MATHEMATICAL PROGRAMMING LANGUAGE --USER'S GUIDE

by

Donald R. Woods

STAN-CS-76-561
AUGUST 1976

COMPUTER SC IENCE DEPARTMENT
School of Humanities and Sciences
STANFORD UN IVERS ITY

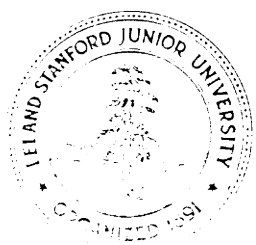


TABLE OF CONTENTS

PREFACE

Mathematical Programming Language (MPL) is a programming language specifically designed for the implementation of mathematical software and, in particular, experimental mathematical programming software. In the past there has been a wide gulf between the applied mathematicians who design mathematical algorithms (but often have little appreciation of the fine points of computing) and the professional programmer, who may have little or no understanding of the mathematics of the problem he is programming. The result is that a vast number of mathematical algorithms have been devised and published, with only a small fraction being actually implemented and experimentally compared on selected representative problems.

MPL is designed to be as close as possible to the terminology used by the mathematician while retaining as far as possible programming sophistications which make for good software systems. The result is a programming language which (hopefully!) allows the writing of clear, concise, easily read programs, especially by persons who are not professional programmers.

Use of This Manual

As this manual is intended for use by people with little or no programming experience, as well as by those who are significantly more experienced, it has been organized into three sections. Section I describes those features of MPL which are necessary, or at least extremely handy, for doing anything useful with the language. These include such things as basic syntax (what a program looks like), simple operations (+, -, etc.), and the more useful commands. Section II describes features which are helpful for doing anything complicated, such as procedures, row and column vectors, and so forth. Finally, Section III describes features which are available for doing anything fancy and wonderful. In using this manual to learn MPL, we recommend that you stop at the end of each Section in order to assimilate and experiment with what you have just learned. Proceed to the next Section once you have the confidence (or need) to do so.

Although this manual is not intended as a general text on programming, there will be occasional comments on programming techniques, which are intended solely for the inexperienced programmer. These discussions are set in slightly smaller type to distinguish them from the rest of the manual. Such topics as how to use keypunching machines or how to run jobs on the computer are not covered at all.

Acknowledgement

We gratefully acknowledge the Stanford Artificial Intelligence Laboratory for providing the programs and equipment which produced this manual.

SECTION I

1: GETTING STARTED

To give you some idea of what MPL is all about, we are starting off with a simple but substantial sample of an MPL program, You are not expected to understand it yet, but if you desire proof that MPL is more than just some professor's bad dream you may punch a copy of the deck shown (substituting your own computer account number in place of X000, your bin number in place of 123, and your keyword in place of FOO) and try running it yourself.

The program reads from data cards (also shown) 3 parameters: m, r, and n. It then reads 2 matrices, A (m by r) and B (r by n). Finally, it verifies the relation $\|AB\|_F \leq \|A\|_F \cdot \|B\|_F$, where $\|M\|_F$ represents a special function of a matrix M, defined by

$$\|M\|_F = (\Sigma_{i,j} M_{i,j}^2)^{1/2}$$

(This function, in case you're interested, is known as the "Frobenius norm". If it sounds esoteric, fear not. You needn't understand its significance to understand the program,) The deck is shown on the next page.

The cards PROGRAM through **END** contain the MPL program; the indentation used through it is for readability only and in no way affects the program. The cards full of numbers near the end of the deck are the data. For the curious-but-lazy, the norms should be roughly (according to hand calculations) $\|AB\|_F = 26.644156$, $\|A\|_F = 2.930017$, and $\|B\|_F = 16.52967$. Briefly, the program works as follows. After the PROGRAM card which marks the start of the program, the next 3 lines define the parameters and data, and simultaneously read the values from the data cards supplied at the end of the deck, Next, we define a special function called Norm with a matrix as its only argument. This function starts a summation with zero and goes through each row and column of the matrix, adding the square of each element. Then it raises the sum to the 0.5 power and "yields" the result. The marks |_ and _| are used to separate from the rest of the program the set of statements that define how the **FUNCTION** is evaluated. Finally, the Norm function is evaluated with three different matrix arguments and the relation described earlier is tested. On the basis of this test, some appropriate messages are written out.

```
//SAMPLE JOB (X000,123),'MPL USER'
/*KEY FOO
// EXEC MPLC
//COMP.SYSIN DD *
PROGRAM
GIVEN (m, r, n) INTEGERS;
GIVEN A REAL MATRIX m BY r, B REAL MATRIX r BY n;

FUNCTION F := Norm (M) WHERE M IS MATRIX, F SCALAR;
|_      DEFINE sum := 0.0;
        FOR i IN {1,...,ROWSIZE(M)},
            FOR j IN {1,...,COLSIZE(M)},
                sum := sum + M(i,j)*M(i,j);
        F := sum ** 0.5
_|;

IF Norm (A*B) > Norm (A) * Norm (B) THEN
        WRITE (<<It didn't work!>>)
ELSE
        WRITE (<<It worked (what a surprise!).>>);
WRITE (<<For the record, Norm (A*B) is:>>, Norm (A*B),
        <<Norm (A) is:>>, Norm (A),
        <<Norm (B) is:>>, Norm (B))
END.
/*
//GO.SYSIN DD *
3  4  2

0.5   1.0   0.2   -2.
.25   -.7   1.2   0.0
-.1   0.3   .45   -1.

1.25   -2.3
0.55   10.6
-8.6   -7.9
.35   -4.15
/*
```

If you find all of this difficult to believe, go back and examine the program until you are convinced that a computer just *might* be able to figure it out. Trust us: it's been done. The problem at hand is getting you to the point where you could have written the program yourself (assuming you had some far-fetched reason for wanting to do so).

Having now given you a taste of the main course, as it were, we shall back off and start over with the appetizer.

1.1: Characters and Symbols

The character set for MPL consists of a subset of EBCDIC, the IBM 360 character set. A "character" is a single stroke on the terminal or keypunch. The following groups of characters are recognized by MPL:

Alphabetic: abcdefghi jklmnopqrstuvwxyz
ABCDEFGHIJKLMNOPQRSTUVWXYZ

Special Alphabetic Characters: _ $ '

Numeric: 0123456789

Special Characters: # & * ( ) - + = : ; " , . / | < > ¬ { }

Blank: (1 stroke of the space bar on a terminal or keypunch)

The above are the only characters which have any meaning in MPL. In addition, however, there are certain multiple-character symbols consisting of 2 or more special characters occurring together *with no intervening spaces*, which have special meanings distinct from those of the individual characters. These symbols are:

**	:=	¬=	,...'
' '	' '	<=	<(
_|	>>	>=	)>

Note: The symbols <(and)> are treated exactly like the characters { and }, respectively, and are available only because { and } are not easily produced on a keypunch. Throughout this manual, however, { and } will be used exclusively, as they are much more readable and closer to the mathematical notations with which they correspond.

Whenever a string of adjacent characters first appears that forms one of the above symbols, the group is interpreted by MPL as being the multiple-character symbol. Thus *** is always considered to be a ** symbol followed by a *, as opposed to vice versa or 3 single * characters. (All three interpretations happen to be invalid syntax in an MPL program, but that does not concern us here.) Note in particular that in testing a relation such as n < (a*b), the characters < and (*must* be separated by a blank to avoid being treated as a single <(symbol. Similar cautions apply to) followed by >.

1.2: Identifiers and Keywords

"Identifiers" in MPL are symbolic names that are used to represent variables, procedures, and labels. They consist of strings of from 1 to 72 non-blank characters (preceded and followed by a delimiter (as defined in section 1.5) to mark the beginning and end of the string) satisfying the following rules:

1. The first character must be alphabetic.

2. The second and following characters may be numeric, alphabetic, the dollar sign ($), or the underscore (_).

3. Any number of single quotes (') may appear, but only at the end of the string.

Examples,

A1_2	legal
1AB	illegal, first character not alphabetic
A'b	illegal, quote not at end
fix''	legal, two single quotes at end okay
Mx"	illegal, double-quote not allowed
i100$	legal
_ABC	illegal, first character not alphabetic
I	legal

Typically, if the same identifier appears in more than one place in a program, it refers to the same object each time. This allows you to compute a value and save it using some identifier name, and later refer to that earlier value by using that name. Normal practice is to use an identifier name which has some mnemonic significance pertaining to its intended use, such as "Norm" in the sample on page 2 representing the Frobenius norm function and "sum" for the cumulative sum.

Certain identifiers have a special pre-defined meaning in HPL and may not be used as normal identifiers in the sense described above. These identifiers, called keywords, are exclusively upper-case alphabetic characters, and are printed in this manual using bold-face type to help you distinguish them from other identifiers being used for variables.

Keywords are used to represent commands and other program specifications. Keywords in the sample program include PROGRAM, GIVEN, INTEGERS, REAL, MATRIX, BY, FUNCTION, and many others.

A complete list of keywords is supplied in Appendix D.

1.3: Constants

There are several types of constant quantities which may be represented in an HPL program. Some of them will be covered later, when you are more proficient in the language. For now, you need only concern yourself with 2 types: integer and real.

1.3.1:. Integer constants

An integer constant is represented by a string of decimal digits optionally preceded by a + or - sign (a leading + is ignored). The magnitude of the number must not be greater than $2^{31}-1$, or 2147483647. Note that groups of digits are *not* separated by commas.

1.3.2: Real constants

Due to the way in which the computer handles real values, a real constant x is subject to a magnitude restriction such that either $10^{75} > |x| > 10^{-78}$, or else $x = 0$. Real values are accurate to 14 hexadecimal digits, or about 15-16 decimal digits. In the course of computing new real values from two or more old values, imprecisions may of course accumulate. (If you don't see how this could be so, try adding one-third to itself 3 times, maintaining accuracy to, say, 5 decimal digits. You'll get 0.33333+0.33333+.033333=0.99999 instead of 1.00000.) Arranging computations so as to minimize these and related imprecisions is a fairly sophisticated area of computer science, but you should at least not be overly surprised if your program computes, say, $e^{(\ln 2)}$ and produces 1.9999999999997.

Real constants may be written using any of the following forms:

±ddd.ddd ±.ddd
±ddd.dddE±nn ±.dddE±nn
±dddE±nn

where 'ddd' represents any number (at least 1) of decimal digits, '±' may be either '+' or '-' or omitted entirely (it may *not* be a blank), and 'nn' represents 1 or 2 decimal digits. If an 'E' appears, it indicates that the value of the constant preceding the 'E' is to be considered multiplied by 10 raised to the power of the number following. (The 'E' stands for 'Exponent'. For various mystical reasons, a 'D' may be used interchangeably with the 'E'.)

Examples:

12.5	Real constant
1.1.30E10	Illegal real (2 decimal points)
-12E5	Real constant with value -1200000
10D99	Illegal real (value is $>10^{75}$)
109	Integer constant
-0	Integer constant with value 0
12E 3	Illegal real (imbedded blank not allowed)
109.	Real constant
.1D-5	Real constant with value .000001
1,000,000	Illegal (don't put commas in numbers)

1.4: Comments

In an HPL program, any string of characters enclosed in double-quotes, " ... " is ignored along with the surrounding quotes. Such strings, called (comments, may contain any characters except double-quotes and may not be continued to another card (if a comment will not fit on one card, put it on 2 cards as 2 separate quoted strings). When a comment is the last item on a line, the trailing quote may be omitted.

Comments are generally used to include descriptions of how the program is supposed to work. It is considered good practice to include liberal amounts of such documentation in your programs, particularly any programs which you expect somebody else will have to understand, and also any

programs to which you yourself anticipate coming back after a few weeks of disuse. (You'd be amazed at how difficult it can be to remember how your own programs work!) Of course, with simple programs which you intend to use for a few days and then discard, the time involved in commenting makes it less worthwhile, but you should still comment such programs in order to get into the habit. Care should also be exercised that comments provide useful information. To take a section of program which reads M+M' and comment "add M and M'" is not overly illuminating. Much more constructive would be something like "add perturbation matrix to original data".

1.5: Delimiters

Each identifier, keyword, integer, or real constant must be punched wholly on a single card. (For example, the constant 79 may not be crritten with the 7 on one card and the 9 on the next,) Also, identifiers, keywords, and constants must be separated by delimiters, i.e., something to indicate where they begin and end. Delimiters include blanks, special characters (e.g. parentheses), and the end of the card. Multiple blanks may be used to add to readability: they are ignored (except insofar as they act as a delimiter). Blanks may also be added for readability wherever other delimiters occur, For instance, (X,Y) is equivalent to (X , Y).

So much for what programe look like. Let us move on to what they can include.

2: DATA TYPES AND STRUCTURES

Probably the most central concept in most modern programming languages is the notion of variables. The idea is very similar to that of algebra--the use of symbolic names (identifiers) to represent potentially unknown, and possibly varying, quantities. Thus we can assign some value to a variable, x, and proceed to compute a result based on this value of x. We can then change the value of x and go back so as to use the same program to compute a new result based on this new value.

A variable in MPL is any identifier which can be assigned a value. Variables in MPL have two primary characteristics: type (the values they may assume, e.g., real) and structure (the shape in which they are arranged, e.g., vector). These characteristics are not to be confused with the actual values assigned to a given variable.

2.1: Types

As was the case with constants, although MPL allows several different variable types, only a few need concern you at this point.

2.1.1: Real

A double-precision floating-point data element used to hold most numeric quantities. (If this jargon loses you, don't worry about it. It effectively means that a real variable can take on the same sort of values as can be represented by a real constant. 'Data element' refers to an element of storage used to hold data in the memory of the computer.)

