AN INTRODUCTION TO DIGITAL COMPUTERS AND THE MAD LANGUAGE

by

Brice Carnahan

Assistant Professor of Chemical Engineering
and Biostatistics

University of Michigan

AN INTRODUCTION TO THE ORGANIZATION AND LANGUAGES OF DIGITAL COMPUTERS

Digital computers may be viewed both as logic manipulators and as information or data processing devices. Given a meaningful list of orders or commands (a program), a general purpose digital computer is designed to accept information (data), manipulate the information logically or arithmetically as indicated by the list of commands, and display the results of the processing action. The term general purpose computer applies to that class of computing machines which has not been designed to solve a specific problem (for example, missile guidance), but rather to solve essentially any "computable" problem. Although "computability" has a rigorous mathematical meaning, an intuitive understanding of the term is adequate. A computable problem is one which can be stated mathematically or symbolically and for which a terminating solution procedure or algorithm can be outlined in an unambiguous step-by-step fashion.

While one tends to view a digital computer as a unit, i.e., a single problem-solving machine, every computer is in fact a collection of a large number of inter-connected electro-mechanical devices, all directed by a central control unit. Fortunately, an understanding of computer operation (and the ability to use a computer) does not require detailed knowledge either of electronics or of hardware construction. An overall view of the computer's organization with emphasis on function rather than electrical or mechanical details is sufficient.

Digital Computer Organization

Viewed functionally, all equipment items associated with a digital computer can be grouped into four general categories:

1. Memory
2. Input-Output
3. Arithmetic
4. Control

The machine shown in Figure 1 is a hypothetical one, but is typical of all currently available digital computers. Specific operating details for each of the many computers now in use will differ from the machine described here, and from each other. Figure 1 shows an over-all schematic of current digital computer organization. Each of the four functional categories will be discussed individually.

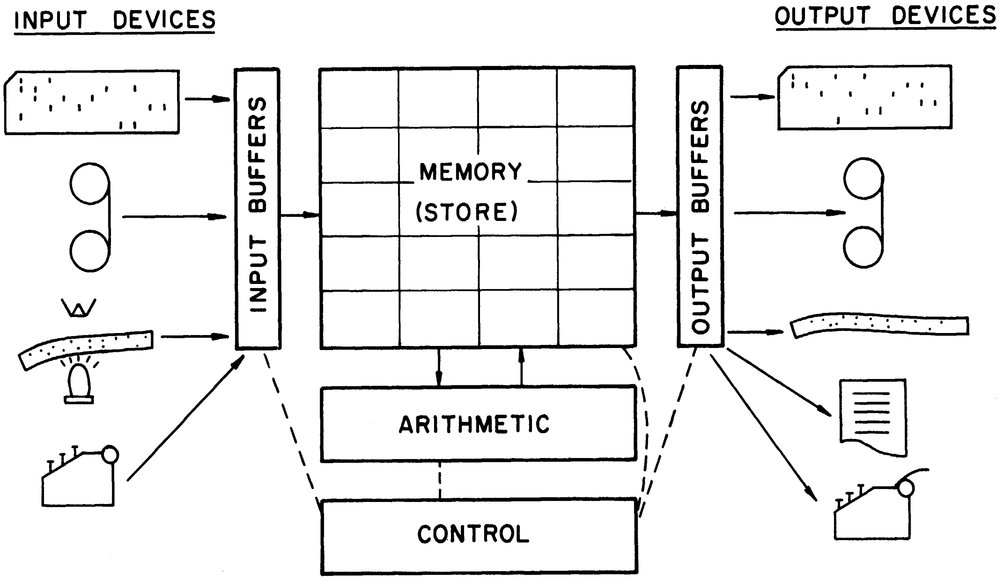


Figure 1. Over-all Schematic - Digital Computer Organization

1. Memory

The memory or store is the heart of the digital computer. The memory of most existing computers is a collection of electronic devices made of ferromagnetic materials which can be permanently magnetized by appropriate electrical impulses. Nearly all such storage elements are stable in only one of two states; the two-state polarity of the magnets produced in an individual storage element can be used to represent any binary choice, for example (+, -), (yes, no), etc. Because of the preponderance of arithmetic manipulations by computers, the polar orientation of the magnets is usually assumed to represent the two digits 0 and 1, hence, the term digital computer. The memory is simply a part of the machine where a large number of digits can be saved. Since almost any kind of information (e.g., letters, symbols, etc.) can be coded as a sequence of digits, the memory can be viewed functionally as a place to store any kind of information.

Most of the newer computers use a large number of small ferromagnetic toroids called magnetic cores as the store. Each small donut-shaped core is capable of being magnetized in one of two possible north-south polar orientations, and each can then be said to store or save one of the two digits 0 or 1. By suitable electrical impulses the polarity of the core can be reversed, and hence the stored digit changed from 0 to 1 or vice-versa. Most computers designed for scientific (as opposed to business) applications employ the binary number system, i.e., the number system with a base 2, for internal arithmetic operations. Such computers are called binary computers.

For illustrative purposes we will describe a hypothetical computer with a memory consisting of ten-state rather than two-state devices. Let each of the ten states represent one of the ten decimal digits, 0, 1, 2,...,9; then one storage element can be used to store any of the ten decimal digits. Let the total memory of this hypothetical computer consist of 10,000 such ten-state devices, so that a total of 10,000 digits may be stored in the memory at any one time.

To simplify the problem of locating any sequence of digits in the memory, the over-all collection of storage elements in most scientific computers is divided into smaller collections containing just a few, say 10, digits called cells, words, memory or machine words, storage, memory, or machine locations among others. A machine in which the number of digits in each word is not variable is said to have fixed word length. Each word in the memory is assigned a numeric address, usually in sequential order starting with address 0. In the 10,000 decimal digit memory, let the word length be fixed and equal to 10, so that the memory contains a total of 1000 words (this memory would normally be termed a "1K store" in computer parlance). Let there be a sign (+ or -) associated with each word as well. If we assign to the first collection of 10 digits, i.e., to the first word, the address 000, to the second word the address 001, to the third the address 002, etc., and to the last or 1000th word the address 999, just three digits are needed to describe the address of any of the 1000 machine words.

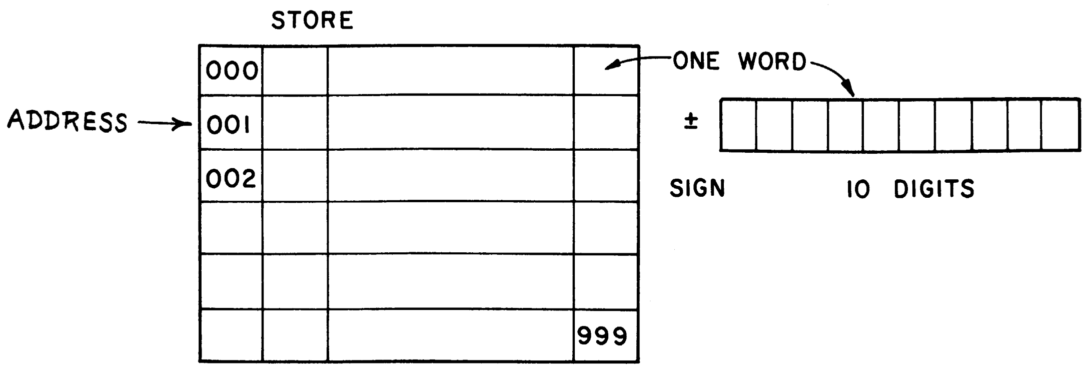


Figure 2. Division of the Memory Into Words

It is very important to distinguish between the address of a memory word and the content of the memory word. The three-digit address specifies which word in the memory is to be examined. The content of that address is the ten-digit number (plus sign) stored in the memory elements of that particular word in the store.

The stores of most digital computers are built so that the content of a memory word may be retrieved or *read* without destroying it ("non-destructive read-out"); on the other hand, when a new number is stored or *written* into a memory word, the previous content of that word is lost ("destructive read-in"). This is completely analogous to the operation of a magnetic tape recorder. Recorded information may be played back without destroying it; when a new signal is recorded over previous information, the earlier recording is destroyed or erased in the process. Note that in our machine only a finite set of numbers ($2\times10^{10} - 1$ altogether) can be represented, namely all numbers (ignoring any placing of a decimal point)

$$-9999999999 \text{ to } +9999999999 \ .$$

Thus it is not possible to represent irrational numbers (or for that matter any number with more than 10 significant digits) in the computer's memory. For this reason, information which is essentially *continuous* (e.g., an analog signal) must be put into *discrete digital* form before processing on a digital machine.

2. *Input-Output*

The function of the input-output equipment is to allow communication between the user of the machine and the store. There is a large number of such devices in use. Some of the more common input devices are (1) punched card readers, (2) punched paper tape readers, (3) typewriters, (4) magnetic tape units. Each of these devices has a substantial amount of mechanical hardware associated with it. For example, the punched card reader senses the location of the punched holes in a card by physically contacting a set of conducting brushes above the card and a platen below the card containing an electrode for each possible punched hole location. Complete contact between brush and platen is made only if a hole is present in a particular location on the card. A paper tape reader normally senses the punched holes photo-electrically. Patterns of "holes" and "no holes" in the tape can be detected as impulses of light and dark when the punched tape passes between a lamp and sensing photocells. On an input typewriter the depression of keys causes some electrical impulses to be sent to the computer.

Between these predominantly mechanical input devices and the computer's store, which operates completely electronically, there are conversion devices which we will simply label the "input buffers." The function of this buffering equipment is to accept impulses sent by the card reader, tape reader, typewriter, or any other input device, convert the impulses into appropriate internal form and store the accepted information in the memory. For example, a card might have holes punched in the first 10 columns to represent a 10-digit number. The card reader senses the locations of the holes on the card and sends some signals to the buffering equipment; subsequently the ten digits would be stored in some memory word. Which of the 1000 words is to be used is determined by the program and will be described later.

The output devices are generally quite similar to the input devices, e.g., a card punch, a magnetic tape, a paper tape punch, a typewriter, a printer, etc. Often similar input-output devices are housed in a single unit; for example, a card or paper tape read-punch unit, a magnetic tape unit which can both write and read (i.e., record and play back), etc. Between the memory and the output devices there is again some buffering equipment.

At this point our hypothetical machine consists of input devices, a memory, and output devices (with suitable buffering equipment at the interfaces). The machine can accept information from the outside, store the information in digital form in the memory, and display information present in the memory.

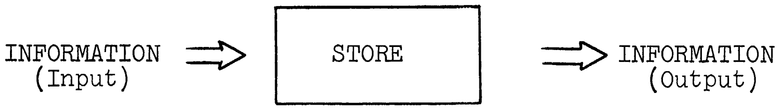


Figure 3
Information is Brought into the Memory via the Input Devices
Information in the Memory is Displayed on the Output Devices

3. Arithmetic

Obviously it is not enough to have the ability to save and retrieve information. To solve a problem we would like to read data into some words of the memory, operate on these data in some meaningful way to produce results (which could be stored in other words of the memory) and finally to display the results stored in the memory.

In order to do operations on information in the memory (for the moment these operations may be assumed to be arithmetic in nature), calculating equipment analogous to the cogs and mechanical links of a desk calculator is needed. In a digital computer, such operations are performed by strictly electronic devices. The *arithmetic unit* of a computer contains all the necessary circuitry to carry out the standard arithmetic operations on the contents of memory words (on the numbers stored in the memory) and, in addition, can perform many other manipulations such as the shifting or digitwise examination of numbers, the comparison of numbers for sign, relative magnitude and so forth. Each digital computer has a fixed number of distinctly different *operations* called *machine instructions* which the arithmetic unit is capable of executing. Some instructions are used to control the reading and writing operations of the input-output devices. Most large machines have 100-200 such operations in their instruction repertoire. The instruction repertoire for each type of machine is usually different from that of all other machines.

One may consider the arithmetic unit to contain registers (similar to the sets of dials on a desk calculator) which have immediate two-way access to any word in the memory. Most computers have at least two such registers. For example, in the IBM 7090 the most important of these are called the accumulator (AC), and the multiplier quotient unit (MQ). Operations involving addition and subtraction are done in the accumulator, which corresponds in every way to the accumulating register (one of the sets of dials) in a desk calculator. For example, the 10 digits from location 000 could be put into the accumulator, the 10 digits from location 001 could be added to the contents of the accumulator, and finally the resulting contents of the accumulator, namely the sum of the two ten-digit numbers, could be stored back into the memory in location 002. For multiplication and division operations, the accumulator and the multiplier quotient unit are both used. Some of the instructions involve other registers in the arithmetic unit. Most machines, for example, have a set of very useful counting and address modification registers called index registers.

With the addition of the arithmetic unit, the digital computer now begins to take a meaningful form. The machine is capable of reading data from the outside and entering it into the memory. The contents of various memory words can be manipulated in the arithmetic unit by means of the operations which the computer is designed to carry out. The results of these operations can be stored in the memory along with the original data and subsequently retrieved for display on the output equipment. The sequence of events is thus as follows:

1. Read data into memory via the input equipment.
2. Operate (in arithmetic unit) on data stored in memory.
3. Store results of operations in the arithmetic unit in the memory.
4. Retrieve results from the memory via the output equipment.

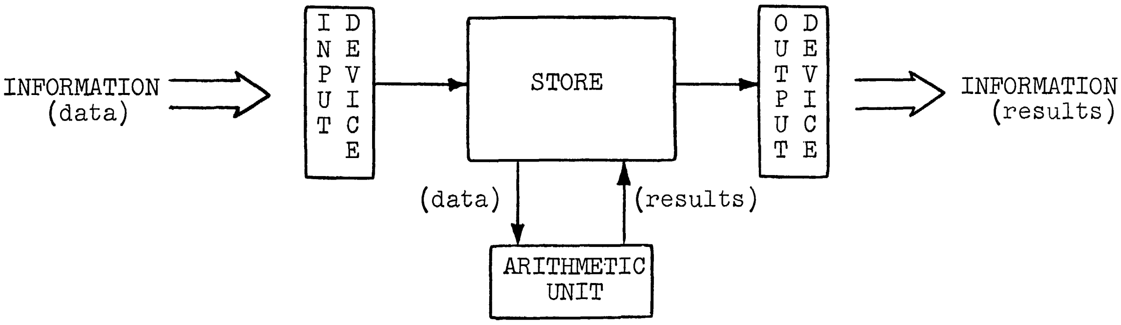


Figure 4. Flow of Information in a Digital Computer

The Control Unit

4. Control

Obviously, in order to produce useful results (to process data in a meaningful way) the computer must have associated with it a controlling device which supervises the sequence of activities taking place in all parts of the machine. This control equipment must decide (1) when (and with which input device) to bring information into the memory, (2) where to place the information in the memory, (3) what sequence of operations or manipulations on information in the memory is to be done in the arithmetic unit, (4) where intermediate or final results of operations in the arithmetic unit are to be saved in the memory , and (5) when (and on which output device) results are to be displayed.

With the addition of the control unit (see Figure 1) we now have a machine which can accept data, operate on the data to produce results, and display the results for the machine user, i.e., a machine which is capable of solving suitably stated and defined problems, given the set of commands to be carried out.

How does the machine user indicate what the machine is to do to solve his problem? First he must examine his problem and then outline a step-by-step procedure, sometimes called an algorithm, for its solution. Then he makes a list of commands from the machine's instruction repertoire (called a program) which he wants the machine to execute to implement the algorithm. The instructions must be ordered in the proper sequence and only those instructions which the machine is designed to execute, namely those in the instruction repertoire, may appear in the program.

When one uses a desk calculator, the available instructions consist of addition, subtraction, multiplication, division, shifting, clearing registers, entering the contents of the keyboard into the registers, and so forth. Unless the calculator is designed to take square roots automatically, one can not command the calculator to compute a square root. Instead, some numerical procedure which uses only the available operations is required. In a completely analogous way, a digital computer can be instructed to carry out only those instructions which have been incorporated into its design.

When using a desk calculator the sequence of instructions to be executed is determined by the machine user. The user functions as the control unit in deciding which number or operation is to be used next, etc. With the very high internal operating speeds of the digital computer (on the larger machines hundreds of thousands of individual instructions may be executed per second), it is impractical to have the machine user stand before the console pushing buttons in sequence as he does at the keyboard of a desk calculator. Consequently, some other approach is necessary to allow very rapid processing of machine instructions. Because direct communication between the machine and its environment involves the use of slow mechanical equipment, any approach which requires such contact continuously is impossibly slow. The solution to this problem is as follows. The sequence of instructions, i.e., *the program, is stored in the memory* along with the data and results. Hence, general purpose computers currently available are known as *stored program computers*. Since only numbers, i.e., decimal digits, may be stored in the machine's memory, *the instructions must be coded as digits* before being put in the memory.

