CALMA

UCS

INTERACTIVE GRAPHIC SYSTEMS
A division of United Computing Systems Inc

gpl II progammer's reference manual

GPL II PROGRAMMER'S REFERENCE MANUAL

FIRST EDITION

April, 1979

Document No. 579-2

PREFACE

This is the First Edition of the GPL II Programmer's Reference Manual. It accompanies the GDS II Reference Manuals, Volume I and II, and the GDS II Menu Manual. It presents an explanation of the GPL II Programming Language, knowledge of which will provide the user the facility to extend the system to meet his needs in the future.

TABLE OF CONTENTS

Section Number and Title **Page**

SECTION 1
INTRODUCTION

The use of the GPL II Programming Language requires a general knowledge of the GDS II System and assumes familiarity with the GDS II Command Language. The GDS II Command Language and the GPL II Programming Language have been designed as different approaches to the same system. Once the procedures for creating GDS II Menus in terms of GDS II Command Primitives have been mastered, only a little additional effort is required to place the complete resources of the GPL II Programming System at the user's disposal.

The progression from system operation to menu definition, and on to programming, follows a natural order. Although there is nothing to be "unlearned", there are new concepts to be mastered at each step and new benefits to be realized. While the menu facility gives the user the opportunity to customize the system to meet his immediate needs, the GPL II Programming Language facility gives the user the opportunity to extend the system to meet his needs in the future.

The remainder of this chapter presents an informal discussion of the basics of the GPL II Programming Language.

1.1 Interactive Mode

The GPL II Programming Language System has two basic modes of operation. In interactive mode, GPL II statements are entered and executed immediately. In program definition mode, GPL II statements are collected into a program and saved for later execution. The interactive mode of execution is best exemplified by the normal use of the GDS II System: A command or statement is entered to cause the desired effect. If the command or statement is to be repeated, it must be typed in again. One solution for the problem of command repetition is to define a new menu button which effectively types in the desired string of characters with a single keystroke. In many cases, however, it is necessary to repeat a sequence of commands with provision for exceptions or with different parameters, and so on. The GPL II Programming Language makes it possible to execute one or more commands, to obtain feedback from the system, to manipulate data, and to conditionally execute more commands. Of course, once a program has been defined, it can be called and executed as often as desired while in interactive mode.

1.2 Source Files

A GPL II program is stored as a file in the GDS II system. The procedures for program creation, maintenance, and listing follow the general rules for text manipulation in the GDS II System. Since text editing is such a fundamental part of system operation, the same basic techniques are used independently of where the text is stored. This discussion assumes that the user is familiar with editing text associated with a menu button or located in a drawing file, and that no detailed editing techniques need be discussed here. In fact, there is little difference between the procedures needed to associate text with a button and defining a subroutine in a source file and then associating a call to the subroutine with a button.

While the basic editing techniques are the same throughout the system, there are special features which may be useful in a particular application. These include positioning a source file to a particular statement identified by number, and so on. All such features are defined in the editor documentation for easy reference.

1.3 Work Area

The GPL II Programming System maintains a work area for each user which is essentially independent of the remainder of the GDS II System. These areas are maintained in such a way that work in progress at one station is not affected by work in progress at any other station. The work area serves as a repository for both compiled programs and data elements. Before a GPL II program can be executed, it must be translated into a more suitable form. The result of this translation is stored in the work area. Intermediate results created in the process of execution are also stored in the work area. While the format of the data stored in the work area is of little concern to the user, it is important that the work area contents be distinguished from the system data base contents. In general, an operation which affects the current library set will not affect the work area and vice versa.

1.4 GPL II Commands

The GPL II Programming System defines commands that are applicable to the work area as a whole. A command consists of a right parenthesis, the command name, optional parameters, and a carriage return. The syntax is chosen deliberately so that a command can be entered only from the operator console; a system command can not be part of a program. One example of a command which demonstrates why this is so is the CLEAR command. This command,)CLEAR, initializes the work area. Since this command cannot be placed in a program that is executed from the work area, the program cannot wipe itself out.

1.5 Data Elements

The data structures used in the GPL II Programming System form the foundation for the programming language. A thorough knowledge of the data structures used is very important for writing effective and efficient GPL II programs.

The simplest data structure is illustrated by a scalar, which is a single constant. The scalar value can be a number or a character of text. The next level of complexity is the vector, which is a sequence of numbers or characters. All of the individual values in a vector can be referenced collectively or individually, depending on the operation programmed. Given the linear sequence of values in a vector, the next step is to define a matrix, which is a two-dimensional pattern of values. This procedure is repeated to define arrays of higher rank.

The concept of shape is very important in the GPL II Programming Language. Certainly, the number of values in a vector is just as important as the magnitude of any individual value. Similarly, the number of rows and columns in a matrix is just as important as an individual datum. If we presuppose that a frequent use of a matrix is to use the shape of n rows by 2 columns for the vertices of a polygon, the importance of shape becomes even clearer. This is so since polygons are generally characterized by the number of sides, that is, one less than the number of rows in the matrix representing the polygon.

The use of the term *shape* refers to the way in which data values are presented algebraically. Within the constraints of the line length of the output device, a matrix is literally output as a rectangular pattern of values. (It is for output considerations that an n by 2 matrix is used to represent a polygon rather than a 2 by n matrix.) Of course, the term *shape* can also be used in a geometric sense, but in GPL II programs, only the shape associated with a data structure is of importance.

The definition of shape allows a zero value count. That is, a null vector is one case of a data structure that has a null component. It is possible to use other configurations to advantage, for example, an array of 0 rows and 2 columns. Examples of such situations will be considered in later sections.

The description of a matrix as a two-dimensional pattern of values must also be considered in an algebraic sense. In an n by 2 matrix used to represent a polygon, the first column of values is used for the x coordinates and the second column of values is used for the y coordinates. Each row of the matrix therefore represents one vertex of the polygon. It is assumed that the first vertex is connected to the second by a straight line, and so on. The polygon is closed if the first and last vertices have the same coordinate values. Note that a line in three-dimensional space could be defined in the same way by simply using a third column of values for z coordinates. It must be emphasized that while a matrix is presented as a two-dimensional pattern of values, a matrix is definitely not limited to solving a two-dimensional problem. To avoid confusion, as much as possible, a matrix will be defined as an array of rank 2, and the more common method of describing a matrix as a two-dimensional data structure will not be used.

The use of the term *mode* refers to the format in which individual data items are stored in an array. The two basic modes are character and numeric. The numeric mode can be further categorized according to the internal representation used for the number. A fundamental rule of the data structure is that all elements of an array must have the same mode. Further, the operations which can be performed on an array can be restricted to either character or numeric data modes. The various formats used for numeric values are of no consequence in this regard since automatic type conversion from one numeric format to another is supplied. Conversion from character to numeric mode and vice versa must be programmed explicitly, however. Of course, this conversion is easily handled using operators defined in the GPL II Programming Language.

It must be emphasized that shape and mode are independent attributes of a data structure. Although character vectors and numeric matrices are very common, it is perfectly legal to define numeric vectors and character matrices.

While the data structures defined thus far have great utility, there are two important limitations when arrays are considered. The first limitation is that all matrices must be rectangular in shape. All rows and columns must have the same number of data values. In particular, a character matrix is not well suited for a paragraph of text unless the text has been justified so that all lines are the same length. A more convenient data structure is a list in which each list element is defined as a character vector. This allows paragraphs of text to be stored in more compact form and to be manipulated more easily. The second limitation of arrays is that a single mode must be defined for all data values. This restriction does not apply to lists--a mode is defined for each list element. A list allows, for example, a coordinate and a line of text to be referenced as a single data structure.

The list structure in GPL II is deliberately kept simple in the interests of implementation and execution efficiency. In particular, a list element can at most be an array; it cannot be another list. In more formal terms, recursive list processing is not supported.

1.6 Simple Assignment

A data structure can be created interactively by giving the name of the data structure, an assignment specification, and the desired data value. All storage is allocated dynamically depending on the shape and mode of the desired data structure.

For example, typing the statement AA := 1 creates the data structure AA and assigns the scalar value 1. Any previous value assigned to AA will be erased.

1.7 Elided Output

The result of any expression typed in is output to the operator's console by default. Elided output is really a special case of assignment in which the variable name and assignment symbol are elided, that is, omitted. Given the variable AA created above, typing AA would cause the value of AA to be output to the console. However, typing BB := AA creates a variable BB which has a shape and value vector copied from AA.

