Systems

BASIC Language Reference Manual

This publication contains a complete description of the BASIC programming language as it is defined by IBM.

BASIC is a terminal-oriented language used by both programmers and non-programmers for the solution of problems requiring lengthy or repetitive computations.

The syntax and semantics of BASIC are presented in this publication for readers who are already acquainted with the fundamental techniques and terminology of programming. Topics covered include program structure, data representation, and statement descriptions. Sample BASIC programs and a formalized definition of the language are included in appendixes.

IBM

Preface

This publication contains a complete description of the BASIC programming language as it is defined by IBM.

BASIC is a terminal-oriented language which was developed for people who wish to solve problems of a mathematical nature, but who need not acquire expertise in the extensive discipline of modern computer programming.

The syntax and semantics of the BASIC language are presented here for the reader who is already acquainted with the fundamental techniques and terminology of programming. The first section of the book describes BASIC program structure and data representation. In the second section, the BASIC statements are grouped according to general function (data definition, input/output, etc.) and presented with examples of their use.

Sample BASIC programs and a formalized definition of the language are included in appendixes.

FIRST EDITION (June 1970)

Contents

Introduction ... 5

Syntax Conventions ... 7

Part 1: BASIC Program Structure

Statements ... 11
Statement Numbers ... 11
Statement Lines ... 11

The BASIC Character Set ... 13
Alphabetic Characters ... 13
Numeric Characters ... 13
Special Characters ... 13
Use of Blanks ... 14

Data Representation ... 15
Arithmetic Data ... 15
Magnitude ... 15
Arithmetic Precision ... 15
Arithmetic Data Formats ... 15
Integer ... 15
Fixed-point ... 15
Floating-point ... 15
Arithmetic Values ... 16
Arithmetic Constants ... 16
Internal Constants ... 17
Arithmetic Variables ... 17
Character Data ... 17
Character Constants ... 17
Character Variables ... 18
Arrays ... 18
Arithmetic Arrays ... 19
Character Arrays ... 19
Naming Conventions for Variables and Arrays ... 20
Functions ... 20
Expressions ... 21
Arithmetic Expressions and Operators ... 21
Priority of Operators ... 22
Character Expressions ... 23
Relational Expressions ... 23

Part II: BASIC Statements

BASIC Statements ... 27

The Assignment Statement ... 29

Descriptive Statements ... 31
The DEF Statement ... 31
The DIM Statement ... 32

Control Statements ... 35
The GOTO Statement ... 35
The IF Statement ... 36
The FOR and NEXT Statements ... 36
The GOSUB and RETURN Statements ... 38
The REM Statement ... 39
The PAUSE Statement ... 39
The STOP Statement ... 40
The END Statement ... 40

Input/Output Statements ... 43
The DATA Statement ... 43
The READ Statement ... 44
The RESTORE Statement ... 44
The INPUT Statement ... 45
The PRINT Statement ... 46
The PRINT USING and Image Statements ... 49
The PUT Statement ... 52
The GET Statement ... 53
The RESET Statement ... 54
The CLOSE Statement ... 54

Matrix Operations ... 57
Redimensioning Arithmetic Arrays ... 57
The MAT Assignment Statement ... 58
Simple ... 58
Addition and Subtraction ... 59
Multiplication ... 59
Scalar Multiplication ... 60
Inversion Function ... 61
Transpose Function ... 62
Identity Function ... 62
ZER Function ... 63
CON Function ... 63
MAT READ Statement ... 64
MAT INPUT Statement ... 65
MAT PRINT Statement ... 66
MAT PRINT USING Statement ... 67
MAT GET Statement ... 68
MAT PUT Statement ... 69

Appendixes

Appendix A: Implementation-defined Restrictions ... 71

Appendix B: Collating Sequence of the BASIC Character Set ... 73

Appendix C: Sample BASIC Programs ... 75

Appendix D: Formalized Definition of the BASIC Language ... 77

Illustrations

Table 1. Naming Conventions for Variables and Arrays ... 20
Table 2. Intrinsic Functions ... 20
Table 3. Packed Print Zone Lengths for Arithmetic Expressions ... 47
Table 4. Carriage Position in PRINT Statement ... 48

Introduction

The BASIC programming language implemented by IBM is based on the BASIC language developed at Dartmouth College as a tool for teaching the fundamentals of programming. As the use of terminals has increased, more and more students, as well as scientists and engineers, have found BASIC to be a relatively simple yet powerful tool for the solution of problems requiring lengthy or repetitive computations. Because BASIC programs and data are usually entered line by line from an interactive terminal, debugging is facilitated, and results are available almost immediately.

Despite its comparative simplicity, BASIC provides many of the convenient features usually associated with more extensive higher-level languages. For example, both arithmetic and character data may be manipulated in BASIC. Numbers can be expressed in any of three formats (integer, fixed-point, and floating-point), and the BASIC user may choose the level of arithmetic precision with which the numbers are handled.

Data items may be treated individually or grouped together to form arrays of either one or two dimensions. The BASIC language provides extensive facilities for performing matrix mathematics, including matrix inversion and multiplication.

As a built-in feature of the language, BASIC has the ability to compute a number of such common mathematical and trigonometric functions as square roots, logarithms, sines, and hyperbolic sines. In addition, the user is given the means to define and name his own often-needed functions through the use of the DEF statement.

BASIC programs consist of statements that specify such operations as assigning values to variables, defining functions and array sizes, and controlling the flow of program execution. Loops, conditional and unconditional transfers of control, and subroutines may all be controlled by BASIC statements. In addition, BASIC statements direct the transmission of data between programs and terminals, and between programs and on-line data files.

At the terminal, the BASIC language is used in conjunction with a command language which consists of instructions for such procedures as beginning and ending sessions at the terminal, running and saving programs, and listing or modifying the contents of program libraries. Command languages vary from system to system, and a detailed description of them does not fall within the scope of this book. The BASIC language, however, is essentially the same in all IBM implementations. Those aspects of it that may vary among IBM implementations, such as arithmetic precision, will be noted in the text. A special appendix, listing all such aspects, appears in the back of this book.

Syntax Notation Conventions

The syntax notation conventions used to illustrate the definitions in this document are:

a. Upper-case letters, digits, and special characters represent information that must appear exactly as shown.
b. Lower-case letters represent information that must be supplied by the user.
c. Information contained within brackets [] represents an option that can be omitted.
d. The appearance of braces {} indicates that a choice must be made between the items contained in the braces.
e. The appearance of the vertical bar | indicates that a choice must be made between the item to the left of the bar and the item to the right of the bar.
f. An ellipsis (a series of three periods) indicates that the preceding syntactical unit may be used one or more times in succession.
g. A list whose length is variable is specified by the format x_1, x_2, . . . , x_n. This format indicates that a variable number of items may be specified, but that at least one is required (commas must separate the items).
h. The appearance of one or more items in sequence indicates that the items (or their replacements) must also appear in the specified order.

Part I: BASIC Program Structure

Statements

A BASIC program consists of a group of numbered statements. The statements may be of either the executable or non-executable type.

Executable statements are those which specify a program action such as assigning a value to a variable (the LET statement), printing a value at the terminal (the PRINT statement), or directing the order of program flow (the GOTO statement).

Non-executable statements are those that specify information which is necessary for program execution. The DATA statement, which provides values to be used, and the DIM statement, which specifies the size of data arrays, are typical non-executable statements.

Executable and non-executable statements may be intermixed when a BASIC program is entered at the terminal. The maximum number of statements permitted in a single BASIC program varies among IBM implementations of the language.

Statement Numbers

Each statement in a BASIC program must be preceded by a statement number. The statement number determines the position the statement will occupy in the program when it is compiled. BASIC statements are executed in numerical order, regardless of the order in which they are entered at the terminal, unless, of course, the normal order of execution is altered by branches, loops, or subroutines.

The maximum number of digits permitted in a statement number varies among IBM implementations of BASIC, but no implementation permits blank characters before or within a statement number.

Statement Lines

A BASIC statement and its statement number together are called a statement line. At the terminal, statement lines are entered one per line; they must not be split between lines, nor can there be more than one statement line per print line. A typical statement line appears as follows:

```
10   LET X = 2*X + Y
```

Statement Number — BASIC Statement

The BASIC Character Set

The characters which have syntactic meaning in BASIC fall into three categories: alphabetic, numeric, and special characters. All elements that make up a BASIC program are constructed from characters in these three categories, with the exception of comments and character constants, either of which may contain any character permitted by the machine configuration on which the BASIC program is processed.

The BASIC character set is arranged according to the Extended Binary Coded Decimal Interchange Code (EBCDIC) collating sequence (See Appendix B).

Alphabetic Characters

The alphabetic characters in BASIC are the upper- and lower-case letters of the standard English alphabet (A-Z and a-z) and the following three characters, called the "alphabet extenders":

@ (the commercial "at" sign)
# (the number or pound sign)
$ (the currency symbol)

Corresponding upper- and lower-case letters of the standard alphabet are evaluated identically and may be used interchangeably; however, the characters on the same keys as the alphabetic extenders are treated differently from the extenders and may not be used in their places. Thus, the symbols A2 and a2 are equivalent, while the symbols $$ and 4$ (where the digit 4 is the lower-case character corresponding to the upper-case $) are not equivalent.

When BASIC is used with languages other than English, the three alphabet extenders can be used to cause printing of letters that are not in the standard English alphabet. In such instances, the EBCDIC representation of the added character is the same as that of the alphabet extender which it replaces in printing.

Numeric Characters

The numeric characters in BASIC are the digits 0 through 9.

Special Characters

There are twenty-four special characters in BASIC. They are:

CHARACTER	NAME
	Blank
=	Equal sign or assignment symbol
+	Plus sign
−	Minus sign
*	Asterisk or multiplication symbol
/	Slash or division symbol
↑	Up-arrow or exponentiation symbol
(	Left parenthesis
)	Right parenthesis
,	Comma
.	Point or period

CHARACTER	NAME
'	Single quotation mark
"	Double quotation mark
;	Semicolon
:	Colon
!	Exclamation symbol
&	Ampersand
?	Question mark
>	Greater than symbol
<	Less than symbol
≠	Not equal symbol
≦	Less than or equal to symbol
≧	Greater than or equal to symbol
|	OR sign or vertical bar

Certain special characters may be combined to produce other syntactic forms in BASIC, of which the following combinations are examples:

SYMBOL	MEANING
>=	Greater than or equal to
<=	Less than or equal to
<>	Not equal
**	Exponentiation

Use of Blanks

Blanks may be used freely throughout a BASIC program to improve readability. They have no syntactic meaning except within character constants and in the image specification statement, which specifies the format of printed output. Thus, the following statements are all evaluated identically; that is, the integer value twenty-five is assigned to an arithmetic variable named A2:

```
LET A2 = 25
LETA2=25
LET A 2 = 2 5
L ET A2 = 25
```

Data Representation

The BASIC character set can be used to represent either arithmetic or character data.

Arithmetic Data

Arithmetic data items are simply those having a numeric value. All numbers in BASIC are expressed to the base ten; that is, they are treated as decimal numbers.

Magnitude

The magnitude of a number is its absolute value. The maximum magnitude permitted in BASIC programs varies among IBM implementations of the language.

Arithmetic Precision

In BASIC, the precision of an integer or fixed-point number is the maximum number of digits it may contain. The precision of a floating-point number is the number of digits in the number to the left of the E (see format descriptions below).

Every IBM implementation of BASIC supports two levels of arithmetic precision, designated "short form" and "long form." The minimum level of precision for the two forms is six significant decimal digits and eleven significant decimal digits, respectively.

The BASIC user specifies the level of precision (long or short form) under which his program is to be run by a command language statement entered when the program is executed.

Arithmetic Data Formats

Arithmetic data may be entered or printed in any of three formats: integer, fixed-point, or floating-point. The appropriate format for a given number depends on its magnitude and the level of arithmetic precision required by the user.

Numbers in any format may be either positive or negative. Negative numbers must be preceded by a minus sign. A plus sign before positive numbers is optional; when no sign is specified, the number is treated as a positive number. The three formats are defined as follows:

Integer Format

Numbers expressed in fixed-point format (F-format) are written as an optional sign followed by a number of digits.

Examples of numbers in integer format are:

0 +2 −23 2683

Fixed-point Format

Numbers expressed in fixed-point format (F-format) are written as an optional sign, optionally followed by a number of digits, followed by a decimal point, followed by a number of digits which are required if no digit precedes the decimal point.

Examples of numbers in fixed-point format are:

33. 33.00 −.3 +3.56

Floating-point Format

Numbers expressed in floating-point format (E-format) are written as an integer or fixed-point number followed by the letter E and an optionally signed one- or two-digit characteristic (exponent).

The value of a floating-point number is equal to the number to the left of the E, multiplied by ten to the power represented by the number to the right of the E. This notation corresponds to standard scientific notation in which numbers are expressed as powers of ten; however, while the number 10^7 is permissible in scientific notation, the number E7 is not a valid floating-point number. The value 10^7 must be expressed as 1E7 in BASIC floating-point format.

Examples of numbers in floating-point format are:

FLOATING-POINT NUMBER	EQUIVALENT DECIMAL VALUE
.25E−4	.000025
+1.0E+5	100000
5E−7	.0000005
−15.33E6	−15330000

Arithmetic Values

Arithmetic values may be entered at the terminal in whichever format the BASIC user finds most convenient. The number one million, for example, could be entered in any of the following ways:

```
1000000
1000000.00
1 E + 6
```

The numeric size of arithmetic values is limited only by the magnitude supported by the implementation of BASIC under which a program is run. The physical length of values entered, however, is also subject to limitation by the implementation, so that some very large and very small numbers may be entered only in E-format. For example, with an implementation that supported a magnitude of 10^{75}, but which limited the number of digits in each value entered to sixteen, the BASIC user could enter the E-format value:

```
1.00000 E+20
```

but not the equivalent I- or F-format number, a one followed by twenty zeros.

The formats of arithmetic values printed at the terminal as output are determined only by the action of the PRINT statement, and not by the form in which the values were originally entered. The BASIC user may control the format of output values, if he wishes, through the PRINT USING and Image statements.

Arithmetic Constants

An arithmetic constant in a BASIC program is either an integer, fixed-point, or floating-point number whose value is never altered during execution of the program. Thus, the integer one is a constant in the following statement:

```
LET X = X + 1
```

In the example below, the integers eleven and four are each constants, but the number 11/4 is not a constant since it is not expressed according to one of the three formats recognized in BASIC:

```
LET X = X + 11/4
```

When this statement is executed, the number 11/4 will be converted to its decimal equivalent before being added to the current value of the variable X.

Internal Constants

An internal constant is an arithmetic constant whose value is pre-defined by the BASIC language processor. Unlike normal arithmetic constants, the internal constants are referred to by names, though like normal constants their values are never altered during program execution. The three internal constants are:

CONSTANT	NAME	MINIMUM SHORT FORM VALUE	MINIMUM LONG FORM VALUE
π	&PI	3.14159	3.1414926536
Natural Log	&E	2.71828	2.7182818285
Square Root of Two	&SQR2	1.41421	1.4142135624

Arithmetic Variables

A variable is a named data item whose value is subject to change during execution of the program.

Arithmetic variables are named by a single letter of the extended alphabet or by a letter of the extended alphabet followed by a single digit. Examples of such names are: #, #3, A, and A2. As stated in the section "Alphabetic Characters," the variables A and A2 can also be referred to by the symbols a and a2.

When a BASIC program is executed, the initial value of all arithmetic variables is set to zero.

Character Data

Character data in BASIC is any data not having a numeric value. Like arithmetic data, character data may be handled in the form of constants or variables.

Character Constants

A character constant is a string of characters enclosed in a pair of single or double quotation marks. Any EBCDIC character may appear in a character constant, including digits and characters which are not part of the BASIC character set. Thus, the following are all valid character constants:

```
"ABCDEF"
'1234567'
"a%%345"
'A B C'
```

The length of a character constant is defined as the total number of characters it contains, including blanks, but excluding the delimiting quotation marks. The maximum number of characters permitted in a single character constant varies among IBM implementations of the BASIC language.

If a character constant bounded by a pair of single quotation marks is to contain a single quotation mark as part of the constant itself, two consecutive single quotation marks must be entered to represent the single quotation mark to be contained in the constant. Unless this procedure is followed, the contained quotation mark will be recognized as the end of the constant. The same procedure is required for character constants containing double quotation marks which are bounded by double quotation marks. The following are some examples of how quotation marks are handled in BASIC character constants:

FORM ENTERED	ACTUAL CONSTANT VALUE	LENGTH
`'its'`	its	3
`"its"`	its	3
`"it's"`	it's	4
`'it"s'`	it's	4
`'"its"'`	"its"	5
`""""its""""`	"its"	5
`'"it"s"'`	"it's"	6

Character Variables

A character variable is a named item of character data whose value is subject to change during execution of the program.

Character variables are named by a single letter of the extended alphabet followed by the currency symbol ($). Examples of such names are: A$ and $$. As stated under the heading "Alphabetic Characters," the variable A$ can also be referred to by the symbol a$.

When a BASIC program is executed, the initial value of all character variables is set to eighteen blank characters.

When character constants are assigned to character variables, the values of the constants are adjusted to a length of eighteen. Longer constants are truncated on the right, and shorter ones are left-justified and padded to the right with blanks.

Arrays

Data items of the same type (arithmetic or character) may be grouped together to form an array. An array is a collection of such data items that is referred to by a single name.

BASIC arrays may be either one- or two-dimensional. A one-dimensional array can be thought of as a row of successive data items. A two-dimensional array can be thought of as a rectangular matrix of rows and columns. The following illustration shows a schematic representation of both types of array.

ONE-DIMENSIONAL ARRAY NAMED A			
A (1)	A (2)	A (3)	A (4)

TWO-DIMENSIONAL ARRAY NAMED B		
B (1, 1)	B (1, 2)	B (1, 3)
B (2, 1)	B (2, 2)	B (2, 3)
B (3, 1)	B (3, 2)	B (3, 3)
B (4, 1)	B (4, 2)	B (4, 3)

Each data item in an array is referred to by the name of the array followed by a subscript in parentheses which indicates its position within the array. The general form for referring to an array member is:

name (e_1 [,e_2])

where *name* is the name of the entire array and e_1 is any positive arithmetic expression whose truncated integer value is greater than zero.

The expression in a subscript referring to a member of a one-dimensional array gives the position of the member in the row, counting from left to right. Thus, the third member of a one-dimensional array named A can be referred to by the symbol A(3), as in this example:

```
LET A(3) = 25
```

The first expression in a subscript referring to a member of a two-dimensional array gives the number of the row containing the referenced member. Rows are numbered from top to bottom. The second expression in the subscript gives the number of the column containing the referenced member. Columns are numbered from left to right. Thus, the second member in the fourth row of a two-dimensional array named B can be referred to by the symbol, B(4,2), as in this example:

```
LET B(4,2) =1.53E6
```

The number of dimensions in an array, and the number of data items in each dimension, is established when the array is declared. In BASIC, arrays may be

declared either explicitly, by use of the DIM statement, or implicitly, by a reference to a member of an array that has not been explicitly declared.

When an array is declared explicitly, the number of dimensions and the maximum number of data items which can be contained in each dimension is specified by the BASIC user through the DIM statement (see section 2, "BASIC Statements").

When an array is declared implicitly, by a reference to a member of it without its name having appeared in a prior DIM statement, it will have the number of dimensions specified in the reference, and each dimension will be able to contain a maximum of ten data items. For example, when no prior DIM statement exists for an array named A, the statement

```
LET A(3) = 50
```

will establish a one-dimensional array containing ten data items, the third of which will have the integer value 50.

Likewise, when no prior DIM statement exists for an array named B, the statement

```
LET B(5,6) = 6.913
```

will establish a two-dimensional array containing ten rows and ten columns (100 items), with the sixth member of the fifth row equal to 6.913.

Arrays containing dimensions of more than ten items may not be implicitly declared. Thus, without appropriate prior DIM statements, the following statements would both result in error conditions:

```
LET A(15) = 22.4
LET B(3,20) = 66.6
```

After an array has been declared, either explicitly or implicitly, it may not be explicitly dimensioned by a DIM statement anywhere in the program. Arithmetic arrays may be redimensioned according to the rules described in the section "Matrix Operations." Character arrays can never be redimensioned.

Arithmetic Arrays

An arithmetic array may contain only arithmetic data and may be of either one or two dimensions.

Arithmetic arrays are named by a single letter of the extended alphabet. Thus, the letter A (or a) may stand for either a single arithmetic variable or an arithmetic array, while the symbol A2 (or a2) may only stand for a single arithmetic variable.

All members of an arithmetic array are initially set to zero when the program is executed.

Before being used in any of the matrix-handling statements provided by BASIC, an arithmetic array must have been previously dimensioned, either explicitly or implicitly. Arithmetic arrays may be redimensioned as described in the section "Matrix Operations."

Character Arrays

A character array may contain only character data and must be only one-dimensional.

Character arrays, like simple character variables, are named by a single letter of the extended alphabet followed by the currency symbol ($). Thus, the name D$ (or d$) may refer to either a simple character variable or a character array.

All elements of a character array are initially set to eighteen blank characters when the program is executed.

Character arrays may not be used in matrix-handling statements and may not be redimensioned.

Naming Conventions for Variables and Arrays

The table below provides a concise review of the names used to refer to variables and arrays in the BASIC language. The symbol "ext" denotes a letter of the extended alphabet.

Table 1. Naming Conventions for Variables and Arrays

DATA TYPE	NAME	EXAMPLES
Arithmetic Variable	ext [digit]	A, a2, $3
Arithmetic Array	ext	A, b, #
Character Variable	ext$	A$, c$, @$
Character Array		

Functions

Often a BASIC user finds it necessary to compute the same mathematical function of many different values during the course of a program. Rather than writing the necessary calculations for each value, he may employ the function capability of the BASIC language.

A BASIC function is a named arithmetic expression that computes a single value from another arithmetic expression. The SIN function, for example, computes the sine of any number of radians. The expression SIN(5), called a "function reference," computes the sine of five radians. Likewise, the function reference SIN(X) represents the sine of the number of radians equal to the value of the variable X.

The value in parentheses following the name of a function is called its "argument." A function reference is thus the name of any function and its argument. Function references may be used anywhere in a BASIC expression that a constant, variable, or array member reference may be used, as illustrated in the following examples:

```
LET A = SIN(&PI) + 1
LET B = SQR (X+3)
LET C = INT (Y) + 3
```

The BASIC language supplies functions that perform a number of common mathematical operations. These are called the "intrinsic functions." In addition, BASIC allows the user to define and name his own frequently used functions through use of the DEF statement (see Part II, "BASIC Statements").

The intrinsic functions provided in all IBM implementations of BASIC are shown in Table 2.

Expressions

An expression in BASIC is any representation of an arithmetic or character value. Constants, variables, arrays, array member references, and function references are all considered expressions. Expressions may also be formed by combining any of these value representations with symbols called "operators."

An operator specifies either the relationship between data items, an arithmetic operation to be performed on them, or whether they are positive or negative. For example, the symbols >, *, and + are operators specifying "greater than," multiplication, and positivity (or addition), respectively.

A special class of expressions, called "relational expressions," is used with the IF statement to test the truth of specified relationships between two values.

