DIGITAL COMPUTER LABORATORY
GRADUATE COLLEGE
UNIVERSITY OF ILLINOIS

A User's Reference Manual For The Michigan Algorithm Decoder (MAD) For The IBM 7090

Library Routine L2-UOI-MAD1-2-RX

Revised Writeup Prepared by

R. Flenner

J. Flenner

Approved by [illegible]

Date June 20, 1962

TABLE OF CONTENTS

TABLE OF CONTENTS

TABLE OF CONTENTS

TABLE OF CONTENTS

TABLE OF CONTENTS

Acknowledgements

The bulk of the information contained in this manual was taken from the manual of the Michigan Algorithm Decoder (MAD) of February, 1962, written by Bruce Arden, Bernard Galler and Robert Graham of the Computing Center of the University of Michigan.

Also providing useful information was "A Primer for Programming with the MAD Language" from the first annual report _Electronic Computers in Engineering Education_, August 26, 1960, of the College of Engineering, University of Michigan, Ann Arbor, Michigan, sponsored by the Ford Foundation.

We are especially indebted to Professor Galler who provided answers to many questions regarding MAD.

Thanks are also extended to Mrs. Mary Ann Cross, Professor J. N. Snyder and Mr. W. J. Wulf for their comments regarding the presentation of the information.

Introduction

An algebraic language translator-compiler is a computer program which reads and translates mathematical algorithms written in a language close to mathematical notation, and produces, i.e., compiles, a computer program which when executed will perform the described algorithms.

MAD, the Michigan Algorithm Decoder, is an algebraic language translator-compiler, programmed at the Computing Center of the University of Michigan. This manual is a description of the MAD language, the language in which algorithms must be written in order to allow MAD to translate them. It is also a guide to the use of MAD.

This manual was designed and organized as a reference manual. It is not intended by itself to meet the requirements of a text book. The book, THE LANGUAGE OF COMPILERS: AN INTRODUCTION, by Bernard A. Galler, to be published by McGraw Hill, does, very satisfactorily, meet the requirements of a text book. The combined use of this manual and that text book should provide the user with a satisfactory introduction to MAD.

This goal of producing an easily used reference manual rather than a text has dictated several of the noticeable features of this book. No index is provided, but a complete Table of Contents labeled by a nested decimal classification system is used. No page numbers are used in the body of the work, but the material is labeled in parallel with the Table of Contents. Gaps provided in the Table of Contents and the use of loose-leaves will enable the manual to be conveniently revised and augmented.

In keeping with the reference nature of the manual, an attempt at precise definition of the terms and elements of the MAD language has been made even at the expense of often being quite formal.

CHARACTER SET

1 A Preliminary Survey

1.1 The Hollerith Character Set

The characters to be used in writing in the MAD language are as follows:

(1) Alphabetic Characters, or Upper Case Letters:
A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P, Q, R, S, T, U, V, W, X, Y, Z;

(2) Numeric Characters, or Decimal Digits:
0, 1, 2, 3, 4, 5, 6, 7, 8, 9;

(3) Special Characters (The characters ", quotation marks, are <u>not</u> characters of the language. They are used here to set off the characters): "+", "-" (minus), "'" (apostrophe), "=", "*", "/", "(", ")", ".", "," (comma), "$", " " (blank space, sometimes represented by the character "b").

Groups (1) and (2) are referred to collectively as Alphanumeric Characters.

1.2 The Sequence of Events in Using MAD

(1) The problem is expressed in the MAD language as described in this manual. This expression of the problem is a "source language program".

(2) The source language program is ingested by the MAD translator-compiler (see section 14, Mechanics of Using MAD). The MAD translator reads and translates the source language program and from it compiles a "machine language" computer program. This computer program is the "object program".

This phase is called "compilation". The time at which the program is compiled is said to be "compilation time".

(3) The object program is executed on the computer. This is the "execution phase" and the time at which the object program is executed is said to be "execution time".

1.3 Statements

A source language program is an expression of a problem in the form of algorithms which, when executed, provide the desired solution.

Algorithms are expressed in MAD by writing a series of "statements". Statements may be either executable statements or non-executable statements, known also as "declarations". It is possible further to subdivide statements into classes such as "control statements", "testing statements", "iteration statements", "input-output statements, etc. All statements which are available in the MAD language are explicitly defined in sections 3, 4, 5, 6 and 7.

Every statement must begin on a card, column 11 of which is blank. There may not be a remnant (a tail end) of a preceding statement on the same card. (This information appears elsewhere in this manual; it is given here, also, for emphasis to aid readers in overcoming preconceived notions due to familiarity with other translators.) Any card is recognized as the last card comprising a statement when the card which follows it has a blank in column 11.

1.4 MAD and the Operating System

All of the properties which are strictly properties of MAD and which are necessary to enable a user to write a MAD program are described in this manual. There are certain other facilities which might be desired and, which, although not described herein, may exist. Generally, if such options exist, they exist as properties of the operating system of which MAD is a part, and the user must consult the description of the properties of the pertinent operating system in order to determine what other useful options are available.

By the same token, it is true in most computing installations that, in order to run a MAD program to solve a problem, the existence of a MAD source program designed to do just that is not sufficient. Again, the operating system of which MAD is a part must be considered. The very properties of the operating system which make it flexible and useful impose requirements on a problem in addition to those necessary to satisfy MAD.

The intention of any operating system is to provide greatly increased flexibility and many useful options with but few accompanying extra requirements.

In particular it is in the province of MAD to translate the source language program and compile an object program. After this, MAD has no more control over the destiny of the particular problem. It is in the province of the operating system to load the MAD object program into the computer, and transfer control to it so that it may execute. After execution of the object program is completed, it is the operating system which assumes control of the computer and proceeds to the next problem.

Examples of properties which may be available to users of MAD but which are in reality properties of the containing operating system are:

(1) Compilation with the option of executing the object program on the same computer run;

(2) Execution of a problem which requires more than one "core load", each of which is an object program resulting from separate compilations. This option is variously known as "ping-pong" or "chaining".

(3) Translation and compilation processing of more than one MAD source program in a single computer run;

(4) Use of an assembly language translator and the MAD compiler together on the same problem.

2 Elements of the Language

In this section the basic building blocks of the language are described. These elements are used in the construction of the statments described in chapters 3, 4, 5, 6 and 7.

A brief discussion of the concept of "mode" is in order. It is a basic concept of the language. The "mode of an element" (and specific mention will be made throughout the remainder of the manual regarding the specific elements to which the concept is applicable), is determined by the quantities which the element in question may represent. The mode of any element in turn influences the manner in which operations are performed on the element.

There are five and only five modes in the MAD language. They are the following:

Integer
Floating Point
Boolean
Statement Label
Function Name

(a) Any element of integer mode will be interpreted as representing a value equal to an integer i such that i satisfies

$$-9999999999 \leq i \leq +9999999999$$

The decimal point is assumed to be immediately to the right of the rightmost digit and no fractional part may appear.

(b) Any element of floating point mode will be interpreted as representing a value equal to any real number r such that r satisfies:

$$10^{-38} \leq |r| \leq 10^{38}$$

Thus such an element is a "mixed number".

Example to cover (a) and (b):

Although the integer 1 is a real number 1 and the real number 2 is an integer, the integers 1 and 2 are quite distinct from the floating point numbers 1 and 2, respectively.

In particular, using octal digits (0 through 7) to represent groups of three binary bits (0, 1) the internal representations of 1 and 2 are as follows in actual internal IBM 7090 representation.

Number	Floating Point	Integer
1	201400000000	000000000001
2	202400000000	000000000002

(c) Any element of Boolean mode will be interpreted as representing a value equal to one of:

"true",
"false".

These values will be represented in MAD in the following way:

"true" by 1B,
"false" by OB.

(d) Any element of statement label mode will be interpreted as representing a value equal to a statement label. See section 2.3, Statement Labels, for a complete description.

(e) Any element of function name mode will be interpreted as representing a value equal to a function name. See section 2.4, Functions, for a complete description.

Although, abstractly, the value of an element of any given mode may be "equal" to the value of an element of one of the other four modes, the format of the representations internal to the computer is different and operations on the elements will be performed differently.

2.1 Constants

The following types of constants may be written in the MAD language.

2.1.1 Integer Constants

Integer constants may be one to ten decimal digits preceded by sign (- or +). Hence, integers can range from -9999999999 to +9999999999 with the decimal point always assumed to be immediately to the right of the rightmost digit but always omitted. Integer constants are of integer mode.

Signs

(1) Sign if present is always placed to the left of the integer.

(2) Negative sign is never omitted.

(3) Positive sign may be (and usually is) omitted.

EXAMPLE: 2, -2, 0, +0, 100 are all integers.

Leading Zeros

Leading (but not trailing) zeros may be omitted.

EXAMPLE: 5 and 005 are the same but 3 and 300 differ.

2.1.2 Floating Point Constants

There are two basic forms, with and without exponent. In all cases a floating point constant F must satisfy:

$$10^{-38} \leq |F| \leq 10^{38}$$

Floating point constants are of floating point mode.

(a) Without exponent the constant contains one to eight digits and a decimal point (.) which must be written but which may appear anywhere in the number. Examples:

0., 1.5, -0.05, +100.0, -4.

(b) With exponent the constant contains from one to eight digits with or without a decimal point, followed by the letter E, followed by the exponent. Exponent is one or two digits preceded by sign and represents the power of ten by which the number to the left of the exponent is to be multiplied.

Examples:

MAD NOTATION	MEANING
.05E-2	$.05 \times 10^{-2}$
-.05E2	$-.05 \times 10^{2}$
5E02	5×10^{2}
5.E2	5×10^{2}

Note that:

(1) Plus signs may be optionally omitted in front of the exponent as well as in front of the number itself.

(2) Decimal point may be omitted, in which case it is assumed to be immediately to the left of the letter E.

2.1.3 Alphabetic Constants

Alphabetic constants consist of from one to six characters from the following set of admissible characters:

"A", "B", ..., "Z": alphabetic characters.
"0", "1", ..., "9": decimal digits.
"+", "-" (minus), "'" (apostrophe), "=", "*",
"/", "(", ")", ".", "," (comma), " " (blank space, sometimes represented by the character "b").

Each alphabetic constant must be delimited on the left and on the right by the special character "$". The mode of alphabetical constants used in arithmetic expressions is integer.

Notice especially that although blank spaces are ignored elsewhere in the language, except where it is specifically indicated that they are not, they count as characters in alphabetic constants.

EXAMPLES: $ABCD$, $TØ BE$, $DEC. 4$, $5 +3=8$.

NOTE: An alphabetic constant is stored internally as an integer. Any alphabetic constant which is written with fewer than six characters will be stored internally left justified with blanks appended on the right; thus, $ABCD$ will be stored internally as if it had been written $ABCDbb$.

Since the character "$" serves to delmit an alphabetic constant, it is not possible to write an alphabetic constant with "$" as one of the internal characters. An alphabetic constant with the character "$" as the left-most of the internal characters may be created, however, by writing the following alphabetic constant:

-$=$

2.1.4 Boolean Constants

There are two Boolean constants:

1B is written for "true",
OB is written for "false".

2.1.5 Octal Constants

Octal constants may be written in two ways, with or without scale factor. The mode of octal constants is integer.

Without scale factor an octal constant is written with from one to twelve octal digits followed by the letter "K".

EXAMPLE: 03K,123K,777777777777K

With scale factor one or more decimal digits follows the "K". This decimal integer is interpreted as the exponent of the power of eight by which the octal integer preceding the "K" is to be multiplied.

EXAMPLE: 127K2 becomes 000000012700,
1K10 becomes 010000000000.

2.1.6 Statement Label Constants

See section 2.3, Statement Labels.

2.1.7 Function Name Constants

See section 2.4, Functions.

2.1.8 Redefinition of The Mode of Constants

Any constant whether Integer, Floating Point, Boolean, Alphabetic or Octal, may be declared to be of a mode other than its normal mode.

CONSTANTS

When a constant which is to have its mode redefined is <u>used</u> it is written in the form

cMi

where "c" is the constant in question written as described above for each of the five types, "M" is the character "M" written following the last character of "c" to indicate mode redefinition and i is one of the integers

0 for Floating Point
1 for Integer
2 for Boolean
3 for Function Name
4 for Statement Label

The conversion performed on the constant in reading it and the form in which it appears internally is that associated with its <u>original</u> mode. The mode of the constant is reassigned to be the mode whose code is i <u>after</u> the standard processing of the constant is completed.

2.2 Variables

The name of any variable consists of from one to six alphabetic or numeric characters, the first of which must be an alphabetic character. The name used for any variable must not be identical to any statement label or to the alphanumeric part of any function name.

EXAMPLES: TEMP,KLMZ,F55,P32K,RESULT,X,ALPHA

2.2.1 Simple Variables

A simple variable is a variable whose name refers to a unique element. It may be thought of as a degenerate vector with one element, or as a matrix of order (1,1).

Simple variables are written in the form

NAME

where NAME adheres to the form of a variable name as defined above. Each of the names in the immediately preceding example is written as a simple variable.

2.2.2 Arrays

An array is a set of elements. The number of elements in the set is greater than one. Every variable name which is the name of an array must be so declared as explained in section 3.6, DIMENSIØN Declarations.

Arrays of order (m,1) or (1,n) where m and n > 1, are "Vectors", "Linear Arrays", or "One Dimensional Arrays". Vectors are stored internally with the elements in adjacent locations.

Two dimensional arrays, or matrices, of order (m,n) are stored internally by rows, i.e., in the order:

$$a_{11},a_{12},\ldots,a_{1n},a_{21},a_{22},\ldots,a_{2n},\ldots,a_{m1},a_{m2},\ldots,a_{mn} .$$

In general, n-dimensional arrays, where $n \geq 2$, are stored in the order determined by varying the rightmost subscript first, then the next rightmost, etc. Thus, a four dimensional array of order (m,n,p,q) would be stored in the order:

$a_{1111}, a_{1112}, \ldots, a_{111q}, a_{1121}, a_{1122}, \ldots, a_{112q}, \ldots, a_{11pq}, a_{1211},$

$\ldots, a_{1npq}, \ldots, a_{mnpq}$.

It is possible in MAD to regard any array of dimension n, even though n may be greater than one, as a linear array. The order of the elements of an array when regarded as a linear array is as described in the preceding paragraph. Any element of an array may thus be referred to by a subscripted variable with a subscript of one element rather than n. This single element subscript of an array element is called a "linear subscript" of the array element. The value of the linear subscript is equal to the position of the given element relative to the element with all subscripts zero, the linear subscript of which is also zero.

See sections 2.2.4, Subscripts of Subscripted Variables, and 3.6.2, Matrix DIMENSIØN Declaration, for additional comments concerning linear subscripts.

2.2.3 Subscripted Variables

A subscripted variable is written in the form

$$\text{NAME } (S_1, S_2, \ldots, S_n)$$

where NAME adheres to the form of a variable name as defined above, and must be the name of an array variable, and each S_i, i = 1, 2, ..., n, is a permissible subscript as described in sections 2.2.4, Subscripts of Subscripted Variables, and 2.2.9, Statement Label Variables. The characters "(" and ")" delimit the subscript and the character "," separates the elements of the subscript. Notice that the subscripts written in the MAD language are written on the same line as the name they are subscripting and are not written below the line as in usual mathematical notation.

At any given time a subscripted variable is the name of a single specific element of an n-dimensional array of elements.

At any given time the value of a subscripted variable is a single specific element of an n-dimensional array of elements.

It is a convention of the language that if a name, NAMEA, which is the name of an array, is written alone in an expression as if it were a simple variable, i.e.,

NAMEA

its value is equivalent to the value of the subscripted variable

NAMEA(0,0,...,0).

Examples of subscripted variables whose values are array elements:

```
BETA(I)
X(J,5)
Y(7)
N(B+4*F,Q)
MA(K(Z+5)+T(1)+6)
BETA ≡ BETA(0)
X ≡ X(0,0)
```

2.2.4 Subscripts of Subscripted Variables

With one exception, the following discussion applies to all types of subscripted variables. See section 2.2.9, Statement Label Variables, for the exception.

As far as the translator is concerned, any arithmetic expression (see section 2.6, Arithmetic Expressions) may be used as a subscript when the subscript is a linear subscript. This is true whether the dimension of the array is one or greater than one.

The qualification "as far as the translator is concerned" arises from the fact that the value of the expression must be greater than or equal to zero. This restriction applies when the expression is evaluated at execution time; violations are not discernible by the translator at compilation time.

If the value of the subscript expression used as a linear subscript is in floating point mode (see section 2.6.4, Mode and Arithmetic Expressions), it will be truncated to an integer before it is used as a subscript.

When, in the general form of a subscripted variable, the dimension $n \geq 2$, each subscript element may be an arithmetic expression but the expression must be of integer mode. Moreover, the use of subscript expressions having values of other than integer mode will go undetected by the translator when $n \geq 3$. Unlike the case of linear subscripts, it may be meaningful for a subscript expression appearing as an element in a non-linear subscript, to have a value less than zero. Care must be exercised. See section 3.6.2, Matrix DIMENSIØN Declaration, for a complete discussion.

2.2.5 Integer Variables

All of the preceding general discussion of variables applies to the integer variables. Values represented: any integer number I satisfying

$$-9999999999 \leq I \leq +9999999999$$

These variables are of integer mode.

2.2.6 Floating Point Variables

All the preceding general discussion of variables applies to the floating point variables. Values represented: any real number F satisfying

$$10^{-38} \leq |F| \leq 10^{38}.$$

These variables are of floating point mode.

2.2.7 Boolean Variables

The values represented by Boolean variables are

1B = true,

0B = false.

These variables are of Boolean mode.

The discussions under the sections Simple Variables, Array Variables, Subscripted Variables and Subscripts of Subscripted Variables apply to Boolean variables.

2.2.8 Function Name Variables

The values represented by function name variables are any function name, see section 2.4, Functions. These variables are of function name mode.

The discussions under the sections Simple Variables, Array Variables, Subscripted Variables and Subscripts of Subscripted Variables apply to function name variables.

2.2.9 Statement Label Variables

The values represented by statement label variables are any statement labels, see section 2.3, Statement Labels. These variables are of statement label mode.

The discussion of Simple Variables applies to statement label variables. Subscripted statement label variables are more restricted than general subscripted variables. Because of the way in which elements of statement label vectors are defined, "arrays" of statement label mode are restricted to one dimensional arrays. Thus, subscripts of subscripted statement label variables must satisfy $n = 1$, of the general form of a subscripted variable described in section 2.2.3, Subscripted Variables.

2.3 Statement Labels

Statements are defined in **sections** 3 and 4. A statement must be labeled when it is desired to have another statement refer to it.

A statement label may take on one of two forms:

(a) C

where C consists of from one to six alphabetic characters or decimal digits, the first of which must be alphabetic;

(b) C(n)

where C is as in (a), n is an integer constant and the special characters "(" and ")" delimit n on the left and right respectively. A statement label of this form is an element of a statement label vector.

A statement label is a constant of statement label mode.

A statement label must not be identical to any variable name or to the alphanumeric part of any function name.

A statement label appears in the label field, columns 1 through 10, of the statement it identifies. When a statement extends to additional cards (see section 14.1.2.2, Continuation Cards), the statement label need not be punched on the additional cards.

2.4 Functions

The name of a function is written in the form

C.

where C may consist of from one to six alphabetic characters or decimal digits, the first of which must be alphabetic. The special character "." is part of the name and must follow the last character of C. A function name is a constant of function name mode. The alphanumeric portion of the name must not be identical to any variable used in the program, or to any statement label.

Examples of function names:

```
SIN.
SQRT.
ELØG.
```

2.4.1 Single Valued Functions

The value of a single valued function is represented, in general, by the form:

$$C.\ (A_1,A_2,\ldots,A_n)$$

In the form, "C." is a function name and each A_i, $i = 1,2,\ldots,n$, is an argument of the function (see section 2.4.5, Arguments of Functions). The special characters "(" and ")" delimit the field of arguments and the special character "," separates pairs of arguments. The single valued functions may be used by specifying the appropriate value form as an operand in any arithmetic expression (see section 2.6, Arithmetic Expressions).

The value form of a function is also often referred to as the "call" for the function, i.e., the value form "calls" for the function to be evaluated.

Examples of values of single valued functions:

```
SIN.(3.141596)
SQRT.(X+Y*Z-10.)
MLTRGF.(A,B-C*D,2.*X,5)
```

2.4.2 Non-Single Valued Functions

Since the result of the operation of a single valued function upon its arguments is by definition a single value, it made sense in section 2.4.1 to define the value of a single valued function. In MAD the facility exists for allowing arguments to be operated upon by "functions" which may not be used in any expression. These are called non-single valued functions. The notation to indicate such an operation is identical to that used for the value of a single valued function, namely,

$$C.\ (A_1,A_2,\ldots,A_n)$$

where the elements of the form are exactly as described in section 2.4.1, Single Valued Functions. The only distinction, which is, however, a non-trival one, is that the name used for any non-single valued function must be different from the names of all single valued functions and conversely.

The additional mechanism required to execute non-single valued functions is explained in section 4.8 EXECUTE Statement.

Example of the specification of a non-single valued function which is to sort the n elements of some list according to prescribed rules:

S∅RT.(LIST,N)

2.4.3 Translator Defined Functions

Certain commonly used functions are already available to the MAD translator. The programmer may use these functions merely by writing them in the appropriate manner. A list of the currently available translator defined functions may be found in the manual describing library subroutines.

2.4.4 Programmer Defined Functions

The facility exists in MAD to permit the programmer to define for his use functions of any complexity. Two types of functions, "internal functions" and "external functions", may be defined. The procedure for defining functions is explained in section 3.8 Single Statement INTERNAL FUNCTI∅N Definition, and in section 6, Function Definitions.

The form to indicate the use of any function of either of these types is the same as that described in section 2.4.1, Single Valued Functions. Programmer defined functions may be either single valued or non-single valued functions.

2.4.5 Arguments of Functions

In general, as far as the MAD translator is concerned, any argument of any function may be any expression. For a given function however, the function itself imposes restrictions on the arguments. Therefore, in determining the restrictions upon the arguments for a given function, it is necessary to consult the function description.

2.4.6 The Mode of a Single Valued Function

By "the mode of a function" is meant "the mode of the value of a function", e.g., if the mode of the value of a function is integer, then the mode of the function is integer.

The mode or modes of various arguments of a function do not prejudice the mode of the function. There may be functions all of whose arguments may have modes differant from the mode of the value of the function.

2.5 Arithmetic Operations

A number of unary and binary arithmetic operations are available. The arithmetic of the binary operations is integer arithmetic or floating point arithmetic according to the modes of the operands. If both binary operands are integer then integer arithmetic is used. If both operands are floating point, floating point arithmetic is used. If the operands are of mixed modes, the operand in the integer mode is first converted to floating point; the arithmetic operation is performed as a floating point operation, yielding a floating point result.

2.5.1 Unary Operations

These operate on the single operand immediately to the right of the operator.

Operation Symbol	Definition	Example
.ABS.	Absolute Value	.ABS.(X+Y) means |(X+Y)|
+	Identity	+(X/Y) means (X/Y)
-	Negation	-(Q*P) means -Q*P

2.5.2 Binary Operations

These operations involve the operand to the left and the operand to the right of the operation.

ARITHMETIC OPERATIONS

Oper. Symbol	Definition	Example
+	Addition	Z4+QUØ
-	Subtraction	A-XYZ
*	Multiplication. Juxtaposition may not be used to signify multiplication.	2.3*I
/	Division. If both operands are integers the result is made an integer. The fractional part of the true quotient is <u>truncated</u> (not rounded).	D3B/(C-3)
.P.	Exponentiation. Raise the left operand to the power which is the value of the right operand.	VAR.P.2 means $(VAR)^2$

2.6 Arithmetic Expressions

2.6.1 Definition of Arithmetic Expressions

(a) Every unsigned constant, whether integer, floating point or alphabetic, is an arithmetic expression. Every individual variable, whether integer or floating point, is an arithmetic expression. Every subscripted array variable, whether integer or floating point, is an arithmetic expression. Every value of a single valued function, whether integer or floating point, is an arithmetic expression.

(b) If V is any arithmetic expression of a type in (a) above then both

```
+ V
- V
```

are arithmetic expressions.

(c) If E is any arithmetic expression, then

```
(E)
```

is an arithmetic expression, and also both

```
+ (E)
- (E)
```

are arithmetic expressions.

(d) If F is any arithmetic expression, then all of the following are arithmetic expressions:

```
.ABS. F
F + V
F + (E)
F - V
F - (E)
F * V
F * (E)
F / V
F / (E)
F .P. E
```

(e) The only arithmetic expressions are those arising in (a) through (d) which do not exceed a length which can be contained on ten cards.

2.6.2 Hierarchy of Operations in Arithmetic Expressions

The order in which individual terms of an expression are to be evaluated and combined must be unique and in the MAD language is made so by defining the hierarchy of the arithmetic operations. Unless altered by parentheses the order of arithmetic operations performed within an arithmetic expression is given by the following list:

Operation Symbol	
.ABS. , +	(unary operations) (equal hierarchy)
.P.	
-	(unary operation)
* , /	(equal hierarchy)
+ , -	(binary operations) (equal hierarchy)

Within an expression, operations of equal hierarchy are performed from left to right unless otherwise indicated by parentheses.

Example 1: The expression

```
A + B/C + D .P. E*F - G
```

means

$$A + \frac{B}{C} + D^{E} \times F - G$$

Example 2:
```
A * B/C * D/E * F
```

means

$$\frac{\frac{AB}{C} \times (D)}{E} \times F$$

Example 3: C(K) + A(3) * B(J)/9.7 + 3.5 * P

means

$$C(K) + \frac{A(3) * B(J)}{9.7} + 3.5 * P$$

Example 4: A + B - C + D - E

means

(((A + B) - C) + D) - E

Example 5: X/Z * Y/R * S

means

(((X/Z)* Y) /R) * S

2.6.3 Parentheses in an Arithmetic Expression

Parentheses as used in the usual algebraic sense may be used to override the usual rules of precedence for a given expression and are frequently used for the purpose of simplifying mathematic expressions.

Example 1: A * B - C/D + E * F

means

$$AB - \frac{C}{D} + EF$$

but

A * (B - C)/(D + E) * F

means

$$\frac{A(B - C)}{(D + E)} F$$

Example 2: (C(K) + A(3) * B(J))/(9.7 + 3.5 * P)

means

$$\frac{C(K) + A(3) * B(J)}{9.7 + 3.5 * P}$$

2.6.4 Mode and Arithmetic Expressions

As just defined, it is possible for an arithmetic expression to contain operands of either integer mode or floating point mode. When an arithmetic expression is evaluated, the result, which is the "value of the arithmetic expression", must also be either of integer or floating point mode. Thus it is proper to speak of the mode of the value of an arithmetic expression; a somewhat looser terminology, "the mode of an arithmetic expression", is sometimes used in place of this.

In an arithmetic expression an alphabetic constant is a constant of integer mode.

An arithmetic expression is considered to be in the floating point mode if <u>any</u> operand of any arithmetic operation in the expression is in the floating point mode. If all operands are integer (or alphabetic), then the expression is considered to be in the integer mode. In this determination arguments, though not values, of functions are ignored.

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

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

are all floating point expressions while the expressions

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

are all integer expressions.

ARITHMETIC EXPRESSIONS

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

$$Y + 3$$

if Y is in the floating point mode it cannot be added directly to the integer 3. But for one precaution the user need not be concerned with this since the instructions necessary for the conversion of the integer to floating point form before adding are automatically inserted by the translator during the translation process. The precaution is that if the integer being converted is greater than 134,217,728 (i.e., 2^{27}) then an improper conversion will take place.

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

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

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

$$T_1 = I * J$$
$$T_2 = T_1/K$$
$$T_3 = 7/3$$
$$T_4 = Y + T_3$$
$$T_5 = T_4 + T_2$$

and T_5 will be the value of the expression.

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

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

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

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

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

2.7 Mathematical Relations

In order to permit comparison of the algebraic values of pairs of arithmetic expressions (see section 2.6, Arithmetic Expressions), statement label expressions (see section 2.10, Statement Label Expressions) or function name expressions (see section 2.11, Function Name Expressions), the following mathematical relations are provided:

MAD Symbol	Mathematical Symbol	Meaning
.L.	$<$	E.L.F means "E is less than F"
.LE.	$\leq$	E.LE.F means "E is less than or equal to F"
.E.	$=$	E.E.F means "E is equal to F"
.NE.	$\neq$	E.NE.F means "E is not equal to F"
.G.	$>$	E.G.F means "E is greater than F"
.GE.	$\geq$	E.GE.F means "E is greater than or equal to F"

2.8 Boolean Operations

The following Boolean, or logical, operations are available. Let M and P be Boolean Expressions as defined in section 2.9, Boolean Expressions.

MAD Operation Symbol	Corresponding Logical Symbol	Definition
.NØT.	$\sim$	The value of .NØT.M is 1B if and only if the value of M is OB.
.OR.	$\vee$	The value of M.ØR.P is OB if and only if both M and P have the value OB.
.EXØR.	$\oplus$	The value of M.EXØR.P is 1B if and only if exactly one of M or P has the value 1B.
.AND.	$\wedge$	The value of M.AND.P is 1B if and only if both M and P have the value 1B.
.THEN.	$\supset$	The value of M.THEN.P is OB if and only if the value of M is 1B and the value of P is OB.
.EQV.	$\equiv$	The value of M.EQU.P is 1B if and only if the value of M is equal to the value of P.

2.9 Boolean Expressions

2.9.1 Definition of Boolean Expressions

(a) Boolean constants, Boolean simple variables, Boolean subscripted array variables and Boolean-valued single valued functions are Boolean expressions (see section 2.1.4, Boolean Constants, 2.2.7, Boolean Variables, and 3.2, Mode Declaration).

(b) If E and F are arithmetic expressions, then the following are Boolean expressions:

```
E  .L. F
E .LE. F
E  .E. F
E .GE. F
E  .G. F
```

(c) If G and H are <u>both</u> statement lable expressions or <u>both</u> function name expressions, then

```
G  .E. H
G .NE. H
```

are Boolean expressions.

