,F6.2*$,
          1  NAME(1)...NAME(5), PAYNR, UGROSY,FICA,NETPAY
           TRANSFER TO START
           END OF PROGRAM
$DATA
GEORGE WASHINGTON              12345678  100.    4800.
JOHN ADAMS                     12345679  200.    4900.
THOMAS JEFFERSON               12345680  200.    4600.
JAMES MADISON                  12345681  200.    4700.
JOHN QUINCY ADAMS              12345682  100.     300.
```

Notes on these programs:

1. The maximum number of characters which can be stored in one machine word is six. Hence, we need five machine words to store the 30 characters allowed for the employee's name. We need to give a dimension declaration stating that NAME is actually to be a block and that NAME(5) is the last word of this block. In the read and print statements we specify that the whole block is to be read or printed by writing NAME(1) ... NAME(5) and giving the format specification 5C6, i.e., 5 words of 6 characters.

2. Since the payroll number is an integer (I8, i.e., an 8 digit integer) we give an integer mode declaration stating that PAYNR is an integer. Similarly, since alphabetic information is assumed to be in the integer mode, NAME is also declared to be integer.

Example 2

Problem: Assume that a master tape is available containing basic information for each employee: (1) The employee number, (2) his hourly rate, (3) gross pay to date, (4) amount of withholding tax withheld to date, (5) social security deduction withheld to date, (6) net pay to date, and (7) the number of exemptions. Input will be in the form of m cards representing the current pay record, containing the employee's number and the number of hours worked during the current week. Pay is to be computed at time and a half for any hours worked over forty. We shall assume that the input deck is already sorted according to increasing employee number, but we shall provide for cards which may be out of order. The last input card must have an employee number greater than the last employee number of the master tape.

The withholding tax W is to be computed by the formula:

$$W = .18(\text{Gross pay} - 13n)$$

where n is the number of exemptions. If n is negative, we set $W = 0$ (see Note 2 below). The social security deduction FICA is 3 per cent of gross pay up to \$4800, with no deduction for gross pay over \$4800.

A program is desired which will produce a listing (for each input card) of (a) employee number, (b) gross pay this week, (c) withholding tax, (d) FICA, (e) net pay for the week. Moreover, a new updated master tape should be prepared, with provision for saving the previous master tape as well. As much checking as possible should be incorporated, including specifying to the operator the number of the master tape needed, and the number to be assigned to the new tape produced by the program, and the automatic checking that the correct tape has been mounted on the unit.

Note 1: Abbreviations used here are outlined in Example 1, except for the following new terms:

AWITHY	accumulated withholding tax for year
AFICAY	accumulated social security deduction for year
ANETY	accumulated net pay for year
EXEMPT	number of exemptions

Note 2: In the computation of the gross pay for the current week we shall find it useful to be able to compute a function (which we shall call EXCESS) of two numbers, say a and b, whose value is 0 if $a \leq b$, and a-b if $a > b$. A formula for this function is

$$\text{EXCESS.}(a,b) = \frac{a - b + |a - b|}{2},$$

where | | denotes the usual "absolute value." In fact, by using EXCESS, a simple one-line formula is:

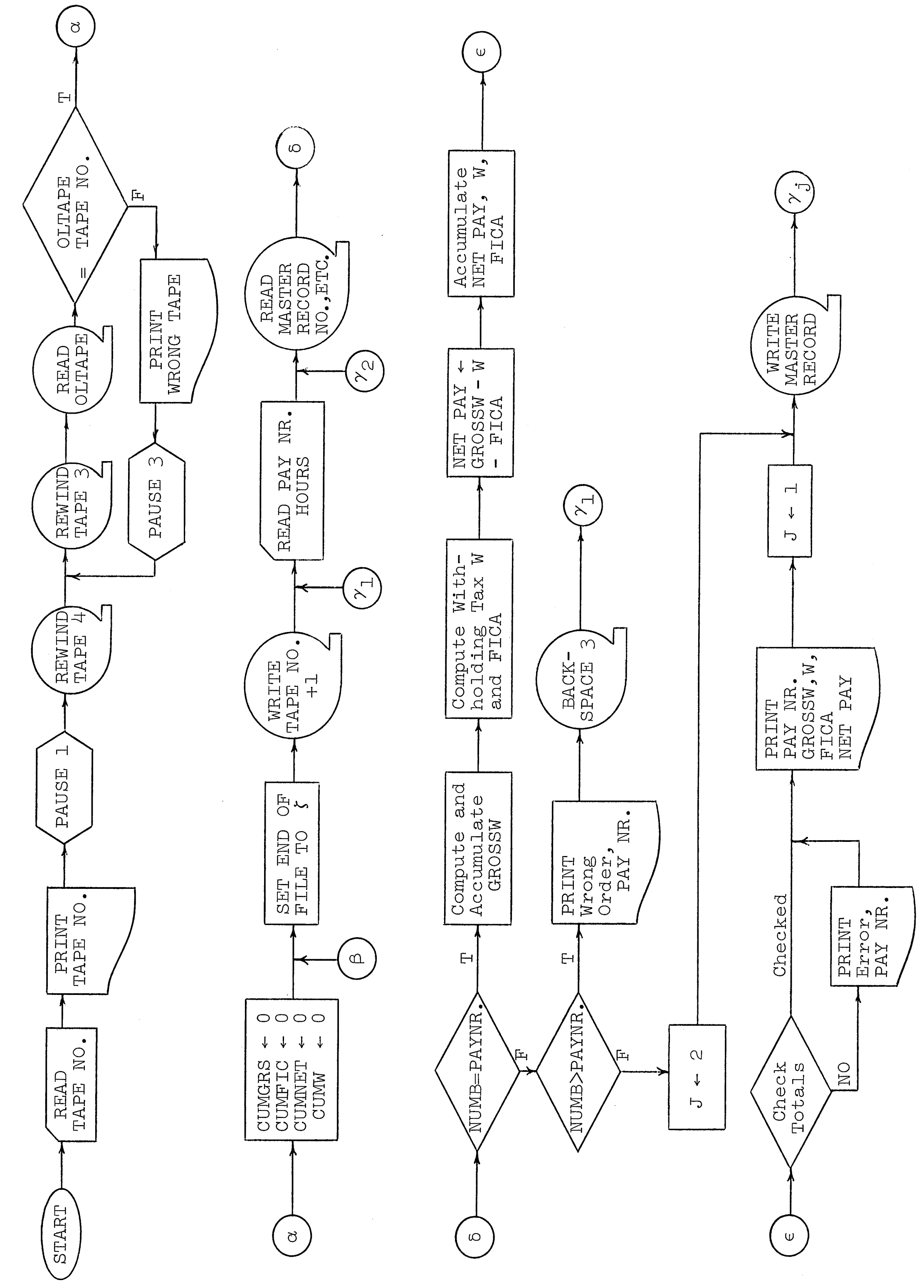
$$\text{FICA} = .03 \times \text{EXCESS.}(\text{GROSSW, EXCESS. (AGROSY, 4800.)})$$

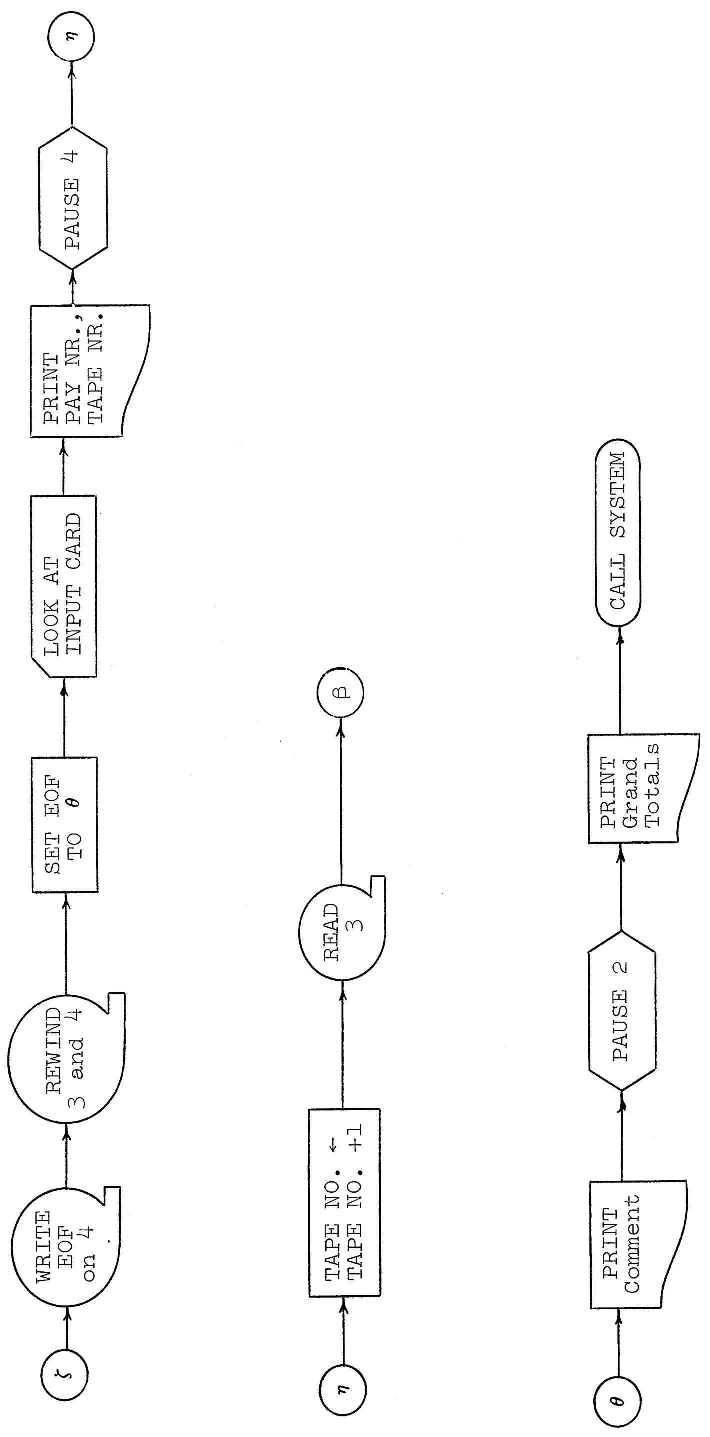
where AGROSY is assumed to already contain GROSW, i.e., to have been updated already. We shall also apply this function in the case of the withholding tax to guarantee that we do not make a negative deduction. Thus

```
W = .18*EXCESS. (GROSSW, 13*EXEMPT)
```

Note 3: To check the order of input cards (normally in order of increasing employee number with a large employee number greater than the last employee number on the master tape) the program uses the subroutine SETEOF.(LABEL), where LABEL is the statement label of a statement to be executed if an end of file condition is detected during reading.

Since the last input card has a large employee number the first end of file condition is normally detected at the end of processing, but an illegal input card may also exist with a high employee number. After the first end of file is detected the end of file return is changed and the input tape checked for end of file. If no end of file exists a comment is printed to change tapes and processing begins again.


START
READ TAPE NO.
PRINT TAPE NO.
PAUSE 1
REWIND TAPE 4
REWIND TAPE 3
READ OLTAPE
OLTAPE = TAPE NO.
T
α
F
PRINT WRONG TAPE
PAUSE 3
α
CUMGRS ← 0
CUMFIC ← 0
CUMNET ← 0
CUMW ← 0
β
SET END OF FILE TO ζ
WRITE TAPE NO. +1
γ1
READ PAY NR. HOURS
γ2
READ MASTER RECORD NO.,ETC.
δ
δ
NUMB=PAYNR.
T
Compute and Accumulate GROSSW
Compute With-holding Tax W and FICA
NET PAY ← GROSSW - W - FICA
Accumulate NET PAY, W, FICA
ε
F
NUMB>PAYNR.
T
PRINT Wrong Order, PAY NR.
BACK-SPACE 3
γ1
F
J ← 2
ε
Check Totals
NO
PRINT Error, PAY NR.
Checked
PRINT PAY NR. GROSSW,W, FICA NET PAY
J ← 1
WRITE MASTER RECORD
γj



ζ
WRITE EOF on 4
REWIND 3 and 4
SET EOF TO θ
LOOK AT INPUT CARD
PRINT PAY NR., TAPE NR.
PAUSE 4
η
η
TAPE NO. ← TAPE NO. +1
READ 3
β
θ
PRINT Comment
PAUSE 2
PRINT Grand Totals
CALL SYSTEM