2.1.2: Integer

A single-precision fixed-point data element used to hold signed integers, particularly variables used as counters.

2.1.3: Labels

Instructions in a program are executed in sequential order. Sometimes, however, you may want to 'branch' to another part of the program. The place which you wish to go to must be given a name, called a label, so that the MPL compiler can locate it. For example, if you wish the program to return to some starting point, you might label that point START. The form of a label will be discussed later.

A label may only be used as the argument of a GO TO statement, e.g., GO TO START (see section 4.6). Its 'value' is the location in the program code where it occurs. As will be noted later, labels are usually unnecessary, and should be avoided whenever possible since their use makes it more difficult for people to understand the sequential flow of the program, and can also interfere with efficient evaluation of programs by the computer.

2.2: Structures

The structure of a variable is for the most part simply its dimensionality, meaning the number of dimensions it has. There are some more obscure structures which will be introduced later: for now we shall continue to confine ourselves to the basics, as described in the sections below. A variable may be assigned any structure regardless of its type, and vice versa.

2.2.1: Scalar

The "scalar dimensional ity" describes a single datum of the type it is defined as being. Thus a scalar integer is any single integer value, and similarly for scalar reals.

2.2.2: Vector

A vector is a one-dimensional array of scalar data elements, It may be thought of as a finite ordered set of n such elements where $n \geq 0$ is some integer. The values assigned to a vector may be used all at once, as in taking a vector inner product or summing two vectors, or individual elements may be selected by specifying their ordinal position in the vector. This latter operation is known as subscripting and will be described in section 3.2,

2.2.3: Matrix

A matrix is a two-dimensional array of scalar data elements. Like vectors, matrices may be used "en masse" or may be subscripted (using 2 ordinal positions) to yield individual data elements.

2.2.4: Array

An "array" in an MPL program refers exclusively to a three-dimensional array of scalar data elements. Certain operations may be performed on an entire array, or it may be subscripted (using 3 ordinals) to yield single elements.

2.3: Defining Variables

Before any variable is first used (except for FOR index variables and formal procedure parameters, both of which we'll get to later), it must be "declared", i.e., you must tell MPL what kind of beast it is. This is done by describing it in a DEFINE or GIVEN statement. The latter is simply a means of combining a DEFINE with a READ, so we will hold off on it until we come to the READ statement in section 5.3.

The DEFINE statement has the form

DEFINE <identifier list> <attributes>

where <identifier list> is either a single identifier or a list of identifiers (all to be given the same attributes) separated by commas and enclosed in parentheses, and <attributes> is a description of the variable(s) named.

Examples:

```
DEFINE FROG REAL MATRIX p BY q
DEFINE S SCALAR
DEFINE T SCALAR
DEFINE (S,T) SCALARS
```

The last example above is equivalent to the second and third examples combined. Multiple definitions with different attributes may be combined into a single statement by separating them with commas, thusiy:

DEFINE (A, A') REAL MATRICES 5 BY 10, k SCALAR

Another form is the defining assignment statement. It is described in greater detail in the section on assignment statements. Briefly, its action is to define a variable whose value is that of an expression and whose attributes are those of that expression, It has the form

DEFINE <identifier> := <expression>

The symbol '=' may be substituted for ':='. Thus, for example, assuming we

had previously defined a matrix cai led 'A' and assigned it some values, then the statement

```
DEFINE B-A
```

would create a second matrix with the same dimensions and values as the first. The two types may be combined, as in

```
DEFINE p=5,q=8,
       M MATRIX p BY q,
       M' MATRIX q BY p
```

The attributes specified for each definition consist of a single " type" attribute and a single "dimensional i ty" (structure) at tr i bute, separated by blanks. The order of the two attributes is unimportant, and either or both may be omitted, in which case a "default" attribute is assumed. (After ail, every variable has to have *some* type and structure, and i f you don' t spec i fy anything then MPL has to assume *something.*) Type attributes include:

```
INTEGER
REAL
```

I f omitted, the default is REAL. Dimensionality attributes include:

```
SCALAR
VECTOR <size>
MATRIX <size> BY <size>
ARRAY <size> BY <size> BY <size>
```

where each <size> may be any constant, variable, or expression whose value is a non-negative scalar integer. A vector of size zero is called a "nui l vector"; it has no elements. flatrices and arrays may not have sizes of zero. If the dimensionality attribute is omitted, the default is SCALAR.

For readabi l i ty, the foi lowing plural forms may be used interchangeably with their obvious counterparts:

```
INTEGERS
REALS
SCALARS
VECTORS
MATRICES
ARRAYS
```

Examp l es:

```
DEFINE (i,j,k) INTEGERS;
     "defines 3 scalar integer variables"
DEFINE a, b, c INTEGERS!
     "because there are no parentheses around 'a, b, c'
     "(probably by mistake), this defines a and b as
     "real scalars (no attributes being specified, this
     "is the default) and c as an integer scalar"
DEFINE GLORK_SIZE=10;
```

```
        "since '10' is an integer scalar constant,
        "GLORK-SIZE is defined as a scalar integer
        "variable, and is assigned the value 10"
DEFINE GLORK MATRIX GLORK-SIZE BY GLORK_SIZE+5;
        "assuming GLORK-SIZE is as defined above,
        "GLORK is defined to be a real matrix with
        "10 rows and 15 coi umns"
DEFINE (V,V') VECTORS 100, Array INTEGER
            ARRAY 3 BY -(-3) BY (2+7)/3
        "defines 2 real vectors with 100 elements in each,
        "and a 3x3x3 integer array, the hard way"
```

Note that, in the last example, a l though ARRAY i s a keyword and thus may not be used as a variable name, the identifier "Array" (with lower-case) is perfect ly val id. (It's not particularly recommended, however, since keypunches and some printers are geared toward producing only upper-case letters.) In the third example, note that if the value of GLORK-SIZE is l a ter changed, i t does *not* change the dimensions of GLORK unless GLORK i s explicitly re-defined by executing this (or possibly some other) DEFINE statement. This brings to mind another impor tant point: the same ident i f ier may appear in more than one DEFINE statement. If so, the attributes must be the same in each appearance, though the sires may change. For instance, if--you define a real matrix

```
DEFINE M REAL MATRIX 5 BY 10
```

you may re-define it later as

```
DEFINE M REAL MATRIX 100 BY 7
```

but it must always remain a reai matrix, Thus you could not do

```
DEFINE M VECTOR 50
```

nor

```
DEFINE M INTEGER MATRIX 5 BY 10
```

In general, when MPL defines a variable, any previous values or dimensions which that variable may have had are discarded and lost. The crucial exception to this will come up under "block structure" in Section II of this manual, where variables within a block can have total iy di f ferent def init ions from those outside,

3: EXPRESSIONS

Expressions in MPL consist of one or more values, typical iy variables and/or constants, operated on singly or in pairs by MPL operators. The resui t of any operator may in turn be used as a value for another operation.

3.1: Operators and Precedence

MPL allows many operations natural to the various data structures to be specified in clear yet concise forms. You should take particular note of the fact that certain operators may mean different things at different times, depending upon the dimensionaiity of the things being operated upon. For instance, X*Y would be a matrix product if X and Y were matrices, a vector inner product if they were vectors, and other things in other cases. The following paragraphs discuss each operator and describe generally where it may be legally used.

In these discussions, the term "unary operator" means an operator which acts on the single value which follows it, and "binary operator" means one which acts on two values, one on each side of it. All of this corresponds to standard algebraic notation and terminology.

3.1.1: Pius and minus

Denoted by '+' and '-', these may be used as both unary and binary operators. Unary plus is a null operation; it has no effect. Unary minus negates its argument, acting component-wise on non-scalar arguments. Thus -{5,7,-3,2} yields {-5,-7,3,-2}.

Binary addition and subtraction are allowed between two scalars, between two non-scalars of equal dimensional i ty and size (the operations are performed component-wise), and between a scalar and a non-scalar (the scal ar is added to or subtracted from each element of the non-scalar in a manner analogous to the mathematical interpretation of scalar multiplication).

Examp l es:

7.9 + 9.7	yields	17.6
{1,2,3}+{3,-2,1}	yields	{4,0,4}
{1,4,9} - 3.5	yields	{-2.5,0.5,5.5}
-3.5 + {1,4,9}	yields	{-2.5,0.5,5.5}
-{3.5} + {1,4,9}	yields	an error (differing sizes)
-{3.5}	yields	{-3.5}
{1,2}-{1,2,3}	yields	an error (differing sizes)

Note in the first error-example that MPL interprets {3.5} as a vector of size 1 and does *not* treat it as a scalar in this context.

3.1.2: Multiplication

Denoted by '*', multiplication is allowed as a binary operation between two scalars, two vectors, two matrices, or any non-scalar with a scalar. For vectors, the operation is the inner product, and the two vectors must be the same size. For matrices, standard matrix multiplication is performed, with the usual requirement that the column size of the first operand conforms to the row size of the second. † *If you are*

*Throughout this manual, the "column size" of a matrix is taken to mean the number of columns, as opposed to the size of each column, A simi iar interpretation is applied to "row size".

used to using some other programming language with its own kit of multi-dimensional operations, take particular note of the preceding definition of multiplication. Note that, unlike PL/1 and APL (and others), NPL has no operator which multiplies two vectors component-wise to yield a vector result in a manner analogous to vector addition, i.e., there is no operator ⊛ such that {1,2,3}⊛{4,5,6} yields {4,10,18}.

Examples:

.7*9	yields	6.3
{1,2,4}*8	yields	{8,16,32}
{1,2,4}*{8,8,8}	yields	56

Note that, unlike usual mathematical notation which uses only single characters as identifiers, computer applications often require multiple-character names. Accordingly, two adjacent quantities are not multiplied. I.e., 'AB' is a single identifier, not 'A' times 'B'. Nor is '3(A+B)' allowed to designate '3*(A+B)'.

3.1.3: Division

Denoted by '/', division is permitted only when the divisor (the right-hand argument) is a scalar. Each component of the dividend is divided by the scalar divisor to yield the corresponding component in the result. Division by zero will be flagged as an error when the program is executed.

The quotient has type 'real' if either the divisor or the dividend is real. *Note* especially that when an integer is divided by another integer the result is rounded toward zero (as in FORTRAN) to yield an integer result. Thus 7/3 = (-7)/(-3) = 2, and (-7)/3 = 7/(-3) = -2, whereas 7.0/3 = 7/3.0 = 2.33333...

3.1.4: Relational comparisons

Two values may be compared using the binary operators =, >, <, ¬=, <=, and >=. (The last 3 represent ≠, ≤, and ≥, which do not exist in the character set. Note also that '=>' and '=<' may *not* be used in place of '>=' and '<='.) The outcome from such a comparison may be used to determine the result of an IF statement (described in section 4.3). The two operands of a relational operator must be of the same dimensionality and size (if non-scalar). comparisons between non-scalars are defined in the usual mathematical sense, i.e., the relation must hold between each pair of corresponding elements for it to hold overall. Thus {1,2,3} <= {1,3,5} is 'true', but {1,2,3} < {1,3,5} is 'false'. None of the six relations holds between {7,9,10} and {10,9,7}.

3.1.5: Precedence

When more than one operator is used in an expression, it usually makes a difference in what order the operations are performed. For example, '4+5*2' could mean 4+(5*2), which = 14, or (4+5)*2, which = 18. In MPL, as in most programming languages, the operators have been assigned a precedence specifying what order should be assumed in the absence of parentheses.

In expressions without parentheses operators are evaluated according to the precedence shown below. To force a given operator and its operands

to be evaluated before others it may be necessary to enclose certain terms of the expression in parentheses. Such parenthetical sub-expressions are always evaluated before the surrounding operations are performed.

In the following table, all operators shown at any given level are performed before any from lower levels. Among binary operators of equal precedence, operations are evaluated from left to right: unary operators are evaluated from right to left.

```
First:   subscripting (see next section)
         + - (unary)
         * /
         + - (binary)
Last:    relational5
```

Examples:

```
-2+3*4          yields   (-2)+(3*4) = 10
-(2+3*4)        yields   -(2+(3*4)) = -14
-(2+3)*4        yields   (-(2+3))*4 = -20
1-2-3-4         yields   ((1-2)-3)-4 = -8
1-(2-3-4)       yields   1-((2-3)-4) = 6
1-(2-(3-4))     yields   1-(2-(3-4)) = -2
3+2.4/(.6*.8) < (3+2.4)/.6*.8   yields
                3+(2.4/(.6*.8)) < ((3+2.4)/.6)*.8
                i.e., 8 < 7.2 which is 'false'
```

Parentheses may, and should, also be used to clarify non-obvious expressions. For example, since integer division discards the remainder (see section 3.1.3), and since '/' and '*' have equal precedence, the expression 'm-m/n*n' yields m mod n (assuming m and n are scalar integers > 0). Mathematically, however, many people tend to read this as 'm-m/(n²)', so it is wise to include parentheses, writing it as 'm-(m/n)*n', even though it makes no difference in the actual meaning of the expression.

Precedence remains the same regardless of the structures involved; i.e., matrix multiplication and vector inner product have the same precedence as does scalar multiplication. Scalars were used throughout the above examples only because they make for simpler illustrations.

3.2: Subscripting

The term 'subscripting' refers to the selection of a single datum from a vector, matrix, or array. In MPL subscripting may be thought of as a unary operator that may be applied to any non-scalar variable or expression. Subscripting differs from most operators in that it is specified *after* its operand rather than before as with other operators. The subscripting operator consists of a list of from one to three 'subscripts' separated by commas and enclosed in parentheses. For example, in '(1,X,Y+Z)', the subscripts are '1', 'X', and 'Y+Z'. Each subscript may be

any expression (it may even involve additional subscripting operations) which evaluates to a scalar between 1 and the size of the corresponding dimension (as specified for variables when they are defined). If a subscript is real, it is rounded down to yield the actual subscript, The number of subscripts must equal the dimensionality of the operand being subscripted. Thus the example just given could only be used to subscript an array, or to pick out an element of an expression whose value is an array.