Expressions referring to entire arrays, rather than individual array members, are called array expressions and are discussed in the section "Matrix Operations." Any expression which does not contain a reference to an entire array is called a scalar expression.

Table 2. Intrinsic Functions

FUNCTION NAME	DESCRIPTION
ABS (x)	Absolute value of x
ACS (x)	Arccosine (in radians) of x
ASN (x)	Arcsine (in radians) of x
ATN (x)	Arctangent (in radians) of x
COS (x)	Cosine of x radians
COT (x)	Cotangent of x radians
CSC (x)	Cosecant of x radians
DEG (x)	Number of degrees in x radians
DET (x)	Determinant of an arithmetic array
EXP (x)	Natural exponential of x
HCS (x)	Hyperbolic cosine of x radians
HSN (x)	Hyperbolic sine of x radians
HTN (x)	Hyperbolic tangent of x radians
INT (x)	Integral part of x
LGT (x)	Logarithm of x to the base 10
LOG (x)	Logarithm of x to the base e
LTW (x)	Logarithm of x to the base 2
RAD (x)	Number of radians in x degrees
RND [(x)]	Random number between 0 and 1
SEC (x)	Secant of x radians
SGN (x)	Sign of x (−1, 0, or +1)
SIN (x)	Sine of x radians
SQR (x)	Square root of x
TAN (x)	Tangent of x radians

Arithmetic Expressions and Operators

An arithmetic expression may be an arithmetic variable, array member, constant, or function reference; or it may be a series of the above separated by binary operators and parentheses. Some examples of arithmetic expressions are:

```
A1
X3/(-6)
X+Y+Z
SIN(R)
-6.4
-(X-X**2/2+X)
```

The value of an arithmetic expression is obtained by performing the implied operations on the specified data items according to the rules below.

The five binary arithmetic operators are:

SYMBOL	MEANING
** or ↑	Exponentiation (either form of the operator is acceptable)
*	Multiplication
/	Division
+	Addition
−	Subtraction

The two unary operators are:

+	Positive
−	Negative

Special cases for the arithmetic operators and the resulting actions are as follows:
Exponentiation: The expression A↑B or A**B is defined as the variable A raised to the B power.

1. If A=B=0 an error will occur.
2. If A=0 and B<0 an error will occur.
3. If A<0 and B is not an integer, an error of "a negative number to a fractional power" will occur.
4. If A≠0 and B=0, A↑B is evaluated as 1.
5. If A=0 and B>0, A↑B is evaluated as 0.

Multiplication and Addition: A*B and A+B, multiplication and addition respectively, are both commutative, i.e., A*B=B*A and A+B=B+A, but are not always associative due to low-order rounding errors, i.e., A*(B*C) does not necessarily give the same results as (A*B)*C.
Division: A/B is defined as A divided by B. If B=0, an error "division by zero" will occur.
Subtraction: A−B is defined as A minus B. No special conditions exist.
Unary Operators: The + and − signs may also be used as unary operators. Unary operators may be used in only two situations:

1. Following a left parenthesis and preceding an arithmetic expression, or
2. As the leftmost character in an entire expression which is not preceded by an operator.

For example:
−A+(−(B)) and B↑(−2) are valid.
A+−B or B↑−2 is invalid.

Priority of Operators

Arithmetic expressions are evaluated according to the priorities of the operators involved. Operations with the higher priorities are performed first; those at the same priority level are performed from left to right. The levels of priority of the operators are:

Operator	Priority Level
** or ↑	Highest
unary + and −	↓
* and /	
binary + and −	Lowest

An expression is evaluated by being reduced to its component sub-expressions. A sub-expression is defined as a group which can be read "operand-operator-operand," where an operand is either

a. a simple reference to data (constant or variable)
b. or a subscripted array reference
c. or a function reference
d. or a parenthesized sub-expression.

Starting with the first operator to be executed according to the priority scheme above, the operands of its sub-expression are reduced to simple references to data in a left to right order. This process is repeated as many times as required in a left to right and/or descending order of priority of the remaining operators, until the entire expression is reduced to a simple reference to the evaluated result.

The following examples illustrate the successive steps in the evaluation of four arithmetic expressions according to the rules described above. In each expression, the variables A, B, and C have been assigned the integer values 4, 6, and 2 respectively.

Expression	Evaluation and Result

`-A**2+B/C*2.5`	`-4**2+6/2*2.5`
	`-16 +6/2*2.5`
	`-16 + 3 *2.5`
	`-16 + 7.5`
	`-8.5`

`(-A**2)+B/C*2.5`	`(-4**2)+6/2*2.5`
	`-16 +6/2*2.5`
	`-16 +3 *2.5`
	`-16 + 7.5`
	`-8.5`

`-A**(2+B/C)*2.5`	`-4**(2+6/2)*2.5`
	`-4**(2+3) *2.5`
	`-4**5 *2.5`
	`- 1024 *2.5`
	`-1024 *2.5`
	`-2560`

`-A**((2+B)/C)*2.5`	`-4**((2+6)/2)*2.5`
	`-4** (8/2) *2.5`
	`-4**4 *2.5`
	`- 256 *2.5`
	`-256 *2.5`
	`-640`

Character Expressions

A character expression is a character constant, character variable, or single member of a character array. The only operators ever associated with character expressions are the relational operators described below. The following are examples of valid character expressions:

```
A$        "ABC"        'abc'     D$(4)
```

The following are examples of invalid character expressions:

```
A$(4) + "ING"        'STATEMENT'-"MENT"
```

Relational Expressions

A relational expression compares the values of two arithmetic expressions or two character expressions. The expressions to be compared are evaluated and then compared according to the definition of the relational operator specified. According to the result, the relational expression is either satisfied (true) or not satisfied (false). Relational expressions may appear in a BASIC program only as part of an IF statement.

The relational operators and their definitions are:

Operator	Meaning
=	Equal
<> or ≠	Not equal
>= or ≧	Greater than or equal to
<= or ≦	Less than or equal to
>	Greater than
<	Less than

The general format of a relational expression is:

e_1 relational-operator e_2

where e_i is any expression other than an array or relational expression, and *relational-operator* is any of those described above. Both e_1 and e_2 must be of the same data type (character or arithmetic), and only two expressions may be compared in a single relational expression.

When character data appears in a relational expression, it is evaluated according to the EBCDIC collating sequence (see Appendix B) character by character, left to right. Thus, the following relational expressions would all be satisfied:

```
 "ABC" = 'ABC'
'able' < 'BALL'
"Able" > "BALL"
 "123" > "ball"
'$123' < "able"
```

Part II: BASIC Statements

BASIC Statements

This section presents the statements of the BASIC language arranged according to the functions they perform.

Each functional group of statements is introduced by a list of all the statements in it, including a brief description of each statement. The functional groupings have been chosen for purposes of presentation only; they have no fundamental significance in the language. The groupings are as follows:

1. The Assignment Statement—for assigning data from an expression to a variable within a program.
2. Descriptive Statements—for specifying array sizes and for defining functions.
3. Control Statements—for directing the flow of program execution, including loops, unconditional and conditional branches, subroutines, and interruption and termination of programs.
4. Input/Output Statements—for transmitting data to a program for processing and from a program after processing. The Input/Output statements are further divided into those controlling I/O (Input/Output) of internal files, those controlling I/O of interactive terminals, and those controlling I/O of on-line storage devices.
5. Matrix Operation Statements—for computation and I/O involving entire arrays.

The statements within each functional group are presented in the following format:

1. Function—a short description of what the statement does.
2. General Format—a format definition of the syntax of the statement.
3. Action—a description of how the statement works.
4. Rules—a list of rules governing the use of the statement in a BASIC program.
5. Examples—illustrations of how the statement might appear in a BASIC program.

The Assignment Statement

The Assignment statement, which is the only statement in its group, assigns the value of an expression to one or more variables.

The Assignment Statement

Function:

The assignment statement assigns the value of an expression to one or more variables.

General Format:

[LET]v_1 [,v_n] . . . . = *exp*

where v_i is the name of a variable and *exp* is an expression.

Action:

The expression *exp* is evaluated once, and the resulting value is assigned to the specified variable from left to right.

Character constants containing less than eighteen characters are padded on the right with blanks to a length of eighteen before being assigned to character variables. Character constants containing more than eighteen characters are truncated on the right to a length of eighteen before being assigned. Character constants containing no characters (null) are assigned as eighteen blank characters.

Rules:

1. Data values to the right of the equal sign must be of the same type (arithmetic or character) as the variables to which they are assigned.
2. Subscripted references to array members are permitted in the assignment statement, but unsubscripted array references may appear only in the MAT Assignment statement (see the section, "Matrix Operations").
3. The maximum number of variables permitted on the left side of the equal sign in a multiple assignment statement varies among IBM implementations of the BASIC language.

Examples:

```
10  LET Z$ = "CAT"
20  LET X = 9
30  LET Y(X) =2
40  X, Y(X) = X/Y(X)
```

After execution of statement 10, the character variable z$ will contain the word CAT followed by fifteen blank characters.

After execution of statement 20, the arithmetic variable X will have the integer value 9.

After execution of statement 30, the ninth member of the one-dimensional arithmetic array Y will have the integer value 2.

After execution of statement 40, the arithmetic variable X will have the decimal value 4.5, as will the fourth member of the one-dimensional arithmetic array Y. The action of the assignment statement in statement 40 is to first evaluate the expression on the right according to the current values of the variables X and Y(X), 9 and 2, respectively. The resulting value, 4.5, is then assigned to the variable X. The new value of X, 4.5, is then used in the evaluation of the subscript of the

array variable Y(X), for which purpose only the truncated integer portion, 4, is considered. Thus, the fourth member of array Y is set to the expression value 4.5.

If statement 40 had been LET Y(X),X = X/Y(X), the resulting values would have been 4.5 for the *ninth* member of array Y and for the variable X.

Descriptive Statements

The descriptive statements are non-executable statements used for specifying the size of data arrays and for defining user-written functions. User-written functions simplify coding by allowing the BASIC user to write an expression that will calculate the same mathematical function for a number of different values.

The DEF statement is used to define arithmetic functions.

The DIM statement is used to define the size of data arrays.

The DEF Statement

Function:

The DEF statement is a non-executable statement that defines a user-written function.

General Format:

DEF FN*a* (*v*) = *arithmetic expression*

where *a* is any letter of the extended alphabet and *v* is a single arithmetic variable name, called the "dummy variable."

A reference to a user-written function has the general format:

FN*a* (*x*)

where FN*a* is the name of the function and *x* is an arithmetic expression called the "argument."

Action:

When a reference to a user-written function is encountered in an expression at execution time, the current value of the argument (*x*) is substituted for each occurrence, if any, of the dummy variable (*v*) in the arithmetic expression of the corresponding DEF statement. The expression in the DEF statement is then evaluated and the result is assigned as the value of the function reference in the expression in which it appears.

The values of any program variables or array member references that appear in the arithmetic expression of the DEF statement are evaluated at the time of invocation.

Rules:

1. A function may be defined anywhere in a BASIC program, either before or after references to it.
2. A function of a given name may be defined only once in a program.
3. A function definition may not contain references to itself, nor to other functions which refer to it in their definitions.
4. A function reference to a user-written function may appear anywhere in a BASIC expression that a constant, variable, array member reference, or intrinsic function reference may appear.
5. The dummy variable (*v*) has meaning only in the DEF statement. Consequently, it is possible to have a dummy variable with the same name as a simple arithmetic variable used elsewhere in the program. The BASIC language will recognize each as a unique identifier, and no conflict of names or values will result from this duplicate usage.

6. The maximum number of user-written functions permitted in a BASIC program varies among IBM implementations of the language, as does the maximum number of nested function references in each expression, e.g., X=FNA (FNB(FNC (Z))).

Examples:

After execution of the following series of BASIC statements, the variable Z will have the integer value 500.

```
10  LET Y = 10
20  DEF FNA (X) = X**3/2
30  LET Z = FNA(Y)
```

In the following example, the variable R will have the integer value 8 after execution of statement 60. The argument (10) in statement 60 will have no effect on the value of R since it will be substituted for each occurrence of the dummy variable (X) in the arithmetic expression of the DEF statement, and, in this example, X does not appear there.

```
40  LET Y = 2
50  DEF FNZ (X) =Y**3
60  LET R = FNZ (10)
```

In the next example, the variable R will have the integer value 72 after execution of statement 80. When statement 80 is executed, the current value of Y, which is 2, is substituted for each occurrence of the dummy variable X in the arithmetic expression of statement 100. Since the function FNC, defined in statement 100, uses the function FNB, in its definition, the value 2 is substituted for each occurrence of X in the arithmetic expression of statement 90. The resulting value, 47, is then substituted for the function reference FNB(X) in statement 100. The current value of Y, 2, is then added to 47, and the resulting value, 49, is substituted for the function reference FNC(Y) in statement 80. This value is added to 23, and the resulting value, 72, is assigned to the variable R.

```
70  LET Y = 2
80  LET R = FNC(Y) + 23
90  DEF FNB(X) = 5*X**2+27
100 DEF FNC(X) = FNB(X) + X
```

The DIM Statement

Function:

The DIM statement is used to specify the size of arrays.

General Format:

DIM *name-1* (r_1 [,c_1]) [,*name-n* (r_n[,c_n])] . . .

where *name-i* is the name of an array, and r_i and c_i are positive integers specifying a dimension.

Action:

A one-dimensional array whose name is specified in a DIM statement is defined as having the number of members represented by the integer r_i.

A two-dimensional array whose name is specified in a DIM statement is defined as having r_i number of rows and c_i number of columns.

If a variable name subscripted by one or two expressions is used in a program without prior definition of an array for that name, an array of the appropriate number of dimensions is automatically defined for that name. One-dimensional arrays thus defined contain ten members; two-dimensional arrays contain ten rows and ten columns.

The initial value of all arithmetic array members is zero. The initial value of all character array members is eighteen blank characters.

Rules:

1. An array name may not appear in a DIM statement if it has been previously defined, either implicitly, or explicitly in a prior DIM statement. (Arithmetic arrays may be redimensioned after definition according to the rules explained in the section, "Matrix Operations".)
2. Arrays of one or two dimensions may be defined in a DIM statement.
3. A character array may have only one dimension.
4. The maximum permissible size of one- and two-dimensional arrays varies among IBM implementations of the BASIC language.

Example:

```
20  DIM Z$ (5), A(4,2)
```

The result of the above statement is:

$$A = \begin{bmatrix} 0 & 0 \\ 0 & 0 \\ 0 & 0 \\ 0 & 0 \end{bmatrix}$$

z$ = Five strings of eighteen blank characters each.

Control Statements

The control statements are used to direct the flow of program execution.

The GOTO statement causes control to be transferred either unconditionally (simple form) or conditionally (computed form).

The IF statement is a conditional branch statement that causes control to be transferred according to the result of the evaluation of a relational expression.

The FOR and NEXT statements together define a loop which may be executed a number of times.

The GOSUB and RETURN statements define a subroutine.

The PAUSE statement causes program execution to be interrupted.

The STOP statement causes the termination of program execution.

The END statement causes the termination of program compilation and execution.

The REM statement allows the user to insert comments in a BASIC program.

The GOTO Statement

Function:

The GOTO statement transfers control, either conditionally or unconditionally, to a specified statement.

General Format:

The GOTO statement may be written in either of two forms, simple or computed.

Simple:

GOTO s

Computed:

GOTO s_1 [,s_n] . . . ON x

where s_i is a statement number and x is an arithmetic expression.

Action:

Execution of a simple GOTO statement causes an unconditional transfer of control to the statement whose number is specified.

Execution of a computed GOTO statement causes the arithmetic expression (x) to be evaluated and control to be transferred to the statement whose numerical position in the list of statement numbers (read left to right) is equal to the truncated integer value of the expression. Thus, an expression with a value of 2.75 would cause control to be transferred to the second statement on the list. If the expression has a value less than 1 or greater than the total number of statement numbers listed, control "falls through" to the first executable statement following the computed GOTO statement.

When a simple or computed GOTO statement causes control to be transferred to a non-executable statement, control is passed to the first executable statement following the one specified.

Examples:

The following statement will pass control to statement number twenty:

```
100 GOTO 20
```

When X=4, the following statement will pass control to statement number sixty:

```
50 GOTO 40,60,15,100 ON (X+4)/4
```

The IF Statement

Function:

The IF statement causes control to be transferred according to the result of the evaluation of a relational expression.

General Format:

IF e_1 *op* e_2 {THEN|GOTO} s

where e_1 and e_2 are scalar expressions, *op* is a relational operator, and s is the number of the statement to which control is transferred if the expression is satisfied.

Action:

When an IF statement is executed, the two expressions are compared as specified by the relational operator. If the relationship is true, control is transferred to the specified statement number. If the relationship is not true, control is passed to the first executable statement following the IF statement.

Before being compared, a character constant containing less than eighteen characters is blank-padded on the right to a length of eighteen. A character constant containing more than eighteen characters is truncated on the right to a length of eighteen before comparsion. A character constant containing no characters (null) is compared as eighteen blank characters.

All comparisons are made according to the collating sequence of the Extended Binary Coded Decimal Interchange Code (EBCDIC) (see Appendix B).

In the event that the specified relationship is true and the specified statement is non-executable, control is passed to the first executable statement following the specified statement.

Rules:

1. The expressions being compared must contain data of the same type (character or arithmetic).
2. The keywords THEN and GOTO are interchangeable in the IF statement. Either may be used, but not both.

Examples:

```
30  IF A(3) ≠ X+2/Z THEN 85
40  IF R$ > "CAT"  GOTO 70
50  IF S2  = 37.222 THEN 110
```

The FOR and NEXT Statements

Function:

Together, a FOR statement and its paired NEXT statement delimit a "FOR loop"—a set of BASIC statements which may be executed a number of times. The FOR statement marks the beginning of the loop and specifies the conditions of its execution and termination. The NEXT statement marks the end of the loop.

General Format:

FOR $av = x_1$ TO x_2 [STEP x_3]

.
.
.

(BASIC statements)

.
.
.

NEXT av

where av is a simple arithmetic variable called the control variable, x_1 is an arithmetic expression which assigns an initial value to av, x_2 is an arithmetic expression representing the value of av which will cause execution of the loop to be termi-

nated, and x_3 is an arithmetic expression representing the value of the increment to be added to *av* at the end of each execution of the loop. The arithmetic variable *av* must be the same in any given pair of FOR and NEXT statements.

Action:

When the loop is first executed, the control variable (*av*) is set equal to the initial value (x_1). The statements in the loop are executed, and the specified increment (x_3) is added to the control variable (*av*), which is then compared with the specified final value (x_2). If the control variable (*av*) is still less than (greater than for negative increments) or equal to the final value (x_2), the loop is executed and the cycle continues until an increment is made which renders the control variable greater than (less than for negative increments) the specified final value (x_2). At that time, the specified increment value (x_3) is subtracted from the value of the control variable and control "falls through" to the first executable statement following the NEXT statement.

Rules:

1. The value of the control variable (*av*) may be modified by statements within the FOR loop, but its initial value (x_1), final value (x_2), and increment (x_3) are established during the initial execution of the FOR statement and are not affected by any statements within the FOR loop.
2. If the optional STEP expression is omitted in the FOR statement, the increment value is automatically set to +1.
3. If the initial value (x_1) to be assigned to the control variable is greater than (less than for negative increments) the final value (x_2) when the FOR statement is evaluated, the loop is not executed, no value is assigned to the control variable (*av*), and execution proceeds from the first executable statement following the associated NEXT statement.
4. If the STEP option is to be assigned a value which is contradictory to the increment direction implied by the initial and final values (e.g., FOR X = 1 TO 5 STEP − 1), the FOR loop is not executed, no value is assigned to the control variable (*av*), and execution proceeds from the first executable statement following the associated NEXT statement.
5. If the value of the STEP option is zero, the FOR loop is executed an infinite number of times, or until the control variable is set outside of the specified range.
6. Transfer of control into or out of a FOR loop is permitted; however, a NEXT statement may not be executed unless its corresponding FOR statement has been executed previously.
7. FOR loops may be nested within one another as long as the internal FOR loop falls entirely within the external FOR loop (see example). Nested FOR loops may use the same control variable.
8. The maximum number of levels permitted when FOR loops are nested varies among IBM implementations of the BASIC language.

Examples:

The first example shows a simple FOR loop:

```
20  FOR I= 1 TO 25 STEP 2
                .
                .
                .
      (BASIC statements)
                .
                .
                .
95  NEXT I
```

The second example shows the correct technique for nesting FOR loops. The inner loop is executed 100 times for each execution of the outer loop.

```
10  FOR J= A TO B STEP C(1) **3
          .
          .
          .
(BASIC statements)
          .
          .
          .
150 FOR K = 1 TO 100
          .
          .
          .
(BASIC statements)
          .
          .
          .
280  NEXT K
          .
          .
          .
(BASIC statements)
          .
          .
          .
620 NEXT J
```

The GOSUB and RETURN Statements

Function:

The GOSUB statement transfers control to a specified statement. The RETURN statement transfers control to the first executable statement following the last GOSUB statement executed. Together GOSUB and RETURN statements are used in the creation of subroutines.

General Format:

GOSUB *s*
RETURN [*comment*]

where *s* is the number of the statement to which control is to be transferred and *comment* is one or more EBCDIC characters.

Action:

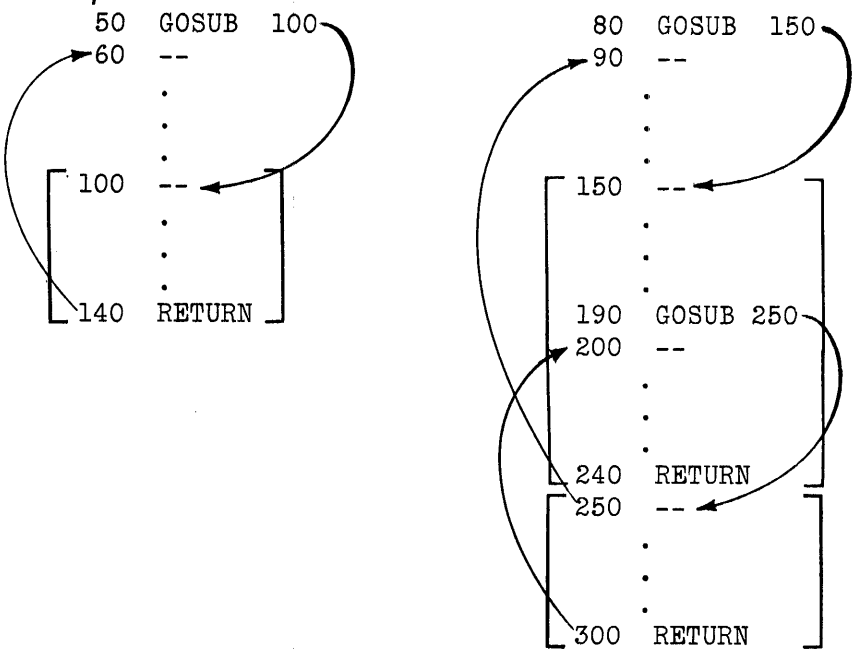
Execution of a GOSUB statement causes an unconditional transfer of control to the statement whose number is specified. If the specified statement is non-executable, control is passed to the first executable statement following the specified statement.

Execution of the RETURN statement causes an unconditional transfer of control to the first executable statement following the last GOSUB statement executed.