```
START      READFORMAT IDENT,TAPENO
           INTEGER TAPENO,PAYNR,NUMB,J,OLTAPE
           PRINT ON LINE FORMAT OPER,TAPENO
           PAUSE NO. 1
           REWIND TAPE 4
TEST       REWIND TAPE 3
           READ BINARY TAPE 3,OLTAPE
           WHENEVER OLTAPE .E. TAPENO,TRANSFER TO MAIN
           PRINT ON LINE FORMAT WRONG
           PAUSE NO. 3
           TRANSFER TO TEST
MAIN       CUMGRS = 0.
           CUMFIC = 0.
           CUMNET = 0.
           CUMW = 0.
REDO       EXECUTE SETEOF.(M FILE)
           WRITE BINARY TAPE 4,TAPENO+1
READ(1)    READ FORMAT EMPLOY,PAYNR,HOURS
READ(2)    READ BINARY TAPE 3, NUMB,RATE,AGROSY,AWITHY,
          1AFICAY,ANETY,EXEMPT
           WHENEVER NUMB .E. PAYNR
           GROSSW = RATE*HOURS+.5*RATE*EXCESS.(HOURS,40.)
           AGROSY = AGROSY+GROSSW
           W = .18*EXCESS.(GROSSW,13.*EXEMPT)
           FICA = .03*EXCESS. (GROSSW,EXCESS.(AGROSY,4800.))
           NETPAY = GROSSW-W-FICA
           AWITHY = AWITHY+W
           AFICAY = AFICAY+FICA
           ANETY = ANETY+NETPAY
           CUMGRS = CUMGRS+GROSSW
           CUMFIC = CUMFIC+FICA
           CUMNET = CUMNET+NETPAY
           CUMW = CUMW+W
           WHENEVER .ABS. (AGROSY-ANETY-AFICAY-AWITHY) .GE. .005, PRINT
          1FORMAT ERROR,PAYNR
           PRINT FORMAT OUTPUT,PAYNR,GROSSW,W,FICA,NETPAY
           J = 1
           OR WHENEVER NUMB.G.PAYNR
           PRINT FORMAT ORDER,PAYNR
           BACKSPACE RECORD OF TAPE 3
           TRANSFER TO READ(1)
           OTHERWISE
           J = 2
           END OF CONDITIONAL
           WRITE BINARY TAPE 4,NUMB,RATE,AGROSY,AWITHY,AFICAY,ANETY,
          1EXEMPT
           TRANSFER TO READ(J)
M FILE     END OF FILE TAPE 4
           REWIND TAPE 3
           REWIND TAPE 4
           EXECUTE SETEOF.(C FILE)
           LOOK AT FORMAT EMPLOY, DUMMY, DUMMY
           PRINT FORMAT NOMAN,PAYNR,TAPENO
           PRINT ON LINE FORMAT NOMAN,PAYNR,TAPENO
           PAUSE NO. 4
           TAPENO=TAPENO+1
           READ BINARY TAPE 3,DUMMY
           TRANSFER TO REDO
C FILE     PRINT ON LINE FORMAT OFF,TAPENO,TAPENO+1
           PAUSE NO. 2
           PRINT FORMAT TOTALS,CUMGRS,CUMFIC,CUMNET,CUMW
           EXECUTE SYSTEM.
          R
           INTERNAL FUNCTION EXCESS.(X,Y)=(X-Y+ .ABS. (X-Y))/2.
          R
          R FORMAT SPECIFICATIONS
```

```
R
 VECTOR VALUES IDENT = $I8*$
 VECTOR VALUES OPER = $15H4MOUNT TAPE NO. I8,S2,30HON TAPE UNI
1T NO. 3,PRESS START*$
 VECTOR VALUES WRONG = $48H4THE WRONG TAPE HAS BEEN USED. PLEA
1SE TRY AGAIN.*$
 VECTOR VALUES EMPLOY = $I8,F10.2*$
 VECTOR VALUES ERROR = $37H0ERROR IN CHECKING TOTALS FOR MAN N
1O. I8*$
 VECTOR VALUES OUTPUT = $1H0,I8,4F20.2*$
 VECTOR VALUES OFF = $24H4REMOVE TAPE 3,LABEL IT I8,S4,23HREMO
1VE TAPE 4, LABEL IT I8*$
 VECTOR VALUES NOMAN = $38H0THERE IS NO MASTER RECORD FOR MAN
1NO.I8/22H0PULL TAPE 3. LABEL IT I8/51H0RESELECT TAPE 4 AS TAP
2E 3 AND HANG BLANK TAPE ON 4/16H0THEN PUSH START*$
 VECTOR VALUES ORDER = $8H0MAN NO.I8,43H IS OUT OF ORDER OR NO
1 MASTER RECORD EXISTS*$
 VECTOR VALUES TOTALS = $13H1CUM. GROSS =F10.2/12H0CUM. FICA =
1F10.2/11H0CUM. NET = F10.2/23H0CUM. WITHHOLDING TAX = F10.2*$
 END OF PROGRAM
```

<u>Example 3</u>

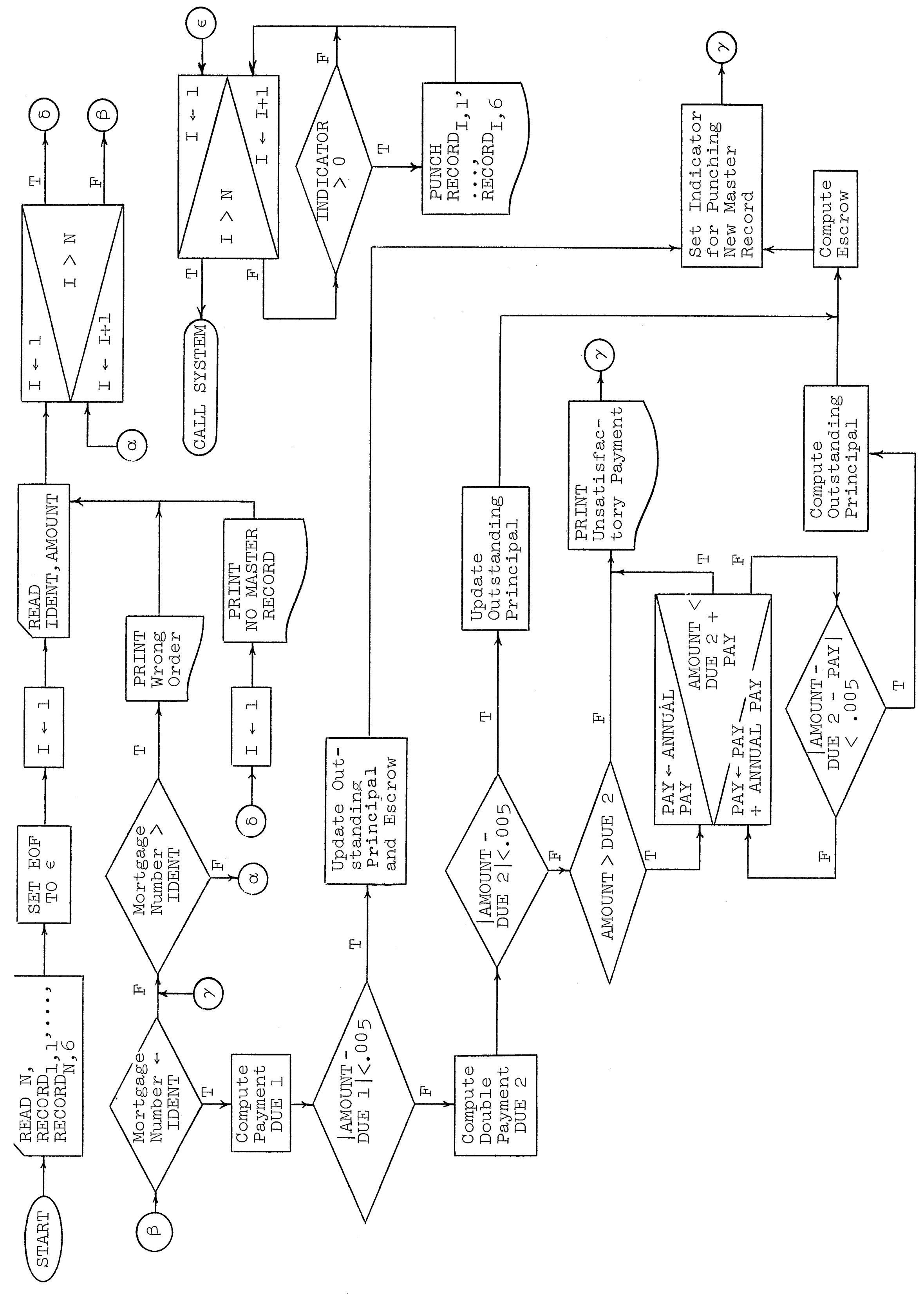
<u>Problem</u>: Mortgage Payment. The type of mortgage we consider here is the fixed principal type for which each installment consists of an interest payment, a fixed amount to be deducted from the outstanding principal, and an additional amount to be placed in escrow--to be used to make insurance and tax payments.

Assume that a master card file is available containing the following information for each mortgage: (1) the mortgage number; (2) amount of outstanding principal; (3) annual payment on principal; (4) interest rate; (5) annual escrow payment; and (6) current escrow balance. There is also a file of cards available containing the current payment record consisting of mortgage number and amount of payment received. The master file and current payment file are assumed to be in order of increasing mortgage number.

The program is to read a card from the current payment record and check to see if it is acceptable. A payment is deemed acceptable if it consists of a single normal payment (i.e., a payment consisting of a single principal payment, a single escrow payment, and an interest payment for a single period) or if it consists of exactly two normal payments and any number (i = 0,1,2, ...) of principal payments.

Note 1: To represent the dollar sign ($) in an alphabetic constant we use the double $$ symbol. Thus $$$$ is stored internally as the six character alphabetic constant $□□□□□ . (See Section 1.13.) The printing of this constant is called for in the statement labeled OVRPY. Using a C1 field code, only the $ (the left most character) will be printed.

Note 2: The current payments are processed until the file is exhausted. The detection of the end of file on reading transfers control to the section of the program which punches the new master file.


START
READ N, RECORD1,1,..., RECORDN,6
SET EOF TO ϵ
I ← 1
READ IDENT,AMOUNT
I ← 1
I > N
I ← I+1
T
δ
F
β
α
β
Mortgage Number ← IDENT
T
F
γ
Mortgage Number > IDENT
T
PRINT Wrong Order
F
α
δ
I ← 1
PRINT NO MASTER RECORD
Compute Payment DUE 1
|AMOUNT-DUE 1|<.005
T
Update Out-standing Principal and Escrow
F
Compute Double Payment DUE 2
ϵ
I ← 1
I > N
I ← I+1
T
CALL SYSTEM
F
INDICATOR > 0
F
T
PUNCH RECORDI,1, ..., RECORDI,6
|AMOUNT - DUE 2|<.005
T
Update Outstanding Principal
F
AMOUNT > DUE 2
F
PRINT Unsatisfac-tory Payment
γ
T
PAY ← ANNUAL PAY
AMOUNT < DUE 2 + PAY
PAY ← PAY + ANNUAL PAY
T
F
|AMOUNT - DUE 2 - PAY| < .005
T
F
Compute Outstanding Principal
Compute Escrow
Set Indicator for Punching New Master Record
γ

```
           DIMENSION RECORD(200*7)
           INTEGER I,NUMB
START      READ FORMAT SIZE,NUMB
           READ FORMAT MASTER,(I=1,1,I.G.NUMB,RECORD(I,1)...RECORD(I,6))
           EXECUTE SETEOF.(UPDATE)
           I = 1
CARDS      READ FORMAT PAYMT, IDENT,AMOUNT
           THROUGH B, FOR I = 1,1,I .G. NUMB
           WHENEVER RECORD(I,1) .E. IDENT
             DUE1 = RECORD(I,3)+RECORD(I,5)+RECORD(I,4)*RECORD(I,2)
             WHENEVER .ABS. (AMOUNT-DUE1) .L. .005
               RECORD (I,2) = RECORD(I,2)-RECORD(I,3)
               RECORD(I,6)=RECORD(I,6)+RECORD(I,5)
               TRANSFER TO CODE
           OTHERWISE
             DUE2 = 2.*DUE1 - RECORD(I,4)*RECORD(I,3)
           END OF CONDITIONAL
           WHENEVER .ABS. (AMOUNT-DUE2) .L. .005
             RECORD(I,2)=RECORD(I,2)-2.*RECORD(I,3)
             TRANSFER TO ESCROW
           OR WHENEVER AMOUNT .G. DUE2
             THROUGH C, FOR PAY = RECORD(I,3),RECORD(I,3),
          1  AMOUNT .L. DUE2+PAY
C            WHENEVER .ABS. (AMOUNT-DUE2-PAY) .L. .005, TRANSFER TO
          1    PAID
             TRANSFER TO OVRPAY
PAID           RECORD(I,2)=RECORD(I,2)-2.*RECORD(I,3)-PAY
ESCROW         RECORD(I,6)=RECORD(I,6)+2.*RECORD(I,5)
CODE           RECORD(I,7)=1.