Examples:

```
DEFINE n := 10;
DEFINE V INTEGER VECTOR n,    "V thus has 10 elements"
   M MATRIX 2 BY 3,
   M' MATRIX 3 BY 2,
   i=1, j=2;

"Assume there is more code inserted here which assigns
"values to V, M, and M' as follows:
"          V_p=p^2 for 1≤p≤10, M_pq=p+q, M'_pq=p/q
"Then: "
```

V(7)	yields	7th element of V or 49
V(n/j+i)	yields	6th element of V or 36
V(V(i)+V(j))	yields	5th element (V(1)+V(2)=5) of V, or 25
V(0)	yields	an error (0 isn't between 1 and 10)
M(2)	yields	an error (matrices need 2 subscripts)
i(j)	yields	an error (scalars can't be subscripted)
M(i,j)	yields	1st row, 2nd column of M, or 3
M(V(i),V(j))	yields	an error (V(j)=4, but the second subscript must be between 1 and 3)
M(V(i),V(j)/M(1,1))	yields	M(1,4/2), or 3
V(M'(3,2))	yields	V(1.5), i.e. V(1), or 1
(M*M')(i,j)	yields	1st row, 2nd column of (M*M'), or 10
M*M'(i,j)	yields	the 2 by 3 matrix which is M multiplied by the scalar M'(1,2)=0.5 (remember subscripting has higher precedence than multiplication)
(V*V)(i)	yields	an error (V*V is a scalar and cannot be subscripted)

3.3: Vector Generators

MPL has various constructs which allow you to write vectors in different ways depending upon the structure of the values in them. These constructs are known in general as "vector generators", The two most useful forms are described here. In these discussions, the word "set" is not used in the strict mathematical sense. Rather, the MPL vector generators give results which are more accurately described as ordered sets.

3.3.1: N-tuples

An "N-tuple" is written as a list of scalar data elements, all of the same type, separated by commas and enclosed in braces, {}. (Recall that '<(' and ')>' are equivalent to '{' and '}'.) This represents a vector whose size is the number of components specified and whose elements are those in the list, in sequence. Thus {1,0,6,6} represents a 4-element integer vector with first element 1, second element 0, and remaining elements both 6. The individual components may be any scalar expressions, e.g. {i, j*k, V(i/j), {1,2}*{3,4}}, but keep in mind that the types must all be the same. Thus {1, 3.14159, 10} is invalid: it would have to be written as {1., 3.14159, 10.}.

You'll recall that when an expression is used as the <size> of a vector in a DEFINE statement, later changes in the values of variables in the expression does not affect the size of that vector, Similarly, if an expression is used in an N-tuple, changing the values of variables used in the expression does not affect earlier uses of the N-tuple. Thus, if you assign V the value {i,i*i,i*i*i} when i has the value 2, then V will be {2,4,8}. If i is then given the value 3, V remains as {2,4,8}, not {3,9,27}. *In general, nothing which occurs while a program is executing has any retroactive effects.*

3.3.2: Index sets (arithmetic progressions)

A vector whose elements, in sequence, form an integer arithmetic progression may be written as

```
{<first>,<second>,...,<last>}
```

where <first>, <second>, and <last> each represents any scalar integer expression. Note that ',...,' is a single symbol in MPL and thus may not have any imbedded blanks, This format, called an "index set" because of its use with FOR statement indices, represents a vector whose elements are the arithmetic progression defined by <first> and <second>. That is, the first element Is the value of <first>, the second element is <first>+α, the third is <first>+2α, and so on, where a = <second> - <first>. Also permitted is the alternate form

```
{<first>,...,<last>}
```

in which case the increment a is taken to be 1.

The vector consists of all elements of the progression, in sequence, such that no element is greater than <last> if α>0, or no element is less than <last> if α<0. If <first> itself does not satisfy the condition just stated then the vector is null; it has size zero. If <first>=<second> an error results, since this would generate an infinite number of components,

Examples:

{1,3,...,9}	yields {1,3,5,7,9}
{1,...,9}	yields {1,2,3,4,5,6,7,8,9}
{5,...,1}	yields {5,6,...,1}, a null vector
{5,4,...,1}	yields {5,4,3,2,1}

```
{-1, -2,. .., -5}     yields {-1,-2,-3,-4,-5}
{-1,...,-5}           yields {-1,0,1,...,-5}, a null vector
{3,6,...,20}          yields {3,6,9,12,15,18}
{1,1.05,...,2}        yields an error (non-integer)
{{1},...,{5}}         yields an error (non-scalar)
{1,...,5,8}           yields an error (can't combine index sets
                             with -extra items, so this is
                             not {1,2,3,4,5,8})
ii, i*i,...,120/i} yields  {3,9,15,21,27,33,39} i f i=3
                           {4,16,28} i f i=4
                           {5} if i=5
                           a null vector if i=20
                           an error if i=0 or 1 (<first>=<second>)
```

4: PROGRAM STRUCTURE

At long last we have reached the point where you shall learn how to put al l the pieces together to form an MPL program. The sections which follow d i scuss the forms used in MPL programs. If you are not particularly experienced at programming (a n d if you are then you shouldn't be reading these fine-print sections anyway) you may also find useful the occasional discussions concerning how one can use the various forms. So, without further ado, we proceed on to

4.1: Simple and Compound Statements

In MPL the smallest unit of program control is knoun as a "simple statement". The preceding, Unfortunately, is not such a simple statement. The reason is that some so-called "simple statements" can contain other, smal ler simple statements. For instance, in the statement

IF i-j THEN do-this ELSE do-that

the total meaning is to test uhether i=j is true, and if so to perform the sub-statement 'do-this', else to perform 'do-that'. Here, 'do_this' and 'do-that' are both simple statements, and the entire line containing them is also a simple statement. Fortunately, since we are not dealing with the quantum mechanics of programming, the precise meaning of 'smallest unit of program control' is not vital, and our main concern is to distinguish 'simple' statements from 'compound' ones.

In addition to the various constructs which combine simple statements as parts of larger simple statements (as i l lustrated above), there is a way to combine arbitrary statements to form what is called a "compound statement". This is done by, separating the statements by semicolons (;) and enclosing the uhole thing between the pair of keywords 'BEGIN and END. (This syntax should be familiar to ALGOL programmers.) Alternatively, '|_'

and '_|' may be substituted for 'BEGIN' and 'END'. Compound statements are syntactically equivalent to simple statements (i.e., they may be used wherever simple statements can), so simple and compound statements are conglomerately referred to as just 'statements'.

Executing a compound statement is exactly equivalent to executing each of the sub-statements in sequence. The difference is that a compound statement is treated as a single statement, and can thus be used in places where a single statement is required. For example, instead of writing

```
IF i=j, do_this;
IF i=j, do_that;
IF i=j, do_something_else
```

we may write

```
IF i=j, |_ do_this; do_that; do_something_else _|
```

Note, however, that in the second form, if 'do_this' changes the value of i or j, we will still continue with 'do_that' and 'do_something_else', which is probably what we want. To do the same thing using the first form, we would have had to first create some spare variable (often called a 'temporary' variable) to 'remember' whether i=j were true, i.e., whose value is unaltered by doing do_this or do_that.

If you are used to FORTRAN or PL/1 and are not familiar with ALGOL's use of semicolons (which is the same as MPL's, in case you wonder why we bring it up), it may help you to bear in mind the following maxim: Semicolons separate statements from other statements. They do not necessarily separate statements from keywords, They do not necessarily mark the ends of statements. Thus we do not write

```
BEGIN; do-this;
IF i=j THEN do-that; ELSE do_tell; END;
```

but instead

```
BEGIN do-this;
IF i= j THEN do-that ELSE do-tell END
```

The keyword END may have to have a semi-colon after it if another statement follows it, as in

```
BEGIN do-this;
IF i-j THEN do-that ELSE do-tell END;
do-away
```

If you don't see how it could fail to have another statement following it, consider this possibility:

```
IF a-b THEN
      BEGIN do-this;
      IF i-j THEN do-that ELSE do-tell END
ELSE
      do-away
```

Simple statements in MPL may be broken into two classes--the

assignment statement and keyword statements. Keyword statements are identified by the appearance of some keyword or another as the first thing in the statement, and are used for control functions and I/O (input/output). Assignment statements are used to set variables equal to new values or sets of values. We shall describe the latter first,

4.2: Assignment Statement

The assignment statement has the form

<left side> := <expression>

The symbol '=' may be used in place of ':='. This statement causes the value(s) of the <expression> to be assigned to the <left side>, replacing previous values (if any) assigned there. The <left side> must be either a variable or a subscripted variable: in the latter case only the specified element is affected by the assignment. (We introduce the nomenclature '<left side>' so that we may refer to it when the same construction is used in other types of statements,) The <expression>, as you would expect* is any valid MPL expression.

The variable named in the left side must have been previously defined (see section 2.3), and the left side must have the same dimensionality and size as the expression. (Thus if the left side is a subscripted variable (that is, a single component of a non-scalar variable), the expression must be scalar, regardless of the dimensions of the variable.) if the expression is scalar but the left side is an unsubscripted non-scalar variable, the scalar value will be assigned to each component of the non-scalar variable. A real value assigned to an integer is truncated toward zero to yield an integer value.

The expression is evaluated completely before any of the resultant values are assigned to the left side. Thus if M is a square matrix, the statement

```
M := M*M
```

computes M-squared and temporarily stores it elsewhere, then transfers it to where M is stored, It does it this way because if each element of M were replaced as soon as the corresponding element of M-squared were computed, it would affect the computation of later elements.

Although the symbol '=' (as used in FORTRAN and PL/1) is permitted in place of ':=' (as used in ALGOL), the latter is recommended as it can help you remember that the statement is an assignment, not an algebraic statement of equality. The canonical example used to emphasize this is

```
i := i+1
```

This is a valid statement, meaning 'replace the value of i by that value plus 1' and store the new value in the same location in memory. It in no way implies the algebraic impossibility 'i equals i+1'.

An alternate form of the assignment statement is

DEFINE <variable> := <expression>

where, again, '=' may be used. Note, however, that the left side may not be subscripted. That is, it must refer to an entire variable and not to one of its components* When using this form, the variable need not have been previously defined nor, if previously def inecl, need i t have the same dimensions as the expression. If previously defined, however, it must have the same type and structure as the expression, since type and structure are not allowed to change (see section 2.3). The effect of this statement is to evaluate the expression, define the variable to have the same type and dimensional i ty as the expression, and then assign the values just evaluated. As with ordinary DEFINE statements, any previous dimensions or values of the variable are discarded and forgotten.

Examp l es:

```
DEFINE V VECTOR 4,
     p=3, q=2,  "p and q are now scalar integers"
     M MATRIX p BY q,   "M i s 3 by 2"
     M' MATRIX q BY p;"M' is 2 by 3"
DEFINE i := 3;  "i is now a scalar integer"
V := (i,i*i,...,25);  "V has the value (3.,9.,15.,21.)"
V(i) := V*V/5;  "V is (3.,9.,151.2,21.)"
V := 73;  "Since V is real, the 73 is treated as 79.0,
          "V has the value (79.0,79.0,79.0,79.0)"
V := (1,...,3);  "Illegal (V has size 4)"
DEFINE V := (0.,0.,0.);  'Redefines V as a 3-element
                          "real zero-vector"
V := (1,...,3);  "Legal this time since V is now size 3.
                 "V becomes (1.0, 2.0, 3.0)"
M := M';  "I l legal (sires don't match)"
M(1,2) := M'(2,3);  "Copies one element from M' into M"
M := M'(2,3);  "Replaces every element of M by the
               "sca l ar M'(2,3)"
DEFINE W INTEGER VECTOR 3;
W := (-7.9, 0., 7.9); "S i n c e  W is integer, it is
                       "assigned (-7, 0, 7)"
DEFINE W := (-7.3, 0., 7.9);  "Illegal (W is integer
                  "and cannot be redefined as real)"
```

4.3: IF Statement

The IF statement causes one of two statements to be executed depending upon some condition. The form is

IF <condition> THEN <statement 1> ELSE <statement 2>

where a comma (,) may be used in place of 'THEN' and 'OTHERWISE' in place of 'ELSE'. The <condition> may be any of the relational operations

discussed in section 3.1.4. (More compl icated conditions ui l l be described in Section II.) If it is true, then <statement 1> is executed and <statement 2> is ignored. If the condition is false, then <statement 1> is ignored and <statement 2> is executed. Both <statement 1> and <statement 2> must be single statements, but they may be compound statements. The IF statement, in the form shown above, is itself a single statement regardless of the internal structure of <statement 1> and <statement 2>.

If no action is desired when the condition is false (no action, that is, other than proceeding to the next statement), the 'ELSE <statement 2>' may be omi tted. This leads to the canonical 'ambiguous else' problem, which is resolved in the canonical manner: An ELSE-clause is always associated uith the most recent IF uhich does not as yet have a corresponding ELSE. If you are fami liar with this, you may skip the fine print below.

