

**M**ICHIGAN  
**A**LGORITHM  
**D**ECODER

NOVEMBER 1963

# The Michigan Algorithm Decoder

*“Though this be madness, yet there is method in ’t.”*

Shakespeare: *Hamlet*

The Michigan Algorithm Decoder (MAD) is a computer program which translates algebraic statements describing algorithms to 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. Certain extensions and adaptations which the authors believe make the language more useful were made, however. It should be understood in what follows that a computer program may consist of several sections utilizing the language described herein or other languages if desired. The sections can be translated independently and are linked together just prior to the actual execution of the program.

The original translating program was written for an IBM 704 computer with 8192 words of magnetic core storage, 8192 words of magnetic drum storage and 6 magnetic tapes. The program was subsequently written for the IBM 709/90 computers with 32768 words of core storage. Some of the features described here are included only in this later version. The programming and the preparation of this manual were done at The University of Michigan Computing Center and the language has been widely used by University of Michigan students and staff.

B. Arden  
B. Galler  
R. Graham

# 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 order to present a problem to a digital computer for solution, it is necessary to transmit to the machine a statement of the problem, a procedure for solving it (usually called an algorithm for the solution of that problem), and the data which is needed in the solution. In fact, a statement of the problem is not really necessary, since the algorithm and data are sufficient for a computer, provided the algorithm is 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  $A = \{a_0, a_1, a_2, \dots, a_n\}$  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:

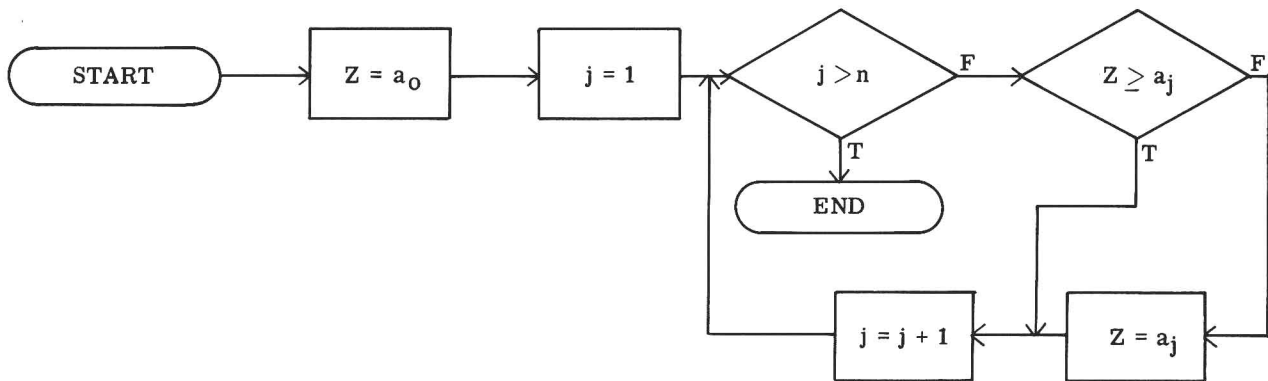
- (1) Let  $Z = a_0$ .
- (2) Let  $j = 1$ .
- (3) If  $j > n$ , the problem is done,\* go to step 7; otherwise go on.
- (4) If  $Z < a_j$ , let  $Z = a_j$ ; otherwise, go on.

\*This test is redundant the first time, but after  $n$  times through steps (3) - (6) it will terminate the process for us.






- (5) Let  $j$  increase by 1.
- (6) Return to step 3.
- (7)  $Z$  is the answer.

This may be illustrated by following “flow diagram”:



Note that the following conventions have been used here:

- A. Computation occurs in rectangular boxes. 
- B. Decisions, based on comparisons, occur in diamond-shaped boxes. 
- C. The “=” inside the computation boxes () is meant in the dynamic sense: “Compute the value of the expression on the right and let that now be the value of the variable whose name appears on the left.”

Although this problem does not require it, we frequently ask a question like “Does  $j = n$ ?” The “=” in this context is a *relation* and yields either “YES” or “NO” when placed between two arithmetic expressions.