Rules:

1. More than one GOSUB statement may be executed before a RETURN statement is executed, but when a RETURN statement is executed, there must be at least one GOSUB statement already executed for which a corresponding RETURN statement has not been executed.
2. The maximum number of levels permitted when subroutines are nested varies among IBM implementations of the BASIC language.

Examples:



The REM Statement

Function:

The REM statement allows the BASIC user to insert comments in the program listing.

General Format:

REM [*comment*]

where *comment* is one or more EBCDIC characters.

Action:

The REM statement is non-executable. It appears in the program listing, but has no effect on program execution.

Rule:

A REM statement may appear anywhere in a BASIC program.

Example:

```
10  REM THIS PROGRAM DETERMINES THE COST PER UNIT
```

The PAUSE Statement

Function:

The PAUSE statement causes program execution to be interrupted.

General Format:

PAUSE [*comment*]

where *comment* is one or more EBCDIC characters.

Action:

When a PAUSE statement is encountered during program execution, execution is interrupted and the following message is printed out at the terminal:

PAUSE AT *s*

where *s* is the number of the PAUSE statement.

When a PAUSE statement is executed immediately after a PRINT or MAT PRINT statement, the message PAUSE AT *s* is printed on the line below the last line of output from the PRINT or MAT PRINT statement, even if the final delimiter of that statement is a comma or semicolon.

Rule:

The procedure for subsequent resumption of program execution after a PAUSE statement varies among IBM implementations of the BASIC language.

Example:

The following statement would cause the message PAUSE AT 80 to be displayed at the terminal and processing to be suspended until the user executed the appropriate resumption procedure.

```
80  PAUSE
```

The STOP Statement

Function:

The STOP statement causes program execution to be terminated.

General Format:

STOP [*comment*]

where *comment* is one or more EBCDIC characters.

Action:

When a STOP statement is executed, program processing is terminated and all open files are automatically closed. Unlike the END statement, which functions identically at execution time, the STOP statement has no effect on the compilation of the program.

Rule:

A STOP statement may appear anywhere in a BASIC program.

Example:

```
75  STOP
```

The END Statement

Function:

The END statement causes the termination both of compilation and of execution of a BASIC program.

General Format:

END [*comment*]

where *comment* is one or more EBCDIC characters.

Action:

The END statement indicates the logical end of a program. When it is encountered during program compilation, it causes any statements which follow it numerically to be excluded from the program. When an END statement is encountered at execution time it causes termination of processing and closing of all open files.

Rule:

The END statement is optional. If omitted by the user, END will be assumed by the system to follow the highest-numbered statement in the program.

Example:

```
999  END
```

Input/Output Statements

The input/output statements are used for transmitting data to a program for processing, and from a program after processing. They are divided into three subgroups:

1. Statements controlling the creation and use of internal files:
   - The DATA statement creates an internal data file.
   - The READ statement assigns values to variables from the internal data file.
   - The RESTORE statement causes the internal data file to be positioned at its beginning.
2. Statements controlling data transmission to and from interactive terminals:
   - The INPUT statement allows the user to assign values to variables from the terminal at execution time.
   - The PRINT statement causes data values to be printed at the terminal.
   - The PRINT USING and Image statements together allow the BASIC user to have data values printed at the terminal in a format of his own choosing.
3. Statements controlling the writing and reading of files stored on on-line storage devices:
   - The PUT statement causes data values to be placed in an external file.
   - The GET statement causes data values to be assigned to variables from an external file.
   - The RESET statement causes a file to be repositioned at its beginning.
   - The CLOSE statement causes external files to be deactivated.

The DATA Statement

Function:

The DATA statement is a non-executable statement that creates an internal data file from which values are supplied to variables in corresponding READ statements.

General Format:

DATA *constant* [, *constant*] . . .

where *constant* is either an arithmetic or character constant.

Action:

Before execution time, a single table is constructed containing all the values from all the DATA statements in the program in their order of appearance by statement number. At the same time, a pointer is set to the first item in the table. The pointer is advanced through the table, item by item, as the data is supplied to READ statement variables.

Rules:

1. Each item of data in a DATA statement must be of the same type as that specified by the variable to which it is to be assigned in the corresponding READ statement. Thus, if the third constant in the DATA statement is a character constant, then the READ statement variable to which it is assigned must be a character variable.
2. DATA statements may be placed anywhere in a BASIC program, either before or after the READ statements to which they supply data.

Example:

```
10  DATA 'JONES', 15.00, 'SMITH',20.50
```

The READ Statement

Function:

The READ statement assigns values to variables from the data table created by DATA statements.

General Format:

READ v_1 [, v_n] . . .

where v_i is a simple arithmetic or character variable, or a subscripted reference to a single array member.

Action:

When a READ statement is executed, successive values from the data table are assigned to the variables in the READ statement from left to right, beginning at the current position of the data table pointer.

Subscripts of array variables in the READ statement are evaluated as they occur; thus, an assigned variable in a READ statement may be used subsequently as the subscript of another variable in the same statement.

Character constants containing less than eighteen characters are padded on the right with blanks to a length of eighteen before being assigned to character variables. Character constants containing more than eighteen characters are truncated on the right to a length of eighteen before being assigned. Character constants containing no characters (null) are assigned as eighteen blank characters.

Rules:

1. Each data value read from the table must be of the same type (character or arithmetic) as the variable to which it is assigned.
2. A READ statement is invalid if there are no DATA statements in the program.
3. If the data table is exhausted and unassigned variables still remain in the READ statement, an error condition results.

Examples:

```
10  DATA 'JONES', 15.00, 'SMITH', 20.50
20  READ A$,A1,B$,B1
30  DATA 1,2,3,4,5,6
40  READ A,B,C,X(A),X(B),X(C)
```

After execution of the above statements, the character variables A$ and B$ will contain the characters strings JONES and SMITH respectively, each padded on the right with blanks to a length of eighteen. The arithmetic variables A1 and B1 will contain the decimal values 15.00 and 20.50, respectively. The arithmetic variables A, B, and C will contain the integer values 1, 2, and 3 respectively, and the first three members of the one-dimensional array named X will contain the integer values 4, 5, and 6, respectively.

The RESTORE Statement

Function:

The RESTORE statement causes the subsequent READ statement to begin assigning values from the first item in the first DATA statement of the program.

General Format:

RESTORE [*comment*]

where *comment* is one or more EBCDIC characters.

Action:

The RESTORE statement returns the data table pointer from its current position to the beginning of the table. The optional comment is a character string that does not effect the execution of the statement.

Rules:

1. A RESTORE statement in a program containing no DATA statements is ignored. No error condition results.
2. A RESTORE statement for an already restored table is ignored.

Example:

After the following series of statements is executed, the variables A and C will each have a value of one, and the variables B and D will each have a value of two.

```
10  DATA 1,2
20  READ A,B
30  RESTORE
40  READ C,D
```

The INPUT Statement

Function:

The INPUT statement allows the BASIC user to assign values to variables from the terminal at execution time.

General Format:

INPUT v_1 [, v_n] . . .

where v_i is a variable reference.

Action:

When an INPUT statement is encountered at execution time, it causes a question mark to be printed out at the terminal and program execution to be temporarily interrupted. The user then enters a list of values which are assigned, in order of appearance, to the variables specified in the INPUT statement. When the complete list has been entered, signalled by a carriage return, program execution resumes.

Subscripts of array variables in the INPUT statement are evaluated as they occur; thus an assigned variable in an INPUT statement may be used subsequently as the subscript of another variable in the same statement.

Character constants entered at the terminal that contain less than eighteen characters are padded to the right with blanks to a length of eighteen before being assigned to character variables. Character constants containing more than eighteen characters are truncated to the right to a length of eighteen before being assigned. Character constants containing no characters (null) are assigned as eighteen blank characters.

Arithmetic data values are assigned to variables in the form (short or long) which has been specified for the program. Thus, long-form values entered at the terminal during execution of a program for which short-form arithmetic has been specified are truncated to the implementation-defined number of significant digits for short form (at least six) before being assigned. Likewise, short-form values entered at the terminal during execution of a program for which long-form arithmetic has been specified are zero-filled to the implementation-defined number of digits for long form (at least eleven) before being assigned.

When an INPUT statement is executed immediately after a PRINT or MAT PRINT statement in which the final delimiter is a comma or semicolon, the question mark generated by the INPUT statement is printed directly following the last data item on the same print line. In all other instances, the question mark appears as the first character on the next print line.

Rules:

1. Each value entered must be of the same data type (character or arithmetic) as the corresponding variable reference in the INPUT statement. Data types may be mixed in the same statement.

2. Each value entered must be separated from the next by a comma. Two consecutive commas are treated as an error. A carriage return ends the series of values entered.
3. Only a single line of input may be entered in response to an INPUT statement.
4. A character constant in the input stream must be bounded by a pair of single or double quotation marks.
5. The number of values entered at execution time must be equal to the number of variable references specified in the INPUT statement.
6. The procedure for retry or re-entry of data after an error varies among IBM implementations of the BASIC language.

Example:

```
10  INPUT A$, R(3), X, Y(X)
?   "DOG", 4E-7, 8, .013
```

The PRINT Statement

Function:

The PRINT statement causes the values of specified scalar expressions to be printed at the terminal. The format of printed values is standardized, but the PRINT statement allows the BASIC user to control the spacing between values on the printed line.

General Format:

PRINT *exp-1* [[,|;] [*exp-n*]] . . .

where *exp-i* is a scalar expression and the comma and semicolon are delimiters which specify the position of the carriage.

As the format description indicates, a comma or semicolon delimiter is optional. A scalar expression in a PRINT statement may be followed by a delimiter of one or more blank characters, or of no characters at all (that is, directly concatenated to the next specified expression). Such specifications are evaluated as a "null" delimiter. Null delimiters may only be used between two expressions when one, and only one, of them is a character constant. Two consecutive character constants, or two expressions of which neither is a character constant, must be separated by either a comma or semicolon delimiter.

Action:

When a PRINT statement is executed, each specified expression value is converted to the appropriate standard output format, as described below, and printed at the terminal in the order in which it appears in the PRINT statement. The carriage is then positioned as specified by the delimiter immediately following the expression.

Standard Output Formats: If the expression value to be printed is a character expression, the actual characters contained or referred to in it are printed at the terminal, with the exception of trailing blanks in character variables or array members.

If the expression value is an arithmetic expression it is converted for printing to one of the following standard output formats. (The letter P in these descriptions denotes the maximum number of digits provided by the implementation for long and short form arithmetic.)

1. I-format (integer) consisting of a sign (blank or minus), and up to P significant decimal digits for integers whose absolute value is equal to or greater than zero, and less than 1E+P. Printed values are rounded off, not truncated.
2. E-format (floating-point), consisting of a sign (blank or minus), up to P significant decimal digits, a decimal point following the first digit, the letter E, and a signed exponent consisting of two digits. E-format is used to print numbers, not included in the I-format described above, whose absolute value is less than 1E−1 or greater than or equal to 1E+P. Printed values are rounded off, not truncated.

3. F-format (fixed-point), consisting of a sign (blank or minus), up to P significant digits, and a decimal point in the appropriate position. F-format is used to print numbers whose absolute values are not included in the I- and E-format descriptions above. Printed values are rounded off, not truncated.

The following examples show how various arithmetic values would be printed in response to a PRINT statement in a program run under short form arithmetic in an IBM implementation of BASIC providing the minimum precision, six significant digits. The symbol b represents a blank character.

VALUE GIVEN	VALUE PRINTED
123	b123
1234567	b1.23457E+06
123.4	b123.400
12345.678	b12345.7
12345.645	b12345.6

Spacing of Printed Values: The converted value of each expression specified in the PRINT statement is printed at the terminal in its own print zone. Print zones may be either "full" or "packed," as specified by the delimiter following the expression, and a single printed line may be made up of values in either or both zone types.

The full print zone, which is specified by a comma, is always eighteen characters in length, measured from the first character of the expression value to be printed, regardless of the length or data type of the expression. Since most printed values are shorter than eighteen characters, a line of full print zones usually produces widely spaced output.

The packed print zone, which is specified by a semicolon or null delimiter, varies in length according to the length and data type of the expression which it contains. Packed zones usually produce a denser line of output than full zones.

If the expression to be printed is a character constant, the length of the packed print zone containing it is equal to the length of the character string itself, including all trailing and embedded blanks, but neither counting nor printing the enclosing single or double quotation marks.

If the expression is a character variable or a member of a character array, the length of the packed print zone containing it is equal to the length of the character string, minus any trailing blanks.

If the expression is arithmetic, the length of the packed print zone containing it is determined by the length of the converted value, including sign, digits, decimal point and exponent, as shown in Table 3. (Note that positive numbers are preceded by a blank character in the sign position, as described in the standard output formats above.)

Table 3. Packed Print Zone Lengths for Arithmetic Expressions

LENGTH OF CONVERTED DATA ITEM	LENGTH OF PACKED PRINT ZONE	EXAMPLE (b represents a blank)
2- 4 characters	6 characters	b17.3b
5- 7 characters	9 characters	b17.357bb
8-10 characters	12 characters	−45.63927bbb
11-13 characters	15 characters	b1.73579E−23bbb
14-17 characters	18 characters	−892270493115663bb

Positioning of the Carriage: The movements of the carriage at the terminal before, during and after the printing of expression values depends on both the type of expression being printed and the delimiter following it in the PRINT statement. Table 4 shows the variety of carriage actions which are possible.

Table 4. Carriage Positions in PRINT Statement

DATA TYPE	DELIMITER	CARRIAGE POSITION FOR PRINTING	CARRIAGE POSITION AFTER PRINTING
Arithmetic Expression	Comma	If the line contains sufficient space to accommodate the value, printing will begin at the current carriage position. If not, printing will start at the beginning of the next line.	The carriage will be moved past any remaining spaces in the full print zone. If the end of the line is encountered the carriage will be moved to the beginning of the next line.
	Semicolon	"	The carriage will be moved past any remaining spaces in the packed print zone. If the end of the line is encountered, the carriage will be moved to the beginning of the next line.
	Null (Not end of statement)	"	The carriage will be left at the print position immediately following the data item.
	Null (End of statement)	"	The carriage will be moved to the beginning of the next line.
Simple Character Variable or Subscripted Character Array Reference	Comma	If at least 18 spaces remain on the line, printing will start at the current carriage position. If less than 18 spaces remain on the line, printing will start at the beginning of the next line.	The carriage will be moved past any remaining spaces in the full print zone. If the end of the line is encountered, the carriage will be moved to the beginning of the next line.
	Semicolon	Printing will start at the current carriage position. If the end of the line is encountered before the data item is exhausted printing of the remaining characters will begin on the next line.	The carriage will be moved past any remaining spaces in the packed print zone. If the end of the line is encountered, the carriage will be moved to the beginning of the next line.
	Null (Not end of statement)	"	The carriage will be left at the print position immediately following the end of the data item.
	Null (End of statement)	"	The carriage will be moved to the beginning of the next line.
Character Constant	Comma	If at least 18 spaces remain on the line, printing will start at the current carriage position. If less than 18 spaces remain on the line, printing will start at the beginning of the next line. If the end of the line is encountered before the character constant is exhausted, printing of the remaining characters will begin on the next line.	The carriage will be moved past any remaining spaces in the full print zone. If the end of the line is encountered, the carriage will be moved to the beginning of the next line.
	Semicolon or Null (Not end of statement)	Printing will start at the current carriage position. If the end of the line is encountered before the character constant is exhausted, printing of the remaining characters will begin on the next line.	The carriage will be left at the print position immediately following the constant.
	Null (End of statement)	"	The carriage will be moved to the beginning of the next line.
Null	Comma	No printing will occur.	The carriage will be moved 18 spaces. If the end of the line is encountered, the carriage will be moved to the beginning of the next line.
	Semicolon	"	The carriage will be moved three spaces. If the end of the line is encountered, the carriage will be moved to the beginning of the next line.
	Null	"	If the null data item is the first item on the list, the carriage will be moved to the beginning of the next line. Otherwise, no movement of the carriage will occur.

Examples:

STATEMENT	PRINTED OUTPUT
`10 PRINT 'A','B'`	`A` –17 blanks– `B`
`20 PRINT 'A';'B'`	`AB`
`30 LET A$="B"`	
`40 PRINT 'A' A$`	`AB`
`50 PRINT A$ 'A',A$;A$`	`BA` –16 blanks– `BB`
`60 PRINT A$;'A'`	`BA`
`70 LET A$ = ''`	
`80 PRINT 'A';A$;'A'`	`AA`

The PRINT USING and Image Statements

Function:

The PRINT USING statement and its associated Image statement allow the BASIC user to have program values printed at the terminal in a format of his own choosing. The PRINT USING statement specifies the values to be printed and the statement number of the Image statement to be used. The Image statement specifies the format of the line to be printed.

General Format:

PRINT USING *statement-number* [*,ref-1 . . . ,ref-n*]
:[[*c . . .*] *format*]. . .

where *statement-number* is the number of the Image statement to be used, and *ref-i* is the name of a single variable or array member to be printed. The symbol for an Image statement is a colon (:) following the statement number. The term *c* represents any EBCDIC character other than the pound sign (#), and the term *format* represents a conversion specification, as described below.

Action:

When a PRINT USING statement is executed, the specified references are evaluated, and their values are edited, in order of appearance in the PRINT USING statement, into the corresponding format specifications in the specified Image statement. The EBCDIC characters represented by *c* in the general format description in the print line are printed exactly as entered in the Image statement itself.

If the carriage is not at the beginning of a new line before printing begins, it is so positioned. After printing, the carriage is repositioned to the beginning of the next line.

Format Specifications: For each occurrence of the pound sign (#) in an Image statement, a single space is reserved in the print line for a character in the corresponding data reference of the associated PRINT USING statement. The pound sign may represent either character or arithmetic data. Decimal points and the plus and minus signs, like the characters represented by *c* in the general format description, are printed as entered, provided the values are appropriate to the specified signs. (See below under "Conversion of Data Reference Values" for a discussion of the printing of signs in the Image statement.)

1. The various format specifications are:
   a. *Character-format*—one or more # characters

      Example: #[#]...

   b. *I-format* (integer format)—an optional sign followed by one or more # characters.

      Example: [+|-]#[#]...

   c. *F-format* (fixed-decimal format)—an optional sign followed by either:
      (1) no #'s, a decimal point, one or more # characters
      (2) one or more #'s, a decimal point, no #'s
      (3) one or more #'s, a decimal point, one or more # characters

      Example: [+|-]{[#]... . #[#]...|#[#]... .[#]...}

   d. *E-format* (exponential format)—either the I- or F-format (given above) followed by four ! characters or four | characters

      Example: {I-format|F-format} {!!!!|||||}

2. The following rules define the start of a format-specification:
   a. A # sign is encountered and the preceding character is not a # sign, decimal point, plus sign, or minus sign.
   b. A plus or minus sign is encountered, which is followed by:
      (1) A # sign or
      (2) A decimal point which is followed by a # sign.
   c. A decimal point is encountered, which is followed by a # sign and:
      (1) The preceding character is not a # sign, plus sign, or minus sign or
      (2) The preceding character string is an F-format specification.
3. The following rules define the end of a format-specification that has been started:
   a. A # sign is encountered and:
      (1) The following character is not a # sign or
      (2) The following character is not a decimal point or
      (3) The following character is a decimal point and a decimal point has already been encountered or
      (4) The following four consecutive characters are not ! or | characters.
   b. A decimal point is encountered and:
      (1) The following character is not a # sign or
      (2) The following character is another decimal point or
      (3) The following four consecutive characters are not ! or | characters.
   c. Four consecutive ! or | characters are encountered.

Conversion of Data Reference Values: When the data referred to is a character value, the characters contained in it are edited into the line in the conversion specification including sign, pound sign, decimal point, and !!!! or ||||.

If an edited character reference is shorter than its format specification, blank padding occurs on the right. If an edited character reference is longer than its format specification, it is truncated on the right. A character constant containing no characters (null) causes blank padding of the entire format specification.

An arithmetic expression is converted in accordance with its format specification as follows:

1. If the format specification contains a plus sign, and the expression value is positive, a plus sign is edited into the line.
2. If the format specification contains a plus sign, and the expression value is negative, a minus sign is edited into the line.
3. If the format specification contains a minus sign, and the expression value is positive, a blank is edited into the line.
4. If the format specification contains a minus sign, and the expression value is negative, a minus sign is edited into the line.
5. If the format specification does not contain a sign, and the expression value is negative, the negative number will be printed with a minus sign provided that the format specification is long enough to contain both the number and the sign. If the format specification is not long enough, asterisks are edited into the line instead of the negative expression value.
6. The expression value is converted according to the type of its format specification as follows:

   I-format: The value of the expression is converted to an integer, rounding any fraction.

   F-format: The value of the expression is converted to a fixed-point number, rounding the value or extending it with zeros in accordance with the format specification.

   E-format: The value of the expression is converted to a floating-point number, rounding the value or extending it with zeros in accordance with the format specification.

If the length of the arithmetic expression value is less than or equal to the length of the format specification, the expression value is edited, right-justified, into the line. If the length of the format specification is less than the length of the expression value, asterisks are edited into the line instead of the expression value.

Some examples of reference values and the way they are printed under various format specifications are as follows:

Format Specification	Reference Value	Printed Form
###	123	123
###	12	b12
###	1.23	bb1
##.##	123	*****
##.##	1.23	b1.23
##.##	1.23456	b1.23
##.##	.123	bb.12
##.##	12.345	12.35
###!!!!	123	123E+00
###!!!!	12.3	123E−01
###!!!!	.1234	123E−03
##.##!!!!	123	12.30E+01
##.##!!!!	1.23	b1.23E+00
##.##!!!!	.1234	12.34E−02
##.##!!!!	1234	12.34E+02

Rules:

1. The maximum number of Image statements permitted in a single BASIC program varies among IBM implementations of the language.
2. If the PRINT USING statement contains at least one data reference, and no format specification appears in the corresponding Image statement, an error condition results.
3. If the number of data references in the PRINT USING statement exceeds the number of format specifications in the corresponding Image statement, a carriage return occurs at the end of the Image statement and the Image statement is reused for the remaining data references.
4. If the number of data references in the PRINT USING statement is less than the number of format specifications in the corresponding Image statement, the print line is terminated at the first unused format specification.
5. Image statements are non-executable and may be placed anywhere in a BASIC program, either before or after the PRINT USING statements that refer to them.

Examples:

```
30 PRINT USING 40, A,B
40 :RATE OF LOSS #### EQUALS ####.## POUNDS
```

Printed Output:

```
RATE OF LOSS  342 EQUALS   42.02 POUNDS
               Value        Value
               of A         of B
```

The PUT Statement

Function:

The PUT statement causes data values to be placed in a specified file.

General Format:

PUT *file-reference*, v_1 [,v_n] . . .

where *file-reference* is a character constant enclosed in single or double quotation marks, and v_i is a constant, simple variable, function reference, or subscripted array reference.

Action:

When a PUT statement is executed, the specified scalar reference values are entered from left to right in the specified file, beginning at the current file position. The file is arranged sequentially so that the first value entered in response to a PUT statement will be the first value assigned from the file when it is used for program input.

All arithmetic data is truncated or zero-filled, as necessary, to conform to the form of arithmetic (long or short) specified for the program before the values are placed in files.