SECTION 2
SYMBOLS

The term *symbol* is used to refer to the building blocks of the GPL II Programming Language. Possible symbols include constants, names, operators, and keywords. Symbols are used to build expressions as will be discussed in Section 3.

2.1 Character Set

Symbols are constructed from one or more characters. Any character on the GDS II Station Keyboard which can be entered into a source program using the Text Editor is legal at some point in a GPL II program. Of course, at a specific point in a program, only certain symbols are syntactically correct. These rules, however, are more easily defined in terms of symbols than in terms of sequences of arbitrary characters. For example, the letter E can be used as part of a program name, a variable name, a keyword, a real constant, and so on. The symbol ENDSUB always has the same meaning, even when taken out of context.

With one exception, all devices in the system support the same character set. The exception is the character for underscore, which is printed as a left arrow by some devices.

2.2 Records

A record is defined as an arbitrary sequence of characters that is terminated with a carriage return. The concept of a record has general applicability throughout the GDS II system. In fact, all input typed at the operator's console is collected on a record basis. The system task that collects input records provides character deletion, line deletion, digitizer conversion, and other essential capabilities. Since this task serves other programs besides the GPL II Programming System, no generalized processing of the contents of the input record is possible at this level.

Since the task of collecting a record from the console input device involves echoing each character on the console output device, a record has a fixed maximum length. For the purposes of the GPL II Programming System, a record can be logically continued by placing an up arrow immediately before the carriage return. The up arrow and carriage return are "invisible" as far as further processing by the GPL II system is concerned. This definition of a logical record follows the convention established by the RDOS Command Line Interpreter.

This manual will proceed to define symbols in terms of characters, expressions in terms of symbols, statements in terms of expressions, and programs or functions in terms of statements. Of all of these levels, the statement corresponds to a logical record. The GPL II Commands that will also be defined are entered in the form of a logical record.

To digress for a moment, the use of a special character to nullify the effect of a carriage return is preferred over the ALGOL approach of defining a special record terminator. Treating the exception as a special case rather than the rule is deemed more suitable for interactive use.

2.3 Names

The rules for formulating names in the GPL II Programming Language follow the same general rules used throughout the GDS II System. That is, the first character of a name must be either a dollar sign or a letter of the alphabet in either upper or lower case. If a name has more than one character, the second and succeeding characters must be either a dollar sign, a letter, a digit, or an underscore character. The name is terminated by the first character that is not of the above form. (It should be noted that an up arrow and carriage return are understood to be "invisible".) While there is essentially no limit to the number of characters that can be written in a name, 32 characters of significance will be retained. In other words, two names which are identical in the first 32 characters are not recognized as being different names.

Some names have been appropriated for system use. Certain names are used as symbols for operators or as structural keywords; a complete list of all reserved words can be found in Appendix A. These names can be used either in strictly upper case or in strictly lower case, i.e., PROCEDURE or procedure. The user is cautioned against using such names as *Procedure* in order to avoid confusion.

The most frequent use of names is to reference variables. Writing the name AA in an expression references the current value defined for the variable AA. A new value may be assigned to the name AA by using the assignment statement to be defined later. Other uses of names include subroutine calls. Since the list of GDS II system names and primitives, which may be referenced by GPL II programs, are strictly in upper case, the user should consider using lower case names in programs subject to change.

Examples:

Correct	Incorrect
AA	A A
aa	a.
POLY3	3POLY
Long←name	

2.4 Character Constants

Character constants are declared by enclosing the data in double quotation marks. Any printable character except a double quote, a less-than sign, or a greater-than sign represents itself in a character constant. Spaces and tabs can also be used in character constants. Other characters must be encoded when used as part of a character constant. (In the interest of consistency, the GPL II Programming Language employs the same rules defined for Data General's DG/L programming language.)

Within a character constant, the syntax of a special character value consists of a less-than sign, the character code, and a greater-than sign. The character code can be either a name or an octal constant. All currently defined names are either two or three characters chosen to be a mnemonic for the special character. Octal values can be in the range of 0 to 177, inclusive. Leading zeros can be used with octal values if desired. The following table illustrates the special character codes:

MNEMONIC	OCTAL	NAME
<NUL>	<0>	null
<BEL>	<7>	bell
<HT>	<11>	horizontal tab
<LF>	<12>	line feed
<FF>	<14>	form feed
<CR>	<15>	carriage return
<NL>	<15>	new line
<ESC>	<33>	escape
<QT>	<42>	quote
	<74>	less than
	<76>	greater than
<DEL>	<177>	delete(or rubout)

Character Scalars

A character scalar is created by enclosing, effectively, one character in double quotes. A scalar representing a double quote, an up arrow, or any special character can be created as noted above.

Examples:

"a"
"A"
"<CR>"
"<QT>"

Character Vectors

Character vectors are created by enclosing zero or more than one effective character in double quotes.

Examples:

""	(a null vector)
"abc"	
"A<CR>B"	(an A, a carriage return, and a B)

2.5 Numeric Constants

Numeric constants can be defined in three different formats. It is always good practice to choose the format of a constant according to the way in which the value will be used. Because automatic type conversion is supplied, however, this practice is not mandatory. Of course, avoiding the automatic type conversion can increase execution efficiency by a small amount and may improve the clarity of the program as well.

Logical Scalars

The simplest numeric constant is a logical scalar. A logical constant can have only one of two values, one or zero. On input, these values can be selected by using TRUE or FALSE, respectively.

Examples:

Correct	Incorrect
0	
1	
TRUE	
FALSE	Any value not shown on the left

Integer Scalars

The integer data representation can be used for whole numbers within the range of minus 32768 to plus 32767. This range represents the total range of values that can be stored in a 16-bit word using a sign bit. Positive integer constants are specified by simply entering the digits in successive character positions. Negative integer constants are specified by entering a negative sign followed immediately by the magnitude of the value. No spaces, commas, or any other character may be used between the sign and the first digit of the value or between any two digits of the value. In the GPL II Programming Language, it is very important to distinguish between a negative sign used in an integer constant and the symbol for the negation operator. If a space is used between the sign and the first digit of the number, the negation operator is interpreted. (In a programming language that has only scalar constants, or even in a GPL II program which deals only with scalar quantities, the negation operator applied to a scalar constant produces the same effect as a negative constant. The distinction is critical in vector constants, however, so the user is cautioned against overuse of the negation operator.)

Examples:

Correct	Incorrect	
-5	-50000	(Doesn't fit in 16 bits)
3	1.5	(Has a fractional part)
127	1,000	(Comma illegal)

While the constant 1 is treated as a logical constant, it can be used just as if it were an integer constant. Of course, the converse is not true: integer values cannot be used when logical values are required. Furthermore, although 50000 is an integral value, it is stored as a real constant because of the magnitude. Real constants which have integral values can always be used when integer constants are nominally required, provided that the value is in range.

Real Scalars

The real data representation can be used for any numeric value in the approximate range of 10 to the negative 78th power to 10 to the 75th power. The internal format used for real numbers provides a precision of from 51 to 54 bits, or 16.2 decimal digits. Therefore, integral numbers even larger than 10 to the 16th power can be stored exactly. (The actual limit is 16914398509481983.) This high degree of precision also applies to nonintegral values. For example, the value of pi is taken to be 3.14159263589793.

The basic format used for real numbers consists of an optional negative sign, the integral part of the number, a decimal point, and the fractional part. If the integral part of the number is zero, it can be omitted if the decimal point and fractional part are specified. That is, a zero is not required before the decimal point if the number is positive or between the negative sign and the decimal if the number is negative. If the fractional part of the number is zero, it can be omitted if the integral part and the decimal point are specified.

Examples:

```
2.15
-2.5
-0.5
-.5
.5
0.5
0.5000
0.0
128
-30000
10000000000000000000000
10000000000000000000001
```

The input value can indicate more precision than what is available without generating an error. The last two examples above result in the same number in the internal representation.

Because of the large range possible with real numbers, engineering notation is frequently used for inputting real numbers. This allows a scale factor defined as a power of 10 to be applied to the number. The form of the scale factor is the letter E followed by an optional negative sign and an integral power of 10. The scale factor can be used only as a suffix for a real number in the form described above. This is so since E10, for example, is a perfectly valid variable name.