The 'ambiguous else' refers to the following statement, which is perfectly valid in MPL:

```
IF i>1 THEN
     IF i>5 THEN j=0
     ELSE j=1
```

(Note, incidentally, that the 'j=0' and 'j=1' are assignments, not conditions. See why we prefer ':='?) The problem here is that the 'ELSE j=1' could be part of either IF, the other IF being without an 'ELSE-clause'. If the ELSE belongs to the first IF, the net meaning is "if i>5 then j is assigned the value 0, if i≤1 then j gets 1, if 5≥i>1 then j is unchanged". If the ELSE belongs to the second IF, the meaning is "if i>5 then j gets 0, if 5≥i>1 then j gets 1, if i≤1 then j is unchanged". MPL resolves this by the rule given above, which is the same way ALGOL, SAIL, PL/1, and others do. In the example shown, this corresponds to the second interpretation. To force the first interpretation, an IF may be put inside a compound statement:

```
IF i>1 THEN
     |_ IF i>5 THEN j=0 _|
     ELSE j=1
```

Since the '_|' cannot appear inside the middle of a statement, the 'ELSE j=1' must be part of the first IF.

Examp l es:

```
IF x-0 THEN i=1          "If x is zero then i is assigned
     ELSE i-2            "the value 1, otheruise 2"

IF  i  --  j,            "If i differs from j then
  |_ DEFINE TEHP :=i;    "interchange their values"
     i  :=  j;  j  :=TEHP _|
```

4.4: WHILE Statement

I t has been proven that al l program control structures (i.e., things which control the order in which a program executes, such as IF statements, as opposed to things which control what a program does, such as data structures and assignment statements) can be reformed into three basic structures: compound statements, IF statements, and loops--the abi l ity to repeat a sect ion of code

without having to write it more than once in the program. Now, although these three forms are sufficient for designing any program, other forms are often more convenient. MPL has some of these more convenient forms, as we'll see later. We've described compound statements and IF statements--that leaves loops.

Typical uses of loops in MPL might be: (i) creating a Hilbert matrix by repeating the statement M(i, j):=1.0/(i+j-1) for various values of i and j; (ii) computing successive terms of a Taylor series until the terms become insignificant; (iii) performing an entire program using different sets of data by reading some data, executing the program, and then repeating those two steps until there is no more data to be read. Notice that in the first case the statement is to be repeated with certain variables (i and j) having different values over some range. In the last two cases the repetition is to continue as long as some condition is met; either the terms are still significant or there is data left. The first type of loop is written using a FOR statement. The second uses a WHILE statement, which we shall now describe.

The WHILE statement provides a means for repeating the execution of a statement as long as some condition remains true. The form is

WHILE <condition> DO <statement>

where a comma (,) may be used in place of 'DO'. The <condition> is the same as for IF statements, as described in section 4.3, and the <statement> is any one statement, possibly compound. The condition may be false to begin with, in which case the <statement> is not executed at all. The above form is equivalent to

```
loop:
IF <condition>,
    |_ <statement>; GOTO loop_|
```

where the 'GO TO' statement means just what you'd expect. (If you're not sure just what you'd expect, forget we mentioned it.)

Examples:

```
"(1) Compute n! = 1·2·3·....·(n-1)·n, called 'n factorial'.
     "The variable 'fact' contains the result."
     DEFINE i := n;  "Copy n into a temporary variable'
     DEFINE fact := 1.0:  "Initial ize result to 1 "
     WHILE i>0 DO  "Multiply by each factor"
          |_  fact  := fact*i; i := i-1  _|
     'The above loop could also be done using a FOR statement,
     "as we shall see later"

"(2) The following is Euclid's algorithm for finding the
     'greatest common divisor of two positive integers, m and n.
     "Unlike the previous example, this loop changes the values
     "of its input variables,"
     WHILE n ¬= 0,
          BEGIN DEFINE temp := m-(m/n)*n;
     "As noted earlier, the above yields m mod n"
               m := n;
               n := temp
          END
     'The answer is the current value of m"

"(3) This next example doesn't do anything useful. It does,
```

```
"however, use much of what we've covered so far, so it
"might be a useful exercise for you to step through it
"and see how it works."
DEFINE V VECTOR 9,
     (i,j) INTEGERS:
i := 1;
j := 20;
WHILE i ¬= (i/2)*2 "i.e., while i is odd" DO
     IF j¬= (j/2)*2 THEN
          WHILE j<30 DO
          |_   i:=i-1; j:=j+i-7/i; V(1.5*i):=i+j  _|
     ELSE
          WHILE i*j<100 DO
          |_   i:=i+2; j:=j-1; V(i/2):=j  _|
"When all. the thrashing is over, i is 4, j is 32, and V is
"{19.,29.,17.,33.,α,36.,31.,β,28.}, where a and β could
"be anything, since we never assigned any values to V(5)
"or V(8). If you have trouble getting these results,
"remember that integer division truncates. i should take
"on the sequence of values 1, 3, 5, 7, 6, 5, 4, 3, 5, 4.
"j should be 20, 19, 18, 17, 22, 26, 29, 30, 29, 32."

"(4) This final example compute9 the sine of a variable X,
     "using a rather blunt approach--evaluation of the Taylor
     "series until additional terms are insignificant. That
     "is, adding them to the sum produce9 no change in i ts
     "value. This happens due to the limited precision of
     "real values: 1.0 t 1E-20 yields 1.0, 'exactly'."
     DEFINE term := X; "First term of series"
     DEFINE sin :=X; "Sum of series after 1 term"
     DEFINE old :=0; "Value to compare against"
     DEFINE i := 1; "Index of last term computed"
     WHILE old ¬= sin,
     |_   old := sin; "Remember what we had so far"
          i :=  it2;                "Compute next term: terms are
          term := -term*X*X/((i-1)*i); "±X^i/i! for i odd"
          sin := sin t term    "Add term into series"
     |
     'Note that if X=0, the loop is never executed, and
     "the program immediately yields sin=0, as it should.'
```

Actually, although example (4) above is a perfectly good example of one way you might use a WHILE loop, there is a 'bug' in it. Can you spot it? It is a rather obscure problem, which we'll cover in section 7.

4.5: FOR Statement

The FOR statement allows repetitive execution of a statement with a varying index variable: i.e., a variable which is automatically assigned a new value before each execution of the statement. Note: If you're used to

similar loop-structures in ALGOL, FORTRAN, PL/1, or the like, watch closely--MPL is just a wee bit different. The form of the statement is

FOR <for index> IN <integer vector> DO <statement>

where '=' or ':=' may replace 'IN' and a comma (,) may replace 'DO'. The <for index> is any variable, sometimes called the 'index variable' of the FOR loop. It need not have been defined previously. Whether or not it has, it will be temporarily redefined as a scalar integer for the duration of the FOR statement, after which it will regain its previous attributes and value (if any). The <integer vector> is just what it says--any expression whose value is a vector of integers. The <statement> is any single statement, as with WHILE loops.

For each element of the integer vector, in sequence, the index variable is assigned the value of that element and the <statement> is executed. Thus the simple statement

```
FOR i = (1,...,3) DO V(i) := i*i
```

is equivalent to the compound statement

```
|_ i := 1; V(i) := i*i;
   i := 2; V(i) := i*i;
   i := 3; V(i) := i*i _|
```

The <integer vector> may be null, in which case the <statement> is never performed. Thus

```
FOR i IN (1,...,n) DO <statement>
```

will do nothing if $n \le 0$.

Examples:

"(1) Compute a Hilbert matrix of order n.i.e., $H_{ij}=1/(i+j-1)$"

```
DEFINE H MATRIX n BY n;
FOR i IN (1,...,n),
    FOR j IN (1,...,n),
        H(i,j) := 1.0 / (i+j-1)   "Use 1.0 to avoid integ div"
"At this point, we are outside the i and j loops, so that
"these i and j are no longer defined"
```

"(2) Find the sum of the elements of an integer vector V"

```
DEFINE sum := 0;
FOR i IN V DO
    sum := sum + i   "Note, not V(i)"
"As an exercise, suppose V=(3,2,2). Compare the results of
"sum:=sum+i with those of sum:=sum+V(i)."
```

"(3) Compute the first 100 fibonacci numbers: 0,1,1,2,3,5,8,13..."

```
DEFINE FIB VECTOR 100;   "Make a place for them"
FIB(1) := 0 ;            "Start the recurrence relation"
FIB(2) := 1;
FOR i = (3,...,100) DO
```

```
        FIB(i) := FIB(i-1) + FIB(i-2)

"(4) Find n! = 1·2·...·n"
     DEFINE fact := 1.0;
     FOR i := {1,...,n}, fact := fact*i

"(5) Sieve of Eratosthenes. Print a list of all primes ≤ 10000"
     DEFINE A INTEGER VECTOR 10000;
     "We shall have A(i)=1 if i is prime, else A(i)=0."
     A=1;  A(1)=0;  "Start with everything ≥2 tentatively prime"
     FOR i := {1,...,10000},
          IF A(i)=1, "i is prime"
          |_ WRITE i:
               "Now to' sieve out al i multiples of i"
               FOR j := {2*i,3*i,...,10000}, A(j)=0
          _|
```

Note: Since the FOR index variable temporarily 'overrides' any other variable with the same name, it is bad practice to use the same name for a FOR index as for a 'normal' variable, as it can lead to confusion should you wish to refer to the 'normal' variable while in the FOR loop. For example:

```
DEFINE a=1;
FOR a= {2},
     b=a;
c=a
"b is now 2, while c has the value 1"
```

Within the statement, or 'body', of the FOR loop, you're not al lowed to change the values of the FOR-vector. If you do so, the results are at best confusing, at worst disastrous, and in any event unpredictable.

Examp l es:

```
DEFINE sum := 0;
DEFINE V := {2,...,6};
FOR i IN V,
|_   sum := sumti;
     IF i <= 5,
          V(i) := V(i) + 1    "This is a no-no!"
_|
"The sum will probably be 2+4+4+7+6 = 23, but we
"don't guarantee it."

DEFINE sum := 0;
DEFINE V := {1,...,100};
FOR i IN V,
|_   DEFINE V := {0,i,...,100}; "Good luck!"
     sum := sum + (V*V)
_|
"We won't even try to guess what this would do!'
```

4.6: GO TO Statement

A GO TO statement causes the program to continue executing starting from some specified statement, instead of proceding normally to the next statement in sequence. The form of the CO TO statement is

GO TO <label>

where the <label> identifies the statement to which to transfer (see section 2.1.3). A label is any identifier (section 1.2) and is 'attached' to its associated statement by a separating colon, thusly:

<label>: <statement>

Examp l es:

```
"(1) Test two values, save the larger one and replace it by zero'
     IF x > y, GO TO X-BIGGER;
     z := y;                    "y is bigger"
     y := 0;
     GO TO JOIN:                "Skip around next part"
  X-BIGGER:. z := x;            "x is bigger"
     x := 0;
  JOIN: "Program continues from here"

"(2) Compute the sum of the first n squares, assuming n≥0"
     DEFINE i := 0;             "Set up a counter"
     DEFINE sum : = 0;          "Accumu l at ed sum"
  LOOP: sum := sum + i*i;       "Add next square"
     i := i+1;
     IF i <= n, GO TO LOOP      "Repeat unti l i>n"
```

Actually (and hopefully), you may be wondering why the above examples are so clumsy. In fact, you may be wondering why we used GO TO statements at all, since the same effects can be obtained by the following programs:

```
IF x > y,
     |_ z := x; x := 0 _| "x    is    bigger"
ELSE
     |_ z := y: y := 0 _|"y is bigger"

DEFINE sum := 0;            "Accumu l at ed sum"
FOR i : = (1,...,n), sum := sum t i*i
```

Wel l, you're right. The latter examples *are* more readable. They also produce more efficient programs. This is because when the program reaches a GO TO, then whoever is trying to understand the program' be it you or the MPL compiler, has to drop everything and figure out what variables and loops exist at the section of program to which you've gone,

For these and other considerations, more and more programmers are beginning to favor "goto-less" code. (In case you hadn' t guessed, we the

authors share this view.) In fact, it has been proven† that, given the constructs BEGIN-END, IF-THEN-ELSE, and WHILE, it is never necessary to use a GO TO. In all fairness, however, certain programs <u>are</u> clearer if GO TOs are used. (Some languages have additional constructs to handle these cases, but MPL does not.) For example, suppose you have 3 vectors called X, Y, and Z, and you want to count the nymber of Pythagorean trios, i.e., values of $x \epsilon X$, $y \epsilon Y$, and $z \epsilon Z$ such that $x^2+y^2=z^2$. Thus if X={8,7,20}, Y={24,15,6}, Z={10,17,25}, you would count $8^2+15^2=17^2$, $8^2+6^2=10^2$, $7^2+24^2=25^2$, and $20^2+15^2=25^2$, for a total of 4. All right, no problem, here's the program:

```
DEFINE count := 0;
FOR x IN X, FOR y IN Y, FOR z IN Z,
    IF x*x + y*y = z*z,
        count := count+1
```

All very simple, very straightforward, very goto-less, But now suppose we just want to find any one such trio. In that case, once we find one trio we want to 'stop' the FOR loops, since it is a waste of time to look any further. Unfortunately, there's no easy way to do this, so we are better off just leaving the loop altogether. The new program would be:

```
FOR x IN X, FOR y IN Y, FOR z IN Z,
    IF x*x + y*y = z*z,
        |_ DEFINE Trio := {x,y,z}; GO TO FOUND _|;
DEFINE Trio := {1,1,1}; "Special vector to show no trio found"
FOUND: "Rest of program... Note that if no trio exists, it can be
    'determined by testing if Trio = {1,1,1}."
```

4.7: <u>STOP Statement</u>

Presumably at some point your program will have done all that is expected of it, and you will want it to stop. You can do this in either of two ways. You can reach the end of the program (i.e., the last card), at which point you just sort of fall off the edge of the world, and the program wraps things up and goes away. If you don't happen to be at the end of the program, you can stop by executing a STOP statement, which consists of the single keyword

```
STOP
```

(You could get the same effect by putting a label at the end of the program and going to that label, but heaven forbid that we should make you use a GOTO when all you really want to do is STOP.)