             OTHERWISE
OVRPAY         PRINT FORMAT REJECT,$$$$ ,DOLLAR,AMOUNT
             END OF CONDITIONAL
           OR WHENEVER RECORD(I,1) .G. IDENT
             PRINT FORMAT ORDER,IDENT,DOLLAR,AMOUNT
           OTHERWISE
B            CONTINUE
             I=1
             PRINT FORMAT NONE,IDENT
           END OF CONDITIONAL
           TRANSFER TO CARDS
UPDATE     THROUGH D, FOR I=1,1,I .G. NUMB
D          WHENEVER RECORD(I,7).G. 0.,PUNCH FORMAT MASTER,RECORD(I,1)...
          1RECORD(I,6)
          R
          RFORMAT SPECIFICATIONS
          R
           VECTOR VALUES SIZE=$I10*$
           VECTOR VALUES MASTER=$F10.0,5F10.2*$
           VECTOR VALUES PAYMT = $F10.0,F10.2*$
           VECTOR VALUES REJECT = $20H0PAYMENT ON MORTGAGE,F10. ,3H,  ,C
          11,F10.2,19H IS UNSATISFACTORY.*$
           VECTOR VALUES ORDER = $26H0PAYMENT CARD FOR MORTGAGE,F10.0,3H
          1,  C1,F10.2,44H IS OUT OF ORDER OR NO MASTER RECORD EXISTS.*$
           VECTOR VALUES NONE = $41H0NO MASTER RECORD EXISTS FOR MORTGAG
          1E NO.,F10.0*$
           END OF PROGRAM
```

Example 4

Problem: Computation of actuarial commutation columns based on an arbitrary set of mortality rates and an interest rate, as an external function to be used by another program.

Analysis: Commutation columns, which are very important tools in actuarial problems, are generated very easily by means of the formulas given below. The quantities M_x, N_x, and D_x in these formulas occur most often in combination, as in the computation of P_x. Assuming a population of some initial size (at $x = b_o$) (here 1,000,000), ℓ_x is the number living at age x (so that $\ell_{b_o} = 1{,}000{,}000$), q_x is the mortality rate, and d_x is the number of deaths at age x. Thus $d_x = q_x \cdot \ell_x$. The quantity D_x is computed by the formula $D_x = \ell_x(1 + i)^{-x}$, where i is the interest rate. Another quantity, C_x is given by the formula $C_x = d_x(1 + i)^{-(x+1)}$. It can be used, for example, to compute the cost of term insurance, since $\frac{C_x}{D_x}$ is the premium for one year term insurance of \$1 at age x.

The sums M_x and N_x are obtained by the formulas

$$M_x = \sum_{y=x}^{\infty} C_y \,, \qquad N_x = \sum_{y=x}^{\infty} D_y$$

We note that for some w, we always have $q_w = 1$, so that $\ell_{w+1} = 0$ (since $\ell_{w+1} = \ell_w - d_w = \ell_w - \ell_w = 0$), therefore $D_{w+1} = 0$, $d_{w+1} = 0$, $C_{w+1} = 0$, and the sums for M_x and N_x are actually finite sums.

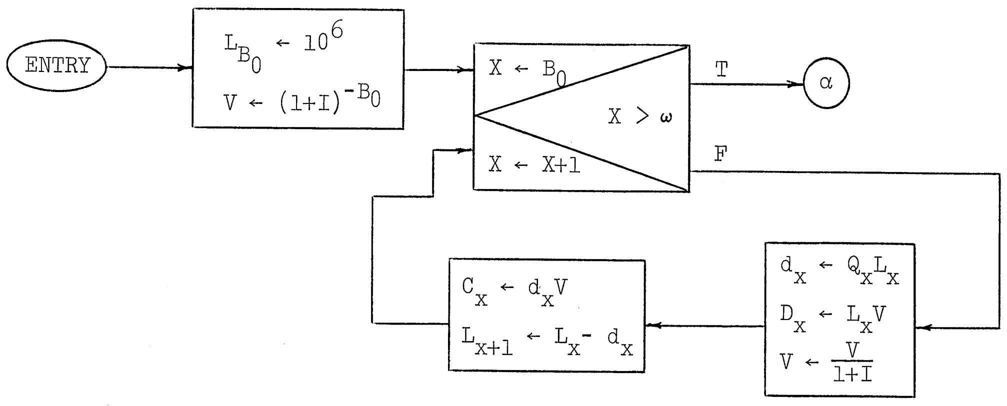
The three most useful quantities computed here are (1) $P_x = M_x/N_x$, which is the annual premium payable for an entire life for \$1 of whole life insurance, (2) $A_x = M_x/D_x$, which is the present value at age x of a whole life annuity of \$1, first payment at age x.

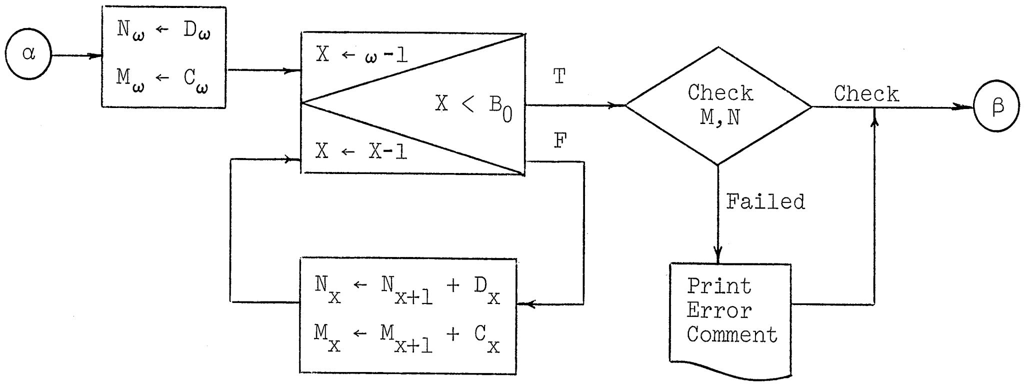
Printing of results is under control of an input variable PRINT. Certain relationships must hold between some independently computed values, and these are used as checks on the computation:

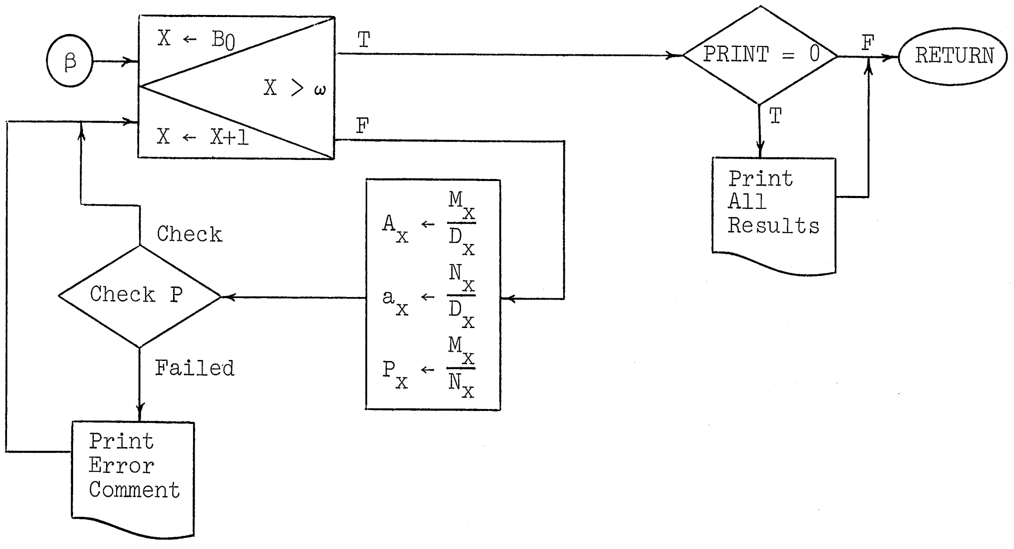
$$M_{b_o} + N_{b_o+1} = N_{b_o}/(1+i)$$

$$P_x = 1/a_x - i/(1+i)$$

These cannot be expected to come out exactly equal, because of round-off, but they should differ by very little.


ENTRY
$L_{B_0} \leftarrow 10^6$
$V \leftarrow (1+I)^{-B_0}$
$X \leftarrow B_0$
$X > \omega$
$X \leftarrow X+1$
T
F
α
$d_x \leftarrow Q_x L_x$
$D_x \leftarrow L_x V$
$V \leftarrow \frac{V}{1+I}$
$C_x \leftarrow d_x V$
$L_{x+1} \leftarrow L_x - d_x$


α
$N_\omega \leftarrow D_\omega$
$M_\omega \leftarrow C_\omega$
$X \leftarrow \omega - 1$
$X < B_0$
$X \leftarrow X-1$
T
F
$N_x \leftarrow N_{x+1} + D_x$
$M_x \leftarrow M_{x+1} + C_x$
Check M,N
Check
Failed
Print Error Comment
β


β
$X \leftarrow B_0$
$X > \omega$
$X \leftarrow X+1$
T
F
$A_x \leftarrow \frac{M_x}{D_x}$
$a_x \leftarrow \frac{N_x}{D_x}$
$P_x \leftarrow \frac{M_x}{N_x}$
Check P
Check
Failed
Print Error Comment
PRINT = 0
F
T
Print All Results
RETURN

The Program:

```
R
R SAMPLE CALLING PROGRAM
R
 READ FORMAT $12F6.5*$, Q(0)...., Q(100)
 DIMENSION(Q,L,SMALLD,BIGD,C,N,M,BIGA,SMALLA,P)(120)
 EXECUTE COMFCN.(Q,0,099,.03,L, SMALLD,BIGD,C,N,M,BIGA,SMALLA,
1 P,1)
   INTEGER PRINT
   END OF PROGRAM
```

The External Function:

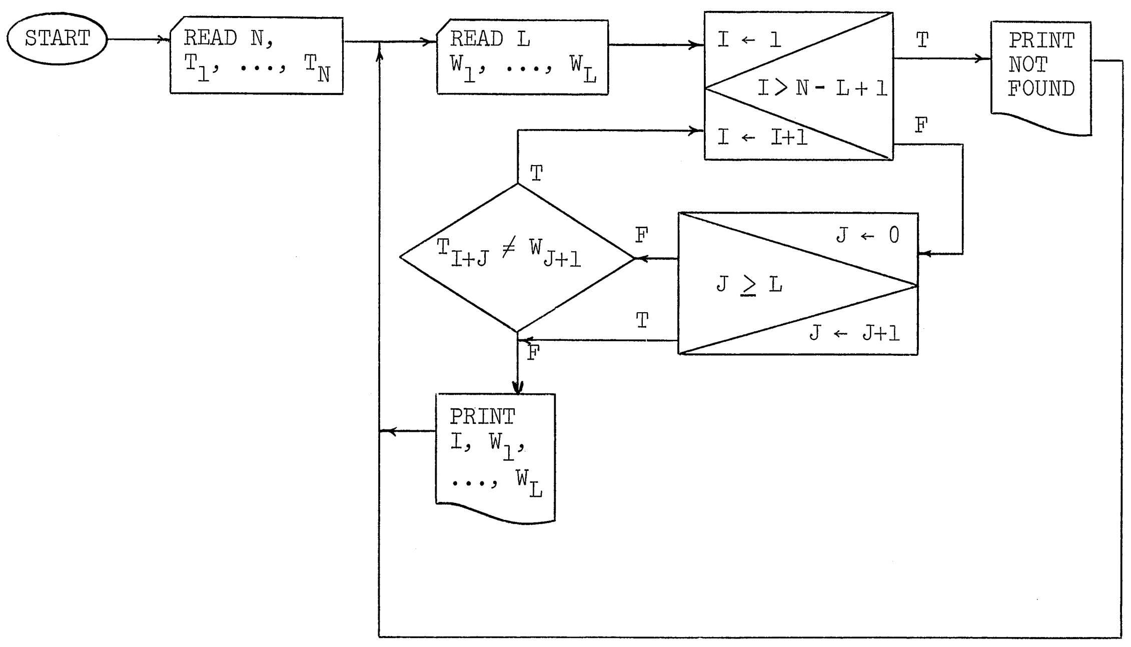
```
        R COMMUTATION TABLE FUNCTION
        R IF PRINT = 0,SUPPRESS PRINTING
         EXTERNAL FUNCTION(Q,BZERO,OMEGA,I,L,SMALLD,BIGD,C,N,
        1 M,BIGA,SMALLA,P,PRINT)
         ENTRY TO COMFCN.
         INTEGER BZERO,OMEGA,PRINT,X
         L(BZERO) = 1E6
         V = (1.+I) .P. -BZERO
         THROUGH A,FOR X = BZERO,1,X .G. OMEGA
         SMALLD(X) = Q(X)*L(X)
         BIGD(X) = L(X)*V
         V = V/(1.+I)
         C(X) = SMALLD(X)*V
       A L(X+1) = L(X)-SMALLD(X)
         N(OMEGA) = BIGD(OMEGA)
         M(OMEGA) = C(OMEGA)
         THROUGH B, FOR X = OMEGA-1,-1, X .L. BZERO
         N(X) = N(X+1) + BIGD(X)
B        M(X) = M(X+1) + C(X)
         WHENEVER .ABS.(M(BZERO)+N(BZERO+1)-N(BZERO)/(I+1.)) .G. 1.,
        1 TRANSFER TO MNERR
E        THROUGH G ,FOR X = BZERO,1,X .G. OMEGA
         BIGA(X) = M(X)/BIGD(X)
         SMALLA(X) = N(X)/BIGD(X)
         P(X) = M(X)/N(X)
         WHENEVER .ABS. (P(X)-1./SMALLA(X) + I/(I+1.)) .G.
        1 1E-4,TRANSFER TO PERROR
G        CONTINUE
         WHENEVER PRINT .E. 0,FUNCTION RETURN
        R
        ROUTPUT GENERATOR
        R
         PRINT FORMAT HEAD01,I
         VECTOR VALUES HEAD01 = $1H1,4HI = F5.4//
        1 4H   X,S13,4HQ(X),S18,4HL(X),S15,10HSMALL D(X),
        2 S8,1HX*$
         PRINT FORMAT F1,(X=BZERO,1,X.G.OMEGA,X,Q(X),L(X),SMALLD(X),X)
         VECTOR VALUES F1 = $1H0,I3,3E22.9,I7*$
         PRINT FORMAT HEAD02
         VECTOR VALUES HEAD02 = $1H1,T5,1HX,S11,8HBIG D(X),
        1 S16,4HC(X),S18,4HM(X),S18,4HN(X),S11,1HX*$
          PRINT FORMAT F2,(X=BZERO,1,X.G.OMEGA,X,BIGD(X),C(X),
         1 M(X),N(X),X)
          VECTOR VALUES F2 = $1H0,I4,4E22.9,I7*$
          PRINT FORMAT HEAD03
          VECTOR VALUES HEAD03 = $4H1    X,S11,8HBIG A(X),
         1 S13,10HSMALL A(X), S15,4HP(X),S11,1HX *$
          PRINT FORMAT F3,(X=BZERO,1,X .G. OMEGA,X,BIGA(X),
         1 SMALLA(X),P(X),X)
          VECTOR VALUES F3 = $1H0,I3,3E22.9,I7*$
          FUNCTION RETURN
```