Examples:

```
1.0E3
1000

1.E-3
.001
```

6.28E2
628.

Note that the pairs of numbers in the examples above are equivalent forms of the same value.

Finally, the decimal point can be omitted in a real number specification if the scale factor is specified. Hence, 1E3 is the same as 1.0E3. If both the decimal point and the scale factor are omitted, the format is still legal, but, depending on the magnitude, the result may not be stored as a real number. A number such as 3000 follows the rules for integer numbers described above.

Numeric Vectors

Numeric vectors are created by using multiple values of the form defined for numeric scalars with successive values separated by spaces or tabs. The mode of the vector depends on the mode or modes used for the values in the vector. The result will be in logical mode if, and only if, all values are in logical mode. The result will be in integer mode if, and only if, one or more values are written in integer mode and any remaining values are written in logical mode. The result will be in real mode if any value in the vector is written in real mode. Note that the mode for the entire vector need not be indicated by the format of the first value in the sequence.

Examples:

0 1 0	(A logical vector)
0 1 2	(An integer vector)
0. 1 2	(A real vector)
1 1.5 1000	(A real vector)
0 -1 2	(An integer vector with a negative value)

SECTION 3
EXPRESSIONS

The term *expression* is used to refer to an operand symbol or to a sequence of operator and operand symbols that are syntactically correct. The expression defines what operands and operators are to be used and in what order operations are to be executed. Execution of an expression typically results in a data structure that can be used as an operand in another expression. The rules for creating an expression and determining the execution order are covered in this section.

3.1 Operands

An operand is a data structure. An operand can contain one or many data values, depending on the rank and shape. Depending on the use to which the operand is put, only part of the complete data structure may be of interest, i.e., either the data values or the shape. Examples of operands include constants, variable references, or the result of an expression.

3.2 Structure Operators

The structure of an operand is as important as the actual data values contained within the operand. Various aspects of the data structure are accessible and redefinable with the structure operators.

3.2.1 TYPEOF Operator

The TYPEOF operator returns a character vector indicating the type associated with a name. The TYPEOF operator is a special case in that the name need not be associated with a data structure. The eight possible types are: "UNDEFINED", "NUMERIC", "CHARACTER", "NULL", "LIST", "FUNCTION", "PROGRAM", and "LABEL".

The TYPEOF operator is very useful in interactive mode for program debugging. Consider the following terminal session:

```
Operator Input          Response

)CLEAR                  WORK AREA INITIALIZED
TYPEOF AA               UNDEFINED
AA := 1
TYPEOF AA               NUMERIC
AA := "AB"
TYPEOF AA               CHARACTER
AA := ""
TYPEOF AA               NULL
AA := 1 2;"ABC"
TYPEOF AA               LIST
TYPEOF AA[1]            NUMERIC
TYPEOF AA[2]            CHARACTER
```

A second use for the TYPEOF operator is determining whether an operation is legal. This operator is described first so that it can be used in the remaining operator descriptions.

3.2.2 Ravel Operator

The ravel operator returns the values of an array data structure as a vector.

Format: , array

The ravel of a scalar is defined as a one-element vector. The ravel of a vector is equivalent to the original vector. The ravel of an array of higher rank is defined as a vector containing all the elements of the array taken in row order. That is, the column subscript varies the fastest, then the row, then the plane, and so on for arrays of higher rank. For a matrix, this is equivalent to saying that all the values in the first row are taken, then all the values in the second row, and so on until all rows are exhausted. This is the same order in which the values appear when an array is formatted for output.

Examples:

```
,1 is the vector of one element containing 1
,"a" is a one element vector containing a
,1 2 3 is the vector 1 2 3
,"ABC" is the same as "ABC"
,AA is the vector 1 2 3 4 5 6 if AA is pictured as:
    1 2 3
    4 5 6
```

Error Conditions: The operand must be an array data structure. If the operand is a list, a DOMAIN ERROR results.

3.2.3 SHAPE Operator

The SHAPE operator returns the shape of an array operand.

Format: SHAPE array

The shape is defined as a vector with an element count equal to the rank of the array operand. If the element count is greater than zero, the values in the vector define the element count along each coordinate. If the operand is scalar, the rank is zero, so the shape is a null vector. If the operand is a vector, the rank is one, so the shape is a one-element vector. The value of the vector is the number of elements in the operand. If the operand is a matrix, the rank is two, so the shape of the result is a vector with two elements. The first value is the number of rows in the matrix; the second value is the number of columns. The shape is defined analogously for array operands of higher rank.

Examples:

SHAPE 1	is a null vector
SHAPE "A"	is a null vector
SHAPE ""	is 0
SHAPE "abc"	is 3
SHAPE 1. 2	is 2
SHAPE AA	is 2 3 if AA is a 2 by 3 matrix.

Error Conditions: The operand must be an array data structure. If the operand is a list, a DOMAIN ERROR results.

3.2.4 RANK Operator

The RANK operator returns the rank of an array operand. The rank of a scalar is zero, the rank of a vector is 1, the rank of a matrix is 2, and so on.

Format: RANK array

The rank of an operand is formally defined as the SHAPE of the SHAPE of an operand.

Examples:

RANK 1	is 0
RANK 2	is 1
RANK AA	is 2 if AA is any matrix
RANK "A"	is 0
RANK ""	is 1
RANK "abcd"	is 1

Error Conditions: The operand must be an array data structure. (DOMAIN ERROR).

3.2.5 SIZE Operator

The SIZE operator returns the number of values in an array data structure. The size of a scalar is one, the size of a vector is its element count, the size of a matrix is the product of its rows and columns, and so on.

Format: SIZE array

The SIZE of an array operand is formally defined as the product reduction of its SHAPE vector.

Examples:

SIZE ""	is 0
SIZE 100	is 1
SIZE AA	is 6 if AA is, for example, a 2 by 3 matrix

3.2.6 RESHAPE Operator

The RESHAPE operator creates a new data structure, given the desired shape vector and value vector. The reshape operator is dyadic; the shape vector is written on the left and the value vector is written on the right.

Format: vector RESHAPE array

The shape of the result of a reshape operation is completely specified by left operand. Note that this operand uniquely specifies the rank and size of the result. In particular, a null vector used as the left operand results in a scalar shape. A one-element vector (or scalar) results in a vector shape. A vector with two elements results in a matrix and so on.

The values in the resulting data structure are completely specified by the right operand. If the resulting shape requires fewer values than specified, the extra values are ignored. That is, it is possible to select only the first element of a vector, for example, and turn it into a scalar. If the resulting data structure requires more values than specified, the existing values are reused in a cyclic manner.

The shape of the right operand is ignored by the reshape operator.

The values in the right operand are automatically raveled.

Examples:

```
2 3 RESHAPE 1 2 3 4 5 6 is      1 2 3
                                4 5 6
3 2 RESHAPE 1 2 3 4 5 6 is      1 2
                                3 4
                                5 6
2 3 RESHAPE 1 2 3 4 is          1 2 3
                                4 1 2
2 3 RESHAPE 1 2 3 4 5 6 7 is    1 2 3
                                4 5 6
AA := 2 3 RESHAPE 1 2 is        1 2 1
                                2 1 2
6 RESHAPE AA is                 1 2 1 2 1 2
1 RESHAPE AA is                 ,1 (a vector)
"" RESHAPE AA is                1 (a scalar)
```

3.2.7 IOTA Operator

The IOTA operator creates a new data structure that is a vector of integers. The iota operator requires a single operand which may take various forms. If the SIZE of the operand is one and the value is positive, a vector of that length is created containing the first SIZE positive integers. If the SIZE of the operand is one and the value is zero, a null vector is created. If the SIZE of the operand is one and the value is negative, an error is detected since it is impossible to have a resulting SIZE less than zero.

An index vector can be created with an initial value other than one by using a vector operand with two elements. In this case, the first value in the operand vector defines the initial value in the resulting index vector, and the second value in the operand defines the last value in the resulting index vector. Either or both values may be negative or zero, provided that the second value is algebraically greater than, or equal to, the first. This is required since the SIZE of the resulting vector is determined by the difference in the two values.

An index vector can be created with a specific initial value and a specific delta value by using a vector operand with three elements. In this case, the first value defines the starting value, the second value defines the step size, and the last value in the operand defines the terminating value in the result.