A STOP may be used anywhere in the program. You could be inside a FOR loop inside an IF inside a WHILE loop; it doesn't matter. Once you execute a STOP, that's the last thing the program will do. (This is not to imply that you can never use the program again! But you <u>will</u> have to tell the computer to go get MPL again before you can rerun your program.)

†Bohm and Jacopini, *Communications of the ACM*, May 1966.

5 : INPUT/OUTPUT

If the primary purpose of this manual were to teach programming to novices, as opposed to teaching MPL to everyone, we would probably have started with this section. The terms "input" and "output" refer respectively to the reading of data into the program (typically from cards) and to the printing of results. It is difficult to conceive of any useful program which does not at some point need to produce some sort of output indicating the results of its computations, and few programs are so simple as to require no input data. Such data could be 'built into' a program by assigning constants to several variables, but by inputting the data you make it possible to change that data without having to modify the program.

5.1: Unformatted Data

The input/output (I/O) statements described here are "unformatted". This means that data being read is in the form of a series of constants (as described in section 1.3), as many (or as few) per card as you desire, with one or more spaces separating them. Values being output are printed in a fixed form which leaves room for several digits even if they are not there. For example, an integer is printed as 14 characters, many of which may be leading blanks, followed by 2 blanks. Real values print using 22 characters, again followed by 2 blanks. Examples of unformatted output will be shown shortly.

These data forms are called "unformatted", or sometimes "free-format", to distinguish them from (what else?) "formatted" data, which will be introduced in Section II. Formatted I/O allows you to specify exactly how many characters are to represent the external data (on cards or in the printout), how many significant digits to print, and so forth. This is nice for output if you need to set up tables or other output requiring alignment into columns, etc. For input, however, free-format is usually preferable. As an example of free-format input data, any of the following would be acceptable data to represent the real vector [2.,3.,4.].

```
2.0  3.0  4.0

or

2 3
4

or

2.
(blank card)
                .3E+1
     +4D0
```

For all I/O statements, matrices and arrays are processed in row-major order. That is, a 2 by 3 matrix M would have its elements read or written in the order M(1,1), M(1,2), M(1,3), M(2,1), M(2,2), M(2,3). (This is the same as in all major languages except FORTRAN.)

5.2: Output

Unformatted output is accomplished via the WRITE statement, which has the form

WRITE <expression list>

where ANSWER may be used in place of WRITE. The <expression list> is a list of one or more expressions separated by commas. The list may be enclosed in parentheses if you feel this makes it clearer, but they are not required. The expressions (which may be variables, constants, or more complicated expressions) have their values printed in the order in which they appear in the WRITE statement. Each WRITE statement starts a new line in the output, but more than one value may be written on a line by a single WRITE,

Example:

```
DEFINE V INTEGER VECTOR 3,
       s REAL SCALAR:
s = -7.9;
V = 1012345, -43210, 0);
WRITE s;        "See sample output below
WRITE (V);      "to see what this does."
ANSWER s*s, V*V, (V*s)/100000
"This last statement writes a real scalar, ah
"integer scalar, and a real vector"
```

The above would produce:

```
-7.90000000000000
   12345          -43216                0
 62.4100000000000     2019503125     -0.975255000000000     3.41359000000000
```

5.3: Input

Unformatted input is done with the READ, statement, which has the form

READ <left-side list>

The <left-side list> is a list of <leftside>s as defined in section 4.2, which is to say that each must be either a variable or a subscripted variable, and must be previously defined. As with the WRITE statement, the items in the list are separated by commas and the list may (if desired) be enclosed in parentheses. The effect of the READ statement is to scan input data for as many values as are needed to fill the items named, and assign those values to the items in the list. Note: Only the first 72 characters of each input line are used.

Example:

```
DEFINE V VECTOR 5;
DEFINE i = 2;
READ V, i, V(i)
```

If the input data looked like

```
1.2  -3  4E2  20-2  7.9  4  -3E-3  12.34
```

then the READ would first assign V the vector (1.2, -3.0, 400.0, 0.02, 7.9). It would then assign i the value 4, and finally read V(4), so V would end up as (1.2, -3.0, 400.0, -0.003, 7.9). The '12.34' would be unused and could be read by a later READ statement.

An' alternate form of the input statement is the GIVEN statement. It looks exactly like a DEFINE statement with the keyword GIVEN replacing DEFINE.

```
GIVEN (a, b, c) INTEGERS,
    V REAL MATRIX 2 BY 3
```

A GIVEN statement is equivalent to a DEFINE followed by a READ. Thus the above example would define 3 integer scalars and a real matrix, and would read 9 data items and assign the values to the newly-defined variables.

Each variable is read before the next is defined, so the statement

```
GIVEN Vsize INTEGER,
    V VECTOR Veize
```

is perfectly valid. By the time V is defined, Vsize will have been assigned a value.

5.4: Writing Messages

You'll probably want to include some words in your output, to identify what all the numbers mean. Restages are also commonly used to say what happened if something goes wrong.

Arbitrary strings of characters may be output by enclosing them between double-angles ('<<' and '>>') and including them as items in a WRITE statement.

Example:

```
IF y=0,
    WRITE <<Error: y is zero.>>
OTHERWISE
    WRITE <<x/y = >>, x/y
```

Such strings may not include a '>>' symbol, since that symbol is being used to identify the end of the string.

6: HOW TO USE MPL

In Section II we' l l show you some of the fancier ways you can use MPL. For now we' l l cover just the 'standard' way to run an MPL program. There are three aspects to this. First, there are a few finishing touches to put on your program. Second, there are some-details concerning how to type or punch it. Third, there are the control cards (JCL) which turn the program into a job which can be run on the computer. None of this is really very compl icated, so let's get on with it.

Assuming you've written your program as a sequence of statements, separated by semicolons, just precede the first statement with the line

```
PROGRAM
```

and follow the last statement with the line

```
END.
```

(note the period), Thus a program to read a square matrix and print its square might look like this:

```
PROGRAM.
GIVEN n INTEGER,
      M MATRIX n BY n;
ANSWER M*M
END.
```

In typing or punching your program, be careful never to use more than the first 72 characters of each line. Anything typed in columns 73 through 80 wi l l appear in your program listing, but wi l l be ignored by the MPL compi ler. Thus they may be used for sequence informat ion or WYLBUR l i ne number s.

I f you don' t know about WYLBUR, don' t worry about i t . As for what we mean by 'sequence information', a frequent use of columns 73-80 of punched cards is to put numbers there, in effect numbering the cards of the deck. Then if you accidentally drop your deck there are machines which wi l l sort the cards back in to their proper order, a nice feature to have with large decks. If you do dec ide to put sequence numbers on y o u r cards, use multiples of 10 (10, 20, 30, etc.) instead of numbering the cards 1, 2, 3 . . . That way you'll be able to insert new cards later without having to renumber everything.

Finally, here is the JCL, showing you what the whole 'job' should look l ike. Lower-case letters indicate information which should be euppl ied by you.

```
//jobname JOB (accounting information),'your name'
/*KEY keyword
// EXEC MPL
//COMP.SYSIN 0  0 *
   your program goes here, complete with PROGRAM and END I ines
/*
//GO.SYSIN 0  0 *
   free-format input data goes here: if your program uses no READ
   or GIVEN statements, the GO.SYSIN card and the following /* card
   may be omitted
/*
```

7: RESTRICTIONS, CAUTIONS, AND PITFALLS

I t is our purpose in this section to emphasize certain specific problems you should watch out for. Some of these are restrictions which we have not as yet ment ioned. Others are things which, though legal in MPL, should be used with caution. Still others are pitfal Is into which people new to MPL often seem to stumble. All of these warnings are presented here in an order less jumble: the precise nature of any specific problem (i.e., whether it is a restriction, a caution, or a pitfall) should be clear from its description.

7.1: MPL In General

7.1.1: Definitions

We have mentioned this once before, but it deserves to be mentioned again. If you leave the parentheses out of a DEFINE statement, it wi I I usually still be a valid statement, but it will generally not mean what you intended, Thus the two statements

```
DEFINE (a, b, c) INTEGERS:
DEFINE a, b, c INTEGERS:
```

perform different actions. The first defines three integers. The second defines c to be integer, but a and b to be real, since no attributes are specified for them.

7.1.2: Redefinitions

Another problem to watch out for arises from the fact that only the dimensions of a variable may change through redefinition; dimensional i ty and type must (at least until we learn about block structure) remain fixed throughout the program. This does not generally cause difficulties, but

can lead to trouble in the use of 'temporary' variables--variables intended to be used briefly in one small section of the program. An example in section 4.3 included this code for interchanging the values of two variables.

```
DEFINE  temp  := i; i := j; j := temp
```

Suppose that, elsewhere in this same program, you wish to swap the values of two other variables, x and y, and you use the code

```
DEFINE  temp  := x; x := y; y := temp
```

If i and j are integers whereas x and y are real, or i and j are scalars whereas x and y are vectors (among other possibilities), this is invalid. It would be caught by the MPL compiler as an attempt to define temp as two different objects. It is probably bad programming practice in general to use the same name for variables, even temporary variables, with two different uses. We shall not argue the point here, In MPL, however, it is certainly wise to use different names for different variables. In the preceding example, you might use 'temp_i' and 'temp_x' for the tuo temporary names.

7.1.3: Vector aenerators

Remember that each element of an N-tuple must be of the same type. Thus {1, 1.25, 1.5, 1.75, 2} is invalid and would have to be written as {1., 1.25, 1.5, 1.75, 2.}. Recall also that only integers may appear in an arithmetic progression, so the preceding may not be replaced by {1., 1.25,..., 2.}.

Another thing to bear in mind with regard to arithmetic progressions is that the symbol ',...,' may not have any internal blanks. Thus {1, 2, ... 10} is invalid. Although this caution applies to all multi-character symbols, we emphasize it in this particular case due to the tendency to follow all commas with blanks.

7.1.4: FOR loops

This is actually the opposite of a warning. It is a reminder that something *is* safe. You were warned once about changing the vector of a FOR statement during the FOR loop. You should remember though that changing a variable which appears in a vector does not affect that vector, so a FOR statement with the vector {1,...,n} is allowed to change n. As a, perhaps unlikely, example, suppose you wish to replace n by the sum of the numbers 1 through n. This could be done by the loop:

```
FOR i IN {1,...,n-1}, n:=n+i
```

This is probably not the clearest way to do this, but our point here is that it at least is legal.

7.2: Compiler-Specific Warnings

The MPL 'compiler', in case you've read this far without finding out, is the program which reads in your MPL program and transforms it into instructions the computer can understand. At the time of this writing there is only one MPL compiler, and it has certain quirks which ought to be brought up.

7.2.1: List sizes

In a DEFINE statement, the number of variable names which may be grouped together by parentheses and given a common set of attributes is limited to 30. Thus, the statement

```
DEFINE (v01, v02, v03, v04, v05, v06, v07, v08,
        v09, v10, v11, v12, v13, v14, v15, v16,
        v17, v18, v19, v20, v21, v22, v23, v24,
        v25, v26, v27, v28, v29, v30, v31) INTEGERS
```

is invalid, and would have to be broken up, as in

```
DEFINE (v01, v02, v03, v04, v05, v06, v07, v08,
        v09, v10, v11, v12, v13, v14, v15) INTEGERS,
  - - . (v16, v17, v18, v19, v20, v21, v22, v23,
        v24, v25, v26, v27, v28, v29, v30, v31) INTEGERS
```

The same limit also applies to the number of variables or expressions which may appear in any single I/O statement,

7.2.2: Writing expressions

If the keyword WRITE is followed by a left parenthesis, the MPL compiler assumes that what you are doing is enclosing the output list in parentheses (you may recall that this is optional, such that

WRITE a, b, c and WRITE (a, b, c)

are equivalent statements). Unfortunately, this can lead to trouble, as in the statement

```
WRITE (a+b)*(c+d)
```

This statement is valid according to the definition of the language, but the MPL compiler gets confused by it. So if the first thing in a WRITE statement starts with a parenthesis, be sure you include the 'optional' parentheses around the entire list, thusly:

```
WRITE ((a+b)*(c+d))
```

7.2.3: Constant subscripts

While your program is running, every subscript will be checked before it is used so as to insure it is in the proper range, For efficiency, however, this checking is foregone when the subscript is a constant. (The

compiler assumes you know better than to explicitly use the 10^{th} element of a 5-element vector!) This means that it will go undetected if by some mishap you do misuse a constant subscript, say by using the wrong matrix or by thinking that V(0) exists. The specific case of a zero (or, in general, non-positive) constant subscript may be detected by a later version of the compiler, but at present it is not.

7.3: Programming Pitfalls

7.3.1: WHILE loops

It should be fairly obvious that if you want a WHILE loop to terminate normally (without your having to use a GO TO or STOP statement) then something within the loop must affect the WHILE-condition. That is, for example, if you have a loop 'WHILE i>j', then something in the loop had better change the value of either i or j, lest the loop never terminate.

Yet despite this obvious fact, it seems to be a common problem, and 'infinite loops' come up every day. We don't know the reasons--perhaps by the time a programmer finishes writing a 30-statement loop he tends to forget all about that one little statement he had been meaning to tack on at the end. If that is indeed part of the problem, remember that it's perfectly all right, when writing a program, to skip a little ways down the page and write the end of the loop before sitting back to decide what goes ahead of it.