```
PERROR    PRINT FORMAT PERR•P(X)•SMALLA(X)•I
          VECTOR VALUES PERR = $27H0ERROR ON P CHECK•  P(X) = E18•9•
         1 S10•13HSMALL A(X) = E18•9•S10•4HI = F5•4*$
          TRANSFER TO G
MNERR     PRINT FORMAT MNERR1
          VECTOR VALUES MNERR1 = $19H0ERROR ON M•N CHECK*$
          TRANSFER TO E
          END OF FUNCTION
$DATA
  2258   577   414   338   299   276   261   247   231   212   197   191
   192   198   207   215   219   225   230   237   243   251   259   268
   277   288   299   311   325   340   356   373   392   412   435   459
   486   515   546   581   618   659   703   751   804   861   923   991
  1064  1145  1232  1327  1430  1543  1665  1798  1943  2100  2271  2457
  2659  2878  3118  3376  3658  3964  4296  4656  5046  5470  5930  6427
  6966  7550  8181  8864  9602 10399 11259 12186 13185 14260 15416 16657
 17988 19413 20937 22563 24300 26144 28099 30173 32364 34666 37100 39621
 44719 54826 72467100000
```

3. Symbol Manipulation and Recursive Function Examples

3.1 Problem: Find the first occurrence of an arbitrary string of characters in a given text.

Analysis: Let N be the number of characters in the text and T(1) ... T(N) be the text stored one character per word. Let L be the number of letters in the string which is stored one character per word in W(1) ... W(L).



The Program:

```
               DIMENSION T(720),W(30)
               NORMAL MODE IS INTEGER
               READ DATA  (VALUE OF N)
               READ FORMAT TXT,T(1)...T(N)
     ALPHA     READ DATA  (VALUE OF L)
               READ FORMAT TXT,W(1)...W(L)
               THROUGH SCAN, FOR I=1,1,I .G. N-L+1
               THROUGH TST, FOR J=0,1,J .GE. L
     TST       WHENEVER T(I+J) .NE. W(J+1), TRANSFER TO SCAN
               PRINT FORMAT OUT,I,W(1)...W(L)
               TRANSFER TO ALPHA
     SCAN      CONTINUE
               PRINT FORMAT NOT
               TRANSFER TO ALPHA
               VECTOR VALUES TXT=$72C1*$
               VECTOR VALUES OUT=$11HOCHARACTER I3,13H IS ST+RT OF 30C1*$
               VECTOR VALUES NOT = $H'OSTRING NOT FOUND'*$
               END OF PROGRAM
```

3.2 Problem: Evaluate the recursive function,

$$f(0) = 1$$

$$f(n) = f(n-1)\times n$$

Analysis: This is the definition of n! . Although n! can be evaluated directly using a THROUGH statement, in this example it will be evaluated using its recursive definition to illustrate how recursive functions can be handled in MAD.

The Program:

```
               EXTERNAL FUNCTION FACT. (N)
               NORMAL MODE IS INTEGER
               WHENEVER N .E. 0, FUNCTION RETURN 1
               SAVE RETURN
               SAVE DATA N
               T1 = FACT.(N-1)
               RESTORE DATA N
               RESTORE RETURN
               FUNCTION RETURN T1*N
               END OF FUNCTION
```

In order to use this function the calling program would have to specify a list for use in the SAVE and RESTORE statements. The following is an example of a program which uses FACT.

```
               DIMENSION LIST (100)
               NORMAL MODE IS INTEGER
               SET LIST TO LIST,100
               LIST=0
     BACK      READ FORMAT IN, NR
               PRINT FORMAT OUT, NR, FACT.(NR)
               TRANSFER TO BACK
               VECTOR VALUES IN = $I2*$
               VECTOR VALUES OUT = $4HON= I3,14HN FACTORIAL= I11*$
               END OF PROGRAM
```

3.3 Problem: To find the greatest common divisor of two integers Z and Y.

Analysis: The greatest common divisor is defined recursively by three equations:

$$\text{GCD.}(Z,Y) = \begin{cases} \text{GCD.}(Y,Z) & \text{if } Y > Z \\ Y & \text{if REM.}(Z,Y) = 0 \\ \text{GCD.}(\text{REM.}(Z,Y),Y) & \text{otherwise} \end{cases}$$

where REM.(A,B) is the remainder of A/B. The function GCD expects the arguments to be found on the temporary storage list as the two most recent additions. The use of the list as a parameter list makes the establishment of dummy variables unnecessary. This is less efficient than the usual way of defining functions but serves to remove many pitfalls encountered in using dummy variables with recursive functions.

```
EXTERNAL FUNCTION
INTERNAL FUNCTION REM.(A,B)= A - (A/B)*B
ENTRY TO GCD.
NORMAL MODE IS INTEGER
RESTORE DATA Z,Y
  WHENEVER Y .G. Z
  SAVE RETURN
  SAVE DATA Z,Y
  X = GCD.(0)
  RESTORE RETURN
  FUNCTION RETURN X
  OR WHENEVER REM. (Z,Y) .E. 0
  FUNCTION RETURN Y
END OF CONDITIONAL
SAVE RETURN
SAVE DATA REM.(Z,Y),Y
X = GCD.(0)
RESTORE RETURN
FUNCTION RETURN X
END OF FUNCTION
```

Note: When called upon for a value, a function such as GCD must have at least one argument (in this example a dummy argument of zero is used), even though the argument is never called upon. This is because GCD is the name of the function while GCD.(...) is the value of the function.

Note: The SET LIST TO statement need be executed once, either in the main program or in a subprogram (but before any use of SAVE or RESTORE), since the SAVE and RESTORE statements always refer to the current list.

An example of a program using GCD is

```
          NORMAL MODE IS INTEGER
          SET LIST TO LIST
          DIMENSION LIST (50)
S         READ DATA  (VALUES OF M AND N)
          SAVE DATA M,N
          PRINT FORMAT OUT,M,N,GCD.(0)
          TRANSFER TO S
          VECTOR VALUES OUT = $1H0,3HM= ,I7,S10,3HN= ,I7,S10,5HGCD = I7
         1*$
          END OF PROGRAM
```

3.4 <u>Problem</u>: To evaluate Tschebychev polynomials.

<u>Analysis</u>: The Tschebychev polynomial T(N,X) is defined recursively as follows:

$$T(N,X) = \begin{cases} 1 & \text{if } N = 0 \\ X & \text{if } N = 1 \\ 2\times X\times T(N-1,X)-T(N-2,X) & \text{otherwise} \end{cases}$$

It is important to understand that when an expression is written as an argument of a function its value is computed and stored in a temporary location. It is this location (or address) which is actually used as the argument of the function. The implication of this use of a temporary location is that often expressions cannot be used as arguments of recursive functions.

```
           EXTERNAL FUNCTION (N,X)
           ENTRY TO TSCHEB.
           INTEGER N,Z
           WHENEVER N .E. 0, FUNCTION RETURN 1.
           WHENEVER N .E. 1, FUNCTION RETURN X
           SAVE RETURN
           SAVE DATA N-2
           Z = N-1
           Y = 2.*X*TSCHEB.(Z,X)
           RESTORE DATA Z
           SAVE DATA Y
           M=TSCHEB.(Z,X)
           RESTORE DATA Y
           RESTORE RETURN
           FUNCTION RETURN Y-M
           END OF FUNCTION
```

A Program which uses TSCHEB is

```
           SET LIST TO LIST
           DIMENSION LIST(1000)
 BEGIN     READ FORMAT INPUT,N,X
           PRINT FORMAT OUTPUT,N,X,TSCHEB.(N,X)
           TRANSFER TO BEGIN
          RFORMATS
           VECTOR VALUES INPUT=$I6,F10.2*$
           VECTOR VALUESOUTPUT=$1H0,4HN=   I6,4H X= F10.2,
          111HOFUNCTION= F15.6*$
           END OF PROGRAM
```

Chapter IV

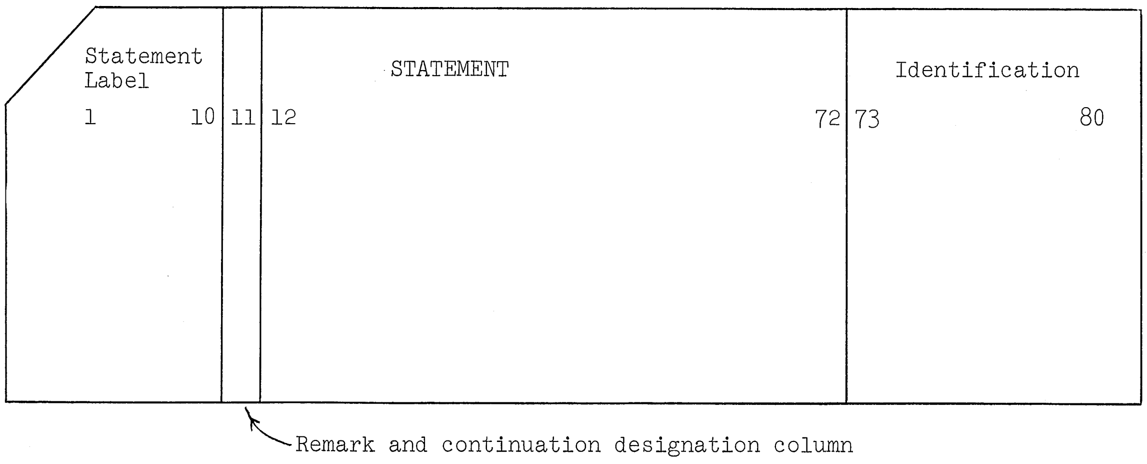
MECHANICS OF USING MAD

"My dear! I really must get a thinner pencil. I can't manage this one a bit; it writes all manner of things that I don't intend."

Lewis Carroll, Through the Looking Glass

1. Card Format

MAD Statements are punched according to the following card format:



1.1 Statement labels may be punched anywhere in Columns 1 - 10. Spaces are not relevant.

1.2 Column 11 is used to designate a remark (R) or, alternately, a continuation of a statement (0, 1, 2, ..., 9). The digits used to indicate continuation cards need not be in any particular order but there may be at most 10 cards in a statement.

Please observe that remarks are not statements, and cannot have continuation cards. Continuation cards following remarks are considered as continuations of the preceding statement. Diagnostics occurring after remarks refer to the preceding statement.

1.3 The statement may start anywhere in Columns 12 - 72. With the exception of characters enclosed between "$ ", spaces are not relevant. (Spaces may not be relevant even if they are so enclosed (see Section 2.15) in format specifications.)

1.4 The identification information in Columns 73 - 80 is not used by the computer for any purpose but printing during translation and is arbitrary. Actually, it is good practice to use the first four of the available eight columns as a mnemonic code. Presumably, the last four would contain some sequence code.

2. Diagnostics

During the process of translation many kinds of errors in the formation of statements and the allocation of storage can be detected. To understand this error detection and the subsequent printing of diagnostic comments some knowledge of the structure of the translator is helpful. The translation from statements to machine code is accomplished in three major sections:

(1) The decomposition of the original statements into arrays of binary operations and pseudo-operations.

(2) The analysis of all of the declarative information in order to allocate variable storage and identify the arithmetic types (i.e., modes) of variables.

(3) The combination of the information produced from (1) and (2) to translate the arrays to relocatable binary programs.

When an error is encountered in one of these sections the translation does not proceed to the next section. However, insofar as possible, the entire set of statements is processed through the section in which the error is detected and therefore more than one error may be detected. It should be understood then, that not all detectable errors may be found because:

(a) They are detectable only in a later stage of the translation.

(b) Some types of errors make it impossible to attempt further detection within the section in which it occurs.

(c) One error may actually obscure another error.

Occasionally, an error in one statement may be such that it causes the translator to misinterpret a second statement, thus giving an error indication even though no error exists in the later statement.

The printed diagnostic comment may very often have an alternative or ambiguous form. This results from the fact that it is frequently not possible to determine what form was intended--merely that the present structure is not admissible--and therefore some of the alternative possibilities are suggested by the comment.

A list of variables used in the program, but to which only one reference is made, is printed out for each MAD program. This list does not include any variables appearing in PROGRAM COMMON, ERASABLE, DIMENSION, VECTOR VALUES, or EQUIVALENCE statements. If READ DATA, PRINT OCTAL RESULTS, PRINT BCD RESULTS, READ AND PRINT DATA, or PRINT RESULTS are used in the program, these variables are simply printed in the list, but if none of these five types of statements is used in the program, then the variables appearing in this list are all assigned to the same location, under the assumption that they are not purposely used for anything except perhaps redundant labeling of statements. This list has proven to be a valuable debugging aid, since misspelled names almost invariably show up on this list. It should therefore be checked carefully whenever it appears.

3. Structure of Subroutines

The information in this and the following sections of Chapter IV is developed in much greater detail in other Computing Center write-ups. However, the following sections should be sufficient for the general use of MAD.

Subroutines which are written for use by MAD programs--whether written in MAD as functions or in UMAP--must be relocatable and must operate from the calling sequences the translator produces. Consider, for example, the function call--FN.(A,B,C)--which might appear in the body of a statement. Assume that B is an array which has an associated dimension vector BDIM. Using UMAP notation for illustrative purposes, the calling sequence produced would be:

```
TSX   FN,4
PAR   A
BLK   B,0,BDIM
PAR   C
```

Input-output routines utilize two types of parameters--the regional and single variable types. In addition a format specification location is given. The parameter operation code used is IOP and the end of the parameter list is indicated by an IOP operation with a blank address. Thus the statement

```
READ FORMAT FMT, BETA, X(1) ... X(100), K
```

would produce the calling sequence

```
TSX   .READ,4
IOP   FMT
IOP   BETA
IOP   X-1,0,X-100
IOP   K
IOP
```

On occasion it is useful to use the regional notation in subroutines which are not in the input-output category, for example, G.(GAMMA, DELTA, Z(10) ... Z(20)). The calling sequence would be

```
TSX   G,4
PAR   GAMMA
PAR   DELTA
BLK   Z-10,0,Z-20
```

It is important to notice that in this example, as well as in the first, the parameters --if executed as instructions--would produce no operation.

It is beyond the scope of this manual to discuss the structure of relocatable programs (see Executive System write-up). It is sufficient to say that a relocatable program must contain--in addition to the actual instructions in the program--information as to which addresses must be relocated at the time of loading for execution and which addresses must not. In addition, the first card (or record) of such programs must contain information about the size of the program, the number of subroutines it calls on, the amount of storage it will share with other subroutines, the location of the list of subroutines it calls on, and the names by which the routine itself is referred to. The symbolic names of the subroutines called on must appear as the first words after this information.

The execution of MAD programs requires the use of a loading routine to relocate and store the program and subroutines. Also, there are certain subroutines which may be automatically called for by a MAD program without an explicit reference to them in the source program.

External functions are produced in a form compatible with other subroutines, except that no attempt is made in external functions to save and restore any high-speed registers, such as index registers, sense indicators, etc.

4. System Subroutines

The use of the following names for functions (subroutines) should be avoided except where the operation is the one indicated here.

SYSTEM - Entry to this routine causes a return to the operating system. The END OF PROGRAM statement produces a call for this routine.

ERROR - Entry to this routine also causes a return to the operating system. However, if a dump of storage was requested of the operating system such a print of storage will be produced before the return to the system. The ERROR RETURN statement may produce a call for this routine.

APPENDIX A:

Allowable Abbreviations in MAD

"That is not said right," said the caterpillar.
"Not quite right, I'm afraid," said Alice timidly;
"some of the words have got altered."
Lewis Carroll, Alice in Wonderland

Abbreviations may be used for the key words or phrases of most of the commonly used statements and declarations in MAD. These abbreviations are listed below. The version of the program produced by the MAD translator as output will have the full phrase instead of the abbreviation, for easier reading. The form of the abbreviation is always the same; viz, the first and last letter of the phrase, with a prime (') between. An example of the use of these abbreviations is:

```
W'R  X .L. Y, T'O ALPHA
W'R  X .E. Y+1
     Z = J
O'R  X .E. Y+2
     Z = J+2
O'E
     Z = J+3
E'L
```

The following is the list o
not all statements and not all d

B'E	BACKSPACE FILE OF TAPE (not BACKSPACE RECORD OF TAPE)
B'N	BOOLEAN
C'E	CONTINUE
D'R	DEFINE BINARY OPERATOR (not DEFINE UNARY OPERATOR)
D'N	DIMENSION
E'L	END OF CONDITIONAL
E'N	END OF FUNCTION (not EXTERNAL FUNCTION or ERROR RETURN)
E'M	END OF PROGRAM
E'O	ENTRY TO
E'E	ERASABLE (not EXECUTE or EQUIVALENCE)
F'F	FOR VALUES OF
F'T	FLOATING POINT
F'E	FORMAT VARIABLE (not FUNCTION NAME)
F'R	FULL SYMBOL TABLE VECTOR
F'N	FUNCTION RETURN
I'O	IF LOAD POINT TRANSFER TO
I'R	INTEGER
I'N	INTERNAL FUNCTION
L'F	LISTING OFF
L'N	LISTING ON
L'T	LOOK AT FORMAT
M'R	MODE NUMBER
M'E	MODE STRUCTURE
N'S	NORMAL MODE IS
N'N	NORMAL MODE IS BOOLEAN
N'T	NORMAL MODE IS FLOATING POINT
N'E	NORMAL MODE IS FUNCTION NAME
N'R	NORMAL MODE IS INTEGER

N'L	NORMAL MODE IS STATEMENT LABEL
O'R	OR WHENEVER
O'E	OTHERWISE
P'R	PARAMETER
P'T	PRINT FORMAT (<u>not</u> PRINT COMMENT or PUNCH FORMAT or PRINT ON LINE FORMAT)
P'S	PRINT RESULTS (<u>not</u> PRINT BCD RESULTS or PRINT OCTAL RESULTS)
P'N	PROGRAM COMMON
R'A	READ AND PRINT DATA (<u>not</u> READ DATA or RESTORE DATA)
R'E	READ BINARY TAPE (<u>not</u> READ BCD TAPE or REWIND TAPE)
R'T	READ FORMAT
R'F	REFERENCES OFF
R'N	REFERENCES ON (<u>not</u> RESTORE RETURN)
S'S	SAME SEQUENCE AS
S'A	SAVE DATA
S'N	SAVE RETURN
S'O	SET LIST TO
S'E	SET LOW DENSITY TAPE (<u>not</u> SET HIGH DENSITY TAPE)
S'L	STATEMENT LABEL
S'R	SYMBOL TABLE VECTOR
T'H	THROUGH
T'O	TRANSFER TO
U'E	UNLOAD TAPE
V'S	VECTOR VALUES
W'R	WHENEVER
W'E	WRITE BINARY TAPE (<u>not</u> WRITE BCD TAPE)

APPENDIX B:

ADVANCED USES OF DIMENSION INFORMATION

"This is called teamwork. I furnish the brains. You furnish the muscles, the aches and the pains."

Dr. Seuss

Introduction

There is a simple formula which gives the relationship between the linear subscript of an element and its matrix subscripts. In what follows, assume that in each array the base element (i.e., the element with all subscripts equal to 1) has linear subscript b. For example, if $b = 6$, $A(1,1)$ would coincide with $A(6)$, $A(1,2)$ would coincide with $A(7)$, etc. In general, if A is an $m \times n$ array, and $A(i,j)$ coincides with $A(r)$, then $r = n(i - 1) + (j - 1) + b$. Thus, if in the example $b = 6$, $i = 1$, $j = 2$, then $r = n(1 - 1) + (2 - 1) + 6 = 7$, so $A(1,2)$ coincides with $A(7)$. If B is a three-dimensional $m \times n \times p$ array, and if $B(i,j,k)$ coincides with $B(r)$, then we have the formula

$$r = np(i - 1) + p(j - 1) + (k - 1) + b$$

Since r is a linear subscript, it must not assume negative values. By letting b take on larger values, one can allow i, j, and k to assume zero (and even negative) values. We may, for example, wish to determine the value that b should have to allow i to vary over the range $(-10 \ldots 20)$ and j to vary over the range $(0 \ldots 15)$ for an array A. To do this we may picture $A(-10,0)$ as the "first element". If this element is to coincide with $A(1)$, so that $A(-10,1)$ will be $A(2)$, etc., then we note that $n = 16$ (the number of columns), since $15 - 0 + 1 = 16$, and, for $r = 1$, $i = -10$, and $j = 0$, we have

$$1 = 16(-10 - 1) + (0 - 1) + b$$

$$b = 178$$

This means that $A(1,1)$ is really $A(178)$, and, for instance, $A(0,15) = A(176)$, since $16(0 - 1) + (15 - 1) + 178 = 176$.

The Dimension Vector

During the computation it is necessary that the program which handles array subscripts be able to establish the correspondence between the array subscripts and the linear subscript. Toward this end, certain information about the array A is stored in a separate vector, say in $D(k)$, $D(k + 1)$, ..., . The necessary information, as seen from the formula

$$r = n(i - 1) + (j - 1) + b$$

consists of the fact that A is 2-dimensional, together with the values of b and n. For a 3-dimensional $m \times n \times p$ array, the fact that it is a 3-dimensional array is needed, and the values of b, n, and p, since

$$r = np(i - 1) + p(j - 1) + (k - 1) + b$$

Note that the number of rows, i.e., the span of the first subscript, is not needed. In general, the "dimension vector" contains the following information:

D(k) = no. of subscripts (normally greater than 1)

D(k+1) = b

D(k+2) = no. of columns, i.e., span of the second subscript

D(k+3) = span of the third subscript

⋮

(The starting index k is arbitrary.) When the "standard" dimension declaration is used, such as

(1) DIMENSION A((-10 ... 20) * (0 ... 15))

this dimension information (with b = 178) is automatically preset in an internally created vector named .MODE1, which is also automatically declared to be of integer mode and which is itself dimensioned high enough automatically. In addition, to allow the subroutine SETDIM to check that the new settings of subscript ranges do not overflow the storage allocated to the array, the amount of storage for the array is included in the dimension information in the "decrement part" of D(k). In UMAP notation, D(k) would have the form

```
PZE 2,,497
```

if the array is dimensioned as in (1) above, since $31 \times 16 = 496$, and one extra location is allowed for A(0). We shall indicate this pair of numbers here by using brackets, e.g., [2,497]. When the array is used as an argument in a call for an external function, the address of the dimension vector, D(k), is included in the same parameter word in the call, so the function (i.e., subroutine) will automatically have access to the dimension information.

Naming the Dimension Vector

Sometimes it will be useful to be able to modify the dimension information during the execution of the program. The subroutine SETDIM allows a certain amount of modification, but occasionally it is convenient to change b, or even the number of subscripts. One might also wish to base decisions on the current values of D(k), D(k+1), etc. It is therefore useful to be able to provide an ordinary MAD symbol to be used instead of .MODE1, so that references to it may be made in other parts of the program. This is accomplished by following the dimensioning values by a comma and the name (possibly subscripted with a constant integer subscript) to be used as the name of the dimension vector. For example, one might write

```
DIMENSION V(10 * (3 ... 20), VDIM)
```

In this case, one would automatically have preset values corresponding to these assignments:

$VDIM(0) \leftarrow [2, 181]$

$VDIM(1) \leftarrow -1$

$VDIM(2) \leftarrow 18$

During execution of the program, one could then interrogate or manipulate this information in the usual way. Moreover, VDIM itself will be automatically dimensioned high enough (in this example, 2) to have storage allocated to handle this pre-set information. The subroutine SETDIM may be used as before to make the usual changes.

Occasionally, it will be useful to provide dimension information directly, via VECTOR VALUES or READ DATA or READ FORMAT statements, and bypass all of the automatic features of the standard declaration. In this case, one may declare only the total amount of storage desired for the array, together with the name of the dimension vector, such as:

DIMENSION A(100, ADIM)

Now ADIM is regarded as the head of the dimension vector for A, and its mode will automatically be integer, but no storage will be allocated for it (so that it needs a separate dimension declaration of its own), and no values will be pre-set into it. Dimension information may be stored explicitly into ADIM by a VECTOR VALUES declaration, by reading in values as data, or by computation.

Example:

The calling program for Example 5 (Chapter III) may be reprogrammed as shown below. The dimension vector for the A array is called AD, and two of its three values are preset in a VECTOR VALUES declaration. The third element (number of columns) is assigned as a result of data input. Use is made of an EQUIVALENCE declaration to relate N and AD(2).