We return to the example problem. The algorithm 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 was “ $j > n$ ”.
- (b) Before the first “iteration”, some initialization may be performed. In the example,  $Z = a_0$  and  $j = 1$ .
- (c) After the “body” of the loop is performed, (the comparison, in the example), some variable is incremented, and the termination condition is tested again. In our example,  $j$  is increased by 1, and if  $j \leq n$ , the “body” of the loop is computed again - with the new value of  $j$ .

It is often convenient to take advantage of this standard structure of a loop, and use an “iteration box” in the flow diagram. This box would have to indicate the “scope”; i.e., the extent of the body of the loop, 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 contents of a typical iteration box, (not related to the example above) might be:

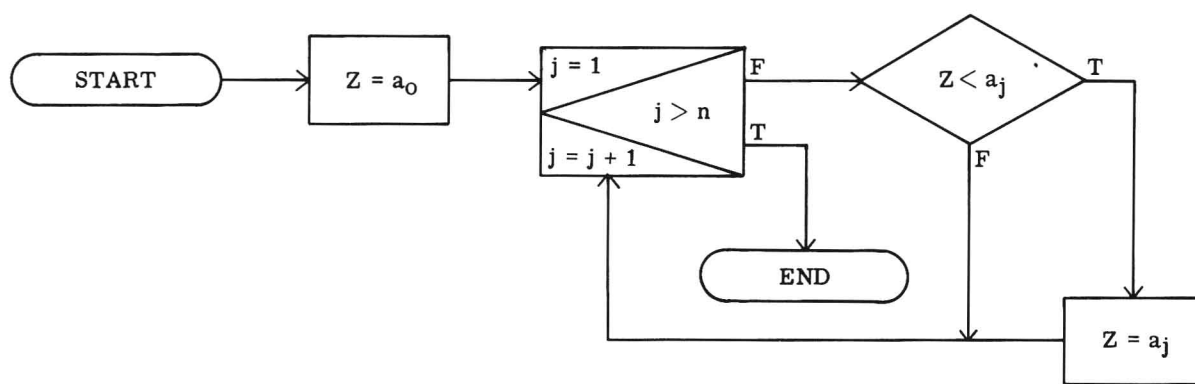
Repeat the computation through the box labeled (for example) “BACK”, starting with the variable  $\alpha$  having the value 12, and increasing it by the amount 3 after each time until either  $\alpha > 90$ , or  $|x + y| \leq \epsilon$ .

We shall abbreviate this by the form:

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

Here we have substituted “.G.” for “>”, “.LE.” for “≤”, “.ABS.” “(X+Y)” for “|x+y|”. (The reason for these simple substitutions is the lack of characters such as >, ≤ as input to the computer. Such details are fully presented in Chapter II of this manual.”

The flow diagram for our previous example, then, may be rewritten as follows: (In order to keep the notation familiar when possible, the above-mentioned character substitutions will not be made in diagrams.)



The three operators specified in a “THROUGH” statement are written in a single box to indicate the correspondence to a single statement.

The question still remains: How does one communicate the algorithm to the computer? A translator such as MAD is designed according to the philosophy: Once the algorithm has been stated, as in a flow diagram, it should be presented to the computer directly in that form, or as near to it as possible. The “translator” then has the job of producing a translation from a flow diagram representation of the algorithm to a machine language representation of the same algorithm, i.e., into the basic code of the machine. In other words, aside from the details of preparing the flow diagram in a form acceptable as input to the translator, the user’s work ends with the diagram itself.

It should be remembered that the MAD language was designed with several important criteria in mind:

- (a) Speed of translation - the choice of some words of more than six characters (e.g., WHENEVER, TRANSFER, THROUGH) enables the translator to recognize the statement type with a minimum of analysis.
- (b) Generality - as few restrictions on the construction of statements and expressions have been introduced as possible.
- (c) Ease of adding to the language - desired additions can be made easily, since most of the necessary information can be stored in tables during translation.