Examples:

```
IOTA 5 is         1 2 3 4 5
IOTA ,5 is        1 2 3 4 5
IOTA 1 5 is       1 2 3 4 5
IOTA 1 1 5 is     1 2 3 4 5
IOTA 1 2 5 is     1 3 5
IOTA 0 is         ""
IOTA 1 is         ,1
IOTA 1 1 1 is     ,1
IOTA -1 3 is      -1 0 1 2 3
```

3.2.8 LENGTH Operator

The LENGTH operator is defined for list data structures. The length of a list is the number of elements in a list.

Format: LENGTH list

Examples: LENGTH 1;2;3 is 3

3.2.9 Equality Operator

The equality operator allows two data structures to be tested for equality. The result of the comparison is a logical scalar, TRUE or FALSE.

Two structures are equal only if several tests are satisfied. First, both structures must have the same TYPE. If the TYPEOF structure A is NUMERIC and the TYPEOF structure B is LIST, the two structures are obviously not equal. Second, if both structures are arrays of the same type, then both must have identical SHAPEs; if both structures are lists, then both must have identical LENGTHs. Finally, both structures must have identical values in order for the two structures to be equal.

Format: structure1 = structure2

3.3 Concatenation Operators

Concatenation allows two data structures to be merged together. Concatenation of array structured operands, for example, allows another row or column to be appended to a matrix. An alternate form of concatenation allows list data structures to be built. A common property in all forms of concatenation is that the SIZE of the result is always the sum of the SIZEs of the operands.

3.3.1 Array Concatenation Operator

The array concatenation operator allows two arrays of conformable shape to be merged. The concatenation of two vectors each having one data value in order to create a vector with two data values is the simplest example of concatenation. Technically, the concatenation of two scalars to form a two-element vector is an example of lamination (see Section 3.8) since the result has a higher rank tha operand.

Format: array , array2
array ,[axis] array2

The first rule for determining whether two operands can be concatenated is that both operands must be of the same mode. Both operands must be numeric or both operands must be in character mode. The internal representation of numeric data is not important; automatic mode conversion is used where appropriate. For example, concatenation of a numeric integer scalar with a real vector results in a real vector. However, concatenation of any number with a character vector results in a DOMAIN ERROR.

The second rule for determining whether two operands can be concatenated is that both operands must either have the same rank or differ in rank by one. The rank of the result is always the same as the greater of the two operand ranks. An example of concatenating two operands of the same rank is the concatenation of two vectors. Because of the above rule, it is also possible to concatenate a scalar with a vector or a vector with a scalar and obtain a vector result.

The third rule for determining whether two operands can be concatenated is that the operands must be conformable in shape. This rule is applicable when one or both of the operands is of matrix rank or higher. Although this rule sounds complicated, it is basically a matter of common sense. Before defining conformability, it is necessary to describe additional information that may be required in concatenating matrix or higher rank operands.

Concatenation requires an axis of application when one or both of the operands are of matrix rank or higher. The difficulty is illustrated in concatenating, for example, two 2 by 2 matrices. The desired effect could be the placement of the second operand along the right side of the first operand. An equally valid result would be the placement of the second operand below the first in the resulting data structure. The shape of the result can be either 4 rows by 2 columns or 2 rows by 4 columns. To resolve this difficulty, an axis of concatenation is required to indicate whether the number of rows or the number of columns is to be increased. In effect, the concatenation operator is subscripted to provide this additional information. If the axis of concatenation is omitted, the last coordinate is assumed. Therefore, it is never necessary to specify an axis of concatenation when the highest rank involved is one.

The axis of concatenation is required if only one of the operands is a matrix. If it is desired to concatenate a 2 by 2 matrix with a 2 element vector, the axis of concatenation specifies whether the vector is considered to be a row vector or a column vector.

Examples:

1, 2 is	1 2
"abc","d" is	"abcd"
"A","bc" is	"Abc"

3.3.2 List Concatenation Operator

The list concatenation operator allows any two operands to be merged into a list data structure. Either or both operands can be a list or an array data structure. No restrictions are placed on mode compatibility or shape conformability.

The list concatenation is not intended to be used in place of the array concatenation operator. There are numerous operators which are defined for arrays but which are not defined for lists. Lists are very useful in situations where an array cannot be used. Perhaps the standard example is treating a coordinate and a line of text as a single data structure.

The symbol for list concatenation is the semicolon. Multiple semicolons can be used in succession to indicate a null list element. A very common use of a list is to specify a subscript. In this case, the list is enclosed in square brackets, and a null first element or a null last element can be specified. (The subscript operation is also an example of a case in which a null list element is more than a simple "placeholder".) The second common use of a list is as parameters to a system routine. In this case, the list can optionally be enclosed in braces. This form is optional because of the frequency with which it is used in an interactive system. This form would be required, however, if a null first or last list element were to be passed. This form can also be used to generate a null list.

Format: list ; list2
list ; list2

Examples:

```
AA := 100 200;"line of text"
BB := AA;"xyz"
CC := ;"line 1<CR>";;"line 3<CR>"
```

3.4 Subscript Operators

The subscript operation allows a portion of a data structure to be selected. The subscript operation can be applied to select one or several elements of an array data structure. An alternate form of subscript allows one element of a list to be selected.

3.4.1 Array Subscript Operator

The array subscript operation allows a subarray to be selected from an array data structure. The subscript is written as a list with the length of the list matching the rank of the data structure. In the simplest case, each element of the list is a scalar and the result of the subscript is a scalar. If V is a vector, then V[2] selects the second element of V. If M is a matrix, then M[2;4] selects the element located in the second row and the fourth column.

The subscript operation is not limited to selection of a single element. If any of the subscript terms is non-scalar, the result will be non-scalar. In fact, the rank and shape of the result are completely determined by the ranks and shapes of the subscript terms.

The subscript operation may be used on the left of an assignment. The subscript in this case selects the elements of the array to be updated.

Format: array [list]

The first rule for a legal subscript operation is that the RANK of the array must equal the LENGTH of the subscript list. (One consequence of this rule is that a scalar can be subscripted by explicitly using a null list. While this does not have great utility in a programming mode, it is very useful in subsequent sections of this manual.)

The SHAPE of the result of the subscript operation is created by concatenating the SHAPEs of all items in the subscript list. Therefore, the following conditions also hold: the RANK of the result is the sum of the RANKs of the subscript terms, and the SIZE of the result is the product of the SIZEs of the subscript terms.

All items in the subscript list must have a TYPE which is NUMERIC or NULL.

An example of non-scalar subscripting will be covered in detail. Consider a matrix, AA, created as follows:

```
AA := 3 4 RESHAPE IOTA 9
```

Then, AA would be output, and may be pictured, as follows:

```
1 2 3 4
5 6 7 8
9 1 2 3
```

Now, consider a subscript reference AA[1 3;1 2 4]. We can select the first and third rows and the first, second, and fourth columns:

```
-1-2-3-4-

 5 6 7 8

-9-1-2-3-
```

By selecting all intersections, we have the following:

```
1 2 4
9 1 3
```

Hence, the result is a 2 by 3 matrix. To re-examine the original subscript operation, note that the first subscript term has a SHAPE of ,2 and the second term has a SHAPE of ,3. By the rules of concatenation, the resulting SHAPE IS ,2 3. Summing the RANKs of the terms gives 2, the LENGTH of the result. Multiplying the SIZEs of the subscript terms gives 6, the SIZE of the result.

For another example, consider the same matrix AA defined above. Now consider a subscript operation such as AA[2;4 3]. We know that the SHAPE of the result is a one element vector--because the SHAPE of the scalar 2 is a null vector and the SHAPE of the vector 4 3 is a two-element vector. So, the RANK of the result is one and the SIZE is two. It should be clear that the result is the vector 8 7. Notice that the fourth column was selected before the third column; therefore, the 8 precedes the 7 in the result.

The subscript operation also features the capability of selecting all values along a coordinate. In the matrix example, all elements of a row or a column can be selected. This capability is programmed by using a null list element in the desired coordinate position. To continue with the matrix AA from above, AA[2;] is equivalent to AA[2;1 2 3 4] since there are four columns in AA. The result of this subscript operation is then 5 6 7 8.

The elided subscript is convenient in addressing n by 2 matrices representing polygons. If PP is a 5 by 2 matrix representing a polygon, PP[1;] is the first row of PP. A reference to PP[1;] returns the x and y values for the first point in the polygon. Similarly, PP[5;] is the last row in PP and the last point in the polygon.