7.3.2: Usina integers instead of reals

You may recall that, when we described WHILE loops back in section 4.4, we noted the existence of a bug in one of the examples. The error is a common one, though subtle, and involves the unintentional use of an integer variable where a real one is needed. Let us reproduce the program for purposes of perusal,

```
'This little example computed the sine of a variable X,
'using a rather blunt approach--evaluation of the Taylor
"series until additional terms are insignificant.  That
"is, adding them to the sum produces no change in its
"value.  This happens due to the limited precision of
"real values; 1.0 + 1E-20 yields 1.0, 'exactly'."
DEFINE term := X;  "First term of series"
DEFINE sin := X;  'Sum of series after 1 term"
DEFINE old := 0;  "Value to compare against"
DEFINE i := 1;  "Index of last term computed"
WHILE old ¬= sin,
|_   old := sin:  "Remember what we had so far"
     i := i+2;                "Compute next term: terms are
     term := -term*X*X/((i-1)*i);  "±X^i/ i ! for i odd"
     sin := sin + term     "Add term into series"
 |
'Note that if X=0, the loop is never executed, and
"the program immediately yields sin=0, as it should.'
```

The intent is that-, within the loop, the variable 'old' is used to store the current value of the series summation. Then the next term in the series is computed and added into the sum. The WHILE I oop compares the new sum with the old one and terminates if they are equal, indicating that the latest term was insignificant. The problem is that 'old' is initial ly clef ined equal to zero, an *integer* zero! so 'old' is an integer and, for typical values of X, will always remain zero (since for small X the sum is always between ±1). When we do 'old := sin', the value i s rounded toward zero, discarding the fractional part of 'sin'. But when the loop tests whether 'old ¬= sin', it converts 'old' to real for the comparison; since 0.0 ¬= sin, the test fails and the looping never terminates.

Let's drive this home with another example. Suppose we wanted to compute the harmonic number $H_n = 1+(1/2)+(1/3)+...+(1/n)$. We try the following program:

```
DEFINE Hn := 0;
FOR i IN (1,...,n),
     H n := Hn t (1.0/i)
```

Since Hn is being defined as an Integer, we first add 1.0 to it, giving a total of 1, but after that its value wi ll never be changed. When i=2, Hn is assigned 1+0.5, which gets rounded down to 1. Then i-3, and Hn is assigned 1+0.333..., which again is 1. And so forth.

The problem of using integer constants when defining what is meant to be a real variable is hard to spot, because the symptoms may vary. In the sine computation, the program works for X=0, but loops forever for any other value of X, making it appear that the series simply isn't converging. In the harmonic sum, the program will finish but yields erroneous results. So the best time to watch for this is when you're writing the program; you' l l no doubt have more than enough bugs to worry about once you try running it!

7.3.3: Using reals instead of integers

Despite the title above, this problem bears little relation to that just discussed, except insofar as both involve the use of a data type which is unable to maintain the accuracy you need. In this instance the problem centers on the fact that real values are actually stored as binary fractions, with a limited number of significant bits (binary digits). Thus when you write 0.001, which is an infinite repeating fraction in binary, what you actually get is the nearest finite approximation, which is sl ightly above or below 0.001, The difference is practically negligible, al though in doing thousands of operations the rounding error may -accumulate. This is the realm of the numerical analyst, and we shal l not discuss i t here. There are times, however, when even an insignificant 'rounding error such as one part in 10^{14} may suddenly make a difference.

Suppose we wish to find the sum: sin(.002) t sin(.004) + . . . + sin(.998). We could try doing it this way:

```
DEFINE sum-of-sines := 0.0,
     argument := 0.002;
WHILE argument <1.0,    "Can't use FOR with real values"
|_   DEFINE X := argument:
     "This routine for computing sin X should be familiar
     "to you by now."
     DEFINE term := X;
     DEFINE sin := X;
     DEFINE old := 0.0;
     DEFINE i := 1;
     WHILE old ¬= sin,
     |_   old := sin;
          i := i+2;
          term := -term*X*X/((i-1)*i);
          sin := sin t term
     _|;
     sum-of-sines := sum-of-sines + sin;
     argument := argument t 0,002
_|
```

This might work, but it might not. Suppose that the nearest approximation to 0.002 turns out to be slightiy smaller, say $0.002-\epsilon$? (This happens to be the case; $\epsilon \approx 1.26\times10^{-20}$.) Then we will get $\sin(.002-\epsilon)$ t $\sin(.004-2\epsilon)$ + ... + $\sin(.998-499\epsilon)$. Of course, ϵ is very small, so the errors in the sines are not enough to worry about. But after we compute $\sin(.998-499\epsilon)$ and add it to the sum, we increment the argument to be $1.000-500\epsilon$. Since *this is* less *than 1.0* the loop will not terminate until we have added $\sin(1-.500\epsilon)$ to the sum, which is not what we wanted to do.

We could take care of this by using

```
WHILE argument <= 0.338
```

but this would fail if 0.002 turned out to be approximated by $0.002+\epsilon$, since $0.998+499\epsilon$ is greater than 0,998. If we use a 'middle ground', as in

```
WHILE argument < 0,999
```

this will work, but it is dodging the issue and is unclear besides. The recommended tactic here is to use an integer counter, since integer values are represented exactly, then divide to obtain the real values required. In the case above, this becomes

```
DEFINE sum-of-sines :=0.0;
FOR argument =(2,4,..,998),    "Can use FOR this way!"
|_   DEFINE X := argument/1000.0;
     "Compute sin as before...code not shown"
     sum-of-sines := sum_of_sines t sin
_|
```

7.3.4: 'Mixed mode' arithmetic

Most of the time you can mix integer and real values in an expression and never have to worry. HPL will automatically treat integers as the equivalent real values whenever real and integer values are being combined. We have already warned about the consequences of accidentally assigning a real result into an integer variable.-. Our point here is a different one. The appearance of a real value in an expression does not cause everything in the expression to be treated as real. An integer value remains an integer until the 'last moment', when it is about to be combined with a real value. As an example, suppose i and j are integers with the values 1 and 2, respectively, and x is real with value 2.0. Then the expression `j+i/x` is real with value 2.5, while the expression `x+i/j` is also real but with value 2.0. This is because the division i/j is done as an integer division, and `1/2` yields zero.

To see how this might arise in practice, suppose the harmonic summation discussed earlier had been written as

```
DEFINE Hn := 0.8;     "Remember to make it real!"
FOR i IN {1,...,n},
     Hn := Hn + 1/i     "Should use '1.0/i'"
```

This will yield exactly the same results as if Hn were defined to be an integer, since all terms for i>1 will be zero.

7.3.5: Integer overflow

When we discussed integers and reals in section 1.3 we mentioned limits on the magnitudes of such values, You might well ask, what happens if these limits are exceeded? With reals the result is fairly clear, If a real value gets too large (say you try to compute 1E50*1E50) you will get an error message. If it gets too small (as in 1E-50*1E-50) the result is taken to be zero. If an integer gets too large, the results are more vague.

If an integer expression results in a value not between -2147483648 and +2147483647 (yes, we didn't admit it before, but negative integers can get 1 larger than positive ones), i.e. -2^{31} and $+2^{31}-1$, then 2^{32} is added to or subtracted from the value until it is within the required range. Thus if you add 1 to 2147483647, you get -2147483648, and if you double this you get 0. This can have bizarre results. Suppose you want to per form some loop for i={0,500000000,...,2000000000}, and for some obscure reason you use a WHILE instead of a FOR. Let us look at what might happen.

```
DEFINE i := 0;
WHILE i <= 2000000000,
|_   WRITE i;
     i := i + 500000000
_|
```

This will write 0, 500000000, 1000000000, 1500000000, and 2000000000, as expected. But then it adds 500000000 to i and gets -1794967296, which is less than 2000000000, so the loop continues, printing -1794967296,

-1294967296, . . . , 1705032704, At this point adding 500800000 again yields a negative result, -2089934592, and on i t goes, The loop will print a total of 38 numbers, after which i becomes 2115098112.

But now comes the best part. Due to the way the HPL compiler handles index sets in FOR loops, if you use the vector {0,500000000,...,2000000000} in a FOR loop you wil l get the same effect as above! That is,

FOR i IN {0,500000000,...,2000000000}, WRITE i

will print the same 38 numbers. (This is really a compiler-specific 'bug', but it is a problem which is likely to be shared by future MPL compi iers, so watch out!) Interestingly, the statement

WRITE {0,500000000,...,2000000000}

works correctly, printing only 5 numbers.

Another place where integer overflow might cause trouble is when you are multiplying to create an over-sized number, which you then intend to divide back down. For instance, in computing a binomial coefficient such as $\binom{n}{3}$ using the formula n*(n-1)*(n-2)/6, you will get erroneous results for values of n between 1292 and 2345, (For n>2345 the final resui t would be too large anyway.) For n=2000, for instance, the answer you want is 1331334000. Unfortunately, 2000*1999*1998 yields -601930592, and the final resui t comes up -100321765. In such cases, where you expect integer expressions near 2^{31} use real values instead. Real variables can represent integer6 up' to about 10^{12} uith no rounding errors, and though they are somewhat slower to use, they atleast work, Remember that rea l division does not truncate as does integer division, even when the values involved happen to be whole numbers.

SECTION II

You have just (presumably) learned the rudimentaries of MPL--enough to be able to use it for non-trivial programs. If you are new to programming (or vi ceversa) you may have been overwhe l med by how much there i s to MPL. We apologize; practice it, use it, become fami l iar with it. If you are accustomed to programming, you may have been disappointed by how l i t t l e there is to HPL. We apologize: keep reading.

These fine-print sections will become scarcer as you proceed through the manual. There are two reasons for this. First, you are hopefully becoming a more sophisticated programmer. Second, the subject matter is becoming more complex, so what might have been presented in fine print is instead presented normally, for the benefit of experienced programmers and novices alike.

1: DATA TYPES AND STRUCTURES (REVISITED)

Al l right, we admit it. There *are* data types other than integer and real, and structure6 other than scalar, vector, matrix, and array. If you've read this far (and if you haven't then you had better catch up in a hurry) then you should be ready to hear about some of them,

1 .1: More Data Types

Warning! The data types about to be introduced here can *not* be input using unformatted READ or GIVEN statements. Formatted input wi l l be discussed in section 6.2.

1.1.1: Logical

A logical constant or variable may have one of two values, 'true' or 'false'. Logical constants consist of one of the keywords TRUE or FALSE. As you shal l see, the condition of an IF statement is real ly a logical expression, and may be more complex than was shown in Section I. It may also be simpler. For examp l e, the statement

```
IF flag THEN x := 1.0/x
```

. is valid if 'flag' is a logical scalar variable. We'll see more of this in section 2.1 and section 3.2.

In a DEFINE or GIVEN statement, logical variables are declared with the type attribute **LOGICAL.**

1.1.2: Character

Character constants and variables are strange beasts. The only other language we know of which handles them similarly to MPL is APL, so if you're not familiar with APL you should pay very close attention.

A character datum is any single character which you can punch, type, or in any way force into the computer. For example; if you take any punched card, it may be thought of as containing exactly 80 individual characters, There are two cautions you should keep in mind. First, characters are not restricted to the MPL character set given at the start of Section I. Thus, although the character '?' has no meaning in MPL, it may be stored in a character variable. Similarly, a multi-character symbol such as '<(' or '¬=', which MPL treats as a single entity, cannot be stored as a single character. Thus ',...,' taken as character data would have to be treated as 5 distinct characters.

A 1-character constant is written by putting the character between double-angles ('<<' and '>>'). For example, <<$>> is a scalar character constant whose value is the single character '$'. Character constants with more than 1 character are wri tten the same way, and are treated as vectors. Thus <<FROG>> is a 4-element vector equivalent to {<<F>>,<<R>>,<<O>>, <<G>>}. MPL does not currently permit the use of <<>>, which would be a 0-element character vector. Note that multiple blanks, which are normal ly ignored, are counted as separate characters in a character constant. Thus <<X Y>> is a s-character vector, while <<X Y>> is a $-character vector. A character constant cannot contain the sequence '>>', since that is used to denote the end of the string. The examples include a way of 'creating' such a constant if you should happen to want it.

Character variables are defined using the type attribute CHARACTER.

Examp l es:

```
DEFINE Name CHARACTER VECTOR 20;
Name := <<*>>;  "Name now contains 20 asterisks"
Name(20) := <<.>>;  "Name is now 19 t's followed by a period"
Name := <<George>>;  "Illegal (<<George>> is only a
                      "6-element vector)"
DEFINE Name := <<George>>;  "Name is now 6 elements"
IF Name = <<George>>,  "This test is 'true"'
     WRITE Name:
IF Name = <<Dantzig>>,  "I l legal (can't compare 6-element
     WRITE Name:          "vector with 7-element vector)"
DEFINE Name := <<X>>;  "Il legal (<<X>> is scalar, and you
                        "can't redefine a vector as a scalar)"
```

"Here are three ways to create a string of n asterisks."

```
DEFINE string1 CHARACTER VECTOR n;
string1 := <<*>>;  "Set every element to '*'"
DEFINE s_temp = {1,...,n} - {0,...,n-1}, "Yields n ones"
     string2 = <<*.>>(s_temp);
"<<*.>>(1) is the first character of '*.', namely '*'.  Because
"s_temp contains n l's, string2 is assigned n *'s.  You must in-
"clude the period (or some other random character) in <<*.>> so
'that it will be a vector and can thus be subscripted (<<*>> is
```

```
"a scalar) . Notice also the use of a vector (stemp) as a sub-
"script. We haven't mentioned that yet, but we'll get to it in
"section 2.2. In the meantime, we can of course combine things
"and get the following simpler form."
DEFINE string3 = <<*,>>({1,...,n}-{0,...,n-1});

"Here's how to create the character string '>>'. We take the
    "3-character string '.>.' and extract the '>', twice."
    DEFINE special-string =<<.>.>>({2,2});
```

You might also have realized by now that, when you were writing messages as described **back** in Section I, you were actually writing character constants.