It is obvious that a different set of criteria could lead to a different language. The details of the form of statements in this "input language" are the subject of Chapter II.\* We are concerned here with introducing some of the basic ideas, such as the loop, etc. The input form of our example would be:

```

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

```

The DIMENSION statement assigns a block of storage in the computer large enough to handle  $a_0, a_1, \dots, a_{100}$ , if necessary. The INTEGER statement declares J and N to be integers. We have already seen that different shaped boxes are used for operations on integers, such as subscript modification and counting. The reason for this is that integer arithmetic can be done more simply and efficiently, usually with less round-off error, than arithmetic on non-integers. [Numbers which may have fractional parts are usually written in the so-called "scientific notation", such as  $3.1 \times 10^{-6}$ , are called floating point numbers.] Numbers are assumed to be in the floating-point mode unless otherwise declared, as in the INTEGER statement in the example. The WHENEVER statement above 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 and y, 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").

Returning again to the example problem on the largest of a set of numbers, we observe that no provision was made for obtaining the values of  $n, a_0, a_1, \dots, a_n$  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 are 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

\*The reader may find some clarifying effect if he rereads Chapter I after reading Chapter II.

\*\*After the logician George Boole.

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

```
EXTERNAL FUNCTION (N,A)
ENTRY TO MAX.
Z = A(0)
THROUGH BACK, FOR J = 1, 1, J .G. N
BACK WHENEVER Z .L. A(J), Z = A(J)
FUNCTION RETURN Z
INTEGER J,N
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 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 a concrete 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 repeatedly solving, by Newton's method, of the equation  $f(x) = a^x + x = 0$ , taking a different value for  $a$  each time but with the restriction that  $a \geq 1$ . This method involves the repeated evaluation of the formula

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

(the prime denotes the derivative with respect to  $x$ ) 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 approximate numbers the latter condition becomes: until  $|f(x_{i+1})| < \epsilon$ , where  $\epsilon$  is a small positive number.

To evaluate the iterative formula above 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,  $x_0, x_1, \dots, x_n$ , 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

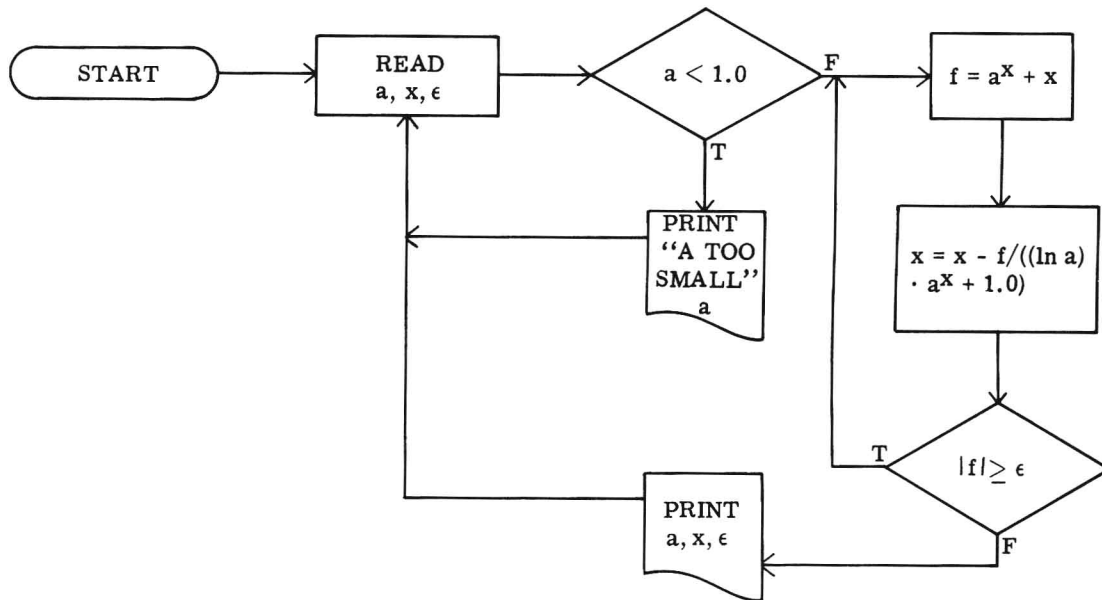
$$x \text{ is replaced by } x - \frac{f(x)}{f'(x)}$$

Often a left arrow ( $\leftarrow$ ) is used to mean "is replaced by" but in the actual statements produced for the computer the "=" symbol conventionally has this meaning. It is important to realize what the "=" means in this context; it is different from the usual use of the symbol 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 substitution.

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

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

although we will for convenience break this into two statements. The entire computational procedure can be represented by the following flow diagram.



The corresponding statements are:

```

      ST  READ FORMAT ALPHA, A, X, EPS
          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 FORMAT ALPHA, A, X, EPS
        TRANSFER TO ST
PRT     PRINT FORMAT BETA, A
        TRANSFER TO ST
        VECTOR VALUES ALPHA = $$S1,3F15.6*$
        VECTOR VALUES BETA  = $17HOA TOO SMALL, A = F15.6*$
        END OF PROGRAM
  
```

- (1) The first statement, labeled ST, causes a value for a, x, and  $\epsilon$  to be read into computer storage. A block of adjacent storage locations in the computer (a "vector") named ALPHA is designated as containing a description of how the three numbers were punched on the card. (This description - which is the value of ALPHA - is described in a later statement; see (9) below.)
- (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 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 for the function  $\log_e$  and the item in parentheses following this name indicates the variable whose natural logarithm is desired.
- (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 statement causes three numbers - the current values for  $a$ ,  $x$ , and  $\epsilon$  to be printed. The arrangement and form of the numbers on the printed page are controlled by the format description which is the value of the vector named ALPHA.
- (7) The TRANSFER statement following 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 (here  $a$  must be  $< 1.0$ ) according to a format description stored in BETA. The format description also causes the printing of some constant alphabetic information preceding the number; namely, the remark "A TOO SMALL, A = " .