3.4.2 List Subscript Operator

The list subscript operator allows a list element to be selected from a list data structure.

Format: list {scalar}

3.5 Monadic Array Operators

A monadic array operator is defined to be one of a special class of operators in the GPL II Programming Language. While these operators are generally less familiar than their counterparts, the dyadic array operators, these operators are simpler and will be discussed first. The best example of a monadic array operator, for those who have FORTRAN experience, is negation.

All monadic array operators share several common properties. First, the SHAPE of the result of any monadic array operation is the same as the SHAPE of the operand. For example, a monadic array operator applied to a scalar always returns a scalar result. Second, monadic array operators can be applied only to array data structures; a list data structure cannot be manipulated with any of these operators. Finally, monadic array operators are defined only for numeric values.

3.5.1 Identify Operator

The identity operator allows a numeric array data structure to be stored in the most compact internal representation.

Format: + array

The identity operator is defined as a no operation in the mathematical sense since execution of the identity operator results in a data structure having the same shape and data values as the operand. While the identity operator exists primarily for the sake of completeness, it does have the property of reformatting an array into the most efficient internal representation. The identity operator can be useful when large arrays are manipulated.

The result of numerous operations in the GPL II Programming Language are stored in real mode. Examples include multiplication and division of integer quantities. If the result of a division operation is known to have integral values, the result can be forced into integer mode with the identity operator.

Examples:

```
+1 is
                1
+1 -3 is        1 -3
```

Error Conditions: The operand must be in numeric mode.

3.5.2 Negation Operator

The negation operator is the monadic form of subtraction. Given an array operand in numeric form, execution of the negation operator results in a data structure having the same shape as the operand and each non-zero data value negated.

Format: - array

Negation is formally defined in terms of subtraction. Negation is identical to subtracting the operand from zero. If Z := -A, then SHAPE Z equals SHAPE A, and Z[L] = 0 - A[L]

Examples:

- 5 is	-5
- 1 3 is	-1 -3
- 1 -3 is	-1 3
-(-1 -3) is	1 3
- A is	-2 3 -4 if A is 2 -3 4

Error Conditions: The operand must be a numeric array.

CAUTION

The negation operator must be distinguished from the negative sign, particularly where non-scalar values are concerned. The negation of the scalar 5, shown above, is equivalent to the constant minus 5. The negation of a vector containing a plus 5 and a minus 5, however, is not equivalent to a vector containing two minus fives. While misusing the negation operator for a negative sign costs only an insignificant amount of execution time when scalars are involved, the same mistake with vectors or arrays is more serious.

3.5.3 Signum Operator

The signum operator distinguishes among negative, zero, and positive numbers. Given an array operand in numeric form, execution of the signum operator results in a data structure having the same shape as the operand. Each data value in the result is either plus one, zero, or minus one, according to whether the corresponding value in the operand was positive, zero, or negative, respectively.

Format: * array

The signum operator is formally defined as follows: If Z := * A, then SHAPE Z equals SHAPE A, and

Z[L] is 1 if A[L] >
Z[L] is 0 if A[L] = 0
Z[L] is -1 if A[L] > 0

Examples:

*10.5 is	1
*0 is	0
*-0.5 is	-1
*(-1 0 8 3) is	-1 0 1 1

Error Conditions: The operand must be a numeric array.

3.5.4 Reciprocal Operator

The reciprocal operator is the monadic form of division. Given an array operand in numeric form, execution of the reciprocal operator results in a data structure having the same shape. Each data value in the result is the reciprocal of the corresponding data value in the operand.

Format: % array

The reciprocal operation is formally defined in terms of division. If Z := % A, then SHAPE Z equals SHAPE A, and Z[L] = 1 % A[L]

Examples:

%5 is	.2
%.25 is	4.
%1 is	1.

Error Conditions: The operand must be a numeric scalar or array. All values in the operand must have a defined value for the reciprocal.

If a zero operand value is found, a ZERO DIVISOR error occurs (this is a non-fatal error; the corresponding value in the result is set to the largest possible value, and execution continues.) Since the range of real numbers is not symmetric with respect to zero, a FLOATING POINT OVERFLOW error is also possible with the reciprocal operation.

3.5.5 Absolute Value Operator

The absolute value operator returns the absolute value of a number.

Format: ABS array

The absolute value operator is formally defined as follows: If Z := ABS A, then SHAPE Z equals SHAPE A, and

Z[L] = A[L] if A[L] >= 0
Z[L] = -A[L] if A[L] < 0

Examples:

ABS 2.5 is	2.5
ABS -3 is	3
ABS 5 0 -100 is	5 0 100

Error Conditions: The operand must be a numeric array.

3.5.6 Floor Operator

The floor operation returns the largest integral value that is not greater than the specified number. The floor operator is also known as the entier function.

Format: FLOOR array

The FLOOR operator is defined as follows: If Z :=FLOOR A, then, SHAPE Z equals SHAPE A, and Z[L] = A[L] less any fractional part of A.

The floor operation historically has been used as the basis for a rounding operation. For any number, adding 0.5 to the number, and taking the floor, rounds the number to the nearest integral value. For example, the floor of 1.1 + 0.5 is 1 while the floor of 1.6 + 0.5 is 2. Note that the greatest integer is defined in an algebraic sense. For example, the floor of -0.5 is -1 while the floor of 0.5 is 0.

Examples:

FLOOR 1 is	1
FLOOR 1.6 is	1
FLOOR 2.1 is	2
FLOOR -1.6 is	-2
FLOOR -2.1 is	-3

Error Conditions: The operand must be a numeric array.

Note: Fuzz is used in taking the floor. That is, the FLOOR of 2.99999999999999 is taken as 3.

3.5.7 Ceiling Operator

The ceiling operator returns the smallest integral value that is not less than the specified number.

Format: CEILING array

The ceiling operator is formally defined as follows: If Z := CEILING A, then SHAPE Z = SHAPE A, and Z[L] = A[L] if A[L] has no fractional part, A[L] + 1 otherwise.

3.5.8 Exponential Operator

The exponential operator raises the constant *e* to the specified power. The constant *e* is defined as the base of the natural logarithms.

Format: EXP array

Examples:

EXP 1 is	2.7182818284590
EXP 2 is	7.3890560989306

3.5.9 Natural Logarithm Operator

The natural logarithm operator returns the logarithm of the operand to the base *e*.

Format: LN array

Examples:

LN 2 is	0.69314718055995

3.5.10 Pi Operator

The pi operator returns the specified multiple of pi.

Format: PI array

The pi operator is formally defined to be the operand times the constant pi.

The PI operator is especially convenient in interactive mode since PI 1 returns the value of pi to 16+ decimal places.

Examples:

PI 1 is	3.1415926535898
PI %180 is	0.017453292519943

3.5.11 Sine Operator

The sine operator returns the trigonmetric sine of an angle specified in degrees.

Format: SIN array

Examples:

SIN 0. is	0.
SIN 30. is	0.5
SIN 45. is	0.70710678118655
SIN 60. is	0.86602540378444
SIN 90. is	0.99999999999999995

3.5.12 Cosine Operator

The cosine operator returns the cosine of an angle specified in degrees.

Format: COS array

Examples:

COS 0. is	0.99999999999999995
COS 60. is	0.5
COS 90. is	0.

3.5.13 Tangent Operator

The tangent operator returns the tangent of an angle specified in degrees.

Format: TAN array

The tangent function is not defined for any multiple of 90 degrees. Computing the tangent of 90 degrees results in a FLOATING POINT OVERFLOW warning message. The largest possible value is used in this case and execution continues.

Examples:

TAN 0. is	0.
TAN 45. is	1.

3.5.14 Arctangent Operator

The arctangent operator returns the arctangent of an angle with the result specified in degrees.

Format: ARCTAN array

The arctangent returns a result in the range of -90 degrees to +90 degrees.

Examples:

ARCTAN -1.7320508075689 is	-60.
ARCTAN 0. is	0.
ARCTAN 1. is	45.
ARCTAN 3. is	71.565051177078

3.5.15 NOT Operator

The NOT operator returns the ones complement of the operand.

Format: NOT array

The NOT operator is frequently used in Boolean operations.