```
                  DIMENSION A(500,AD)
                  VECTOR VALUES AD = 2,1,0
                  EQUIVALENCE (N,AD(2))
                  INTEGER N
                  FORMAT VARIABLE N
                 RTHE FOLLOWING STATEMENT READS IN THE
                 RVALUES OF  N  AND THE MATRIX  A,  PROVIDING
                 RTHEY ARE SO IDENTIFIED ON THE DATA CARDS,
                 RTHUS ALSO SETTING UP THE DIMENSION
                 RINFORMATION FOR  A.
      READ        READ DATA
                  EXECUTE TRANS.(A,N)
                  PRINT FORMAT MATRIX, A(1,1) ... A(N,N)
                  TRANSFER TO READ
                  VECTOR VALUES MATRIX = $1H0, 'N' F12.6*$
                  END OF PROGRAM
```

Other forms

In the one-dimensional case, one normally writes only the highest subscript to be used, as in the declaration

DIMENSION V(100)

and no dimension vector is provided at all. It is possible to have the automatic provision of a word containing the number of subscripts (in this case, 1) and the total amount of storage allocated (in this example, 101, for V(0), ..., V(100)). This is accomplished by writing one of the forms:

DIMENSION V(100*)

DIMENSION V(100*,VDIM)

In the first form the internally created symbol .MODE1 is used to name the one-word dimension vector, which contains [1, 101].

If more than one declaration appears for some variable, the largest amount of storage so declared is allocated, but a check is made that only one dimension vector has been named. In the automatic case, each declaration is associated with a different subscript for .MODE1, so multiple definitions of this kind cannot be allowed for the same array. If there is no dimension vector, such as for a vector dimension declaration, no check is made.

Access vectors

Each use of the subscript notation $A(i,j)$ where A is an $m \times n$ array, involves evaluation of the expression $n(i - 1) + (j - 1) + b$, where n = the number of columns in A and b is the base point, i.e., $A(1,1) \equiv A(b)$. A method which is much more efficient is to preset or compute at the beginning of the program, an auxiliary or "access" vector: $V(1) = b - 1$, $V(2) = b - 1 + n$, $V(3) = b - 1 + 2n$, ..., $V(m) = b - 1 + (m-1)n$. Note that the i^{th} element in the vector V is the linear subscript of the element in A immediately preceding the first element in the i^{th} row. To refer to $A(i,j)$ one now writes $A(V(i) + j)$. The base point and number of columns may be changed during computation, but each time either is changed the elements of V must be re-computed. The usual subscript restrictions still hold, i.e., $i \geq 0$, $(V(i) + j) \geq 0$. For example, if A is a 4×3 matrix $(m = 4, n = 3)$ and $b = 1$, then $V(1) = 0$, $V(2) = 3$, $V(3) = 6$, and $V(4) = 9$. The concept of an access vector may be extended for use with three or higher dimensional arrays. For a three-dimensional array one would need a two-dimensional access array which, in turn, could make use of its own access vector.

Storage Functions

In order to conserve storage, it may be desirable to store only one half of a symmetric matrix, or an upper (or lower) triangular matrix, etc. Perhaps only the non-zero elements of a sparse matrix might be stored. For such arrays, it is necessary to use a subscription subroutine other than the standard one supplied by MAD which expects to find the entire matrix stored by rows (in the two-dimensional case). Such special subscription subroutines may be written as any other INTERNAL or EXTERNAL FUNCTIONS in MAD or in UMAP, as usual. The arguments must be the array name, followed by the n subscripts (integer expressions) in the usual order. The value of the function which is to be computed is the linear subscript to be used to obtain the value desired. For example, if one wished to store only the upper triangle (plus diagonal) of an upper triangular two-dimensional array (by rows, starting at $A(2)$), the subscription function F could be defined as follows: (Assume a constant zero stored at $A(1)$)

$$F.(A,i,j) = \begin{cases} 1, & \text{if } i > j \\ \frac{(2(n+1)-i)(i-1)}{2} + j - i + b, & \text{if } i \leq j \end{cases}$$

where n is the number of columns in the matrix A, and b is the base point. Of course, if n is not a constant, it will have to be made available to the subscription subroutine, probably via PROGRAM COMMON, as will the base point b, if used. It will automatically be available, of course, if the subroutine is written as an INTERNAL FUNCTION. Note that in the case $i > j$, the value of F.(A,i,j) is the linear subscript 1 with the corresponding value A(1) = 0.

In order to notify MAD that such a subscription function is to be used, one merely inserts the name of the function as the first entry of the dimension vector of the appropriate array, moving the number of dimensions, etc., down by one in the dimension vector. Thus, the above example might be indicated as follows:

```
DIMENSION C(100,ADIM)

VECTOR VALUES CDIM = F.,2,2,10
```

if there were ten columns.

The following is a MAD definition for F given as an INTERNAL FUNCTION assuming it is embedded in the same program with the two preceding declarations. (Some of the statements employ abbreviations which can be found in Appendix A.)

```
 INTERNAL FUNCTION F.(A,I,J)
 EQUIVALENCE (CDIM(1),B),(CDIM(3),N)
 I'R I,J,B,N
 W'R I .G. J
 FUNCTION RETURN 1
 O'E
 FUNCTION RETURN
1 (2*(N+1) - I)*(I-1)/2+J-I + B
 E'L
 END OF FUNCTION
```

Note that the "highest subscript" of 100, which appears in the DIMENSION declaration for A, refers to the highest subscript produced by F, i.e., the storage actually used. The dimension information (i.e., 2, 2, 10) describes the array as it would be if the entire array were in storage and no storage function F were used. Thus, F uses subscripts and dimension information related to the external version of the array, and it produces a linear subscript related to the actual (reduced) storage of the array. Although dimension information could always be computed or brought in as data instead of via a VECTOR VALUES declaration, the one exception to this is now the name of the subscription function. If used, it must occur as the first entry in a VECTOR VALUES declaration, and it must be followed immediately by at least one integer in the same declaration. It should be understood that if this feature is used, references to matrix elements by means of subscripts are made as if the matrix were stored as originally described in this manual. Thus, subroutines do not need to know whether a calling program is or is not using such storage conservation functions.

The subroutine SETDIM may be used as before to make the usual changes in the dimension information.

The Library Storage Functions SYMM and TRANSP

Two special subscription functions are available from the subroutine library:

(a) SYMM will handle two-dimensional symmetric arrays for which only the upper half (plus diagonal) is stored by rows.

(b) TRANSP will handle a two-dimensional array stored by rows (in full) but considered to be in transposed form. In other words, if a matrix B is designated as having subscription function TRANSP, then a call for B(6,2) will, in fact, produce the linear subscript of B(2,6).

Both SYMM and TRANSP automatically have access to the dimension vector (including the base point), so no provision needs to be made for putting any information in PROGRAM COMMON, for example. In fact, if the subscription function is written in UMAP, it may be able to use the fact that MAD provides the address of the dimension vector in the decrement part of the parameter which contains the address of the array, in this case the first parameter. Note, however, that the dimension information has been moved down by one because of the presence of the name of the subscription function; and this must be accounted for in any references to the dimension vector.

APPENDIX C:

THE DEFINITION OF OPERATIONS

"It is a bad plan that admits of no modification."
Publelius

(This write-up replaces the appendix, <u>The Define Facility in MAD</u>, in some previous MAD manuals. This is an extended version which differs in some details from the facility described earlier.)

Introduction

It is possible, when using the MAD language, to define new operations or redefine existing operations. This allows the translator's built-in ability to decompose expressions to be used when expressions are written involving these new definitions. As a prelude to introducing the statements for expressing such definitions the following observations are made to help motivate the form they will take.

(1) In writing a translator (or any large program which is subject to frequent change) it is expeditious to include as much of the algorithm as possible in the form of tables which are then interpreted during the execution of the translator program. Changing large portions of the algorithm can then be accomplished by merely changing independent table entries as opposed to altering a highly interrelated maze of executed instructions.

(2) A basic component of mathematically oriented languages is the expression. The principal virtue of this notational construction is that, regardless of complexity, it is formed in a very regular (in fact, recursive) way from operations that are usually binary, in the sense that two operands are involved. One of the primary tasks of a translator is to decompose expressions into their constituent operations. The translation of expressions to machine instructions can then be accomplished by simply producing for every operation its equivalent in machine language.

(3) Since the operands in binary operations may also be expressions, the computation of an expression may require the temporary storage of the values of subexpressions. For instance, in $(a + b) \times (c + d)$ the values of the factors must be computed first and temporarily stored before the multiplication can be done. One of the most valuable features of a statement-type language is that these storage steps are implicit and need not be written by the programmer. Whenever possible, in the interests of efficiency, the arithmetic registers should be used for temporary storage. Thus, in the above example, $(c + d)$ could be computed and stored and then $(a + b)$ computed. The latter result need not be stored, but could be transferred to the Multiplier-Quotient register in preparation for the multiplication.

(4) Variables can be categorized by their range. The pre-defined ranges (i.e., sets of possible values) of variables in MAD are floating point, integer, Boolean, function name, and statement label. The type of range of a variable or constant is called the <u>mode</u> of the variable or constant. It is common and convenient practice to distinguish operations only by the modes of the operands. Thus, the same operator, +, is used to designate both the addition of integers and the addition of floating point numbers.

(5) A binary operation is a single valued function of two variables; that is, given two operands one result, r_1, is produced. It is sometimes useful, however, to consider this basic function as a special case of an operation where two results are produced, mainly to facilitate the treatment of mode conversions. If $(a + b)$ is to be computed, where a is an integer and b is floating point, the preliminary "operation" of conversion must be undertaken to make the operands compatible for addition. This conversion can be viewed as an operation with two outputs or results: r_1, a value for $\underline{a}$ (possibly unchanged from its original form) and r_2, a value for $\underline{b}$ (possibly unchanged from its original form). Operations requiring only one operand (unary operations) are in the special case category, also, as are the substitution or assignment "operations." (The latter, arising from constructions such as $a \leftarrow b$, are really unary, identity operations where the single result is given a name. The name, a, is certainly not an operand in the usual sense, since no value from its range was involved in the operation, nonetheless it is convenient to regard such an operation as having two operands and no result.)

(6) Operators have a precedence (or rank) associated with them. For example, $\times$ has a greater precedence than $+$ (i.e., it is executed before $+$) in the expression $(a + b \times c)$. One way to specify the precedence of an operator is to relate it to the precedence of an operator that has already been defined. Using P( ) for "precedence of," "op_1" for the new operator, and "op_2" for an existing operator, the declarations

$$P(op_1) = P(op_2),$$
$$P(op_1) < P(op_2),$$
or $$P(op_1) > P(op_2)$$

assign a precedence to op_1.

In the latter two cases, when one of the inequalities is used, it can be understood that the precedence assignment to op_1 is such that there is no other existing operator, op_3, such that

$$P(op_1) \leq P(op_3) < P(op_2),$$
or $$P(op_1) \geq P(op_3) > P(op_2).$$

Much of the rationale for permitting the definition of operations is contained in items (1) - (3), and some of the necessary statement constructions arise from the considerations of items (4) - (6). Although internal and external functions may be used to define operations, these schemes do not take advantage of the built-in capacity of the translator to translate expression structure (or <u>phrase</u> structure). The regular formation of expressions permits the use of very complex notation without the need of explicitly writing the intermediate storage steps that are required. Whenever the operation is (1) local (i.e., a component of an expression), (2) can be expressed in relatively few machine instructions (these will be included as open subroutines), and (3) the operator can be given a precedence relative to the other operators, then the operation definition will be useful. The basic scheme is to write, first, the name of the operator and its relative precedence, second, the mode structure of its operands and results and, third, the machine instructions which constitute the definition of the operation.

Operations must be defined before the first occurrence of the operation in a statement. The term "before" here is in the sense of physical order rather than order of execution, since operator definitions are, of course, relevant only at the time of translation. Notice that this implies that the same operation may have different meanings at different points in the program. Also, since the order of appearance is not necessarily the same as the order of execution, the different meanings may be interspersed in computation. The precedence of existing operations cannot be changed. Also the same symbol cannot be defined to be both a unary and binary operator even though this situation does exist with certain built-in operations.

Operator-Mode Structure

In the MAD translator expressions are decomposed into a set of binary operations (or special cases thereof) where the operands may be constants, variables, or the results of other expressions (i.e., temporary storage cells). The expression $(a + b) \times (c + d)$ written as a set of distinct operations is

$$T_1 \leftarrow a + b$$

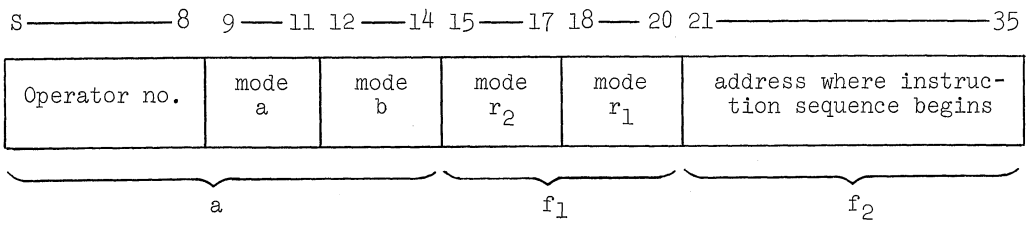
$$T_2 \leftarrow c + d$$

$$T_3 \leftarrow T_1 \times T_2$$

The "triples", as the rightmost three columns are called, are processed in top-to-bottom sequence. The modes of the operands are determined, and then the operator and these two modes are used as an argument in a table look-up to find the sequence of machine instructions that are equivalent to the operation. Actually, these three items are first converted to numeric form before the table is consulted. The numeric codes for the pre-defined modes are:

0	Floating point
1	Integer
2	Boolean
3	Function name
4	Statement label

Three additional modes (5, 6, 7) may be defined. The operators are represented internally by an integer. For example, the operator + is represented by 001 and * by 003; a complete list is given in Addendum I. Assuming that a and b are of integer mode, the argument for the table look-up obtained from the first line above would be 00111, writing the numbers in operator-mode a-mode b order. In the translator the operator-mode table (OPMD) is a table with single word entries which are packed according to the following format.



The built-in operations (i.e., pre-defined) of the MAD language are the initial values of this operator-mode table. The result of the table look-up, an address found in the f_2 section of an entry, refers to a machine instruction sequence which is stored in the TMTX region. Defining new operations consists of augmenting or altering these existing tables. Considering then, that there is a set of already defined arguments and corresponding functions, the modification can take four forms:

(1) The addition of new operator-mode arguments and corresponding instruction sequences.

(2) The addition of new operator-mode arguments which correspond to already-defined sequences.

(3) The addition of new sequences which correspond to either existing operators with new modes or existing operator-mode arguments.

(4) The assignment of an existing sequence to correspond to either existing operators with new modes or existing operator-mode arguments.

It is not possible to alter the precedence of the predefined operators.

The MAD statements for these four cases are shown below. The statement of precedence of the defined operator is included and the necessary variation for unary operations is shown. Examples are given below.

(1) $$\text{DEFINE} \begin{Bmatrix}\text{UNARY}\\ \text{BINARY}\end{Bmatrix} \text{ OPERATOR } \begin{bmatrix}\text{defined}\\ \text{op}\end{bmatrix} \text{, PRECEDENCE } \begin{Bmatrix}\text{SAME AS}\\ \text{LOWER THAN}\\ \text{HIGHER THAN}\end{Bmatrix} \begin{bmatrix}\text{existing}\\ \text{op}\end{bmatrix}$$

$$\text{MODE STRUCTURE } \begin{bmatrix}\text{mode}\\ \text{no.}\end{bmatrix} = \begin{bmatrix}\text{mode}\\ \text{no.}\end{bmatrix} \begin{bmatrix}\text{defined}\\ \text{op}\end{bmatrix} \begin{bmatrix}\text{mode}\\ \text{no.}\end{bmatrix}$$

(defining sequence)

(2) $$\text{DEFINE} \begin{Bmatrix}\text{UNARY}\\ \text{BINARY}\end{Bmatrix} \text{ OPERATOR } \begin{bmatrix}\text{defined}\\ \text{op}\end{bmatrix} \text{, PRECEDENCE } \begin{Bmatrix}\text{SAME AS}\\ \text{LOWER THAN}\\ \text{HIGHER THAN}\end{Bmatrix} \begin{bmatrix}\text{existing}\\ \text{op}\end{bmatrix}$$

$$\text{MODE STRUCTURE } \begin{bmatrix}\text{mode}\\ \text{no.}\end{bmatrix} = \begin{bmatrix}\text{mode}\\ \text{no.}\end{bmatrix} \begin{bmatrix}\text{defined}\\ \text{op}\end{bmatrix} \begin{bmatrix}\text{mode}\\ \text{no.}\end{bmatrix} \text{, SAME SEQUENCE}$$

$$\text{AS } \begin{bmatrix}\text{mode}\\ \text{no.}\end{bmatrix} \begin{bmatrix}\text{existing}\\ \text{op}\end{bmatrix} \begin{bmatrix}\text{mode}\\ \text{no.}\end{bmatrix}$$

(3) $$\text{MODE STRUCTURE } \begin{bmatrix}\text{mode}\\ \text{no.}\end{bmatrix} = \begin{bmatrix}\text{mode}\\ \text{no.}\end{bmatrix} \begin{bmatrix}\text{existing}\\ \text{op}\end{bmatrix} \begin{bmatrix}\text{mode}\\ \text{no.}\end{bmatrix}$$

(defining sequence)

(4) $$\text{MODE STRUCTURE } \begin{bmatrix}\text{mode}\\ \text{no.}\end{bmatrix} = \begin{bmatrix}\text{mode}\\ \text{no.}\end{bmatrix} \begin{bmatrix}\text{existing}\\ \text{op}\end{bmatrix} \begin{bmatrix}\text{mode}\\ \text{no.}\end{bmatrix} \text{, SAME SEQUENCE}$$

$$\text{AS } \begin{bmatrix}\text{mode}\\ \text{no.}\end{bmatrix} \begin{bmatrix}\text{existing}\\ \text{op}\end{bmatrix} \begin{bmatrix}\text{mode}\\ \text{no.}\end{bmatrix}$$

The MODE STRUCTURE statement may be varied to accommodate the special cases:

(1) The mode number to the left of the operator may be omitted when the operation is unary.

(2) The = sign and the preceding mode number may be omitted when the operation is the substitution or assignment type.

(3) When the operator-mode combination is such that a "conversion operation" is called for, the modes of the two operands after conversion are required to the left of the = sign; i.e.,

$$\begin{bmatrix}\text{mode}\\ \text{no.}\end{bmatrix} \begin{bmatrix}\text{defined}\\ \text{op}\end{bmatrix} \begin{bmatrix}\text{mode}\\ \text{no.}\end{bmatrix} = \ldots$$

Note that the form of the MODE structure statement allows the use of parameters in place of integer mode numbers. For example:

```
PARAMETER MODE6(6),MODE(5)
:
MODE STRUCTURE 1 = MODE*MODE6
```

(4) The defined operator must conform to MAD operator notation--i.e., an existing single symbol operator or an extended symbol of the form

$$.\begin{bmatrix}\text{6 or fewer}\\ \text{alphabetic char.}\end{bmatrix}.$$

fining Sequences

The instruction sequences which correspond to the operations are written and ultimately key-punched in a form that is essentially that of assembly language without a location field. The three letter mnemonic code for the machine operation appears in columns 8 - 10 of an input card; an asterisk (*) in column 11 indicates an indirectly addressed instruction; and the symbolic forms of the address, tag, and decrement appear in that order starting in column 16. These latter three items appear separated by commas and without any intervening spaces. The effect of blanks in one of the three instruction parts after the operation may be indicated by adjacent commas or by the terminal blanks in the instruction. The mnemonic designations of the permissible machine instructions are listed in Addendum II. In addition to machine instructions, three pseudo instructions may appear. It should be kept in mind that these instructions are not executed, but are simply patterns for the sequences which are included in the object program that the translator produces. As patterns, the translator must interpret every instruction in a selected sequence to determine, first, if it is a machine instruction (as opposed to a pseudo-instruction) and should therefore be included in the object program. Second, the parts of the instructions must also be interpreted so that the addresses, tags, and decrements appropriate for the particular context can (in the object program) replace the codes that appear in the defining sequences. The pseudo-instructions control the order of interpretation (JMP), specify the method of terminating the interpretation (OUT), and physically end sequences (END). Starting with the actual machine instructions, the permissible symbols in the address and decrement parts are:

FORM	MEANING
A	address of the left (A) operand.
A + 1	address of the next address in sequence.
B	address of the right (B) operand.
B + 1	address of the next address in sequence.
DT	address of the lower of two consecutive temporary storage cells.
DT + 1	address of the higher of two consecutive temporary storage cells.
T	address of a single temporary storage cell.
LOC ± [integer]	address where the instruction including this symbol will be stored in the object program, with an unsigned integer added or subtracted. The integer is taken modulo 128.
= [variable name] ± [integer]	address of the designated variable, with a non-negative integer less than 2^{15} added or subtracted. The [variable name] <u>cannot</u> be a dummy variable.
= [constant]	address of the designated constant which is written in any of the forms permitted in MAD.
= [function name]	address of the entry to the designated function. The terminal period must be included as part of the name.
± [integer]	The integer is taken modulo 128.

The tag portion of the instruction may be any of the integers 0, 1, 2, ..., 7 with exception--the STR instruction may not have a non-zero tag.

As an aid in understanding the interpretive process it is useful to further describe the meaning of the symbol T (and DT). The most direct method for assigning temporary storage addresses is to reserve an element from a linear array T for every operation in an expression. Thus, the example $(a + b) \times (c + d)$ when written in triple form would require the assignment of three locations because there are three operations involved:

$$T_1 \leftarrow a + b$$

$$T_2 \leftarrow c + d$$

$$T_3 \leftarrow T_1 \times T_2$$

Since, whenever possible, the arithmetic registers are used for temporary storage, operation sequences can be considered to terminate with the result(s) left in the arithmetic registers. If a result is not immediately used in the next operation, it must then be stored. The first step in the translation of the second operation would produce an instruction to store the result(s) of the previous operation into T_1. Thus, the symbol T, when used in translating the i^{th} instruction, can be regarded as representing the name T_{i-1}. Actually, it would be uneconomical to reserve a temporary storage cell for every operation in an expression since many would not be required due to the "connected" character of consecutive operations. Therefore, the subscript in the name T_{i-1} does not correspond to an actual position in a linear array. The symbol DT has the same interpretation except that it is the name of the first of two consecutive storage elements; DT + 1 is the name of the second of these consecutive locations. Temporary locations are reused whenever possible.

The controlling pseudo-operation JMP causes the selection of one of two possible successors in the interpretive process, depending on the value of one of six Boolean variables. The values of these variables vary during the translation process depending upon (1) the relation between operands and (2) the conditions when terminating the interpretation of a sequence. More specifically, these variables are designated in the tag portion of a JMP pseudo-instruction and are the following.

AT	=	1 if the A operand is the result of the operation represented by the preceding triple, otherwise 0.
BT	=	1 if the B operand is the result of the operation represented by the preceding triple, otherwise 0.
AC	=	1 if, during the execution of the instructions, a number would be in the accumulator at this point, otherwise 0.
MQ	=	1 if, during the execution of the instructions, a number would be in the multiplier-quotient register at this point, otherwise 0.
LA	=	1 if, during the execution of the instructions, a number would be in the logical accumulator (P bit instead of sign bit) at this point, otherwise 0.
[blank]	=	1

During the interpretation of the JMP instruction, the next instruction is given by the address if the current value of the Boolean variable given in the tag is 1, otherwise the address of the next instruction is taken from the decrement. These addresses are designated relative to the current address in the translator and are of the form *± [non-negative integer].

The terminal pseudo-operation OUT completes the interpretation of a sequence and also designates, by a code in the address part, the final state of the arithmetic registers. There are eight possible codes.

AC	-	during execution a number would be left in the accumulator (arithmetic)
MQ	-	during execution a number would be left in the multiplier-quotient register
LAC	-	during execution a number would be left in the logical accumulator
ACQ	-	numbers would be left in the accumulator and multiplier-quotient registers
SQ	-	if there would be numbers left in the accumulator and multiplier-quotient at the start of this operation, they would still be there (status quo)
Z	-	no numbers would be left in the arithmetic registers
AAC	-	the A result would be left in the accumulator
BAC	-	the B result would be left in the accumulator

The last two are necessary when two results are obtained, as in a "conversion" operation.

When two operands are incompatible, the conversion merely transforms the operands to an acceptable form; it does not cause the specified operation to be carried out. Accordingly, when either of the last two exits is taken, an automatic re-search of the operator-mode arguments is made with the conversion results as operands. Thus, the search process is continued until an ordinary binary (or unary) operation is encountered.

The pre-defined binary operation of integer addition will serve to illustrate a defining sequence. The operator-mode table entry is $\underbrace{0011}_{a}\underbrace{10}_{f_1}\underbrace{1\alpha}_{f_2}$.

The ones and zeros are octal digits. The symbol α designates the address where the first instruction of the following sequence is stored in the translator.

It is presumed here that previous results will not be left in the logical accumulator in the contexts in which this operation will appear.

```
JMP     *+2,MQ,*+1
JMP     *+3,AC,*+13
JMP     *+4,AT,*+1
JMP     *+5,BT,*+8
JMP     *+11,AT,*+1
JMP     *+4,BT,*+8
XCA
JMP     *+8
XCA
ADD     A
OUT     AC
STQ     T
JMP     *+2
STO     T
CLA     A
ADD     B
OUT     AC
END
```

Thus, if the previously interpreted sequence left a result only in the AC, and if that result were the A operand here (so that AC = AT = 1, MQ = BT = 0), the interpreter would meet in turn the following elements of the sequence:

```
JMP  *+2,MQ,*+1
JMP  *+3,AC,*+13
JMP  *+11,AT,*+1
ADD  B
OUT  AC
```

and produce as part of the object code:

```
ADD  B
```

with B replaced by the address of the B operand.

Examples:

The following two examples are complete in the sense that the mode structure statements are included as well as the defining sequence.

(1) Define a binary operation .EV. with integer operands, where the result is a bitwise "exclusive or" of the two operands. (It happens that this operation has been permanently added to the MAD language; nevertheless this example is still illuminating.)

```
        DEFINE BINARY OPERATOR .EV., PRECEDENCE SAME AS .V.
        MODE STRUCTURE 1 = 1 .EV. 1
JMP     *+7,AC,*+1
JMP     *+1,MQ,*+3
JMP     *+9,AT,*+1
JMP     *+10,BT,*+13
JMP     *+1,LA,*+4
JMP     *+4,AT,*+1
JMP     *+8,BT,*+12
STO     T
CAL     A
ERA     B
OUT     LAC
XCL
JMP     *-3
XCL
ERA     A
OUT     LAC
STQ     T
JMP     *-9
SLW     T
JMP     *-11
END
```

(2) Define a binary operation with double-precision operands of mode 5 which produces a double-precision number as a result which is the product of the two operands. The multiplication is to be accomplished by a subroutine DPM which has one operand in the AC and MQ and the other is specified in parameter form. (Remember that the MODE NUMBER 5 declaration statement permits the assignment of variables to this new mode.) Notice that this sequence allows for the possibility that the previous result may be a single value and hence not a double-precision number. The assumption is made, also, that any previous result of mode 5 which is "connected" to this operation is in both the accumulator and multiplier-quotient registers.