EXCEPTION: If G or H are elements of a vector preset by a VECTØR VALUES statement (see section 3.7, VECTØR VALUES Declaration), then G .E. H and G .NE. H are not expressions.

(d) If M and P are Boolean expressions, then the following are Boolean expressions:

```
(M)
.NØT. M
M .ØR. P
M .EXØR. P
M .AND. P
M .THEN. P
M .EQV. P
```

(e) The only Boolean expressions are those arising from (a) through (d).

Examples of Boolean expressions:

```
(X .G. 3 .AND. Y .LE.2).ØR.(GAMMA.L.EPSILN)
(.ABS.(X1-X2)/X1.LE.EPSILN).AND.(F.(X1).L.EPSILN)
((P .AND. M) .THEN.M) .EQV. (P.ØR. .NØT.P)
```

2.9.2 Hierarchy of Operations in Boolean Expressions

As for arithmetic expressions, the hierarchy of operations which may appear in a Boolean expression must be defined in order to establish a unique order in which the terms of the expression are evaluated and combined. Unless altered by parentheses, the order of operations performed within a Boolean expression is given by the following list:

<u>Operation Symbols</u>

.ABS., + (equal hierarchy)

.P.

-

*, /(equal hierarchy)

+, - (equal hierarchy)

.E., .NE., .G., .GE., .L., .LE. (equal hierarchy)

.NØT.

.AND.

.ØR., .EXØR. (equal hierarchy)

.THEN.

.EQV.

Within an expression, operations of equal hierarchy are performed from left to right unless otherwise indicated by parentheses.

Examples:

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

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

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

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

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

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

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

(8) P.AND..NØT.P .EQV.Q means (P.and.(.NØT.P)).EQV.Q

2.9.3 Parentheses in Boolean Expressions

Parentheses are used in the usual way in Boolean expressions.

2.9.4 Mode and Boolean Expressions

If an expression is a Boolean expression (see section 2.9.1, Definition of Boolean Expressions), then the expression has a value which is of the Boolean mode; a somewhat looser terminology is to say that "the mode of the expression is Boolean".

2.10 Statement Label Expressions

A statement label expression may be any of the following:

(a) a statement label constant;

(b) a statement label variable;

(c) a function value of statement label mode.

There are no other statement label expressions.

If an expression is a statement label expression, then the expression has a value which is of statement label mode; a somewhat looser terminology is to say that "the mode of the expression is statement label".

Examples:

(a) If each of the following:

```
LABEL
C(3)
XYZPDQ
```

appears in the label field of some statement, then each one is a constant of statement label mode and hence each is a statement label expression.

(b) If each of the variables

```
X
YZABC
IJK(Q)
```

has been declared to be of statement label mode (see section 3.2, Mode Declaration), then each is a statement label expression. Notice that the entire array IJK must be of statement label mode.

(c) If the function value given by

```
PAIR.(X,YZABC,IJK(Q))
```

is of statement label mode, then this is a statement label expression.

2.11 Function Name Expressions

A function name expression may be any one of the following:

(a) a function name constant;

(b) a function name variable;

(c) a function value of function name mode.

There are no other function name expressions.

If an expression is a function name expression, then the expression has a value which is of function name mode; a somewhat looser terminology is to say that "the mode of the expression is function name".

Examples:

(a) If each of the following:

```
SIN.
ATAN.
FUNCTN.
```

is a function name, then each is a function name constant and hence each is a function name expression.

(b) If each of the variables

```
A123
C2BX11
LØGA(12,P)
```

has been declared to be of function name mode (see section 3.2, Mode Declaration), then each is a function name expression.

(c) If the function value given by:

```
XYZF.(A1,A2)
```

is of function name mode, then this is a function name expression.

2.12 Summary of Expressions

Four kinds of expressions have been defined and may occur in the MAD language:

Arithmetic Expressions, section 2.6.

Boolean Expressions, section 2.9.

Statement Label Expressions, section 2.10.

Function Name Expressions, section 2.11.

Of these, the Boolean expression is the most general since all of the others may appear as sub-expressions of a Boolean expression.

When the terminology "any expression" is used, it is to mean any of the above four types of expressions.

3 DECLARATIONS (Non-executable statements)

The purpose of declarations is to furnish information to the translator. With the exception of function declarations, all such statements may occur anywhere in a program. Declarations may have statement labels, but they are ignored by the translator and may not be referenced by other statements.

3.1 Remark Declaration

Each card containing a Remark declaration must have an "R" in column 11. The statement itself in columns 12-72 is any string of allowable characters (see sec. 1.1, The Hollerith Character Set) and is ignored by the translator. It is reproduced where it occurred in the printed listing of the source language program, thus furnishing information to the reader of the program.

Example:

Cols. 1-10	Col. 11	Cols. 12-72	Cols. 73-80
	R	THE FOLLOWING IS THE 3RD OF A SET OF 5. 3/12/62	

3.2 Mode Declaration

Variable and function values may be declared to be one of the following modes:

FLØATING PØINT
INTEGER
BØØLEAN
FUNCTIØN NAME
STATEMENT LABEL

The form of the mode declaration is

$$\mathcal{M}\ U_1, U_2, \ldots, U_n$$

where $\mathcal{M}$ is one of the five modes listed above and each U_i, separated from the next by ",", is a variable name or function name. In particular, no subscripted variable form may appear in the mode declaration. No U_i may be declared to be of more than one mode throughout a program.

Examples: INTEGER ADDN., Z5X, ALPHA
BØØLEAN BAR, ANA., N32
FUNCTIØN NAME BETA, CLT.

MAD assumes the mode of all variables and function names to be FLØATING POINT unless otherwise declared. This normal mode may be altered by a statement of the form

NORMAL MODE IS $\mathcal{M}$

where $\mathcal{M}$ is one of the five modes listed above. Only one such statement may occur in a given program and this declared normal mode is in effect throughout the program, regardless of where it appears in the program.

Also:

(1) All constants are assigned a mode consistent with their respective forms. (See sec. 2.1, Constants.)

(2) A vector which is a dimension vector of some array in a DIMENSIØN declaration is assigned INTEGER mode (see sec. 3.6.2, MATRIX DIMENSIØN).

(3) A vector which is preset by a VECTØR VALUES declaration is assigned a mode consistent with the assigned values (see sec. 3.7 VECTØR VALUES).

3.3 EQUIVALENCE Declaration

The EQUIVALENCE declaration is of the form:

$$\text{EQUIVALENCE}\ (V_1, V_2, \ldots, V_n), (V_{n+1}, V_{n+2}, \ldots, V_{n+p}), \ldots, (V_{n+p+q+1}, V_{n+p+q+2}, \ldots, V_{n+p+q+r})$$

where each V_i is a variable name or a subscripted variable, the subscript of which is linear. All the elements indicated within a pair of parentheses will be assigned to occupy the same storage location throughout the program. An array name V written without subscript is by convention taken to mean the element V(0, 0, . . . , 0). If one array element from each of two different arrays appears within a pair of parentheses these are made equivalent and a one-to-one equivalence is thereby induced upon the overlapping remaining elements of the two arrays. One element from each of any number of different arrays may appear within a pair of parentheses.

Variable names within a group need not be of the same mode; the mode must be established by the appropriate mode declaration for each variable name.

Any number of groups of equivalences may occur in any one EQUIVALENCE declaration (up to ten cards/statement), and any number of EQUIVALENCE declarations may appear at any place within a program.

Examples:

(a) A single statement:

```
EQUIVALENCE  (A, B, C2), (XLF, TSH), (P43A, XXX, ZZZ), (Q, R), (SIX, I6)
```

(b) A portion of a program:

```
        .
        .
        .
EQUIVALENCE  (A, B, C2), (XLF, TSH)
        .
        .
        .
EQUIVALENCE  (P43A, XXX), (Q, R)
EQUIVALENCE  (XXX, ZZZ)
        .
        .
        .
EQUIVALENCE  (SIX, I6)
        .
        .
        .
```

The results of examples (a) and (b) are identical.

3.4 PRØGRAM CØMMØN Declaration

The PRØGRAM CØMMØN declaration has the form

PRØGRAM CØMMØN $V_1, V_2, \ldots, V_n$

where each V_i is a variable name separated from the next by ",". This declaration causes the specified simple variables and entire arrays to be stored in an area separate from the usual storage and separate from ERASABLE storage, section 3.5. These variables are not stored overlapping in storage as in EQUIVALENCE. They are stored successively in order of their appearance.

Reoccurrences of the PRØGRAM CØMMØN declaration do not erase the variables already assigned to PRØGRAM CØMMØN; the new assignments are appended to the previous list.

(1) One use of this declaration is the provision for several sections of a program to refer to variables and arrays by the same name, thus allowing the sections of the program to be translated and checked out independently of each other. A program segmented in this fashion would have the form of a main program plus several EXTERNAL FUNCTIØN programs, section 6.5, with the main program being used primarily to call on each of the EXTERNAL FUNCTIØN programs. Both the main program and the segments must contain the necessary PRØGRAM CØMMØN declarations. Although variables and arrays to be used jointly by several EXTERNAL FUNCTIØNs can be communicated as arguments to the functions, assigning them to PRØGRAM CØMMØN makes them available with greater ease to all of the functions. The reservation of PRØGRAM CØMMØN storage is performed only once and for the main program only; the PRØGRAM CØMMØN declaration in the EXTERNAL FUNCTIØN segment allows the proper storage references to be made. If PRØGRAM CØMMØN is to be used in this way (main program and EXTERNAL FUNCTIØNs) the main program must be loaded into the computer first.

(2) Another use for the PRØGRAM CØMMØN declaration is in the situation where a program is so large that it cannot all be contained in the computer at once. The program must then be written in segments, and if one segment is to use the results of a previous segment's computation, the variables involved should be declared in PRØGRAM CØMMØN storage and will be retained throughout. The PRØGRAM CØMMØN and DIMENSIØN (sec. 3.6) declaration

which set up the storage allocation must be identical in all segments which use these variables and arrays. Additional variables and arrays may be added to the end of the PRØGRAM CØMMØN list by any segment.

3.5 ERASABLE Declaration

The ERASABLE declaration has the form

$$\text{ERASABLE}\ \ V_1, V_2, \ldots, V_n$$

where each V_i is a simple variable or an array, separated from the next by ",". The variables and arrays assigned to ERASABLE are not overlapping as in EQUIVALENCE, but are put in a storage separate from the usual and separate from PRØGRAM CØMMØN. Each ERASABLE declaration deletes the effect of any previous ERASABLE declaration, thus allowing variables and arrays to occupy the same storage at different times.

Notice that external functions and translator defined functions may, and do, make use of erasable storage. Therefore care must be exercised if the programmer wishes data in erasable storage to remain intact after the operation of such a function. The data which is to be preserved must appear as an entry in an ERASABLE declaration so that entries which appear to the left of it represent at least as much erasable storage as is to be used by the function which is to execute in the interim and which requires the greatest amount of erasable storage. The entry or entries to the left of the critical entry in the ERASABLE declaration may be a dummy or dummy entries solely for the purpose of skipping over the non-safe erasable storage.

Example: Suppose the following program segment appears:

```
.
.
.
ERASABLE  X, Y, Z
.
.
.
Z = 12.7
.
.
.
R = SQRT. (A)
.
.
.
T = R + Z
.
.
.
```

where SQRT. is assumed to use two locations of erasable storage and X and Y are assumed to be expendable at least for this segment. The variable Z is calculated before the SQRT. function is executed and is used afterwards. The expendable variables X and Y, by virtue of appearing to the left of Z in the ERASABLE declaration, will be assigned the erasable locations which will be destroyed by the operation of SQRT., thus preserving Z until required, since Z is assigned to a safe location one beyond those destroyed by SQRT..

3.6 DIMENSIØN Declarations

Vectors and matrices may be declared in the same DIMENSIØN declaration. The separation of their descriptions here is done for the sake of clarity. If a variable is to be an array it must be declared in a DIMENSIØN declaration; this declaration need not appear in the program before the first use of the name.

3.6.1 Vector DIMENSIØN Declaration

The form of a DIMENSIØN declaration for vectors (one-dimensional arrays) is

$$\text{DIMENSIØN } V_1(p_1),\ V_2(p_2),\ \ldots,\ V_n(p_n)$$

where each V_i is a variable name and is followed by an integer constant enclosed in parentheses. This integer constant is the largest value that the subscript of V_i will assume during execution of the program. The size of the region reserved for the array V_i will be this integer +1. The "," separates each declared vector from the next. The subscript of an element of a vector array should not attain a value less than zero during execution of the program.

3.6.2 Matrix DIMENSIØN Declaration

The form of a DIMENSIØN declaration for matrices (n - dimensional arrays where $n \geq 2$) is

$$\text{DIMENSIØN } V_1(p_1,D_1(x_1)),\ V_2(p_2,D_2(x_2)),\ \ldots,\ V_m(p_m,D_m(x_m))$$

where each V_i is a variable name (other than Statement Label) and is followed by two arguments enclosed in parentheses. (The general form of a subscripted variable which is an element of V_i is

$$V_i\ (s_1,\ s_2,\ \ldots,\ s_{n_i})$$

where n_i is the number of dimensions of V_i). Each p_i, the first argument, as in vector DIMENSIØN above, is an integer constant whose value is the largest that the linear subscript of V_i can assume. Each $D_i(x_i)$, the second argument, defines the "dimension vector" for the associated V_i array in the following way:

D_i	is a variable name;
x_i	is an integer constant designating a specific element in the D_i vector;
$D_i(x_i)$	contains an integer constant, n_i, whose value is the number of dimensions of V_i;
$D_i(x_i+1)$	contains the linear subscript (integer) of the V_i array which is also to be the base element of the V_i matrix, i.e., V(1, 1, ..., 1);
$D_i(x_i+2)$	contains an integer constant which is the largest value that the 2nd subscript of V_i may assume;
.	.
.	.
.	.
$D_i(x_1+n_i)$	contains an integer constant which is the largest value that the nth subscript of V_i may assume.

(The lowest value of each subscript of V_i is assumed to be 1).

The dimension vector, $D_i(x_i)$, is automatically of INTEGER mode and may not be declared other than INTEGER. D_i must itself be dimensioned, either by DIMENSIØN or by VECTØR VALUES (sec. 3.7); D_i must be dimensioned to have its largest linear subscript $\geq x_i+n_i$.

As is mentioned above, it is possible to declare an array in such a way that the linear subscript of the base element (the element with all subscripts = 1) is greater than 1. When this is done it becomes meaningful, and it is permitted, to refer to the elements with linear subscripts less than the linear subscript of the base element but not less than the linear subscript zero, using the multi-element subscript (non-linear subscript) form with the values of the subscripts equal to zero and even less than zero. See example 3 below.

Examples:

(1) DIMENSIØN XA(400,V), V(3)

XA is a 401-element array whose elements may be referred to as XA(0), XA(1), ..., XA(400). V contains the 3-element dimension vector for the matrix XA (V is itself a 4-element array and is automatically of INTEGER mode); if V has been preset as follows:

V(0) = 2;
V(1) = 6;
V(2) = 13;
V(3) may be used as an integer variable elsewhere;

and if XA(I, J) is any element in the matrix XA:

(a) XA has 2 dimensions, since V(0) = 2;
(b) the base element of XA, XA(1,1), is XA(6), since V(1) = 6;
(c) the allowable range of J is $-5 \leq J \leq 13$, since V(2) = 13.

So that the elements XA(0) through XA(5) are not "within" the matrix XA, and XA(6), XA(7), ..., XA(395), ..., XA(400) may be referred to as XA(1,1), XA(1,2), ..., XA(30, 13), ..., XA(31, 5). Although XA(0) through XA(5) are not "within" the matrix XA, the following is true:

$$\begin{aligned}
XA(5) &\equiv XA(0,13) \equiv XA(1,0)\\
XA(4) &\equiv XA(0,12) \equiv XA(1,-1)\\
XA(3) &\equiv XA(0,11) \equiv XA(1,-2)\\
XA(2) &\equiv XA(0,10) \equiv XA(1,-3)\\
XA(1) &\equiv XA(0,9) \equiv XA(1,-4)\\
XA(0) &\equiv XA(0,8) \equiv XA(1,-5)
\end{aligned}$$

(2) DIMENSIØN Y5(250, J(5)), J(15)

Y5 is a 251-element array whose elements may be referred to as Y5(0), Y5(1), ..., Y5(250).

J contains the 5-element dimension vector for the matrix Y5; if J has been preset as follows:

J(0) through J(5) may be used as integer variables elsewhere;
J(5) = 4;
J(6) = 20;
J(7) = 4;
J(8) = 6;
J(9) = 3;
J(10) through J(15) may be used as integer variables elsewhere;

and if Y5(I, K, L, M) is any element in the Y5 matrix:

(a) Y5 has 4 dimensions, since J(5) = 4;

(b) the base element of Y5, Y5(1,1,1,1), is Y5(20), since J(6) = 20;

(c) the allowable range of K is $1 \leq K \leq 4$, since J(7) = 4;

(d) the allowable range of L is $1 \leq L \leq 6$, since J(8) = 6;

(e) the allowable range of M is $1 \leq M \leq 3$, since J(9) = 3.

So that the elements Y5(0) through Y5(19) are not "within" the matrix Y5, and

Y5(20) ≡ Y5(1,1,1,1);
Y5(22) ≡ Y5(1,1,1,3);
Y5(23) ≡ Y5(1,1,2,1);
Y5(35) ≡ Y5(1,1,6,1);
Y5(37) ≡ Y5(1,1,6,3);
Y5(38) ≡ Y5(1,2,1,1);
Y5(91) ≡ Y5(1,4,6,3);
Y5(92) ≡ Y5(2,1,1,1);
Y5(235) ≡ Y5(3,4,6,3);
and Y5(250) ≡ Y5(4,1,5,3).

NOTE 1: The general formula for computing a linear subscript L of the array X from a known matrix subscript $(w_1, w_2, \ldots, w_i, \ldots, w_n)$, where the upper values of w_2 through w_n are $W_2, W_3, \ldots, W_i, \ldots, W_n$, and where the linear subscript of the base element X(1,1,...,1) is B, is:

$$L = B + (w_n - 1) + \sum_{i=1}^{n-1} [(w_{n-i} - 1) \prod_{j=0}^{i-1} W_{n-j}]$$

E.g., given Y5(3,4,6,3) ≡ Y5(w_1, w_2, w_3, w_4) above,

$$L = 20 + (3-1) + [(6-1)(3)] + [(4-1)(3)(6)] + [(3-1)(3)(6)(4)] = 235$$

and Y5(3,4,6,3) ≡ Y5(235).

NOTE 2: An algorithm for computing for the matrix X a subscript $(w_1, w_2, \ldots, w_n)$ which corresponds to the linear subscript L, where the linear subscript of the base element is B, and the upper bounds on $w_2, w_3, \ldots, w_n$ are $W_2, W_3, \ldots, W_n$ is:

$$(1)\quad L - B = R$$

$$(2)\quad \frac{R}{\prod\limits_{i=2}^{n} W_i} = Q_1 + \frac{r_1}{\prod\limits_{i=2}^{n} W_i}\ ;\ Q_1 + 1 = w_1$$

$$(3)\quad \frac{r_1}{\prod\limits_{i=3}^{n} W_i} = Q_2 + \frac{r_2}{\prod\limits_{i=3}^{n} W_i}\ ;\ Q_2 + 1 = w_2$$

$$(4)\quad \frac{r_2}{\prod\limits_{i=4}^{n} W_i} = Q_3 + \frac{r_3}{\prod\limits_{i=4}^{n} W_i}\ ;\ Q_3 + 1 = w_3$$

.

.

.

$$(n)\quad \frac{r_{n-2}}{W_n} = Q_{n-1} + \frac{r_{n-1}}{W_n}\ ;\ Q_{n-1} + 1 = w_{n-1}$$

$$(n+1)\quad r_{n-1} + 1 = w_n$$

E.g., given Y5(250) and $W_2 = 4$, $W_3 = 6$, $W_n \equiv W_4 = 3$ above:

$$(1)\quad 250-20 = 230$$

$$(2)\quad 230/72 = 3 + \frac{14}{72}\ ;\ 4 = w_1$$

$$(3)\quad 14/18 = 0 + \frac{14}{18}\ ;\ 1 = w_2$$

$$(n)\quad 14/3 = 4 + \frac{2}{3}\ ;\ 5 = w_3$$

$$(n+1)\quad 3 = w_n$$

3.6.3 Automatic DIMENSIØN

There are two cases in which MAD performs automatically the necessary DIMENSIØNing:

(1) If L is a statement label name, and n is the largest subscript which appears with L in the statement label field (sec. 2.3, Statement Labels) then an automatic DIMENSIØN L(n) occurs, i.e.,

n+l locations are reserved for the L vector. (No harm is done if L is also dimensioned by the programmer).

Example: If the following appear as labels of various statements throughout the program:

```
LABELV(1)
LABELV(3)
LABELV(7)
```

and LABELV does not appear in a DIMENSIØN declaration, then 8 locations will be reserved for the LABELV vector automatically.

(2) If part or all of a linear array is preset by VECTØR VALUES (sec. 3.7), the array need not be declared by DIMENSIØN unless the size of the array automatically reserved by VECTØR VALUES is not sufficiently large.

3.7 VECTØR VALUES Declaration

Any vector or portion of a vector (or array, when using its linear subscripts) may be preset by a VECTØR VALUES declaration of the form:

$$\text{VECTØR VALUES } \mathcal{V} = c_0, c_1, \ldots, c_n$$

where $\mathcal{V}$ is any variable V or linearly subscripted variable V(m); each c_i is any constant, and all c_i are of the same mode. The elements of the vector V through V(n), or V(m) through V(m+n), are preset with the values c_0 through c_n at compilation time. The mode of the vector $\mathcal{V}$ is automatically assigned by MAD to be that of c_i, and may not be declared to be a mode other than that of c_i. A region of n+1 (or m+n+1) storage locations is reserved for $\mathcal{V}$, but this region may be enlarged by a DIMENSIØN declaration or by another VECTØR VALUES declaration.

For alphabetic constants, the effects of VECTØR VALUES may be extended to the following:

$$\text{VECTØR VALUES } \mathcal{V} = \$c_0c_1c_2\ldots c_n\$,\ \$c_{n+1}c_{n+2}c_{n+3}\ldots c_{n+p}\$,$$
$$\ldots,\ \$c_{n+p+q+1}c_{n+p+q+2}\ldots c_{n+p+q+r}\$$$

where there may appear between $'s a string of characters (see sec. 2.1.3 for allowable characters in alphabetic constants) of any length. Each c_i is considered to be a group of six characters. If the <u>last</u> c_i between $'s does not contain six characters, blanks are appended to the right to make a group of six.

It is allowed to intermingle integer constants and alphabetic constants in any VECTØR VALUES declaration, e.g.,

```
VECTØR VALUES BETA = 42, $THISbISbNUMBERb42.$, 3
```

and BETA will have reserved for it five locations which will be preset as follows:

```
BETA(0) = 42
BETA(1) = THISbI
BETA(2) = SbNUMB
BETA(3) = ERb42.
BETA(4) = 3
```

VECTØR VALUES DECLARATION

Example:

```
VECTØR VALUES ALPHA(32) = $33HOTHISbSITUATIØNb
                          CREATESbANbERRØR*$,
                          $16HbREADbMØREbDATA*$
```

The vector ALPHA, elements 32 through 41, are being preset to be used as a message at execution time. The characters between $'s define print formats (sec. 5.5.3, PRINT FØRMAT Statement).

Forty-two locations are reserved for ALPHA, ALPHA(0) through ALPHA(41). Values will be preset as follows:

ALPHA(32) = 33HOTH	ALPHA(33) = ISbSIT
ALPHA(34) = UATIØN	ALPHA(35) = bCREAT
ALPHA(36) = ESbANb	ALPHA(37) = ERRØR*
ALPHA(38) = 16HbRE	ALPHA(39) = ADbMØR
ALPHA(40) = EbDATA	ALPHA(41) = *bbbbb

In order to print these messages, the program would have to contain

```
PRINT FØRMAT ALPHA(32)
PRINT FØRMAT ALPHA(38)
```

RESTRICTIONS:

(1) Vectors which have been declared in ERASABLE storage may <u>not</u> be preset by VECTØR VALUES.

(2) Vectors which have been declared in PRØGRAM CØMMØN storage may not be preset with statement labels or function names; also, these vectors may be preset only in a 1-section program, or, in an n-section program if the PRØGRAM CØMMØN region is identical in all n sections.

3.8 Single Statement INTERNAL FUNCTIØN Definition

A description of the procedures necessary to enable the user to define more general types of functions appears in sec. 6, Function Definitions.

The single statement internal function definition is the simplest type of function definition. Since this is an internal function, it is translated as part of the main program. This single statement has the form:

$$\text{INTERNAL FUNCTIØN}\ \ F.\ (A_1, A_2, \ldots, A_n) = E$$

where F. is a function name (see sec. 2.4, Functions), the A_i's are "dummy arguments" and E is any expression which is consistent with the declared mode of F..

The "dummy arguments" are used in the expression E to indicate the correct correspondence between a variable name used in E and a position in the list of arguments. When the internal function F. is <u>used</u>, every occurrence of the "dummy variable" A_i in the expression E will be replaced by the value of the argument which occupies the position in the list of arguments of the "dummy argument" A_i.

The form of a dummy argument may be:

(1) a name of a simple variable,

(2) a name of an array variable,

(3) a name of a function.

Notice that neither constants nor subscripted variables may appear as <u>dummy</u> arguments and that arguments of a function do not appear following the function name in the <u>dummy</u> argument list.

Names used as dummy arguments must be distinct from all other names in the program. Names which appear as <u>dummy</u> arguments may not appear in any PRØGRAM CØMMØN, ERASABLE or EQUIVALENCE declaration.

The modes of the dummy arguments must be declared as for other variables if of other than normal mode.

Dummy arguments which are array names need not be dimensioned.

The alphanumeric portion of the name F. of the defined function must be distinct from all other names used in the program, and from the names of the functions already available to the translator. For a list of these names see the manual of library subroutines.

SINGLE STATEMENT INTERNAL FUNCTIØN DEFINITIONS

Ordinarily, all the names of the dummy argument list will appear in the expression E (otherwise, the unused ones need not appear as dummy arguments). Names of variables or functions which do not appear as dummy arguments, but which are defined elsewhere in the program, may also appear in the expression E. The value of such a variable or function in the expression is the current value at the time the function F. is used, either by the EXECUTE statement or as a term in an expression.

In the use of a function an argument may be any expression which agrees in mode with the corresponding dummy argument.

A single statement internal function definition may appear anywhere within a program except in another internal function (see sec. 6.4, Internal Function Definitions).

In the example:

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

which might be used in the statement (the statement label is BETA):

```
          BETA  ZQ = PØLY. (M + 1, Y, SIN.) + PØLY.(M - 1, Z, CØS.)
```

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

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

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

4 Basic Executable Statements

4.1 Substitution Statement

This statement has the form:

V = F

That is, the left side, V, of the "=" sign consists of either an individual variable or a subscripted array variable, and the right side, F, consists of any expression of the same mode. The only exceptions to this mode requirement are the cases:

(1) If the variable on the left is of integer mode then the value of a floating point expression on the right will be converted to integer mode.

(2) If the variable on the left is of floating point mode then the value of an integer expression on the right will be converted to floating point mode.

These conversions are the only automatic ones and any other disagreement of mode is not permitted.

The substitution statement is to be interpreted in the following way:

(1) compute the value of the expression on the right,

(2) convert it, if necessary and possible, to the mode of the variable on the left of the "=" sign, and

(3) give the variable on the left the value which results from steps (1) and (2).

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

Y = 1

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

Y = 1.

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

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

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

would convert to the following integers, respectively:

300000, 0, 34, - 3456810000

4.2 TRANSFER TØ Statement

This statement has the form:

TRANSFER TØ $\mathscr{S}$

Here $\mathscr{S}$ may be any expression in statement label mode and in particular may be any statement label. Execution of this statement causes the computation to continue from the statement whose label is the value of $\mathscr{S}$.

Examples: (1) TRANSFER TØ SUMX

(2) TRANSFER TØ SWTCH (K+2)

If K=4 then the value of SWTCH(K+2) is SWTCH(6).

4.3 Conditional Statements

There are two types of conditional statements.

4.3.1 Simple Conditional

This statement has the form:

WHENEVER B, Q

Here B is a Boolean expression and Q any executable statement except the following: END ØF PRØGRAM, another conditional, THRØUGH or EXECUTE. If at the time of execution of this statement, the expression B has the value 1B, the statement Q is executed. If, however, B=0B, then Q is skipped and computation continues from the next statement in sequence. The comma in this statement, as in other statements containing punctuation, must be written.

Examples:

```
WHENEVER XM.LE.1, TRANSFER TØ END
WHENEVER I.GE.N.AND.J.NE.I-1, I=0
```

4.3.2 Compound Conditional

This type of conditional has the form:

$\mathcal{S}_1$ WHENEVER B_1

$\left.\begin{array}{l}\cdots\\ \cdots\end{array}\right\} Q_1$

$\mathcal{S}_2$ ØR WHENEVER B_2

$\left.\begin{array}{l}\cdots\\ \cdots\end{array}\right\} Q_2$

...

...

$\mathcal{S}_k$ ØR WHENEVER B_k

$\left.\begin{array}{l}\cdots\\ \cdots\end{array}\right\} Q_k$

$\mathcal{S}_{k+1}$ END ØF CØNDITIØNAL

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

ØR WHENEVER 1B

or alternately the statement

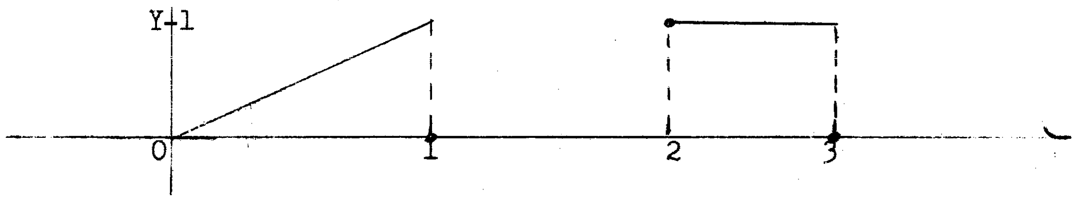
ØTHERWISE

The $\mathscr{S}_i$ are statement labels which need not be used unless desired; k may equal 1 (if no "ØR WHENEVER..." statements are used). Here $B_1, \ldots, B_k$ are Boolean expressions, and the execution of this block of statements is as follows:

Each B_i is tested in turn, starting with B_1. If B_1 has the value OB, then B_2 is tested, etc. As soon as some expression, say B_i, has the value 1B, then the statements between (but not including) $\mathscr{S}_i$ and $\mathscr{S}_{i+1}$ (i.e. Θ_i) are executed. At this point the execution of the entire block is considered ended, and computation continues from the first statement after the declaration labelled $\mathscr{S}_{k+1}$. In other words, at most one of the alternative computations is performed; e.g., that one which immediately follows the first expression B_i which has the value 1B. If none of the B_i has the value 1B, none of the computation in the scope of these statements is performed.