A file is activated for output by the first execution of a PUT (or MAT PUT) statement using its file name. At the time, the file is positioned at the beginning before scalar references are written into it.

A file is deactivated in response to a CLOSE statement, or at the end of execution of a program.

Rules:

1. A file currently activated as an input file may not be specified in a PUT statement. It must first be closed.
2. The maximum number of data items permitted in each file varies among IBM implementationss of the BASIC language.
3. Naming conventions for on-line data files vary among IBM implementations of the BASIC language.

Example:

```
30 PUT "ABF", Z3, 5*X-7, A, D$, 9.005
```

The GET Statement

Function:

The GET statement causes values to be assigned to variables from a specified file.

General Format:

GET *file-reference*, v_1[,v_n] . . .

where *file-reference* is a character constant enclosed by single or double quotation marks, and v_i is either a simple variable or a subscripted reference to an array member.

Action:

A file is activated for input by the first execution of a GET (or MAT GET) statement specifying its file name, at which time the file is positioned at the beginning and values are assigned from it to the variables specified in the GET statement. Subsequent GET statements for the same file cause values to be assigned beginning at the current file position.

Subscripts in variable references are evaluated as they occur, from left to right. Thus, an assigned variable in a GET statement may be used subsequently as the subscript of another variable in the same GET statement.

Arithmetic values are assigned in the form (long or short) specified for the program in which the GET statement appears. Thus, arithmetic values from a file which was created in long form are truncated to the implementation-defined number of significant digits for short form (at least six) before being assigned in a program for which short-form arithmetic has been specified. Likewise, arithmetic values from a file which was created in short form are zero-filled to the implementation-defined number of digits for long form (at least eleven), before being assigned in a program for which long-form arithmetic has been specified.

A file is deactivated in response to a CLOSE statement or at the end of program execution.

Rules:

1. A file currently activated as an output file may not be be specified in a GET statement. It must first be closed.

2. Each value assigned in a GET statement must be of the same data type (character or arithmetic) as its corresponding variable.
3. Naming conventions for on-line data files vary among IBM implementations of the BASIC language.

Example:

```
70 GET "ABF", X,Y,Z,A(4), A(5), D$, E$
```

The RESET Statement

Function:

The RESET statement causes an input or output file to be positioned to the beginning.

General Format:

RESET *file-reference-1* [,*file-reference-n*] . . .

where *file-reference-i* is a literal constant enclosed in single or double quotes.

Action:

Execution of a RESET statement for a specified file positions an internal pointer so that subsequent GET or PUT references to the file will refer to the first item in it.

The RESET statement does not close files. If a file is to be used for both output and input during execution of a single program, it must be closed between output and input references. When a file is opened in response to a PUT or GET statement, it is automatically reset.

Rules:

1. If a file specified in a RESET statement is not currently active, its name in the RESET statement will be ignored.
2. Naming conventions for on-line data files vary among IBM implementations of the BASIC language.

Example:

The RESET statement in 120 repositions the file named IN to the beginning. The GET statement in 130, therefore, reads the first *three* values of IN into A, B, and C, respectively.

```
60 GET 'IN' X, Y, Z
.
.
.
120 RESET 'IN'
130 GET 'IN', A, B, C
```

The CLOSE Statement

Function:

The CLOSE statement causes input and output files to be deactivated.

General Format:

CLOSE *file reference-1* [,*file-reference-n*] . . .

where *file-reference-i* is a literal constant enclosed in single or double quotes.

Action:

The file or files specified in the CLOSE statement are deactivated. An implicit CLOSE statement is automatically executed for each active file at the completion of program execution.

Rules:

1. If a file is to be used for both output and input during execution of a single program, it must be closed between output and input references.
2. If a file specified in a CLOSE statement is not active at the time the CLOSE statement is executed, its appearance in the statement is ignored.
3. Naming conventions for on-line data files vary among IBM implementations of the BASIC language.

Example:

```
25 CLOSE 'CDF',"ABF"
```

Matrix Operations

In mathematics, a matrix is a group of arithmetic values arranged in a rectangular system of rows and columns. A vector is a series of arithmetic values arranged in a single row.

A matrix in the BASIC programming language is an arithmetic array of one or two dimensions. MAT statements specify operations which are performed on the entire collection of members of an array at once, rather than on each member individually (scalar operations).

There are seven MAT statements, each performing a function for an entire array corresponding to the functions performed for single values by the equivalent scalar statement. The MAT statements are:

MAT Assignment
MAT READ
MAT INPUT
MAT PRINT
MAT PRINT USING
MAT GET
MAT PUT

Array expressions used in the MAT assignment statement may be in either binary or unary form. The binary array expressions are:

A + B	The sum of two matrices.
A − B	The difference of two matrices.
A * B	The product of two matrices.
(*x*) * A	The product of a scalar value (*x*) and a matrix (A).

The unary array expressions used in the MAT assignment statement are:

An array itself

ZER	The zero array function
CON	The unity array function
IDN	The identity matrix function
INV	The inverse matrix function
TRN	The transpose matrix function

The following pages contain complete descriptions of the matrix operations available in the BASIC language.

Redimensioning Arithmetic Arrays

It is sometimes desirable to change the dimensions of an arithmetic array during the course of a BASIC program. As an example, an array of four rows and three columns can be schematically represented this way:

A(1,1)	A(1,2)	A(1,3)
A(2,1)	A(2,2)	A(2,3)
A(3,1)	A(3,2)	A(3,3)
A(4,1)	A(4,2)	A(4,3)

If the array were named A, a reference to the fifth item in the array would be written A(2,2).

This array could be redimensioned according to the rules below so that it was arranged as three rows and four columns. It could then be represented this way:

A(1,1)	A(1,2)	A(1,3)	A(1,4)
A(2,1)	A(2,2)	A(2,3)	A(2,4)
A(3,1)	A(3,2)	A(3,3)	A(3,4)

Now, a reference to the fifth item would be written A(2,1).

If the array were redimensioned again, this time to contain three rows and two columns, it could be represented this way:

A(1,1)	A(1,2)
A(2,1)	A(2,2)
A(3,1)	A(3,2)

A reference to the fifth item would now be written A(3,1).

Arithmetic arrays may be redimensioned when used in any of the following matrix operations:

MAT READ statement
MAT GET statement
MAT INPUT statement
CON function
IDN function
ZER function

Both one- and two-dimensional arrays may be redimensioned, but the number of dimensions in the array may be not be changed during redimensioning.

The total number of elements in an array after redimensioning must not exceed the number originally specified when the array was declared. Thus, the array in the example above could not be redimensioned to contain three rows and five columns since a total of fifteen members would then be required.

Both implicitly and explicitly declared arithmetic arrays may be redimensioned.

The MAT Assignment Statement (Simple)

Function:

This statement assigns the elements of one array to another array.

General Format:

MAT *name-1* = *name-2*

where *name-i* is the name of an arithmetic array.

Action:

The value of each element of the array specified to the right of the equal sign is assigned to the corresponding element position in the array to the left of the equal sign.

Rules:

1. Before being used in a MAT assignment statement, both arrays specified must have been previously defined, either implicitly, or explicitly in a DIM statement.
2. If both arrays specified in the simple MAT assignment statement do not have identical dimensions, program execution is terminated.

Example:

```
 20  DIM A (2,2), B(2,2)
              .
              .
              .
100 MAT A = B
```

The resulting values are represented below:

$$\text{If B} = \begin{bmatrix} a & b \\ c & d \end{bmatrix} \qquad \text{A is } \begin{bmatrix} a & b \\ c & d \end{bmatrix}$$

The MAT Assignment Statement (Addition and Subtraction)

Function:

These statements assign the sum or difference of the elements of two arrays to the elements of a third array.

General Format:

MAT *name-1* = *name-2*{+|−}*name-3*

where *name-i* is the name of an arithmetic array.

Action:

The values of the elements of the arrays specified to the right of the equal sign are added or subtracted, as specified, and the result of the operation is assigned to the corresponding element position in the array to the left of the equal sign.

Rules:

1. Before being used in a MAT assignment statement, all three arrays specified must have been previously defined, either implicitly, or explicitly in a DIM statement.
2. If all three arrays specified in the statement do not have identical dimensions, program execution is terminated.

Example:

```
10 DIM X(2,2), Y(2,2), Z(2,2)
                .
                .
                .
100 MAT X = Y + Z
```

The resulting values are represented below:

$$\text{If Y} = \begin{bmatrix} a & b \\ c & d \end{bmatrix} \text{ and z} = \begin{bmatrix} e & f \\ g & h \end{bmatrix} \text{ x is } \begin{bmatrix} a+e & b+f \\ c+g & d+h \end{bmatrix}$$

The MAT Assignment Statement (Multiplication)

Function:

This statement performs the mathematical matrix multiplication of two arithmetic arrays and assigns the product to a third.

General Format:

MAT *name-1* = *name-2* * *name-3*

where *name-i* is the name of a two-dimensional arithmetic array.

Action:

In matrix multiplication, an array A of dimensions (p,m) and an array B of dimensions (m,n) yield a product array C of dimensions (p,n) such that for i = 1,2 . . . p and for j = 1,2, . . . n,

$$C(i,j) = \sum_{k=1}^{m} A(i,k) * B(k,j)$$

Rules:

1. Before being used in the MAT assignment statement, all three arrays specified must have been previously defined, either implicitly, or explicitly in a DIM statement.
2. The array specified to the left of the equal sign may not be the same array as either array to the right of the equal sign.
3. If any of the following relationships is not true, program execution will be terminated:
   a. The number of columns in the array specified by *name-2* in the format example must be equal to the number of rows in the array specified by *name-3.*
   b. The number of rows in the array specified by *name-1* must equal the number of rows in the array specified by *name-2.*
   c. The number of columns in the array specified by *name-1* must equal the number of columns in the array specified by *name-3.*

Example:

```
10 DIM Z(2,2), Y(2,2), Z(2,2)
                .
                .
                .
100 MAT Z = X * Y
```

The resulting values are represented below:

$$\text{If } x = \begin{bmatrix} a & b \\ c & d \end{bmatrix} \text{and } y = \begin{bmatrix} e & f \\ g & h \end{bmatrix} \; z \text{ is } \begin{bmatrix} a*e + b*g & a*f + b*h \\ c*f + d*h & c*e + d*g \end{bmatrix}$$

The MAT Assignment Statement (Scalar Multiplication)

Function:

This statement causes the elements of an arithmetic array to be multiplied by the value of an arithmetic expression, and the resulting products to be assigned to the elements of another arithmetic array.

General Format:

MAT *name-1* = *(x)* * *name-2*

where *name-i* is the name of an arithmetic array and the parenthesized *x* is an arithmetic expression.

Action:

The scalar expression is evaluated, and that value is multiplied by the value of each element in the array to the right of the equal sign. The resulting products are then assigned to the elements of the array to the left of the equal sign, in the corresponding positions.

Rules:

1. Before being used in the MAT assignment statement, both arrays specified must have been previously defined either implicitly, or explicitly in a DIM statement.
2. If both arrays specified do not have identical dimensions, program execution is terminated.

Example:

```
20 DIM X(2,2), Y(2,2)
                .
                .
                .
100 MAT Y = (4) * X
```

The resulting values are represented below:

If x = $\begin{bmatrix} a & b \\ c & d \end{bmatrix}$ Y is $\begin{bmatrix} 4*a & 4*b \\ 4*c & 4*d \end{bmatrix}$

The MAT Assignment Statement (Inversion Function)

Function:

This statement causes one array to be assigned the mathematical matrix inverse of another array.

General Format:

MAT *name-1* = INV (*name-2*)

where *name-i* is the name of a two-dimensional arithmetic array.

Action:

The matrix inverse of the array specified by the name to the right of the equal sign is assigned to the array specified by the name to the left of the equal sign. For a square array A of dimensions (m,m) the inverse array, B if it exists, is an array of identical dimensions such that:

A*B = B*A = I

where I is an identity matrix.

Not every matrix has an inverse. The intrinsic function DET (see "Functions") may be used to determine if a given array has an inverse. The inverse of array A exists if DET (A) $\neq$0.

Rules:

1. Before being used in a MAT assignment statement, both specified arrays must have been previously defined, either implicitly, or explicitly in a DIM statement.
2. Both arrays specified must be square, and both must have identical dimensions.
3. The same array name may not be used on both sides of the equal sign, even if the matrix is square.

Example:

```
20 DIM X (2,2), Y (2,2)
                .
                .
                .
80 IF DET (Y) = 0 THEN 100
90 MAT X = INV (Y)
100 END
```

The resulting values are represented below:

If Y is $\begin{bmatrix} 1 & 1 \\ 1 & 2 \end{bmatrix}$ Then X is $\begin{bmatrix} 2 & -1 \\ -1 & 1 \end{bmatrix}$

The MAT Assignment Statement (Transpose Function)

Function:

This statement causes one array to be replaced by the matrix transpose of another array.

General Format:

MAT *name−1* = TRN (*name−2*)

where *name-i* is the name of a two-dimensional arithmetic array.

Action:

The transpose matrix of the array specified by the name to the right of the equal sign is assigned to the array specified by the name to the left of the equal sign. Thus, the value of element (x,y) in the former is assigned to element position (y,x) in the latter, where x and y are row and column numbers of the elements.

Rules:

1. Before being used in a MAT assignment statement, both specified arrays must have been previously defined, either implicitly, or explicitly in a DIM statement.
2. The number of rows in each array must be equal to the number of columns in the other.
3. The same array name may not be used on both sides of the equal sign, even if it is a square matrix.

Example:

```
40 DIM A (3,2), B(2,3)
60 MAT B = TRN (A)
```

The resulting values are represented below:

If A is $\begin{bmatrix} a & b & c \\ d & e & f \end{bmatrix}$ Then B becomes $\begin{bmatrix} a & d \\ b & e \\ c & f \end{bmatrix}$

The MAT Assignment Statement (Identity Function)

Function:

This statement causes an arithmetic array to assume the form of an identity matrix. It may also be used to redimension the array.

General Format:

MAT *name* = IDN [(x_1 ,x_2)]

where *name* is the name of an arithmetic array containing equal numbers of rows and columns, and x_i is an arithmetic expression.

Action:

Each element of the specified array for which the values of both subscripts are equal, e.g., A (2,2) or A (3,3), is assigned the integer value one (1). Each element of the specified array for which the subscripts are unequal, e.g., A (2,3) or A (3,1), is assigned the value zero (0).

If the optional arithmetic expressions follow the keyword IDN in the statement, the truncated integer portions of their values are used to redimension the array before the assignment of one or zero to each of its elements.

Rules:

1. Before being used in a MAT assignment statement, the specified array must have been previously defined, either implicitly, or explicitly in a DIM statement.
2. Redimensioning of the array must not change the original number of dimensions, nor exceed the original number of array members.
3. The specified arithmetic array must be a square matrix; that is, the number of rows must equal the number of columns. If redimensioning is specified, the truncated integer portions of the specifying arithmetic expressions must be equal.

Example:

```
50 DIM X (5,5)
60 MAT X = IDN (4,4)
```

The resulting values are represented below:

$$X \text{ is } \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

The MAT Assignment Statement (ZER Function)

Function:

This statement assigns the value zero (0) to all elements of an arithmetic array. It may be used to redimension the array.

General Format:

MAT *name* = ZER [(x_1 [,x_2])]

where *name* is the name of an arithmetic array and x_i is an arithmetic expression.

Action:

The value zero (0) is assigned to each element of the specified array. If the optional arithmetic expressions follow the keyword ZER in the statement, the truncated integer portions of their values are used to redimension the array before assignment of the zero to each element.

Rules:

1. Before being used in a MAT assignment statement, the specified array must have been previously defined, either implicitly, or explicitly in a DIM statement.
2. Redimensioning of the array must not change the original number of dimensions, nor exceed the original number of array members.

Example:

```
40 DIM Y (3,3)
50 MAT Y = ZER
```

The resulting values are represented below:

$$Y \text{ is } \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

The MAT Assignment Statement (CON Function)

Function:

This statement assigns the value one (1) to all elements of an arithmetic array. It may also be used to redimension the array.

General Format:

MAT *name* = CON [(x_1 [,x_2])]

where *name* is the name of an arithmetic array and x_i is an arithmetic expression.

Action:

The integer value one (1) is assigned to each element of the specified array. If the optional arithmetic expressions follow the keyword CON in the statement, the truncated integer portions of their values are used to redimension the arrays before assignment of the integer one to each element.

Rules:

1. Before being used in a MAT assignment statement, the specified array must have been previously defined, either implicitly, or explicitly in a DIM statement.
2. Redimensioning of the array must not change the original number of dimensions, nor exceed the original number of array members.

Example:

```
20 DIM X (4,5)
     .
     .
     .
80 MAT X = CON (3,3)
```

The resulting values are represented below:

$$\text{x is} \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

The MAT READ Statement

Function:

The MAT READ statement causes arithmetic values from the data table established by DATA statements to be assigned to the members of an arithmetic array without referencing each array member individually. The MAT READ statement may also be used to redimension arithmetic arrays.

General Format:

MAT READ *name-1*[(x_1[,x_2])][,*name-2*[(x_3[,x_4])]]...

where *name-i* is the name of an arithmetic array and x_i is an arithmetic expression.

Action:

At the beginning of program execution, a pointer is set to the first value in the data table, if one exists. When a MAT READ statement is encountered, successive values from the data table are assigned to the arrays in the MAT READ statement. The values are assigned to the array references by rows, beginning at the current position of the data table pointer.

All arithmetic data is truncated or zero-filled, if necessary, to conform to the form of arithmetic (long or short) specified for the program in which the values are assigned.

If the optional arithmetic expressions follow the array names in the MAT READ statement, the truncated integer portions of the expression values are used to redimension the arrays before values are assigned from the data table.

Rules:

1. Before use in a MAT READ statement, arrays must have been previously defined, either implicitly, or explicitly in a DIM statement.
2. Redimensioning of arrays in a MAT READ statement must not change the original number of dimensions nor exceed the original number of members.
3. All data values assigned must be arithmetic.
4. If the data table is exhausted and unassigned arrays or array members remain in the MAT READ statement, an error condition results.
5. The MAT READ statement is invalid if there are no DATA statements in the program.
6. The pointer in the data table may be reset by use of the RESTORE statement.

Example:

```
70 MAT READ $(2,X-3), A, B(12)
```

The MAT INPUT Statement

Function:

The MAT INPUT statement allows the BASIC user to assign values from the terminal at execution time to members of an arithmetic array without specifying each array member individually. The MAT INPUT statement may also be used to redimension arithmetic arrays.

General Format:

MAT INPUT *name-1* $[(x_1[,x_2])][,$*name-2*$[(x_3[,x_4])]]\ldots$

where *name-i* is the name of an arithmetic array and x_i is an arithmetic expression.

Action:

When a MAT INPUT statement is executed, it causes a question mark to be printed out at the terminal and program execution to be temporarily interrupted. The user then enters a list of arithmetic values, a row at a time, which are assigned to members of the specified arrays by rows. Each new row is requested by the system with the printing of two question marks as soon as the previous row is completed.

All arithmetic data is truncated or zero-filled, if necessary, to conform to the form of arithmetic (long or short) specified for the program in which the values are assigned.

If the optional arithmetic expressions follow the array names in the MAT INPUT statement, the truncated integer portions of the expression values are used to redimension the arrays before data values are entered at the terminal.

When a MAT INPUT statement is executed immediately after a PRINT or MAT PRINT statement in which the final delimiter is a comma or semicolon, the question mark generated by the MAT INPUT statement is printed directly following the last data item on the same print line. In all other instances, the question mark appears as the first character on the next print line.

Rules:

1. Array references may be one- or two-dimension arrays.
2. Before use in a MAT INPUT statement, an array must have been previously defined, either implicitly, or explicitly in a DIM statement.
3. Redimensioning of arrays in a MAT INPUT statement must not change the original number of dimensions nor exceed the original number of members.
4. The final entry for each row of each array must be followed by a carriage return to signify end of row.
5. If a line is full and input data remains to be entered for the same row, the last value entered must be followed by a comma before the carriage is returned to continue.
6. All data values entered must be arithmetic.
7. If the number of values entered for a row does not equal the number of members in the corresponding row of the array, an error conditions results.
8. The procedure for retry or re-entering of data after an error varies among IBM implementations of the BASIC language.

Example:

```
90 MAT INPUT A, B(20), C(5,20)
```

The MAT PRINT Statement

Function:

The MAT PRINT statement causes the values of all members of a specified arithmetic array to be printed at the terminal without references to each array member individually.

General Format:

MAT PRINT *name-1* [,|; *name-n*] . . . [,|;]

where *name-i* is the name of an arithmetic array and the characters enclosed in braces (comma and semicolon) are delimiting characters which determine the position of the print element.

Action:

When a MAT PRINT statement is executed, each array specified in the statement is converted to a specified output format and printed at the terminal. (See the PRINT statement for information on print zones and the conversion of values for printing.)

Each array in the MAT PRINT statement is printed by rows, the first row of each array beginning at the start of a new line and being separated from the preceding line by two blank lines. The remaining rows of each array begin at the start of a new line and are separated from the preceding line by a single blank line.

After the printing of each array member, the carriage is repositioned as specified by the delimiting character, as described below. After the final or only full array has been printed, the carriage is positioned to the start of the next line.

Printing of Converted Array Members: If the line contains sufficient space to accommodate the value of the converted array member, printing starts at the current carriage position.

If the line does not contain sufficient space to accommodate the value of the converted array member, printing begins at the start of the next line.

Positioning of the Carriage After Printing: If the delimiter is a comma, the carriage is moved past any remaining spaces in the full print zone. Should the end of the line be encountered during the movement, the carriage is moved to the beginning of the next line.

If the delimiter is a semicolon, the carriage is moved past any remaining spaces in the packed print zone. Should the end of the line be encountered during the movement, the carriage is moved to the beginning of the next line.

If the final delimiter is null, it is treated as a comma.

Rules:

1. Before being used in a MAT PRINT statement, arrays must have been previously defined, either implicitly, or explicitly in a DIM statement.
2. Array references may be to one- or two-dimension arrays.
3. All data values printed must be arithmetic.
4. Null delimiters are not permitted in a MAT PRINT statement except as the final delimiter.

Example:

```
20  MAT PRINT D,X
```

In the following example, assume that there are 18 spaces from the beginning of one print zone to the next.

```
10  DIM A (15), X(2,2)
20  MAT READ A
30  DATA, 1,2,3,4,5,6,7,8,9,10,11,12,13,14,15
40  MAT X = CON
50  PRINT A,X
      .
      .
      .
1      2      3      4      5      6      7

8      9      10     11     12     13     14

15


1      1

1      1
```

The MAT PRINT USING Statement

Function:

The MAT PRINT USING statement and its associated Image statement allow the BASIC user to have the values of all the members of a specified arithmetic array printed at the terminal in a format of his own choosing, without having to specify each array member individually.

General Format:

MAT PRINT USING *s*, *name-1*[,*name-n*] . . .

where *s* is the number of the associated Image statement and *name-i* is the name of an arithmetic array.

Action:

Each array referenced in the MAT PRINT USING statement is printed by rows at the terminal according to the format defined by the associated Image statement. (See the PRINT USING and Image Statements for information on format specification.)

When printed, the first row of each array begins at the start of a new line and is separated from the preceding line by two blank lines. Each succeeding array row begins at the start of a new line and is separated from the preceding row by one blank line. After the last or only array has been printed, the carriage is repositioned to the beginning of the next line.