How might we go about coding the sequence of instructions in the machine's repertoire? In our hypothetical machine each word in the memory contains ten digits with an associated sign. Therefore, for convenience, let us design the machine so that one coded machine instruction can be placed in one memory word, i.e., let one instruction consist of ten digits and a sign. Divide the instruction word into four segments as shown in the figure below:

Sign	1	2	3	4	5	6	7	8	9	10	Digit Position
±	θ	A	A	A	B	B	B	C	C	C	

Let the sign and the first digit (± θ) represent the *operation* which the machine is to carry out. Since there are ten possible digits which may appear in this digit position, the machine's instruction repertoire will be restricted to a total of twenty possible operations (-9,-8,...,-1,-0,+0,+1,+2,...,+9). Divide the other nine digits into groups of three, such that the digits AAA in positions 2, 3, and 4 compose the three-digit *address* of an operand A, BBB (in positions 5, 6, 7) the three-digit *address* of an operand B, and CCC (in positions 8, 9, 10) the three-digit *address* of an operand C. Let the meaning of an instruction be as shown in Table A where $\bar{A}$ means the ten-digit *contents* (with sign) of location AAA, $\bar{B}$ means the ten-digit contents (with sign) of location BBB, and $\bar{C}$ means the ten-digit contents (with sign) of location CCC.

Machine Language Instructions

Table A Instruction Repertoire

Operation Code $\pm\theta$	Operation	Meaning
+0	READ	Read $\overline{A}$, $\overline{B}$, $\overline{C}$ from one data card.*
+1	ADD	$\overline{C} \leftarrow \overline{A} + \overline{B}$
+2	SUB	$\overline{C} \leftarrow \overline{A} - \overline{B}$
+3	MPY	$\overline{C} \leftarrow \overline{A} \times \overline{B}$
+4	DIV	$\overline{C} \leftarrow \overline{A} / \overline{B}$
+5	PUNCH	PUNCH $\overline{A}$, $\overline{B}$, $\overline{C}$ on one card.**
+6	TRA	Transfer to address AAA for the next instruction.
+7	STZ	$\overline{A} \leftarrow +0000000000$ (store ten zeros in location AAA).
.	.	
.	.	
.	.	

* Let $\overline{A}$ be the ten-digit number in columns 1-10, $\overline{B}$ be the ten-digit number in columns 11-20, $\overline{C}$ be the ten-digit number in columns 21-30 of a punched card.

** $\overline{A}$, $\overline{B}$, $\overline{C}$ punched in columns 1-10, 11-20, and 21-30, respectively.

Assume that instructions are normally placed in sequence in the machine, i.e., if the first instruction is stored in location 000, the second is in location 001, the third in location 002, etc. The machine will be designed to advance to the next location automatically for its next instruction. The transfer instruction is needed to allow an instruction to be executed out of the normal sequence if desired. A machine in which instructions are placed in sequentially addressed memory words is termed a *sequential machine*.

A typical instruction might be

+3500501502

which, when interpreted as an instruction, would mean multiply ($\pm\theta = +3$) the ten-digit contents $\overline{A}$ of location 500 (AAA) by the ten-digit contents $\overline{B}$ of location 501 (BBB) and store the results as the ten-digit contents $\overline{C}$ of location 502 (CCC), retaining the ten most significant digits in case $\overline{A} \times \overline{B}$ has more than 10 significant digits. When a list of the operations or instructions which the machine is to carry out is coded in such a numeric form, the program is said to be written in the *machine's language*.

Note that we have not specified that a certain part of the memory could contain only instructions while others could contain only data or results. A coded instruction has the appearance of just another ten-digit number and cannot be distinguished a priori from some data item which might be stored in the memory. Such a storage scheme is termed ambiguous and gives the computer great power because the machine is able to treat its instructions as data; thus, if desired, the program may be written to modify its own instructions (as well as data) while it is being executed by the computer.

As an example of a simple program, consider the calculation of the following expression:

$$U = (X + Y)/Z \quad .$$

Assume that the first instruction in the program will be placed in location 000, the second in 001 and so forth, and that the value of the variables X, Y, Z, and U are to be stored in the following memory locations:

Variable	Address
X	501
Y	502
Z	503
U	504

Note that the letters X, Y, Z, U are simply symbols or names for the numeric addresses 501, 502, etc.; thus, X, Y, Z, and U are "symbolic addresses" for memory locations.* A flow diagram or algorithm which describes the complete calculation might be as shown in Figure 5. The machine language program for this algorithm is:

Instruction Address	Operation	Addresses of Operands			Action
	$\pm$ θ	AAA	BBB	CCC	
000	+7	000	000	000	Contents of 000 set to zero.
001	+0	501	502	503	Read $\overline{X}$, $\overline{Y}$, $\overline{Z}$
002	+1	501	502	505	$\overline{T} \leftarrow \overline{X} + \overline{Y}$
003	+4	505	503	504	$\overline{U} \leftarrow \overline{T} / \overline{Z}$
004	+5	501	502	503	Punch $\overline{X}$, $\overline{Y}$, $\overline{Z}$
005	+5	504	000	000	Punch $\overline{U}$
006	+6	001	000	000	Go back to instruction 001 and repeat the process.

* There is an unfortunate, but probably unavoidable, ambiguity connected with the use of such symbols. Most programmers choose (quite reasonably) to use symbolic addresses which are identical with the names of variables in the problem being solved. Hence, $\overline{X}$, the contents of symbolic address X is usually the value of a problem variable named X. Thus $\overline{X}$ = X in many (probably most) cases, a source of considerable confusion to many beginners.

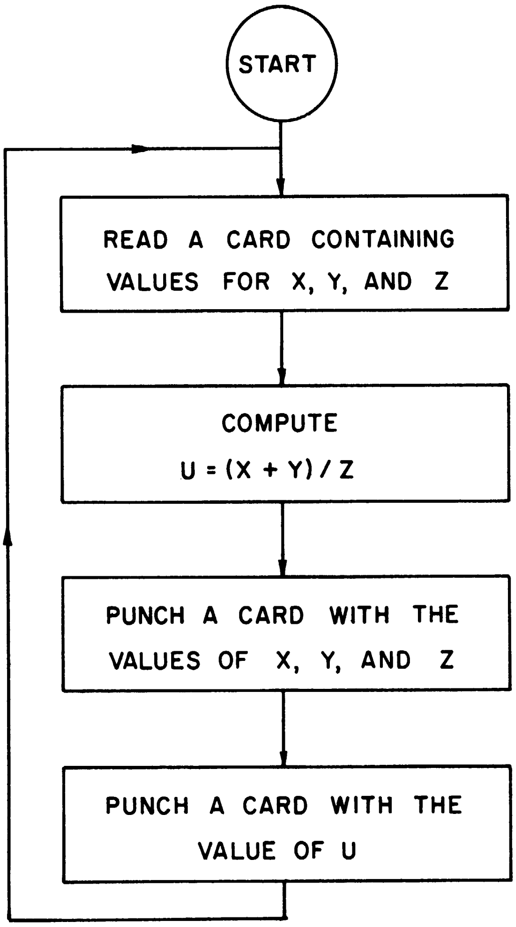

START
READ A CARD CONTAINING VALUES FOR X, Y, AND Z
COMPUTE U = (X + Y) / Z
PUNCH A CARD WITH THE VALUES OF X, Y, AND Z
PUNCH A CARD WITH THE VALUE OF U

Figure 5. Flow Diagram. Program to Compute U = (X + Y)/Z

A new symbol T was introduced in the instruction in location 002, to save the results of an intermediate calculation ($\overline{X}$ + $\overline{Y}$). T is assigned location 505. Instruction 005 will cause the value of $\overline{U}$ and 20 zeros to be punched on a card (note that the very first instruction placed 10 zeros in location 000, thus changing the first digit from 7 to 0). Having processed one data card and produced the two result cards for one set of X, Y, Z values, the program returns to location 001, reads another card with new values for X, Y, and Z and repeats the calculation. As long as there are data cards in the card reader, the machine will continue to process them, one at a time.

We have not yet specified how the program itself is read into the memory or how the machine is directed to start at location 000 for the first instruction. Each computer has the necessary circuitry built in to get started by pushing appropriate buttons on the console. This is a boot-strapping operation which is not particularly complicated and will not be described here.

It should probably be mentioned that our hypothetical machine would be called a three-address machine because three operand addresses appear in each instruction. Although such machines have been built, most machines now in use have just one operand address per instruction, and are termed <u>single</u> <u>address</u> machines. In these machines, for binary operations such as addition, the other operand is assumed to be present already in one of the registers of the arithmetic unit. For example, our single addition instruction $\overline{C} \leftarrow \overline{A} + \overline{B}$ is equivalent to three one-address instructions which might be the following:

$(\overline{AC}) \leftarrow \overline{A}$	(Put the ten-digit operand A into the accumulator (AC))
$(\overline{AC}) \leftarrow (\overline{AC}) + \overline{B}$	(Add the ten-digit operand B to the accumulator)
$\overline{C} \leftarrow (\overline{AC})$	(Store the result (left in the accumulator) in the memory location saved for operand C)

Symbolic Computer Languages

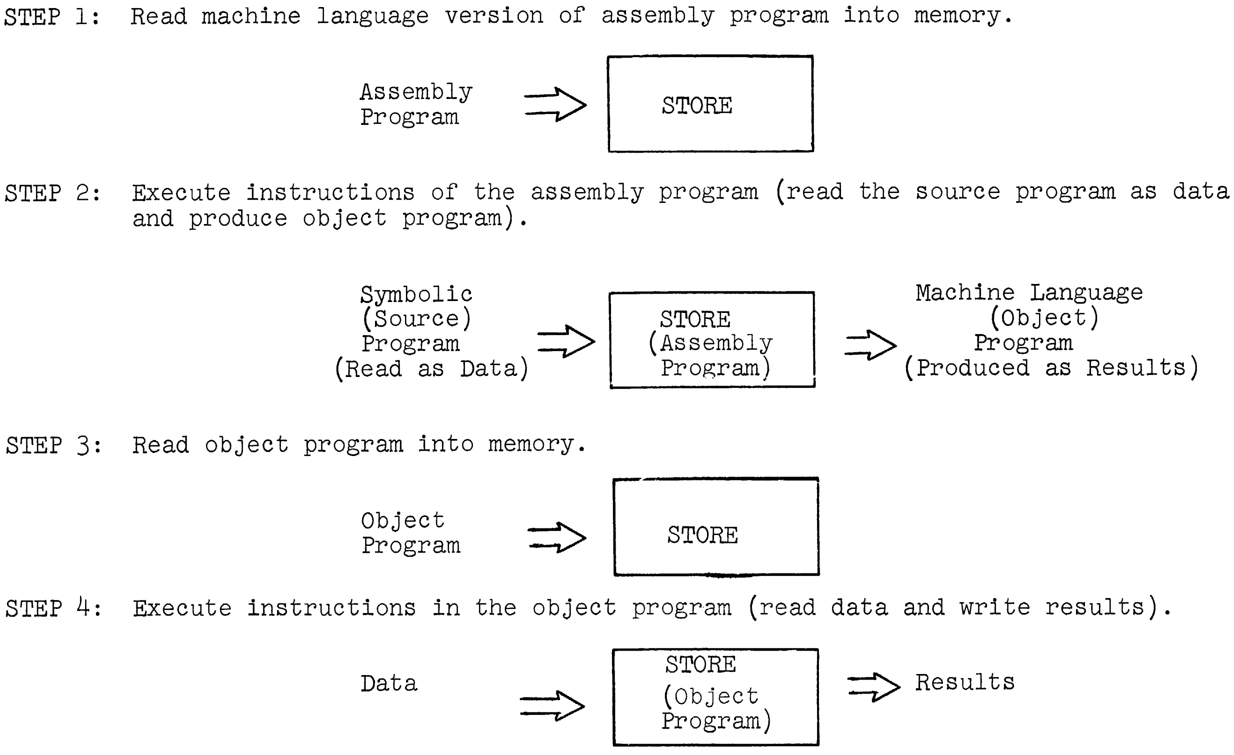
As is obvious from the preceding example, the preparation of machine language programs is very tedious. A complicated problem might require thousands or tens of thousands of such ten-digit instructions, making the writing of an error-free program virtually impossible. In addition, in order to write programs in the machine's language, one must be familiar with all (or nearly all) the instructions in the machine's instruction repertoire. In binary computers, i.e., where only the digits 1, 0 can be stored, the machine language programming problem is even more complex since a machine instruction might consist of a sequence of 30 to 60 1's and 0's in various patterns. Obviously, better ways of communicating with the computer are required to achieve any kind of programming efficiency. The first approach to this problem was the development of strictly alphabetic or symbolic languages for describing algorithms. Symbols rather than digits or sequences of digits are used to represent the operation codes and the memory locations involved. For example, we might write the program of the previous page as follows:

Symbolic Instruction Address	Operation	Operand AAA	Operand BBB	Operand CCC
START	STZ	START		
CYCLE	READ	X	Y	Z
	ADD	X	Y	T
	DIV	T	Z	U
	PUNCH	X	Y	Z
	PUNCH	U		
	TRA	CYCLE		
START	SYMB	000		
X	SYMB	501		
Y	SYMB	502		
Z	SYMB	503		
U	SYMB	504		
T	SYMB	505		

Symbolic Computer Languages

The operation SYMB is not actually one of the machine operations, and so is termed a *pseudo-operation*; the only function of SYMB is to relate the symbolic names and their corresponding numeric addresses in the memory. Having written an algorithm or procedure in this symbolic form, it is a simple mechanical problem to code each of the symbols with the appropriate digits to produce the machine language program. Because the computer itself is especially well adapted for this sort of mechanical detail, machine language programs have been written for most machines which automatically translate from the mnemonic to digital code. Such programs are termed *assembly programs* (some of the better known ones are: SOAP (IBM 650), SAP (IBM 704), and FAP (IBM 709/7090)).

The processing sequence is shown schematically in Figure 6 below. First the *machine language* version of the *assembly program* is read into the machine's memory; since the program is already in the machine's language, the machine can carry out its instructions immediately. The program to be translated is punched in the symbolic language on cards and is read as *data* by the assembly program. The letters in the symbolic language program (converted to digital form automatically by the input buffering equipment) are analyzed by the assembly program (i.e., are manipulated in the arithmetic registers according to the instructions in the assembly program) to produce the machine language equivalent. The *machine language* version of the *original symbolic program* is then read into the machine's memory for subsequent execution. The symbolic version of the algorithm is usually called the *source program*; the machine language equivalent is called the *object program*.


STEP 1: Read machine language version of assembly program into memory.

STEP 2: Execute instructions of the assembly program (read the source program as data and produce object program).

STEP 3: Read object program into memory.

STEP 4: Execute instructions in the object program (read data and write results).

Figure 6. Translation and Execution of a Symbolic Source Program

The development of assembly languages has resulted in a very great saving of time and effort by computer programmers; however, it is still necessary to have detailed knowledge of the instruction repertoire (different for each computer). Casual users would prefer to have the ability to communicate with the machine in a more familiar symbolic form using algebraic notation, English words, etc. For example, writing the program of the preceding section in a form such as

```
START     READ DATA X, Y, Z
          U = (X + Y)/Z
          PUNCH X, Y, Z, U
          TRANSFER TO START
```

would considerably simplify the programming problem. Detailed knowledge of instructions in the repertoire would be unnecessary, and, since it is fairly general, the language might hopefully be used to describe programs for more than one computer (making it possible to interchange programs with other computer users).

Machine language programs called compilers are available for most machines. These accept algorithms stated in algebraic form similar to that shown above. A compiler operates very similarly to an assembly program. A source program (this time written in the algebraic language) is first translated to an object (machine language) program. The object program is then stored in the machine's memory and the appropriate machine instructions are executed by the computer.

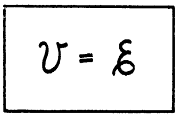
The development of compiler-level languages (sometimes called procedure or problem-oriented languages (POL)) was a tremendous breakthrough for computer users, permitting relatively inexperienced people to communicate problem solving procedures directly to the computer. One need not be an expert to use a computer effectively. This general area of computer research involving translation from symbolic languages to machine language or to other symbolic languages is often termed automatic programming. Sometimes the translating programs, i.e., assembly programs, compilers, etc. are called the software.