Be careful to always include the closing '>>' because, unlike comnien t s, character constants can run over several cards. Consider the following example.

```
IF x=y THEN
     DEFINE word = <<Frog::
ELSE
     DEFINE word =<<Toad>>
```

Here the '>>' was accidentally typed as '::', a fairly typical keypunching error. The intended statement would have set 'word' equal to one of two 4-character vectors. The statement as actually shown either leaves 'word' alone (if x≠y) or sets it to be a very long vector (148 characters, to be precise) starting uith 'Frog::' and several blanks, and ending with '= <<Toad'.

1.2: More Data Structures

There are a couple fairly bizarre data structures which we'll save for Section III. For now we shal l introduce two fairly common mathematical structures, namely row and column vectors.

12.1: Row vectors

A row vector is treated, mathematically at least, exactly like a 1 by n matrix, where n is the size of the row vector. A row vector may thus be used in the same places a matrix could be used, and is subject to the usual rules of matrix operations. As for the program, however, row vectors look a lot like ordinary vectors. They are defined with a single dimension (using the keyword ROW or, optionally, the pair of keywords ROW VECTOR) and are subscripted with a single subscript. See the examples in section 1.2.3 for cases where the difference between vectors and rows becomes apparent.

1.2.2: Column vectors

Column vectors are n by 1 matrices just as rows are 1 by n. They are defined using the keyword COLUMN or, optionally, COLUMN VECTOR.

1.2.3: Special transformations

To convert vectors into rows and columns, and vice versa' there are some special operations provided. (These are actually 'library functions', as will be described in section 4.5.) To change a vector V of any type (integer, character, etc.) into a row or column of the same type and size, write ROW(V) or COLUMN(V). To change any row or column vector RC into an ordinary vector, write VECTOR(RC)†. Note: These operations do not actually *change* the structure of V. Rather they yield a result which has the same values as V but is in a different structure. This is just like the ordinary arithmetic operations. If you write -V it does not cause the value of V to change, but instead yields a result whose value is negative that of V.

For convenience, MPL recognizes the special case of a vector value being assigned to a row or column variable. Technically, this should be illegal, since you can't assign a matrix variable a vector value. But, provided the row or column is the same size as the vector, MPL lets you get away with it.

Let's see some of these concepts in action.

Examples:

```
DEFINE V INTEGER VECTOR 10'
     R INTEGER ROW VECTOR 10,
     (C1,C2) INTEGER COLUMNS 10;
V := {1,...,10};
R := {1,...,10};  "Legal (see second paragraph above)'
R := ROW({1,...,10});  'Equivalent to preceding statement"
CL := R;  "illegal (10 by 1 is not the same as 1 by 10)"
C1 := V: "Legal"
C2 := (C1-1)*2;  "C2 is nou {0,2,...,18}"
DEFINE Z = C1*C2;  "Illegal (incompatible dimensions)"
"Note: You can multiply one column by another, provided
"the second column has size 1."
DEFINE Y = R*C1;   "Y is a scalar, value 1^2+2^2+...+10^2=385"
DEFINE X = C1*R;   "X is a 10 by 10 matrix, X(i,j)=i*j"
DEFINE W = X*C1;   "Matrix times a column yields a column"
DEFINE W = R*X;    "Row times matrix yields a row, so this is
                    "illegal (can't redefine structure of W)"
"Note that, though rows and columns are both matrices, MPL
"considers this a redefinition of structure.  Note also
"a matrix and a normal vector cannot be multiplied at all."
V := VECTOR(R)t2*VECTOR(C1);  "V is now {3,6,...,30}"
DEFINE bland = {V(4),R(4),C1(4),X(4,4)};
"This last statement (above), with typical bland disregard
```

†Although 'ROW', 'COLUMN', and 'VECTOR' are normally keywords, their use here is distinct from their keyword interpretations, so we shall not print them in bold-face.

"for the actual structures involved, pulls out one element
"each from a vector, a row, a column, and a matrix, demon-
"strating how one subscripts each structure. The resulting
"4-element vector happens to be (12,4,4,16)."

2: EXPRESSIONS (REVISITED)

Now the fun really begins. We're about to laden you with all sorts of additional operators and related hocus-pocus. Some of these unfortunately do not have a standard mathematical notation: some of them have standard notations which do not readily adapt themselves to keypunching, Thus we've had to concoct our own notations. Where possible we have used 'standard' cohiputer-language forms, Oh well, enough excuses. Let's get on with it.

2.1: More Operators

2.1.1: Exponentiation

Denoted by the Z-character symbol '**', exponentiation is allowed as a binary operation between scalars only. Thus exponentation may not be used to find powers of a matrix, nor to square every element of an array.

The result is real if either operand is real: otherwise it is integer. If the second operand (the exponent) is real then the first operand (the base) must be positive.

Examples:

5**3	yields	125
2.5**(-2)	yields	0.16
.7**.9	yields	0.725418 (approximately)
(-1)**k	yields	-1 if k is odd +1 if k is even an error if k is not an integer
(-3)**3	yields	-27
3**(-3)	yields	0 ($3^{-3} = (1/27) = 0$)
(-3.0)**(-3)	yields	-0.037037 (approximately)
(-3)**(-3.0)	yields	an error (first operand (-3) not positive)
3**(-3.0)	yields	0.037037

2.1.2: Horizontal and vertical concatenation

Denoted respectively by '|' and '#', horizontal and vertical concatenation are allowed as binary operations on vectors, rows, columns, and matrices. If two things are to be concatenated, they must be the same type (e.g., character), with the exception that one may be integer and the

other real. (In this case the integer operand is treated as if it were real.) Moreover, the row sizes (number of rows) must be equal for horizontal concatenation, and the column sizes must be equal for vertical concatenation.

The result of a concatenation operation is always a matrix, Even concatenating two rows yields a 1 by n matrix, not a row. (Mathematically, of course, there is no difference, but MPL thinks of the two structures slightly differently.) One except ion: horizontal concatenation of two vectors yields another vector. Refer to Appendix C for complete information concerning the structure of the result of a concatenation operation.

Examp l es:

```
DEFINE V := {1,...,4}, V' := V*0.5,
     R5 ROW 5, R7 ROW 7,
     C5 COLUMN 5, C7 COLUMN 7,
     tl MATRIX 5 BY 7:

(V-1) | (V+1)    yields  {0,1,2,3,2,3,4,5}
v | V'           yields  {1.,2.,3.,4.,0.5,1.,1.5,2.}
v # V'           yields  a 2 by 4 real matrix
V | <<frog>>     yields  an error (incompatible types)
R5 | R7          yields  a 1 by 12 matrix
R5 # R5          yields  a 2 by 5 matrix
R5 # R7          yields  an error (different column sizes)
C5 | M           yields  a 5 by 8 matrix
M | M            yields  a 5 by 14 matrix
M # M            yields  a 10 by 7 matrix
C7|(R7#M#R7)|C7 yields  a 7 by 9 matrix
```

2.1.3: Reiational comparisons

There's nothing new about relational operators ('=', '¬=', etc.), but we want to point out that what they really do is yield a scalar logical result. These logical results may then be combined using logical operators, which we shall discuss next. Incidentally, you might note that, if 'p' is a logical scalar, then "p = TRUE" is equivalent to "p".

2.1.4: Logical operations

Logical conjunction, disjunction, and negation (commonly known as 'and', 'or', and 'not') are represented respectively by the keywords AND, OR, and NOT. (The character '¬', as used in certain other languages, is allowed in place of 'NOT'.) These operations are defined only on logical values.

AND and OR are allowed as binary operations between logical scalars or between hiatrices (*not* vectors!) of equal sizes. Matrix operations are done coniponen t-w i se. The operations are def inecl in the usual way, i.e., if P is the logical vector {TRUE, FALSE, TRUE, FALSE} and Q is {TRUE, TRUE, FALSE, FALSE} then (if vector operations were allowed, which they're not) we would have:

P AND Q yields {TRUE, FALSE, FALSE, FALSE}
P OR Q yields {TRUE, TRUE, TRUE, FALSE}

NOT is allowed as a unary operator on scalars and non-scalars. It is defined in the usual sense: NOT TRUE yields FALSE and NOT FALSE yields TRUE.

2.1.5: Membership

The operators IN and NOT IN are permitted only between a scalar and a vector, and are used to test whether the scalar occurs as a member of the vector. The operators each yield a scalar logical result. 'x IN V' is TRUE if and only if x∈V; 'x NOT IN V' is equivalent to NOT (x IN V).

The operands must be integer or real. MPL may soon be extended to allow character operands, but this is not currently the case,

The right-hand operand is permitted to be a null vector, in which case the result is FALSE (for IN) or TRUE (for NOT IN).

Examples:

5 IN {1,3,...,10}	yields TRUE
5 IN {1,4,...,10}	yields FALSE
7 NOT IN {1,4,...,10}	yields FALSE
<<X>> NOT IN <<Benedict>>	yields TRUE (not yet implemented)
{3,5} IN {1,3,...,10}	yields an error ({3,5} isn't scalar)
1 IN {1,...,0}	yields FALSE

2.1.6: Precedence

The precedences among operators mentioned so far are:

First: subscripting, 'functions' (ROW, etc.)
+ - (unary)
# |
**
* /
+ - (binary)
relationale, IN, NOT IN
NOT
AND
Last: OR

Examples should don't be necessary here, but two items are worth noting. First, whereas normal mathematical notation interprets a^{b^c} as $a^{(b^c)}$, MPL's precedence rules (left to right among binary operators of equal precedence) will cause it to interpret a**b**c as $(a^b)^c$. use a**(b**c) to get the first interpretation. In fact, use (a**b)**c for the second interpretation, just to save yourself some confusion. Second, although -1**k will yield the same results as (-1)**k, we recommend you get into the habit of including the parentheses anyway, since some languages would treat the unparenthesized form as -(1**k).

2.2: Subscripting

In the course of learning to use MPL, have you ever uanted to specify a single row of a matrix? Al I but the first element of a vector? The reversal of a vector? Did you go crazy trying to do these things by subscripting one element at a time? We hope not, because i f you needed these things it was probably time to go on to Section II. Well, here you are in Section II, and here's the secret: You can use vectors as subscripts, For instance, (1,3,5)((1,3)) yields (1,5), i.e. the 1st and 3rd elements of (1,3,5).

The first thing you might notice is that, whereas the result of subscripting has hitherto been a scalar value, the resul t above was a vec tor. The general rule is as follows: The result of a subscripting operation has one level of dimensionality for each vector subscript, Thus, i f A is an array (and is at least 2 by 2 by 2) then

AU, 2, 2)	yields a scalar
A((1,2), 2 , 2)	yields a 2-element vector
A((1,2), 2 , (1,2))	yields a 2 by 2 matrix
A((1,2), (2), (1,2))	yields a 2 by 1 by 2 array

A row or column subscripted by a vector yields a new row or column with the same number of elements as the subscript.

Note: When a matrix is subscripted by a vector in one dimension and a scalar in the other, MPL treats this as a special case. Instead of yielding a l-dimensional vector result, as the above examples would imply, MPL produces a row or column vector, depending upon which subscript was the vector. This was generally found to be more useful than a l-dimensional result. If you uant a vector result, use the 'VECTOR(...)' construction described in section 1.2.3.

Further note: MPL wi l l currently produce a run-time error if you use a vector to subscript an array. This oversight should be corrected in the near future.

You may al so use, in place of a numeric subscript, a single asterisk ('*'). This has the same effect as if you had used a subscript of '(1,...,n)' where n is the size of the corresponding dimension. Thus, if M is a 5 by 7 matrix, then M(3,*) is the 3rd row of the matrix, and by the rule just given it would be a T-element row vector. M(*,2) would be the 2nd column of the matrix, Such subscripts are sometimes referred to as "cross-sect ions".

Reaching into the future again (section 4.5), we wi l l tel l you now that if you need to know the size of a vector V, you may write 'SIZE (VI'. Similarly, to find the number of rows or columns in a matrix M, use 'ROWSIZE (MI' or 'COLSIZE (M)', respectively. We mention these now because (a) they're handy to have around and (b) we're going to use them in the upcoming examples, and it uould be nice if you knew what we were doing.

Examp l es:

```
DEFINE V :=(2,5,...,50);
WRITE V((3,9,3,15));  'Wri tes 8, 26, 8, and 44"
DEFINE Vrev := V((SIZE(V),SIZE(V)-1,...,1));
"Vrev is now the reverse of V, i.e. (50,47,...,2)."
DEFINE V := V((10,...,SIZE(V)) | (1,...,9));
"V has been 'rotated'.  It is nou (29,32,...,47,50,2,5,...,26)."
DEFINE V :=V((2,...,SIZE(V)));
"The first element of V has been 'discarded'."
DEFINE R= ROW(V), C = COLUMN (VI,
     M  = C((14,2)) * R((7,...,9));
"M is a 2 by 3 matrix which looks like: 1000 40 100
"                                       1750  70  175
M := M(*, (COLSIZE(M),COLSIZE(M)-1,...,1));
"This reverses each row of M. M is now: 100 40 1000
"                                        175 70 1750

"As a final example, the following code would create a matrix
"P' which is the transpose of a real matrix P. There is an
"easier way of doing this, but we'll talk about that later.
'The idea here is to give you another look at how cross-sec-
"tions work."
DEFINE, P' MATRIX COLSIZE (P) BY ROWSIZE(P);
FOR i :=(1,...,ROWSIZE(P)), "For each row of P,
     P'(*,i) := P(i,*);          "copy it into a column of P'"
```

Vector subscripts must in general be non-null, that is they must have at least one element. In the special case of a vector subscripted by another vector the vector subscript may be null, in which case the result is also a null vector.