Rules:

1. Before being used in a MAT PRINT USING statement, an array must have previously defined, either implicitly, or explicitly in a DIM statement.
2. One- or two-dimension arrays may be specified in a MAT PRINT USING statement.
3. If the Image statement specified in the MAT PRINT USING statement does not contain at least one conversion specification, an error condition results.
4. If the number of members in the array row exceeds the number of conversion specifications in the associated Image statement, a carriage return occurs at the end of the Image statement and the Image statement is reused for the printing of the remaining members of that row.
5. If the number of members in the array row is less than the number of conversion specifications in the specified Image statement, the line for that row is terminated at the first unused conversion specification.
6. All values printed must be arithmetic.

Example:

```
10 DEF A (4,3)
20 : ### ##.## ##.##
30 MAT A = CON
40 MAT PRINT USING 20, A
```

The output would appear as follows:

```
 1  1.00  1.00E00
 1  1.00  1.00E00
 1  1.00  1.00E00
 1  1.00  1.00E00
```

The MAT GET Statement

Function:

The MAT GET statement causes values from a specified file to be assigned to all members of a specified arithmetic array without referring to each array member individually. The MAT GET statement may also be used to re-dimension arithmetic arrays.

General Format:

MAT GET *file-reference*, *name-1*[(x_1[,x_2])] [,*name-2*[(x_3[,x_4])]]

where *file-reference* is a character constant enclosed in single or double quotation marks, *name-i* is the name of an arithmetic array, and x_i is an arithmetic expression.

Action:

A file is activated for input by the first execution of a MAT GET (or GET) statement specifying its file name. The file is positioned at the beginning and values from the file are assigned to the specified arrays row by row. Subsequent MAT GET statements for the same file cause values to be assigned from the current file position.

All arithmetic data is truncated or zero-filled, as necessary, to conform to the form of arithmetic (long or short) specified for the program in which the values are assigned.

If the optional arithmetic expressions follow the array names in the MAT GET statement, the truncated integer portions of the expression values are used to redimension the arrays before data values are assigned from files.

A file is deactivated in response to a CLOSE statement or at the end of program execution.

Rules:

1. The arithmetic arrays specified in the MAT GET statement may be either one- or two-dimensional.
2. Before being used in a MAT GET statement, an array must have been previously defined, either implicitly, or explicitly in a DIM statement.
3. Redimensioning of arrays in a MAT GET statement must not change the original number of dimensions nor exceed the original number of members.
4. A file currently activated as an output file may not be specified in a MAT GET statement. It must first be closed.
5. If the input file is exhausted before a specified array is filled, program execution is terminated.
6. Naming conventions for on-line data files vary among IBM implementations of the BASIC language.

Example:

```
50 MAT GET "ABF", A, B(10), Z(5,20)
```

The MAT PUT Statement

Function:

The MAT PUT statement causes the value of each member of an arithmetic array to be placed in a specified file without referring to each array member individually. The MAT GET statement may also be used to redimension arithmetic arrays.

General Format:

MAT PUT *file-reference, name-1*[,*name-n*] . . .

where *file-reference* is a character constant enclosed in single or double quotation marks and *name-i* is the name of an arithmetic array.

Action:

A file is activated for output by the first execution of a MAT PUT (or PUT) statement specifying its file name. The file is positioned at the beginning and values of array members are placed in it row by row. Subsequent MAT PUT statements for the same file cause values to be placed in it starting at the current file position.

All arithmetic data is truncated or zero-filled, as necessary, to conform to the form of arithmetic (long or short) specified for the program before the values are placed in files.

A file is deactivated in response to a CLOSE statement or at the end of program execution.

Rules:

1. The arithmetic arrays specified in the MAT PUT statement may be either one- or two-dimensional.
2. Before being used in a MAT PUT statement, an array must have been previously defined, either implicitly, or explicitly in a DIM statement.
3. A file currently activated as an input file may not be referenced in a MAT PUT statement. It must first be closed.
4. Naming conventions for on-line data files vary among IBM implementations of the BASIC language.

Example:

```
60 MAT PUT "ABF", A, M, Q
```

Appendix A: Implementation—defined Restrictions

The following aspects of the BASIC language are subject to restrictions in each IBM implementation of the language. The actual values of the restrictions vary among the implementations and may be found in the appropriate documentation for each implementation.

- Arithmetic Precision*
- Maximum magnitude of numeric values
- Maximum number of digits per statement number
- Maximum number of characters in a character constant entered at the terminal
- Maximum number of digits in an arithmetic value entered at the terminal
- Maximum size of arrays
- Maximum number of data items per file
- Maximum number of statements per program
- Maximum number of levels of nested subroutines
- Maximum number of levels of nested FOR loops
- Maximum number of user-written functions per program
- Maximum number of nested function references per expression
- Maximum number of variables per multiple assignment statement
- Maximum number of Image statements per program
- Naming conventions of files and programs
- Procedures for resumption of processing after the PAUSE statement and error conditions.

---

*See under the heading "Arithmetic Precision" for minimum values.

Appendix B: Collating Sequence of the BASIC Character Set

Note that both upper and lower case letters of the standard English alphabet are represented internally by the EBCDIC bit configuration of the upper case characters only.

Character	Internal Hexadecimal Representation	Name
	40	Blank
.	4B	Point or Period
<	4C	Less Than Sign
(	4D	Left Parenthesis
+	4E	Plus Sign
|	4F	Logical OR Sign or Vertical Bar
&	50	Ampersand
!	5A	Exclamation Symbol
$	5B	Dollar Sign
*	5C	Asterisk
)	5D	Right Parenthesis
;	5E	Semicolon
–	60	Hyphen or Minus Sign
/	61	Slash or Division Symbol
,	6B	Comma
>	6E	Greater Than Sign
?	6F	Question Mark
:	7A	Colon
#	7B	Pound or Number Sign
@	7C	Commercial "at" Sign
'	7D	Apostrophe or Single Quotation
=	7E	Equal Sign
"	7F	Double Quotation Mark
↑	8A	Up-Arrow or Exponentiation Sign
≤	8C	Less Than or Equal To Sign
≥	AE	Greater Than or Equal To Sign
≠	BE	Not Equal Sign
A,a	C1	
B,b	C2	
C,c	C3	
D,d	C4	
E,e	C5	
F,f	C6	
G,g	C7	
H,h	C8	
I,i	C9	
J,j	D1	
K,k	D2	
L,l	D3	
M,m	D4	

Character	Internal Hexadecimal Representation
N,n	D5
O,o	D6
P,p	D7
Q,q	D8
R,r	D9
S,s	E2
T,t	E3
U,u	E4
V,v	E5
W,w	E6
X,x	E7
Y,y	E8
Z,z	E9
0	F0
1	F1
2	F2
3	F3
4	F4
5	F5
6	F6
7	F7
8	F8
9	F9

Appendix C: Sample BASIC Programs

Example 1

The following is a simple program for computing compound interest using the formula $A=P(1+R)^N$, where P is the principle, R is the rate of interest per period, and N is the number of interest periods. The value A, the final amount yielded after the specified time, is to be computed.

Input to the program consists of values for the variables P, I, Y, and T, which represent the principle in dollars and cents, the yearly interest rate in percent, the number of years for which the interest is to be computed, and the number of times per year the interest is to be compounded, respectively.

The program is designed to request values for each of the variables, reject negative or zero values, calculate the final amount, and print the answer at the terminal. After each answer is printed, the program asks if the user wishes to make further calculations. If so, new values for the variables P, I, Y, and T are requested. If not, the program ends.

```
10 PRINT "P,I,Y,T";
20 INPUT P,I,Y,T
30 REM BAD DATA CHECK
40 IF P <= 0 THEN 200
50 IF I <= 0 THEN 200
60 IF Y <= 0 THEN 200
70 IF T <= 0 THEN 200
80 REM COMPUTATION
90 LET N = Y*T
100 LET R= I/100/T
110 LET A = P*(1+R)**N
120 REM PRINTING OF ANSWER
130 PRINT USING 140,A
140 : FINAL TOTAL IS $#####.##
150 REM TERMINATION ROUTINE
160 PRINT 'TYPE "YES" OR "NO". ANOTHER';
170 INPUT S$
180 IF S$= "YES" GOTO 10
190 GOTO 999
200 REM ERROR ROUTINE
210 PRINT "INCORRECT DATA. PLEASE RETYPE."
220 GOTO 10
999 END
```

A typical session using this program might appear as follows:

```
P,I,Y,T?        1000,6,10,4
 FINAL TOTAL IS $ 1813.97
TYPE "YES" OR "NO".  ANOTHER?        "yes"
P,I,Y,T?        100,7,00,12
INCORRECT DATA. PLEASE RETYPE.
P,I,Y,T?        100,7,20,12
FINAL TOTAL IS $  403.81
TYPE "YES" OR "NO". ANOTHER?       "no"
```

Example 2

This program approximates the sum:

$$S = \sum_{n=0}^{\infty} \frac{1}{x^n}$$

Statement 10 gives the variable x a value of 1.065. Statement 20 computes the term and adds it to the sum, s, which is initially zero. Statement 30 increments the value of N, the number of terms. Statement 40 compares the tentative "next sum" with the sum at that point, and, if they are different, the sequence of statements 20, 30, and 40 is repeated. When the limit of precision is reached, that is, when the computer can no longer distinguish between the "present sum" and the "next sum" the values of the number of terms (N), the sum of terms (S), and the last term are printed out and the program ends.

```
10  LET X=1.065
20  LET S= S+1/(X**N)
30  LET N= N+1
40  IF S ≠ S+1/(X**N) THEN 20
50  PRINT 'NUMBER OF TERMS:', (N-1)
60  PRINT USING 70, S
70  :THE SUM OF TERMS: ##.##############
80  PRINT 'THE LAST TERM:', 1/(x**(N-1))
90  END
```

When run under an IBM implementation of BASIC providing seven and eleven significant digits in E-format for short and long form respectively, the program produces the following output:

```
NUMBER OF TERMS:        176
THE SUM OF TERMS:       16.38363647460938
THE LAST TERM:          1.536384E-05
```

In long form, the results are:

```
NUMBER OF TERMS:        528
THE SUM OF TERMS:       16.38461538461450
THE LAST TERM:          3.6258271359E-15
```

Appendix D: Formalized Definition of the BASIC Language

INTRODUCTION

1.1 SCOPE. This standard establishes:

1. The syntax of the individual BASIC statements.
2. The semantics of both individual BASIC statements and aggregates of the individual statements.
3. The language which must be implemented by a processor to satisfy the requirements of a BASIC processor.
4. The minimum precision required for numerical quantities and operations herein referred to as the "apparent precision."

This standard does not establish:

1. The mechanism by which programs are transformed for use on a data processing system, herein referred to as a "processor."
2. The manual operations required for setup, initiation, and control of the use of such programs on data processing equipment.
3. The size or complexity of a program which will exceed the capacity of any specific data processing system or processor.
4. The range or precision of numerical quantities provided they minimally satisfy the apparent precision defined in the requirements.

REQUIREMENTS

2.1 Processor Requirements.

2.1.1 IBM BASIC processors must accept all of the syntax described in Sections 2.2 and 2.3 of this standard. The processors may not support any syntax beyond that which is described in Sections 2.2 and 2.3 except in the manner as defined in Section 3.2.

2.1.2 Syntax accepted must be interpreted as specified by the semantics in Section 2.2.

2.1.3 The results of execution of a BASIC program must be in conformance to the semantics as defined in Section 2.2.

Specification Table of Contents

2.2	BASIC Language
2.2.1	BASIC Program Elements
2.2.1.1	Character Set
2.2.1.2	Use of Blanks
2.2.1.3	Comments
2.2.2	Data Types
2.2.2.1	Arithmetic Data
2.2.2.2	Character Data
2.2.2.3	Data Aggregates-Arrays
2.2.3	Expressions and Operators
2.2.3.1	Character Expressions
2.2.3.2	Arithmetic Expressions
2.2.3.2.1	Scalar Operators
2.2.3.2.2	Priority of Arithmetic Operators
2.2.3.3	Array Expressions
2.2.3.4	Relational Operators
2.2.4	Statements
2.2.4.1	Descriptive Statements
2.2.4.1.1	DIM
2.2.4.1.2	DEF
2.2.4.2	Input/Output Statements
2.2.4.2.1	Internal File I/O Statements
2.2.4.2.1.1	DATA
2.2.4.2.1.2	READ
2.2.4.2.1.3	RESTORE
2.2.4.2.1.4	MAT READ
2.2.4.2.2	Interactive Terminal I/O Statements
2.2.4.2.2.1	INPUT
2.2.4.2.2.2	MAT INPUT
2.2.4.2.2.3	PRINT
2.2.4.2.2.4	MAT PRINT
2.2.4.2.2.5	PRINT USING and Image
2.2.4.2.2.6	MAT PRINT USING
2.2.4.2.3	External File I/O Statements
2.2.4.2.3.1	GET
2.2.4.2.3.2	MAT GET
2.2.4.2.3.3	PUT
2.2.4.2.3.4	MAT PUT
2.2.4.2.3.5	RESET
2.2.4.2.3.6	CLOSE
2.2.4.3	Data Movement and Computational Statements
2.2.4.3.1	LET
2.2.4.3.2	MAT Assignment
2.2.4.4	Control Statements
2.2.4.4.1	GOTO
2.2.4.4.2	GOSUB and RETURN
2.2.4.4.3	FOR and NEXT
2.2.4.4.4	IF
2.2.4.4.5	PAUSE
2.2.4.4.6	STOP
2.2.4.4.7	END
2.2.4.5	REM Statement
2.2.5	Functions
2.2.5.1	User Functions
2.2.5.2	Intrinsic Function
2.2.5.3	Internal Constants
2.3	Syntax of the BASIC Language

2.2 BASIC Language

2.2.1 BASIC Program Elements. This section, 2.2 BASIC Language, will define the syntax and semantics of the BASIC Language. Section 2.3, BASIC Language Syntax, will summarize in a formal manner the syntax of the entire language.

The BASIC Language is defined in terms of a terminal input data processing system, and the few restrictions inherent in the definition are related to this terminal input concept. The action taken for all errors in BASIC processing is implementation defined.

2.2.1.1 Character Set. The character set of BASIC is in the collating sequence of the Extended Binary-Coded-Decimal Interchange Code (EBCDIC). The following characters are directly related to the syntactic structure of BASIC programs.

Alphabetic Characters: A through Z, a through z, and the three alphabetic extenders, currency symbol ($), the number sign (#), and the commercial "at" sign (@). The upper case and lower case alphabetic letters are treated equivalently and may be used interchangeably.

Numeric Characters: There are ten digits, 0 through 9.

Special Characters: There are 24 special characters.

NAME	CHARACTER
Blank	
Equal sign or assignment symbol	=
Plus sign	+
Minus sign	-
Asterisk or multiply symbol	*
Slash or divide symbol	/
Up-arrow or exponentiation symbol	↑
Right parenthesis	)
Left parenthesis	(
Comma	,
Point or period	.
Single quotation mark	'
Double quotation mark	"
Semicolon	;
Colon	:
OR sign or vertical bar	|
Exclamation symbol	!
Question mark	?
Ampersand	&
"Less than" symbol	<
"Greater than" symbol	>
"Not equal" symbol	≠
"Less than or equal to" symbol	≤
"Greater than or equal to" symbol	≥

Special characters are combined to create other syntactic forms in BASIC. For example:

"greater than or equal"	>=
"less than or equal"	<=
"not equal"	<>
exponentiation	**

All elements that make up a BASIC program are constructed from the preceding BASIC character set. There are two exceptions: character constants and comments which may contain any character permitted by a particular machine configuration.

2.2.1.2 Use of Blanks. Blanks may be used freely throughout a BASIC program with two exceptions. Blanks are meaningful within:

1. Character constants

2. Image specifications

In all other instances, blanks are ignored and assume no syntatic meaning.

2.2.1.3 Comments. Comments may be included in a BASIC program through the use of a REM statement whose sole purpose is to enable a user to insert comments or remarks throughout his program. Similarly, all single keyword statements in BASIC allow a user to insert comments following the keyword. The single keyword statements are:

```
END
PAUSE
REM
RESTORE
RETURN
STOP
```

2.2.2 Data Types. In BASIC, data may be referenced through the use of a variable or a constant. A variable is a symbolic name (identifier) whose value may change during the execution of a program. A constant has a value that cannot be changed.

The two types of data supported by BASIC are arithmetic and character-string data.

2.2.2.1 Arithmetic Data. An item of arithmetic data is one with a numeric value. Arithmetic data items have the characteristics of base, scale, and precision.

The base of an arithmetic data item is decimal.

The scale of an arithmetic data item is either fixed-point or floating-point. A fixed-point data item is a number in which the position of the decimal point is either explicitly indicated by a decimal point or implicitly assumed to be to the right of the data item. A floating-point number is a fixed-point number followed by an optionally signed exponent to indicate a scaling factor. The exponent specifies the assumed position of the decimal point, relative to the position in which it appears.

The precision of an arithmetic data item is the maximum number of decimal digits the data item may contain, in the case of fixed-point, or the maximum number of significant decimal digits (excluding the exponent) to be maintained. This standard defines precision in terms of two execution time modes, short and long. Minimum short form "apparent" precision (S) is six significant decimal digits, and minimum long form "apparent" precision (L) is eleven significant decimal digits. This "apparent" precision defines the minimum precision which will be guaranteed for all BASIC operations. Each implementation must support a real or internal precision equal to or greater than its "apparent" precision. A reference to precision throughout this standard henceforth will refer to the apparent precision.

Whenever an expression is evaluated and is assigned to a numeric variable, the precision is maintained. The assigned item is aligned on the decimal point. Leading zeros are inserted if the assigned item contains fewer integer digits than the real precision; trailing zeros are inserted if the assigned item contains fewer fractional digits. An error will occur if the assigned item contains too many integer digits; truncation on the right will occur if it contains too many fractional digits. That is, if an arithmetic operation yields a loss in magnitude, an error will occur; if the loss is precision, truncation will occur.

A decimal fixed-point constant consists of one or more decimal digits with an optional decimal point. If no decimal point appears, the point is assumed to be immediately to the right of the rightmost digit. A sign may optionally precede a decimal fixed-point constant. The sign is considered to be part of the constant.

A decimal floating-point constant is written as a decimal fixed-point constant followed by the letter E, followed by an optionally signed one- or two-digit decimal integer constant. The entire constant may be preceded by a sign. The sign is considered to be part of the constant.

A simple arithmetic variable identifier consists of a letter, or a letter followed by a digit. Simple arithmetic variables can be used to represent only arithmetic data. The initial value of arithmetic variables is zero.

2.2.2.2 Character Data. A character string can include any character recognized by the particular hardware configuration. Any blank included in the character string is included in the length count.

Character string constants must be enclosed within single or double quotation marks. If the delimiting characters are single quotation marks, a double quotation mark may be included in the string, and vice versa. If a single quotation mark is to be a part of a string delimited by single quotation marks, it must be represented as two consecutive single quotation marks. Double quotation marks must be treated similarly. The length of a character string is the number of characters between the enclosing quotation marks. If two single or double quotation marks are used within a character string to represent quotation marks, they are counted as a single character.

For example:

```
'IT''S'
```

actually is:

```
IT'S of length 4.
```

A simple character variable's identifier consists of a letter followed by a dollar sign.

The maximum number of characters that can be assigned to a character variable is 18. The initial value of all character variables is 18 blanks. Character variables can be used to represent only character data.

2.2.2.3 Data Aggregates-Arrays. Single data elements of the same data type, numeric or character, may be grouped together to form an array. An array is an n-dimensional collection of members referenced by a single name (identifier). An individual member of the array is referred to by giving its relative position within the array.

Arithmetic Arrays: An arithmetic array can have either one or two dimensions. The arithmetic array identifier consists of a single letter, and the array can only contain arithmetic data. All members of an arithmetic array are initially set to zero when the program is executed. Associated with each dimension of an array is an

extent. The lower bound of an array (i.e., the subscript reference to the first member of an array) is always one. The upper bound of an extent is dependent upon whether the array has been explicitly defined through the use of a DIM statement or implicitly defined by a reference to an array without an explicit corresponding DIM statement. A one-dimensional arithmetic array is processed as a row vector, that is, it consists of a single row of members.

The extents of the dimensions of an arithmetic array are allocated in the following manner.

1. An array explicitly defined with a single dimension N is assigned the extent of 1 through N. The array contains N members.

2. An array explicitly defined with two dimensions M,N is assigned the extents of 1 through M and 1 through N. The array contains M times N members.

3. An array implicitly defined by reference with a single subscript is assigned the extent of 1 through 10. The array contains 10 members.

4. An array implicitly defined by reference with a double subscript is assigned the extents of 1 through 10 and 1 through 10. The array contains 100 data members.

The following specific rules are associated with the use of arithmetic arrays:

1. A member of an arithmetic array is referenced in the form a(e1[,e2]) where a is the arithmetic array name, and e1 and e2 are arithmetic expressions. The comma and expression e2 may not appear if the array is one-dimensional. When a reference to an array member is made, e1 and then (if applicable) e2 is evaluated at the point of reference.

2. An arithmetic array cannot be explicitly dimensioned by a DIM statement after it has been defined either explicitly or implicitly in a logically preceding statement.

3. An arithmetic array must be either explicitly or implicitly defined before its use in any of the MAT statements.

4. A one-dimensional array may be referenced by one and only one subscript. Two-dimensional arrays may be referenced by two and only two subscripts.

5. An array may be re-dimensioned as long as the original number of dimensions is not changed and the original total number of members is not exceeded. For example: if an array X was explicitly dimensioned as a 4 by 5 member array, the following references to a member after redimensioning would be valid, X(5,4), X(2,10). The following would be invalid, X(3,10) because the total number of members would be 30.

An arithmetic array may be re-dimensioned by its occurrence as a target in a MAT input statement or by use of the intrinsic functions ZER, CON, or IDN. The extents may be changed within the constraints of rule 5 above.

Character Arrays: A character array may have only one dimension. The character array identifier consists of a single letter followed by a dollar sign ($), and the array may only contain character data. All members of a character array are initially set to 18 blanks when the program is executed. The single dimension of a character array has an associated extent. The lower bound of the character array (i.e., the first member) is always one. The upper bound of an extent is dependent upon whether the array has been explicitly defined through the use of a DIM statement or implicitly defined by a reference to the array without a corresponding DIM statement. A character array is processed as a row vector, that is, it consists of only a single row of members.

The extent of the dimension of a character array is allocated in the following manner:

1. A character array explicitly defined with a dimension N is assigned the extent of 1 through N. The array contains N members.

2. A character array implicitly defined by reference with a single subscript is assigned the extent of 1 through 10. The array has 10 members.

The following specific rules are associated with the use of character arrays:

1. A member of a character array is referenced in the form a(e) where a is the character array name and e is an arithmetic expression.

2. A character array cannot be explicitly dimensioned by a DIM statement after it has been defined either explicitly or implicitly in a logically preceding statement.