Among the statements of Θ_i for any i there may appear other compound statements. The maximum permissible nesting depth of compound conditional statements is 30.

Example: The evaluation of the function whose graph is



might be given by the section of the program:

```
WHENEVER X .LE. 0. .ØR. 1. .LE. X .AND. X .L. 2. .ØR. X .GE.3
  Y = 0.
ØR WHENEVER 0. .LE. X .AND. X.L.1.
  Y = X
```

```
    ØR WHENEVER 2. .LE. X .AND. X .L. 3.
      Y = 1.
    END ØF CØNDITIØNAL
```

This section of program could be rewritten in another way.

```
    WHENEVER 0..LE. X .AND. X .L. 1.
      Y = X
    ØR WHENEVER 2..LE. X .AND. X .L. 3.
      Y = 1.
    ØTHERWISE
      Y = 0.
    END ØF CØNDITIØNAL
```

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

4.4 CØNTINUE Statement

This statement has the simple form:

```
CØNTINUE
```

It serves as an entry point in the program, and causes no computation to be performed by its presence or absence. The example in sec. 4.5.2, THRØUGH Statement, illustrates the use of CØNTINUE.

4.5 THRØUGH (Iteration) Statement

This statement causes the block of statements which follows immediately afterwards to be repeatedly executed, each time varying the value of some variable, until the specified list of values for that variable is exhausted, or until some specified condition is satisfied. The THRØUGH statement may take either of the following two forms.

4.5.1 THRØUGH $\mathcal{S}$, FØR VALUES ØF V = E_1, E_2, ..., E_m

Here $\mathcal{S}$ is the statement label of the last executable statement in the block to be repeated. The block of statements following (and not including) the THRØUGH statement, up to <u>and including</u> the statement whose label is $\mathcal{S}$, will be called the "scope" of the THRØUGH Statement. Following the word ØF appears the name of the iteration variable V, which may be either an individual variable or subscripted array variable of any mode. To the right of the "=" sign may appear any number of expressions E_1, ..., E_m. The modes of the E_i bear the same relationship to the mode V as they would in the Substitution statement V = E_i (see sec. 4.1).

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

An example of this type of statement is:

```
     THRØUGH A, FØR VALUES ØF BETA = 3, 4, X5, Y(6 + I) + 3
     J = 5 * BETA + 6
     J1 = J .P. 5 - 1
A    X(BETA) = J1 * CØS.(2. * THETA)
```

4.5.2 THRØUGH $\mathcal{S}$, FØR V = E_1, E_2, B

Here $\mathcal{S}$ is a statement label which defines the "scope" exactly as in (sec. 4.5.1) above (with the exception: if $\mathcal{S}$ is the label of the THRØUGH statement itself, the iteration will proceed as described below, but the scope will be empty, and the iteration will consist only of the incrementing of V by E_2 and the testing). Following the word FØR is the name of the iteration variable V which may be an individual variable or subscripted array variable of any mode. The modes of E_1 and E_2 are subject to the rules which would apply to the substitution statements V = E_1 and V = V + E_2. B is a Boolean Expression.

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

Example: The following program segment illustrates this type of THRØUGH statement.

```
        .
        .
        .
        L = 1
        K = 1
        A = D(1,1)
        THRØUGH ST1, FØR I = 1,1, I .G. 10
        THRØUGH ST1, FØR J = 1,1, J .G. 10
        WHENEVER A .LE. D(I, J), TRANSFER TØ ST1
        K = I
        L = J
        A = D(I, J)
ST1     CØNTINUE
        .
        .
```

This program will locate the algebraically smallest element of the ten-by-ten, 100 element, matrix D, and the column and row subscripts of the first such element , if the smallest element occurs more than once.

The statements

```
L = 1     and
K = 1
```

initialize the column and row subscript indicators to 1.

The statement

```
A = D(1,1)
```

initializesthe variable A, which is to hold the "smallest element so far", to the base element of the matrix.

The statements

```
THRØUGH ST1, FØR I = 1,1, I .G. 10
THRØUGH ST1, FØR J = 1,1, J .G. 10
```

define the iteration. The first will count from the first row through the tenth row and the second will count from the first column through the tenth column. Since the second THRØUGH statement is nested inside the first it will count through the 10 columns for each of the 10 rows.

The statement

```
WHENEVER A .LE. D(I, J), TRANSFER TØ ST1
```

makes the comparison between A, the smallest so far, and the next untested element D(I, J). If A is less than or equal to D(I, J) it is still the smallest so far, so control is transferred to statement ST1.

If A is greater than D(I, J), transfer is not made; instead the statements

```
K = I
L = J
```

are executed which update the row and column subscript indicators to the subscripts of the new "smallest element so far" and the statement

```
A = D(I, J)
```

updates the "smallest element so far" variable A, to the new "smallest element so far".

The statement

```
ST1    CØNTINUE
```

results in no operation. It does, however, provide a statement inside the scope of the iteration to which the WHENEVER statement can transfer to continue the iteration but which will change nothing else.

In all cases, every reference to an expression E_i will involve its current value at the time of reference. Moreover, the variable V may have its value changed at any time during the execution of the scope. In a statement of the form of section 4.5.1, the value of V will be reset by the value of the next E_i for the next computation of the scope. In a statement of the form of section 4.5.2, the current value of V will be used for incrementing, testing, etc.

Examples:

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

```
              INTEGER J,N
              Y = 0.
              THRØUGH PØLY, FØR J = N, -1, J .L. 0
    PØLY      Y = X * Y + C(J)
```

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

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

```
    NEW   THRØUGH NEW, FØR X = XO, (CØS.(X)-X)/(SIN.(X) + 1.),
          1     .ABS.(CØS.(X) - X) .L. EPSLØN .AND.
          2     .ABS.((CØSX.(X) - X)/(SIN.(X) + 1.)) .L. EPSLØN
```

where XO is the initial guess. The digits "1" and "2" which appear at the left of the second and third lines serve to show that the second and third lines appear on second and third cards; i.e., the statement is "continued" from the first card and occupies three cards. See sec.14.1.2.2, Continuation Cards.

(iii) If the transformation of the iteration variable is to be performed within the scope of the iteration, one may use a zero increment. E.g., if J is an integer variable, and the scope of the iteration is to be performed for those multiples of 2 which are not multiples of 5 and which are less than the value of the integer K, one might write the iteration as follows:

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

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

```
    LØØK    THRØUGH LØØK, FØR I = 1, 1, C(I) .E. $,$  .ØR. I .G. K
            WHENEVER I .G. K, TRANSFER TØ NØCØMA
```

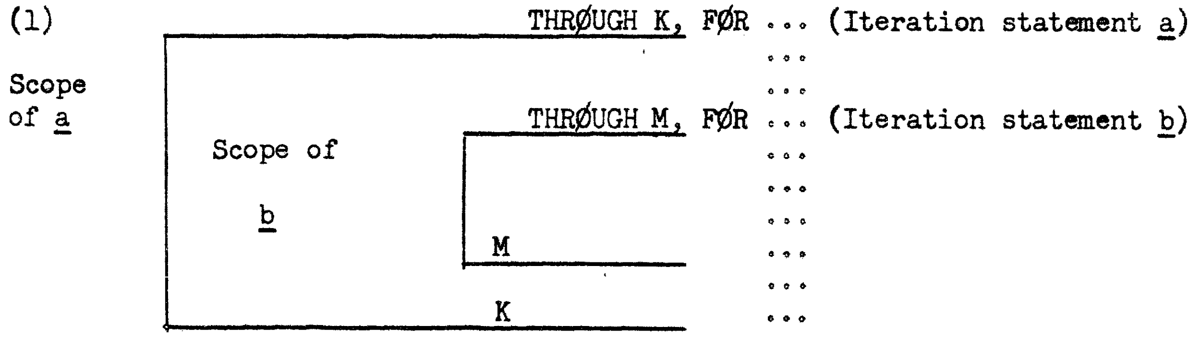
4.6 Nested THRØUGH Statements

As indicated in sec. 4.5, the "scope" of an iteration statement is the block of statements designated for repeated execution:

```
                          THRØUGH END, FØR V = E1, E2, B
           ⎧              ...
     scope ⎨              ...
           ⎩  END         ...
```

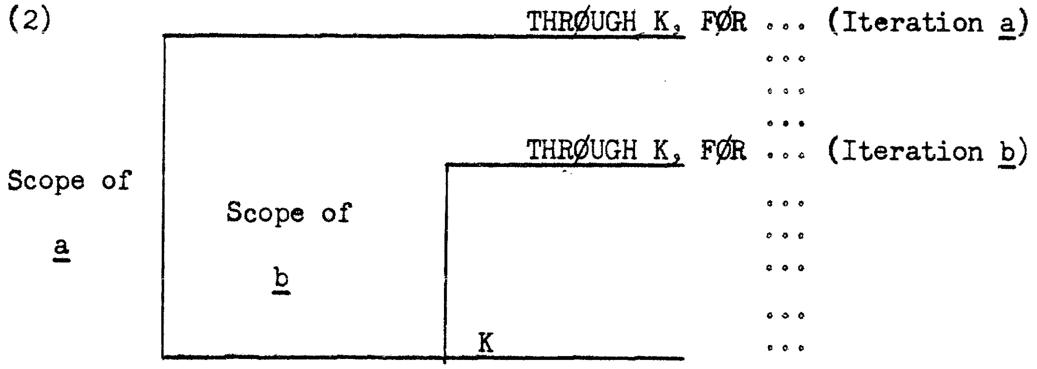
Some of the statements within the scope of an iteration may, themselves, be iteration statements. However, if iteration statement b is in the scope of iteration statement a, then the scope of b must be entirely within the scope of a. The following diagrams represent some valid configurations:

(1)



```
THRØUGH K, FØR ...
...
...
THRØUGH M, FØR ...
...
...
...
M    ...
...
K    ...
```

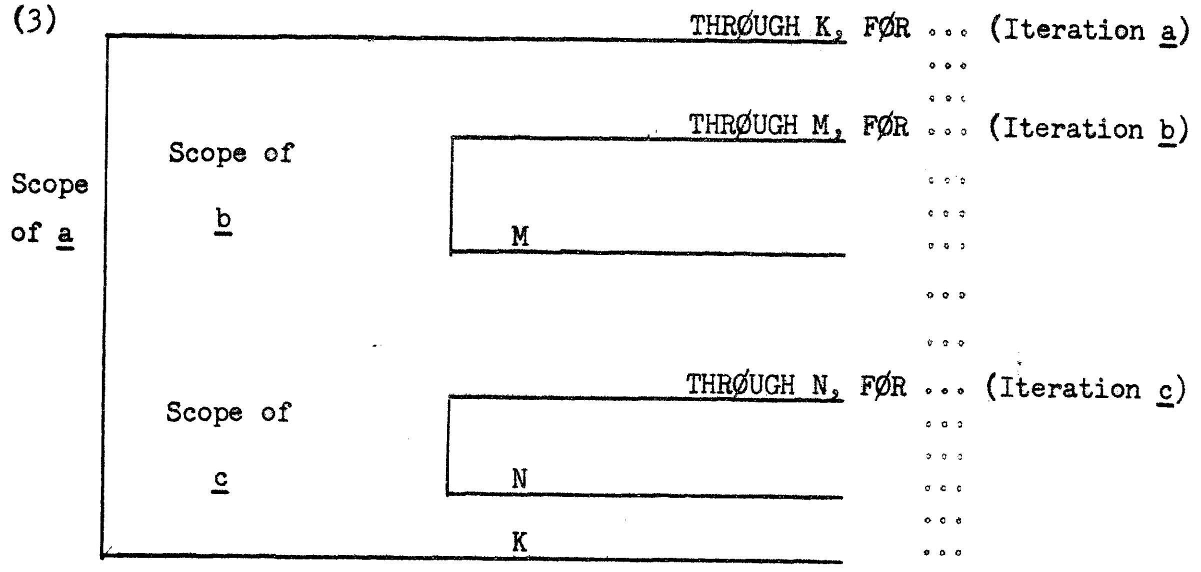
(2)



```
THRØUGH K, FØR ...
...
...
...
THRØUGH K, FØR ...
...
...
...
...
K    ...
```

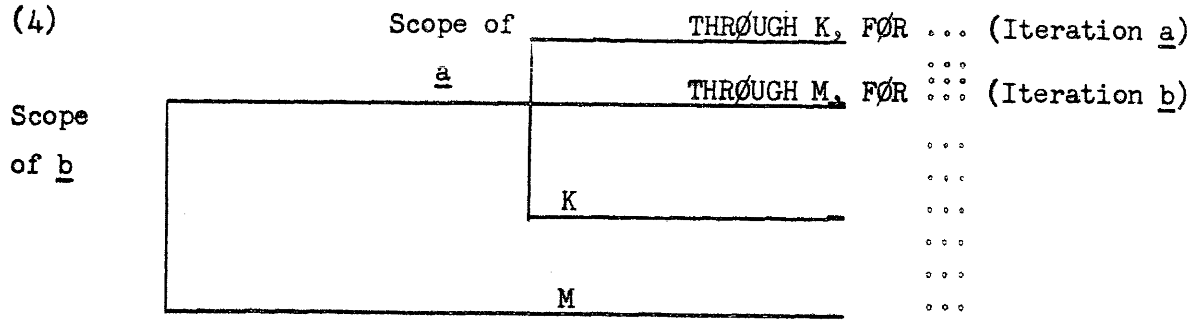
Here, although the scopes of a and b both end on statement K, iteration b is incremented and tested first. Therefore, iteration b is completed before iteration a is incremented.

(3)


THRØUGH K, FØR ... (Iteration a)
THRØUGH M, FØR ... (Iteration b)
Scope of a
Scope of b
M
THRØUGH N, FØR ... (Iteration c)
Scope of c
N
K

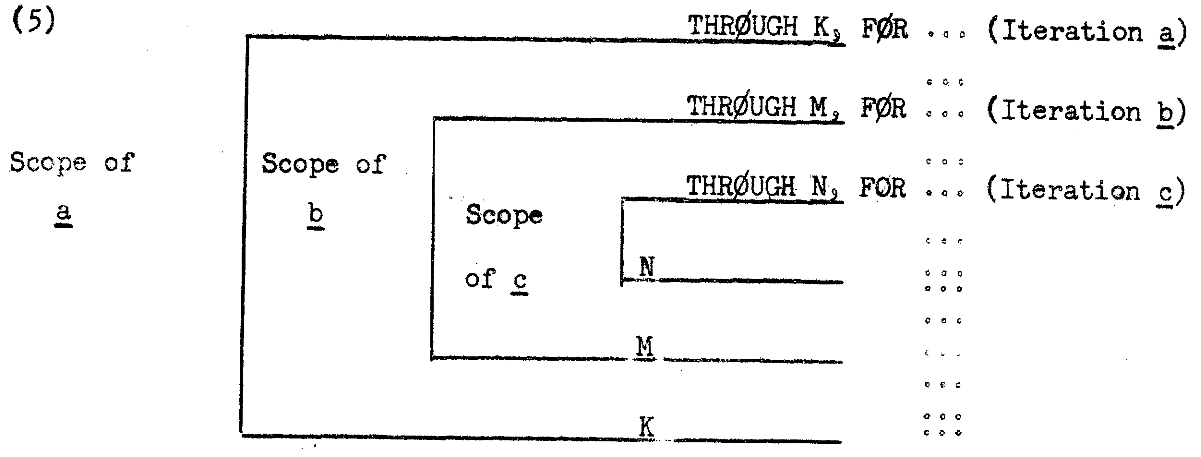
The following diagram represents an invalid configuration:

(4)


Scope of a
THRØUGH K, FØR ... (Iteration a)
THRØUGH M, FØR ... (Iteration b)
Scope of b
K
M

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

(5)


THRØUGH K, FØR ... (Iteration a)
THRØUGH M, FØR ... (Iteration b)
THRØUGH N, FØR ... (Iteration c)
Scope of a
Scope of b
Scope of c
N
M
K

NESTED THRØUGH STATEMENTS

In example (5), iteration a has a nesting depth 0, iteration b has nesting depth 1, and iteration c has nesting depth 2. In example (3), both b and c have nesting depths of 1.

There are no restrictions on jumping into or out of the statements in the scope of an iteration. If the program jumps out of the iteration and the iteration variables are not modified while outside the iteration, and if control is returned to the statement in the iteration following the "jump out" statement, then the execution will continue as if it had not been interrupted.

4.7 PAUSE NØ. Statement

This statement has the form:

PAUSE NØ. n

This statement indicates a breakpoint in the program, and it causes the computer to stop in such a way that the operator can manually start it and automatically go on to the next statement in the program. The number "n" is an octal integer containing up to 5 digits, which will be displayed on the computer console for the operator to note when the computer stops, thus indicating the point in the program at which the stop occurred. This statement is used only in very special circumstances; the majority of users will have no need for it.

4.8 EXECUTE Statement

The EXECUTE statement may have either of two forms:

(a) EXECUTE C. (A_1, A_2, ..., A_n)

where C. is the name of the function and the A_i are permissible arguments for the particular function (see sec. 2.4, Functions); and

(b) EXECUTE C.

where C. is the name of the function and the particular function does not require arguments.

This statement is provided to permit non-single valued functions to be executed.

Example: To execute the function of section 2.4.2, Non Single Valued Functions, which sorts a list, write:

```
EXECUTE  SØRT. (LIST, N)
```

It is not meaningful to write

```
EXECUTE  SIN. (X)
```

4.9 END ØF PRØGRAM Statement

This statement has the form:

END ØF PRØGRAM

This statement must be the physically last statement in the program (i.e., the last card); it serves to terminate compilation of the program. If it is executed it will terminate the computation of the object program. Execution of this statement at execution time will transfer control to the operating system in which the translated program is imbedded. An alternate way of terminating the execution of a program - i.e. returning to the operating system - is to attempt to execute an input statement when the data has been exhausted (see sec. 5.5, Basic Input - Output Statements).

5 Input and Output

In this section the various statements, specifications, and lists which may be used to read in fresh data, output results, and save and read data onto and from auxiliary storage, are described.

5.1 Simple Input-Output Statements

These statements are referred to as Simple Input-Output Statements because their use does not involve the use of Format Specifications and Input-Output Lists (sections 5.2 through 5.3).

Of these, the input statements are used in order to read any data which may appear on cards following the user's program in which these statements appear. The output statements are used in order to cause results calculated when the program is executed to be printed on paper (listed) or punched on cards.

5.1.1 READ DATA Statement

The form of this statement is

READ DATA

This statement causes information to be read from cards. The values to be read and the variable names are punched in the data cards in a sequence of fields of the form:

$$V_1 = n_1, V2 = n_2, V_3 = n_3, \ldots, V_k = n_k *$$

The $V_1, \ldots, V_k$ are the variable names and $n_1, \ldots, n_k$ are the corresponding values. Reading is continued from card to card until the terminating mark * is encountered. Fields cannot be divided between cards, so that the last character in a card not terminated by an asterisk would normally be a comma. However, as a convenience, the end of the card is treated as an implied comma and hence this final comma may be omitted. The variable names may designate simple variables or elements of linear and two-dimensional arrays. The subscripts on the array variables must be integer constants. The values may be floating point, integer, octal, Boolean, or alphabetic with the forms described in section 2.1, Constants.

SIMPLE INPUT-OUTPUT STATEMENTS

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

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

would be the same as

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

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

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

A simple integer variable may take on a value equal to a group of six or fewer Hollerith characters delimited by dollar signs, i.e., an alphabetic constant.

Longer strings of Hollerith characters may be entered as elements of arrays. Such strings are divided into six character groups for storage.

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

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

A MAD program deck having in its body a READ DATA statement would be followed by the preceding data cards. At the time the READ DATA is executed (not compiled), these three cards would be read and the values in storage of the computer would be set as follows:

X1 would become 1.2
Y1 would become -6.8
INDEX would become 4
A(4) would become 3.1
A(5) would become -10.93
A(6) would become 12.6
MATRIX(2,1) would become 25E-2
MATRIX(2,2) would become 1.8E-10
MATRIX(2,3) would become 3.14E-8
STRING(1) would become ENDbØF
STRING(2) would become PRØBLE
STRING(3) would become Mbbbbb

Note that although the components of the data cards resemble substitution statements (X1=1.2, Y1=-6.8, etc.), these cards are <u>not</u> parts of the source language program and no attempt should be made to compile them.

5.1.2 PRINT RESULTS Statement

The form of this statement is

PRINT RESULTS $\mathcal{T}$

where $\tilde{\mathcal{T}}$ is a list of the form

$$T_1, T_2, \ldots, T_n$$

and the T_i's are terms of the list. The T_i's may be

(1) simple variables;
(2) subscripted variables;
(3) block designations of the form

$$V(s_1, s_2, \ldots, s_n) \ldots V(r_1, r_2, \ldots, r_n)$$

where the $V(i_1, i_2, \ldots, i_n)$ are subscripted variables.

Note that expressions may not appear as terms T of the list but subscripts of subscripted variables in the list may be expressions (see section 2.2.4, Subscripts of Subscripted Variables).

The printed output is analogous to the data input in that the numbers printed are preceded by the appropriate variable name and an equal sign, e.g., X = -12.4, and only the initial elements of arrays are so labeled. Elements of three and higher dimensional arrays will be labeled with the equivalent linear subscript. If dummy variables (in a function definition) are included in the list the specific variables assigned to such variables during execution will not be labeled but simply preceded by ...

Example statements are:

```
PRINT RESULTS X1, Y1, Z(1), ..., Z(N+1), MTX(1,1)...MTX(M,N)
PRINT RESULTS X(1,3,4,2), ..., X(N+1,4,4,1)
```

5.1.3 PRINT CØMMENT Statement

This statement has the form:

```
PRINT CØMMENT $S$
```

Here S designates a string of no more than 120 Hollerith characters. These characters may not include dollar signs and here blanks are not ignored. The string, delimited by dollar signs as indicated, will be printed. The first character will be interpreted as a carriage control code (see section 5.2.14.2(b)).

An example statement is:

```
PRINT CØMMENT $1 JØHN PUBLIC, MATH 373 PRØBLEM 1$
```

5.2 Format Specifications

When information is read from (or punched into) a card into (or from) a computer, it is necessary to know how this information has been allocated among the available columns of the card. Similarly, whenever information is to be printed by a printer (either on-line or off-line), it is necessary to know how this information has been allocated among the columns available on the printer. A description of each allocation is called a format specification. Usually, but not always, a list of variables (see section 5.3, The Input-Output List) whose values are to be printed, punched or read, is associated with a format specification. (Occasionally, the information is contained entirely in the format specification, so the list may be empty.)

In substance, a format specification is a string of alphanumeric characters and a restricted subset of special characters, all terminated by the (very) special character "*". All characters which may be used are mentioned explicitly in the following subsections of this section.

A format specification, in order to be used, must be stored in successive elements of a vector of integer mode; therefore, it is stored in groups of six characters per element. A format specification should be stored in such a way that the first character in the specification is the left-most of the six in a vector element. If the specification is stored in such a way that the first character is <u>not</u> the left-most in an element, then those characters to the left of it <u>must</u> be blanks.

A vector may be preset with a format specification by the use of a VECTØR VALUES declaration (see section 3.7). Also, the elements of the vector may be computed or read in as data.

5.2.1 Single Line Format Specifications

A single line format specification has the form:

$$T_1, T_2, \ldots, T_n *$$

where the T_i's are terms of the format specification, the character "," separates pairs of terms and is included in the format specification, and the character "*" follows the last term of the format specification and must appear in the format specification; it serves to terminate the specification.

Any term T_i may be one of the following:

(a) a Basic Field Description (see section 5.2.11, Basic Field Descriptions - A Resumé);

(b) A multiple Basic Field Description (see section 5.2.12, Multiple Basic Field Descriptions);

(c) a Scaled Field Description (see section 5.2.13, Scaled Field Descriptions).

5.2.2 Multiple Line Format Specifications

A single format specification may be made to refer to more than one line or card at the time it is used.

A multiple line format specification has the form

$$\lambda_1 / \lambda_2 / \cdots / \lambda_{q-1} / \lambda_q \ *$$

where each λ_i may have the form of a complete single line format specification without a terminating "*" but with carriage control if required (see section 5.2.14.2, Carriage Control) or any λ_i may be blank to indicate a blank line or card for output or an ignored card for input; the character "/" is used to terminate a single line format specification λ within the multiple line format specification, and the character "*" is used to terminate the last single line format specification of the multiple line format specification.

5.2.3 Format Fields

Each format specification describes successive fields across the available columns, starting from the left. If the specification describes fewer than the total number of available columns, the remainder of the line or card will be filled in with blanks. If, at execution time, a format specification is used which describes more than the total number of available columns, an error indication will be given and the problem will be terminated.

The seven types of fields which may be described in a format specification appear in the following list:

S-Field; skip or blank information
I-Field; integer
F-Field; fixed point number
E-Field; floating point number
N-Field; octal number
C-Field; BCD characters
H-Field; Hollerith information

The terminology "fixed point number" used in connection with F-fields and "floating point number" used in connection with E-fields is rather unfortunate but is difficult to avoid. It is meant solely to provide a means of distinguishing "the form of the information which occupies an F field" from "the form of the information which occupies an E field" and throughout this section is used solely for that purpose. MAD does not handle "fixed point numbers" internally.

"Format Field" is a name for two concepts taken together.

First, there is the "Basic Field Description". This is an item which appears in a format specification.

Second, there is the "Field Information". This refers to the information and the form of the information which may appear on a card or a printed line in the field described by the corresponding Basic Field Description.

Both the Basic Field Descripton and the Field Information for each of the seven types of format fields are described in the following seven sections (section 5.2.4 through 5.2.10).

5.2.4 S-Fields

5.2.4.1 S-Field Basic Field Description

The basic field description for an S-field has the form:

Sn

where the character "S" indicates an S-field and n is a decimal integer equal to the number of columns in the field.

5.2.4.2 S-Field Information

If an S-field basic field description appears in a format specification which is used for input, any information which appears in the corresponding columns will be ignored.

If an S-field is used for output the corresponding columns will be blank.

Example: The basic field description

S31

indicates that thirty-one columns are to be skipped.

5.2.5 I-Fields

A number in an I-field has the form of an integer internally.

5.2.5.1 I-Field Basic Field Description

The basic field description for an I-field has the form

In

where the character "I" indicates an I-field and n is a decimal integer equal to the number of columns in the field.

5.2.5.2 I-Field Information

The information in an I-field may have any of the following forms:

+m

-m

m

where m is any integer satisfying

$$0 \leq m \leq 9999999999$$

If, as in the third form, no sign is punched, the integer will be assumed to be positive. Any possible sign must be counted in determining the field size.

If an I-field is used for input:

All blanks in the field are ignored;

If the entire field is blank, the value will be set equal to minus zero;

Leading zeros need not be punched;

Trailing zeros must be punched;

If the integer is negative the "-" sign must be punched.

If an I-field is used for output:

For positive integers, "+" signs are <u>not</u> printed or punched;

For negative integers, "-" signs <u>are</u> printed or punched;

If the integer internally contains fewer digits than provided for by the field size, it will print <u>right</u> justified in the field with blanks in the remainder of the field;

If the integer internally contains <u>more</u> digits than provided for by the field size, the <u>least</u> significant digits will be printed and the sign and the remainder of the most significant digits will not appear;

No decimal point, ".", is printed or punched in an I-field.

Example: The basic field description

I3

will cause three columns of an input card to be treated as a decimal integer. The same basic field description will cause information printed or punched into some particular three columns to have integer representation.

5.2.6 F-Fields

A number in an F-field has the form of a floating point number internally.

5.2.6.1 F-Field Basic Field Description

The basic field description for an F-field has the form

Fn.k

where the character "F" denotes an F-field, n is a decimal integer equal to the number of columns in the field, k is a decimal integer equal to the number of digits to the right of a missing decimal point, and the character "." (period) separates n and k. The integer k is always interpreted mod 10, e.g., k = 13 is equivalent to k = 3.

5.2.6.2 F-Field Information

If an F-field is used for input:

The information may have any one of the following forms:

$\pm m.\ell$	$\pm m.\ell E+e$	$\pm m\ell E+e$
$m.\ell$	$m.\ell E\pm e$	$m\ell E\pm e$
$\pm m\ell$	$\pm m.\ell Ee$	$\pm m\ell Ee$
$m\ell$	$m.\ell Ee$	$m\ell Ee$
	$\pm m.\ell\pm e$	$\pm m\ell\pm e$
	$m.\ell\pm e$	$m\ell\pm e$

where m is a decimal integer, ℓ is a decimal fraction, and e is a decimal integer equal to the exponent of the power of 10 by which the number $\pm m.\ell$ or $\pm m\ell$ is to be multiplied; e may contain one or two digits, no more.

The character "E", if punched, indicates that an exponent follows;

If a sign ("+" or "-") does not appear as the left-most character, the number is assumed to be positive;

The sign which follows the character "E" is the sign of the exponent, e;

If no sign character follows "E" the exponent e will be assumed to be positive;

Notice that the "E" is <u>necessary</u> only if the sign of the exponent is not punched and that the sign of the exponent is <u>not</u> necessary only if the "E" is punched and the exponent is positive;

All blanks are ignored;

If an entire F-field is blank the value will be set equal to minus zero;

If the F-field information is in one of the forms <u>with</u> a decimal point ("."), this "." in the field information will <u>override</u> the effect of the number k in the basic field description. In that case, trailing zeros in ℓ need not be punched;

If the form of the field information is one for which the decimal point is <u>not</u> punched, then trailing zeros must not be omitted;

Any number of digits may be used in the field but only eight digits of precision are retained.