Examples:

NOT 1 is	-2

3.6 Dyadic Array Operators

A dyadic array operator is defined to be one of a special class of operators in the GPL II Programming Language. The familiar operations of addition, subtraction, multiplication, and division are all examples of dyadic array operators.

A dyadic array operator is not limited to scalar operands. Two matrices, each having 3 rows and 2 columns, can be added, value by value, by programming a single addition operation. Further extensions common to all dyadic array operators will be defined in this section.

Both operands specified in a dyadic array operation must have comformable shapes. If both operands have the same SHAPE vector, the operands are conformable by definition. In particular, two scalar operands are always conformable. The shape of the result of such an operation matches the shape of the operands.

3.6.1 Addition Operator

The addition operator returns the sum of two operands.

Examples:

2 + 2 is	4.
-3 + 8 is	5.
1 + -10 is	-9.
2 4 6 + 1 -3 5 is	3. 1. 11.

3.6.2 Subtraction Operator

The subtraction operator returns the difference of two operands.

Examples:

2 - 2 is	0.
-3 - 8 is	-11.
1 3 5 - 2 1 0 is	-1. 2. 5.

Note that care must be used in visually distinguishing the subtraction operator from a negative sign. The rule is that a minus sign followed immediately by a digit is interpreted as a negative sign. A minus sign followed by some other character followed by a number or a variable is interpreted as a subtraction operator or a negation operator according to whether or not anything preceeds the minus sign.

3.6.3 Multiplication Operator

The multiplication operator returns the product of two operands.

Examples:

2 * 2 is	4.
5 * 10 is	50.

3.6.4 Division Operator

The division operator returns the quotient of two operands.

Examples:

2 % 2 is	1.
3 % 7 is	0.42857142857143

3.6.5 Modulo Operator

The modulo operator returns the remainder of the division of two operands.

Examples:

5 MOD 2 is	1.
5.2 MOD 1.5 is	0.7

3.6.6 Minimum Operator

The minimum operator returns the lesser of two operands.

Examples:

3 MIN 7 is	3
7 MIN 3 is	3

3.6.7 Maximum Operator

The maximum operator returns the greater of two operands.

Examples:

3 MAX 7 is	7
7 MAX 3 is	7

3.6.8 Power Operator

The power operator returns the left operand to the specified power.

Examples:

2 POWER 2 is	4.
2 POWER .5 is	1.414........

3.6.9 Logarithm Operator

The logarithm operator returns the logarithm of the right operand to the base specified by the left operand.

Examples:

10 LOGBASE 2 is	0.30103...
2 LOGBASE 32 is	5.

3.7 Array Extension

In the preceding section, it was said that the operands for array operators are conformable if the operands have identical shapes.

A second way in which operands can be conformable is if one of the operands has a SIZE of one. The operand that has an element count of one is effectively extended to match the shape of the other operand. This operand is typically a scalar, so this is called scalar extension. A vector or matrix having one element is also extendable, however. If both operands have a SIZE of one but different SHAPE vectors, the operand having the lowest RANK is extended. The shape of the result of such an operation matches the shape of the operand having the greater rank.

Array extension along a specified coordinate can also be programmed. In this case, the ranks must differ by one and the shape of the operand having the higher rank, modified to exclude the value for the specified coordinate, must match the shape of the other operand.

To illustrate the various ways in which dyadic array operators can be extended, consider the arrays created by:

```
MM := 3 2 RESHAPE IOTA 6
NN := 3 2 RESHAPE -1 5 8 3 9 6
VV := 10 20
SS := 5
TT := 3
```

The various forms of extension will use the addition operator.

```
SS + TT is 8
MM + NN is           0  7
                    11  7
                    14 12
MM + SS is           6  7
                     8  9
                    10 11
MM + [1] VV is      11 22
                    12 23
                    13 24
```

SECTION 4 STATEMENTS

The term *statement* is used to refer to a logical construction in GPL II Programming Language. An executable statement is a statement which contains one or more operators. (Some statements, e.g., compiler directives or remarks, do not generate object codes.) Examples of executable statements discussed in this chapter include the assignment statement, and the do statement.

Any executable statement can be labeled. A statement label consists of a name and a colon. The rules for the statement label name are the same as those used in defining variable names. The following are examples of labeled statements:

```
ALPHA:  AA := 1
BETA:   GO TO ALPHA
```

The term *block* is used to refer to a sequence of statements that can be used as well-defined points in the program. One form of conditional statement defined below specifies that if the result of a certain expression is true, a block of statements is to be executed. Since blocks are used in ther situations as well, the general rules for blocks will be discussed here.

A block can contain any number of statements, specifically including the case of a single statement. Anywhere a block is called for, a single statement is sufficient. Any type of executable statement can be programmed in a block. This implies, of course, that blocks can be nested. A DO statement block can contain another DO statement, and so on. Any statement in the block can be labeled; however, a label in a block can be referenced only from within the current block or from a block at a subordinate level. That is, a branch out of a block can be programmed but a branch into a block is illegal. In general, a branch into a DO statement block where the execution of the loop is controlled by a FOR clause would result, if permitted by the compiler, in an undefined loop control variable. (If a branch out of a loop is programmed, the value of the loop control variable does remain defined in GPL II.)

While the use of a block is always implicit in the context of a program, e.g., DO block ENDO, explicit block declaration can also be programmed. Delimiting a block with BEGIN and END keywords may seem more natural to some programmers who have ALGOL experience. The programmer can choose to declare a block at any point in the program. Whenever a block is declared, however, the restriction against branching into the middle of the block is always enforced.

4.1 Assignment Statements

An assignment statement is used to associate a data structure with a variable name or to reassign selected data values in an existing data structure. The assignment statement consists of either a variable name or a subscripted variable name, an assignment symbol, and an expression. The symbol for assignment can be either a colon followed immediately by an equal sign, or two less-than signs in succession. The first symbol has the advantage of following the ALGOL precedent, while the second symbol is easier to use in interactive mode. The first form of assignment, using the := symbol, is the more general case in that the space required for the result is allocated independently of any previous space reserved for that variable. The second form of assignment, using the << symbol, specifies that the value to be assigned to the variable must be conformable with the previous value. This form, if consistently used in interactive mode, insures that, for example, a large data array will not be destroyed inadvertently by using the same name for a scratch variable.

In the following discussion, assignment is referred to as an operator. This is done to acknowledge the fact that multiple assignments can be done in a single statement. In effect, assignment is an operation that has the lowest operator precedence.

4.1.1 Variable Assignment

A variable assignment statement consists of a variable name, an assignment symbol, and an expression. The assignment symbol is frequently read as *becomes,* that is, the variable becomes the value and shape of the expression on the right-hand side. Any previous value associated with a variable is erased. (The previous statement applies within the execution of any given program or function. This will be re-examined in the light of local and global variables in the next section.)

Format: var := expression

There are no restrictions on the expression on the right-hand side of an assignment operation. The TYPE of the expression can be NUMERIC, CHARACTER, NULL, or LIST.

Consider the following simple assignment statement:

```
AA := 1
```

In this case, the expression is the scalar constant one. So, the value of AA becomes the scalar constant one. A value may previously have been assigned to AA; if so, the previous value is lost.

A previous value of a variable is erased as part of the function of the assignment operation. For example:

```
AA := AA + 1
```

In this statement, the addition operator is executed first since it has higher precedence than assignment. Therefore, whatever value AA has is increased by one to obtain a value for the expression. The new value is assigned to the variable AA and the old value is discarded. While this explanation has tacitly assumed that AA was a scalar, the statement is equally valid if AA is a vector, a matrix, or an array of any rank so long as the TYPE of AA is NUMERIC. (Note that these restrictions result from using the addition operator; they are not limitations on the assignment operation.)

Examples:

Assignment	Form of the result
AA := 1 + 2	Scalar
BB := 2 3 4 5	Vector
CC := 2 3 RESHAPE 1 2	Matrix
DD := 10 20;"text"	List

4.1.2 Replacement Assignment

A replacement assignment statement consists of a variable name, an assignment symbol, and an expression. Any previous value associated with the variable is replaced, provided that the new value conforms in RANK and SHAPE with the old value. This form of assignment is intended for both programmed and interactive use. The replacement assignment is more efficient in programs since the same space used by the old value of the variable is reused. The replacement assignment is useful in interactive mode since it protects against data loss while debugging, and it is slightly easier to type.