3. A character array may not be used in a MAT statement. Therefore, a character array cannot be re-dimensioned.

4. A character array may be referenced by one and only one subscript.

2.2.3 Expressions and Operators. An expression is a representation of a value. A single constant or a variable is an expression. Constants, variables, and function references may be combined with operators and parenthesis to form an expression. Three forms of expressions are defined for BASIC: scalar, array, and relational. The result of the evaluation of a scalar expression is a single value -- a scalar. A scalar expression may be either an arithmetic expression or a character expression. The result of the evaluation of an array expression is a collection of values -- an array. A relational expression is used only in the context of an IF statement and results in a true or false value.

2.2.3.1 Character Expressions. A character expression may be composed of a character variable, character array member, or a character constant.

In all operations with character constants, except for output via PRINT or PRINT USING statements, character constants containing fewer than 18 characters will be blank padded on the right to 18 characters. A character constant containing more than 18 characters will be truncated on the right to 18 characters. Character constants containing no characters, a null character string, will be interpreted as 18 blank characters.

Character constants in PRINT and PRINT USING statements are processed with a length defined by their enclosing quotation marks with no truncation or padding.

2.2.3.2 Arithmetic Expressions and Operators. An arithmetic expression may be an arithmetic variable, arithmetic array member, constant or function reference, or a series of the above separated by binary operators and parentheses and preceded by unary operators. A discussion of function references, which may only be arithmetic, will be found in Section 2.2.5.

2.2.3.2.1 There are five binary operators, two unary operators and the right and left parentheses. The evaluation of an arithmetic expression is performed left to right with the priority of various operators defining the order of evaluation. The priority of operators will be defined later.

The five binary arithmetic operators are:

** or ↑	exponentiation (either form of the operator is acceptable)
*	multiplication
/	division
+	addition
-	subtraction

The two unary operators are + and -.

Special cases for the arithmetic operators and the resulting action are as follows:

Exponentiation: A↑B or A**B is defined as A raised to the B power.

1. If A=B=0 an error will occur.

2. If A=0 and B<0 an error will occur.

3. If A<0 and B is not an integer, an error of "a negative number to a fractional power" will occur.

4. If A≠0 and B=0, A↑B is evaluated as 1.

5. If A=0 and B>0, A↑B is evaluated as 0.

Multiplication and Addition: A*B and A+B, multiplication and addition respectively, are both commutative (i.e., A*B=B*A and A+B=B+A), but are not always associative due to low-order rounding errors (i.e., A*(B*C) does not necessarily give the same results as (A*B)*C).

Division: A/B is defined as A divided by B. If B=0, an error "division by zero" will occur.

Subtraction: A-B is defined as A minus B. No special conditions exist.

Unary Operators: The + and - signs may also be used as unary operators. Unary operators may be used in only two situations:

1. Following a left parenthesis and preceding an arithmetic expression, or

2. As the leftmost character in an entire expression which is not preceded by an operator.

For example:

-A+(-(B↑(-2))) is valid.
A+-B or B↑-2 is invalid.

2.2.3.2.2 Priority of Arithmetic Operators. An expession is evaluated operation-by-operation, from left to right. The priority of the operators in an

expression will determine the order of evaluation.

Operators	Priority
↑ or **	Highest
unary +, unary -	↓
*, /	
binary +, binary -	Lowest

An arithmetic expression is evaluated in a left to right order. The order of evaluation of an expression is determined as follows:

1. Since an operand may appear to be concurrently an operand to two operators, the priority of operators determines to which operator an operand is associated for purposes of evaluation. This requires that the two operators in question (if both exist) be compared in terms of their priority. The operation defined by the operator of the highest priority, of the two being compared, is performed first to reduce a subexpression (operand-1 operator operand-2) to a simple reference to data. Note that a subexpression for a prefix operator is "operator operand." If the operators are of equal priority, the operation defined by the operator on the left is performed first.

2. An operand may be:

   a. A constant,

   b. scalar reference,

   c. array reference,

   d. intrinsic function,

   e. user defined function,

   f. parenthetical subexpression.

When the decision of which operation is to be performed first, using Rule 1 above, the operands are accessed and reduced to simple references to data prior to the execution of the operator. The operands are accessed and reduced to simple references in a left to right order.

2.2.3.3 Array Expressions. Array expressions are composed of operations which are performed on the entire collection of members of an arithmetic array. Arithmetic array expressions consist of unary or binary operands.

A unary array expression may have one of the following forms:

A	An array itself
ZER	Zero array function
CON	Unity array function
IDN	Identity matrix function
INV	Inverse matrix function
TRN	Transpose matrix function

A binary array expression may have one of the following forms:

A+B	Sum of two matrices
A-B	Difference of two matrices
A*B	Product of two matrices
(e)*A	Product of the scalar value of e and the matrix A

Matrix multiplication, the inverse function, the transpose function, and the identity function are restricted to two-dimensional arrays only.

The definition of the preceding operations is as follows:

Terminology: Let A (m,n) represent an array with dimensions m rows and n columns. A (i,j) represents the member at the ith row and jth column in A.

Array Assignment

For A (m,n), the array B = A is an (m,n) array such that

```
B (i,j) = A (i,j)

          for i = 1,2,...m
              j = 1,2,...n
```

or

For A (m), the array B = A is an (m) array such that

```
B (i) = A (i)

        for i = 1,2,...m
```

Matrix Addition

```
A+B
```

For A (m,n) and B (m,n), the array C = A + B is an (m,n) array such that

```
C (i,j) = A (i,j) + B (i,j)

          for i = 1,2,...m
              j = 1,2,...n
```

or

For A (m) and B (m), the array C = A + B is an (m) array such that

$$C(i) = A(i) + B(i)$$

$$\text{for } i = 1,2,..m$$

Matrix Subtraction

A-B

Same as Matrix Addition except that the operation is subtraction.

Matrix Multiplication

A*B

For A (p,m) and B (m,n), the array C = A * B is a (p,n) array such that

$$C(i,j) = \sum_{k=1}^{m} A(i,k) * B(k,j)$$

$$\text{for } i = 1,2,...p \quad j = 1,2,...n$$

Scalar Multiplication

(e)*A

For A (m,n) and r the value of the expression e, then B = (e) * A is an (m,n) array such that

$$B(i,j) = r * A(i,j)$$

$$\text{for } i = 1,2,...m \quad j = 1,2,...n$$

or

For A (m) and r the value of the expression e, B = (e) * A is an (m) array such that

$$B(i) = r * A(i)$$

$$\text{for } i = 1,2,...m$$

The expression, e, is evaluated only once, and the result, r, is used throughout the rest of the evaluation of the statement.

Zero Array

ZER[(e,[,e_2])]

An array A (m,n) such that

$$A(i,j) = 0$$

$$\text{for } i = 1,2,...m \quad j = 1,2,...n$$

or

An array A (m) such that

$$A(i) = 0$$

$$\text{for } i = 1,2,...m$$

Unity Array

CON[(e,[,e_2])]

Same as Zero Array except that each member equals 1.

Identity Matrix

IDN[(e_1[,e_1])]

A square array A (m,m) such that

$$A(i,j) = 0$$

$$\text{for } i = 1,2,...m \quad j = 1,2,...m$$

$$\text{and } i \neq j$$

$$A(i,j) = 1$$

$$\text{for } i = j$$

Inverse Matrix

INV(A)

For the square array A(m,m), the inverse array of A (if it exists) is the array B(m,m) such that

$$A*B = B*A = I$$

where I is an m by m identity matrix.

The above relationship is approximately true since internal precision may not yield exactly an identity matrix.

Not every matrix has an inverse; matrices that do not are called "singular." The

function DET may be used to evaluate the determinant of a square array. A singular matrix is one for which the determinant is zero. An attempt to evaluate the inverse of a singular matrix will result in an error.

Transpose Matrix

TRN(A)

For A (m,n) the transpose matrix of A is the (n,m) array B such that

B (j,i) = A (i,j)

for i = 1,2,...m
j = 1,2,...n

2.2.3.4 Relational Expressions. Relational expressions are of the general form:

e1 relational-operator e2

The semantic of a relational expression is that the expression is either satisfied (true) or not satisfied (false). The relational operators are binary and are defined as:

Operator	Definition
=	Equal
<> or ≠	Not equal
>= or ≥	Greater than or equal
<= or ≤	Less than or equal
>	Greater than
<	Less than

Two forms of relational expressions are allowable in BASIC, arithmetic and character. The expressions e1 and then e2 are evaluated and their values are compared according to the definition of the relational operator used. The evaluation of the entire relational expression results in the expression being either satisfied or not satisfied.

2.2.4 Statements. The definition of the BASIC statements is presented in this section grouped according to their functions. The following classes of statements exist in BASIC.

1. Descriptive
2. Input/Output
3. Data Movement and Computational
4. Control

These classes are defined solely for the convenience of presenting the BASIC statements, and in a few cases, the choice of class is arbitrarily chosen for a particular BASIC statement.

In the following descriptions of the individual statements, the syntactical definition is stated without the required preceding statement number and the following carriage return.

The program is considered to be ordered with respect to statement numbers in ascending numerical order. The phrase "previous statement" refers to the statement with the next lower statement number. The phrase "next logical statement" refers to the statement with the next higher statement number. A program is executed in the logical order (i.e., in ascending sequence of statement numbers) unless that order is modified by the execution of a GOTO, GOSUB, IF, or RETURN statement.

2.2.4.1 Descriptive Statements. BASIC has two statements which are referred to as descriptive statements:

DIM	is used to explicitly dimension an array
DEF	is used to define a user function

2.2.4.1.1 DIM Statement

Syntax

DIM array-dimension-specification

[,array-dimension-specification]...

Description

The DIM statement is non-executable and is used to explicitly define array sizes. Arrays of one or two dimensions can be defined; there is no limitation on the size of an array other than the availability of storage. Indexing begins at 1 for each dimension.

DIM statements may appear anywhere in a program, but an array name cannot appear in a DIM statement after (i.e., according to an ascending logical sequence of statement numbers) it has either been defined implicitly or explicitly. If a subscripted variable is used without being defined in a DIM statement, the processor automatically defines an array with the appropriate number of dimensions of ten members each. An array must either be defined implicitly or explicitly in a DIM statement before its use in a MAT statement. (See Section 2.2.2.1.3 for a detailed description of dimensioning rules.) A MAT statement may redimension an array only through the ZER,

CON, IDN functions, and the MAT INPUT, MAT GET, MAT READ I/O statements.

2.2.4.1.2 DEF Statement

Syntax

```
DEF FN alphabetic-character
  (arithmetic-variable) =
  arithmetic-expression
```

Description

The DEF statement is non-executable and is used to define user functions. A user function is called in other statements in a similar manner as the intrinsic functions (see Section 2.2.5.2). When used in an executable statement, the function name, FN alphabetic-character, is specified followed by an arithmetic expression in parentheses. When the function is called, the value of this argument is substituted for any appearance of the dummy variable in the DEF expression. This dummy variable is the variable name given as the argument in the DEF statement. The use of a variable name in a DEF statement does not assign any value to the variable name used. The evaluated arithmetic value of the expression is returned to the calling expression to replace the user function reference in the expression.

The function cannot contain references to itself or to other functions which reference it. Function references may not be recursive at any level. A DEF statement can appear anywhere in a program, and the function may be referenced anywhere within a BASIC program.

2.2.4.2 Input/Output Statements. Input/output statements are used to transmit data to a program for processing and from a program after processing. Input data may be transmitted to a program internally (READ), externally from an interactive terminal (INPUT), or externally from an on-line storage device (GET). Output data may be transmitted from a program externally to an interactive terminal (PRINT or PRINT USING) or to an on-line storage device (PUT).

Input/output statements are of two forms:

1. A simple form which performs I/O upon scalar values and lists of scalar values.
2. A complex form which performs I/O upon arrays. The syntax of this form is the simple form prefaced with the keyword MAT.

The input/output statements are:

Simple Form	Complex Form
GET	MAT GET
INPUT	MAT INPUT
PRINT	MAT PRINT
PRINT USING	MAT PRINT USING
PUT	MAT PUT
READ	MAT READ

The internal file statements are:

```
DATA
RESTORE
```

The external file control statements are:

```
RESET
CLOSE
```

Input/output statements are further classified and described in terms of the three modes of I/O they perform.

1. I/O of internal files
2. I/O of interactive terminals
3. I/O of on-line storage devices

2.2.4.4.1 Internal File I/O Statements. The internal file I/O statements in BASIC are:

DATA	to create an internal data file.
RESTORE	to position the data file to its beginning.
READ MAT READ	to assign the values of the data file to variables or arrays.

2.2.4.2.1.1 DATA Statement

Syntax

```
DATA  constant [,constant]...
```

Description

DATA statements are non-executable and are used to create a data file, internal to the program that can be used to initialize variable references in READ statements during execution. They can be placed anywhere in the program, but the logical order of the DATA statements in a program determines the order of the values in the data file. All DATA statements in a program are collected into a single data file

2.2.4.2.1.2 READ Statement

Syntax

READ variable-reference [,variable-reference]...

Description

At the beginning of execution of a program, a pointer is set to the first value in the data file, if one exists. When a READ statement is encountered, successive values from the data file are assigned to the variables in the READ statement beginning at the current data file position. The execution of a READ statement is in error if there are no DATA statements in the program.

Each data value read must be of the same type (character or arithmetic) as the variable to which it is assigned. If the data file is exhausted and unassigned variables remain in the READ statement, an error results. Subscripts in READ statements are evaluated as they occur. (Thus, an assigned variable reference in a READ statement may be subsequently used as a subscript in that statement).

Truncation will occur if the arithmetic data exceeds the implementation supported precision. Zero fill will occur if the arithmetic data is less than the implementation supported precision.

2.2.4.2.1.3 RESTORE Statement

Syntax

RESTORE [string-character]...

Description

The RESTORE statement resets the DATA pointer to the first value in the data file. The RESTORE statement is ignored if there are no DATA statements in the program. The string characters following the keyword RESTORE is simply a comment.

2.2.4.2.1.4 MAT READ Statement

Syntax

MAT READ arithmetic-array-reference [,arithmetic-array-reference]...

Description

The array references in the MAT READ statement are assigned values from the data file by rows, beginning at the current data file position. Array references previously must either have been implicitly defined or explicitly defined by a DIM statement. If the optional expressions follow the array names, the truncated integer portion of the expressions is used to redimension the arrays before the values are assigned. Redimensioning must not change the number of original dimensions nor exceed the number of original members.

At the beginning of execution of a program, a pointer is set to the first value in the data file, if one exists. When a MAT READ statement is encountered, successive values from the data file are assigned to the arrays in the MAT READ statement.

Each data value read must be arithmetic. If the data file is exhausted and unassigned arrays or array members remain in the MAT READ statement, an error results.

Conversion of arithmetic input is a function of the precision specification (S or L) which is employed during execution. This implies that all arithmetic data which is read is either truncated or zero padded to appropriate precision specification (S or L), if required, prior to assignment to internal variables.

The pointer may be reset by executing the RESTORE statement. The MAT READ statement is invalid if there are no DATA statements in the program.

2.2.4.2.2 Interactive Terminal I/O Statements. The interactive terminal I/O statements in BASIC are:

INPUT	to input data into a program from a terminal to be assigned to variables.
MAT INPUT	to input data into arithmetic arrays within a program from a terminal.
PRINT	to output data to a terminal.
MAT PRINT	to output values of arithmetic arrays to a terminal.
PRINT USING	to output formatted data to a terminal.
MAT PRINT USING	to output formatted values of arithmetic arrays to a terminal.

This group of statements is referred to as interactive terminal statements because they either request data to be supplied, or

they generate printed output back to the user at execution time.

2.2.4.2.2.1 INPUT Statement

Syntax

```
INPUT variable-reference [,variable-
  reference]...
```

Description

Execution of an INPUT statement causes a question mark to be printed at the terminal, at the current print position, which is then activated for input. The user enters a list of values which are assigned to the variable references in the INPUT statement. Each value entered must be of the same type (character or arithmetic) as the corresponding variable reference. Each value must be separated from the next value by a comma. A carriage return ends the series of values entered.

Only a single line of input may be entered in response to an INPUT statement.

Conversion of arithmetic input is a function of the precision specification (S or L) which is employed during execution. This implies that all arithmetic data which is read is either truncated or zero padded to appropriate precision specification (S or L) prior to assignment to program variables.

A character constant in the input stream must be delimited by single or double quotation marks.

Subscripts on variable references are evaluated as they occur. An error occurs if the number of values entered does not equal the number of variable references in the INPUT statement, or any of the values are of the wrong data type.

The procedure for retry or re-entering of data, after an error, is implementation defined.

2.2.4.2.2.2 MAT INPUT Statement

Syntax

```
MAT INPUT arithmetic-array-reference
  [,arithmetic-array-reference]...
```

Description

Execution of a MAT INPUT statement causes a question mark to be printed at the terminal which is then activated for input. The user enters a list of values which are assigned to members of the specified arrays by rows, each new row being requested with two question marks by the system as the previous row is completed. A single question mark is used to indicate a request for input to the first row of an array.

Array references may be one- or two-dimensional arrays which previously must either have been implicitly or explicitly defined by a DIM statement. If the optional expressions follow the array names, the truncated integer portion of these expressions is used to redimension the arrays before the values are entered.

Redimensioning must not change the number of original dimensions nor exceed the number of original members.

Conversion of arithmetic input to internal format is a function of the precision specification (S or L) which is employed during execution. This implies that all arithmetic data which is read is either truncated or zero padded to appropriate precision specification (S or L) prior to assignment to internal variables.

Each value is separated from the next value by a comma. The final entry for each row of each array is followed by a carriage return from the user to signify end of row. If a line is full and input data remains to be entered, the last value entered must be followed by a comma before the carriage is returned to continue. An error will occur if the number of values entered for a row does not equal the number of members in the corresponding array or if any of the values are not arithmetic. The procedure for retry or re-entering of data, after an error, is implementation defined.

2.2.4.2.2.3 PRINT Statement

Syntax

```
PRINT [print-reference] [character-constant
  [,|;] print-reference| [character-
  constant]{,|;} [print-reference]]...
  [character- constant|,|;]
```

Description

Each data item in the PRINT statement (print-reference, character-constant, or null) is converted to a specified output format and printed at the terminal, the carriage being positioned as specified by the delimiting character (comma, null delimiter, or semicolon) or item following the data item being considered. A null delimiter is indicated by the absence of an explicit delimiter, either a comma or a semicolon.

Each line is constructed from two types of print zones, full or packed. Print zones are defined relative to the carriage position at which a data item begins. The length of a line cannot be less than a full print zone; a full print zone consists of 18 characters.

If the data item is an arithmetic expression, the size of the packed print zone is determined by the size of the converted field (including the sign, digits, decimal point, and exponent) as follows:

Converted Print Field	Packed Print Zone
2- 4 characters	6 characters
5- 7 characters	9 characters
8-10 characters	12 characters
11-13 characters	15 characters
14-17 characters	18 characters

Short form apparent precision or significant number of digits will be referred to throughout this standard as S.

Where:

S≥6 significant digits

Long form apparent precision or significant number of digits will be refferred to throughout this standard as L.

Where:

L≥11 significant digits

If the data item is a character reference, the size of the packed print zone is 18 characters minus any trailing blanks.

If the data item is a character constant, the size of the packed print zone equals the length of the string enclosed by quotation marks.

Each data item is converted to output format as follows:

1. Arithmetic expressions in short-form arithmetic are formatted as follows:

   a. I-format consisting of a sign (blank or minus) and up to S significant decimal digits for integers whose absolute value is greater than or equal to zero and less than 1E+S. Printed values are rounded.

   b. E-format consisting of a sign (blank or minus), up to S significant decimal digits, a decimal point following the first digit, the letter "E", and a signed exponent for numbers, not included in the I-format described above, whose absolute value is less than 1E-1 or greater than or equal to 1E+S. Printed values are rounded.

   c. F-format consisting of a sign (blank or minus), up to S significant digits, and a decimal point in the proper position for numbers not included in the I- or E-formats described above. Printed values are rounded.

2. Arithmetic expressions in long form arithmetic are formatted as follows:

   a. I-format consisting of a sign (blank or minus) and up to eleven significant decimal digits for integers, whose absolute value is greater than or equal to zero and less than 1E+L. Printed values are rounded.

   b. E-format consisting of a sign (blank or minus), and up to eleven significant decimal digits, a decimal point following the first digit, the letter "E", and a signed exponent for numbers not included in the I-format described above, whose absolute value is less than 1E-1 or greater than or equal to 1E+L. Printed values are rounded.

   c. F-format consisting of a sign (blank or minus), up to L significant digits, and a decimal point in the proper position for numbers not included in the I- or E-formats described above. Printed values are rounded.

The converted data item will be printed as follows:

1. If the data item is an arithmetic expression, the converted field will be printed as follows:

   a. If the line contains sufficient space to accommodate the value, printing will start at the current carriage position.

   b. If the line does not contain sufficient space to accomodate the value, printing will start at the beginning of the next line.

2. If the data item is a character reference or a character constant, the converted field will be printed as follows:

   a. If the delimiting character is a comma and at least 18 spaces

remain on the line, printing will start at the current carriage position. If the end of the print line is encountered before a character constant is exhausted, printing of the remaining characters will begin on the next line.

b. If the delimiting character is a comma and fewer than 18 spaces remain on the line, printing will start at the beginning of the next print line. If the end of the line is encountered before a character constant is exhausted, printing of the remaining characters will begin on the next line.

c. If the delimiting character is not a comma, printing will start at the current carriage position. If the end of the print line is encountered before the data item is exhausted, printing of the remaining characters will begin on the next line.

After the converted data item has been printed, the carriage will be positioned as specified by the delimiting character:

1. If the data item is a print reference, the carriage will be positioned as follows:

   a. If the delimiter is a comma, the carriage will be moved past any remaining spaces in the full print zone; if the end of the print line is encountered, the carriage will be moved to the beginning of the next print line.

   b. If the delimiter is a semicolon, the carriage will be moved past any remaining spaces in the packed print zone; if the end of the print line is encountered, the carriage will be moved to the beginning of the next print line.

   c. If the delimiter is null (not end of statement) followed by a character constant, the carriage is left at the print position immediately following the data item.

   d. If the delimiter is null (end of statement), the carriage will be moved to the beginning of the next print line.

2. If the data item is a character constant, the carriage will be positioned as follows:

   a. If the delimiter is a comma, the carriage will be moved past any remaining spaces in the full print zone; if the end of the print line is encountered, the carriage will be moved to the beginning of the next line.

   b. If the delimiter is null (not end of statement) or is a semicolon, the carriage is left at the print position immediately following the data item.

   c. If the delimiter is null (end of statement), the carriage will be moved to the beginning of the next print line.

3. If the data item is null, the carriage will be positioned as follows:

   a. If the delimiter is a comma, the carriage will be moved 18 spaces; if the end of the line is encountered, the carriage will be moved to the beginning of the next print line.

   b. If the delimiter is a semicolon, the carriage will be moved three spaces; if the end of the print line is encountered, the carriage will be moved to the beginning of the next line.

   c. If the delimiter is null (end of statement) and the null is the first data item in the list, the carriage will be moved to the beginning of the next print line.

2.2.4.2.2.4 MAT PRINT Statement

Syntax

```
MAT PRINT arithmetic-array
  [{,|;}arithmetic-array]...[,|;]
```

Description

Each array in a MAT PRINT statement, which previously must either have been defined implicitly or explicitly by a DIM statement, is printed in row-major order. The first row of each array begins at the start of a new line, and is separated from the preceding line by two blank lines. The remaining rows of each array begin at the start of a new line, and are separated from the preceding line by a single blank line. After the final or only array has been printed, the carriage will be repositioned to a new print line. One- or two-dimensional arrays may be printed.