In determining the field size, n, in the basic field description the count must include any possible occurrences of the sign of the number, "+" or "-", a decimal point, ".", an "E", and an exponent sign "+" or "-" as well as maximum number of digits in m, ℓ, and e, combined.

If an F-field is used for output:

The printed or punched information will have one of the following forms:

$$m.\ell$$
$$-m.\ell$$
$$m$$
$$-m$$

where m is a decimal integer and ℓ is a decimal fraction rounded to k digits;

The character "+" is not printed, the form without sign represents a positive number;

One of the two forms m or -m will occur when k=0 (the "." is not printed);

The number will be right justified in the field in all cases;

If the field size, n, is larger than required for information, blanks will be printed or punched in the remaining columns to the left;

If the field size is smaller than required, information will be output from right to left until the field is exhausted; notice that a sign, "-", and/or a decimal point, ".", which otherwise would print or punch may be lost in this event;

In determining the field size n the count must include possible occurrences of the characters "-" and "." as well as the maximum total number of digits in m and ℓ .

Example: The basic field description

F9.3

will cause the following data all to be given the <u>same</u> machine representation, namely that of 3.9962×10^2:

+39962
39962
399.62
3.9962E+2
3.9962E2
3996.2E-1
etc.

The basic field description

F9.3

will cause the machine representation of 3.9962×10^2 to be printed or punched as

399.620

right justified in the 9 columns.

5.2.7 E-Fields

A number in an E-field has the form of a floating point number internally.

5.2.7.1 E-Field Basic Field Description

The basic field description for an E-field has the form:

En.k

where the character "E" denotes an E-field, n is a decimal integer equal to the number of columns in the field, k is a decimal integer equal to the number of digits to the right of a missing decimal point and the character "." separates n and k. The integer k is always interpreted mod 10, e.g., k = 22 is equivalent to k = 2.

5.2.7.2 E-Field Information

If an E-field is used for input:

The information may have any one of the following forms:

$\pm m.\ell$	$\pm m.\ell E\pm e$	$\pm m\ell E\pm e$
$m.\ell$	$m.\ell E\pm e$	$m\ell E\pm e$
$\pm m\ell$	$\pm m.\ell Ee$	$\pm m\ell Ee$
$m\ell$	$m.\ell Ee$	$m\ell Ee$
	$\pm m.\ell\pm e$	$\pm m\ell\pm e$
	$m.\ell\pm e$	$m\ell\pm e$

where m is a decimal integer, ℓ is a decimal fraction, and e is a decimal integer equal to the exponent of the power of 10 which the number $\pm m.\ell$ or $\pm m\ell$ is to be multiplied; e may contain one or two digits, no more.

The character "E", if punched, indicates that an exponent follows;

If a sign ("+" or "-") does not appear as the left-most character, the number is assumed to be positive;

The sign which follows the character "E" is the sign of the exponent, e;

If no sign character follows "E" the exponent e will be assumed to be positive;

Notice that the "E" is <u>necessary</u> only if the sign of the exponent is not punched and that the sign of the exponent is <u>not</u> necessary only if the "E" is punched and the exponent is positive;

All blanks are ignored;

If an entire E-field is blank the value will be set equal to minus zero;

If the E-field information is in one of the forms <u>with</u> a decimal point ("."), this "." in the field information will <u>override</u> the effect of the number k in the basic field description. In that case, trailing zeros in ℓ need not be punched;

If the form of the field information is one for which the decimal point is not punched, then trailing zeros must not be omitted;

Any number of digits may be used in the field but only eight digits of precision are retained.

In determining the field size, n, in the basic field description the count must include any possible occurrences of the sign of the number, "+" or "-", a decimal point, ".", an "E", and an exponent sign "+" or "-" as well as maximum number of digits in m, ℓ and e, combined.

If an E-field is used for output:

The printed or punched information will have one of the following forms:

$m.\ell Ebe_1e_2$	$mEbe_1e_2$
$-m.\ell E-e_1e_2$	$mE-e_1e_2$
$-m.\ell Ebe_1e_2$	$-mEbe_1e_2$
$-m.\ell E-e_1e_2$	$-mE-e_1e_2$

where m is a decimal integer, ℓ is a decimal fraction rounded to k digits, "b" represents a blank and e_1 and e_2 are the two digits of the exponent of the power of 10 by which the $m.\ell$ or $-m.\ell$ is to be multiplied; e_1 is always punched or printed, even when equal to zero;

The four forms in the right hand column result when k = 0;

The number will be right justified in the field in all cases;

If the field size, n, is larger than required for the information, the remaining columns to the left will be blank;

If the field size is smaller than required, information will be output from right to left until the field is exhausted; notice that signs, "-", and/or decimal points "." which otherwise would print or punch may be lost in this event;

In determining the field size, n, the count must include possible occurrences of the characters "-" and "." and the two exponent digits as well as the maximum total number of digits in m and ℓ.

Example: The basic field description

E10.2

will cause the following data all to have the same machine representation, namely that of 24.629

24.629
24629.E-3
24629.-3
etc.

The basic field description

E10.2

will cause the machine representation of 24.629 to be printed or punched as

246.29-01

right justified in the 10 columns.

5.2.8 K-Fields

There is a one-to-one correspondence between the 36 bits internally and the bits represented by the number in a K-field. Any octal digit represents 3 binary bits as follows:

Octal	Binary
0	000
1	001
2	010
3	011
4	100
5	101
6	110
7	111

5.2.8.1 K-Field Basic Field Description

The basic field description for a K-field has the form

Kn

where the character "K" denotes a K-field and n is a decimal integer equal to the number of columns in the field.

5.2.8.2 K-Field Information

If a K-field is used for input:

The information in a K-field may have any one of the following forms:

$$+\,p$$
$$-\,p$$
$$p$$

where p is an octal integer satisfying

$$0 \leq p \leq 777777777777;$$

The octal integer p either contains or implies a value for each of the 36 binary bits in a computer word; the left-most of these 36 bits coincides with the sign bit; the sign of the number is determined by the logical "or" of the sign, if any, and the left-most of the 36 bits as given by the following list ("+" is 0, "-" is 1):

Sign in Field	Left-Most Bit	Sign of Number
+	0	+
-	1	-
b	0	+
+	1	-
-	0	-
b	1	-

All blanks in the field are ignored;

If the entire K-field is blank the value will be set equal to plus zero;

Leading zeros need not be punched;

Trailing zeros must be punched;

The field size must include possible occurrences of a sign character in the count.

If a K-field is used for output:

The information punched or printed will have the following form:

$$p$$

where p is an octal integer;

No sign is punched or printed; the sign of the number is indicated by whether the left-most of 12 octal digits is less than 4 (+) or greater than or equal to 4 (-);

The number is right justified in the field;

If the field size is larger than required for the information the remaining columns to the left are filled with blanks;

If the field size is smaller than required for the information the information is output from right to left until the field is exhausted and the most significant part does not appear.

Example: The basic field description

K3

will cause the appropriate 3 columns of data card to be interpreted as containing three <u>octal</u> digits. For example the configuration

043

will become the machine word

000 000 000 000 000 000 000 000 000 000 100 011

5.2.9 C-Fields

The Hollerith card code for each character corresponds to the internal 6-bit BCD code for the same character.

5.2.9.1 C-Field Basic Field Description

The basic field description for the C-field has the form:

Cn

where the character "C" denotes a C-field and n is a decimal integer equal to the number of columns in the field.

5.2.9.2 C-Field Information

The information in a C-field has the form:

c

where c is any string of Hollerith characters available on the computer (see section 1.1, The Hollerith Character Set). Blanks are <u>not</u> ignored.

If a C-field is used for input:

> Characters are taken from the field from left to right either until six columns have been read or until the field has been exhausted whichever occurs first;
>
> The information is left justified internally;
>
> If the field size, n, is less than six, the right-most 6-n characters are filled with blanks;
>
> If the field size n is greater than six, the right-most n-6 characters are lost;

If a C-field is used for output:

> Characters are output from left to right either until six have been transmitted or until the field has been exhausted, whichever comes first;
>
> The information is left justified in the field;
>
> If the field size, n, is greater than six, the right-most n-6 characters are filled in with blanks;
>
> If the field size, n, is less than six the right-most 6-n characters are lost.

Examples:

(1) If the format specification

```
2C3*
```

were used to read a card punched with

```
ABCDEFGHIJK
```

in columns 1 through 11, the two computer words involved would contain

```
ABCbbb
DEFbbb
```

(2) If the format specification

```
C6*
```

were used to read the card of example (1), the single computer word involved would contain

```
ABCDEF
```

(3) If the format specification

C7,C3*

were used to read the card of example (1), the two computer words involved would contain

ABCDEF

HIJbbb

5.2.10 H-Fields

The purpose of the H-Field is to permit the inclusion of strings of Hollerith characters (see section 1.1, The Hollerith Character Set) directly in the format specification itself.

5.2.10.1 H-Field Basic Field Description

The basic field description for an H-field has the form

nHc

where the character "H" denotes an H-field, c is any string of Hollerith characters and n is a decimal integer equal to the number of characters in the string c counting from the character immediately following the "H".

Although, for every other type of field, the basic field description which appears in a format specification must be separated from a following basic field description by a character ",", for the H-fields, since the number n explicitly defines the number of characters in the basic field description, the character "," after the last character in the string c is optional.

5.2.10.2 H-Field Information

If an H-field appears in a format specification which is used for input, the Hollerith characters which appear in the corresponding card columns will replace the n characters in the string c in the format specification itself.

If an H-field is used for output, the n characters of the string c will be printed or punched in the corresponding columns. The characters of the string c in the format specification are not disturbed.

Example: The basic Field Description

3HABC

or alternately,

3H

will cause, upon reading of a data card, the characters "ABC" in the field description to be replaced by the Hollerith characters from the appropriate three columns of the card.

The basic field description

3HABC

will cause printing, or punching, of the characters "ABC" on the output in the appropriate columns.

5.2.11 Basic Field Descriptions - A Resumé

The purpose of this section is merely to gather together all in one place a recapitulation of the possible seven types of basic field descriptions. They are:

Sn	(see section 5.2.4.1)
In	(see section 5.2.5.1)
Fn.k	(see section 5.2.6.1)
En.k	(see section 5.2.7.1)
Kn	(see section 5.2.8.1)
Cn	(see section 5.2.9.1)
nHc	(see section 5.2.10.1)

5.2.12 Multiple Basic Field Descriptions

If several consecutive fields can be described by the same basic field description, repetition may be avoided by using a multiple basic field description.

(a) a multiple basic field description may have the form

iD

where D is any one of the basic field description forms of section 5.2.10, Basic Field Descriptions - A Resumé, and where i is a decimal integer equal to the number of consecutive fields with form D;

(b) if M is a basic field description of the form

D

or a multiple basic field description of the form

iD

or a scaled field description S, where S may have the form of either a scaled basic field description (see section 5.2.13.1) or a scaled multiple basic field description (see section 5.2.13.2), then a multiple basic field may have the form

$$j(M_1, M_2, \ldots, M_q)$$

where j is a decimal integer equal to the number of times the group of q field descriptions M_1, M_2, ..., M_q is to be repeated and where the characters "(" and ")" delimit the group of field descriptions on the left and right respectively.

(c) a multiple basic field description may have no other form than those given by (a) and (b) above.

Notice in particular that parenthesized groups may not be nested.

Examples:

(1) the format specification

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

is a short way of writing the format specification

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

They are logically equivalent.

(2) The format specification

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

is logically equivalent to the format specification

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

5.2.13 Scaled Field Descriptions

It is possible to write an F-field or an E-field description with a scale factor which will be applied to every number to which the description is applied.

5.2.13.1 Scaled Basic Field Descriptions

A scale factor may be applied to a basic field description. Such a scaled basic field description may have any one of the following forms:

```
  sPD
+ sPD
- sPD
```

where D is either an F-field basic field description (see section 5.2.6.1) or an E-field basic field description (see section 5.2.7.1), the character "P" (for "power") denotes a scale factor and s is a decimal integer equal to the exponent of the power of 10 times which the number is to be multiplied. If the exponent is negative the sign "-" must be punched; if the exponent is positive, the sign "+" is optional.

5.2.13.2 Scaled Multiple Basic Field Descriptions

A scale factor may be applied to a restricted form of a multiple basic field description. Such a scaled multiple basic field description may have any one of the following forms

```
  sPiD
+ sPiD
- sPiD
```

where D is either an F-field basic field description (see section 5.2.6.1) or an E-field basic field description (see section 5.2.7.1), i is a decimal integer equal to the number of consecutive fields described by D (so that

iD

alone has the form of the primitive multiple basic field description of section 5.2.12 (a)), the character "P" denotes a scale factor and s is a decimal integer equal to the exponent of the power of 10 times which the number in <u>each</u> of the consecutive i fields is to be multiplied. If the exponent is negative, the sign "-" must be punched; if the exponent is positive, the sign "+" is optional.

5.2.13.3 Scaled F-Fields

When a scale factor is applied to an F-field the following formula is true:

$$EN = IREN * 10^{\text{Scale Factor}}$$

where EN stands for "External Number" and IREN stands for "Internal Representation of External Number". The scaling (multiplication by $10^{\text{Scale Factor}}$) is done before the number is converted for output and after the number has been read and converted for input. Notice that scaling actually changes the value of the number in an F-field.

Example:

Suppose the format specification

3F7.3*

is used and as a result the three numbers which follow are printed:

bb0.522b-1.567b93.671

If, instead, the format specification

-2P2F7.3,F7.3*

were used, the same numbers would print as:

bb0.005b-0.016b93.671

5.2.13.4 Scaled E-Fields

When a scale factor is applied to an E-field the basic field description may be used only for output. Although the number is modified the exponent is also modified so that the value is unchanged; only the form in which the number is printed is changed.

Example:

Suppose that the format specification

```
E18.4*
```

is used and as a result the number

```
bbbbbbb0.9321E-03
```

is printed.

If the format specification

```
2PE18.4*
```

were used instead, the same number would print as

```
bbbbbbb93.2100E-05
```

5.2.14 Format Specifications and Reading, Punching and Printing

With the following two exceptions, specifications for reading or punching cards and printing lines are identical.

5.2.14.1 Available Columns

(a) Cards

Whether reading or punching, the maximum number of card columns is 80. A format specification used for cards may describe 80 columns. If more than 80 columns are described at execution time indication of an error is given and the job is terminated. It is an often useful convention that the card columns used for data be limited to 72, leaving the remaining columns for purposes of identification. Machine configurations at some installations may impose a limit of 72 columns.

(b) Print Lines

As in the case of cards, the number of characters available for a print line depends on the equipment being used. When printing on-line 119 columns are available. Otherwise (off-line) 132 columns are available. On-line printing is to be used extremely rarely. If more than 132 columns are described at execution time, indication of an error is given and the problem is terminated.

5.2.14.2 Carriage Control

(a) The character destined for the first column of a card, when punching, has no special significance. It is regarded in the same light as the remaining 79 columns. It is punched in column 1 and the successive characters go to successive columns.

(b) The left-most character output for a print line is the "carriage control character." It controls the printer carriage "vertical motion," immediately before the line is printed, i.e., "controls the preprint skip." The code for the carriage control character is given by the following table:

If the left-most character is:	The preprint skip is:
blank	single space
0	double space
+	no space
-	triple space
1	sheet eject (skip to next page)
2	skip to next half page
4	skip to next quarter page
6	skip to next sixth page

This control character is specified in a format specification in addition to the 119 or 132 information positions.

FORMAT SPECIFICATIONS

In any format specification which is used to print a line, the left-most character output is detached to be used as carriage control and is not printed. The appearance of the line which is printed is as if the carriage control character had gone to "print position zero" which is non-existent. The second character is printed in print position 1, the third in print position 2, etc.

Note that blanks count as characters.

5.3 The Input-Output List

The purpose of the input-output list is to specify sources or destinations of the information transmitted. An input-output list, which will be referred to as $\mathcal{L}$, has the form

$$T_1, T_2, \ldots, T_n$$

where the T_i's are the elements of the list. The elements of an I/O list may be:

(1) simple variables or subscripted array variables;

(2) blocks, which may be written in one of the two forms:

(a) $V(s_1, s_2, \ldots, s_n) \ldots V(r_1, r_2, \ldots, r_n)$ where $V(s_1, s_2, \ldots, s_n)$ and $V(r_1, r_2, \ldots, r_n)$ are subscripted array variables with the s_i's and r_i's any integer valued arithmetic expressions and where the characters "..." indicate that all consecutive elements of the n-dimensional array from $V(s_1, s_2, \ldots, s_n)$ through $V(r_1, r_2, \ldots, r_n)$ <u>inclusive</u> are to be transmitted;

(b) $V(s_1, s_2, \ldots, s_n), \ldots, V(r_1, r_2, \ldots, r_n)$ where commas (",") appear between the end elements in accordance with usual mathematical notation. The appearance of the commas is the only difference between (a) and (b) and the meanings are identical.

If the block specification is used in an Input-Output list for reading or writing <u>binary tape</u>, the linear subscript corresponding to $s_1, s_2, \ldots, s_n$ may not be greater than the linear subscript corresponding to $r_1, r_2, \ldots, r_n$. The elements of the list are transmitted from left to right, but, for <u>binary tapes</u>, <u>within a block</u> the order is $V(r_1, r_2, \ldots, r_n)$ to $V(s_1, s_2, \ldots, s_n)$, i.e., right to left.

In the particular case when a list is used for <u>output</u> only and <u>not</u> input, additional elements of the list may be:

(3) any constants;

(4) any expressions.

THE INPUT-OUTPUT LIST

Example of a list which may be used either for input or output:

```
A,B,K(3),J(25*I-L),A(K+1)...A(L*2)
```

Example of a list which may be used for output only:

```
AB,D2.5,MTY(1),...,MTX(N),P(14),J(I,K)
```

5.4 Relationship Between the IØ List and the Format Specification

The "list" consists of a set of names of variables to or from which information is to flow. Except for Hollerith strings imbedded in the format specification itself (see section 5.2.10) and the S fields, each field in the specification refers to one item on the list. For this purpose, a regional entry on the list, such as A(6)...A(8), counts as several names of variables (in this case, the three variables A(6),A(7), and A(8)). During the transmission of information, the input or output subroutine scans both the list and the specification simultaneously, correlating corresponding entries, and associating a field size, a type of conversion, etc., to each variable. If a Hollerith string is encountered in the specification, it is immediately transmitted, and it is not associated with any item on the "list".

For example, if the list consisted of:

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

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

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

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

```
456.010bb-16.1251E+02bbbbbbbbbbbbbM=50bbbbbbbbbJ=17
```

The same list would look as follows with the format specification

```
1H1, 2F11.3, S16, 2HM=I3, S9, 2HJ=I3*:
456.010bb-1612.510bbbbbbbbbbbbbbbbM=50bbbbbbbbbJ=17
```

As stated above, a specification may not account for more than 80 columns on a card. It may happen, however, that a list calls for more information than can appear on a single card. Or perhaps only a certain part of each card is to be read. The determining factor in every case is whether or not the entire list has been accounted for. After each card is read according to the format specification, the list is consulted;

if it is not yet satisfied, another card is read, and so on. It is important to realize that the specification is not necessarily scanned from the beginning when a new card is read. In fact, the specification scanner moves to the left from the end of the specification (the *) until it hits a left parenthesis not in an H field. (If there is no left parenthesis, it will move to the beginning of the specification). It then examines the characters just to the left of this left parenthesis to see if they are a multiplicity indication (see section 5.2.2). The information from this left parenthesis (together with the multiplicity, if any) to the end of the specification now becomes the format specification until the list is satisfied. A similar statement may be made for printed or punched output.

Thus, if the specification which follows

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

is used with an I/Ø list which contains more than 11 elements (say 19 or 27), then the first line printed (or read) would have three fixed point numbers. Subsequent lines would all be printed (or read) according to the specification

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

until the last element of the I/Ø list were transmitted.

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

```
I6/(6E10.5)*.
```

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

If a specification contains a Hollerith string of the form $nHa_1a_2\ldots a_n$, certain conventions are observed:

(1) If the list is satisfied, but the next field specification is a Hollerith string, the string is transmitted anyway;

(2) On input, i.e., reading from cards, when a Hollerith string is encountered in the specification, the information in the corresponding columns of the input card will be brought in and will replace the Hollerith string itself within the format specification. This can then be used as a specification for output. For example, this is useful for labeling a set of data and causing the label to appear on the output along with a date, etc.

Thus, a card punched as follows:

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

might be read in with a format specification

```
72H                        (72 blank spaces)                *.
```

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

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

See the example in section 3.7, VECTØR VALUES Declaration.

5.4.1 Input and Output of Boolean Values

For the purposes of input and output the Boolean values 0B and 1B are considered as integers and an I-field should be used to transmit them. Externally these values have the form of the digits 0 and 1 respectively.

5.4.2 Input and Output of Statement Label Values

Output of statement label constants may be accomplished by means of a C-field or an H-field. Input and outputof variable values may be accomplished by means of K-fields but involves concepts beyond the scope of this manual. This will be of no use to most users and should not be attempted without a deeper investigation of MAD.

5.4.3 Input and Output of Function Name Values

Output of function name constants may be accomplished by means of a C-field or an H-field. Input and output of variable values may be accomplished by means of K-fields but involves concepts beyond the scope of this manual. This will be of no use to most users and should not be attempted without a deeper investigation of MAD.

5.5 Basic Input-Output Statements

Of these, the input statements are used in order to read any data which may appear on cards following the user's program in which these statements appear. The output statements are used in order to cause results calculated when the object program is executed to be printed on paper (listed) or punched on cards.

Throughout the following sections the following notation will be used:

F: F is the name of the first element of a format specification vector which must be of integer mode. F may be written as

(a) an array name,

(b) a subscripted variable,

(c) a simple variable in case the format specification is less than or equal to six characters.

See section 5.2, Format Specifications.

$\mathcal{L}$: $\mathcal{L}$ is an input-output list, see section 5.3, The Input-Output List.

5.5.1 READ FØRMAT Statement

This statement has the form

READ FØRMAT F, $\mathcal{L}$

This statement causes data in the users data deck to be read into list $\mathcal{L}$ in accordance with the format at F. If the purpose is merely to read characters into a format specification comprised entirely of an H-field, in which case the list will be empty, the statement may be written

READ FØRMAT F

5.5.2 PUNCH FØRMAT Statement

This statement has the form

PUNCH FØRMAT F, $\mathcal{L}$

This statement causes data in the list $\mathcal{L}$ to be punched onto cards in accordance with the format at F. If the purpose is to punch using a format specification comprised entirely of an H-field, in which case the list will be empty, the statement may be written

PUNCH FØRMAT F

5.5.3 PRINT FØRMAT Statement

This statement has the form

PRINT FØRMAT F, $\mathcal{L}$

This statement causes data in the list $\mathcal{L}$ to be printed in accordance with the format at F. If the purpose is to print a comment or heading line using a format specification comprised entire of an H-field, in which case the list will be empty, the statement may be written

PRINT FØRMAT F

5.6 PRINT ØN LINE FØRMAT Statement

This statement has the form

PRINT ØN LINE FØRMAT F,$\mathcal{L}$

The list $\mathcal{L}$ is printed on-line in accordance with the format at F. If the purpose is to print a comment using a format specification comprised entirely of an H-field, in which case the list will be empty, the statement may be written

PRINT ØN LINE FØRMAT F

After $\mathcal{L}$ has been printed, a skip of one-sixth page is produced to allow the operator to read the comment.

This statement provides the facility for printing comments to the computer operator on the on-line printer at execution time. The usage is quite restricted; and the statement will not be used by most users.

5.7 Auxiliary Tape Storage Statements

In this section, the following notation will be used:

N: N is an integer valued expression. The value will be used as a tape number and must be in the range [1, 10]. To determine which numbers in particular may be used, see the manual describing the operating system in which MAD is imbedded.

5.7.1 READ BCD TAPE Statement

This statement has the form:

READ BCD TAPE N, F, $\mathcal{L}$

This causes the list $\mathcal{L}$ to be read from tape N in BCD mode in accordance with the format at F.

5.7.2 WRITE BCD TAPE Statement

This statement has the form:

WRITE BCD TAPE N, F, $\mathcal{L}$

This causes the list $\mathcal{L}$ to be written onto tape N in BCD mode in accordance with the format at F.

5.7.3 READ BINARY TAPE Statement

This statement has the form:

READ BINARY TAPE N, $\mathcal{L}$

This causes consecutive words from the next record on tape N to be read into the list $\mathcal{L}$ in binary mode. Transmission stops when the list is exhausted or when the end of record is reached, whichever occurs first.

5.7.4 WRITE BINARY TAPE Statement

This statement has the form:

WRITE BINARY TAPE N, $\mathcal{L}$

This causes the list $\mathcal{L}$ to be transmitted as consecutive words comprising the next (1) record on tape N, in binary mode.

5.7.5 REWIND TAPE Statement

This statement has the form:

REWIND TAPE N

This causes tape N to be rewound, i.e., positioned at load point.

5.7.6 END ØF FILE TAPE Statement

This statement has the form:

END ØF FILE TAPE N

This causes an end-of-file mark to be written on tape N.

5.7.7 BACKSPACE RECØRD ØF TAPE Statement

This statement has the form:

BACKSPACE RECØRD ØF TAPE N

This causes tape N to be moved backwards to the beginning of the preceding record.

5.7.8 BACKSPACE FILE Statement

This statement has two forms:

(1) BACKSPACE FILE ØF TAPE N

This causes tape N to be moved backwards until an end-of-file mark, the load point gap or the load point is encountered. If it is an end-of-file mark which is encountered, this statement will cause the tape to be positioned immediately preceding the end-of-file mark.

(2) BACKSPACE FILE ØF TAPE N, IF LØAD PØINT TRANSFER TØ S

The action for this statement is exactly as the preceding one with the additional facility that, if the tape is already at load point, the program transfers to the next executable statement given by the statement label expression S.

5.7.9 Action of End-of-File and End-of-Tape

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

If an end-of-file is encountered while executing a READ FØRMAT, READ BCD TAPE, or READ BINARY TAPE statement the subroutine SYSTEM. (a subroutine which returns control to the system in which the translated program is imbedded) is automatically entered. This action can be changed by executing the statement

EXECUTE SETEØF. (S)

The subroutine SETEØF. sets the read routines to transfer to the executable statement labeled S when an end-of-file is encountered. If the statement EXECUTE SETEØF. (O) is executed the read routines will be reset to enter SYSTEM. when an end-of-file is encountered.

If an end-of-tape is encountered while executing a WRITE BCD TAPE or WRITE BINARY TAPE statement, no special action is taken. If writing continues, the tape may run off the reel. This can be changed by the executing the statement

EXECUTE SETETT. (S)

The subroutine SETETT. sets the write tape routines to transfer to the executable statement labeled S when an end-of-tape is encountered. Executing EXECUTE SETETT. (O) will reset the write tape routines to the normal situation.

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

6 Function Definitions

In this section the executable statements and declarations necessary to define function subprograms are described; for use of defined functions see secs. 2.4.1, Single Valued Functions, 2.4.2, Non-single Valued Functions, and 2.4.5, Arguments of Functions.

A "function definition" is a sequence of executable statements from section 4 and declarations from section 3 qualified by and delimited by the declarations and executable statements of sec. 6.3, Function Definition Statements, below. Further restrictions are imposed upon the sequence of statements by the considerations given in sec. 6.4, Internal Function Definitions, and sec. 6.5, External Function Definitions.

As has already been alluded to, there are two main types of programmer defined functions: the internal function and the external function. Either of those types may include single valued functions or non-single valued functions (non-single valued functions are sometimes referred to as "procedures").

A "recursive function" of either type may also be defined. For the purposes of this manual a recursive function is a function whose definition calls for the function being defined or calls for a function which ultimately calls for the function being defined.

The general structure of the definition of a recursive function is the same as for any other function but it will include some of the statements described in section 7, List Manipulation Statements.

When defining a recursive function, it is necessary to keep in mind that names are used as function arguments and not values (see section 15, Examples, for further considerations with regard to recursive functions).

The name of a defined function must be distinct from the names of functions already available to the translator. For a list of these names see the manual of library subroutines available.

Each function definition (except Single Statement INTERNAL FUNCTIØN Definitions, sec. 3.8) may define any number of functions and/or any number of procedures.

6.1 Dummy Arguments

Some functions may be defined which require no arguments. Often, however, it is convenient to be able to specify which values are to be used during the evaluation of a function, i.e., to specify arguments. A discussion of the form of the arguments when a function is used appears in secs. 2.4.1, Single Valued Functions, 2.4.2, Non-Single Valued Functions, and 2.4.5, Arguments of Functions.

We are concerned here with the form of the arguments when the function is defined; these are called "dummy arguments".

A dummy argument is used in the statements of the function definition to indicate a correspondence between the variable name in the statement and a position in the list of arguments. When the function is used every occurrence of the "dummy variable", say A_i, in the statements of the definition will be replaced by the value of the expression in the position of A_i in the list of dummy arguments.

The form of a dummy argument may be:

(1) a name of a simple variable,

(2) a name of an array variable,

(3) a name of a function.

Notice that neither constants nor subscripted variables may appear as dummy arguments and that arguments of a function do not appear following the function name in the dummy argument list.

Names which appear as dummy arguments in any function declaration may not appear in a PRØGRAM CØMMØN, ERASABLE or EQUIVALENCE declaration.

The modes of dummy arguments must be declared as for other variables if of other than normal mode.

Dummy arguments which are array names need not be dimensioned.

Example: The list

```
(SIMPLE,FCN.,ARRAY)
```

is a valid dummy argument list. The following list is not:

```
(FCN.(X,Z+2.,VAR1), ARRAY(1,9))
```

6.2 The Use of a Defined Function

In the use of a function, either in an expression or in an EXECUTE statement, the arguments may be constants, variables or expressions. However, if one of the dummy arguments in the definition appears to the left of the character "=" in a substitution statement among the defining statements, then it is not meaningful to use a constant or an expression in the position of that argument in the call for the function. The arguments in the call for a function cannot be checked by the translator for correspondence in mode and number to the arguments in the definition. It is left to the user to see that they agree.