```
          MODE STRUCTURE 5 = 5*5
      JMP     *+2,AC,*+1
      JMP     *+4,MQ,*+5
      JMP     *+6,AT,*+1
      JMP     *+9,BT,*+1
      JMP     *+12,MQ,*+15
      STQ     T
      CLA     A
      LDQ     A + 1
      TSX     =DPM.,4
      TXH     B
      TXH     B + 1
      OUT     ACQ
      TSX     =DPM.,4
      TXH     A
      TXH     A + 1
      OUT     ACQ
      STO     DT
      STQ     DT + 1
      JMP     *-12
      STO     T
      JMP     *-14
      END
```

Diagnostics

The definition ability just described gives the programmer the ability to design, in part, his own compiler. As with any step in the direction of greater generality the door is opened for mistakes which are very difficult to diagnose. There are some checks built into the MAD translator which may indicate difficulties with sequences. The operation definer is advised to understand the cause for the following diagnostic comments. The statement

ERROR [no.] IN PART III - **[octal no.]**** ...

contains, in the string of asterisks, the twelve octal digit contents of the accumulator at the time the error was encountered. In some of the following diagnostics this information is useful in determining the difficulty.

SEQUENCE FOR OP-MODE ARGUMENT [op mode combinations] INCORRECT OR TOO LONG -- If an attempt is made to interpret an instruction outside of the table of currently defined sequences then the sequence is incorrect in some way.

Since the defined sequences are inserted as open subroutines for each occurrence of the operations, an interpretive sequence should not exceed in length some reasonable number such as 100. Loops in interpretation will cause this diagnostic comment also. The number of instructions that were interpreted will be in the accumulator.

TEMPORARY [temp.no.] SHOULD HAVE BEEN RESERVED BY OPERATOR [op.no.] IN THE STATEMENT ENDING -- When an operand is a reference to a temporary storage cell (i.e., the result of a previously computed operation) then some previous sequence, if the definitions are correct, must include a step storing that operand. Such a step would appear as STO T or some similar instruction. The erroneous statement is printed following this comment unless the statement required more than one card, in which case only the last card image is printed.

ILLEGAL OPERATION OR ADDRESS MNEMONIC IN DEFINING SEQUENCE -- When an incorrect code is used in a definition, this comment is printed. The octal representation of the offending code is in the accumulator. (See Addendum I)

OPERATOR-MODE NOT IN TABLE -- If the

$$\text{SAME SEQUENCE AS } \begin{bmatrix}\text{mode}\\ \text{no.}\end{bmatrix} \begin{bmatrix}\text{existing}\\ \text{op}\end{bmatrix} \begin{bmatrix}\text{mode}\\ \text{no.}\end{bmatrix}$$

statement is used and the designated operator-mode combination is not in the currently defined table, this diagnostic comment is printed.

INVALID OPERATOR-MODE COMBINATION $\begin{bmatrix}\text{operator-mode-}\\ \text{combination}\end{bmatrix}$ (000/M/M) IN THE STATEMENT ENDING -- This error may be caused by definition difficulties or more often, by the appearance of operands of incorrect mode in expressions involving either the pre-defined or programmer-defined operations. An operator-mode combination has been encountered which is not defined. The left three octal digits of the number inserted in the statement are the operation number, the next digit is the mode of the left operand, and the next digit is the mode of the right operand. If the erroneous statement required several cards, only the last card image is printed, but the error may be anywhere in the statement.

TEMPORARY [temp.no.] RESERVED BUT NOT USED AS AN OPERAND IN THE STATEMENT PRECEDING -- This comment does not stop translation since it is not caused by an error that would result in incorrect computation. Some sequence resulted in the inclusion of an unnecessary store operation. The line number of the offending operation is one greater than the temporary number shown. The statement involved is not the one printed but the preceding one.

Comments

The strategy for defining single and double valued operations is apparent from the examples included here and, more directly, from considering the register structure of the machine. The technique is to terminate a sequence with operations that would leave the result(s) in the arithmetic register(s) and then make it the initial task of sequence interpretation to determine what store instructions, if any, must be inserted. This basic scheme is certainly useful for experimenting with single-valued operations and for double-valued operations such as double-precision operations, complex operations, approximation-error arithmetic operations, range number computations, etc.

It is interesting to speculate on what would be necessary to extend this definition ability to arbitrary vector-valued operations. First, the limited arithmetic registers could not be used for temporary storage, and the storage into temporary registers would be an explicit part of a sequence and not relegated to the initial part of the succeeding instruction. This would require a slightly different interpretation of the symbol T, which could be readily accomplished, however. Since temporary storage would always be required, the conditional structure of such sequences would be much simpler. The other problem is, of course, that the temporary storage cells must become arrays, but here again the existing temporary assignment structure is directly usable with the addition of one facility. If, just prior to the evaluation of an expression composed of vector valued operations, the temporary storage elements are set to addresses of another storage array, spaced so that the regions between addresses can accommodate the vector results, then the single temporary cells may be used for indirect reference. By indirectly addressing all operands, statements such as

```
E = ((A*B)+C)*D
```

could be executed, where the designated variables are conformable matrices. One difficulty is that the dimension information used to set up the addresses for indirect referencing would be limited to that available at the time of translation. It is debatable whether the facility gained is worth the additional complexity of the translator in view of these restrictions and the relative ease with which functions currently may be used to specify matrix operations. However, installations using MAD may wish to experiment with adding this more general definition ability.

Pre-defined Packages

Three complete packages of definitions are available for automatic inclusion in MAD source programs. These are designed to facilitate vector and matrix arithmetic, double precision arithmetic, and complex number arithmetic. The statement

$$\text{INCLUDE } \mathcal{P}$$

where $\mathcal{P}$ is one of MATRIX, DOUBLE PRECISION, or COMPLEX will make the program behave as if the source cards for the corresponding definition package had been physically inserted into the MAD program at that point. (The word INCLUDE may be abbreviated I'E.) Complete writeups for these three packages are available in the Computing Center's 7090 Executive System Manual.

APPENDIX D:

DOUBLE STORAGE MODE DECLARATION

The format of this declaration is

DOUBLE STORAGE MODE $\mathscr{L}$

where $\mathscr{L}$ is a list of integers, each in the range 0 to 7. An occurrence of this declaration has three effects:

(1) the amount of storage allocated for each variable of the specified mode(s) is doubled,

(2) all variable and constant linear[†] subscripts applied to variables of the specified mode(s) are doubled, and

(3) in all define sequences, the expressions A + 1, B + 1 and DT + 1 are interpreted as A - 1, B - 1 and DT - 1, respectively.

Statements presetting variables with a double storage mode, such as VECTOR VALUES, etc., must now provide a pair of values (enclosed in parentheses) for each subscript position in the vector. Examples of this (for mode 1 declared to have double storage) are:

```
VECTOR VALUES V = (1,$A$), (2,$B$),(3,$C$)
VECTOR VALUES Q(3),...,Q(6) = (1,0)
```

For mode 6:

```
VECTOR VALUES M = (1.3M6,0), (2.4,1.)
VECTOR VALUES N,...,N(3) = (2.4M6,0)
```

Note that the mode will be determined by the first constant, as usual.

[†]For multi-dimensional arrays the equivalent linear subscript is doul
prior to its use.

ADDENDUM I

Pre-Defined Operators in MAD

NAME	OCTAL CODE	PRECEDENCE	OPERATION
+	001	8	$a + b \rightarrow$ AC
-	002	8	$a - b \rightarrow$ AC
*	003	9	$a * b \rightarrow$ AC
/	004	9	$a/b \rightarrow$ MQ
=	005	1	$b \rightarrow a$
.ABS.	012	14	$\|b\| \rightarrow$ AC
.P.	013	11	$a^b \rightarrow$ AC
.V.	061	12	$a \vee b \rightarrow$ LA (full word)
.A.	065	13	$a \wedge b \rightarrow$ LA (full word)
.N.	066	14	$b \rightarrow$ LA (full word)
.LS.	067	14	left shift a, b binary positions $\rightarrow$ LA (full word)
.RS.	070	14	right shift a, b binary positions $\rightarrow$ LA (full word)
.NEG.	045	10	$- b \rightarrow$ AC
..RTN.	076	not applicable	$b \rightarrow$ AC, a designates return address (mode 4)
..DIF.	035	not applicable	$a - b \rightarrow$ AC, result is always mode 2

Of the pre-defined operators only these may be referred to or redefined by definition statements. The pre-defined operators not listed constitute an unalterable basic set whose meaning (semantic content) is used in the decomposition of expressions. The two primary syntatical structures arising from this set are (1) subscription, and (2) Boolean expressions. Accordingly, the value of subscripted expressions must be of integer mode, and the operands of the Boolean operators (not the full word operations) must be of Boolean mode. Due to the method of decomposing Boolean expressions, even the operands of the relations .E., .NE., .G., .GE., .L., .LE. must be treated in a Boolean manner. The existence of the implicit operator .DIF, which is explained below, permits the use of the relations with newly defined modes. There are two other operators, also, which may not be written explicitly in a statement and yet are subject to definition.

.NEG. The symbol " - " is used in statements to indicate both the unary and the binary operator, and it is always clear from context which was intended. Some distinction must be made when the operator alone is written, and the symbol .NEG. is used for unary minus (i.e., negation).

..RTN. This symbol, which is obviously invalid in a statement, stands for the operation of placing the appropriate value(s) in the arithmetic register(s) and then returning from a function to its calling program. It is analogous to the right hand side of a substitution statement (the b operand) and then a transfer to a given address (the a operand is the address of a word whose decrement contains the complemented return address).

As such there is no result. As an example, if the result of a function were a double precision number, say mode 5, the following would be a reasonable definition.

```
    MODE STRUCTURE 4 ..RTN. 5
JMP     *+3,BT,*+1
CLA     B
LDQ     B+1
LXD     A,4
TRA     1,4
OUT     ACQ
END
```

..DIF. In the decomposition of Boolean expressions (or Boolean scan) the operations involving .AND., .OR., .NOT., and the relations are not executed as such, but are converted into tests and transfers so that only the necessary evaluations of sub-expressions are made. The relations are usually basic operands in such expressions, and the first step in determining the truth or falsity of such a relation is to form the difference of the two operands. Then the difference may be tested for some combination of the possible positive, negative and zero values. Thus, if the difference is defined for some new type of operand, and the numeric result left in the accumulator is (somewhat arbitrarily) designated to be of Boolean mode, then the relations may be used with these newly defined entities. For example, ..DIF. could be defined for double precision numbers (say, mode 5) as follows.

```
    MODE STRUCTURE  2 = 5 ..DIF. 5
JMP     *+20,AT,*+1
JMP     *+23,BT,*+1
JMP     *+4,LA,*+1
JMP     *+2,AC,*+1
JMP     *+6,MQ,*+10
JMP     *+7,MQ,*+3
SLW     T
JMP     *+7
STO     T
JMP     *+5
STQ     T
JMP     *+3
STO     DT
STQ     DT+1
CLA     A
SUB     B
TNZ     LOC+3
CLA     A+1
SUB     B+1
OUT     AC
SUB     B
TNZ     LOC+3
XCA
JMP     *-5
SUB     A
TNZ     LOC+3
XCA
SUB     A+1
CHS
OUT     AC
END
```

If, by error, attempts are made to use subscripts of other than integer mode, diagnostic statements may be produced by the translator which display numeric, pre-defined subscription operators. The following is a short list of octal subscription operator codes to aid in recognizing these cases.

024, 025, 033, 073

ADDENDUM II

Permissible Machine Instructions
(for use in operator definition)

Listed in UMAP Mnemonic Form

ACL	MPY	SXA
ADD	MSE	SXD
ADM	MZE	TIF
ALS	NOP	TIO
ANA	NZT	TIX
ANS	OAI	TLQ
ARS	OFT	TMI
AXC	ONT	TNO
AXT	ORA	TNX
CAL	ORS	TNZ
CAS	OSI	TOV
CHS	PAC	TPL
CLA	PAI	TQO
CLM	PAX	TQP
CLS	PBT	TRA
COM	PDC	TSX
DCT	PDX	TTR
DVP	PIA	TXH
ERA	PXA	TXI
FAD	PXD	TXL
FAM	PSE	TZE
FDP	PZE	UAM
FMP	RIA	UFA
FRN	RIS	UFM
FSB	RND	UFS
FSM	RQL	USM
HPR	SBM	XCA
IIA	SLQ	XCL
IIS	SLW	XEC
LAC	SSM	ZET
LAS	SSP	
LBT	STA	
LDC	STD	
LDI	STI	
LDQ	STL	
LGL	STO	
LGR	STP	
LLS	STQ	
LRS	STR	
LXA	STT	
LXD	STZ	
MPR	SUB	