Each array member is converted to a specified output format and printed, the carriage being repositioned as specified by the delimiting character following the array name. (See Section 2.2.4.2.2.3, PRINT Statement for specific format information on print zones and conversion of values.)

The converted array member will be printed at the terminal as follows:

1. If the line contains sufficient space to accommodate the value, printing will start at the current carriage position.

2. If the line does not contain sufficient space to accommodate the value, printing will begin at the start of the next line.

After the converted member has been printed, the carriage will be positioned as specified by the delimiting character:

1. If the delimiter is a comma, the carriage will be moved past any remaining spaces in the full print zone; if the end of the line is encountered, the carriage will be moved to the beginning of the next line.

2. If the delimiter is a semicolon, the carriage will be moved past any remaining spaces in the packed print zone. If the end of the line is encountered, the carriage will be moved to the beginning of the next line.

3. If the final delimiter is a null, it will be treated as a comma.

2.2.4.2.2.5 PRINT USING and Image Statements

Syntax

```
PRINT USING statement-number
  [,scalar-reference]...

: [{not # string-character}...|
  conversion-specification]...
```

Description

Each scalar reference in the PRINT USING statement is edited into a line as directed by an Image statement, and the line is printed at the terminal. The statement number in the PRINT USING statement is the statement number of the associated Image statement.

Image statements are non-executable and may be placed anywhere in a program; they specify format pictures for single print lines. String characters appearing in an Image statement are printed exactly as they are entered. Conversion specifications appearing in an Image statement specify Integer, Fixed Point, or Floating Point format (see Section 2.3 BASIC Language Syntax). Each scalar reference is edited (in order of appearance in the PRINT USING statement) into a corresponding conversion specification (in order of appearance in the referenced Image statement).

If the PRINT USING statement contains at least one scalar reference and no conversion specification appears in the referenced Image statement, an error occurs. If the number of scalar references in the PRINT USING statement otherwise exceeds the number of conversion specifications in the Image statement, a carriage return occurs at the end of the Image statement and the Image statement is reused for the remaining scalar references. If the number of scalar references in the PRINT USING statement is less than the number of conversion specifications in the Image statement, the line is terminated at the first unused conversion specification.

The carriage is repositioned to a new line, if required, before printing the edited line. The carriage is repositioned to a new print line after printing is completed.

Each scalar reference is converted to output format as follows:

1. The meaning of a scalar reference is extracted from the specified string and edited into the line, replacing all elements in the conversion specification (including sign, #, decimal point, and |||| or !!!!). If an edited character reference is shorter than the conversion specification, blank padding occurs on the right. If an edited character reference is longer than the conversion specification, truncation occurs on the right. A character constant containing no characters results in blank padding of the entire conversion specification.

2. An arithmetic expression is converted in accordance with its conversion specification as follows:

   a. If the conversion specification contains a plus sign and the expression value is positive, a plus sign is edited into the line.

   b. If the conversion specification contains a plus sign and the

expression value is negative, a minus sign is edited into the line.

c. If the conversion specification contains a minus sign and the expression value is positive, a blank is edited into the line.

d. If the conversion specification contains a minus sign and the expression value is negative, a minus sign is edited into the line.

e. If the conversion specification does not contain a sign, the expression value is negative and the conversion specification is large enough to contain the number and the minus sign, then the negative number will be printed. If the expression value is positive and the conversion specification is large enough to contain the number, the number is printed without a sign. Otherwise, asterisks are edited into the line instead of the expression value.

f. The expression value is converted according to the type of its conversion specification as follows:

I-format: The value of the expression is converted to an integer, rounding any fraction.

F-format: The value of the expression is converted to a fixed-point number, rounding the value or extending it with zeros in accordance with the conversion specification.

E-format: The value of the expression is converted to a floating-point number, rounding the value or extending it with zeros in accordance with the conversion specification. The four exclamation marks or OR signs in an E-format specification are used to indicate the print positions of the exponent part of a floating point number, for example E±xx, where x is any numeric digit.

3. If the length of the arithmetic expression value is less than or equal to the length of the conversion specification, the expression value is edited, right-justified, into the line. If the length of the expression value is greater than the length of the conversion specification, asterisks are edited into the line instead of the expression value.

The following rules define the start of a conversion specification:

1. A number sign is encountered and the preceding character is not a number sign decimal point, plus sign, or minus sign.

2. A plus or minus sign is encountered, which is followed by:

   a. A number sign or

   b. A decimal point which is followed by a number sign.

3. A decimal point is encountered, which is followed by a number sign and:

   a. The preceding character is not a number sign, plus sign, or minus sign or

   b. The preceding character string is a fixed point conversion specification.

The following rules define the end of a conversion specification that has been started:

1. A number sign is encountered and:

   a. The following character is not a number sign or

   b. The following character is not a decimal point or

   c. The following character is a decimal point and a decimal point has already been encountered or

   d. The following four consecutive characters are not exclamation points or OR signs.

2. A decimal point is encountered and:

   a. The following character is not a number sign or

   b. The following character is another decimal point or

   c. The following four consecutive characters are not exclamation points or OR signs.

3. Four consecutive exclamation points or OR signs are encountered.

2.2.4.2.2.6 MAT PRINT USING Statement

Syntax

```
MAT PRINT USING statement-number,
  arithmetic-array [,arithmetic-array]...
```

Description

Each array in the MAT PRINT USING statement, which previously must have been either implicitly or explicitly defined by a DIM statement, is printed by rows at the terminal according to the format defined by an Image statement (see Section 2.2.4.2.2.5 PRINT USING Statement). The statement number in the MAT PRINT USING statement is the statement number of the Image statement used. One- or two- dimensional arrays may be printed.

If the Image statement does not contain at least one conversion specification, an error occurs. If the number of members in the array row exceeds the number of conversion specifications in the Image statement, a carriage return occurs at the end of the Image statement and the Image statement is reused for the remainder of that row. If the number of members in the array row is less than the number of conversion specifications in the Image statement, the line for that row is terminated at the first unused conversion specification.

The first row of each array begins at the start of a new line and is separated from the preceding line by two blank lines. Each succeeding array row begins at the start of a new line and is separated from the preceding row by one blank line. When the last or only array has been printed, the carriage is repositioned.

2.2.4.2.3 External File I/O Statements. The external file statements in BASIC are:

GET	to input data into a program from an external file to be assigned to variables.
MAT GET	to input data into a program from an external file to be assigned to arithmetic arrays.
PUT	to output data to an external file.
MAT PUT	to output values of arithmetic arrays to an external file.
RESET	to reposition an external file at the beginning.
CLOSE	to deactivate an external file and disassociate it from the program.

This group of statements is referred to as external file I/O statements since they process data and data collections which may be retained at the end of a program and may be accessed at a later time.

BASIC external files are processed as a continuous stream of arithmetic constants expressed as a floating point notation in a character format, and character data expressed as character constants enclosed by quotes. Data items on external files are separated from each other by a blank.

Arithmetic constants are written onto an external file in either S or L precision (see Section 2.2.4.2.2.3 PRINT Statement) depending upon the mode of execution. A file generated with S precision arithmetic constants may be accessed in a long precision execution of a program with the extra significance generated by zero padding. Similarly, a file generated with L precision arithmetic constants may be accessed in a short precision execution of a program with the extra digits being truncated. However, an access of a long precision number which would cause a loss of magnitude would cause an error.

Character data is written onto external files as eighteen characters delimited by single quotation marks. Character strings read into character variables from an external file are either extended to eighteen characters with blank padding on the right, if the source field has fewer than eighteen characters, or directly assigned if equal to eighteen characters.

2.2.4.2.3.1 GET Statement

Syntax

```
GET file-reference, variable-reference
  [,variable-reference]...
```

Description

The variable references in the GET statement are assigned values read from the specified file, beginning at the current file position. Each value read must be of the same type (character or arithmetic) as the corresponding variable reference in the GET statement. Subscripts in variable references are evaluated as they occur since the variable references are assigned from left to right.

Conversion of arithmetic input to internal format is a function of the precision

specification (S or L) which is employed during execution. This implies that all arithmetic data which is read is either truncated or zero padded to appropriate precision specification (S or L) prior to assignment to internal variables.

A file is activated for input by the first execution of a GET or MAT GET statement for the file, at which point the file is positioned at the beginning and the variable references are assigned the values read from the file. A file is deactivated when it is closed or at the end of execution of a program.

A file currently activated as an output file cannot be referenced by a GET statement. It must first be closed.

2.2.4.2.3.2 MAT GET Statement

Syntax

```
MAT GET file-reference,
  arithmetic-array-reference
  [,arithmetic-array-reference]...
```

Description

The array references in the MAT GET statement are assigned values from the specified file by rows, beginning at the current file position. Array references may be one- or two-dimensional arrays, which previously must have been either implicitly or explicitly defined by a DIM statement. If the optional expressions follow the array names, the truncated integer portion of these expressions is used to redimension the arrays before the values are entered. Redimensioning must not change the number of original dimensions nor exceed the number of original members.

Whether short-form arithmetic is specified and the file was created in long-form, or long-form arithmetic is specified and the file was created in short-form, all arithmetic data read is truncated to an implementation defined number of significant digits or zero filled to an implementation defined number of digits.

A file is activated for input by the first execution of a MAT GET or GET statement for the file. The file is positioned at the beginning and then the arrays are assigned the values read from the file. A file is deactivated when it is closed or at the end of execution of a program.

A file currently activated as an output file cannot be referenced by a MAT GET statement. It must first be closed.

2.2.4.2.3.3 PUT Statement

Syntax

```
PUT file-reference, scalar-reference
  [,scalar-reference]...
```

Description

Scalar references in the PUT statement are written into the specified file beginning at the current file position.

Whether short- or long-form arithmetic is specified, arithmetic data written will be truncated to an implementation defined number of significant digits or zero filled to an implementation defined number of digits.

A file is activated for output by the first execution of a PUT or MAT PUT statement for the file. The file is positioned to the beginning and the scalar references are written into the file. A file is deactivated when it is closed or at the end of execution of a program.

A file currently activated as an input file cannot be referenced by a PUT statement. It must first be closed.

2.2.4.2.3.4 MAT PUT Statement

Syntax

```
MAT PUT file-reference, arithmetic-array
  [,arithmetic-array]...
```

Description

Values from the specified array are written into the specified file by rows, beginning at the current file position. The arrays previously must have been defined either implicitly or explicitly by a DIM statement.

Whether short- or long-form arithmetic is specified, all arithmetic data written will be truncated to an implementation defined number of significant digits or zero filled to an implementation defined number of digits.

A file is activated for output by the first execution of a MAT PUT or PUT statement. The file is positioned at the beginning and then the arrays are written into the file. A file is deactivated when it is closed or at the end of execution of a program.

A file currently activated as an input file cannot be referenced by a MAT PUT statement. It must first be closed.

2.2.4.2.3.5 RESET Statement

Syntax

```
RESET file-reference [,file-reference]...
```

Description

File references are repositioned at the beginning of the file. If a specified file is not active, its appearance in the RESET statement will be ignored.

2.2.4.2.3.6 CLOSE Statement

Syntax

```
CLOSE  file-reference [,file-reference]...
```

Description

The CLOSE statement is used to deactivate a file, that is, it disassociates a file from its attribute, input or output. An implicit CLOSE is executed for each active file at completion of the execution of a program. If a file is to be used both as an input file and an output file during program execution, it must be closed between input and output references. If a specified file is not active, its appearance in the CLOSE statement will be ignored. An abnormal termination of a program due to an error raised in the program will cause all files to be closed.

2.2.4.3 Data Movement and Computational Statements. The data movement statements referred to in this section deal solely with internal data assignment of an expression to a variable. The two forms of assignment are:

LET	which assigns the value of a scalar expression to a variable.
MAT Assignment	which assigns the values of an array expression to an arithmetic array.

2.2.4.3.1 LET Statement

Syntax

```
[LET]  {arithmetic-reference
        [,arithmetic-reference]...  =
        arithmetic-expression|
        character-reference
        [,character-reference]...  =
        character-expression}
```

Description

The value of the arithmetic or character expressions are evaluated once and assigned to the arithmetic or character references in a left to right order as in the input statements. The exact expansion of a multiple LET statement is as follows:

```
LET var-1 [(sub11[,sub12])],...var-n [(sub
    n1[,sub-n2])]=exp
```

is equivalent to:

```
LET temp = exp

[LET tsub11 = sub11]

[LET tsub12 = sub12]

LET var-1 [(tsub11 [,tsub12])] = temp
  .
  .
  .
[LET tsub-n1 = sub-n1]

[LET tsub-n2 = sub-n2]

LET var-n [(tsub-n1 [,tsub-n2])] = temp
```

A general rule is that all multiple list assignments, either by a LET or an input statement (GET, INPUT, READ), are to be treated in the above manner. That is, subscripts for the first variable are evaluated and then the assignment is made followed by the evaluation of the second variable's subscripts and its assignment. This process continues in a left to right order until the list is exhausted.

2.2.4.3.2 MAT Assignment Statement

Syntax

```
MAT arithmetic-array = {arithmetic-array|
  arithmetic-array
  {+|-} arithmetic-array|arithmetic-array
  * arithmetic-array|
  (arithmetic-expression)
  * arithmetic-array|{CON|ZER|IDN}
  [arithmetic-array-subscript]|{INV|TRN}
  (arithmetic-array)}
```

Description

The MAT statement evaluates the array expression to the right of the equal sign and assigns the result to the array to the left of the equal sign. The dimensions specified for the array on the left must conform to the array expression on the right. If the array expression involves matrix multiplication, inversion, or transposition, the array named on the left must not appear in the expression. An array must be defined either implicitly or

explicitly by a DIM statement before it can appear in a MAT statement.

The array operations available are: (where A and B are arithmetic array names)

A	A simple array reference
A + B	Sum of two matrices
A - B	Difference of two matrices
A * B	Product of two matrices[1]
(e) * A	Product of the scalar value of e and matrix A
ZER [(e_1[,e_2])]	produces e_1 [by e_2] zero array[2]
CON [(e_1[,e_2])]	produces e_1 [by e_2] unity array[2]
IDN [(e_1,e_1)]	produces e_1 by e_1 identity matrix[1] [2]
INV(A)	Inverse matrix of A[1] [3]
TRN(A)	Transpose matrix of A[1]

If e_1 is not specified for ZER, CON, IDN functions, the dimensions used are those of the array being assigned.

Notes:

1. Restricted to two-dimensional arrays only.
2. If the optional expressions are included, the truncated integer portion of these expressions is used to redimension the arrays on the left of the equal sign before the operation is performed. Redimensioning must not change the original number of dimensions nor exceed the original number of members.
3. The INV of a singular array will cause an error. The action of this error is implementation defined.

2.2.4.4 Control Statements. Control statements in BASIC are used to direct the flow of execution of a program. The following statements constitute the set of control statements in BASIC.

GOTO	Transfers control, unconditionally or conditionally, to a specific statement.
GOSUB	Transfers control unconditionally to a group of statements.
RETURN	Returns control to the first executable statement following the last active GOSUB statement.
FOR	Is the initial delimiter of a group of statements which will continue to execute a number of times until an iteration criteria is satisfied.
NEXT	Is the end delimiter of a FOR group.
IF	Is a conditional branch statement dependent on the truth or false value of a relational expression.
PAUSE	Causes suspension of the execution of a program until it is resumed by a user.
STOP	Causes the end of a program execution.
END	Indicates the end of a program and therefore is logically the last statement in a program.

The control statements are defined in the following subsections.

2.2.4.4.1 GOTO Statement

Syntax

```
GOTO statement-number
[[,statement-number]...
     ON arithmetic-expression]
```

Description

Execution of a simple GOTO statement (without the ON option) causes an unconditional transfer of control to the statement whose statement number is specified. If the ON option is employed, the GOTO is called a "computed GOTO." The words GOTO are followed by an n-element statement-number list. The value of the arithmetic expression is computed at execution time; the program branches to element i of the statement-number list, where i is the truncated integer value of the arithmetic expression. If $i < 1$ or $i > n$, control is passed to the next logically executable statement in the program.

If the statement branched to is a non-executable statement, control will be passed to the first logically executable statement following the statement specified.

2.2.4.4.2 GOSUB and RETURN Statements

Syntax

```
GOSUB statement-number

RETURN [string-character]...
```

Description

Execution of a GOSUB statement causes a transfer of control to the statement whose statement number is specified. The GOSUB statement also sets up a return path such that, when a RETURN statement is executed, control is returned to the next logically executable statement following the last GOSUB statement executed. An active GOSUB is one where a GOSUB statement has been executed without a complimentary execution of a RETURN statement. GOSUB statements may be multiply active in a recursive manner.

GOSUB statements may be used in any manner but care should be used in defining recursive GOSUB loops, i.e., a GOSUB into an area of the program that contains a GOSUB leading back to the first GOSUB. This could result in an infinite loop.

If the statement branched to is a non-executable statement, control will be passed to the first logically executable statement following the specified statement.

Execution of a RETURN statement without an active GOSUB will cause an error.

2.2.4.4.3 FOR and NEXT Statements

Syntax

```
FOR arithmetic-variable = arithmetic-
  expression

   TO arithmetic-expression

   [STEP arithmetic-expression]

NEXT arithmetic-variable
```

Description

The FOR and NEXT statements delineate a "FOR loop": a set of statements which are executed zero or more times. The FOR statements are paired because the same arithmetic variable occurs in both statements. FOR-NEXT pairs must be properly nested inside each other. That is, a FOR-NEXT loop properly nested within another FOR-NEXT loop must be completely enclosed within the outer loop. FOR-NEXT loops may not overlap each other.

The three arithmetic expressions in the FOR statement are called the initial value, the final value, and the increment, respectively. These expressions are evaluated only during initial execution of the FOR statement, and are not affected by any statements within the FOR loop. The arithmetic variable is called the control variable and may be modified within the FOR loop. If the initial value is greater than (less than for negative increments) the final value at evaluation time, the loop is not executed and the value of the control variable is left unchanged.

When the loop is first executed, the control variable is set to the initial value. The statements in the loop are executed, and the increment is added to the control variable. This process continues until the control variable is greater than (less than for negative increments) the final value. At this time control is passed to the first executable statement logically following the NEXT statement. At this point, the control variable has a value equal to the value which caused the loop to fail less the value of the STEP expression.

If the STEP arithmetic expression is omitted, it is assumed to be +1. Transfer of control into or out of a FOR loop is allowable within the constraints that a NEXT statement cannot be executed if its associated FOR statement has not been executed and is, therefore, inactive.

A FOR loop is active as long as a FOR statement has been executed, and the loop has not been completed. An active FOR-loop may be re-entered via the FOR-statement, but the effect is an immediate deactivation of the previous generation of the FOR-loop.

An exact expansion of the FOR loop is as follows:

```
          FOR v = exp-1   TO exp-2

            [STEP exp-3]
             .
             .
             .

          NEXT v
```

is equivalent to:

```
10   e1 = exp-1

20   e2 = exp-2

30   e3 = exp-3|e3 = 1  if the STEP
          is not explicitly stated
```

```
40    IF e3<0   GO TO 70

50    IF e1>e2  GO TO 210

60    GO TO 80

70    IF e1<e2  GO TO 210

80    v = e1

90      .
        .
        .
      loop
        .
        .
        .
150   v = v + e3

160   IF e3≥0   GO TO 190

170   IF v<e2   GO TO 200

180   GO TO 90

190   IF v<=e2  GO TO 90

200   v = v - e3

210     .
        .
        .
```

2.2.4.4.4 IF Statement

Syntax

```
IF {arithmetic-expression
     relational-operator
     arithmetic-expression|
     character-expression
     relational-operator
     character-expression}
     {THEN|GOTO} statement-number
```

Description

The two arithmetic or character expressions in the IF statement are compared. If the specified relation is satisfied, control is passed to the statement whose statement number is specified. If the statement branched to is a non-executable statement, control will be passed to the first logically executable statement following the specified statement. If the specified relation is not satisfied, control is passed to the next logically executable statement in the program.

Relational operations on character data is always performed on strings of 18 characters in length by previously extending the strings on the right with blanks, or by truncation on the right when required.

2.2.4.4.5 PAUSE Statement

Syntax

```
PAUSE [string-character]...
```

Description

This statement causes program execution to be interrupted after printing the following message:

```
PAUSE AT LINE statement-number
```

where the statement-number is the statement number of the PAUSE statement.

The procedure for subsequent resumption of program execution is implementation defined.

2.2.4.4.6 STOP Statement

Syntax

```
STOP [string-character]...
```

Description

Program execution is terminated. This statement causes exactly the same action as an END statement during execution. However, unlike the END statement which is the last logical statement in the program, the STOP statement may appear anywhere in the program.

2.2.4.4.7 END Statement

Syntax

```
END [string-character]...
```

Description

The END statement indicates the logical end of a program, i.e., any statements numerically following END are ignored. END is optional and, if omitted, is assumed to follow the highest numbered statement in the program.

2.2.4.5 REM Statement

Syntax

```
REM [string-character]...
```

Description

The REM statement is non-executable and is used to insert a comment into a program. It can be placed anywhere in a program.

2.2.5 Functions. A function reference is a shorthand notation for expressing an algorithm to be evaluated, resulting in the function reference being replaced at execution time with a numeric value. Three types of functions will be discussed in this section:

User functions	where the user defines the algorithm.
Intrinsic function	parameterized invocations of a system defined algorithm.
Internal constants	non-parameterized invocation of a system defined algorithm which returns a constant result.

2.2.5.1 User Functions. User functions are defined by the user through the use of the DEF statement. Once defined, they may be used like intrinsic functions in arithmetic expressions.

During execution, any statement that references the user function will supply an expression argument whose value is used in any function in place of any dummy variable. All other arithmetic data elements in the user function take on the values current at the time of function invocation.

2.2.5.2. Intrinsic Functions. IBM-supplied routines can be used to compute the values of various mathematical functions. Each of these functions has a single argument (optional with RND), which can be any valid arithmetic expression and produces a single arithmetic result. An invalid argument will cause an error. The DET function is an exception in that its argument must be a reference to an arithmetic array. A function reference may be used anywhere in arithmetic expressions that a variable or a constant can be used. The intrinsic functions defined are:

SIN (X)

Computes the sine of X radians.

COS (X)

Computes the cosine of X radians.

TAN (X)

Computes the tangent of X radians.

COT (X)

Computes the cotangent of X radians.

SEC (X)

Computes the secant of X radians.

CSC (X)

Computes the cosecant of X radians.

ASN (X)

Computes the arc sine (in radians) of the real number X. where:

$(-\ /2) \leq ASN(X) \leq (\ /2)$

ACS (X)

Computes the arc cosine (in radians) of the real number X. where:

$0 | ACS(X) \leq$

ATN (X)

Computes the arctangent (in radians) of the real number X. where:

$-(\ /2) < ATN(x) < (\ /2)$

HSN (X)

Computes the hyperbolic sine of the real number X.

HCS (X)

Computes the hyperbolic cosine of the real number X.

HTN (X)

Computes the hyperbolic tangent of the real number X.

DEG (X)

Computes the number of degrees in X radians.

RAD (X)

Computes the number of radians in X degrees.