6.3 Function Definition Statements

6.3.1 INTERNAL FUNCTIØN Declaration

This declaration indicates that an internal function is to be defined by the statements which follow. This declaration has the form:

INTERNAL FUNCTIØN (A_1, A_2, ..., A_n)

where the A_i are the dummy arguments for the function. Note that no function name appears in the declaration. This statement must necessarily appear as the <u>first</u> statement of an internal function definition. Section 6.4, Internal Function Definitions, contains further discussion regarding the definition of internal functions.

6.3.2 EXTERNAL FUNCTIØN Declaration

This declaration indicates that an external function is to be defined by the statements which follow. This declaration has the form:

EXTERNAL FUNCTIØN (A_1, A_2, ..., A_n)

where the A_i are the dummy arguments for the function. Note that no function name appears in the declaration. This statement must necessarily appear as the <u>first</u> statement of an external function definition. Section 6.5, External Function Definitions, contains further discussion regarding the definition of external functions.

6.3.3 ENTRY TØ Declaration

This declaration has the form:

ENTRY TØ F.

where F. is a name of a function which is to be defined by this function definition. This declaration marks the entry to the function F.. Execution of the function F. begins with the first executable statement following the ENTRY TØ F. declaration.

In a single program a given function name may appear in only one ENTRY TØ declaration; also a given function name may appear in only one ENTRY TØ declaration within a given function definition. However, a single function definition may contain many ENTRY TØ declarations each with a distinct name F., to designate entry points to several function names within the function definition.

6.3.4 FUNCTIØN RETURN Statement

This is an executable statement which terminates the execution of the function and returns to the place in the program where the function name was used in such a way that computation may procede using the result or results of the function.

There are two forms of the FUNCTIØN RETURN statement:

(1) FUNCTIØN RETURN E

where E is any expression, the mode of which is consistent with the mode of the function being evaluated. At the time of return the value of the function will be made the value of E. The mode of the expression E is not checked by the translator. If a function name is to be used in any expression, i.e., if it is to be a "single valued function" (see sec. 2.4.1, Single Valued Functions), then the FUNCTIØN RETURN statement which terminates the execution of the function must have this form.

Example: The statement

```
FUNCTIØN RETURN  BETA/K5-4.*D
```

causes the value of the function in which it appears to be set equal to the expression

```
BETA/K5-4.*D
```

(2) FUNCTIØN RETURN

This form of the FUNCTIØN RETURN statement, which does not include an expression, is used to terminate the execution of any function which is not a "single valued function" in the sense of this manual, i.e., any function which is not used in an expression.

It should be noted that a function may have a single result, such as setting a single variable equal to a calculated value, and still not be a "single valued function" in the sense of this manual.

On the other hand, a function may be defined which yields several results, such as setting several variables equal to several calculated values, but which may nevertheless be a single valued function if it terminates execution by a FUNCTIØN RETURN statement of the form under (1) above.

A single FUNCTIØN RETURN statement may be used by many functions within the function definition.

Examples:

(1) The internal function definition

```
INTERNAL FUNCTIØN  (X,Y,Z)
ENTRY TØ  FCN.
X = SIN.(Y) .P. 2 + SIN.(Z) .P. 2
FUNCTIØN RETURN
END ØF FUNCTIØN
```

has a single result; the value of X is set equal to the value of the expression

$$\sin^2(Y) + \sin^2(Z)$$

Nevertheless, this is not a "single valued function" since the FUNCTIØN RETURN statement used contains no expression.

(2) The internal function definition

```
INTERNAL FUNCTIØN  (X,Y,Z)
ENTRY TØ  FCN2.
X = SIN.(Y)
Y = CØS.(Z)
Z = ATAN.(X)
FUNCTIØN RETURN  X + Y + Z
END ØF FUNCTIØN
```

is a "single valued function" whose value is equal to the value of the expression

$$X + Y + Z$$

even though, in addition, new values for all of X, Y and Z are calculated.

6.3.5 ERRØR RETURN Statement

This statement has the form:

ERRØR RETURN

It is an executable statement which may be used in a function definition if desired.

If the right-most argument in the call of the function in the definition of which the ERRØR RETURN statement appears is an expression of statement label mode (see sec. 2.10, Statement Label Expressions), and if the ERRØR RETURN statement is executed, then execution of the function is terminated and control is transferred to the statement the label of which is equal to the value of the aforementioned statement label expression.

If the ERRØR RETURN is used in a function definition and a call for the function is made such that the right-most argument is <u>not</u> of statement label mode, and if the ERRØR RETURN statement is executed, control will be returned to the system and the execution of the program will be terminated. An error indication will be printed.

6.3.6 END ØF FUNCTIØN Declaration

This declaration has the form:

END ØF FUNCTIØN

It delimits the extent of any function definition and hence must appear as the physically last statement in any function definition (with the exception of the Single Statement **INTERNAL FUNCTION** Definition of section 3.8). It is not an executable statement and the execution of the function may not be terminated by attempting to execute it. If it is executed an error indication will be given and the problem will be terminated at <u>execution</u> time.

6.4 Internal Function Definitions

An internal function definition is a function definition which is to be translated as part of the main program.

Names used as dummy arguments must be distinct from all other names in the program.

The name of the defined function must be distinct from all other names used in the program.

Names of variables or functions which do not appear as dummy arguments but which are defined elsewhere in the program may also appear in the internal function definition statements. The value of such a variable or function is the current value at the time the internal function is used.

Internal function definitions of all kinds (including single statement definitions), may occur anywhere in the program, except within another internal function definition. Internal function definitions may occur within external function definitions.

Example: The following is an internal function definition of the function whose name is NAMEF.:

```
INTERNAL FUNCTIØN  (P,D,F.)
ENTRY TØ NAMEF.
TEMP = P
P = F.(D)
D = F.(TEMP)
FUNCTIØN RETURN  0
END ØF FUNCTIØN
```

This is a single valued function whose value is a constant integer zero. In addition the function sets P equal to F.(D) and D = F.(P) where F. is any defined function name (other than NAMEF.) used in a call for NAMEF..

6.5 External Function Definitions

An external function definition is a function definition which is to be translated completely independently from the main program in which it is to be used. The external function definition as a whole appears completely outside any other program. Because of this an external function definition is an entirely separate, complete program. Thus it must contain its own DIMENSIØN declarations, mode declarations, PRØGRAM CØMMØN declarations, ERASABLE declarations and any others which it itself requires. However, it is terminated by an END ØF FUNCTIØN statement and not by an END ØF PRØGRAM statement as are other programs.

Declarations required on the dummy arguments appear within the external function definition.

Since an external function definition is a complete program there is no conflict between names used within it and names used within a main program using it or names used within another external function used by the same main program.

Example: The following is an example of an external function whose value is $1/x$ if $0 < x \leq 1$ and $1/x^2$ if $x > 1$. If $x \leq 0$, one obtains an error return (see section 6.3.5, ERRØR RETURN statement).

```
A  EXTERNAL FUNCTIØN (X)
J  ENTRY TØ INVSF.
G  WHENEVER X.G.O. .AND. X .LE. 1.
C      FUNCTIØN RETURN X .P. -1
H  ØR WHENEVER X .G. 1.
D      FUNCTIØN RETURN X .P. -2
I  ØTHERWISE
E      ERRØR RETURN
K  END ØF CØNDITIØNAL
B  END ØF FUNCTIØN
```

Here the statements are all labelled only for reference in what follows.

This definition program defines a single-valued function of X, called INVSF.. Since no mode declaration is given it is assumed by the translator that X is floating point even though a different normal code were declared in the main program. The value of INVSF.(X) is computed by the use of a compound conditional. If $0 < x \leq 1$, (statement G) then statement C is executed, causing a return to the calling program with the value $\frac{1}{x}$. If the condition $0 < x \leq 1$ is not true, then the condition $x > 1$ is tested (statement H). If $x > 1$, statement D is executed. Finally, if neither of the conditions $0 < x \leq 1$ or $x > 1$ is true, then statement I finds that $x \leq 0$ and statement E (the error return) is executed. If

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

is part of a program and the error return statement is executed during the evaluation of INVSF.(Y) (i.e., $Y \geq 0$), then control is returned to the system in which the translated program is imbedded, with an error flag set. If

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

is part of a program and $Y \leq 0$, then when the ERRØR RETURN statement is executed control transfers to statement ER (then goes on to L), instead of finishing the execution of statement F (and then going to S). Note that the END ØF FUNCTIØN statement will never be executed, but must be present in the definition.

7 List Manipulation Statements

These executable statements facilitate the writing of recursive internal and external functions (see sec. 6, Function Definitions). They cause the designation and use of a vector for the temporary storage of data and function returns.

7.1 SET LIST TØ Statement

This statement has the form:

SET LIST TØ V

where V is the name of an array element, i.e., either an array variable name or a subscripted variable. The name V specifies the initial element in a vector to be used as temporary storage. Consecutive elements will be used as required by executions of the statements which follow in this section The number of elements in the list-vector V is determined by the amount of data the user specifies to be stored there.

In the following the terminology "SAVE statement" will refer to any SAVE DATA or SAVE RETURN statement (see secs. 7.2 and 7.3). The terminology "RESTØRE statement" will refer to any RESTØRE DATA or RESTØRE RETURN statement (see secs. 7.4 and 7.5).

A SET LIST TØ statement must be executed before any SAVE or RESTØRE statement is executed.

A SET LIST TØ statement defines the list-vector to be used by all SAVE and RESTØRE statements until another SET LIST TØ statement is executed.
Example: The statement

SET LIST TØ VECTØR (12)

results in the assignment of the first element of the current list-vector to the 12th element (where 0 is the 0th element), of the vector VECTØR.

7.2 SAVE DATA Statement

This statement has the form:

$$\text{SAVE DATA } \mathcal{L}$$

where $\mathcal{L}$ has the most general form of an input-output list (see sec. 5.3, The Input-Output List). This statement causes the current values of the elements of the list $\mathcal{L}$ to be stored, in the order of their appearance from left to right in the list, in consecutive elements of the current list-vector as specified by the most recently executed SET LIST TØ statement, starting with the first <u>available</u> element of the list-vector. The element which <u>is</u> the first available element of the list-vector is <u>either</u>

(1) the initial element of the list-vector V as specified by the most recent SET LIST TØ statement if no other SAVE statement has been executed since the SET LIST TØ statement,

<u>or</u>

(2) the element one past the last element used by the most recently executed SAVE statement.

Examples:

If the following segment of program is executed:

```
    .
    .
    .
SET LIST TØ LIST3
    .
    .
    .
SAVE DATA MAX, MIN
    .
    .  α
    .
SAVE DATA  LIST1(0), ..., LIST1(4)
    .
    .  β
    .
```

the statement

```
SET LIST TØ LIST3
```

assigns the first element of the current list-vector to LIST3(0).
The statement

```
SAVE DATA MAX,MIN
```

stores the values of MAX and MIN as if the following two substitution statements were executed:

```
LIST3(0) = MAX
LIST3(1) = MIN
```

During the execution of the statements α the next available element in the current list-vector is LIST3(2).
The statement

```
SAVE DATA LIST1(0), ..., LIST1(4)
```

stores values of the elements of the block as if the following substitution statements were executed:

```
LIST3(2) = LIST1(0)
LIST3(3) = LIST1(1)
LIST3(4) = LIST1(2)
LIST3(5) = LIST1(3)
LIST3(6) = LIST1(4)
```

During the execution of the statements β the next available element in the current list-vector is LIST3(7).

7.3 SAVE RETURN Statement

This statement has the form:

SAVE RETURN

This statement is meaningful only when it appears in a function definition. It causes the location in the program which called the function to which the function is to return upon completion to be stored as the next available element in the current list-vector. The determination of the next available element is made as under SAVE DATA Statement above.

7.4 RESTØRE DATA Statement

This statement has the form

RESTØRE DATA $\mathcal{L}$

where $\mathcal{L}$ has the restricted form of an input list (see sec. 5.3, The Input-Output List). If a total of n names N_1, N_2, ..., N_n (n data elements) are designated by the list in order from left to right, and if elements through the kth of the list-vector have been used by SAVE statements, then the variable N_i is replaced by the value of the k-i+1'st element of the list-vector, i.e., N_1 is replaced by the kth element, N_2 by the k-1'st element, ..., N_n by the k-n+1'st element. Notice that the form of the list implies that n names may be designated without the appearance of n terms in the list; a term which is a block designation must be counted to represent as many names as elements in the block.

After this operation is completed, these n elements of the list-vector are made available in the list-vector. This means that the value of the first name in a SAVE statement list executed immediately after a RESTØRE DATA statement will be saved in the k-n+1'st element of the list-vector.

Example:

Suppose that the following segment of program were executed after, in time, the execution of the program segment in the example of section 7.2, SAVE DATA Statement;

```
.
.
.
RESTØRE DATA  LIST1(4), ..., LIST1(0)
.
.      λ
.
RESTØRE DATA  MIN,MAX
.
.      μ
.
```

the first RESTØRE DATA statement has an effect exactly the reverse of the second SAVE DATA statement in the earlier example and the second RESTØRE

RESTØRE DATA STATEMENT

DATA statement has an effect exactly the reverse of the first SAVE DATA statement. During the execution of the statements λ, the next available element in the current list-vector is LIST3(2). During the execution of the statements μ the next available element in the current list-vector is LIST3(0).

7.5 RESTØRE RETURN Statement

This statement has the form

RESTØRE RETURN

This statement is meaningful only when it appears in a function definition. It causes the current last element in the current list-vector to be used as the location to which control is returned upon completion of the function program, i.e., when the next FUNCTIØN RETURN statement is executed. The last element of the list-vector is then made available for use by the next SAVE statement.

12 Advanced Features

This section is devoted to a discussion of features which are extensions of the basic features described in preceding sections. An understanding of the preceding sections is presumed here.

12.1 Subscription Redefinition

In order to conserve storage it is sometimes desirable to store less than the total number of elements of an array. For example, it is often desirable to store only half of a symmetric matrix or only the non-zero elements of a sparse matrix.

This can be done using the features of MAD already described if the user is content to use linear subscripting exclusively for the matrix in question.

Since the standard array subscripting process used by MAD presumes the presence of an entire array stored in the order described in section 2.2.2, Arrays, however, it is not possible to use the array subscripting notation for an abbreviated array with the standard array subscripting procedure.

A special subscription routine may be written by the user as a single valued internal function, external function or assembly language subroutine to permit the user to use the array subscripting notation for an abbreviated array.

12.1.1 Requirements Imposed on the Subscription Routine

(1) The arguments for the routine must have the following form:

$$(\text{NAME}, S_1, S_2, \ldots, S_n)$$

where NAME is the name of the abbreviated array and the S_i's have the values of the n subscripts resulting from the conventional notation, i.e. the use of the subscripted variable

$$\text{NAME}(S_1, S_2, \ldots, S_n)$$

(2) The routine must be a single valued function whose value is equal to the linear subscript of the array NAME which corresponds to the array subscript $(S_1, S_2, \ldots, S_n)$.

SUBSCRIPTION REDEFINITION

A subscript routine written in assembly language may use the fact that the location of the dimension vector, ADIM(0), appears in the decrement part of the calling sequence parameter containing in the address part the first location of the array, A(0) (see sec. 14.3, Structure of Subroutines). Example: Suppose that the matrix A of order n is an upper triangular matrix, with all elements below the main diagonal equal to zero. Instead of storing the entire matrix, it is desired to store only the elements above and on the main diagonal, i.e., only the elements $a_{i,j}$ for which $i \leq j$ where i is the row index and j is the column index. The elements are to be stored by row, i.e. so that:

(1) if the element $a_{i,j}$ where $j < n$, is in matrix location A(k) then the element $a_{i,j+1}$ is in matrix location A(k+1); and

(2) if the element $a_{i,n}$ is in matrix location $A(\ell)$, then the element in matrix location $A(\ell+1)$ is the element $a_{i+1,i+1}$.

Then, assume further that the matrix A is stored in the MAD array A with the base element A(1,1) equivalent to the element A(B), and the array element A(B-1) is set equal to zero whenever computation is performed using the matrix A.

If SUBF. is the name of a function written to calculate the correct linear subscript given the conventional matrix subscripts i and j then the desired value of the function is given by

$$\text{SUBF.}(A,i,j) = \begin{cases} B-1, \text{ if } i > j \\ \dfrac{(2(n+1)-i)(i-1)}{2} + j-i+B, \text{ if } i \leq j \end{cases}$$

where i and j may be, as usual, any integer expressions, but whose values should satisfy $1 \leq i,\ j \leq n$.

The value of SUBF. (A,i,j), then, is the linear subscript of the element A(i,j) if A(i,j) is one of the elements which is actually stored, and is the linear subscript (B-1) of an element whose value is equal to zero, which is all that is necessary, otherwise.

The subscripting function must have access to the values of n and B. If it is an internal function they can be gotten from the dimension vector of A by using the name for it, say ADIM, which the main program used.

Otherwise, for external and assembly language functions, the values of n and B can be made available through PRØGRAM CØMMØN.

12.1.2 Use of the Subscription Routine

Once a routine is written to perform subscripting in other than the conventional manner, it is still necessary to indicate to MAD that this new routine is to be used instead of the standard one.

As always the array, A, in question must appear in a DIMENSIØN declaration in the following form

```
DIMENSIØN ...  A(p,ADIM), ...
```

where ADIM is the name of the dimension vector of A.

In this case (that is when redefining subscription) a statement of one of the following two forms must be used to set at least the first element of the dimension vector:

VECTØR VALUES ADIM = SUBF., d

VECTØR VALUES ADIM = SUBF., d_1, d_2, ..., d_n

where SUBF. is the name of the subscription function to be used with the array A and the d's are integer constants.

As discussed in section 3.6.2, Matrix DIMENSIØN Declaration, it was possible to compute or read in as data the entries in the dimension vector. This still is possible when redefining subscription but each of the entries described in section 3.6.2 must be stored in the dimension vector in the element whose linear subscript is equal to 1 plus the linear subscript of the element as described in section 3.6.2.

The name of the subscription function must appear in the first element of the dimension vector and may be put there in no other way than by presetting with VECTØR VALUES.

The first form of VECTØR VALUES declaration listed above may be used when the dimension entries are to be computed or read in; even if the array dimension, in the element of d, is to be computed or read in, an integer d, possibly a dummy, must appear. The second form is that used when elements can be preset, where d_1 is the array dimension, d_2 is the linear subscript of the base element, etc.

12.1.3 Available Subscription Routines

The following two special subscription routines are available in the subroutine library:

(a) SYMM.

The subroutine SYMM. calculates the correct linear subscript for a two dimensional, symmetric matrix for which only the elements above and on the main diagonal are stored, by rows.

SYMM. has access to the dimension vector of the matrix in question so no information other than the arguments is necessary.

(b) TRANSP.

The subroutine TRANSP. calculates the correct linear subscript for a complete two dimensional matrix, A, stored as the transpose of A; i.e. the transpose of A is stored by rows or A is stored by columns.

For example, a reference to the element A(6,2) using subroutine TRANSP. will result in the linear subscript for the element A(2,6).

TRANSP. has access to the dimension vector of the matrix in question so no information other than the arguments is necessary.

13 Recommended Programming Practices

This is an expandable section designed to have added to it, from time to time, recommendations for programming procedures which will result in increased efficiency in some manner, e.g. speed of compilation, speed of execution, conservation of storage, etc.

For this reason this section contains a separate table of contents.

By definition, this section is incomplete at any given time and there may exist methods other than these which appear here which could be used to accomplish the same ends.

TABLE OF CONTENTS, SECTION 13

13.2 Efficient Programming of Boolean Expressions

The object program produced by the MAD translator evaluates the terms of a Boolean expression from right to left.

It is clear that, in a Boolean expression of the form

T1 .ØR. T2 .ØR. --- .ØR. Tn

which may be called an ".ØR. expression", where the Ti's are any permissible terms and where the Boolean operations

.EXØR.
.AND.

do not appear in the expression, the value of the .ØR. expression is certain to be 1B, true, as soon as, in the evaluation process, the value of some Ti has been found to be 1B. The object program produced by the MAD translator evaluates the terms of an .ØR. expression until the value of some term is 1B or until all the terms have been evaluated, whichever happens first. Thus possible remaining terms, after the evaluation of some term whose value is 1B, are not evaluated needlessly.

This, together with the fact that the terms are evaluated from right to left as mentioned above, implies that, if the programmer has any way of judging, he should write the terms of an .ØR. expression from left to right in order of ascending likelyhood of truthfulness. Thus, in an expression

T1 .ØR. T2

if T2 is likely to be "true" (have value 1B) more often than T1, the expression should be written as it appears above, otherwise the order should be reversed, i.e.

T2 .ØR. T1

It is clear that, in a Boolean expression of the form

T1 .AND. T2 .AND. --- .AND. Tn

which may be called an ".AND. expression", where the Ti's are any permissible terms and where the Boolean operations

.ØR.
.EXØR.

do not appear, the value of the .AND. expression is certain to be OB, false, as soon as, in the evaluation process, the value of some Ti has been found to be OB. The object program produced by the MAD translator evaluates the terms of an .AND. expression until the value of some term is OB or until all the terms have been evaluated, whichever happens first. Thus possible remaining terms, after the evaluation of some term whose value is OB, are not evaluated needlessly.

This, together with the fact that the terms are evaluated from right to left as mentioned above, implies that, if the programmer has any way of judging, he should write the terms of an .AND. expression from left to right in order of descending likelyhood of truthfulness. Thus, in the expression

T1 .AND. T2

if T2 is likely to be "false" (have value OB) more often than T1, the expression should be written as it appears above, otherwise the order should be reversed, i.e.

T2 .AND. T1

13.3 Use of Parameters in Function Definitions

During compilation of a function definition, MAD records in a table, called the Parameter Use Table, all references in the function definition to the dummy arguments of the function, which will be references to the actual arguments when the function is used.

These entries are used to compile into the function subroutine, instructions to initialize the references upon each call of the function.

If too many references are made the Parameter Use Table may not be able to contain them all and compilation cannot be successful (in which case the comment PARAMETER USE TABLE EXCEEDED is printed).

Thus it may be necessary to minimize the number of entries in order to make compilation possible and it is desirable to minimize the number of entries both to shorten the object program and speed up the execution of the subroutine. This may be done in any of the following ways:

(1) If the dummy variable X supplies data to the subroutine, i.e. is input to the subroutine, and is referred to several times, one may use the substitution statement

Y = X

where Y is not a dummy variable, immediately after entry to the subroutine and use Y instead of X thereafter. Then initialization of references to X is done only in the substitution statement and there is only one entry for X in the Parameter Use Table.

(2) If the address of a variable is needed, as in the case of an output argument or an argument which is an array name, it is not possible to use the method of (1) above. Instead it may be possible to put the variable or array in PRØGRAM CØMMØN, by means of identical declarations in both the main program and subroutine if the subroutine happens to be either an external function or an assembly language subroutine. Then the variable or array will not be an argument at all.

14 Mechanics of Using MAD

14.1 Card Format

In order to compile a source language program using the MAD translator the program must be punched on cards. These cards have the following format:

Columns (inclusive)	Content
1 - 10	Statement Label
11	Remark or Continuation Designation
12 - 72	Statement
73 - 80	Identification

14.1.1 Statement Label Field

Statement labels may be punched anywhere in columns 1 - 10. Spaces are ignored.

14.1.2 Column 11

Except for a "Remark", a statement must begin on a card which has a blank in column 11.

14.1.2.1 Remarks Cards

If an "R" is punched in column 11, the card is a Remark Declaration (see sec. 3.1).

14.1.2.2 Continuation Cards

A card which has a decimal digit 0, 1, ---, 9, punched in column 11 is a continuation card. Continuation cards may be used in the event that a statement cannot be put on one card. The order of the digits appearing on consecutive continuation cards is not significant; the statement is ordered by the physical order of the cards comprising it.

A maximum of nine continuation cards may be used, so that a statement may be comprised of no more than ten cards.

14.1.3 The Statement Field

MAD language statements may be punched anywhere in columns 12 - 72 and only in columns 12 - 72. Spaces are ignored except when they appear between pairs of dollar signs ("$"'s).

14.1.4 The Identification Field

The information in the identification field is not translated by MAD. It is transcribed onto the source language listing produced during compilation. The user may punch any legitimate characters in this field.

It is good practice to include in this field identification information and sequence numbers which define the order of the cards in the program deck.

14.2 Diagnostics

During the process of translation many kinds of errors in the formation of statements and the allocation of storage can be detected. To understand this error detection and the subsequent printing of diagnostic comments some knowledge of the structure of the translator is helpful. The translation from statements to machine code is accomplished in three major sections:

(1) The decomposition of the original statements into arrays of binary operations and pseudo-operations;

(2) The analysis of all of the declarative information in order to allocate variable storage and identify the arithmetic types (i.e. modes) of variables.

(3) The combination of the information produced from (1) and (2) to translate the arrays to relocatable binary programs.

When an error is encountered in one of these sections the translation does not proceed to the next section. However, insofar as possible, the entire set of statements is processed through the section in which the error is detected and therefore more than one error may be detected. It should be understood then, that not all detectable errors may be found because:

(a) They are detectable only in a later stage of the translation;

(b) Some types of errors make it impossible to attempt further detection within the section in which it occurs;

(c) One error may actually obscure another error.

Occasionally, an error in one statement may be such that it causes the translator to misinterpret a second statement, thus giving an error indication even though no error exists in the later statement.

The printed diagnostic comment may very often have an alternative or ambiguous form. This results from the fact that it is frequently not possible to determine what form was intended, merely that the present structure is not admissible, and therefore some of the alternative possibilities are suggested by the comment.

After MAD has completed translation of a source language program, a list is printed of all variable names which appeared in the program, but which appeared only once. This list does not include names appearing in any of the following declarations:

PRØGRAM CØMMØN
ERASABLE
DIMENSIØN
VECTØR VALUES
EQUIVALENCE

Variable names which appear in the list are all assigned to the same location, under the assumption that they are not purposely used for anything except perhaps redundant labelling of statements.

The list is a valuable debugging aid due to the fact that it is very likely that in it misspelled names will appear.

14.3 Structure of Subroutines

The information in this and the following sections is to be found in much greater detail in other installation write-ups. However, the following sections should be sufficient for the general use of MAD.

Subroutines which are written for use by MAD programs, whether written in MAD as functions or in assembly language code, must be relocatable and must operate from the calling sequences the translator produces. Consider, for example the function call

```
FN. (A,B,C)
```

which might appear in the body of a statement. Assume that B is an array which has an associated dimension vector BDIM. Using assembly language notation for illustrative purposes, the calling sequence produced would be:

```
TSX  FN,4
TXH  A
TXH  B,O,BDIM
TXH  C
```

Input-output routines utilize two types of parameters, the regional and single variable types. In addition an error return is given as well as a format specification location. The parameter operation code used is STR and the end of the parameter list is indicated by an STR operation with a blank address. Thus the statement

```
READ FORMAT FMT, BETA, X(1) ...X(100), K
```

would produce the calling sequence

```
TSX  READ,4
STR  ERRØR
STR  FMT
STR  BETA
STR  X-1,0,X-100
STR  K
STR
```

On occasion it is useful to use the regional notation in subroutines which are not in the input-output category, for example, G. (GAMMA, DELTA, Z(10)...Z(20)). The calling sequence would be

```
TSX  G,4
TXH  GAMMA
TXH  DELTA
TIX  Z-10,0,Z-20
```

It is important to notice that in this example, as well as in the first, the parameters, if executed as instructions, would produce no operation.

It is beyond the scope of this manual to discuss the structure of relocatable programs. It is sufficient to say that a relocatable program must contain, in addition to the actual instructions in the program, information as to which addresses must be relocated at the time of loading for execution and which addresses must not. In addition, the first card (or record) of such programs must contain information about the size of the program, the number of subroutines it calls on, the amount of storage it will share with other subroutines, the location of the list of subroutines it calls on, and the names by which the routine itself is referred to. The symbolic names of the subroutines called on must appear as the first words after this information.

The execution of MAD programs requires the use of a loading routine to relocate and store the program and subroutines. A slightly modified BSS FORTRAN loader is automatically produced by the operating system in which MAD is imbedded. Also there are certain subroutines which may be automatically called for by a MAD program without an explicit reference to them in the source program.

14.4 Systems Subroutines

The use of the following names for functions (subroutines) should be avoided except where the operation is the one indicated here.

SYSTEM - Entry to this routine causes a return to the operating system. The END ØF PRØGRAM statement produces a call for this routine.

ERRØR - Entry to this routine also causes a return to the operating system. However, if a dump of storage was requested of the operating system such a print of storage will be produced before the return to the system. The ERRØR RETURN statement may produce a call for this routine.

TAPEWR - Entry to this subroutine causes BCD information to be written on tape. The arguments are: (1) location of the format specification, (2) tape number, and (3) a list of variables to be written. The WRITE BCD TAPE statement produces a call for this subroutine.

PRINT - Similar to TAPEWR above except that the use of the peripheral output tape is implied. The PRINT FØRMAT statement produces a call for this subroutine.

TAPERD - Entry to this subroutine causes BCD information to be read from tape. The arguments are: (1) location of the format specification, (2) tape number, and (3) a list of variables to be read. The READ BCD TAPE statement produces a call for this routine.

READ - Similar to TAPERD above except that the use of the peripheral input tape is implied. The READ FØRMAT statement produces a call for this subroutine.

PUNCH - Entry to this subroutine causes BCD information to be written on the peripheral output tape. The arguments are the same as those of PRINT and the PUNCH FØRMAT statement produces the call.

CØMMNT- Entry to this subroutine causes BCD information to be printed on the attached (on-line) printer. The arguments are the same as those of PRINT except that the line spacing is not given in the format specification since an automatic 1/6 page skip is produced. The statement producing the call is PRINT ØN LINE.

SETEØF - This name may appear explicitly in the form SETEØF.(S), where S is an expression of statement label mode designating the point of return when an end-of-file is encountered during the reading or writing of magnetic tape.

SETETT - This name may appear explicitly in the form SETETT.(S), where S is an expression in statement label mode designating the point of return when an end-of-tape is encountered during the reading or writing of magnetic tape.