One compiler language called MAD (Michigan Algorithm Decoder) will be described in detail in a later section. MAD is the language used on the IBM 7090 at The University of Michigan and at several other universities as well. It is similar to most of the compiler languages now in use.

FLOW DIAGRAMS

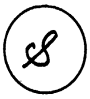
The flow (or block) diagram is the most easily understood and universally recognized form for communicating computing procedures or algorithms. A flow diagram may be defined as an unambiguous sequence of arithmetic and logical operations, expressed in ordinary algebraic notation as far as possible, inscribed in a series of characteristically shaped boxes connected by directed line segments. The particular box shapes used in this text will now be explained.

Substitution



The value of the variable $\mathcal{V}$ is replaced by the value of the expression $\mathcal{E}$.

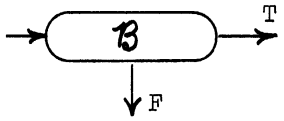
Label



(a) When occurring as $\rightarrow\circledS\rightarrow$ or $\circledS\rightarrow$, the label serves merely as an identification point in the program.

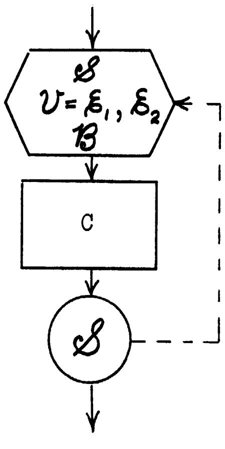
(b) If a particular branch terminates in a label, thus $\rightarrow\circledS$, then control is transferred to the one (and only one) other point in the program where $\circledS$ occurs as in (a).

Conditional Branching



If the Boolean expression $\mathcal{B}$ is true, the branch marked T is followed; otherwise, the branch marked F (false) is followed.

Iteration

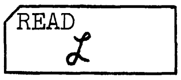


(Return line usually omitted.)

The variable $\mathcal{V}$ is first set to the value of the expression $\mathcal{E}_1$, and if the Boolean expression $\mathcal{B}$ is false, the computations C are performed. $\mathcal{V}$ is then incremented by the value of $\mathcal{E}_2$ and if $\mathcal{B}$ is still false, C is performed again. This process of incrementing (by $\mathcal{E}_2$) and testing is repeated until $\mathcal{B}$ is true, in which case C is immediately bypassed and control transferred to the next point beyond the label $\mathcal{S}$. Note that $\mathcal{E}_2$ and $\mathcal{B}$ may themselves be modified during the iteration sequence.

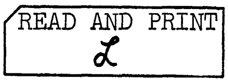
Input and Output

(a)



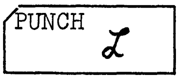
Read from cards the values for the variables comprising the list $\mathcal{L}$.

(b)



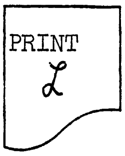
Read from cards the values for the variables comprising the list $\mathcal{L}$, and then immediately print the same values.

(c)



Punch on cards the values for the variables comprising the list $\mathcal{L}$.

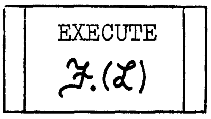
(d)



Print the values for the variables or expressions comprising the list $\mathcal{L}$. Note that the printing of comments, titles, etc. has generally been omitted from the flow diagrams in the text.

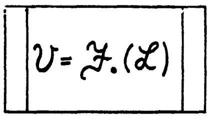
Subroutine Calls

(a)



Perform the subroutine whose name is $\mathcal{F}$., in order to modify the values of the variables appearing in the argument or parameter list $\mathcal{L}$.

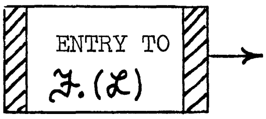
(b)



Perform the subroutine whose name is $\mathcal{F}$., in order to return a single value for $\mathcal{V}$ and possibly also to modify the values of the variables appearing in the argument list $\mathcal{L}$.

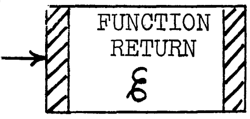
Subroutine Terminal Points

(a)



Begin the subroutine whose name is $\mathcal{F}$. and whose argument list is $\mathcal{L}$.

(b)



Terminate the subroutine and return to the calling program. The expression $\mathcal{E}$, whose value is returned by the subroutine will appear only if the call is in the form (b) above.

The flow diagram for a subroutine will be separate from that for the calling program.

Example

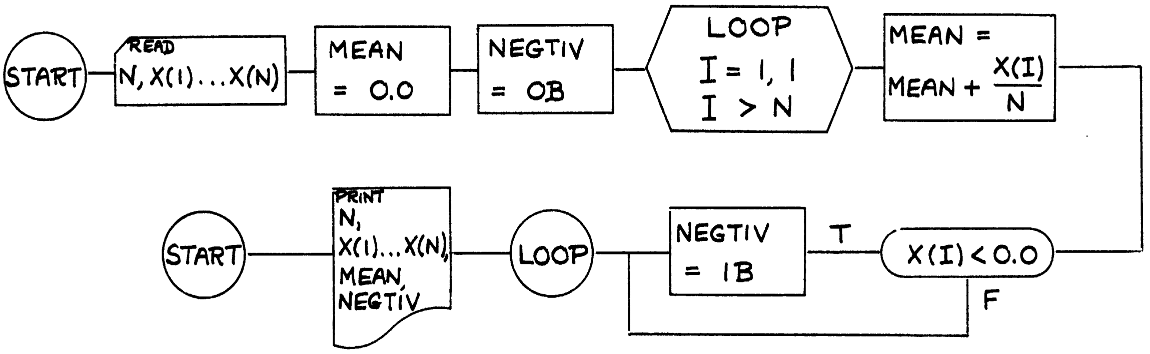
A program is to read values from cards for the variables N (integer) and for the N floating point numbers in the array X(1)...X(N). The mean (MEAN) of the numbers is to be found; also, the Boolean variable NEGTIV is to be given a true or false value (i.e., 1B or 0B) according to whether or not the array contains any negative values. The values of N, X(1)... X(N), MEAN and NEGTIV are to be printed. Provision is to be made for reading and processing as many similar additional sets of data as there may be.

Write flow diagrams which illustrate the necessary sequence of computations. Two equivalent versions are to be considered, viz.,

(a) A flow diagram corresponding to a single program.

(b) Flow diagrams corresponding to a main or calling program (which handles input and output only) together with a subroutine which performs the required computations.

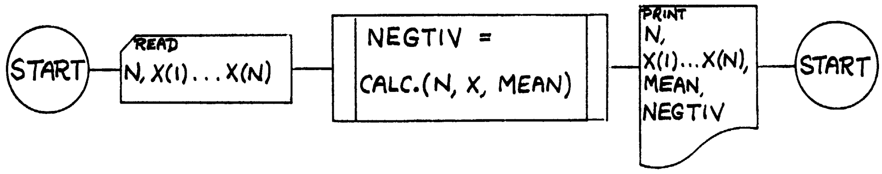
Version I



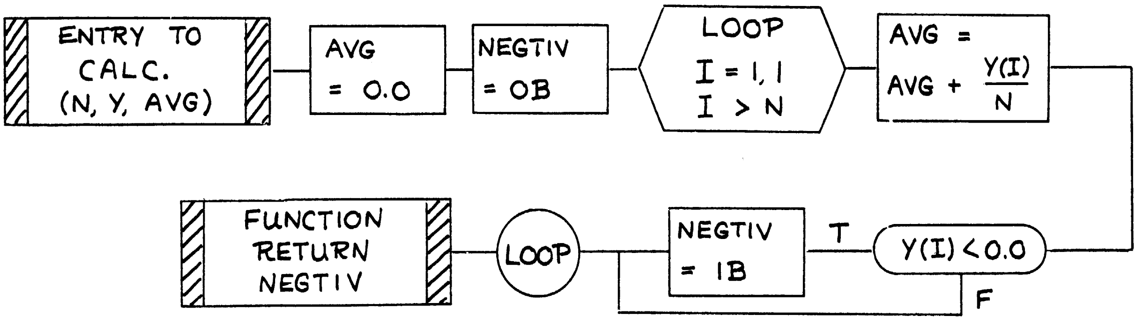
Observe that no arrows need appear in the flow diagram, since, if it is assumed that computations start at the top left hand corner, there is no ambiguity in the subsequent path.

Version II

Calling Program



Subroutine Named CALC.



Note that the argument list in the subroutine (N, Y, AVG) need not comprise the same names as appear in the calling program (N, X, MEAN). There must, however, be a one-to-one correspondence between the two, e.g., N, Y, and AVG in the subroutine are simply formal or "dummy" arguments; they may be viewed as alternative names for the actual variables N, X, and MEAN which appear in the calling program.

A SYNOPSIS OF THE MAD LANGUAGE

MAD (Michigan Algorithm Decoder) is both a language and a computer program. It is a formal language having a well-defined vocabulary and grammar designed to permit the simple and unambiguous description of procedures or algorithms. MAD is also a computer program which translates a procedure written in the MAD language (which the computer cannot interpret directly) into a sequence of machine instructions (the machine's language, which is rather difficult for people to interpret).

What follows is a description of MAD, viewed primarily as a language with little reference to the MAD program, or to the computer for that matter. Learning MAD is not unlike learning a synthetic language such as Esperanto. Fortunately MAD is far simpler and requires only a few hours of study. The vocabulary consists of English words and the grammar incorporates many of the familiar mathematical manipulations of high school algebra. Of course there are a few "rules" to learn, in order to write MAD statements (analogous to sentences in a natural language) which are grammatically correct, i.e., completely unambiguous. It should be noted that all the common higher level algebraic or algorithmic languages are very similar in structure. Once one such language is learned, transition to another is rather simple, requiring at most a few hours to learn the new vocabulary and grammar. MAD is similar to the most widely used of these languages, FORTRAN, and to ALGOL 60, an international language gaining favor as a publication language for algorithms.

A description of the principal features of the MAD language follows.

1. The Character Set:

Alphabetic: The capital letters A, B, C, . . . , Z. } alphanumeric
Numeric: The digits 0, 1, 2, . . . , 9 }
Special: + - * / ( ) . , = $ '

2. Variables:

A variable is a name or symbol which may assume different "values" at various times during implementation of a procedure. The kind of "value" which a variable may assume is determined by its "mode," to be described in the next paragraph. The function of a variable in MAD is completely analogous to that of a symbol or "unknown" in algebra.

Variable names contain 1 to 6 alphanumeric characters (the combined set of letters and digits). The first character must be alphabetic. A general reference to a variable which can assume only "numeric" values will be $\mathcal{V}$.

EXAMPLES: A, X, BETA, D105QX, ABCDE

Subscripted Variables. Any number of subscript expressions ξ_i (see paragraph 7) enclosed in parentheses may follow a variable name. Subscripts are separated by commas if there are more than one. A general reference to a subscripted variable is given by $\mathcal{V}(\xi_1, \xi_2, \ldots, \xi_n)$.

EXAMPLES: ALPHA (I), BG(2), ABC(I, J, A+B*DELTA+1)

3. Modes:

The mode of a variable refers to the form of the value which it may assume. In MAD, there are several different modes; only four need be introduced here:

INTEGER
FLOATING POINT
BOOLEAN
STATEMENT LABEL

The most important variables from the viewpoint of applied scientists are those which assume numeric values. Some variables, denoted integer variables may assume only whole or integer values, e.g., -10, 0, 1, 7520; others termed floating point variables may have fractional parts and in general assume rational-number values where the number of digits is less than or equal to 8.

Number sizes are restricted roughly as follows, where I is an integer variable and F is a floating point variable.

$$-10^9 < I < +10^9$$

$$-10^{38} < F < -10^{-38},\ F = 0.,\ 10^{-38} < F < 10^{38}$$

F is given by at most 8 significant figures.

Variables may also be of "Boolean" mode. The values assumed by such variables are.not numbers but rather "truth" values. MAD permits procedures involving Boolean algebra as well as the more conventional algebra we are most familiar with (see 12, 13 and 14).

Nothing about the name of a variable specifies its mode. It is therefore necessary to "declare" the modes of variables as described later in 21.1.

4. Constants (Numerical):

There are several kinds of MAD constants (i.e., constants of different modes); the numeric constants are described here. The mode of a constant is distinguished by its appearance alone; no mode declaration is required.

A MAD integer constant is a string of one to nine digits without a decimal point, possibly preceded by + or -.

EXAMPLES: 7, -25, 2571432, 0

A floating point constant of the F type consists of one to eight significant digits with a decimal point, possibly preceded by + or -.

EXAMPLES: 7., -15.7254, 0.0113

A floating point constant of the E or exponential typeconsists of an F type floating point number, followed by the letter E followed by an integer constant between -38 and +38. The decimal point need not appear in the fractional part, in which case it is assumed to immediately precede the letter E. The letter E is to be interpreted as "times 10 to the power," and the integer constant as an exponent of 10. Thus -4.25E-3 means (-4.25×10^{-3}). This is sometimes termed the "scientific notation" for numbers.

EXAMPLES: 7.523E-20, 1E10,-0.157254E2

Number sizes for the integer and floating point constants are restricted to the range specified for integer and floating point variables under paragraph 3.

5. Function References:

Often one needs values of a common function such as sine, cosine, square root, etc. Since operations to evaluate these functions are not built into the hardware of most computing machines (the functions must be evaluated numerically) most computing centers have a library of programs available for all users which evaluate the common functions. Thus one good program to compute sines, or square roots, etc. is written by an experienced programmer to relieve the casual user of the necessity of doing a tedious and unrewarding programming job. Such programs are called subroutines or external functions and are supplied automatically simply by making reference to them in the MAD program. As part of the MAD language, one can view such function references simply as a shorthand notation for the numerical operations required for function evaluation.

The naming convention for functions is the same as for variables except that a period is appended at the right. Arguments for the function are enclosed in parenthesis following the period. Common functions which may be called from the library directly are SIN., COS., SQRT., ELOG., EXP., ATAN. The meanings of these are as follows:

MAD Notation	Conventional Notation
SIN. ($\mathcal{E}$)	sin ($\mathcal{E}$)
COS. ($\mathcal{E}$)	cos ($\mathcal{E}$)
SQRT. ($\mathcal{E}$)	$\sqrt{\mathcal{E}}$
ELOG. ($\mathcal{E}$)	ln ($\mathcal{E}$) (logarithm to the base e)
EXP. ($\mathcal{E}$)	$e^{(\mathcal{E})}$
ATAN. ($\mathcal{E}$)	arctan ($\mathcal{E}$)

In each of these cases, the argument expression $\mathcal{E}$ (see paragraph 7), must be of floating point mode. The numeric values of each of the function references, e.g., SIN. ($\mathcal{E}$), are also of floating point mode. A general reference to a function will be $\mathcal{F}$. ($\mathcal{E}$).

EXAMPLES: SIN.(X), SQRT.(B*B-4.*A*C)

6. Arithmetic Operators:

The arithmetic operators and their precedences or the order of execution in parentheses-free expressions are, in descending order:

Conventional Notation	MAD Notation
| | (Absolute value), +	.ABS., + (unary* operators)
a^b (Exponentiation)	.P.
- (Negation)	- (as a unary operator)
x, ÷	*, /
+, -	+, - (as binary operators)

[*] "unary" implies that the operator acts on a single operand.

7. Arithmetic Expressions:

Arithmetic expressions are formed by combining constants, variables (simple or subscripted), function references, arithmetic operators and parentheses in a meaningful way. A general reference to an expression will be $\mathscr{E}$. The following recursive grammatical rules define the possible forms of arithmetic expressions.

	Examples
1. A variable is an expression.	A
2. A constant is an expression.	7.2
3. A function reference is an expression	SIN. (A)
If $\mathscr{E}$ is an expression, then	
4. ($\mathscr{E}$) is an expression.	(A)
5. + $\mathscr{E}$ is an expression.	+A
6. - $\mathscr{E}$ is an expression.	-A
7. .ABS. $\mathscr{E}$ is an expression	.ABS.A
If $\mathscr{E}_1$ and $\mathscr{E}_2$ are expressions, then	
8. $\mathscr{E}_1$ .P. $\mathscr{E}_2$ is an expression.	A.P.7.2
9. $\mathscr{E}_1$ * $\mathscr{E}_2$ is an expression.	A*7.2
10. $\mathscr{E}_1$ / $\mathscr{E}_2$ is an expression.	A/7.2
11. $\mathscr{E}_1$ + $\mathscr{E}_2$ is an expression.	A+7.2
12. $\mathscr{E}_1$ - $\mathscr{E}_2$ is an expression.	A-7.2

Note that by recursive application of these twelve rules, expressions of any complexity may be built up. For example, consider the conventional and equivalent MAD expressions

Conventional	MAD
$\dfrac{\sin\ (x + 3.1\ \sqrt{z}\ )}{(\beta + \ \ \|Q-1.\|)}$	SIN.(X+3.1*SQRT.(Z))/(BETA+.ABS.(Q-1.))