Format: var << expression

Replacement assignment causes all elements in the data structure to be replaced with the elements in a conformable data structure. Replacement assignment does not provide extension in the same way as the dyadic array operators or subscripted assignments do. If AA is a vector of 10 elements, the result of any expression assigned to AA must also be a vector of 10 elements. That is, AA << IOTA 10 is legal, but AA << 1 is not. A further restriction is that the internal format used for numeric data must be compatible in order for the same space to be used. If the named data structure contains elements in integer format, then replacement elements must have integral values.

4.1.3 Subscripted Assignment

A subscripted assignment consists of a variable name, a subscript specification, an assignment symbol, and an expression. Either the replacement assignment symbol or the more general variable assignment symbol can be used.

The subscript operation has been defined and illustrated previously in terms of fetching specific elements of an array or a list for further processing. If the subscript operatin is considered to be a general selection operator, it can be used to select the specific elements of an array or a list to be replaced.

For example, assume that we have a vector AA with the fifth element value of two, that is, AA[5] = 2. To replace just that element, it is natural to write AA[5] << 10. This statement updates the fifth element value without affecting the remaining elements in AA.

As mentioned above, the assignment operator takes on many of the properties of a dyadic array operator. The previous example was that multiple assignments could be used in a statement. For a subscripted assignment, the array extension properties of dyadic array operators also hold. The number of terms selected for update must be conformable with the result of the expression following the same rules. In particular, if a subscripted assignment selects a vector and the result of the expression is a scalar, the scalar will be extended to match the vector. A simple example is setting all elements of a vector to zero. Using the elided subscript form, this is programmed as AA[ ] << 0.

Subscripted assignment will result in the mode of the variable being converted to an equivalent form if necessary. Given a vector of integers, replacing one element of the vector with a real number will result in the vector being converted to real format. There is no side effect of truncation as is typical of most computer languages.

4.2 Branch Statement

As noted above, any GPL II statement can be labeled. A label is required to receive a branch.

A branch statement consists of the keyword GOTO and a statement label. Execution of a branch statement causes control to be diverted to the specified statement.

The use of the branch statement is a matter of personal taste. It is recommended, however, that use of the branch statement should be avoided where possible. (Some recently developed programming languages do not even define a branch statement. We do not go that far, since elimination of the branch statement does not guarantee "structured programming".)

4.3 Conditional Statement

A conditional statement allows a block of statements to be executed based on the result of executing an expression.

4.3.1 IF Statement

The simplest form of a conditional statement consists of an IF-clause, a block, and the ENDIF keyword. The IF-clause consists of the keyword IF followed by an expression. If execution of the expression results in a non-zero value, the associated block is executed. If the result of the expression is non-scalar, only the first numeric value from the raveled result is tested.

Format: IF expression THEN block ENDIF

The IF statement may be extended to include two mutually exclusive cases, as follows:

Format: IF expression THEN block ELSE
block ENDIF

In this form, the first block is executed if, and only if, the first value of the expression result is non-zero. Otherwise, the second block is executed.

The IF statement can be further extended to any number of independent cases, as follows:

Format: IF expression THEN block ELIF
expression THEN block ENDIF

The word ELIF is used as a contraction for the words ELSE IF. In this form, the first block is executed if, and only if, the first value of the expression result is non-zero. This is no different from any other IF-clause. In the event that the first block is not executed, however, the ELIF-clause provides another test to determine whether or not the second block is to be executed. Multiple ELIF-clauses can be programmed if desired. Further, an ELSE block can be used with ELIF-clauses.

The expressions used in a sequence of ELIF-clauses need not be mutually exclusive. The expressions are simple evaluated in the order programmed, and the first successful condition causes the associated block to be executed. All of the clauses are considered to be at the same level of nesting. Consider the following examples:

```
IF AA < 10 THEN BB := 5 ENDIF
IF (AA >= 10) AND (AA < 20) THEN BB := 10 ENDIF
IF AA >= 20 THEN BB:= 15 ENDIF

IF AA < 20 THEN
  IF AA < 10 THEN BB := 5
  ELSE BB := 10
  ENDIF
ELSE BB:= 15
ENDIF

IF AA < 10 THEN BB := 5
ELIF AA < 20 THEN BB := 10
ELSE BB:= 15
ENDIF
```

All three examples are equivalent and set BB to 5, 10, or 15 according to the value of AA. The third form is the most efficient and, more important, is easier to read.

4.3.2 SWITCH Statement

A different approach to conditional execution is provided by the SWITCH statement. This allows any number of mutually exclusive conditions to be processed with one logical statement.

The SWITCH statement consists of the keyword SWITCH followed by an expression, the keyword OF, a case block, and the keyboard ENDSWITCH. The case block consists of as many blocks as desired where each block is prefixed by an constant and a colon.

```
  SWITCH "ABCD" INDEXOF II OF
  CASE 1:     JJ := "A"
  CASE 2:     JJ := "B"
  CASE 3:     JJ := "C"
  CASE 4:     JJ := "D"
              JJ := "NONE OF THE ABOVE"
  ENDSWITCH
```

The constant which identifies the case is not restricted to numeric or scalar values. For example, the character vector constant "ABC" is a legal case identifier. One use of vector numeric constants is shown in the following example. Note that the form "" is a convenient method of specifying a null vector.

```
SWITCH SHAPE MAT OF
CASE "":IX IS NULL"
CASE 2:
CASE 1 2:     "MATRIX REPRESENTS A POINT"
CASE 5 2:     IF MAT[1;] = MAT[5;] THEN
              "MATRIX IS A FOUR SIDED
              POLYGON"
  ENDIF
ENDSWITCH
```

4.4 DO Statements

The DO statement allows a block to be executed iteratively. The various forms include:

```
  FOR variable RANGE vector DO block ENDDO
  WHILE expression DO block ENDDO
  DO block UNTIL expression ENDDO
  DO block ENDDO
```

Note that the FOR-clause performs a special kind of assignment where the control variable is always a scalar which takes on successive values from a vector. FOR variables must be declared as local variables in order to guarantee that the loop control is not disturbed by subroutine calls.

Example:

```
  FOR I RANGE IOTA 10 DO AA[I] := BB[I] ENDDO
```

(Examples in this section are for illustration only and may not be the most efficient method of implementation. The last example, in particular, should be programmed with the equivalent form, AA[IOTA 10] := BB[IOTA 10].)

The RANGE vector in a FOR-clause can be the result of any expression that evaluates to produce a numeric vector. The following are equivalent:

```
  FOR I RANGE 1 2 3 4 5 6 7 8 9 10 DO block ENDDO
  FOR I RANGE 1,2,3,4,5,6,7,8,9,10 DO block ENDDO
  FOR I RANGE IOTA 10 DO block ENDDO
```

The position of a clause with respect to the body of the DO statement indicates when the clause will be executed. If a FOR clause is used, it must be first since the FOR vector is always created once before the loop begins. A WHILE clause, positioned before the body of the loop, is executed before each iteration. An UNTIL clause, positioned after the body of the loop, is executed at the conclusion of each iteration.

Any number of conditional clauses may be used to control a DO loop. If no clauses are used, we have the last form shown above. In this case, it is presumed that a branch statement exists somewhere in the loop, since the loop shown will execute endlessly. In case more than one termination criteria is programmed, the loop executes only until the first criterion is satisfied. Consider the following loop:

```
  DONE := 0
  FOR I RANGE IOTA 10 WHILE NOT DONE DO
    AA[I] := -AA[I]
    IF AA[I] < 0 THEN
      DONE := 1
  ENDDO
```

The above loop will negate successive elements of A until either 10 elements have been negated or a negated element is less than zero.

SECTION 5
SUBROUTINES

The term *subroutine* is used to refer to a self-contained group of statements in the GPL II Programming Language. While interactive input consists of a single statement, a subroutine contains any number of locially related statements which implement a desired capability. The main significiance of subroutines is that a subroutine is the measure of how many statements are compiled at any one time. Conversely, all subroutines are independently compiled.

Subroutines can be categorized according to whether or not the subroutine returns a result and whether or not the subroutine accepts any parameters. The question of whether or not a result is returned has historically been the more important distinction. Subroutines which do return a result are called functions, and subroutines which do not return a result are called procedures. The term *subroutine* is used in a general sense in this manual. The terms *function* and *procedure* are used only where the existence of a result is important.