SETERR - This name may appear explicitly in the form SETERR.(S), where S is an expression of statement label mode designating the point of return after an illegal data character or illegal format specification has been encountered by an input-output subroutine.

It should be understood that the subroutines described above may be called by other subroutines as well as statements. Thus, for example, SETEØF is called by TAPERD, TAPEWR calls PRINT, and all of the input-output routines call ERRØR.

15 Examples of MAD Programs

The following examples illustrate how some programs may be written in the MAD language. Since they were written to illustrate as many features of the language as possible, they are not necessarily the most efficient or elegant programs which could have been written.

15.1 Scientific Examples

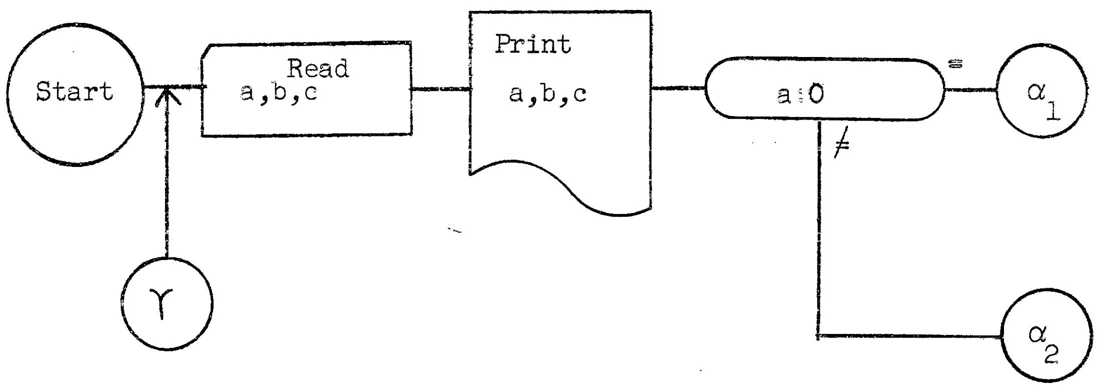
Example 1

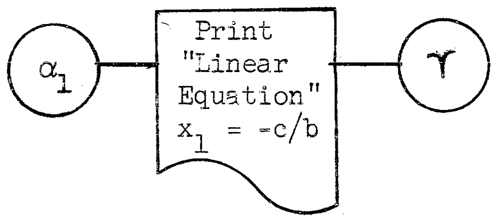
Problem: To solve the quadratic equation $ax^2 + bx + c = 0$ for various sets of coefficients a, b, and c.

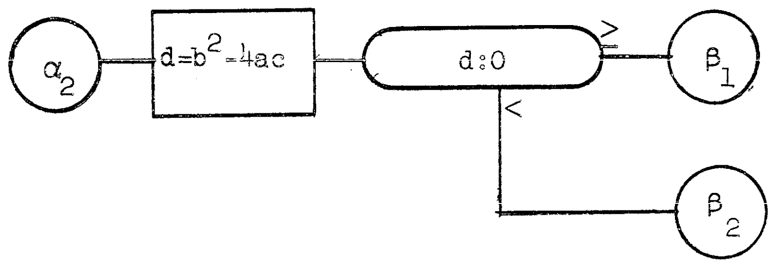
Analysis: Let x_1 and x_2 be the two roots of the equation. Then their values are found by the formulas,

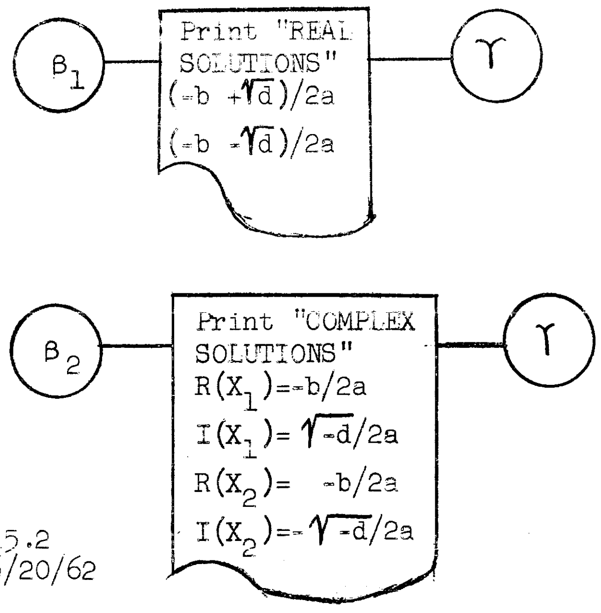
$$x_1 = \frac{-b + \sqrt{b^2 - 4ac}}{2a} \qquad x_2 = \frac{-b - \sqrt{b^2 - 4ac}}{2a}$$

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








β_1
Print "REAL SOLUTIONS" $(-b+\sqrt{d})/2a$ $(-b-\sqrt{d})/2a$
γ
β_2
Print "COMPLEX SOLUTIONS" $R(X_1)=-b/2a$ $I(X_1)=\sqrt{-d}/2a$ $R(X_2)=-b/2a$ $I(X_2)=-\sqrt{-d}/2a$
γ

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

```
              R
              RMAIN PROGRAM
              R
GAMMA          READ FORMAT INPUT,A,B,C
               PRINT FORMAT CHECK,A,B,C
               WHENEVER A .NE. 0,TRANSFER TO ALPHA2
ALPHA1         PRINT FORMAT LINEAR,-C/B
               TRANSFER TO GAMMA
ALPHA2         D = B .P.2 -4.*A*C
               WHENEVER D .L. 0.,TRANSFER TO BETA2
BETA1          PRINT FORMAT REAL,(-B+SQRT.(D))/(2.*A),(-B-SQRT.(D))/(2.*A)
               TRANSFER TO GAMMA
BETA2          PRINT FORMAT COMPLX,-B/(2.*A),SQRT.(-D)/(2.*A),
              1-B/(2.*A),-SQRT.(-D)/(2.*A)
               TRANSFER TO GAMMA
              R
              RFORMAT SPECIFICATIONS
              R
               VECTOR VALUES INPUT = $ 3F10.4*$
               VECTOR VALUES CHECK = $ 4H0A = F10.4,S8,
              13HB = F10.4,S8,3HC = F10.4*$
               VECTOR VALUES LINEAR = $21H0LINEAR EQUATION, X = F10.4*$
               VECTOR VALUES REAL = $21H0REAL SOLUTIONS,X1 =
              1F10.4,S8,4HX2 = F10.4*$
               VECTOR VALUES COMPLX = $19H0COMPLEX SOLUTIONS,
              1S4,7HR(X1) = F10.4,S8,7HI(X1) = F10.4,S8,
              27HR(X2) = F10.4,S8,7HI(X2) = F10.4*$
               END OF PROGRAM

$      DATA
        4.        -8.         4.
         0         5.        10.
        1.         1.         1.
```

Example 2

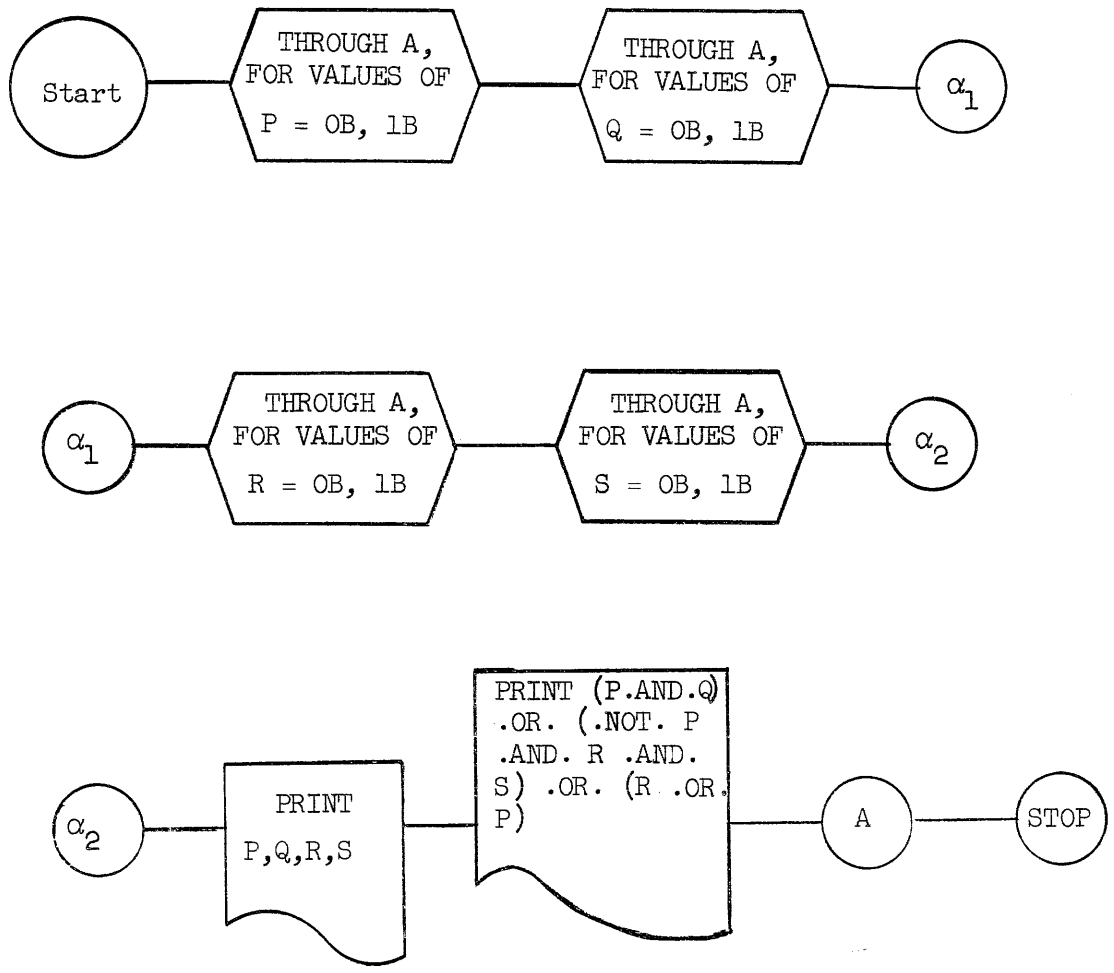
Problem: A logical (Boolean) expression such

T = (P .AND. Q) .ØR. (.NØT. P .AND. R .AND. S) .ØR. (R .ØR. P)

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

P	Q	R	S	T
OB	OB	OB	OB	OB
OB	OB	OB	1B	OB
CB	OB	1B	OB	1B
OB	OB	1B	1B	1B
OB	1B	OB	OB	OB
OB	1B	OB	1B	OB
OB	1B	1B	OB	1B
OB	1B	1B	1B	1B
1B	OB	CB	OB	1B
1B	OB	OB	1B	1B
1B	OB	1B	OB	1B
1B	OB	1B	1B	1B
1B	1B	OB	OB	1B
1B	1B	OB	1B	1B
1B	1B	1B	OB	1B
1B	1B	1B	1B	1B

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


Start
THROUGH A, FOR VALUES OF P = OB, 1B
THROUGH A, FOR VALUES OF Q = OB, 1B
α_1
α_1
THROUGH A, FOR VALUES OF R = OB, 1B
THROUGH A, FOR VALUES OF S = OB, 1B
α_2
α_2
PRINT P,Q,R,S
PRINT (P.AND.Q) .OR. (.NOT. P .AND. R .AND. S) .OR. (R .OR. P)
A
STOP

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

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

NØRMAL MØDE IS BØØLEAN

could have been used as the second statement of this program instead of the BØØLEAN declaration.

Example 3

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

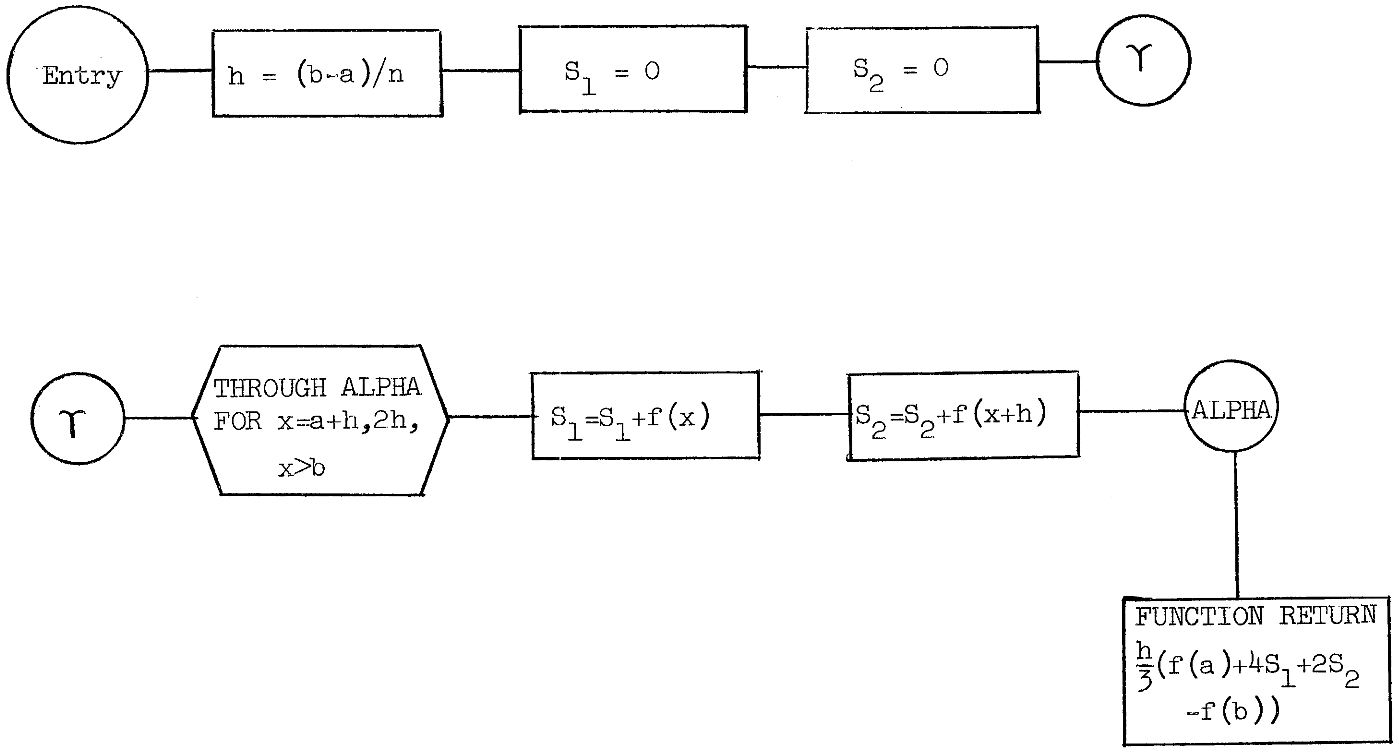
Analysis: By Simpson's Rule, $\int_a^b f(x)dx \approx \frac{b-a}{3N}(y_0 + 4y_1 + 2y_2 + 4y_3 + \cdots$

$+ 4y_{N-1} + y_N)$,

where $y_i = f(x_i)$, and $a = x_0, x_1, \ldots, x_N = b$ are the partition points of the interval [a, b].

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

Flow Diagram:


Entry
h = (b-a)/n
S_1 = 0
S_2 = 0
ϒ
ϒ
THROUGH ALPHA FOR x=a+h,2h, x>b
S_1=S_1+f(x)
S_2=S_2+f(x+h)
ALPHA
FUNCTION RETURN $\frac{h}{3}$(f(a)+4S_1+2S_2 -f(b))

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

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

```
READ        READ FORMAT INPUTS,A,B,N,R
            INTEGER N
            WHENEVER 0. .LE. R .AND. R .LE. 3.
            PRINT FORMAT RESULT,A,B,N,R,SIMPS.(A,B,N,F1.)
            OTHERWISE
            PRINT FORMAT RESULT,A,B,N,R,SIMPS.(A,B,N,F2.)
            END OF CONDITIONAL
            TRANSFER TO READ
           R
           RDEFINITION OF FUNCTIONS
           R
            INTERNAL FUNCTION F1.(X) = SIN.(3.*X)-COS.(R*X+1.)
            INTERNAL FUNCTION F2.(X) = SIN.(3.*X)-COS.(R*X)
           R
           RFORMAT SPECIFICATIONS
           R
            VECTOR VALUES INPUTS = $2F12.4,I6,F12.4*$
            VECTOR VALUES RESULT = $23H1 FOR THE INTERVAL FROM
           1F12.4,3H TO F12.4,5H WITH I6,38H EQUAL SUB-INTERVALS AND
           2PARAMETER F12.4/29H0THE VALUE OF THE INTEGRAL IS F12.4*$
            END OF PROGRAM
```

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

```
           S1 = S1+F.(X)
ALPHA      S2 = S2+F.(X+H)
           FUNCTION RETURN H*(F.(A)+4.*S1+2.*S2-F.(B))/3.
           END OF FUNCTION

$       DATA
           0         2.    10         10.
```

An alternate way to write the first eight lines of this program, illustrating one use of the FUNCTIØN NAME mode, would be:

```
READ       READ FORMAT INPUTS,A,B,N,R
           INTEGER N
           WHENEVER 0..LE. R .AND. R .LE. 3.
           S = F1.
           OTHERWISE
           S = F2.
           END OF CONDITIONAL
           PRINT FORMAT RESULT,A,B,N,R, SIMPS. (A,B,N,S)
           TRANSFER TO READ
           FUNCTION NAME S
```

Example 4

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

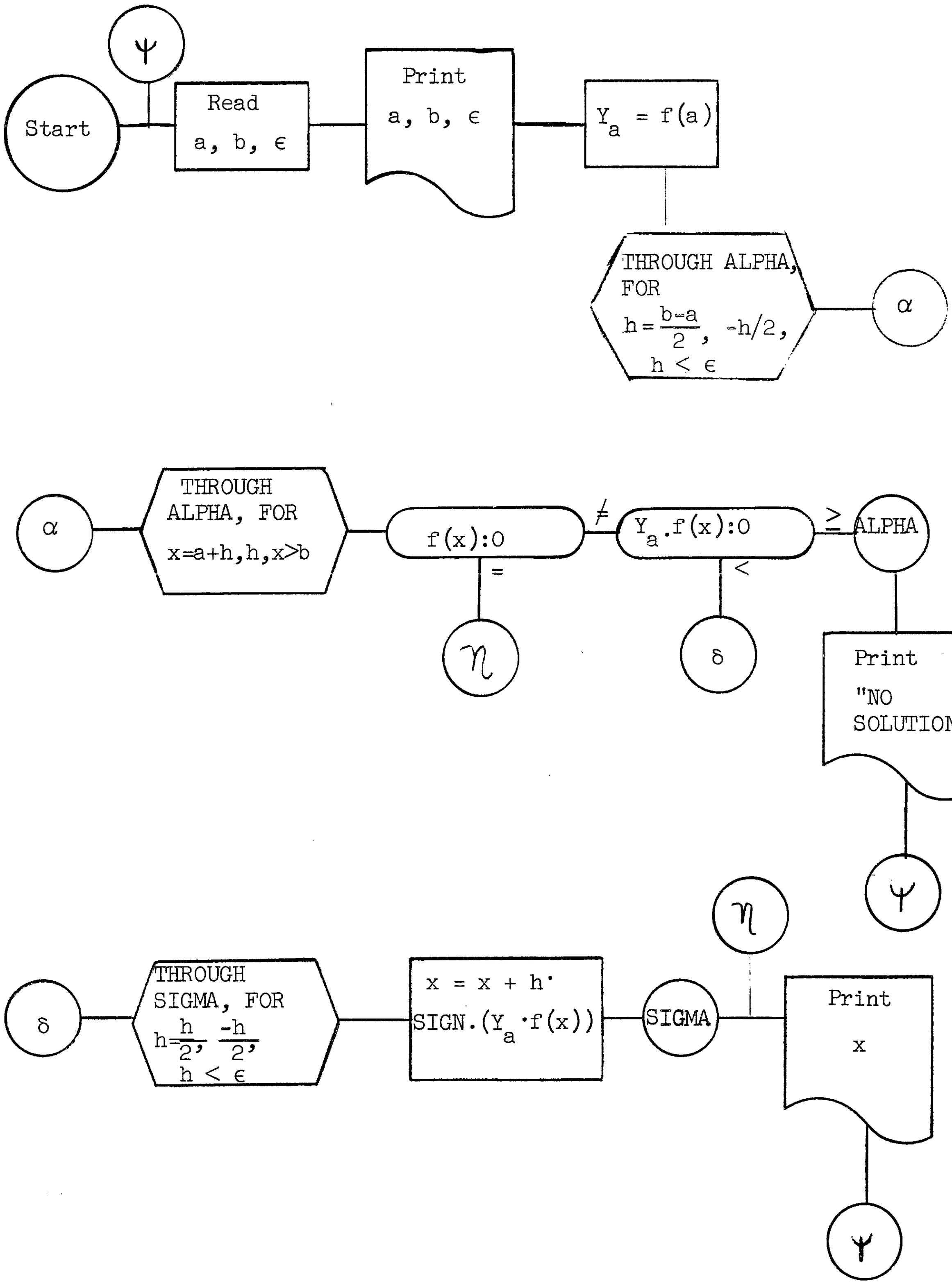
Analysis: We specify a, b, and ϵ as parameters. The method used will be "half-interval convergence," in which the function is evaluated at $x = a$, and then the interval is scanned for a change of sign* in the value of $f(x)$. If no change of sign is found, the scanning is repeated with a step size for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

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

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

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

*A change of sign is detected when the numbers involved have a negative product.



Ψ
Start
Read a, b, ϵ
Print a, b, ϵ
Y_a = f(a)
THROUGH ALPHA, FOR h = (b-a)/2, -h/2, h < ϵ
α
α
THROUGH ALPHA, FOR x=a+h,h,x>b
f(x):0
≠
=
η
Y_a.f(x):0
≥
<
δ
ALPHA
Print "NO SOLUTION"
Ψ
δ
THROUGH SIGMA, FOR h=h/2, -h/2, h < ϵ
x = x + h· SIGN.(Y_a·f(x))
SIGMA
η
Print x
Ψ

Definition

$$\text{SIGN.}(Z) = Z/|Z|$$

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

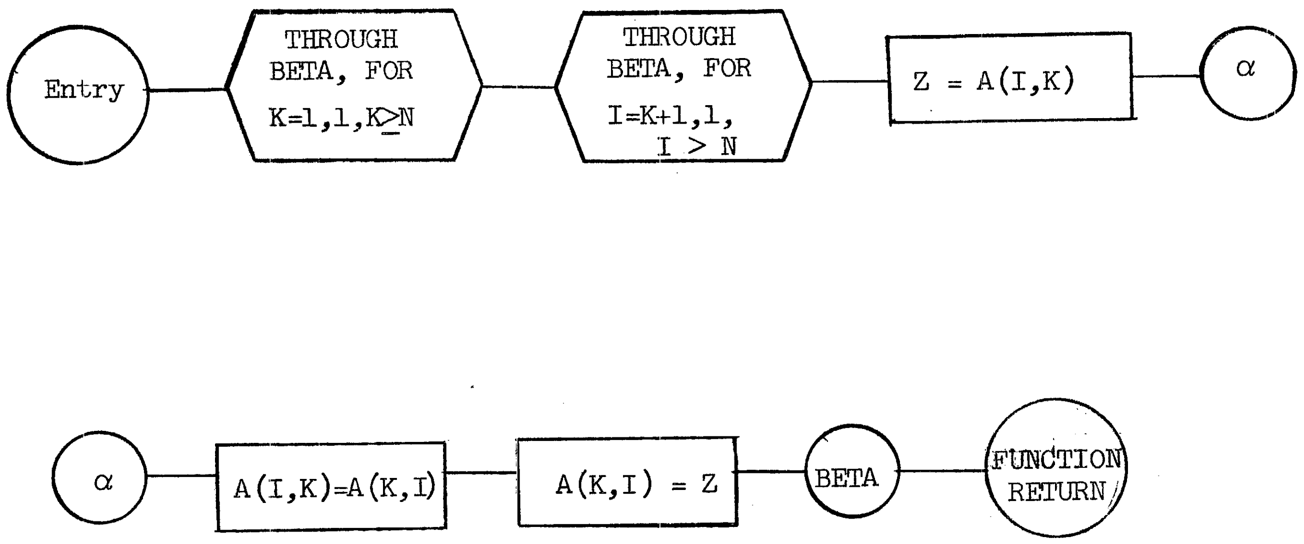
```
              INTERNAL FUNCTION F.(Z)= Z .P. 2 - 2.
PSI           READ FORMAT ABEPS, A,B,EPS
              PRINT FORMAT INVAL, A,B,EPS
              YA = F.(A)
              THROUGH ALPHA,FOR H = (B-A)/2.,-H/2.,H .L. EPS
              THROUGH ALPHA,FOR X = A+H,H, X .G. B
              WHENEVER F.(X) .E. 0.,TRANSFER TO ETA
ALPHA         WHENEVER YA*F.(X) .L. 0.,TRANSFER TO DELTA
              PRINT FORMAT NO ROOT
              TRANSFER TO PSI
             R
             RTHE NEXT SECTION IS ENTERED WHEN A CHANGE
             ROF SIGN IS FOUND
             R
DELTA         THROUGH SIGMA,FOR H=H/2.,-H/2.,H .L.EPS
SIGMA         X = X+SIGN.(YA*F.(X))*H
ETA           PRINT FORMAT ROOT,X
              TRANSFER TO PSI
             R
             RDEFINITION OF SIGN. FUNCTION
             R
              INTERNAL FUNCTION SIGN.(Z) = Z/.ABS.Z
             R
             RFORMAT SPECIFICATIONS
             R
              VECTOR VALUES ABEPS = $ 3F12.4*$
              VECTOR VALUES INVAL = $18H1INPUT*VALUES, A = F12.4,S3,
             13HB = F12.4,S3,5HEPS = F12.4*$
              VECTOR VALUES NO ROOT = $12H0NO*SOLUTION *$
              VECTOR VALUES ROOT = $14H0SOLUTION, X = F12.4*$
              END OF PROGRAM

$       DATA
             1.          2.          .01
```

Example 5

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

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

Flow Diagram:



```
          EXTERNAL FUNCTION (A,N)
          ENTRY TO TRANS.
          THROUGH BETA, FOR K = 1,1, K .GE. N
          THROUGH BETA, FOR I = K+1,1, I .G. N
          Z = A(I,K)
          A(I,K) = A(K,I)
BETA      A(K,I) = Z
          FUNCTION RETURN
          INTEGER N,K,I
          END OF FUNCTION
```

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

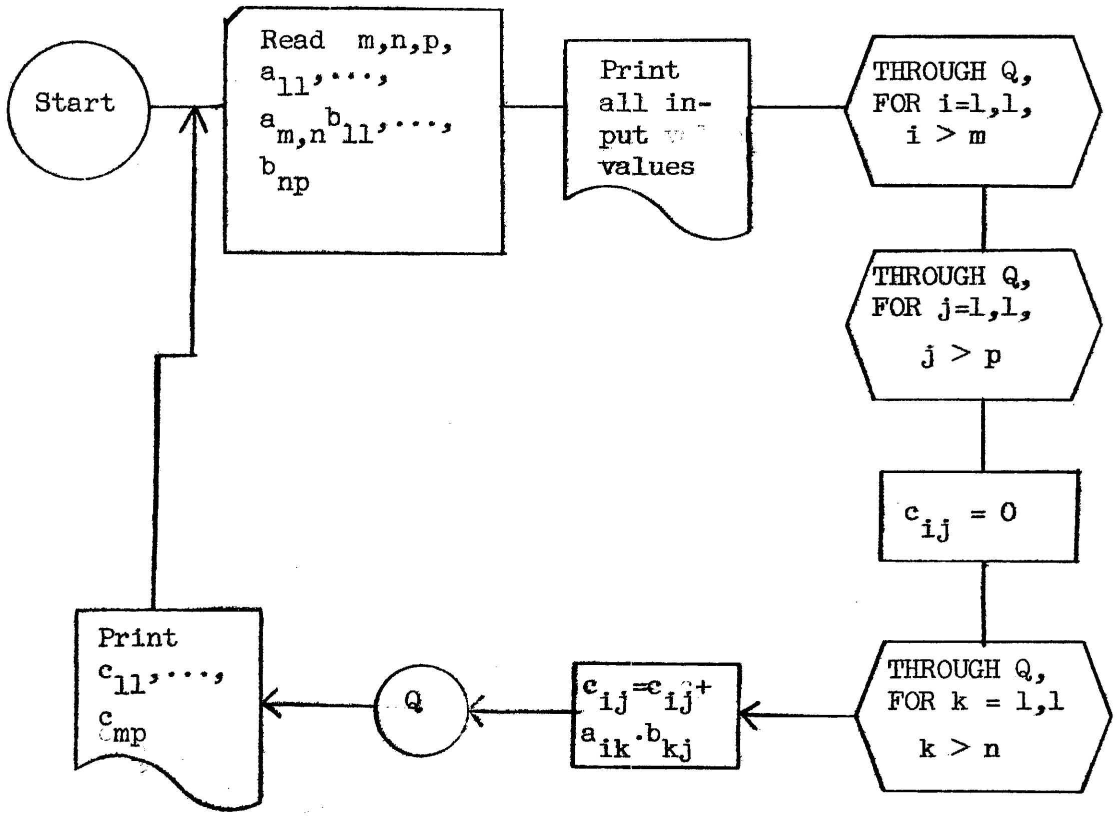
Example 6

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

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

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

Flow Diagram:


Start
Read m,n,p, $a_{11},\ldots,$ $a_{m,n}$ $b_{11},\ldots,$ b_{np}
Print all input values
THROUGH Q, FOR i=1,1, i > m
THROUGH Q, FOR j=1,1, j > p
$c_{ij} = 0$
THROUGH Q, FOR k = 1,1 k > n
$c_{ij}=c_{ij}+ a_{ik} \cdot b_{kj}$
Q
Print $c_{11},\ldots,$ c_{mp}

```
             DIMENSION A(1500,ADIM),B(1500,BDIM),C(1500,CDIM)
             EQUIVALENCE(N,ADIM(2)),(P,BDIM(2))
             INTEGER I,J,K,M,N,P
             VECTOR VALUES ADIM = 2,0,0
             VECTOR VALUES BDIM = 2,0,0
             VECTOR VALUES CDIM = 2,0,0
READ         READ FORMAT INPUT,M,N,P,A(1,1)...A(M,N),B(1,1)...B(N,P)
             PRINT FORMAT INVAL,M,N,P,A(1,1)...A(M,N),B(1,1)...B(N,P)
             CDIM(2) = P
             THROUGH Q, FOR I = 1,1, I .G. M
             THROUGH Q, FOR J = 1,1, J .G. P
             C(I,J) = 0.
             THROUGH Q, FOR K = 1,1,K .G. N
Q            C(I,J) = C(I,J) + A(I,K)*B(K,J)
             PRINT FORMAT RESULT,C(I,1)...C(M,P)
             TRANSFER TO READ
            R
            RFORMAT SPECIFICATIONS
            R
             VECTOR VALUES INPUT = $3I4/(6F12.4)*$
             VECTOR VALUES INVAL = $13H0INPUT*VALUES/4H0M = I6,S6,3HN = I6
            1,S6,3HP = I6//(1H0,8F13.4)*$
             VECTOR VALUES RESULT = $9H1C MATRIX//(1H0,8F13.4)*$
             END OF PROGRAM

$      DATA
   2   3   3
         1.          2.          3.          4.          5.          6.
         7.          8.          9.         10.         11.         12.
        13.         14.         15.
```

Example 7

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

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

We denote the system of equations to be solved by:

$$\begin{array}{llcccccccc} & a_{11}x_1 &+& a_{12}x_2 &+& \cdots &+& a_{1n}x_n &=& a_{1,n+1} \\ (1) & a_{21}x_1 &+& a_{22}x_2 &+& \cdots &+& a_{2n}x_n &=& a_{2,n+1} \\ & \cdot && \cdot &&&& \cdot && \cdot \\ & \cdot && \cdot &&&& \cdot && \cdot \\ & \cdot && \cdot &&&& \cdot && \cdot \\ & a_{n1}x_1 &+& a_{n2}x_2 &+& \cdots &+& a_{nn}x_n &=& a_{n,n+1} \end{array}$$

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

(2) $$a_{kj} = a_{kj}/a_{kk}$$

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

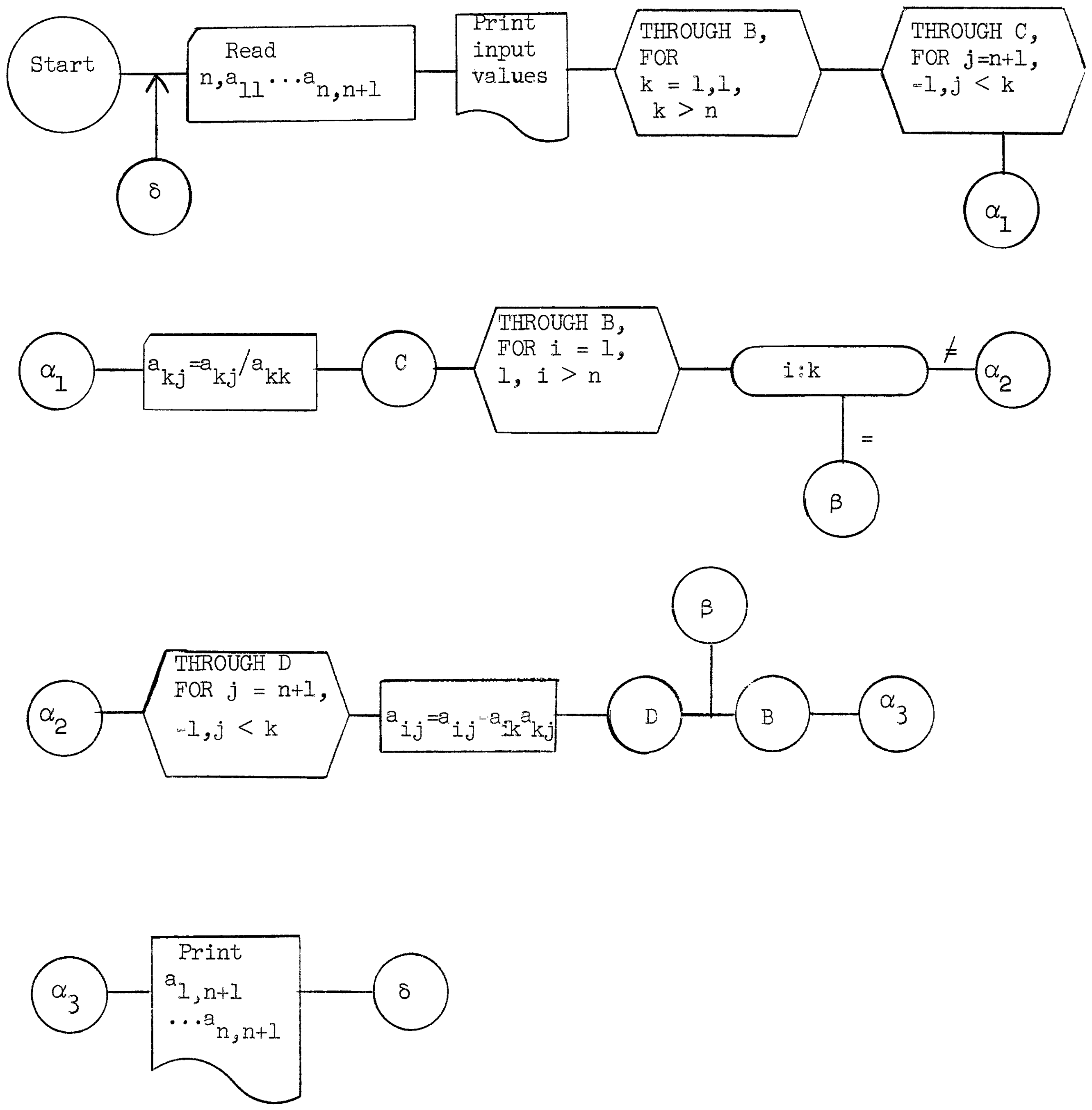
where the value of a_{kj} in (3) is the result of (2). These transformations are performed for $k = 1,2,\ldots,n$. For each (fixed) k, we will let $i = 1$, 2,...,k-1,k+1, ...,n, so as to operate on all rows except i=k. While transforming each row we will cycle on j from right to left; i.e., j = n+1,n,n-1,...,k, and we stop at j=k since for j < k there is no change in the matrix.

The array

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

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

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


Start
Read $n, a_{11} \cdots a_{n,n+1}$
Print input values
THROUGH B, FOR $k = 1, 1, k > n$
THROUGH C, FOR $j = n+1, -1, j < k$
δ
α_1
$a_{kj} = a_{kj}/a_{kk}$
C
THROUGH B, FOR $i = 1, 1, i > n$
$i : k$
$\neq$
α_2
$=$
β
THROUGH D FOR $j = n+1, -1, j < k$
$a_{ij} = a_{ij} - a_{ik} a_{kj}$
D
B
α_3
Print $a_{1,n+1} \cdots a_{n,n+1}$

```
              DIMENSION A(420,ADIM)
              VECTOR VALUES ADIM = 2,0,0
DELTA         READ FORMAT NVAL,N
              ADIM(2) = N+1
              READ FORMAT INPUT, A(1,1)...A(N,N+1)
              PRINT FORMAT INVAL,N,A(1,1)...A(N,N+1)
              THROUGH B, FOR K = 1,1,K .G. N
              THROUGH C, FOR J = N+1,-1,J .L. K
C             A(K,J) = A(K,J)/A(K,K)
              THROUGH B, FOR I = 1,1,I .G. N
              WHENEVER I .E. K, TRANSFER TO B
              THROUGH D, FOR J = N+1,-1, J .L. K
D             A(I,J) = A(I,J)-A(I,K)*A(K,J)
B             CONTINUE
              THROUGH E, FOR I = 1,1,I .G. N
E             PRINT FORMAT RESULT,I,A(I,N+1)
              TRANSFER TO DELTA
              INTEGER I,J,K,N
             R
             RFORMAT SPECIFICATIONS
             R
              VECTOR VALUES NVAL = $ I4* $
              VECTOR VALUES INPUT = $ 6F12.4* $
              VECTOR VALUES INVAL = $7H1 INPUT//4H N = I4//
             17H MATRIX//(1H0,8F12.4)*$
              VECTOR VALUES RESULT = $ 1H0,S20,2HX(,I2,3H) = F12.4*$
              END OF PROGRAM

$       DATA
   3
             1.          1.          1.          6.          -1.          0
             0.         -1.         -1.         -2.          -9.       -32.
```

15.2 Business Data Processing Examples

Example 1

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

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

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

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

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

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

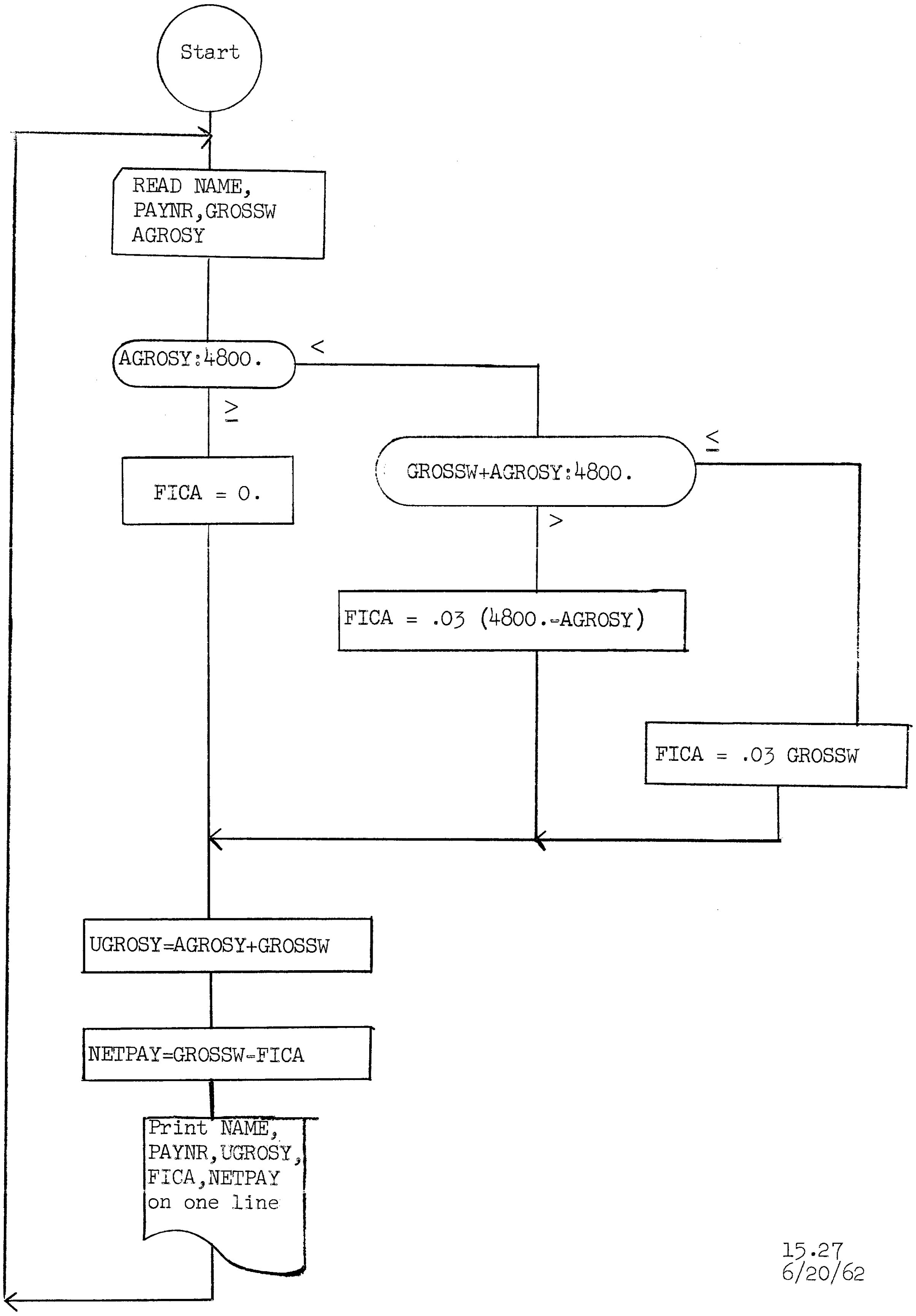
Card Columns	Information
1-30	(a) employee's name
31-38	(b) payroll number
39-44	(c) gross pay for the current week in the form XXX.XX
45-52	(d) accumulated gross pay for the current year in the form XXXXX.XX

The printed output will be in the following format:

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

Flow Chart: We will use the following abbreviations,

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


Start
READ NAME, PAYNR, GROSSW AGROSY
AGROSY:4800.
<
≥
FICA = 0.
GROSSW+AGROSY:4800.
≤
>
FICA = .03 (4800.-AGROSY)
FICA = .03 GROSSW
UGROSY=AGROSY+GROSSW
NETPAY=GROSSW-FICA
Print NAME, PAYNR, UGROSY, FICA, NETPAY on one line

```
START       READ FORMAT IN, NAME(1)...NAME(5),PAYNR,GROSSW,AGROSY
            VECTOR VALUES IN = $5C6,I8,F6.2,F8.2*$
            DIMENSION NAME(5)
            INTEGER PAYNR, NAME
            WHENEVER AGROSY .GE. 4800.,TRANSFER TO BIGGRS
            WHENEVER GROSSW+AGROSY .G. 4800.,TRANSFER TO BIGTOT
            FICA = .03*GROSSW
            TRANSFER TO UPDATE
BIGGRS      FICA = 0.
            TRANSFER TO UPDATE
BIGTOT      FICA = .03*(4800.-AGROSY)
UPDATE      UGROSY = AGROSY+GROSSW
            NETPAY = GROSSW-FICA
            PRINT FORMAT OUT,NAME(1)...NAME(5),PAYNR,UGROSY,FICA,NETPAY
            TRANSFER TO START
            VECTOR VALUES OUT = $1H0,5C6,S3,I8,S3,F8.2,S3,F5.2,S3,F6.2*$
            END OF PROGRAM

$      DATA
GEORGE WASHINGTON                12345678  100.   4800.
JOHN ADAMS                       12345679  200.   4900.
THOMAS JEFFERSON                 12345680  200.   4600.
JAMES MADISON                    12345681  200.   4700.
JOHN QUINCY ADAMS                12345682  100.    300.
```

Alternate Program:

```
START       READ FORMAT IN,NAME(1)...NAME(5),PAYNR,GROSSW,AGROSY
            VECTOR VALUES IN = $5C6,I8,F6.2,F8.2*$
            DIMENSION NAME(5)
            INTEGER PAYNR, NAME
            WHENEVER AGROSY .GE. 4800.
            FICA = 0.
            OR WHENEVER GROSSW + AGROSY .G. 4800.
            FICA = .03*(4800.-AGROSY)
            OTHERWISE
            FICA = .03*GROSSW
            END OF CONDITIONAL
            UGROSY = AGROSY+GROSSW
            NETPAY = GROSSW-FICA
            PRINT FORMAT OUT,NAME(1)...NAME(5),PAYNR,UGROSY,FICA,NETPAY
            TRANSFER TO START
            VECTOR VALUES OUT = $1H0,5C6,S3,I8,S3,F8.2,S3,F5.2,S3,F6.2*$
            END OF PROGRAM

$      DATA
GEORGE WASHINGTON                12345678  100.   4800.
JOHN ADAMS                       12345679  200.   4900.
THOMAS JEFFERSON                 12345680  200.   4600.
```

```
JAMES MADISON                   12345681  200.   4700.
JOHN QUINCY ADAMS               12345682  100.    300.
```

Notes on Example 1

1. The maximum number of characters which can be stored in one machine word is six. Hence, we need five machine words to store the 30 characters allowed for the employee's name. We need to give a dimension declaration stating that NAME is actually to be a block and that NAME(5) is the last word of this block. In the read and print statements we specify that the whole block is to be read or printed by writing NAME(1)... NAME(5) and giving the format specification 5C6, i.e., 5 words of 6 characters.
2. Since the payroll number is an integer (I8, i.e., an 8 digit integer) we give an integer mode declaration stating that PAYNR is an integer. Similarly, since alphabetic information is assumed to be in the integer mode, NAME is also declared to be integer.

Example 2

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

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

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

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

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

Notes on Example 2

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

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

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

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

where | | denote the usual "absolute value". In fact, by using this function, a simple one-line formula for this function is:

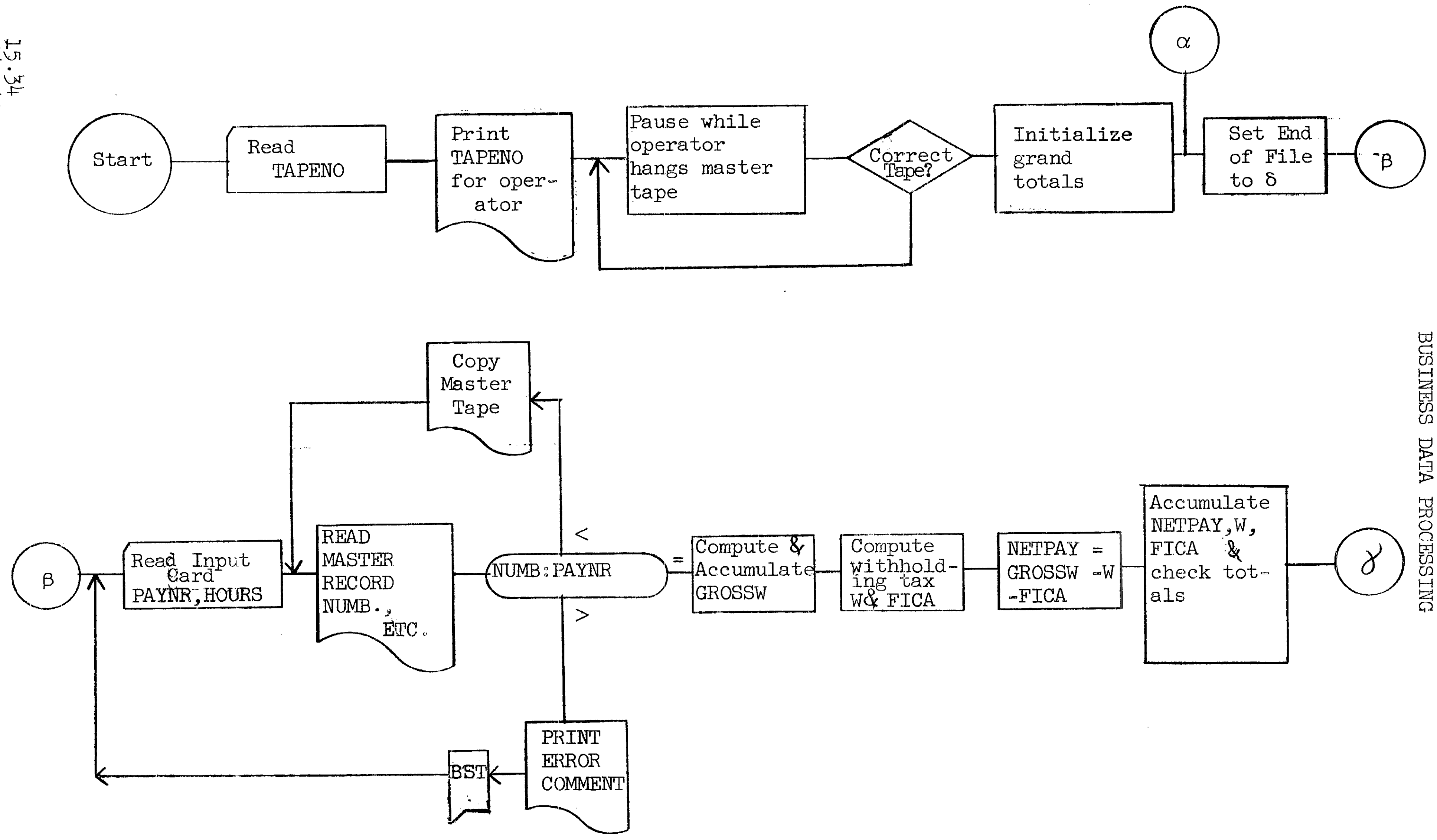
FICA = .03 EXCESS.((GRØSSW - EXCESS.(AGRØSY, 4800.)),0)

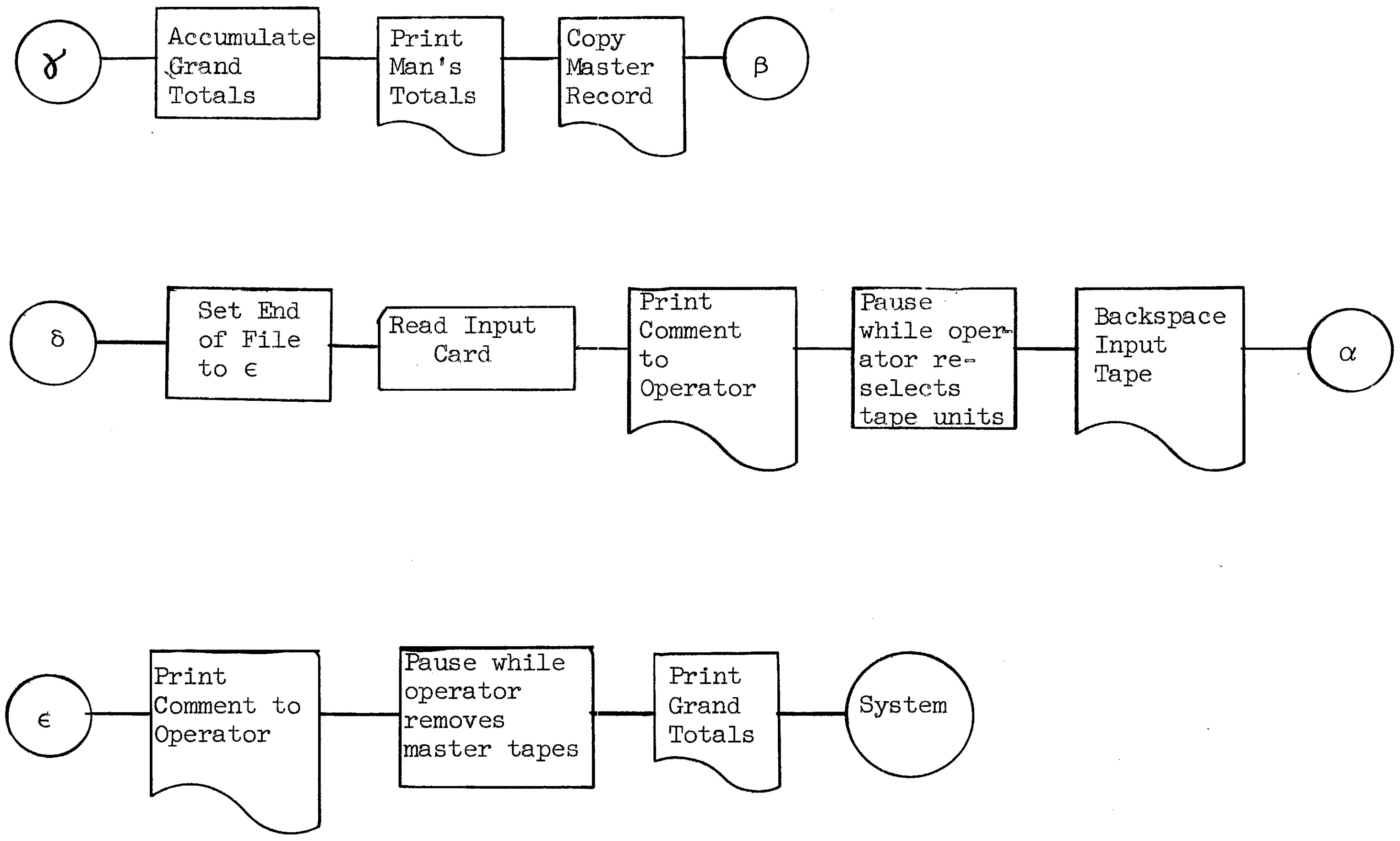
where AGRØSY is assumed to already contain GRØSSW, i.e., to have been updated already. We shall also apply this function in the case of the withholding tax to guarantee that we do not make a negative deduction. Thus

W = .18*EXCESS. (GRØSSW, 13*EXEMPT)

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

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


Start
Read TAPENO
Print TAPENO for operator
Pause while operator hangs master tape
Correct Tape?
Initialize grand totals
α
Set End of File to δ
β
β
Read Input Card PAYNR, HOURS
READ MASTER RECORD NUMB., ETC.
Copy Master Tape
NUMB : PAYNR
<
=
>
Compute & Accumulate GROSSW
Compute withholding tax W & FICA
NETPAY = GROSSW -W -FICA
Accumulate NETPAY, W, FICA & check totals
γ
PRINT ERROR COMMENT
BST


γ
Accumulate Grand Totals
Print Man's Totals
Copy Master Record
β
δ
Set End of File to ε
Read Input Card
Print Comment to Operator
Pause while operator re-selects tape units
Backspace Input Tape
α
ε
Print Comment to Operator
Pause while operator removes master tapes
Print Grand Totals
System

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

```
           REWIND TAPE 4
           EXECUTE SETEOF.(C FILE)
           READ FORMAT EMPLOY,DUMMY,DUMMY
           PRINT FORMAT NOMAN,PAYNR,TAPENO
           PRINT ON LINE FORMAT NOMAN,PAYNR,TAPENO
           PAUSE NO. 4
           TAPENO=TAPENO+1
           BACKSPACE RECORD OF TAPE 7
           READ BINARY TAPE 3,DUMMY
           TRANSFER TO REDO
C FILE      PRINT ON LINE FORMAT OFF,TAPENO,TAPENO+1
           PAUSE NO. 2
           PRINT FORMAT TOTALS,CUMGRS,CUMFIC,CUMNET,CUMW
           EXECUTE SYSTEM.
          R
           INTERNAL FUNCTION EXCESS.(X,Y)=(X-Y+ .ABS. (X-Y))/2.
          R
          RFORMAT SPECIFICATIONS
          R
           VECTOR VALUES IDENT = $I8*$
           VECTOR VALUES OPER = $15H4MOUNT TAPE NO. I8,S2,30HON TAPE UNI
          1T NO. 3,PRESS START*$
           VECTOR VALUES WRONG = $48H4THE WRONG TAPE HAS BEEN USED. PLEA
          1SE TRY AGAIN.*$
           VECTOR VALUES EMPLOY = $I8,F10.2*$
           VECTOR VALUES ERROR = $37H0ERROR IN CHECKING TOTALS FOR MAN N
          1O. I8*$
           VECTOR VALUES OUTPUT = $1H0,I8,4F20.2*$
           VECTOR VALUES OFF = $24H4REMOVE TAPE 3,LABEL IT I8,S4,23HREMO
          1VE TAPE 4, LABEL IT I8*$
           VECTOR VALUES NOMAN = $38H0THERE IS NO MASTER RECORD FOR MAN
          1NO.I8/22H0PULL TAPE 3. LABEL IT I8/51H0RESELECT TAPE 4 AS TAP
          2E 3 AND HANG BLANK TAPE ON 4/16H0THEN PUSH START*$
           VECTOR VALUES ORDER = $8H0MAN NO.I8,43H IS OUT OF ORDER OR NO
          1 MASTER RECORD EXISTS*$
           VECTOR VALUES TOTALS = $13H1CUM. GROSS =F10.2/12H0CUM. FICA =
          1F10.2/11H0CUM. NET = F10.2/23H0CUM. WITHHOLDING TAX = F10.2*$
           END OF PROGRAM
```

Example 3

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

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

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

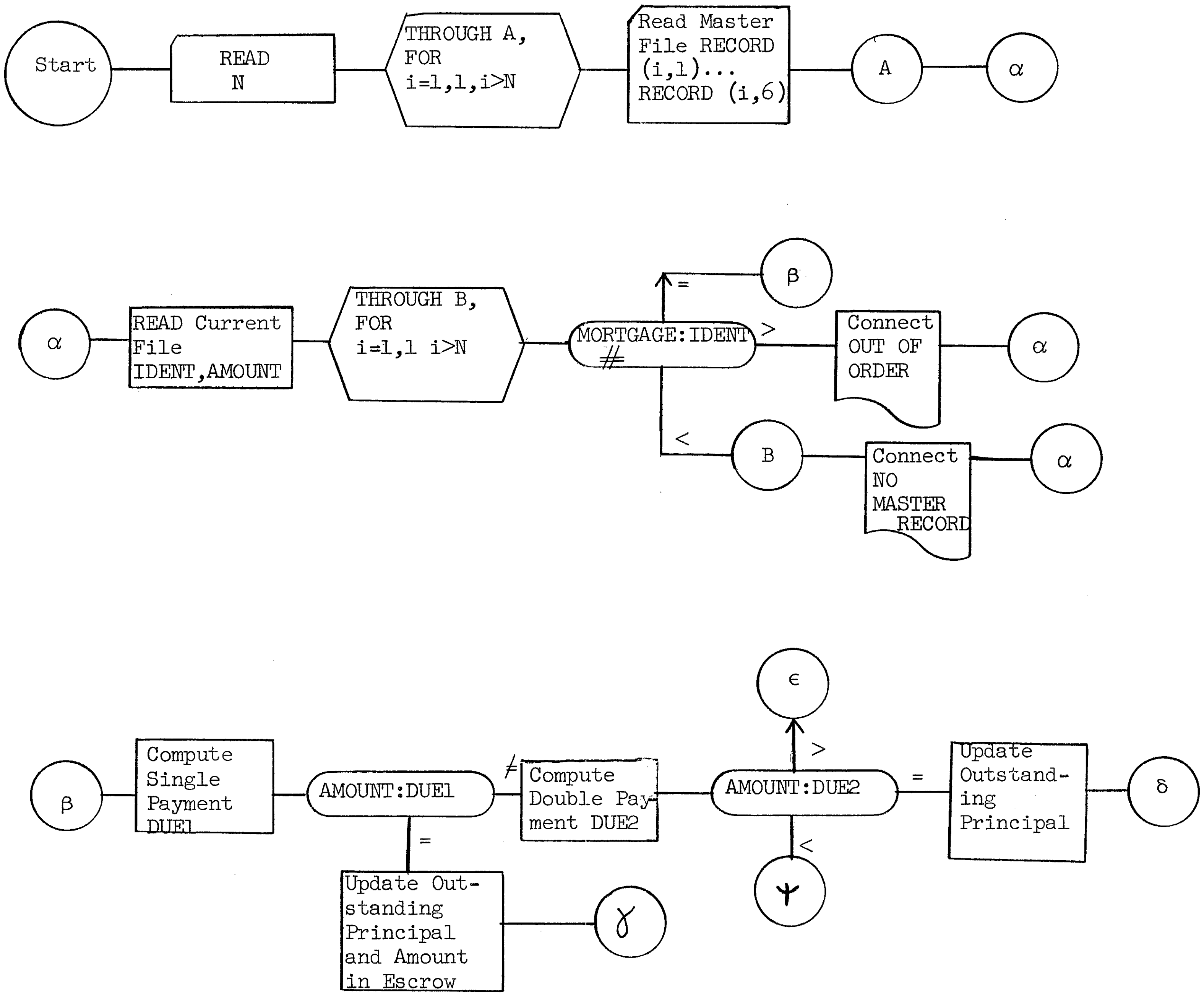
Notes on Example 3

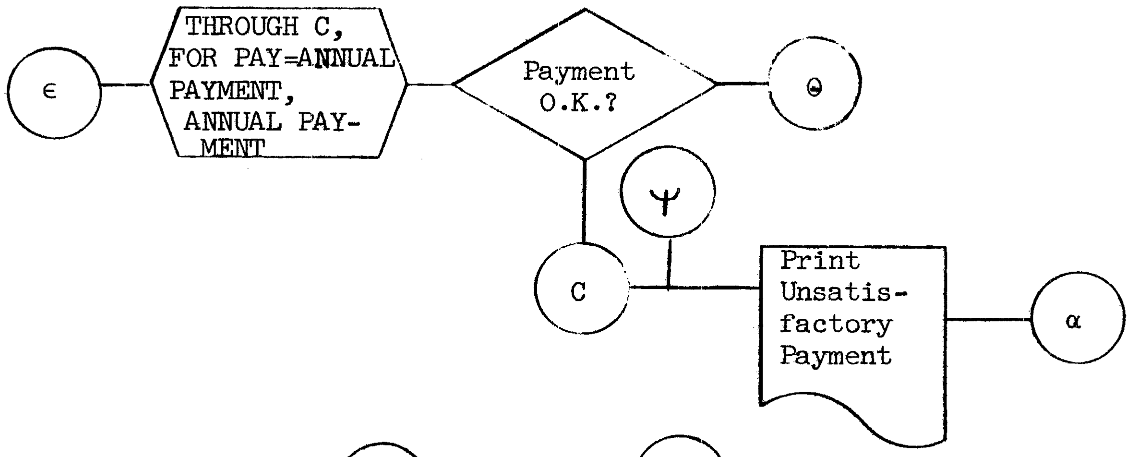
Note 1: Observe the use of the alphabetic constant

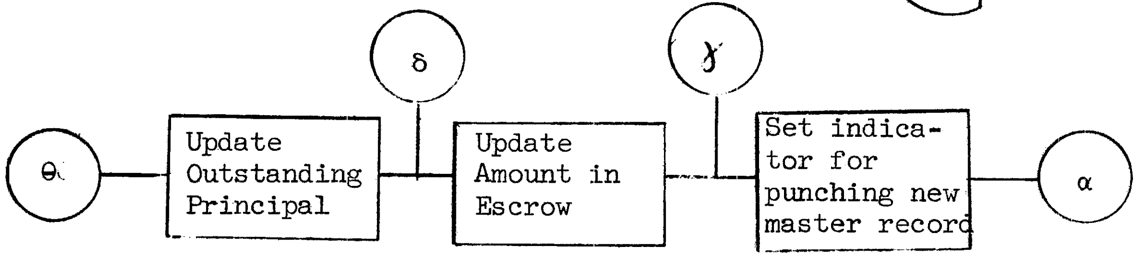
-$=$

to print the character "$" using a C-field specification (see section 2.1.3, Alphabetic Constants).