A typical sequence of rule applications might be as follows. First, translations should be made of all variables, constants, function references and arithmetic operators.

Conventional Notation	MAD Notation	Kind of Entity
sin ( )	SIN. ( )	Function Reference
x	X	Variable
3.1	3.1	Constant
$\sqrt{}$	SQRT. ( )	Function Reference
z	Z	Variable
β	BETA	Variable
Q	Q	Variable
1.	1.	Constant
| |	.ABS.	Arithmetic Operator
+	+	Arithmetic Operator

Conventional Notation	MAD Notation	Kind of Entity
Implied Multiplication	*	Arithmetic Operator
——— (Division)	/	Arithmetic Operator
( ) parentheses	( )	Parentheses (grouping marks)

By rules 1 and 2 each of the variables and constants is by definition an expression.

(a) By rule (3), SQRT.(Z) is an expression.

(b) By rule (9), 3.1 * SQRT.(Z) is an expression.

(c) By rule (11), X+3.1*SQRT.(Z) is an expression.

(d) By rule (3), SIN.(X+3.1*SQRT.(Z))is an expression.

(e) By rule (12), Q-1. is an expression.

(f) By rule (4), (Q-1.) is an expression.

(g) By rule (7), .ABS.(Q-1.) is an expression.

(h) By rule (11), BETA + .ABS.(Q-1.) is an expression.

(i) By rule (4), (BETA + .ABS.(Q-1.)) is an expression.

(j) Then combining the results of steps (d) and (i) using rule (10),

SIN.(X+3.1*SQRT.(Z))/(BETA + .ABS.(Q-1.))

is an expression.

Note that the arithmetic multiplication operation 3.1 $\sqrt{z}$, is implied in conventional notation, but must appear explicitly in MAD notation. The necessity for this rule can be seen by the ambiguity which would result in the expression yz = y•z. In MAD notation YZ without the multiplication operator (*) would be the name of a single variable (see 2.). Y*Z, on the other hand, indicates that two variables, Y and Z, are operands for the multiplication operator.

8. The Substitution Operator:

The simplest sentence or statement type in the MAD language is called the substitution statement (see 20.1 for details) and is of the form

$$\mathcal{V} = \mathcal{E}$$

Here the statement is to be interpreted as follows: Evaluate the expression $\mathcal{E}$ on the right and replace the current value of the variable $\mathcal{V}$ by this computed value, i.e., substitute the value of $\mathcal{E}$ for the current value of $\mathcal{V}$. Perhaps a better notation for the operator would be ←, i.e.,

$$\mathcal{V} \leftarrow \mathcal{E}$$

Unfortunately, there is no arrow available on the IBM keypunches, so the equal sign (=) is used instead. The precedence of the substitution operator is lower than any of the arithmetic operators of paragraph 6.

EXAMPLE: ALPHA = BETA + 3.5 * SIN.(X+Y)

9. The Concept of Precedence:

In an unparenthesized arithmetic expression, the order of arithmetic operator processing is determined by the position of the operator in the precedence list shown in paragraph 6. For example, the expression

B*C.P.ALPHA + A

would be interpreted as

$$b \times c^{\alpha} + a$$

i.e., because .P. has higher precedence than either * or +, the exponentiation operation c^{α} will be done first. Because * has higher precedence than +, $b \times c^{\alpha}$ will be done next and finally the + operator with lowest precedence will be processed to yield the final expression. If operators of equal precedence appear in an expression, then the order of processing is from left to right. For example, the expression

A*B/C*D

would be interpreted as

$$((\frac{a.b}{c})\ d)$$

Often it is necessary to override the usual rules of precedence. This can be accomplished as in conventional notation by using parentheses. When an expression is enclosed in parentheses, it is completely evaluated before being used as an operand for any operator outside the parentheses. In a sense, then, a pair of left and right parentheses can be considered as initiator of highest priority processing action, i.e., it functions as a special operator of highest precedence. The usual rules of precedence apply to the processing of the expression inside the parentheses, however. Extra parentheses may be added without fear of illegal redundancy or inefficient generation of machine language code by the MAD translating program. When parentheses are "nested," processing begins with the innermost set first and proceeds from inside to outside. As a general rule, "WHEN IN DOUBT, PARENTHESIZE."

EXAMPLE: A * B/(C * D) $= \frac{a\ .\ b}{c\ .\ d}$

10. Relational Operators:

The mathematical relations $<$, $\leqslant$, $>$, $\geqslant$, =, $\neq$, are operators used in the formation of simple or atomic Boolean (logical) expressions described in the next paragraph. Since these characters are not included in the character set (except for =) an equivalent MAD representation of the form $.\mathcal{R}.$ is used as follows:

Conventional Notation	MAD Notation
$<$	.L.
$\leqslant$	.LE.
$>$	.G.
$\geqslant$	.GE.
=	.E.
$\neq$	.NE.

11. Simple or Atomic Boolean Expressions:

An atomic Boolean expression $\mathcal{B}$ has the general form:

$$\mathcal{E}_1\ .\mathcal{R}.\ \mathcal{E}_2$$

where $\mathcal{E}_1$ and $\mathcal{E}_2$ are arithmetic expressions (see paragraph 7) and $.\mathcal{R}.$ is one of the relational operators of paragraph 10. These expressions do not have numeric values but instead have truth values, i.e., the value of the expression is either true or false.

EXAMPLE:

MAD Expression,	Conventional Notation
A*B+C.L.ALPHA*SIN.(X+Y)	$A \cdot B+C < \alpha \cdot \mathrm{SIN}(X+Y)$
I+J.E.17	I+J = 17
A.NE.0	$A \neq 0$

Note that in a Boolean expression, the relational equality is written .E., not =. The equal sign (=) is retained only for use as the substitutional operator (8) in the substitution statement described in (20.1).

12. Boolean Constants and Variables:

Constants: There are only two Boolean constants, 1B and 0B which are the values "true Boolean," and "false Boolean."

In MAD a variable may be of Boolean mode in which case it may assume only two possible values, true and false. The naming conventions are the same as described in 4. To indicate that a variable is of Boolean mode, there must be included in the program a BOOLEAN mode declaration described in 21.1.

13. Boolean Operators:

The following set of Boolean operators in the equivalent MAD notation and in order of decreasing precedence are:

Conventional Notation	MAD Notation
$\neg$	.NOT.
$\wedge$	.AND.
$\vee \; \veebar$	.OR., .EXOR.
$\supset$	.THEN.
$\equiv$	.EQV.

14. Boolean Expressions:

In a manner analogous to that for arithmetic expressions (paragraph 7),Boolean expressions are formed by combining Boolean constants, Boolean variables (simple or subscripted), Boolean function references, atomic Boolean expressions, and Boolean operators in a meaningful way. The expression definitions are, as before, recursive in nature. A general reference to a Boolean expression is $\mathcal{B}$.

		Examples
1.	A Boolean variable is a Boolean expression.	SWITCH
2.	A Boolean constant is a Boolean expression.	1B
3.	A Boolean function reference is a Boolean expression.	ALPHA.(X)
	If $\mathcal{B}$ is a Boolean expression, then	
4.	($\mathcal{B}$) is a Boolean expression.	(A.L.C)
5.	.NOT. $\mathcal{B}$ is a Boolean expression.	.NOT.(A.L.C)
	If $\mathcal{B}_1$ and $\mathcal{B}_2$ are Boolean expressions, then	
6.	$\mathcal{B}_1$ AND.$\mathcal{B}_2$ is a Boolean expression.	A.L.C.AND.Q+P.E.0

7. $\mathcal{B}_1$.OR. $\mathcal{B}_2$ is a Boolean expression. A.L.C.OR.Q+P.E.0

8. $\mathcal{B}_1$.EXOR. $\mathcal{B}_2$ is a Boolean expression. SIN.(X).LE.0.724 .EXOR.SWITCH

9. $\mathcal{B}_1$.THEN. $\mathcal{B}_2$ is a Boolean expression. X.L.I+17.THEN.A.L.C

10. $\mathcal{B}_1$. EQV. $\mathcal{B}_2$ is a Boolean expression. A.L.C.EQV.Q +P.E.0

The concept of precedence or the order of execution of operators (see 9) applies to the Boolean operators as well, and parentheses may be used to override precedence as before.

EXAMPLE: A.L.C.AND.B.E.D.OR.E.NE.Q $\equiv$ $((A < C) \wedge (B = D)) \vee (E \neq Q)$

15. A Complete Precedence List for all MAD Operators:

The most important MAD operators may be listed in a single precedence list (in decending order) as shown below. The subscription operator, which has not been specifically mentioned before, is the operator which causes subscripts on a subscripted variable to be computed before any operations can be done on the variable. Function reference evaluation refers to the operations involved in the computing of a function value, once the arguments for the function have been evaluated.

	Function Reference Evaluation - Subscription
Arithmetic Operators	.ABS., + (unary)
	.P.
	- (unary)
	* /
	+ - (binary)
Relational Operators	.L., .LE., .G., .GE., .E., .NE.
Boolean Operators	.NOT.
	.AND.
	.OR., .EXOR.
	.THEN.
	.EQV.
Substitution Operator	=

EXAMPLE: A*SIN.(X+Y).P.2.AND.GAMMA.GE.0.OR..ABS.Q.E.BETA

This will be interpreted as

$$(A \cdot \sin^2(x+y) \wedge \gamma \geq 0) \vee |Q| = \beta$$

16. Mixed Mode Arithmetic:

Arithmetic expressions involving mixed mode arithmetic, i.e., expressions in which operands are of different numeric modes (floating point or integer) are evaluated according to the following rule. With the exception of the exponentiation operation in an expression of the type

$$\mathcal{E}_f\text{ .P. }\mathcal{E}_i$$

where $\mathcal{E}_f$ and $\mathcal{E}_i$ are floating point and integer expressions, any operation involving two operands, one of type $\mathcal{E}_f$ and the other of type $\mathcal{E}_i$, will be carried out entirely in floating point form. This necessitates the conversion of the integer operand to floating point form before the operation can be executed by the machine.

It is usually not wise to mix modes unnecessarily, since additional instructions (storage) and execution time are required to handle the conversion computations. The language, however, has no restrictions on mode mixing.

It should be noted that even though a compound expression may be of mixed mode, elements of the expression may be evaluated in integer mode when two operands for some operation are both of integer mode. This normally causes no difficulty except in the case of the division operator, /. The result of the integer division operation will always be truncated to the integer next smaller in magnitude, i.e., 1/2 will be evaluated as 0, 7/2 will be evaluated as 3, -7/2 as -3, etc.

EXAMPLE:

If I and J are two integer variables having values 1 and 2 respectively and ALPHA is a floating point variable with value 7.5, then:

Expression	Value
ALPHA*I/J	3.75
I*ALPHA/J	3.75
I/J*ALPHA	0.
I+ALPHA/J	4.75
ALPHA+I/J	7.50
ALPHA+J/I	9.50

(Note that this computation appears to be identical with the first two. The order of operator processing is different, however.)

17. Elements of the MAD Language Programs:

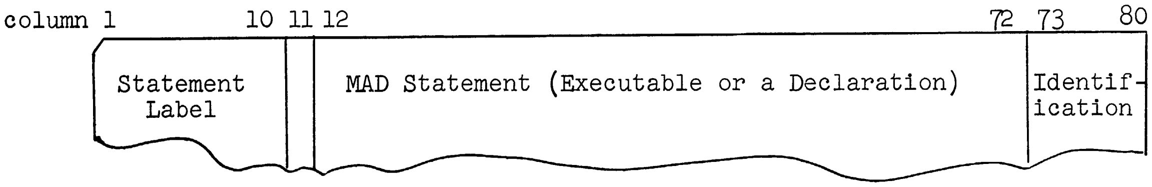
To write meaningful algorithms in the MAD language, the writer must adhere to standard statement types of rigid basic structure involving arithmetic expressions, Boolean expressions, some words from the English vocabulary, and punctuation and grouping marks. The basic statement types are very few in number and once the idea of proper arithmetic and Boolean expression formation has been mastered, the language fits into a simple pattern. There are two fundamentally different types of sentences or statements in the language, executable and non-executable. The executable statements are those statements which actually cause the MAD translating program to generate machine instructions which will be executed at the time the object or machine language version of the MAD program is loaded into the computer's memory. Non-executable statements, or declarations, are special statements which do not cause machine code to be generated, but which give information to the translating program concerning the modes of variables, the amount of space to be assigned to subscripted variables, etc.

18. Statement Labels:

MAD statements (equivalent to sentences in a natural language) which are of the executable type (see 17) may have a label, called a statement label attached to them, usually for identification or reference purposes. The naming convention for statement labels is the same as for other variables described in paragraph 4. The mode of a statement label variable is established by its location on the punched card (see 19) and it is therefore normally not necessary to include a declaration in the MAD program. A label variable may have a single (linear) subscript attached to it. A general reference to a statement label will be $\mathcal{S}$. When it appears in the statement label field on the card (see 19) any subscript on $\mathcal{S}$ must be an integer constant. A label variable $\mathcal{S}$ which does not appear in the label field must be declared in a STATEMENT LABEL mode declaration (see 21.1).

19. The Punched Card Format for MAD:

The format for MAD programs punched on IBM cards is as shown below.



Column	Contents
1-10	Blank or a statement label.
11	If this column contains an R: A remark or comment can appear in columns 12-72. If this column contains a digit, 0, 1, 2, ..., 9: This card contains a continuation of a statement from an earlier card or cards; digit order is ignored but no more than 10 cards may be used in one statement. A blank card is interpreted as a remark card, even when no R is punched in column 11.
12-72	MAD Declaration or executable statement (blanks are ignored).
73-80	Ignored by the translating program; may contain any information (e.g., an ordered number for the card).

20. Executable MAD Statements:

A basic set of MAD statement types will be described using the general script references

$\mathcal{V}$	variable name (arithmetic mode)
$\mathcal{V}_B$	variable name (Boolean mode)
$\mathcal{E}$	arithmetic expression
$\mathcal{B}$	Boolean expression
$\mathcal{S}$	statement label
$\mathcal{K}$	a numeric or Boolean constant
$\mathcal{L}$	a list of variable names or expressions or constants

for the various parts of the statements. Any typed characters (letters, punctuation marks, etc.) shown in the general formulation of the basic statement types are part of the format or basic statement structure and must not be altered in any way.

20.1 The Substitution Statement:

The substitution statement is of general form

$$\mathcal{V} = \mathcal{E}$$

or

$$\mathcal{V}_{\mathcal{B}} = \mathcal{B}$$

where the variable $\mathcal{V}$ is of either integer or floating point mode. The value of the expression on the right is computed, and then "substituted" into the variable on the left, i.e., the variable assumes the new computed value. In the first case, if $\mathcal{V}$ is different in numeric mode (floating point, integer) from $\mathcal{E}$, the appropriate conversion from the mode of $\mathcal{E}$ to the mode of $\mathcal{V}$ will be made automatically before substitution of the value into $\mathcal{V}$. Note that the substitution statement I = I + 1 is completely unambiguous and grammatically correct; the equal sign implies a dynamic rather than a logical equality. The execution of this statement type is much like formula evaluation. Everything on the right is computed first; then the substitution is made.

EXAMPLES:

```
ALPHA = BETA + SIN.(X)/Q
B(3) = B(3) + INCR
A(I,3,K+5*P) = 1.0 + GAMMA
SWITCH = A .E. O .AND. B .LE. ALPHA .OR. D*G .G. H
```

(In the last case SWITCH must be a Boolean variable.)

20.2 The Transfer Statement:

The Transfer statement is of general form:

TRANSFER TO $\mathcal{S}$

Here $\mathcal{S}$ is the statement label of another statement in the program. Ordinarily, statements are processed in the order they are written, i.e., from first card to last. However, when the procedure is executed and a statement of this type is encountered, the next part of the algorithm to be executed begins at the statement labeled $\mathcal{S}$.