The question of how many parameters are accepted by a subroutine is resolved, where important, by the use of a qualifying adjective. A subroutine is said to be niladic if it accepts no parameters. A monadic subroutine accepts one parameter. (That parameter may be a list of any number of items, all addressable through subscripting.) A dyadic subroutine accepts two parameters.

The term *program* is generally accepted to mean a top level procedure. In the GPL II Programming Language, any function or procedure can be invoked while in interactive mode and serve as the main program. The same subroutine can also be called by another subroutine. Therefore, the meaning of *program* is not inherent but merely a consequence of usage. When one subroutine calls another, it will be convenient to use the term *program* to refer to the calling subroutine. Program is a relative definition in that the subroutine can, in turn, be referred to as a program by calling another subroutine.

There are several features common to all subroutines. Any subroutine definition must include two distinct kinds of information. Declaration statements are used to define the external environment of a subroutine. Executable statements are used to define the algorithm to be executed by the subroutine.

There are several kinds of declarations that may be made in a subroutine. The header statement gives the subroutine a name and defines how the subroutine is to be called. External subroutine declarations define what subroutines are called by the subroutine or program in question. Other declarations define what variables are referenced by the subroutine and what kind of reference is required.

The declarations serve the purpose of defining names that are referenced in the subroutine. All names in a subroutine must be declared before they can be used. Consequently, all declarations are typically programmed before any executable statements in the subroutine.

The body of any subroutine can include any of the executable statements discussed in Section 3. (In fact, the branch statement is the one case that really makes sense only in terms of a subroutine.)

The definition of a subroutine is terminated with the ENDSUB keyword. Executiôn of a subroutine is terminated by reaching the end of the definition.

5.1 Subroutine Declaration

A subroutine can be defined to serve any of numerous purposes. Subroutines can be used to construct new GDS II commands in terms of existing primitive functions built into the system. In addition to extending the system, the user can also effectively extend the definition of the GPL II Language. With minor qualifications, the user can define monadic or dyadic functions which extend the GPL II set of operators. Unlike many of the built-in functions, however, user-defined subroutines must be either monadic or dyadic. The same function cannot be used in multiple ways like the addition operator can. Also, user-defined functions cannot be used with reduction, inner product, or outer product formulations.

The header statement defines the name given to a subroutine, defines whether the subroutine is a procedure or a function, and defines what parameters are accepted by the subroutine. The header statement must be the first line of the subroutine.

There are six possible header statements:

```
NILADIC PROCEDURE name
MONADIC PROCEDURE name arg
DYADIC PROCEDURE arg1 name arg2
NILADIC FUNCTION result := name
MONADIC FUNCTION result := name arg
DYADIC FUNCTION result := arg1 name arg2
```

The first two words of the header line supply all information needed to call the subroutine. That is, a subroutine declared to be a NILADIC FUNCTION requires no argument and returns a result. (Remember that neither an argument value nor the result value is limited to scalars in the GPL II Programming Language.) The remainder of the header statement includes the same information in a more graphic form in that this represents a pattern of how the subroutine is actually called. The pattern also defines the names to be used for any input parameters and any function result.

The following example is a complete subroutine. All that it does is an addition; it merely serves to give the addition symbol a name.

```
DYADIC FUNCTION Z := A ADD B
LOCAL VARIABLE A;B;Z
Z := A + B
ENDSUB
```

The association of actual parameters with the dummy parameters A and B will be discussed in a subsequent section.

5.2 External Subroutine Declaration

A subroutine can reference any number of external subroutines so long as all such subroutines are declared. Since all subroutines are compiled independently, an external subroutine declaration is necessary to define the required syntax for the subroutine call.

An external subroutine declaration consists of the first two words of the desired subroutine's header and the subroutine name. Therefore, there are six basic possibilities:

```
EXTERNAL NILADIC PROCEDURE NAME
EXTERNAL MONADIC PROCEDURE NAME
EXTERNAL DYADIC PROCEDURE NAME
EXTERNAL NILADIC FUNCTION NAME
EXTERNAL MONADIC FUNCTION NAME
EXTERNAL DYADIC FUNCTION NAME
```

For convenience, multiple external subroutines of the same type can be declared in one statement with the names separated by semicolons. For example:

```
EXTERNAL DYADIC FUNCTION ADD; SUBTRACT
```

is equivalent to:

```
EXTERNAL DYADIC FUNCTION ADD
EXTERNAL DYADIC FUNCTION SUBTRACT
```

Note that no parameter names are mentioned in an external subroutine declaration. The dummy parameter names used by the called subroutine are irrelevant to the caller.

5.3 Local Variable Declaration

A subroutine can declare variables to be local (as opposed to global or external, since a declaration of some type is required). For the sake of discussion, assume that subroutine QQXV has declared variable Q to be a local variable.

```
NILADIC PROCEDURE QQXV
LOCAL Q
Q := 1
END
```

Now, we need another subroutine, say ZZ, which calls QQXV. In this example, we will refer to ZZ as the program and QQXV as the subroutine.

```
.
.
.
EXTERNAL PROCEDURE QQXV
.
.
.
Q := 3
QQXV
IF Q <> 3 THEN "ERROR"
.
.
.
```

The program ZZ assigns a value to the variable named Q. When QQXV is called and it is determined that Q is a local variable, the previous value of Q is stacked and Q is reinitialized to have no value. That is, immediately after QQXV gains control, TYPE Q EQUALS "UNDEFINED". QQXV assigns Q to have the value 1 and exits. Whenever a subroutine exit is performed, all values associated with local variables are discarded and the previous values, if any, are restored. As far as program ZZ is concerned, the value of Q is the same as before and the word "ERROR" will never be output. The capability of declaring local variables allows a subroutine to be programmed that will not affect any variables declared by any programs that use the subroutine. The absence of "side effects" is a feature that greatly simplifies program development and maintenance. QQXV is an unusual example in that its execution has no effect at all.

The format of a local variable declaration is:

```
LOCAL name1; name2;...;namen
```

5.4 Global Variable Declaration

A subroutine can declare variables to be global (as opposed to local or external). A global variable declaration in a subroutine causes a new instance of that variable to be created with any previous value stacked when the subroutine is called. Unlike local variables, however, a global variable declaration allows the variable to be accessed by lower level subroutines which declare the variable to be external.

The format of a global variable declaration is:

```
GLOBAL name; name1;...;namen
```

5.5 External Variable Declaration

A subroutine can declare variables to be external (as opposed to global or local). An external variable declaration in a subroutine allows a global variable declared by the calling program to be referenced by the same name.

The format of an external variable declaration is:

```
EXTERNAL name; name1;...;namen
```

5.6 Parameter Passing

A parameter is a variable named in the header statement. Parameters are virtually always declared as local variables. Parameters cannot be declared as external variables.

No parameters are allowed in a call to a niladic, subroutine. One parameter is required to be present in a call to a monadic subroutine. Two parameters are required in a call to a dyadic subroutine. For monadic or dyadic calls, the parameters must be array data structures. A parameter may be a list. Each item is then accessible within the list by subscripting. The declaration of a subroutine with a list of parameters is:

```
EXTERNAL MONADIC FUNCTION ATTEMPT
CALLING format is:

  QQ := ATTEMPT VV;XX;WW
```

The "ATTEMPT" subroutine is:

```
MONADIC FUNCTION QQ := ATTEMPT STUFF
LOCAL STUFF;QQ;VV;XX;WW
VV := STUFF[1]
XX := STUFF[2]
WW := STUFF[3]

.
.
.
```

All parameters are passed by value.

For the sake of discussion, assume that subroutine ADD shown above has declared variable Z to be local. When the ADD subroutine is called, a variable named

Z may already exist. If so, the value of Z is stacked and Z is reinitialized to have no value. That is, its type becomes "UNDEFINED". The ADD subroutine can assign a value to Z during the course of its execution. In fact, if Z has been identified as the name for the function result, Z must have a value assigned by the ADD subroutine. In this case, the value of Z becomes the value of the ADD function when ADD returns to its caller. Any previous value of Z is restored as part of the subroutine exit mechanism.

5.7 ENDSUB

Execution of a subroutine is terminated when the ENDSUB statement is encountered. If the subroutine is a function, the variable named for the function result must have had a value assigned at some point during the execution of the function. Given that a value has been assigned to the function result variable, the function result can be reassigned if desired.