EXP (X)

Computes the value of e raised to the X power.

ABS (X)

Computes the absolute value of the real number X by forcing it positive.

LOG (X)

Computes the natural logarithm (base e) of the positive number X greater than zero.

LTW (X)

Computes the logarithm to the base 2 of the positive number X greater than zero.

LGT (X)

Computes the logarithm to the base 10 of the positive number X greater than zero.

SQR (X)

Computes the square root of the non-negative number X.

INT (X)

Returns the integer part of the real number X. If X < 0 then the value returned is the smallest integer ≥ X. INT(-3.14) = -3. If X ≥ 0 then the value returned is the largest integer ≤ X. INT(3.14) = 3.

SGN (X)

Returns the sign of the real number X. If X < 0, SGN (X) = -1; if X = 0, SGN (X) = 0; if X > 0, SGN (X) = +1.

RND [(X)]

Returns a random number in the interval between 0 and 1 according to a uniform distribution on this interval. Each random number is computed from the previous one according to a fixed algorithm.

The random number generator can be initialized by specifying an argument; the argument may be any number. Subsequent references to RND without using an argument will cause the new number to be generated from the previous one.

Each time RND is called with an argument, the generator is initialized with the absolute value of the argument. If RND is called without an argument and there has been no previous initialization, then the generator will initialize itself, using an implementation defined value.

DET (X)

Returns the value of the determinant of the square arithmetic array X. The array must have been either defined implicitly, or explicitly in a DIM statement before its use as an argument in the DET function.

2.2.5.3 Internal Constants. An internal constant is an arithmetic constant whose value is predefined in the BASIC language. It may be used anywhere an arithmetic constant may be used. The three internal constants defined in terms of "apparent" precision are the mathematical values of pi, e, and the square root of 2.

Name	Short-Form Value	Long-Form Value
&PI	3.14159	3.1415926536
&E	2.71828	2.7182818285
&SQR2	1.41421	1.4142135624

2.3 Syntax of the BASIC Language

The syntax conventions used for the following definitions are as defined in the System/360 DSB, Section 3.30.1, and are summarized and expanded by the following:

a. The underscoring of a character is used to indicate that this is a single printable character. This distinguishes it from a syntactic variable consisting of a printable character optionally followed by any number of blank characters.

b. Lower-case letters represent information that must be supplied by the user.

c. Information contained within brackets [ ] represents an option that can be omitted.

d. The appearance of braces { } indicates that a choice must be made between the items contained in the braces.

e. The appearance of the vertical bar | indicates that a choice must be made between the item to the left of the bar and the item to the right of the bar.

f. An ellipsis (a series of three periods) indicates that the preceding syntactic unit may be repeated any number of times.

g. not syntactic-unit-1 syntactic-unit-2. The "not" operator is used to define a unit which contains all of syntactic-unit-2 except for that which is explicitly stated as syntactic-unit-1. For example, a print-special-character (2.3.1.3) is defined as any EBCDIC character except upper- and lower-case alphabetics, the digits 0 through 9, or the three alphabetic extenders.

2.3.1 Character Set

2.3.1.1 print-alphabetic-character ::= 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|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|@|#|$

2.3.1.2 print-numeric-character ::= 0|1|2|3|4|5|6|7|8|9

2.3.1.3 print-special-character ::= not{print-alphabetic-character| print-numeric-character} any-character[1]

2.3.1.4 alphabetic-character ::= print-alphabetic-character[b]...

2.3.1.5 numeric-character ::= print-numeric-character[b]...

2.3.1.6 string-character ::= any-character[1]

2.3.1.7 relational-operator ::= =|<>|≠|>=|≥|<=|≤|>|<

2.3.1.8 = ::= =[b]...

2.3.1.9 < ::= <[b]...

2.3.1.10 > ::= >[b]...

2.3.1.11 ≠ ::= ≠[b]...

2.3.1.12 ≤ ::= ≤[b]...

2.3.1.13 ≥ ::= ≥[b]...

2.3.1.14 * ::= *[b]...

2.3.1.15 / ::= /[b]...

2.3.1.16 + ::= +[b]...

2.3.1.17 - ::= -[b]...

2.3.1.18 ' ::='[␣]...

2.3.1.19 ! ::= ![␣]...

2.3.1.20 , ::= ,[␣]...

2.3.1.21 ; ::= ;[␣]...

2.3.1.22 . ::= .[␣].. .

2.3.1.23 & ::= &[␣]...

2.3.1.24 (::= ([␣]...

2.3.1.25) ::=)[␣]...

2.3.1.26 A ::= {A|a}[␣]...

2.3.1.27 B ::= {B|b}[␣]...

2.3.1.28 C ::= {C|c}[␣]...

2.3.1.29 D ::= {D|d}[␣]...

2.3.1.30 E ::= {E|e}[␣]...

2.3.1.31 F ::= {F|f}[␣]...

2.3.1.32 G ::= {G|g}[␣]...

2.3.1.33 H ::= {H|h}[␣]...

2.3.1.34 I ::= {I|i}[␣]...

2.3.1.35 J ::= {J|j}[␣]...

2.3.1.36 K ::= {K|k}[␣]...

2.3.1.37 L ::= {L|l}[␣]...

2.3.1.38 M ::= {M|m}[␣]...

2.3.1.39 N ::= {N|n}[␣]...

2.3.1.40 O ::= {O|o}[␣]...

2.3.1.41 P ::= {P|p}[␣]...

2.3.1.42 Q ::= {Q|q}[␣]...

2.3.1.43 R ::= {R|r}[␣]...

2.3.1.44 S ::= {S|s}[␣]...

2.3.1.45 T ::= {T|t}[␣]...

2.3.1.46 U ::= {U|u}[␣]...

2.3.1.47 V ::= {V|v}[␣]...

2.3.1.48 W ::= {W|w}[␣]...

2.3.1.49 X ::= {X|x}[␣]...

2.3.1.50 Y ::= {Y|y}[␣]...

Character Set

2.3.1.51 Z ::= {Z|z}[␣]...

2.3.1.52 $ ::= $[␣]...

2.3.1.53 # ::= #[␣]...

2.3.1.54 @ ::=@[␣]...

2.3.1.55 : ::= :[␣]...

2.3.1.56 | ::= | [␣]...

2.3.1.57 " ::= "[␣]...

2.3.1.58 0 ::= 0[␣]...

2.3.1.59 | ::= |[␣]...

2.3.1.60 ␣ ::= blank character [blank character]...

2.3.2 Conversion Specifications

2.3.2.1 integer-specification ::= [+|-] #[#]...

2.3.2.2 fixed-point-specification ::= [+|-]{[#]... . # [#]...|# [#]... . [#]...}

2.3.2.3 floating-point-specification ::=
{integer-specification|fixed-point-specification} {!!!!|!!!!}

2.3.2.4 conversion-specification ::= integer-specification|fixed-point-specification|
floating-point-specification

2.3.3 Constants

2.3.3.1 integer-constant ::= numeric-character...

2.3.3.2 fixed-point-constant ::= [numeric-character]... . numeric-character...|
numeric-character... . [numeric-character]...

2.3.3.3 floating-point-constant ::= {integer-constant| fixed-point-constant}
E[+|-]numeric-character [numeric-character]

2.3.3.4 internal-constant ::= &PI|&E|&SQR2

2.3.3.5 arithmetic-constant ::= fixed-point-constant| integer-constant|
floating-point-constant| internal-constant

2.3.3.6 character-constant ::= '[' '|not ' string-character]...'| "[" "|not "
string-character]..."

2.3.3.7 constant ::= [+|-] arithmetic-constant| character-constant

2.3.4 Names

2.3.4.1 arithmetic-variable ::= alphabetic-character [numeric-character]

2.3.4.2 arithmetic-array ::= alphabetic-character

2.3.4.3 character-variable ::= alphabetic-character $

2.3.4.4 character-array ::= alphabetic-character $

Names

2.3.4.5 user-function ::= FN alphabetic-character

2.3.4.6 intrinsic-function ::= {SIN|COS|TAN|ATN|EXP|ABS|LOG|LTW|LGT|COT|SEC|CSC|ASN|ACS|HSN|HCS|HTN|DEG|RAD|SQR|INT|SGN} (arithmetic-expression)| DET (arithmetic-array)[7]| RND[(arithmetic-expression)]

2.3.4.7 array-dimension-specification ::= arithmetic-array (not 0 integer-constant [,not 0 integer-constant]) | character-array (not 0 integer-constant)

2.3.4.8 character-array-subscript ::= (arithmetic-expression[2])

2.3.4.9 arithmetic-array-subscript ::= (arithmetic-expression[2] [,arithmetic-expression[2]])

2.3.5 References

2.3.5.1 function-reference ::= user-function (arithmetic-expression)| intrinsic-function

2.3.5.2 character-reference ::= character-variable| character-array character-array-subscript

2.3.5.3 arithmetic-reference ::= arithmetic-variable| arithmetic-array arithmetic-array-subscript

2.3.5.4 variable-reference ::= character-reference| arithmetic-reference

2.3.5.5 arithmetic-array-reference ::= arithmetic-array[arithmetic-array-subscript]

2.3.5.6 file-reference ::= character-constant[10]

2.3.5.7 print-reference ::= arithmetic-expression| character-reference

2.3.5.8 scalar-reference ::= arithmetic-expression| character-expression

2.3.6 Expressions

2.3.6.1 character-expression ::= character-reference| character-constant

2.3.6.2 primitive-arithmetic-expression ::= arithmetic-reference| function-reference| arithmetic-constant| (arithmetic-expression)

2.3.6.3 one-arithmetic-expression ::= [one-arithmetic-expression {**|↑}] primitive-arithmetic-expression

2.3.6.4 two-arithmetic-expression ::= [two-arithmetic-expression {*|/}] one-arithmetic-expression

2.3.6.5 three-arithmetic-expression ::= [three-arithmetic-expression {+|-}] two-arithmetic-expression

2.3.6.6 arithmetic-expression ::= [+|-] three-arithmetic-expression

2.3.7 Statements

2.3.7.1 DATA-statement ::= DATA constant [,constant]...

2.3.7.2 DEF-statement ::= DEF user-function (arithmetic-variable) = arithmetic-expression

2.3.7.3 DIM-statement ::= DIM array-dimension-specification [,array-dimension-specification]...

Statements

2.3.7.4 END-statement ::= END [string-character]...

2.3.7.5 FOR-statement ::= FOR arithmetic-variable = arithmetic-expression TO arithmetic-expression [STEP arithmetic-expression]

2.3.7.6 GET-statement ::= GET file-reference,variable-reference [,variable-reference]...

2.3.7.7 GOSUB-statement ::= GOSUB statement-number

2.3.7.8 GOTO-statement ::= GOTO statement-number [[,statement-number]... ON arithmetic-expression]

2.3.7.9 IF-statement ::= IF {arithmetic-expression relational-operator arithmetic-expression| character-expression relational-operator character-expression} {THEN|GOTO} statement-number

2.3.7.10 Image-statement ::= : [{not # string-character}...| conversion-specification]...

2.3.7.11 INPUT-statement::= INPUT variable-reference [,variable-reference]...

2.3.7.12 LET-statement ::= [LET] {arithmetic-reference [,arithmetic-reference]... = arithmetic-expression|character-reference [,character-reference]... = character-expression}

2.3.7.13 MAT-statement ::= MAT arithmetic-array = {arithmetic-array| arithmetic-array {+|-}arithmetic-array[3]| arithmetic-array * arithmetic-array[4] [5] | (arithmetic-expression) * arithmetic-array| {CON|ZER|IDN[7]} [arithmetic-array-subscript][6]| {INV[7]|TRN} (arithmetic-array)[4]}

2.3.7.14 MAT-GET-statement ::= MAT GET file-reference, arithmetic-array-reference [,arithmetic-array-reference][6]...

2.3.7.15 MAT-INPUT-statement ::= MAT INPUT arithmetic-array-reference [,arithmetic-array-reference][6]...

2.3.7.16 MAT-PRINT-statement ::= MAT PRINT arithmetic array [{,|;} arithmetic-array] ...[,|;]

2.3.7.17 MAT-PRINT-USING-statement ::= MAT PRINT USING statement-number, arithmetic-array [,arithmetic-array]...

2.3.7.18 MAT-PUT-statement ::= MAT PUT file-reference, arithmetic-array [,arithmetic-array]...

2.3.7.19 MAT-READ-statement ::= MAT READ arithmetic-array-reference [,arithmetic-array-reference][6]...

2.3.7.20 NEXT-statement ::= NEXT arithmetic-variable

2.3.7.21 PAUSE-statement ::= PAUSE [string-character]...

2.3.7.22 PRINT-statement ::= PRINT [print-reference] [character-constant [,|;] print-reference | [character-constant] {,|;} [print-reference]]... [character-constant|,|;]

2.3.7.23 PRINT-USING-statement ::= PRINT USING statement-number [,scalar-reference]...

2.3.7.24 PUT-statement ::= PUT file-reference, scalar-reference [,scalar-reference]...

2.3.7.25 READ-statement ::= READ variable-reference [,variable-reference]...

2.3.7.26 REM-statement ::= REM [string-character]...

Statements

2.3.7.27 RESET-statement ::= RESET file-reference [,file-reference]...

2.3.7.28 RESTORE-statement ::= RESTORE [string-character]...

2.3.7.29 RETURN-statement ::= RETURN [string-character]...

2.3.7.30 STOP-statement ::= STOP [string-character]...

2.3.7.31 CLOSE-statement ::= CLOSE file-reference [,file-reference]...

2.3.8 Program Structure

2.3.8.1 statement-number ::= [b]... numeric-character [numeric-character]...

2.3.8.2 basic-statement ::= CLOSE-statement| DATA-statement| DEF-statement|
DIM-statement| GET-statement| GOSUB-statement| GOTO-statement|
IF-statement| Image-statement| INPUT-statement| LET-statement|
MAT-statement| MAT-GET-statement| MAT-INPUT-statement|
MAT-PRINT-statement| MAT-PRINT-USING-statement| MAT-PUT-statement|
MAT-READ-statement| PAUSE-statement| PRINT-statement|
PRINT-USING-statement| PUT-statement| READ-statement| REM-statement|
RESET-statement| RESTORE-statement| RETURN-statement| STOP-statement

2.3.8.3 basic-line ::= statement-number basic-statement statement-end

2.3.8.4 for-line ::= statement-number FOR-statement statement-end

2.3.8.5 next-line ::= statement-number NEXT-statement statement-end

2.3.8.6 for-group ::= for-line [statement-group| for-group|return-line]...next-line[9]

2.3.8.7 end-line ::= statement-number END-statement statement-end

2.3.8.8 basic-program ::={not return-line basic-line|for-group}[for-group|
statement-group|return-line]... [end-line]

2.3.8.9 statement-end ::= [8]

2.3.9 Notes

1. Any character possible in the EBCDIC character set is denoted by any-character. It is assumed that the reader knows what these are.

2. Subscripts for character and arithmetic arrays are arithmetic expressions whose values must be positive and the truncated integer portion greater than zero and not exceeding the current maximum.

3. The arrays must have identical dimensions.

4. The array on the left may not appear on the right.

5. The number of columns in the first array must equal the number of rows in the second array.

6. If dimension specifications are supplied, the dimensions of the array will be changed.

7. The array must be square.

8. Statement-end is a terminal-dependent action which indicates the end of a physical line of input, e.g. carriage return. A BASIC statement may not exceed a physical line of input nor may more than one BASIC statement be entered on a physical line.

9. A proper for-group requires that the arithmetic variable referenced in the NEXT-statement be the same as the arithmetic control variable in the FOR-statement.

10. A file-reference is a character constant of an implementation defined length and character content.

Index

&E 17
&PI 17
&SQR2 17

A

ABS function 21
absolute value function 21
ACS function 21
addition 22
alphabet extenders 13
alphabetic characters 13
arccosine function 21
arcsine function 21
arctangent function 21
argument of function 20
arithmetic arrays (*see* arrays, arithmetic)
arithmetic characters 13
arithmetic constants 16
  in expressions 21
arithmetic data 15
arithmetic data formats 15
arithmetic expressions 21
arithmetic precision 15
arithmetic values 16
arithmetic variables 17
  in arithmetic expression 21
  naming of 20
array expressions 21
  binary 57
  unary 57
array operations (*see* matrix operations)
arrays 18, 19
  arithmetic 19, 21
  character 19, 23
ASN function 21
Assignment statement 27, 29
ATN function 21

B

binary operators 21
blanks
  in statement numbers 11
  use of 14
branch (*see* IF, GOTO statements)

C

character constants 17
  assigned to variables 18
  characters permitted in 13
  in Assignment statement 29
  in character expressions 23
  in IF statement 36
  length of 17
  use of blanks in 14
character data 17
character expressions 23
character format 50
character set of BASIC 13-14
character variables 18
  in expressions 23
  naming of 20
characteristic 15
CLOSE statement 54-55
collating sequence 73-74
comma
  as special character 13
  in PRINT statement 48
comments 13
  in END statement 40
  in PAUSE statement 40
  in REM statement 39
  in RESTORE statement 44
  in STOP statement 40
compilation, end of 40
computed GOTO statement (*see* GOTO statement)
CON function 63-64
constant (*see* arithmetic constants; character constants)
control statements 27, 35-42
control variable 36
conversion of data values 51
conversion specification (*see* format specifications)
COS function 21
cosecant function 21
cosine function 21
COT function 21
cotangent function 21
CSC function 21

D

data representation 15
DATA statement 43
DEF statement 31
DEG function 21
degree function 21
delimiters
  in MAT PRINT statement 66
  in PRINT statement 46
descriptive statements 27, 31-33
DET function 21
  in MAT assignment statement 61
determinant of matrix 21
DIM statement 32
dimensions of arrays 19
division
  symbol 22
  by zero 23
DO-loop (*see* FOR and NEXT statements)
dummy variable 31

E

e (natural log) 17
E-format (*see* floating-point format)
EBCDIC collating sequence 73-74
  in IF statement 36
  in relational expressions 24
END statement 40
EXP function 21
exponent (*see* characteristic)
exponential format (*see* floating-point format)
exponential function 21
expressions 21-22
  (*see also* arithmetic expressions; character expressions)

F

F-format (*see* fixed-point format)
files 43
(*see also* input/output statements)
fixed-point format 15
in Image statement 50-51
floating-point format 15-16
in Image statement 50
FOR statement 36
formalized definition of BASIC 77
format specifications 50
full print zone 47, 66
function reference 20
functions 20
intrinsic 20
user written (*see* DEF statement)

G

GET statement 53
GOSUB statement 38
GOTO as keyword in IF statement 36
GOTO statement 35

H

HCS function 21
HSN function 21
HTN function 21
hyperbolic cosine function 21
hyperbolic sine function 21
hyperbolic tangent function 21

I

identity matrix 62
IDN function 62-63
IF statement 36
I-format (*see* integer format)
image specifications (*see* format specifications)
Image statement 49
implicitly defined arrays (*see* arrays)
INPUT statement 45
input/output statements 43-55
INT function 21
integer format 15
in Image statement 50-51
integral part function 21
internal constants 17
interruption of execution (*see* PAUSE and STOP statements)
intrinsic functions 21
INV function 21
inversion of matrices 61

L

LET statement (*see* Assignment statement)
LGT function 21
literal data (*see* character data)
LOG function 21
logarithm function 21
long precision 15
loop (*see* FOR and NEXT statements)
lower-case letters 13
LTW function 21

M

magnitude of numbers 15
mantissa 15
MAT assignment statement
addition and subtraction 59
CON function 63
identity function 62
inversion function 61
multiplication 59
scalar multiplication 60
simple 58
transpose function 62
ZER function 63
MAT GET statement 68
MAT INPUT statement 65
MAT PRINT statement 66
with INPUT statement 45
with PAUSE statement 40
MAT PRINT USING statement 67
MAT PUT statement 69
MAT READ statement 64
matrix operations 57-70
multiplication 22

N

naming of variables and arrays 20
natural logarithm 17
NEXT statement 36
null 46
numbers
(*see also* arithmetic data)
as characters 13
large and small 17
of lines 11
of statements 11
precision of 15
random 21
numeric characters 13
numeric data (see arithmetic data)

O

one-dimensional arrays (*see* arrays)
operands 22
operators 21-22
order of execution 11

P

packed PRINT zone
in MAT PRINT statement 66
in PRINT statement 47
PAUSE statement 39
pi 17
pointer, data 43
precision 15
PRINT statement 46
with INPUT statement 45
with PAUSE statement 40
PRINT USING statement 49
print zones 47
priority of operators 22
program 11
PUT statement 52

Q

question mark
in INPUT statement 45
in MAT INPUT statement 65
quotation marks 17

R

RAD function 21
radian function 21
random number function 21

READ statement 44
redimensioning arithmetic arrays 57
relational expressions 23-24
  in IF statement 36
relational operators 23-24
  in IF statement 36
REM statement 39
RESET statement 54
RESTORE statement 44
restrictions 71
RETURN statement 38
RND function 21
rounding errors 22

S

sample programs 75
scalar expressions 21
scientific notation 16
SEC function 21
secant function 21
semicolon in PRINT statement 48
SGN function 21
short precision 15
sign of number function 21
SIN function 21
sine function 21
special characters 13-14
SQR function 21
square root function 21
standard output formats 46-47
statement line 11
statements
  categories 11
  definition 27
  executable 11
  non-executable 11
  numbers 11
STEP as keyword in FOR statement 36
STOP statement 40
sub-expression 22
subroutines (*see* GOSUB and RETURN statements)
subscripts of array references 18
subtraction 22

T

TAN function 21
tangent function 21
THEN as keyword in IF statement 36
two-dimensional arrays (*see* arrays)

U

unary operators 21
user-written functions (*see* DEF statement)

V

variables (*see* arithmetic variables; character variables)
vectors 57

Z

ZER function 63
zero-divide 23

READER'S COMMENTS

TITLE: BASIC Language
Reference Manual

FORM: GC28-6837-0

Your comments assist us in improving the usefulness of our publications; they are an important part of the input used in preparing updates to the publications. All comments and suggestions become the property of IBM.

Please do not use this form for technical questions about the system or for requests for additional publications; this only delays the response. Instead, direct your inquiries or requests to your IBM representative or to the IBM Branch Office serving your locality.

Corrections or clarifications needed:

Page *Comment*

Please include your name and address in the space below if you wish a reply.

Thank you for your cooperation. No postage necessary if mailed in the U.S.A.

cut along this line

fold fold

FIRST CLASS
PERMIT NO. 33504
NEW YORK, N.Y.

BUSINESS REPLY MAIL

NO POSTAGE NECESSARY IF MAILED IN THE UNITED STATES

POSTAGE WILL BE PAID BY . . .

IBM CORPORATION
1271 Avenue of the Americas
New York, New York 10020

Attention: PUBLICATIONS

fold fold

IBM

International Business Machines Corporation
Data Processing Division
112 East Post Road, White Plains, N.Y. 10601

IBM World Trade Corporation
821 United Nations Plaza, New York, N.Y. 10017
(International)

Printed in U.S.A. GC28-6837-0

GC28-6837-0

Printed in U.S.A. GC28-6837-0

IBM

International Business Machines Corporation
Data Processing Division
112 East Post Road, White Plains, N.Y. 10601

IBM World Trade Corporation
821 United Nations Plaza, New York, N.Y. 10017
(International)