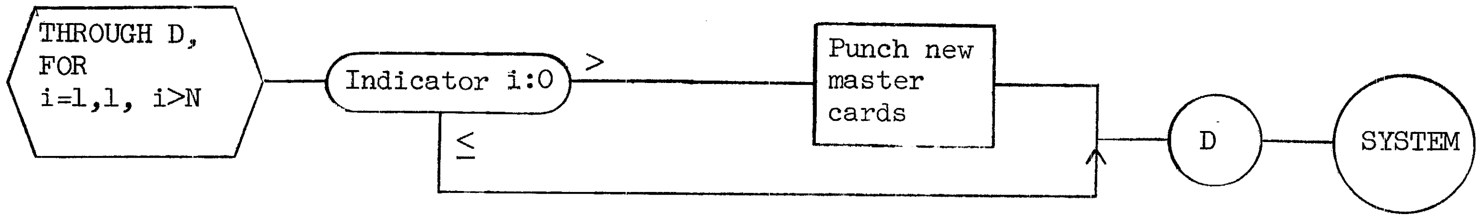
Note 2: The current payments are processed until the file is exhausted. The detection of the end-of-file on reading transfers control to the section of the program which punches the new master file.


Start
READ N
THROUGH A, FOR i=1,1,i>N
Read Master File RECORD (i,1)... RECORD (i,6)
A
α
α
READ Current File IDENT,AMOUNT
THROUGH B, FOR i=1,1 i>N
MORTGAGE:IDENT
=
β
>
Connect OUT OF ORDER
α
≠
<
B
Connect NO MASTER RECORD
α
β
Compute Single Payment DUE1
AMOUNT:DUE1
≠
Compute Double Payment DUE2
=
Update Outstanding Principal and Amount in Escrow
γ
AMOUNT:DUE2
>
ε
=
<
ψ
Update Outstanding Principal
δ


ε
THROUGH C, FOR PAY=ANNUAL PAYMENT, ANNUAL PAYMENT
Payment O.K.?
Θ
Ψ
C
Print Unsatisfactory Payment
α


Θ
Update Outstanding Principal
δ
Update Amount in Escrow
γ
Set indicator for punching new master record
α

On end of file



```
          DIMENSION RECORD(1400,DIM)
          INTEGER I,NUMB
          VECTOR VALUES DIM = 2,1,7
          VECTOR VALUES DOLLAR = -$=$
START     READ FORMAT SIZE,NUMB
          THROUGH A, FOR I = 1,1,I .G. NUMB
A         READ FORMAT MASTER,RECORD(I,1)...RECORD(I,6)
          EXECUTE SETEOF.(UPDATE)
          I = 1
CARDS     READ FORMAT PAYMT, IDENT,AMOUNT
          THROUGH B, FOR I = I,1,I .G. NUMB
          WHENEVER RECORD(I,1) .E. IDENT
          DUE1 = RECORD(I,3)+RECORD(I,5)+RECORD(I,4)*RECORD(I,2)
          WHENEVER .ABS. (AMOUNT-DUE1) .L. .005
          RECORD (I,2) = RECORD(I,2)-RECORD(I,3)
          RECORD(I,6)=RECORD(I,6)+RECORD(I,5)
          TRANSFER TO CODE
          OTHERWISE
          DUE2 = 2.*DUE1 - RECORD(I,4)*RECORD(I,3)
          END OF CONDITIONAL
          WHENEVER .ABS. (AMOUNT-DUE2) .L. .005
          RECORD(I,2)=RECORD(I,2)-2.*RECORD(I,3)
          TRANSFER TO ESCROW
          OR WHENEVER AMOUNT .G. DUE2
          THROUGH C, FOR PAY = RECORD(I,3),RECORD(I,3),
         1AMOUNT .L. DUE2+PAY
C         WHENEVER .ABS. (AMOUNT-DUE2-PAY) .L. .005, TRANSFER TO
         1PAID
          TRANSFER TO OVRPAY
PAID      RECORD(I,2)=RECORD(I,2)-2.*RECORD(I,3)-PAY
ESCROW    RECORD(I,6)=RECORD(I,6)+2.*RECORD(I,5)
CODE      RECORD(I,7)=1.
          OTHERWISE
OVRPAY    PRINT FORMAT REJECT,IDENT,DOLLAR,AMOUNT
          END OF CONDITIONAL
          OR WHENEVER RECORD(I,1) .G. IDENT
          PRINT FORMAT ORDER,IDENT,DOLLAR,AMOUNT
          OTHERWISE
B         CONTINUE
          I=1
          PRINT FORMAT NONE,IDENT
          END OF CONDITIONAL
          TRANSFER TO CARDS
UPDATE    THROUGH D, FOR I=1,1,I .G. NUMB
D         WHENEVER RECORD(I,7).G. 0.,PUNCH FORMAT MASTER,RECORD(I,1)...
         1RECORD(I,6)
         R
         RFORMAT SPECIFICATIONS
         R
```

```
 VECTOR VALUES SIZE=$I10*$
 VECTOR VALUES MASTER=$F10.0,5F10.2*$
 VECTOR VALUES PAYMT = $F10.0,F10.2*$
 VECTOR VALUES REJECT = $20H0PAYMENT ON MORTGAGE,F10. ,3H,  ,C
11,F10.2,19H IS UNSATISFACTORY.*$
 VECTOR VALUES ORDER = $26H0PAYMENT CARD FOR MORTGAGE,F10.0,3H
1,  C1,F10.2,44H IS OUT OF ORDER OR NO MASTER RECORD EXISTS.*$
 VECTOR VALUES NONE = $41H0NO MASTER RECORD EXISTS FOR MORTGAG
1E NO.,F10.0*$
 END OF PROGRAM
```

Example 4

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

Analysis: Commutation columns, which are very important tools in actuarial problems are generated very easily by means of the formulas given below. The quantities M_x, N_x, and D_x in these formulas occur most often in combination, as in the computation of P_x. Assuming a population of some initial size (at $x = b_0$) (here 1,000,000), ℓ_x is the number living at age x (so that $\ell_{b0} = 1{,}000{,}000$), q_x is the mortality rate, and d_x is the number of deaths at age x. Thus $d_x = q_x \cdot \ell_x$. The quantity D_x is computed by the formula $D_x = \ell_x (1+i)^{-x}$, where i is the interest rate. Another quantity, C_x is given by the formula $C_x = d_x \ (1+i)^{-(x+1)}$. It can be used, for example, to compute the cost of term insurance, since $\frac{C_x}{D_x}$ is the premium for one year term insurance of $1 at age x.

The sums M_x and N_x are obtained by the formulas

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

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

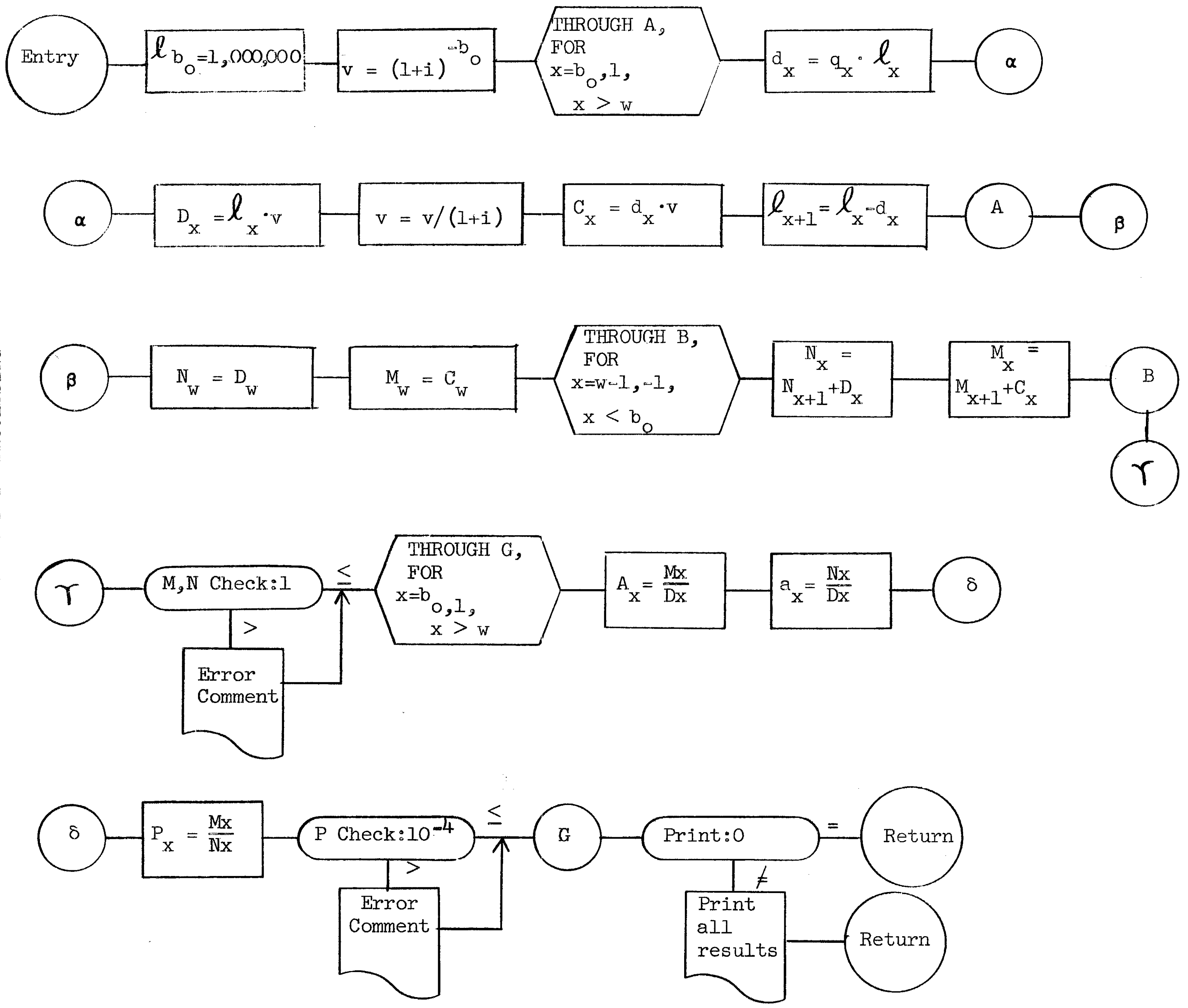
The three most useful quantities computed here are (1) $P_x = M_x/N_x$, which is the annual premium payable for an entire life for $1 of whole life insurance, (2) $A_x = M_x/D_x$, which is the single premium payable at age x for $1 of whole life insurance, and (3) $A_x = N_x/D_x$, which is the present value at age x of a whole life annuity of $1, first payment at age x.

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

$$M_{b_0} + N_{b_0+1} = N_{b_0+1} = N_{b_0}/(1+i)$$

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

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



Entry
ℓ_{b_o}=1,000,000
$v = (1+i)^{-b_o}$
THROUGH A, FOR x=b$_o$,1, x > w
$d_x = q_x \cdot \ell_x$
α
α
$D_x = \ell_x \cdot v$
v = v/(1+i)
$C_x = d_x \cdot v$
$\ell_{x+1} = \ell_x - d_x$
A
β
β
$N_w = D_w$
$M_w = C_w$
THROUGH B, FOR x=w-1,-1, x < b$_o$
$N_x = N_{x+1} + D_x$
$M_x = M_{x+1} + C_x$
B
ϒ
ϒ
M,N Check:1
≤
>
Error Comment
THROUGH G, FOR x=b$_o$,1, x > w
$A_x = \frac{Mx}{Dx}$
$a_x = \frac{Nx}{Dx}$
δ
δ
$P_x = \frac{Mx}{Nx}$
P Check:10^{-4}
≤
>
Error Comment
G
Print:0
=
Return
≠
Print all results
Return

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

The External Function:

```
       RCOMMUTATION TABLE FUNCTION
       RIF PRINT = 0,SUPPRESS PRINTING
        EXTERNAL FUNCTION(Q,BZERO,OMEGA,I,L,SMALLD,BIGD,C,N,
       1M,BIGA,SMALLA,P,PRINT)
        ENTRY TO COMFCN.
        INTEGER BZERO,OMEGA,PRINT,X
        L(BZERO) = 1E6
        V = (1.+I) .P. -BZERO
        THROUGH A,FOR X = BZERO,1,X .G. OMEGA
        SMALLD(X) = Q(X)*L(X)
        BIGD(X) = L(X)*V
        V = V/(1.+I)
        C(X) = SMALLD(X)*V
A       L(X+1) = L(X)-SMALLD(X)
        N(OMEGA) = BIGD(OMEGA)
        M(OMEGA) = C(OMEGA)
        THROUGH B, FOR X = OMEGA-1,-1, X .L. BZERO
        N(X) = N(X+1) + BIGD(X)
B       M(X) = M(X+1) + C(X)
        WHENEVER .ABS.(M(BZERO)+N(BZERO+1)-N(BZERO)/(I+1.)) .G. 1.,
       1TRANSFER TO MNERR
E       THROUGH G ,FOR X = BZERO,1,X .G. OMEGA
        BIGA(X) = M(X)/BIGD(X)
        SMALLA(X) = N(X)/BIGD(X)
        P(X) = M(X)/N(X)
        WHENEVER .ABS. (P(X)-1./SMALLA(X) + I/(I+1.)) .G.
       11E-4,TRANSFER TO PERROR
G       CONTINUE
        WHENEVER PRINT .E. 0,FUNCTION RETURN
       R
       ROUTPUT GENERATOR
       R
        PRINT FORMAT HEAD01,I
        VECTOR VALUES HEAD01 = $1H1,4HI = F5.4//
```

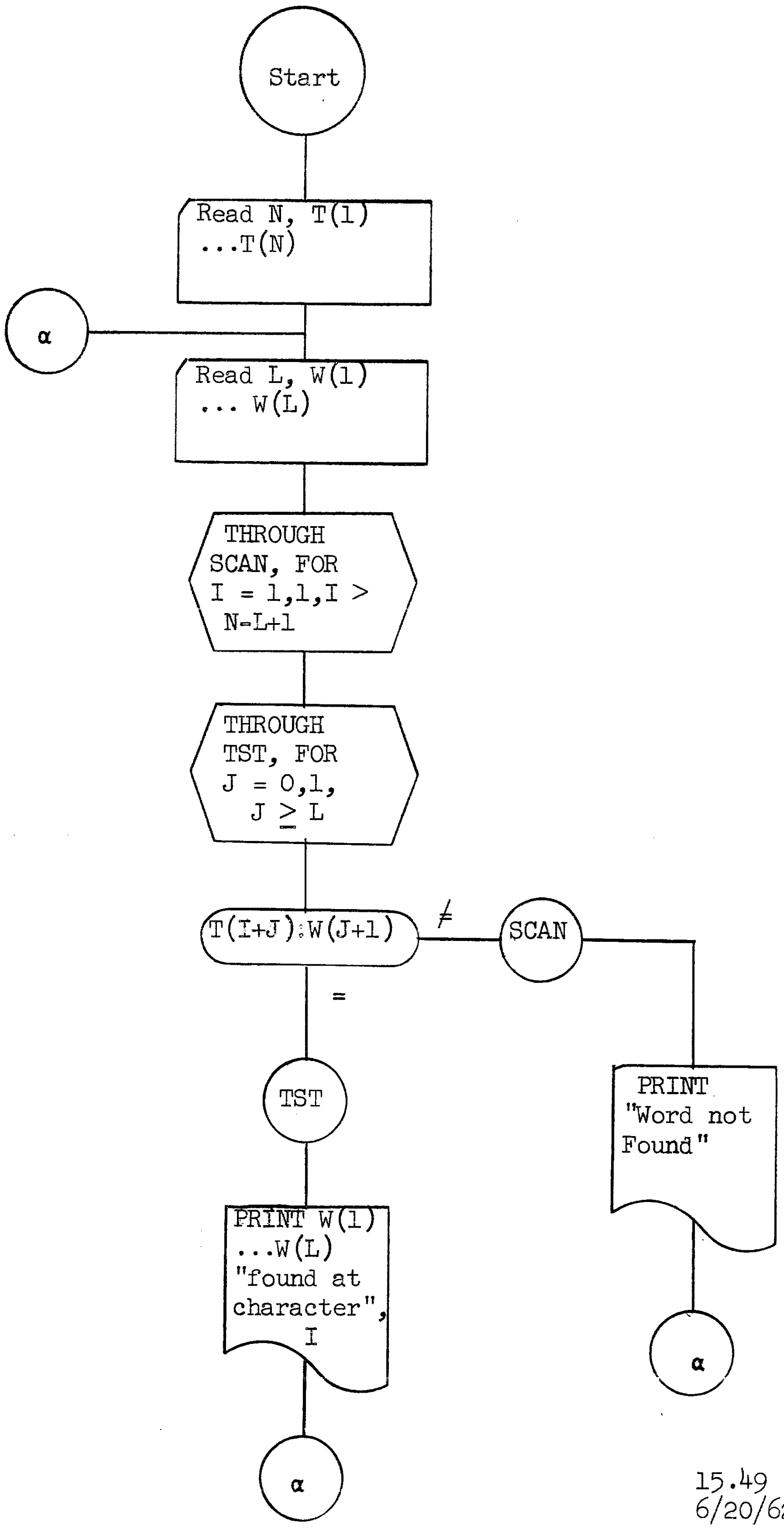
```
          1 4H   X,S13,4HQ(X),S18,4HL(X),S15,10HSMALL D(X),
          2 S8,1HX*$
           THROUGH BETA,FOR X = BZERO,1,X .G. OMEGA
BETA       PRINT FORMAT F1,X,Q(X),L(X),SMALLD(X),X
           VECTOR VALUES F1 = $1H0,I3,3E22.9,I7*$
           PRINT FORMAT HEAD02
           VECTOR VALUES HEAD02 = $1H1,4H   X,S11,8HBIG D(X),
          1 S16,4HC(X),S18,4HM(X),S18,4HN(X),S11,1HX*$
           THROUGH GAMMA,FOR X=BZERO,1,X .G. OMEGA
GAMMA      PRINT FORMAT F2,X,BIGD(X),C(X),M(X),N(X),X
           VECTOR VALUES F2 = $1H0,I4,4E22.9,I7*$
           VECTOR VALUES HEAD03 = $4H1  X,S11,8HBIG A(X),
          1S13,10HSMALL A(X), S15,4HP(X),S11,1HX *$
           THROUGH DELTA,FOR X = BZERO,1,X .G. OMEGA
DELTA      PRINT FORMAT F3,X,BIGA(X),SMALLA(X),P(X),X
           VECTOR VALUES F3 = $1H0,I3,3E22.9,I7*$
           FUNCTION RETURN
PERROR     PRINT FORMAT PERR,P(X),SMALLA(X),I
           VECTOR VALUES PERR = $27H0ERROR ON P CHECK.  P(X) = E18.9,
          1S10,13HSMALL A(X) = E18.9,S10,4HI = F5.4*$
           TRANSFER TO G
MNERR      PRINT FORMAT MNERR1
           VECTOR VALUES MNERR1 = $19H0ERROR ON M,N CHECK*$
           TRANSFER TO E
           END OF FUNCTION

$      DATA
  2258   577   414   338   299   276   261   247   231   212   197   191
   192   198   207   215   219   225   230   237   243   251   259   268
   277   288   299   311   325   340   356   373   392   412   435   459
   486   515   546   581   618   659   703   751   804   861   923   991
  1064  1145  1232  1327  1430  1543  1665  1798  1943  2100  2271  2457
  2659  2878  3118  3376  3658  3964  4296  4656  5046  5470  5930  6427
  6966  7550  8181  8864  9602 10399 11259 12186 13185 14260 15416 16657
 17988 19413 20937 22563 24300 26144 28099 30173 32364 34666 37100 39621
 44719 54826 72467100000
```

15.3 Symbol Manipulation and Recursive Function Examples

Example 1

Problem: Find the first occurrence of an arbitrary word in a given text.

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


Start
Read N, T(1) ...T(N)
α
Read L, W(1) ... W(L)
THROUGH SCAN, FOR I = 1,1,I > N-L+1
THROUGH TST, FOR J = 0,1, J ≥ L
T(I+J):W(J+1)
≠
SCAN
=
TST
PRINT "Word not Found"
PRINT W(1) ...W(L) "found at character", I
α
α

```
          DIMENSION T(720),W(30)
          NORMAL MODE IS INTEGER
          READ FORMAT CNT,N
          READ FORMAT TXT,T(1)...T(N)
ALPHA     READ FORMAT CNT,L
          READ FORMAT TXT,W(1)...W(L)
          THROUGH SCAN, FOR I=1,1,I .G. N-L+1
          THROUGH TST, FOR J=0,1,J .GE. L
TST       WHENEVER T(I+J) .NE. W(J+1), TRANSFER TO SCAN
          PRINT FORMAT OUT,I,W(1)...W(L)
          TRANSFER TO ALPHA
SCAN      CONTINUE
          PRINT FORMAT NOT
          TRANSFER TO ALPHA
          VECTOR VALUES CNT=$I3*$
          VECTOR VALUES TXT=$72C1*$
          VECTOR VALUES OUT=$11HOCHARACTER I3,13H IS START OF 30C1*$
          VECTOR VALUES NOT=$15HOWORD NOT FOUND*$
          END OF PROGRAM
```

SYMBOL MANIPULATION AND RECURSIVE FUNCTION

Example 2

Problem: Evaluate the recursive function,

$$f(0) = 1$$
$$f(n) = f(n-1) * n$$

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

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

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

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

SYMBOL MANIPULATION AND RECURSIVE FUNCTION

Example 3

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

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

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

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

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

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

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

```
         NORMAL MODE IS INTEGER
         SET LIST TO LIST
         DIMENSION LIST (50)
S        READ FORMAT IN,M,N
         SAVE DATA M,N
         PRINT FORMAT OUT,M,N,GCD.(0)
         TRANSFER TO S
         VECTOR VALUES IN = $2I6*$
         VECTOR VALUES OUT = $1H0,3HM= ,I7,S10,3HN= ,I7,S10,5HGCD = I7
        1*$
         END OF PROGRAM
```

Example 4

Problem: To evaluate Tschebychev polynomials.

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

$$T(N,X) = \begin{cases} N = 0 \rightarrow 1 \\ N = 1 \rightarrow X \\ N > 1 \rightarrow 2*X*T(N-1,X)-T(N-2,X) \end{cases}$$

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

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

A Program which uses TSCHEB. is:

```
         SET LIST TO LIST
         DIMENSION LIST(1000)
BEGIN    READ FORMAT INPUT,N,X
         PRINT FORMAT OUTPUT,N,X,TSCHEB.(N,X)
         TRANSFER TO BEGIN
        RFORMATS
         VECTOR VALUES INPUT=$I6,F10.2*$
         VECTOR VALUES OUTPUT=$1H0,4HN=   ,I6,4H X= F10.2,
        111H0FUNCTION= F15.6*$
         END OF PROGRAM
```

20 Summary of MAD Statements

I. Declarations

A. Remark

R in column 11

any remark in columns 12-72

B. Mode

a. $\mathcal{M}$ $U_1, U_2, \ldots, U_n$

where $\mathcal{M}$ is one of:

FLØATING PØINT

INTEGER

BØØLEAN

FUNCTIØN NAME

STATEMENT LABEL

and each U_i is a variable name or function name.

b. NØRMAL MØDE IS $\mathcal{M}$

where $\mathcal{M}$ is one of five listed in a. above.

C. EQUIVALENCE $(V_1, V_2, \ldots, V_n), (V_{n+1}, V_{n+2}, \ldots, V_{n+p}), \ldots,$

$(V_{n+p+q+1}, V_{n+p+q+2}, \ldots, V_{n+p+q+r})$

where each V_i is a variable name or linearly subscripted variable.

D. PRØGRAM CØMMØN $V_1, V_2, \ldots, V_n$

where each V_i is a variable name.

E. ERASABLE $V_1, V_2, \ldots, V_n$

where each V_i is a simple variable or array.

F. DIMENSIØN $V_1(\alpha_1), V_2(\alpha_2), \ldots, V_n(\alpha_n)$

where each V_i is a variable name and each α_i is:

a. for vector DIMENSIØN, α_i is one argument, an integer constant which is the largest value that the subscript of V_i will assume;

b. for matrix DIMENSIØN, α_i represents two arguments; the first is an integer constant, which is the largest value that the subscript of V_i will assume; the second is the name of the first element of a dimension vector for V_i.

G. VECTØR VALUES $\mathcal{V} = c_0, c_1, \ldots, c_n$

or

VECTØR VALUES $\mathcal{V} = \$c_0c, \ldots, c_n\$, \ldots, \$c_{n+p+1}, \ldots, c_{n+p+q}\$$

The two above types of values **may be intermixed.**

H. INTERNAL FUNCTIØN $F.(A_1, A_2, \ldots, A_n) = E$

(single statement internal function definition)

II. Executable Statements

A. Substitution

V = F

B. TRANSFER TØ $\mathcal{S}$

C. Conditional

a. Simple Conditional:

WHENEVER B,Q

b. Compound Conditional:

```
 $\mathcal{S}_1$   WHENEVER $B_1$
          ...
          ...     $\Theta_1$
 $\mathcal{S}_2$   ØR WHENEVER $B_2$
          ...
          ...     $\Theta_2$
*$\mathcal{S}_k$   ØR WHENEVER $B_k$
          ...
          ...     $\Theta_k$
 $\mathcal{S}_{k+1}$ END ØF CØNDITIØNAL
```

* The kth statement may be replaced by:

ØR WHENEVER 1B

or:

ØTHERWISE

SUMMARY OF MAD STATEMENTS

D. CØNTINUE

E. THRØUGH (Iteration)

   a. THRØUGH $\mathcal{S}$, FØR VALUES ØF V = E_1, E_2, ..., E_n

   b. THRØUGH $\mathcal{S}$, FØR V = E_1, E_2, B

F. PAUSE NØ. n

G. EXECUTE C.(A_1, A_2, ..., A_n)

or

EXECUTE C.

H. END ØF PRØGRAM