EXAMPLES:

```
TRANSFER TO START
TRANSFER TO STAT(1)
TRANSFER TO STAT(J)
```

20.3 The Simplified Input Statement:

a) The simplified input statement is of the form:

READ DATA

or

READ DATA $\mathcal{L}$

Execution of this statement at running time will cause the machine to read successive data cards containing the numeric and/or Boolean values of variables in the program. These values for variables in the program are read by the program from data cards which follow and are not part of the MAD program proper, i.e., the data cards are kept separate from the statement cards. Only the first 72 columns of the data card are used and the variable values are punched on the data card in the form

$$\mathcal{V}_1 = \eta_1, \mathcal{V}_2 = \eta_2, \ldots, \mathcal{V}_n = \eta_n$$

where $\mathcal{V}$ or $\mathcal{V}_B$ is the name of a variable and η is its numeric or Boolean value. For sequences of subscripted variable values, values having one or two subscripts can be simplified to the form

$$\mathcal{V}_k = \eta_1, \eta_2, \eta_3, \ldots, \eta_n$$

If there is more data than can be conveniently punched on one data card, more than one card may be used; the only restriction is that a variable name $\mathcal{V}$ or a numeric value η not be split between two successive cards. A single READ DATA statement will cause continued reading of successive data cards until an asterisk (*) is encountered on a data card. At this point the reading of data terminates and the program proceeds to the next statement.

For program clarity, it is sometimes convenient to append a list $\mathcal{L}$ of variable names separated by commas to the READ DATA statement. This list causes no action to be taken, but does serve as a reminder of variables which should appear on the data cards.

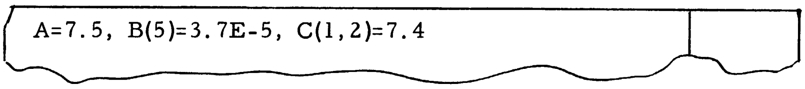
EXAMPLES:

```
READ DATA
READ DATA A, B(5), C(J,K), N, Q(1)...Q(N)
```

Example Data Card Set:

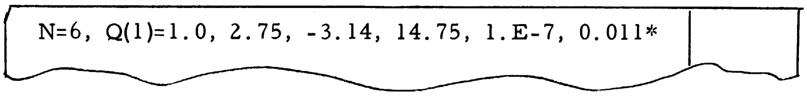
column 1 72 73 80

1st Card



```
A=7.5, B(5)=3.7E-5, C(1,2)=7.4
```

2nd Card



```
N=6, Q(1)=1.0, 2.75, -3.14, 14.75, 1.E-7, 0.011*
```

b) Another input statement is of the form:

READ AND PRINT DATA

or

READ AND PRINT DATA $\mathcal{L}$

This statement causes data cards to be read exactly as does the READ DATA statement. In addition, a copy of the card will be printed as soon as it is read, for clearer identification of the data in the computer output.

20.4 The Simplified Output Statement:

The simplified output statement is of general form

PRINT RESULTS $\mathcal{L}$

where the list $\mathcal{L}$ may contain any of the following:	EXAMPLE
1. a simple variable (numeric or Boolean)	ALPHA
2. a subscripted variable (numeric or Boolean)	A(I, J)

3. a constant (numeric or Boolean)	7.2, 6, 1B
4. a "block" of variables (numeric or Boolean)	Q(1,1)...Q(N,M)
5. an arithmetic expression	C*D/SIN.(X)

List elements are separated by commas, but no comma follows the word RESULTS. When the statement is executed, the current values of the listed variables or expressions are printed with a label on the output sheet from the computer, e.g.,

ALPHA = 4.25000, A(1,2) = 1.72513E-20

Expressions or constants are labeled with three dots

... = 7.20000, ... = 6, ... = 1B

Printing is in the same order as the list elements in the MAD statement. There is no control over page spacing using this statement; all output lines are automatically double spaced.

20.5 The Simplified Comment Statement:

The comment statement is used to write a one-line comment containing any characters in the set, (see 1.) except the \$, and has the general form:

PRINT COMMENT $\$\alpha_0,\alpha_1,\ldots\alpha_n\$$

where the string of characters $\alpha_1, \alpha_2, \ldots, \alpha_n$ $(n < 132)$ is the comment to be printed at the point in the execution of the algorithm where the statement occurs. The first character, α_0, is used for page spacing and is not printed. It may have any of the fol lowing values with subsequent action by the printer.

α_0	Printer Action
blank	Paper will be single spaced before comment is printed.
0	Paper will be double spaced before comment is printed.
4	Comment will be printed at the beginning of the next quarter page.
2	Comment will be printed at the beginning of the next half page.
1	Comment will be printed at the top of the next page.

EXAMPLE:

PRINT COMMENT $1THIS IS THE SOLUTION TO PROBLEM 1$

The comment

THIS IS THE SOLUTION TO PROBLEM 1

will be printed at the top of the next page.

20.6 The Simple or One-Line Conditional:

The simple or one-line conditional is of general form:

WHENEVER $\mathcal{B}$, $\mathcal{Q}$

where $\mathcal{B}$ is any Boolean expression and $\mathcal{Q}$ is any executable statement except another conditional statement, the END of PROGRAM statement, or one of the iteration statements. Note that only one statement is permitted. The comma following the Boolean expression must be present in order to distinguish it from the leading statement of the compound conditional group described in paragraph 20.7.

EXAMPLES:

```
WHENEVER A.L.B.AND.C*GAMMA.G.75., TRANSFER TO START
WHENEVER I.G.75, J=J+1
WHENEVER SIN.(A)+B.E.Z, PRINT RESULTS N, Q, SIN.(A)+B
```

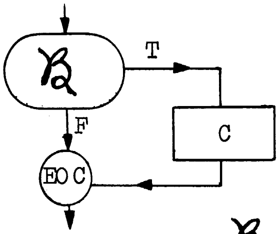
20.7 Compound Conditional Group:

The general conditional group is formed of four distinct statement types:

WHENEVER $\mathcal{B}_1$
OR WHENEVER $\mathcal{B}_2$
OTHERWISE
END OF CONDITIONAL

The initial statement of the group (WHENEVER $\mathcal{B}_1$) must not be followed by a comma. The absence of a comma indicates that the statement begins a compound conditional group and that there will be an END OF CONDITIONAL statement following to terminate the conditional statement group. The simplest form of the compound conditional statement group is:

WHENEVER $\mathcal{B}$
... C } any number of MAD Statements
...
END OF CONDITIONAL



In this case the statements inside the conditional group (C) will be executed only if the expression $\mathcal{B}$ is true. Otherwise computation will by-pass the statements inside the group and continue at the END OF CONDITIONAL statement. The END OF CONDITIONAL statement is executable in the sense that it may have a statement label attached to it, but it causes no execution to take place or any machine code to be generated. Transfer statements may be included in the body of C. Note that the one-line conditional is simply a shorthand form for the case where C contains only one statement.

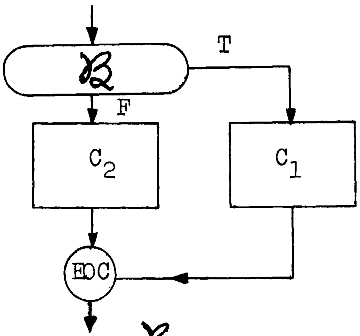
EXAMPLE:

```
WHENEVER I .G. 75
J = J+1                (See example under 20.6)
END OF CONDITIONAL

WHENEVER A .L. B .AND. Q       .E. 0.
B = SIN.(X) + Z
A = ALPHA * Q/N
Z = 1.0
END OF CONDITIONAL
```

The next simplest form of the conditional group is:

WHENEVER $\mathcal{B}$
$\left.\begin{matrix}\ldots\\ \ldots\\ \ldots\end{matrix}\right\} C_1$
OTHERWISE
$\left.\begin{matrix}\ldots\\ \ldots\\ \ldots\end{matrix}\right\} C_2$
END OF CONDITIONAL



In this case the first group of statements labeled C_1 will be executed only if $\mathcal{B}$ is true, in which case the statements C_2 will be by-passed and computation will resume at the END OF CONDITIONAL statement. In all other cases, i.e., when $\mathcal{B}$ is false, the set of statements at C_2 will be executed.

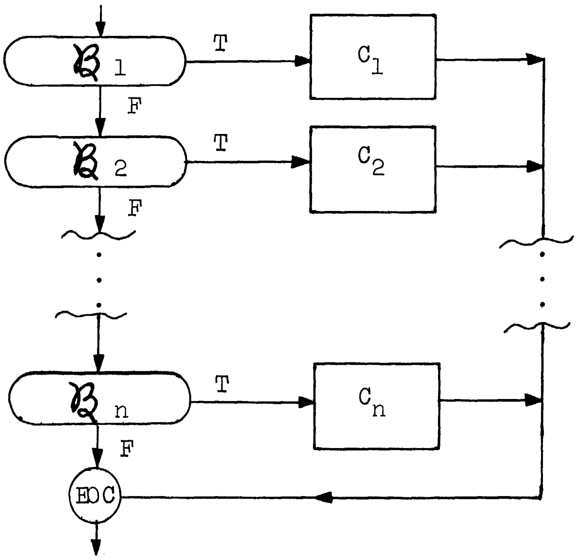
EXAMPLE:

```
WHENEVER A .L. B .AND. Q     .E. O.
   B = SIN.(X)+Z
   A = ALPHA*Q/N
   Z = 1.0
OTHERWISE
   B = COS.(X)+Z
   A = BETA*P/M
   Z = 0.0
END OF CONDITIONAL
```

(Blanks are ignored in MAD statements except when they appear between dollar signs (see 20.5). It is sometimes convenient to indent some statements for clarity.)

A third form of the conditional statement group is:

WHENEVER $\mathcal{B}_1$
$\left.\begin{matrix}\ldots\\ \ldots\\ \ldots\ldots\end{matrix}\right\} C_1$
OR WHENEVER $\mathcal{B}_2$
$\left.\begin{matrix}\ldots\\ \ldots\\ \ldots\end{matrix}\right\} C_2$
.
.
OR WHENEVER $\mathcal{B}_n$
$\left.\begin{matrix}\ldots\\ \ldots\\ \ldots\end{matrix}\right\} C_n$
END OF CONDITIONAL



In this case the successive Boolean expressions are evaluated for truth, starting in order, $\mathcal{B}_1, \ldots, \mathcal{B}_n$. The first $\mathcal{B}_i$ which is true will cause the execution of the statement group C_i immediately following it. No subsequent tests will be made and no other statement group, C_j, $j > i$ will be executed even if the associated $\mathcal{B}_j$ are true. If none of the $\mathcal{B}_i$ are true, then none of the statements C_i will be executed.

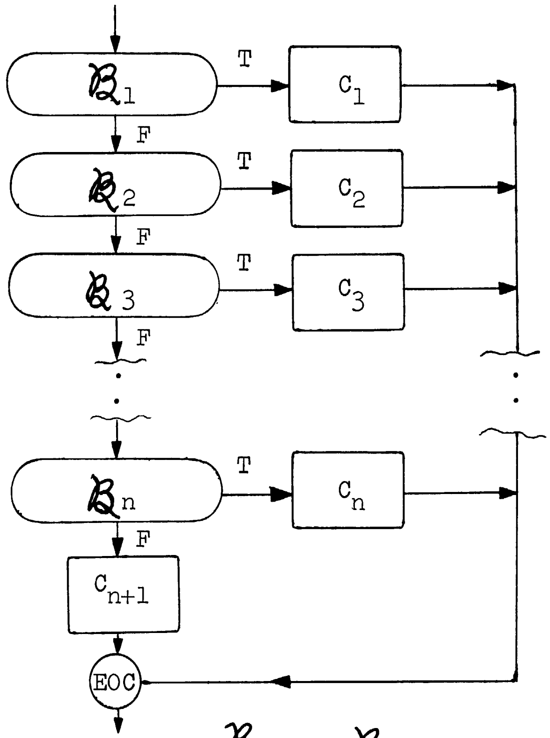
Compound Conditional Statement Group

EXAMPLE:

```
WHENEVER A .L. B
      ALPHA = Q+EPS*15.7
      BETA = 12.3*J
      Z = Z+1
OR WHENEVER C .GE. SIN.(ALPHA)
      ALPHA = Q+EPS*2.3
      Z = -Z
OR WHENEVER A*B .L. 0.
      ALPHA = -ALPHA
      Z = Z-1.
      BETA = .ABS.(Z*P)
END OF CONDITIONAL
```

In its most general form the compound conditional statement may be written:

```
WHENEVER $\mathcal{B}_1$
  ...  }
  ...  }  $C_1$
  ...  }
OR WHENEVER $\mathcal{B}_2$
  ...  }
  ...  }  $C_2$
  ...  }
OR WHENEVER $\mathcal{B}_3$
  ...  }
  ...  }  $C_3$
  ...  }
   .
   .
OR WHENEVER $\mathcal{B}_n$
  ...  }
  ...  }  $C_n$
  ...  }
OTHERWISE
  ...  }
  ...  }  $C_{n+1}$
  ...  }
END OF CONDITIONAL
```



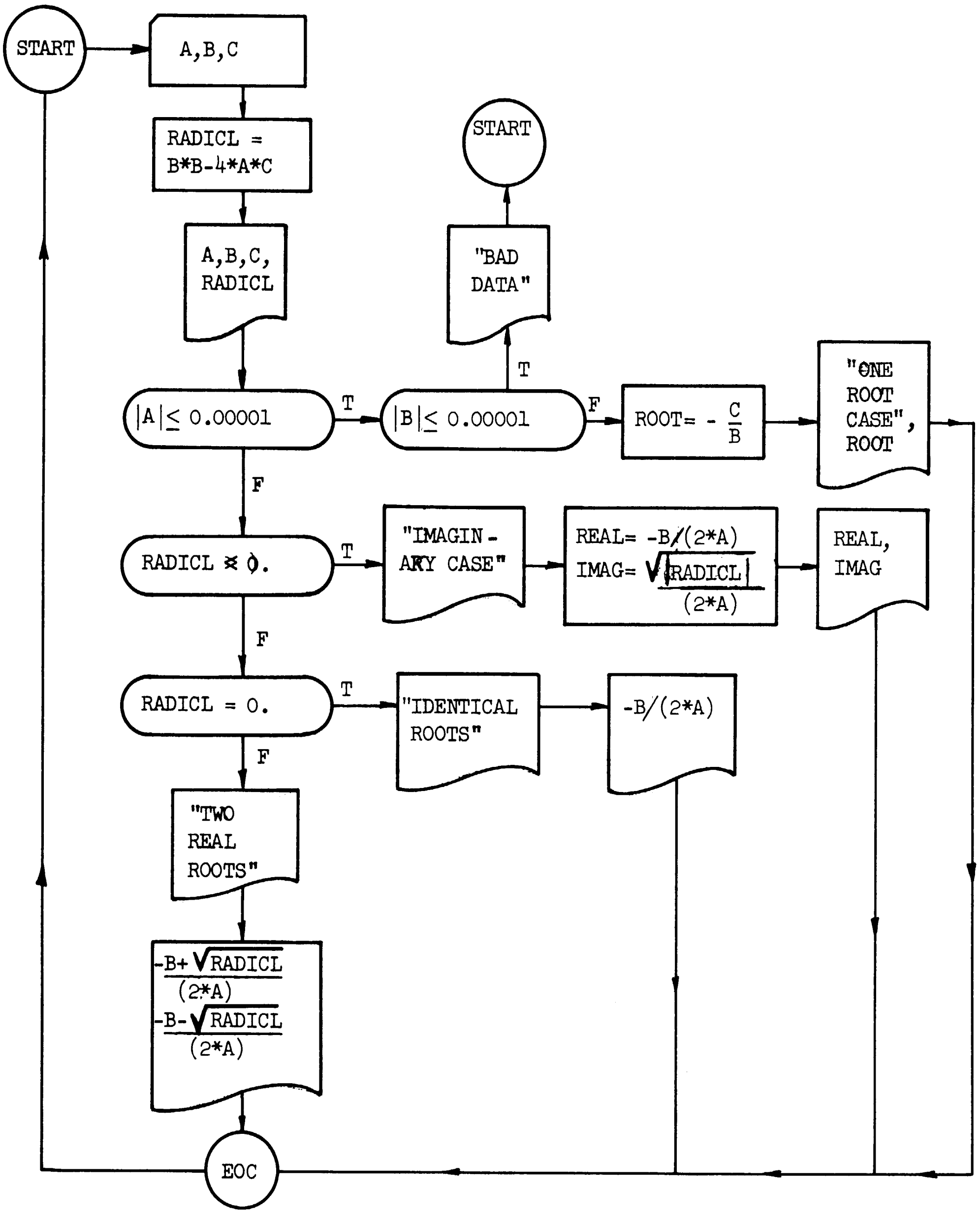
In this case the statements C_{n+1} will be executed only when the expressions $\mathcal{B}_1, \ldots, \mathcal{B}_n$ are all false.