- (9) The next two statements following the last TRANSFER TO ST set the initial values of two vectors named ALPHA and BETA. Although VECTOR VALUES statements can be used to preset vectors to specified numeric values, in both instances here the values are *format descriptions*. The strings of characters between the dollar signs (MAD quotation marks!) are to be taken literally as they appear and stored in designated vectors. The actual format specifications are described in more detail in Chapter II, but, for example, the first description: S1,3F15.6\* means that the first item is to be a space (S1), followed by three (3) numbers printed with a decimal point but no exponent (F), each occupying fifteen card or print positions (15), and that there will be six positions to the right of the decimal point (.6). The asterisk (\*) indicates the termination of the description.
- (10) 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 would continue until it had exhausted the given input values of  $a, x, \epsilon$ .

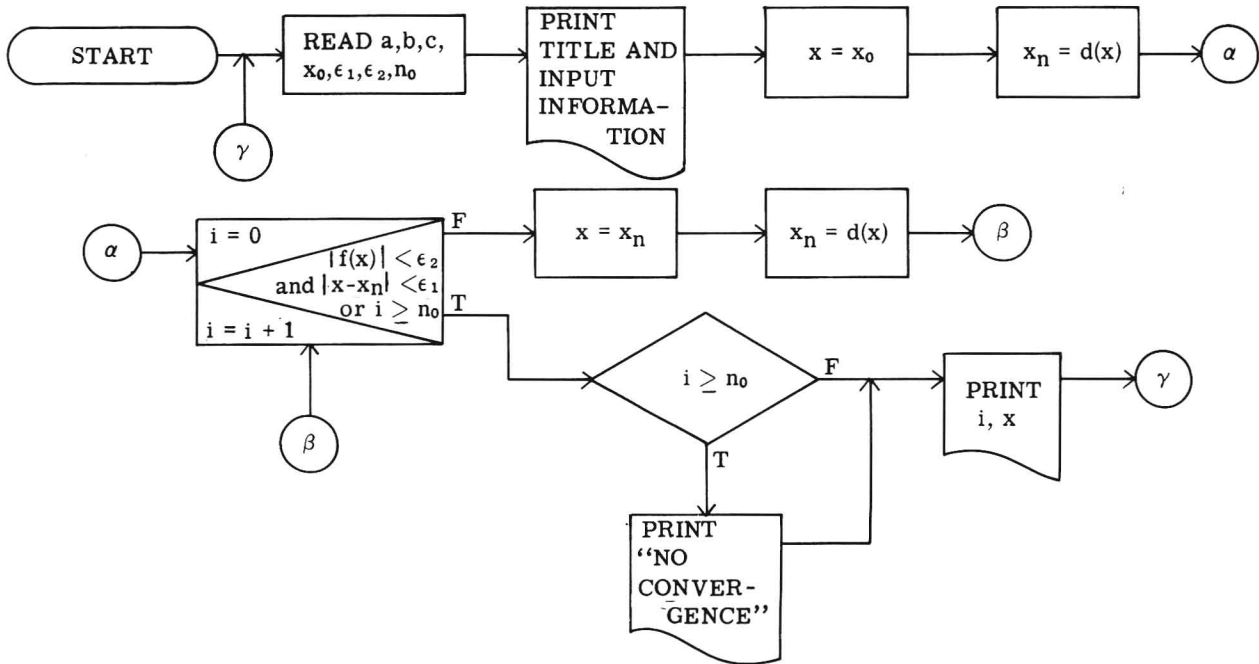
The final example before the description of the language in Chapter II is a more general and more elaborate illustration of Newton's Method. Here, we consider the solution of the equation  $x^3 + ax^2 + bx + c = 0$ , starting with some input value for  $x_0$ , using the iterative formula:

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

which is obtained from the standard Newton's Method 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 upper limit to the number of iterations which we can tolerate. We are requiring that at  $x_i$  the value of the function  $f(x_i)$  be between the upper and lower bounds  $\epsilon_2$  and  $-\epsilon_2$ . Also, the distance between  $x_{i+1}$  and  $x_i$  should be less than  $\epsilon_1$ . Observe that we are never concerned with more than two consecutive approximations to the root - say  $x$  and  $x_m$  (for next  $x$ ) - and that the incrementing of  $i$  is not really necessary.



where  $f(x) = x^3 + ax^2 + bx + c$   
 $d(z) = (2z^3 + az^2 - c)/(3z^2 + 2az + b)$

The program:

```

GAMMA      READ FORMAT CARD, A, B, C, XZERO, EPS1, EPS2, NZERO
            PRINT FORMAT TITLE, A, B, C, XZERO, EPS1, EPS2, NZERO
            X = XZERO
            NEXTX = D.(X)
            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 FORMAT REMARK
            PRINT FORMAT RESULT, I, X
            TRANSFER TO GAMMA
            INTERNAL FUNCTION F.(Y) = Y .P.3 + A * Y.P.2
    
```

```

1      +B*Y+C
      INTEGER I, NZERO
      VECTOR VALUES CARD = $6F10.5, I4*$
      VECTOR VALUES TITLE = $27H1SOLUTION OF CUBIC EQUATION
1      /4H0A = F8.3, S8, 3HB = F8.3, S8, 3HC = F8.3
2      S8, 7HXZERO = F8.3/12H0EPSILON 1 = F8.3, S8,
3      11HEPSILON 2 = F8.3, S8, 3HN = I4*$
      VECTOR VALUES RESULT = $20H0NO. OF ITERATIONS = I4,
1      S20, 3HX = F8.3*$
      VECTOR VALUES REMARK = $15H0NO CONVERGENCE*$
      END OF PROGRAM

```

The first statement, labeled GAMMA, would cause the program to read from a data card the numbers  $a$ ,  $b$ ,  $c$ ,  $x_0$ ,  $\epsilon_1$ ,  $\epsilon_2$ , and  $n_0$ . The format of the input information is to be found in a block of storage labeled "CARD", whose values are set by a VECTOR VALUES statement. The vector CARD actually has as values the format information in alphabetic form (sometimes called Hollerith, or binary-coded-decimal (BCD) form). The construction and analysis of format information is thoroughly discussed in Chapter II.

The second statement in the program causes the input information, properly labeled, to print out, preceded by the title "SOLUTION OF CUBIC EQUATION". The appropriate alphabetic information is contained in the format information which is to be found in the vector TITLE, as set by a VECTOR VALUES statement.

Two functions are defined in this program, and each is designated as an INTERNAL FUNCTION. The statement following that which is labeled BETA illustrates a conditional output statement. If the iteration is terminated because  $i \geq n_0$ , the alphabetic information 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.

The VECTOR VALUES statement, illustrated here, may be used to cause vectors (and matrices) to be initialized to any desired values (even alphabetic values) before the program computation is begun. Of course, these values could be computed or read in as input, if desired, so that the data for a problem could be preceded by its own description of format, in problems where the format may change from one set of data to another.

# Chapter II

## DESCRIPTION OF THE LANGUAGE

*“There is a pleasure sure in being made, which  
none but madmen know.”*

*Dryden: The Spanish Friar*

### 1. Constants, Variables, 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 268435455 in absolute value. 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).

##### 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(=  $5.0 \times 10^2$ ), 5.E2(=  $5.0 \times 10^2$ ).

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), - (dash), \*, /, =, ), (, ., the comma “,” and the blank space, 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. An alphabetic constant is stored internally as an integer, and any 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.

### 1.2 Variables

The name of a 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 col. 11) the statement label need not be punched on the additional cards.

### 1.4 Functions

The name of a function consists of one to six letters or digits followed by a period “.” which must be written. 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 by the proper number of arguments (see section 3.8 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 values of functions. A function name given explicitly in this form will be called a *function name constant*. (See also sec. 2.8.)

### 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 = 7, 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., -ALPHA, meaning the “negative of ALPHA.” 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 value of I. The result is again an integer.
- (i) Full word bitwise logical operations *and* and *or*, written .A. and .V., respectively, meaning the bitwise *and* and *or* of the full binary integer values of the operands. The result is again an integer.
- (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 sec. 1.10). I .LS. J means the value of I shifted *left* J binary places; i.e.,  $I \times 2^J$ . 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.

## 1.6 Arithmetic Expressions

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.
- (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 .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, Boolean individual variables, Boolean subscripted 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. EPSILON), (.ABS. (X1 - X2)/X1 .LE. EPSILON) .AND. (F.(X1) .L. EPSILON), 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. X .L. 2 .AND. 0 .L. X

### 1.9 Parentheses Conventions

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.
- .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:  $\text{SIN.}(X + Y)$  implies that after the sum  $X + Y$  is computed, the operation of actually calling the function  $\text{SIN.}$  must be performed. Similarly, the array element  $A(I + 3 \times J)$  is determined by first evaluating the subscript  $I + 3 \times J$  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  $\text{ABS.}$ ,  $\text{N.}$ , etc.

Examples:

- (1)  $\text{ABS.}(B - C)$  means  $|B - C|$ , while  $\text{ABS.}B - C$  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  $Z = (X + (Y/QA))$
- (10)  $A = -B.P.X$  means  $A = -(B^X)$ .

(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 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 or any arithmetic operation in the expression is in the floating point mode. If all operands are integer (or alphabetic), 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

$$Y + \text{GCD.}(I,J)$$

$$Y + Z - I$$

$$I + 1.$$

$$\text{GCD.}(I, J)/Z$$

are all floating point expressions while the expressions

$$I + \text{GCD.}(I, J)$$

$$(I + J)/3$$

$$I + 1$$

$$\text{GCD.}(I,J)/I$$

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 134,217,728 (i.e.,  $2^{27}$ ) 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):

$$T_1 = I * J$$

$$T_2 = T_1 / K$$

$$T_3 = 7/3$$

$$T_4 = Y + T_3$$

$$T_5 = T_4 + T_2$$

and  $T_5$  will be the value of the expression.