Example:

The following is a complete MAD program to find the roots of the quadratic equation,

$$Ax^2 + Bx + C = 0.$$

Flow Chart.



A Complete MAD Program

MAD Program

```
START     READ DATA A, B, C
          RADICL = B*B - 4.0 *A*C
          PRINT RESULTS A, B, C, RADICL
          WHENEVER .ABS. A .LE. 1.E-5
               WHENEVER .ABS. B .LE. 1.E-5
                    PRINT COMMENT $ BAD DATA$
                    TRANSFER TO START
               END OF CONDITIONAL
               ROOT = -C/B
               PRINT COMMENT  $ ONE REAL ROOT$
               PRINT RESULTS ROOT
          OR WHENEVER RADICL .L. 0.
               PRINT COMMENT $ IMAGINARY CASE$
               REAL = -B/(2.*A)
               IMAG = SQRT.(.ABS.RADICL)/(2.*A)
               PRINT RESULTS REAL, IMAG
          OR WHENEVER RADICL .E. 0.
               PRINT COMMENT $ IDENTICAL REAL ROOTS$
               PRINT RESULTS -B/(2.*A)
          OTHERWISE
               PRINT COMMENT  $ TWO REAL ROOTS$
               PRINT RESULTS ( -B+SQRT.(RADICL))/(2.*A),
                             ( -B-SQRT.(RADICL))/(2.*A)
          END OF CONDITIONAL
          TRANSFER TO START
          END OF PROGRAM
```

It should be noted that for every WHENEVER $\mathcal{B}$ not followed by a comma, there must be a matching END OF CONDITIONAL statement to serve as a terminal bracket for the conditional group. Any number of statements of the type

OR WHENEVER $\mathcal{B}_i$

with corresponding groups of statements, C_i, may be inserted between the WHENEVER and END OF CONDITIONAL statements. There may or may not be a statement segment preceded by the OTHERWISE statement. If present, the OTHERWISE segment must be the last segment preceding the END OF CONDITIONAL statement, i.e., there may not be another OR WHENEVER statement following the OTHERWISE statement in any one compound conditional statement group.

It should also be noted that any one of the statement segments C_i may itself consist of one or more additional compound conditional statement groups. This nesting of conditionals can be continued to any desired depth.

20.8 The Iteration Statement:

Nearly every algorithm has a segment of consecutive statements which are to be processed repeatedly for a finite number of times. This statement sequence is termed a loop. There is normally some variable called the iteration variable or parameter which assumes a different value on each pass through the loop (each time the statement sequence is processed or executed). There are two forms of the iteration statement in the MAD language. The first, called the "for values" iteration statement is used when the iteration parameter is to assume a fixed number of different values (i.e., when the loop is to be executed a fixed number of times), and when these values are not related in some simple way (i.e., ordered and separated by a simply expressed interval).

a. The "For Values" Iteration Statement:

This statement is of general form

$$\text{THROUGH } \mathscr{S}\text{, FOR VALUES OF } \mathscr{V} = \mathscr{E}_1, \mathscr{E}_2, \mathscr{E}_3, \ldots, \mathscr{E}_n$$

When this statement appears in a MAD program, all the statements immediately following it, through and including the statement with the label $\mathscr{S}$ are repeatedly processed. On the first pass through the loop, the iteration variable $\mathscr{V}$ will have the value $\mathscr{E}_1$, on the second pass, the value $\mathscr{E}_2$, on the third pass, the value $\mathscr{E}_3$, etc. Here the variable $\mathscr{V}$ may be simple or subscripted and of any mode. Arithmetic expressions may be of different mode from $\mathscr{V}$ but in most cases this is neither necessary nor desirable.

EXAMPLE:

The following loop will compute the sum

$$a_{19} + a_7 + a_1 + a_k + a_{31} + a_{(p\cdot j)}$$

```
        SUM = 0.
        THROUGH NEXT, FOR VALUES OF I = 19, 7, 1, K, 31,  P*J
NEXT    SUM = A(I) + SUM
```

The following MAD statements evaluate and print the value of the polynomial

$$x^3 + 7.5x^2 + 2.0x + 26.4$$

for values of $x = 7.1, \alpha, \beta\cdot\gamma+q, \sin(z)$

```
        THROUGH LOOP,  FOR VALUES OF X=7.1, ALPHA, BETA*GAMMA+Q, SIN.(Z)
        Y=X.P.3 + 7.5 * X .P. 2 + 2.0 * X + 26.4
LOOP    PRINT RESULTS X, Y
```

Note that the number of necessary multiplication operations (including repeated multiplication to compute the square and cube) could have been reduced by writing the middle statement in so-called "nested" form,

```
Y = ((X+7.5)*X+2.0)*X+26.4
```

b. The "Incremental" Iteration Statement:

This statement, which is used much more frequently than the "for values" statement described above, is of general form

$$\text{THROUGH } \mathscr{S}\text{, FOR } \mathscr{V} = \mathscr{E}_1, \mathscr{E}_2, \mathscr{B}$$

where, in this case, as before, the loop includes all statements following the THROUGH statement *up to and including* the statement labeled $\mathcal{S}$. The iteration parameter $\mathcal{V}$ assumes different values on subsequent passes through the loop. When this THROUGH statement is executed for the first time, the variable $\mathcal{V}$ is set to the value given by the expression $\mathcal{E}_1$. *Before* the loop is executed, even for the first time, the Boolean expression $\mathcal{B}$ is evaluated. If $\mathcal{B}$ is true, then the loop is *not* executed at all and computation proceeds starting with the statement *following* the one labeled $\mathcal{S}$. If $\mathcal{B}$ is *false*, the loop is executed for the first time with $\mathcal{V}$ equal to $\mathcal{E}_1$. The variable $\mathcal{V}$ is then set equal to its initial value $\mathcal{E}_1$ plus the increment given by $\mathcal{E}_2$. (Note that $\mathcal{E}_2$ may be negative, in which case $\mathcal{V}$ is *decremented* rather than incremented.) $\mathcal{B}$ is again tested for its truth value. If *true*, then computation proceeds starting with the statement following the one labeled $\mathcal{S}$. If *false*, the loop is processed for the second time using the value $\mathcal{V} = \mathcal{E}_1 + \mathcal{E}_2$.

In general, after execution of the loop, $\mathcal{V}$ is incremented by the value $\mathcal{E}_2$ (i.e., $\mathcal{V} = \mathcal{V} + \mathcal{E}_2$). Before execution of the loop with the new value, the Boolean expression is checked. If false, the loop is executed. If true, the execution of the loop terminates and computation proceeds starting with the statement following the one labeled $\mathcal{S}$. The variable $\mathcal{V}$ *after* execution of the loop, will have the value it had at the time the Boolean expression became true. *Any* sequence of statements, including input-output, conditional, etc., can appear inside the loop. When a loop is terminated by the execution of a transfer statement, the iteration variable $\mathcal{V}$ retains its current value. A transfer into a loop is permitted but not recommended; special care must be taken to insure that the iteration variable has the proper value.

The iteration statement has five pertinent pieces of information:

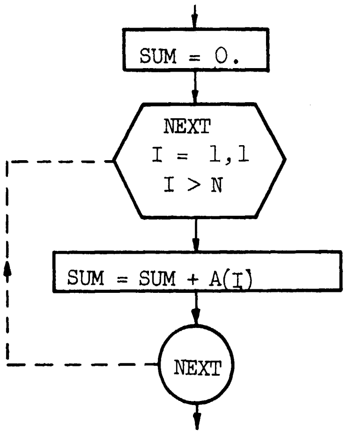
1. A statement label $\mathcal{S}$, to delimit the *scope* or extent of the loop, i.e., the label on the last statement in the loop.
2. An *iteration variable* (parameter) $\mathcal{V}$ which assumes a new value before each pass through the loop.
3. An expression for the initial value of the iteration variable, $\mathcal{E}_1$.
4. An expression for the *increment* of the iteration variable, $\mathcal{E}_2$.
5. A Boolean expression $\mathcal{B}$ to *test for termination* of the looping operation.

EXAMPLE:

The following MAD statement sequence will compute the sum

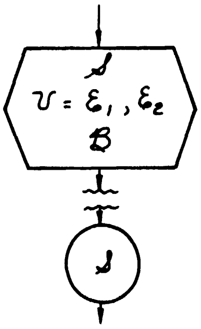
$$\sum_{1}^{n} a_i = a_1 + a_2 + a_3 +,\ldots,+ a_n$$

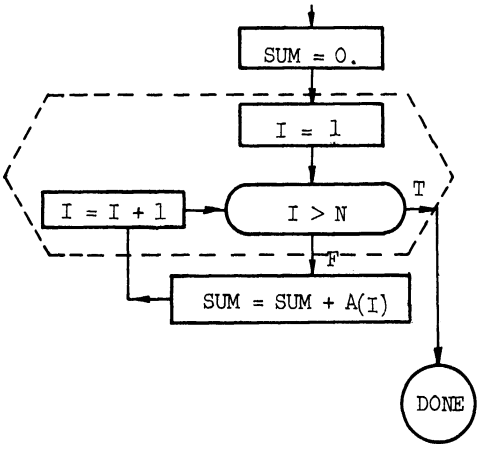
```
                SUM ≐ 0.0
                THROUGH NEXT, FOR I = 1,1,I.G.N
NEXT            SUM = SUM + A(I)
```



Note that without the iteration statement these same operations could also be accomplished with the following MAD statement sequence:

```
                SUM = 0.0
                I = 1
BACK            WHENEVER I .G. N, TRANSFER TO DONE
                SUM = SUM + A(I)
                I = I+1
                TRANSFER TO BACK
DONE            . . .
```





The flow diagramming convention used here is a hexagon containing the five pieces of information from the THROUGH statement and a circle containing the label of the final statement in the loop (in the diagram the label follows the labeled MAD statement). The hexagon is a simplified notation for the initialization, testing and incrementing boxes shown inside the dotted hexagon of the second flow diagram. As before, the expressions $\mathcal{E}_1$ and $\mathcal{E}_2$ may be of any mode and of any complexity. The variable $\mathcal{V}$ may be simple or subscripted and the Boolean expression may or may not be related to the iteration variable $\mathcal{V}$. For example, consider the following loop which computes the sum of even subscripted elements of an array A until the sum exceeds the value 75.4.

```
                SUM = 0.
                THROUGH TOTAL, FOR I=2,2,SUM.G.75.4
TOTAL           SUM = SUM + A(I)
```

After execution of the loop, the variable I will be greater by two than it was on the last pass through the loop, i. e., the last element added into the sum was A(I-2).

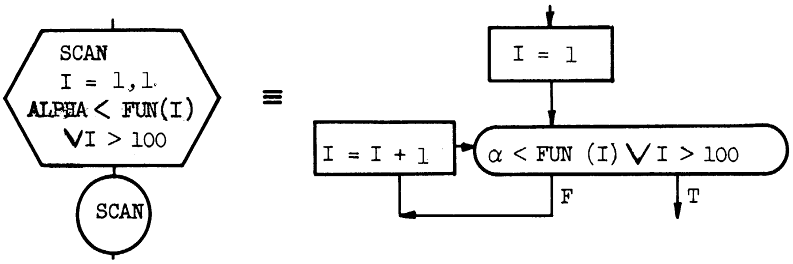
The iteration statements can be nested (a loop inside a loop) to any reasonable depth. For example, suppose it is desired to compute the product of all the elements above the main diagonal of the N x N square matrix or two dimensional array. The pertinent program segment could be as follows:

```
              PROD = 1.0
              THROUGH COMPUT, FOR I = 1, 1, I .E. N
              THROUGH COMPUT, FOR J = I+1, 1, J .G. N
COMPUT        PROD = PROD * A(I, J)
```

In this case the initial value of PROD must be set to 1.0 rather than 0.0 as was done with SUM because operations inside the loop involve multiplication rather than addition, in this example.

The incremental iteration statement can be used very effectively for searching through arrays of numbers, sometimes called "table look-up". The statement labeled $\mathcal{S}$ may then be the THROUGH statement itself. Suppose we have a table of functional values in ascending sequence stored in FUN(1)...FUN(100). Then we could find which two elements in the set are closest to the value of ALPHA with the statement.

```
SCAN        THROUGH SCAN, FOR I = 1, 1,ALPHA.L.FUN(I) .OR.I.G.100
```



In this case loop processing continues until ALPHA < FUN(I) in which case FUN(I-1) $\leqslant$ ALPHA < FUN(I). Should ALPHA be larger than any element of the table, processing would discontinue when I > 100, i.e., when I = 101. A test for the value of I after the loop would establish which case actually caused termination of the search.

The two iteration statements described here, and particularly the second one, are the most powerful statements in the language. The second statement is especially useful for doing repetitive operations on arrays, i.e., subscripted variables, where the subscripts are being modified in a regular way. In cases where the iteration parameter is being used as a subscript inside the loop or simply as a counting variable, it should normally be declared to be of INTEGER mode.

The final statement in a loop, i.e., the one labeled $\mathcal{S}$, may be a one-line WHENEVER statement, an END OF CONDITIONAL statement terminating a compound conditional group, or a CONTINUE statement to be described in the next paragraph, as well as most of the other executable statements. A compound conditional group in a loop must be completely contained inside one loop. Any nested iteration loop must as well be completely contained inside its nesting loop. Two loops may terminate with the same labeled statement, however.

20.9 The CONTINUE Statement:

The CONTINUE statement is composed of the single word

CONTINUE

It causes no computation to be done but is executable and may have a statement label attached to it. It can serve as the terminal junction in a TRANSFER TO $\mathcal{S}$ statement. It is occasionally useful, although not necessarily needed, as the terminal statement of an iteration loop. A card with a statement label but otherwise blank is also interpreted as a CONTINUE statement.

20.10 The END OF PROGRAM Statement:

The last statement in every MAD program must be

END OF PROGRAM

This is an executable statement which may have a statement label. One method of terminating a program is to transfer to a labeled END OF PROGRAM statement or simply to "run off the end of the program" by encountering the END OF PROGRAM statement in the normal processing sequence.

21. Non-Executable Statements (Declarations):

A declaration in the MAD language is a statement which causes no machine code to be generated by the translator. Its primary purpose is to provide information to the translator (during the translation process) about variable modes, array sizes, etc. Since declarations are for translator information only, they may be inserted anywhere in the MAD program before the END OF PROGRAM statement. Declarations should not have statement labels.

21.1 The Mode Declaration Statement

Normally, unless specifically listed in a mode declaration statement in the MAD program, a variable is assumed to be in floating point mode. If it is instead of integer, Boolean, or statement label mode, then it is necessary to include a declaration of the form

INTEGER $\mathcal{L}$
BOOLEAN $\mathcal{L}$
or STATEMENT LABEL $\mathcal{L}$

Here $\mathcal{L}$ is a list of variable names without subscripts which are to be assigned integer or Boolean mode respectively. Occasionally a function reference $\mathcal{F}.(\mathcal{E})$ may also have an integer or Boolean value in which case the function name $\mathcal{F}$. without attached arguments must appear in the list as well (see SUB. below).

EXAMPLES:

INTEGER I, J, N, BETA, SUB., Q

BOOLEAN SWITCH, TEST

21.2 The NORMAL MODE Statement:

Occasionally, one writes a program in which most of the variables are not of floating point mode. It is then necessary either to list all these variables in an INTEGER or BOOLEAN statement (see 21.1) or, if desired, to alter the normal assumed mode from floating point to integer or Boolean. In this case the insertion of a statement of form

NORMAL MODE IS $\mathcal{M}$

where $\mathcal{M}$ is either INTEGER or BOOLEAN will cause the translator to assign the normal mode $\mathcal{M}$ to any variables not mentioned in a mode declaration statement.

EXAMPLES:

NORMAL MODE IS INTEGER

NORMAL MODE IS BOOLEAN

Floating point variables must then be declared using a mode declaration statement of the form

FLOATING POINT

EXAMPLE:

FLOATING POINT ALPHA, X, DERIV, SIN., Y

21.3 The DIMENSION Statement:

a. Linear Arrays (single Subscripted Variables):

Since the translating program must assign memory space to subscripted variables, i.e., array variables, it is necessary to include information about the maximum size of any subscript on the variable in the program. This is done through the use of the DIMENSION statement which is of general form

DIMENSION $\mathcal{V}_1(k_1), \mathcal{V}_2(k_2), \ldots, \mathcal{V}_n(k_n)$

where the $\mathcal{V}_i$ in this case are variable names. If the arrays are linear arrays, i.e., if there is only one subscript attached to the variable name in the program, then $k_1, \ldots, k_n$ are the integer constants which are the maximum values assumed by subscripts in the program. Subscripts in the program may then assume values j_i where $0 \leqslant j_i \leqslant k_i$, $i=1, 2, \ldots, n$.

EXAMPLE:

If A is a subscripted variable, $A_0, \ldots, A_{100}$, and B is another subscripted variable, $B_0, \ldots, B_{500}$, then the appropriate DIMENSION statement is

DIMENSION A(100), B(500)

In this case the translator will save 101 memory locations for the A array and 501 for the B array (note that one location is saved for the zero subscripted element). Frequently a linear array is called a vector.

b. Arrays with More than One Subscript:

If the array has dimension 2 or greater, i.e., if there are two or more subscripts on the variable, then it is necessary to specify in addition to the maximum linear size of the array, some information about the arrangement of the array. The general form is

DIMENSION $\mathcal{V}_1(k_1, \overline{\mathcal{V}}_1(j_1)), \mathcal{V}_2(k_2, \overline{\mathcal{V}}_2(j_2)), \ldots, \mathcal{V}_n(k_n, \overline{\mathcal{V}}_n(j_n))$.

In this case the $\mathcal{V}_i$ are variable names which bear multiple subscripts in the program. k_i is the maximum subscript which would be computed if the array $\mathcal{V}_i$ is viewed as linear, i.e., one less than the total number of memory locations to be saved for the array $\mathcal{V}_i$. $\overline{\mathcal{V}}_i(j_i)$ is a subscripted variable whose value is the first piece of required additional dimensioning information about $\mathcal{V}_i$. It and the few locations which follow it

$\overline{\mathcal{V}}_i(j_i+1), \overline{\mathcal{V}}_i(j_i+2), \ldots$, etc.

are called the <u>dimension</u> <u>vector</u> and contain the following information.

Element	Contents
$\overline{\mathcal{V}}_i(j_i)$	m, the number of dimensions(i.e., 2 for two subscripts, 3 for three subscripts, etc.)
$\overline{\mathcal{V}}_i(j_i+1)$	the linear subscript on the array $\mathcal{V}_i$ to which the base element (the element which has multiple subscripts all equal to 1, e.g., A(1,1), Q(1,1,...,1)) is assigned.
$\overline{\mathcal{V}}_i(j_i+2)$	the maximum span of the second subscript (for two dimensions, this is the number of columns).
$\overline{\mathcal{V}}_i(j_i+3)$	the maximum span of the third subscript (if any).
.	.
.	.
.	.
$\overline{\mathcal{V}}_i(j_i+m)$	the maximum span of the mth subscript.

The dimension vector variable name $\overline{\mathcal{V}}_i$ must be of integer mode. It s use in the DIMENSION statement, however, automatically assigns integer mode to $\overline{\mathcal{V}}_i$ and its inclusion in an INTEGER statement is unnecessary.

This may seem a tedious way to inform the translator about the arrangement of multi-subscripted arrays. However, the use of the dimension vector, i.e., a linear array which contains information about the dimensioning of a multiply subscripted array, allows for great flexibility in the organization of memory during execution, making possible a change in the number of subscripts, the maximum span of various subscripts, etc.., during computation, so long as any modifications do not cause the subscripted element to be outside the storage region assigned the variable. A detailed description of the MAD dimensioning conventions may be found in Reference 4.

In general, for multi-subscripted variables, where no negative or zero subscript values are being used,the elements of the dimension vector can be preset by using the VECTOR VALUES statement (see 21.4). The pair of statements,

```
DIMENSION ALPHA (200, DALPH)
VECTOR VALUES DALPH = 2, 0, q
```

where q is an <u>integer</u> <u>constant</u> (<u>not</u> <u>a</u> <u>variable</u> <u>name</u>) equal to the number of columns in the ALPHA array, will suffice for a doubly-subscripted array. In this case the dimensioning information for ALPHA starts in DALPH = DALPH(0)

Element	Contents	Remarks
DALPH = DALPH(0)	2	Two subscripts
DALPH(1)	0	Base subscript, i.e., ALPHA(1,1) = ALPHA(0)
DALPH(2)	q	q is the number of columns (must be an <u>integer</u> <u>constant</u>).

21.4 Vector Initialization:

Frequently it is useful to preset or initialize values of subscripted variables before execution of the program starts. In this case the MAD statement is of general form

$$\text{VECTOR VALUES } \mathcal{V}(k) = \mathcal{L}$$

where $\mathcal{V}$ is a variable name, k is a single integer constant, and $\mathcal{L}$ consists of a string of constants, all of the same mode (normally). This causes the element $\mathcal{V}$(k) to be preset to the value of the first element of the list, the value of $\mathcal{V}$(k+1) to be set to the second element of the list, etc. It is not necessary to declare the mode of $\mathcal{V}$ in a mode declaration statement; $\mathcal{V}$ is arbitrarily assigned the mode of the first element in the list.

The VECTOR VALUES statement also automatically dimensions the specific variable to have the largest computed subscript (in the following example, 7) unless the variable $\mathcal{V}$ appears with greater maximum subscript in a DIMENSION statement.

EXAMPLE:

Suppose one wanted to set the elements of BETA(1)...BETA(7) equal to 1.0, -7.5, +3.2, +9.6, -6.2, +3.1, +6.4; then the MAD statement would be:

VECTOR VALUES BETA(1) = 1.0, -7.5, 3.2, 9.6, -6.2, 3.1, 6.4

A modification of this statement which allows the presetting of several sequential elements of an array is as follows:

$$\text{VECTOR VALUES } \mathcal{V}(k), \ldots, \mathcal{V}(\ell) = \eta$$

Here all elements of the array $\mathcal{V}$ between and including the elements with subscripts k and ℓ are preset with the same constant η. k and ℓ must be integer constants.

EXAMPLE:

VECTOR VALUES BETA(1),...,BETA(10) = 1.0

22. External Function Definition Forms:

Function references $\mathcal{F}$. ( ) have already been discussed in paragraph 5, In addition to the six subroutines* or external functions* SIN., COS., SQRT., ELOG., EXP. and ATAN., mentioned previously, many other subroutines are available to evaluate a variety of functions (e.g., Bessel functions, the error function, etc.) and perform routine numerical and statistical calculations. Each of these programs has a name different from the names of all other external functions (a program without a name is called a main program). Normally, a list of arguments must be included to communicate information between the calling program and the called subroutine. This list, also known as the calling sequence, is the only information link between the two programs. The calling program is usually, but not necessarily, a main program, i.e., subroutines may also call on other subroutines.**

Each of the functions cited in (5) has just one argument in its calling sequence. For example, in a MAD program, a square root subroutine is supplied automatically, simply by mentioning the name SQRT.,

* These terms are used interchangeably throughout.

** Subroutines which call on themselves are called recursive functions. The MAD language has facility for defining such functions, but they will not be discussed here.

and enclosing a single non-negative floating point argument $\mathcal{E}$ in parentheses, e.g.,

SQRT. (42.7)

SQRT. (X)

SQRT. (B*B - 4.*A*C)

The floating point value of the function reference SQRT. ($\mathcal{E}$) is computed at the proper place and serves as one operand for an arithmetic operator (see 6) e.g.,

X = SQRT. (B*B - 4.*A*C)

Z = (-B - SQRT.(B*B - 4.*A*C))/(2.*A)

Note that a statement consisting only of the reference SQRT. (B*B - 4.*A*C) is meaningless, since there is no indication of what to do with the value of the function.

Many subroutines have more than one argument, since just one may not supply enough information to permit the desired computation. For example, an arctangent subroutine called ATN1., computes the angle ϕ between the positive x axis and the point whose y and x coordinates are $\mathcal{E}_1$ and $\mathcal{E}_2$ respectively such that

$$0 \leqslant \text{ATN1.}(\mathcal{E}_1, \mathcal{E}_2) = \phi < 2\pi \ .$$

Another arctangent subroutine, ATAN., (see (5)) has just one argument $\mathcal{E}$ and returns the principal value of $\tan^{-1}(\mathcal{E})$, i.e.

$$-\frac{\pi}{2} \leqslant \text{ATAN.}(\mathcal{E}) \leqslant \frac{\pi}{2} \ .$$

The additional information transmitted with the two arguments (namely the signs of $\mathcal{E}_1$ and $\mathcal{E}_2$) permits the routine ATN1. to find the angle in any of the four quadrants, while ATAN. with one argument cannot distinguish between angles in quadrants 1 and 3 or 2 and 4.

Often, a subroutine will not be available to solve a particular problem. In this case, the programmer must define his own subroutine. All the MAD statements described in the preceding paragraphs, with the exception of END OF PROGRAM (see 20.10), may be used. The four additional statements required to define a function are shown in paragraphs 22.1 - 22.3.

22.1 Function definition delimiters:

The two statements EXTERNAL FUNCTION ($\mathcal{L}$) and END OF FUNCTION are used to serve as brackets for the function definition. These statements are declarations in the sense that they are not executable and may not have statement labels. They must be physically the first and last statements in the definition:

EXTERNAL FUNCTION ($\mathcal{L}$)

$$\left.\begin{array}{l}\rule{2cm}{0.4pt}\\ \rule{2cm}{0.4pt}\\ \rule{2cm}{0.4pt}\end{array}\right\}\ \text{function definition statements}$$

END OF FUNCTION

The list $\mathcal{L}$ is the argument list or calling sequence for the function. The elements of $\mathcal{L}$, separated by commas, are artificial or dummy names, i.e., they are not actual variables; consequently, no storage space is assigned to them. Their names have local meaning only, i.e., have meaning only in the set of statements constituting the definition. Values for the argument variables are supplied by the calling program. The modes of all variables in $\mathcal{L}$ which are not of normal mode (see 21.2) must be declared in the usual way.

The list $\mathcal{L}$ may contain dummy statement label variables $\mathcal{S}_i$, in which case the $\mathcal{S}_i$ must appear in a statement label mode declaration:

STATEMENT LABEL $\mathcal{S}_1, \mathcal{S}_2, \ldots, \mathcal{S}_m$

In addition, $\mathcal{L}$ may contain dummy function names $\mathcal{F}._i$ (the period must be included and no mode declaration is required). Dummy arguments which are subscripted variables must appear <u>without</u> subscripts in the argument list.

22.2 <u>Entry Points:</u>

The starting or entry point to the program, usually, but not necessarily, the first statement inside the definition, is

ENTRY TO $\mathcal{F}$.

where $\mathcal{F}$ is the name chosen for the function. The naming conventions for $\mathcal{F}$ are the same as for variables (see 2); the period <u>must</u> be present to establish that $\mathcal{F}$ is a function name.

A subroutine may have more than one name (and hence entry point). In such event the only requirement is that the argument list $\mathcal{L}$ be applicable to all function names used, since it appears only once (see 22.1).

22.3 <u>Exit Points:</u>

After the statements which perform the necessary calculations, the statement

FUNCTION RETURN $\mathcal{E}$

or

FUNCTION RETURN $\mathcal{B}$

causes a return to the calling program. The <u>value</u> and <u>mode</u> of the function reference $\mathcal{F}.(\mathcal{L})$ is given by the <u>value</u> and <u>mode</u> of $\mathcal{E}$ or $\mathcal{B}$. There <u>must be at least one exit</u> from a function; there may be more than one.

In some cases functions do not have values (see 22.5). In these cases the exit statement is just

FUNCTION RETURN

22.4 <u>External Functions which Return a Value:</u>

When an external function returns a value (see 22.3), the function reference in the calling program must appear as part of an expression of appropriate mode (normally integer, floating point, or Boolean). Otherwise the value returned cannot be utilized (see 22).

Now consider the definition of a simple program which finds either the maximum or minimum of a set of N elements of an arbitrary linear array A (in A(1)... A(N)). N and A are dummy arguments and the argument list $\mathcal{L}$ is (N, A). Let the function have two names, MIN. and MAX., which return respectively the minimum and maximum value in the array A as the value of the function references MIN. $(\mathcal{L})$ and MAX. $(\mathcal{L})$. Assume that N is of integer mode and that A, MIN. $(\mathcal{L})$ and MAX. $(\mathcal{L})$ are of floating point mode. The program might appear as follows:

```
             EXTERNAL FUNCTION (N, A)
             INTEGER N, I

             ENTRY TO MIN.
                ANSWER = A(1)
                THROUGH NEXT, FOR I = 2, 1, I. G. N
NEXT            WHENEVER A(I) . L. ANSWER, ANSWER = A(I)
             FUNCTION RETURN ANSWER

             ENTRY TO MAX.
                ANSWER = A(1)
                THROUGH NEXT1, FOR I=2, 1, I. G. N
NEXT1           WHENEVER A(I) . G. ANSWER, ANSWER = A(I)
             FUNCTION RETURN ANSWER
             END OF FUNCTION
```

This function is compiled as a separate program or "external" function. Some important features are: (1) the modes of all variables not of normal mode, including arguments, must be declared, (2) subscripted variables in argument lists (e. g., A) must not be dimensioned in the subroutine, (3) there may be more than one entry and exit point in the same external function, (4) the mode of the functions MIN. and MAX. is determined by the mode of the value returned, ANSWER (since the mode of ANSWER is not declared, its mode is floating point); hence, the modes of the functions are also floating point.

Although not apparent from this example, variables which are not arguments are actual variables and have meaning only inside the scope of the definition (I in this program and I in some other program are completely unrelated variables). Subscripted variables which are not arguments must be dimensioned. The reason for this is straightforward. Variables which are not arguments are actual variables and memory space must be assigned them by the translator. On the other hand, variables which are arguments are just dummy names for the true arguments in the calling program; hence no space is needed in the subroutine. Of course, the true arguments must be dimensioned in the calling program.

The functions MIN. or MAX. would be called implicitly, e. g.,

```
Y = 5. *MAX. (M, Z)
```

would cause 5 times the largest of the values Z(1)... Z(M) to be stored in Y.

```
SMALL = MIN. (P, BETA)
```

would cause the smallest of BETA(1)... BETA(P) to be stored in SMALL. In these two cases M and P would have to be declared of integer mode in the calling program. There must be agreement in number, order, and mode between the list of actual variables in the calling program and the list of dummy variables in the function definition.

22. 5 External Functions Which do not Return a Value:

Often a function is required to perform some non-valued operation (such as the ordering of all elements in an array) or to carry out some procedure which has more than one result. An implicit call (see 22. 4) is inadequate since the function reference must have one and only one value.

In such cases, the actual variables corresponding to the dummy variables in the argument list serve not only to supply information to the routine but also as locations for storing the answers. For example, by adding two arguments, LITTLE and BIG, to the dummy argument list, the preceding program could be modified to find both the largest and smallest of the elements A(1)... A(N) with one call. Let the new routine be called MINMAX.

```
                EXTERNAL FUNCTION (N, A, LITTLE, BIG)
                INTEGER N, I
                ENTRY TO MINMAX.
                LITTLE = A(1)
                BIG = A(1)
                THROUGH FIND, FOR I=2, 1, I. G. N
                WHENEVER A(I) . L. LITTLE
                   LITTLE = A(I)
                OR WHENEVER A(I) . G. BIG
                   BIG = A(I)
       FIND     END OF CONDITIONAL
                FUNCTION RETURN
                END OF FUNCTION
```

In this case, two of the dummy variables (LITTLE, BIG) appear on the left of the substitution operator and, hence, are assigned values by the function. Since no expression value appears after FUNCTION RETURN, the function reference MINMAX. has no value; it must be called explicitly by the MAD statement

```
                EXECUTE MINMAX. (ℒ)
```

or simply by

```
                MINMAX. (ℒ)
```

For example,

```
                EXECUTE MINMAX. (M, Z, SMALL, LARGE)
```

or

```
                MINMAX. (M, Z, SMALL, LARGE)
```

would cause the subroutine to assign the smallest and largest of the values Z(1)... Z(M) to the actual variables SMALL and LARGE.

Note that an implicit call such as

```
                Y = MINMAX. (M, Z, SMALL, LARGE)
```

is meaningless, since there is no value returned to assign to Y.

22. 6 Functions Which Both Modify Arguments and Return a Value:

It is also possible to write programs which modify arguments and, in addition, return a value. For example, MINMAX., the function of (22. 5), could be written to calculate and return the mean of the numbers A(1)... A(N) as its value.

```
                EXTERNAL FUNCTION (N, A, LITTLE, BIG)
                INTEGER N, I
                ENTRY TO MINMAX.
                LITTLE = A(1)
                BIG = A(1)
                SUM = A(1)
                THROUGH FIND, FOR I=2, 1, I . G. N
                SUM = SUM + A(I)
                WHENEVER A(I) . L. LITTLE
                   LITTLE = A(I)
                OR WHENEVER A(I) .G. BIG
                   BIG = A(I)
       FIND     END OF CONDITIONAL
                FUNCTION RETURN SUM/N
                END OF FUNCTION
```

In this case the statement

```
Y = MINMAX.(M, Z, SMALL, LARGE)
```

is of acceptable form. The value of MINMAX.($\mathcal{A}$), viz., the mean of the numbers Z(1)... Z(M), is assigned to Y. In addition, SMALL and LARGE are assigned the proper values as before.

23. Internal Functions:

Internal functions are, in general, similar to external functions but have some important differences. An internal function is compiled as part of (internal to) another program (this may be a main program or external function, but not another internal function), rather than separately. As before, the dummy variables have meaning only inside the function definition. Actual (or global) variables in the function are defined to have the same meanings and values which they have elsewhere in the imbedding program (see 22.4 for a comparison with the values of actual variables in external functions).

23.1 Internal Function Definition Form:

The internal function definition form is identical with that for the external function except that the leading statement is

```
INTERNAL FUNCTION (𝒜)
```

The complete function definition may appear anywhere before the END OF PROGRAM statement of the program in which it is imbedded; mode declarations may appear anywhere in the imbedding program.

23.2 One-Line Internal Function

There is a convenient short form for the internal function which may be used if the function has just one name and one value, and can be defined in a single statement. The defining statement is

$$\text{INTERNAL FUNCTION}\mathcal{F}.(\mathcal{A}) = \mathcal{E}$$

or

$$\text{INTERNAL FUNCTION}\mathcal{F}.(\mathcal{A}) = \mathcal{B}$$

Here $\mathcal{F}$. is the name of the function, $\mathcal{A}$ the argument list, and $\mathcal{E}$ or $\mathcal{B}$ the function value. This value may be a function of the actual variables in the program as well as the arguments. For example, suppose a function of the form

$$f(x) = a + \tan(x)$$

is needed frequently. The normal internal function definition given by

```
INTERNAL FUNCTION (X)
ENTRY TO F.
FUNCTION RETURN A + SIN.(X)/COS.(X)
END OF FUNCTION
```

may be shortened to

```
INTERNAL FUNCTION F.(X) = A + SIN.(X)/COS.(X)
```

In both cases, A is an actual variable. A typical call might be

```
Z = F.(B*Y+2.)
```

which would produce calculations equivalent to those for the statement

```
Z = A + SIN.(B*Y+2.)/COS.(B*Y+2.)
```

SUMMARY OF THE BASIC SET OF MAD STATEMENT TYPES

The following is a basic set of MAD statement types. Here as before, $\mathcal{S}$ is any statement label, $\mathcal{V}$ any integer or floating point variable, $\mathcal{V}_{\mathcal{B}}$ any Boolean variable, $\mathcal{E}$ any arithmetic expression, $\mathcal{B}$ any Boolean expression, $\mathcal{Q}$ any of a restricted set of executable statements (see 20.6), $\mathcal{A}$ a string of characters not including \$, $\mathcal{M}$ a mode, and $\mathcal{L}$ a list of some form. For specific details about $\mathcal{L}$ see the description of the statement type in the appropriate paragraphs.

Executable Statement	General Form	
20.1 Substitution	$\mathcal{V} = \mathcal{E}$ or $\mathcal{V}_{\mathcal{B}} = \mathcal{B}$	
20.2 Transfer	TRANSFER TO $\mathcal{S}$	
20.3 Simplified Input	READ DATA READ AND PRINT DATA	READ DATA $\mathcal{L}$ READ AND PRINT DATA $\mathcal{L}$
20.4 Simplified Output	PRINT RESULTS $\mathcal{L}$	
20.5 Simplified Comment	PRINT COMMENT \$ $\mathcal{A}$ \$	
20.6 Simple Conditional	WHENEVER $\mathcal{B}$, $\mathcal{Q}$	
20.7 Compound Conditional Group	WHENEVER $\mathcal{B}_1$ OR WHENEVER $\mathcal{B}_2$ OTHERWISE END OF CONDITIONAL	
20.10 End (Main Program)	END OF PROGRAM	
20.8 Iteration:		
"For Values"	THROUGH $\mathcal{S}$, FOR VALUES OF $\mathcal{V} = \mathcal{E}_1, \mathcal{E}_2, \ldots, \mathcal{E}_n$	
"Incremental"	THROUGH $\mathcal{S}$, FOR $\mathcal{V} = \mathcal{E}_1, \mathcal{E}_2, \mathcal{B}$	
20.9 Label Bearer	CONTINUE	

Non-Executable Statement	
21.1 Mode Declaration	$\mathcal{M}\ \mathcal{L}$
21.2 Normal Mode Declaration	NORMAL MODE IS $\mathcal{M}$
21.3 Dimension Declaration	DIMENSION $\mathcal{L}$
21.4 Vector Initialization	VECTOR VALUES $\mathcal{V} = \mathcal{L}$ (or $\mathcal{V}_{\mathcal{B}} = \mathcal{L}$)

Function Definition Statements	
22.1 Opening Statement (external function)	EXTERNAL FUNCTION $(\mathcal{L})$
22.1 End (functions)	END OF FUNCTION
22.2 Function Entry	ENTRY TO $\mathcal{F}$.
22.3 Function Exit	FUNCTION RETURN $\mathcal{E}$
or	FUNCTION RETURN $\mathcal{B}$
or	FUNCTION RETURN
23.1 Opening Statement (internal function)	INTERNAL FUNCTION $(\mathcal{L})$
23.2 One-Line Function	INTERNAL FUNCTION $\mathcal{F}.(\mathcal{L}) = \mathcal{E}$
or	INTERNAL FUNCTION $\mathcal{F}.(\mathcal{L}) = \mathcal{B}$

A Synopsis of the MAD Language

DECK PREPARATION

Most large computers are routinely operated by a set of machine language programs called the "system", "executive system" or "monitor." These programs permit the automatic handling of large numbers of programs in sequence (called "batching") without manual intervention by the machine operator. The system oversees both the translation (compilation) of programs written in source languages and the running (execution) of the translated object or machine language programs.

In order to permit such automatic processing, a standard deck arrangement is required. What follows is a description of the deck arrangement expected by the Michigan Executive System used on the IBM 7090 at The University of Michigan. Other installations will also require some standard deck format, probably different from the one described here.

For main programs the deck consists of the following cards in order:

a. 2 Identification Cards (Yellow Cards).

b. 1 Specification Card (Blue Card).

c. The MAD Program Statement Cards (Pink Top Cards).

d. 1 Specification Card (Blue Card).

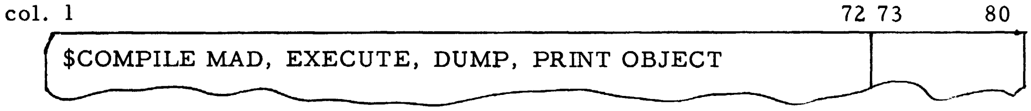
e. The Data Cards for the MAD Program (Pink Top Crads).

a. The two ID cards are identical and have the following format:

Columns	Contents
2-24	User's Name.
32-36	Assigned student project number.
52-54	Execution time estimate in minutes or seconds (This is the time estimate for the execution of the MAD program after it has been translated and put into the machine's memory. If three digits are punched in these columns, then the number is interpreted as minutes, e.g., 001 means one minute. If only two digits are punched (in columns 53-54) and an * is punched in column 52, then the number is interpreted as seconds, e.g., *10 means ten seconds. Decks which require any compilation are limited to a maximum of 2 minutes of execution time.)
58-60	Page estimate: (This is an estimate of the number of pages which will be produced by the machine when the translated program is executed and must not exceed 050 for decks requiring compilation.)
64-66	Card estimate: (This is an estimate of the number of cards which will be produced by the program during execution. The punch statements have not been discussed here, so these columns should contain 000.)

The information on the ID cards is used by the executive system to establish that the machine user has received permission to use the machine, and to determine when a program should be automatically stopped, (i.e., when it runs overtime or attempts to print more pages or punch more cards than originally estimated).

b. The first specification card should appear as follows:

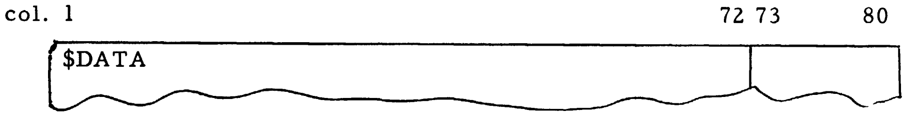


```
$COMPILE MAD, EXECUTE, DUMP, PRINT OBJECT
```

This card causes the executive system to bring the MAD translator into memory to translate the MAD program statements which follow. If the translation (compilation) is successful, then EXECUTE indicates that the translated machine language version of the program should be loaded into memory. A printed version of the machine language or "object" program will be produced if PRINT OBJECT is specified. DUMP causes a listing of memory values to be "dumped" or printed out in case something goes wrong. Although the object program listing and dump are not of much value to a beginner, both are useful debugging aids for experienced programmers.

c. The MAD program must be punched as shown in 19. The card immediately preceding the second specification card (d.) must contain the END OF PROGRAM statement for the program.

d. The second specification card of form



```
$DATA
```

specifies that the cards which follow are data cards for the MAD program.

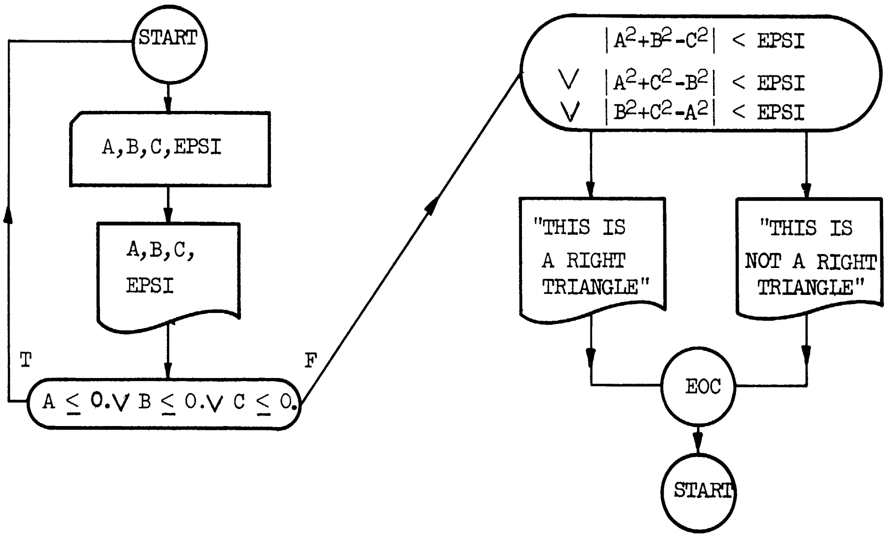
e. The data cards are punched as shown in 20.3 in the sequence they are to be read by the MAD program.

If external functions (subroutines) are present in addition to the main program, each complete function definition starting with the EXTERNAL FUNCTION (α) statement must be preceded by a specification card similar to that shown in b, except that the words EXECUTE and DUMP (which apply to the whole program package rather than to an individual main program or subroutine) need not appear. Each function must end with the END OF FUNCTION statement. There may be as many external functions as desired. The last one is followed by the $DATA specification card and the data cards as described above.

EXAMPLE (main program only):[4]

The following is a simple MAD program which reads from data cards three values for the floating point variables A, B, and C, the lengths for three sides of a triangle. The program then establishes if the sum of the squares of two sides of the triangle agrees (to within a tolerance EPSI) with the square of the third side. If so, the triangle ABC is a right triangle and an appropriate comment is printed; otherwise a comment indicating that ABC is not a right triangle is printed; any number of data sets can be processed. Execution of the program is automatically terminated when a READ DATA statement is encountered and all data cards have already been read.

Flow Diagram


START
A,B,C,EPSI
A,B,C,
EPSI
T
F
$A \leq 0. \vee B \leq 0. \vee C \leq 0.$
$|A^2+B^2-C^2| < EPSI$
$\vee\ |A^2+C^2-B^2| < EPSI$
$\vee\ |B^2+C^2-A^2| < EPSI$
"THIS IS A RIGHT TRIANGLE"
"THIS IS NOT A RIGHT TRIANGLE"
EOC
START

Complete Deck (U. M. System)

```
                     1         2         3         4         5         6         7
Column  →   123456789012345678901234567890123456789012345678901234567890123456789012

             LUTHER CARNAWILKES           D066N               001   001   000
             LUTHER CARNAWILKES           D066N               001   001   000
            $ COMPILE MAD, EXECUTE, DUMP, PRINT OBJECT

                      R   SOLUTION OF RIGHT TRIANGLES PROBLEM

            START      READ DATA A, B, C, EPSI
                       PRINT RESULTS A, B, C, EPSI
                       WHENEVER A .LE. 0. .OR. B .LE. 0..OR. C .LE. 0.,TRANSFER TO
                      1  START
                       WHENEVER .ABS. (A*A+B*B-C*C) .L. EPSI .OR. .ABS. (B*B+C*C-A*A
                      1) .L. EPSI .OR. .ABS. (C*C+A*A-B*B) .L. EPSI
                       PRINT COMMENT $ THIS IS A RIGHT TRIANGLE$
                       OTHERWISE
                       PRINT COMMENT $ THIS IS NOT A RIGHT TRIANGLE$
                       END OF CONDITIONAL
                       TRANSFER TO START

                       END OF PROGRAM
            $DATA
            A = 3., B = 4., C = 5., EPSI = 0.1 *
            A = 4., B = 3., C = 5., EPSI = 0.05 *
            A = 5., B = 3., C = 4., EPSI = 0.01 *
            A = 5.1, B = 3.1, C = 3.9, EPSI = 0.03 *
            A = 5.1, B = 3.03, C = 4.1, EPSI = 0.05 *
            A = 8.9, B = 4.25, C = 1.4, EPSI = 0.01 *
```

Examples

EXAMPLE (Main program plus external function):

The following program segments are for the example discussed and flow charted on pages 744 and 745 (version II).

Complete Deck (U. M. System)

```
                    1         2         3         4         5         6         7
Column ——►  123456789012345678901234567890123456789012345678901234567890123456789012
             LUTHER CARNAWILKES            D066N               *05   005   000
             LUTHER CARNAWILKES            D066N               *05   005   000
            $ COMPILE MAD, EXECUTE, DUMP, PRINT OBJECT

                      R  MAD PROGRAM (MAIN PROGRAM) WHICH CALLS ON EXTERNAL FUNCTION
                      R  CALC.  CALC. FINDS THE MEAN OF THE N VALUES X(1)...X(N)
                      R  AND RETURNS AS ITS VALUE THE BOOLEAN CONSTANT 1B (TRUE) OR
                      R  0B (FALSE), DEPENDING ON WHETHER OR NOT THERE IS AT LEAST
                      R  ONE ELEMENT OF X WHICH IS NEGATIVE.

            START       READ DATA N, X(1)...X(N)
                        NEGTIV = CALC.(N,X,MEAN)
                        PRINT RESULTS N, X(1)...X(N), MEAN, NEGTIV
                        TRANSFER TO START

                        INTEGER N
                        BOOLEAN CALC., NEGTIV
                        DIMENSION X(100)

                        END OF PROGRAM
            $COMPILE MAD, PRINT OBJECT
                        EXTERNAL FUNCTION (N,Y,AVG)

                        INTEGER N, I
                        BOOLEAN NEGTIV

                        ENTRY TO CALC.
                        AVG = 0.
                        NEGTIV = 0B
                        THROUGH LOOP, FOR I = 1, 1, I.G.N
                        AVG = AVG + Y(I)/N
            LOOP        WHENEVER Y(I).L.0., NEGTIV = 1B
                        FUNCTION RETURN NEGTIV

                        END OF FUNCTION
            $ DATA
            N = 4, X(1) = 2.45, 0.00447, -12.33, 4.50 *
            N = 3, X(1) = 1.332E-4, 0.00476, -21.3E-5 *
```

Bibliography

1. Arden, B. W., An Introduction to Digital Computing, Addison-Wesley Publishing Co., Inc., 1963.

2. Galler, B. A., The Language of Computers, McGraw-Hill Book Co., 1962.

3. The MAD Manual, The University of Michigan Computing Center, 1963.

4. Organick, E. I., A Computer Primer for the MAD Language, 1961.