2.3: Set Generators

There is a third type of vector generator to go with N-tuples and index sets. This third type is called a "set generator", The term is unfortunately a bit misleading, since in a sense all vector generators in MPL are set generators. Since set generators are effectively FOR loops put inside vectors, perhaps a better term would be 'FOR vectors', but that's what we cal l the vector inside a FOR statement' 'Vector mappings', perhaps? Oh wel l, we'll call them set generators, and maybe after we've described them you'll be able to think of an appropriate name for them.

The general form of a set generator is

(FOR <for index> IN <integer vector>, <expression>)

w h e r e '=' o r ':=' may rep l ace 'IN' and 'DO' may replace the comma. The <for index> and <integer vector> are the same as for normal FOR statements, namely any variable and any integer vector. As with FOR loops, the index variable need not have been defined, and if it has it will be temporarily redefined within the set generator.

The value of a set generator is the vector whose elements are the <expression> evaluated for each value of the index variable, in sequence. The expression must yield a scalar result. Let us make this clear with some

Examples:

```
{FOR i IN {1,...,5}, i*i}
        yields {1,4,9,16,25}
{FOR i IN {1,9,100,1024}, i**0.5}
        yields {1.,3.,10.,32.}
{FOR i = {FOR j = {5,4,...,1} DO j*j}, i*i/1E3}
        yields {.625,.256,.081,.016,.001}
{FOR j := {1,3,...,10}, {1,...,j}*{1,...,j}}
        yields {1,14,55,140,285}
{FOR frog IN {2,...,4}, 11,...,frog}}
        yields an error (<expression> not scalar)
            not {1,2,1,2,3,1,2,3,4}
```

In case this isn't enough to give you what you want, there's even a more powerful form of set generator, which looks like

(FOR <for_index> IN <integer vector>: <condition>, <expression>)

where '=', ':=', and 'DO' may be substituted as before. Whereas the first form puts a FOR loop inside a vector, this form conditions the FOR loop with an IF statement. The <condition> is any logical expression, and only those values of the index variable which cause the condition to be true will be used in creating the new vector. This probably needs more explanation, so let's try to clear it up with a few more

Examples:

```
{FOR i IN 11,...,5}: i - 2, i*i}
        yields {1,9,16,25}
{FOR j IN {1,...,10}: j<=4 OR j>=8, (j/2)*2}
        yields {0,2,2,4,8,8,10}
{FOR j = {1,..., 4} | {8,...,10}, (j/2)*2}   (horizontal
        concatenation instead of a condition)
        also yields  {0,2,2,4,8,8,10}
{FOR i := {1,...,15}: i*< 7 OR i-(i/3)*3 ¬= 1, i}
        yields {1,2,3,4,5,6,8,9,11,12,14,15}
```

3: MORE STATEMENTS

We shall now introduce some neu statements, as well as some new twists to old statements.

3.1: Subscripted Assignment

Since subscripts have become a little more complicated, so has subscripted assignment (i.e., assignment into a subscripted component of a non-scalar variable). If you recall, back in Section 1 we introduced something called a <left side> which could be assigned values in an assignment or READ statement. At that time we said a <left side> had to be either a variable or a subscripted variable. With the broader scope of subscripts now at your disposal, we must be a bit more specific.

A **<left side>** is either an unsubscripted variable (possibly non-scalar) or a variable subscripted using scalars, asterisks, and/or index sets. This does *not* permit the use of N-tuples, set generators, or vector expressions as subscripts. Such subscripts are valid in an expression, but not within a <left side>. Recall that subscripts of any type are forbidden on the left side of a defining assignment statement.

Examples:

```
DEFINE M MATRIX 3 BY 5,
       V INTEGER VECTOR 3;

V := {1,3,...,5};
M := COLUMN (VI * ROW ({1,...,5});
"M is now 1. 2. 3. 4. 5.
"  3.     6.     9.     12. 15.
"         5. 10, 15. 20. 25."
...,.," /;
M(*,5)M(24);=:=79;
"M is 3. now 6.1. 2 9. 3. 4 7: 8 9:
"          5. 10. 15, 20. 9."
M(2,V) := {11., 9.5, .6};  "Illegal left side (even though
                            "V's value is an index set)"
M(2,{1,3,5}) := {11., 9 . 5 , .6};  "N-tuple illegal in left side"
M(2,{1,3,...,5}) := {11., 9.5, .6};  "Index set is legal"
"M is now 1.  2.   3.   4.   9.
"   11.     6.   9.5  7:   0.6
M(1{12,...,3}) 5. 10. 15.  20.   9. "
   , .,          := M(3,V);
"V is a legal subscript on the right side, though illegal on
"the left. M is now 5.  15.  9.  4.  9.
"                   11.  6.  9.5 7.  0.6
"          5          .  10.  15,  20. 9.  "

"Now for a feu somewhat complicated maneuvers."
M(11,...,2),{2,5,...5}) := M({2,3},4)* MU,{4,2});
"Since M({2,3},4) is a column vector, and M(1,{4,2}) is a
"row vector, the right-hand side is evaluated as the
"matrix product of a 2 by 1 matrix with a 1 by 2 matrix,
"yielding a 2 by 2 matrix which is then assigned into the
"2 by 2 submatrix of M specified on the left. M is now
"   5.   28.   3.    4 105.
"  11.   80.   3.5 7: 300:
"   5.   10.   15.   20.   9."
```

```
M(1,{5,2,...,2}) := VECTOR((M(*,{1,2,4}))({1,2},3));
"The right side subscripts M to get a 3 by 3 submatrix, then
"subscripts that to get a 2-element column vector. This is
'then converted to a normal vector by the VECTOR operation.
'Meanwhile, the left-hand side subscripts a row vector in M.
"This row vector is assigned the 2-element vector on the
"right. . M is now 5. 7. 9. 4. 4.
"                  11: 80. 9:; 7: 300:
"                   5. 10. 15. 20. 9."
M := M(*,1) * M(1,*);
"M(*,1) is a column: M(1,*) is a row. Thus M is assigned
"a new 3 by 5 matrix, M is nou 25. 35. 45. 20, 20.
"                              55. 77. 99. 44. 44.
"                              25. 35. 45. 20. 20."

"It is possible to use a vector subscript other than a con-
"stant in a left side. for our final demonstration, we shall
"illustrate such a situation,"
FOR k := {1,...,3},
    M(k,{1,...,2*k-1}) := 0;
"M is now 0. 35. 45. 20. 20.
"          0. 0, 0. 44. 44.
"          0. 0. 0. 0. 0.'
```

Even though it is quite likely that you yourself ui I I never do anything quite so bizarre as some of the preceding examples, or perhaps because of this, we urge you to work through them unti I you are confident you understand what's going on,

3.2: IF Statement

We shouldn't have to reiterate what an IF statement does or what it looks I ike. What we want to say here is that the <condition> is not restricted to being a relational operation, The <condition> may in fact be any expression which yields a scalar logical result.

Examp I es:

```
"(1) If the scalar i is not yet in V but i<100, append i to V."
    IF i<100 AND i NOT IN V,
        DEFINE V := V | {i};

"(2) Given an integer vector S, create an integer vector V
    'consisting of all positive elements of S, with duplicate
    "elements removed."
    "Since null vectors are not allowed in an assignment, we
    'will put one element at the front of V and then remove
    "it when we are done."
    DEFINE V:={1};
    FOR i IN S,
        IF i>0 AND i NOT IN V,
            DEFINE V := V | {i};
    "We now remove the 'dummy' element from the front of V.
```

```
"This will give an error if S had no positive elements."
DEFINE V := V((2,...,SIZE(V)));

"(3) Verify that 3 vectors, V1, V2, and V3, are mutually orthogonal,
"If they are not, print an error message and stop."
IF V1*V2 ¬= 0 OR V1mV3 ¬= 0 OR V2*V3 ¬= 0,
|_  WRITE <<Vectors not orthogonal,>>;
    STOP
_|;

"(4) Same as (3) above, but apply DeMorgan's law. Which form is
"more readable is a matter of personal preference."
IF NOT (V1*V2 = 0 AND V1*V3 = 0 AND V2*V3 = 0),
|_  WRITE <<Vectors still not orthogonal.>>;
    STOP
_|;
```

Warning! Some languages specify that the left-hand argument of 'AND' is evaluated first, and if the value is false then the right-hand argument is ignored, since the result of the 'AND' is already known to be false. (Similar specifications apply to 'OR'.) *MPL does not do this.* Thus, in NPL, the statement --.

```
IF k>0 AND V(k)=q, etc.
```

will cause trouble when k is non-positive: you will get an error for trying to subscript V(k). You must break up the above statement, as in

```
IF k>0 THEN IF V(k)=q, etc.
```

3.3: CASE Statement

The CASE statement is an extension of the IF statement, An IF statement allows you to execute either one of two statements based on some condition. A CASE statement allows you to execute any one of several statements based on some condition. Here the condition is an integer value from 1 to n, where n is the number of statements among which you are selecting, If the 'condition' has the value k, then the k^{th} of the n statements is selected to be executed. The general form is

```
CASE <scalar integer expression> OF BEGIN <statement 1>;
        <statement 2>; . . . ; <statement n> END
```

where as usual '|_' and '_|' may replace the keywords 'BEGIN' and 'END'. Each <statement i> must be a single statement, but may be a compound statement. If the <scalar integer expression> does not yield a value between 1 and n when the CASE statement is executed then an error will be reported. If we call the expression 'x' the above general form is exactly equivalent to the statement

```
IF x-1 THEN <statement 1>
ELSE IF x-2 THEN <statement 2>
ELSE IF x-3 THEN <statement 3>
     .
     .
ELSE IF x=n THEN <statement n>
ELSE <error>
```

and thus, not surprisingly, it is when we uish to do the above sort of thing that it is a good idea to use a CASE statement.

Example:

```
"Suppose you've just finished executing a program for solving
"a linear programming problem.  The program creates a scalar,
"called 'flag', which is 1 if the problem was infeasible, 2
"if it was unbounded, and 3 if there was an optimal solution,
"in which case 'Xbar' is a vector containing the solution.
"Here is a likely piece of code to do next."
CASE flag OF
|_   WRITE <<Infeasible.>>;
     WRITE <<Unbounded. >>;
     |_   WRITE <<Optimal solution:>>; "Use 2 WRITEs to get
          WRITE Xbar                  "separate output lines"
     _|
_|;
```

Another typical case, ah, that is, situation, where you would probably use a CASE statement, is when you want to run the same program several times uith certain small variations, and you don' t want to have to change the program each time, For instance, suppose you're uri t ing a program to approximate the area under a curve using numerical integration. You're interested in comparing various approximation formulae, say rectangular, trapezoidal, and Simpson's rule. You simply put together a 3-way CASE statement to select among them, and then by putting your whole program (excluding the surrounding PROGRAM and END lines) inside an appropriate FOR l oop for {1,...,3}, you can test al l 3 formulae with one program, run once.

As an aside, we should mention that i t is good programming sense, when you have a program surrounded by a FOR loop like this, to test the program for one of the simpler c a s e s first. In the preceding example, the overall program looks something like this:

```
PROGRAM
FOR case-flag = {1,...,3},
|-

     "Here there is some relatively large chunk of code to do
     "the desired numerical integration a n d print the results.
     "Somewhere in it is a statement of the form:"
     "      CASE case_flag OF
     "      |-     X := (put rectangular formula here);
     "             X := (put trapezoidal formula here);
     "             X := (put Simpson's rule formula here)
     "      _|;
     "Toward the end there is probably another section which
```

```
            " looks like:
            "       WRITE <<Integration formula used:>>;
            "     . CASE case_flag OF
            "       |_      WRITE <<Rectangular>>;
            "               WRITE <<Trapezoidal>>;
            "               WRITE <<Simpson's>>
            "       _|;

            "Eventually it's all done, and we close out the FOR loop."
        _|
        END.
```

Suppose you have just written this program and are about to run it on the computer for the first time. It is the height of arrogance to presume it will work the first time. Assuming it doesn't, you will have wasted 3 times as much computer time (and money) as necessary, since if you had replaced the FOR loop with

```
            FOR case_flag = {1},    "If it runs okay, delete this statement
                                    "and remove quotes from the one below"
        " FOR case_flag = {1,...,3},
```

it probably would have been sufficient to demonstrate whatever problems might exist. Not necessarily--there might be an error in the computation of one of the other formulae, for example. But until you have reason to expect that the program might work next time, it doesn't make sense to gamble at triple-or-nothing. And there might be more than a 3-to-1 risk involved. Suppose that, for each formula, you also want to try each of 3 functions to be integrated, and over each of 5 ranges of integration. It would be inexcusable to run the program for all 45 cases before the bulk of the program had been tested on one or two samples. Using quoted reminders and keeping the alternative statement in the program as a comment (as shown) makes it very easy to make the necessary modifications once the program is working correctly.

3.4: **FOR** Statement

When we introduced set generators in section 2.3 we described two slightly different forms. The first put a FOR loop inside a vector; the second put an IF inside the FOR loop. It may not come as much of a surprise then when we tell you you can do the same thing with FOR loops outside of vectors. The syntax is the same; follow the FOR-vector with '**:<condition>**', and the loop will be executed only for those index values which satisfy the condition. This is all directly analogous to set generators, so we shall not go over it any further here, The examples show an 'old-style' FOR loop and a very similar conditional one.

Examples:

"**(1)** Find the product of the elements of an integer vector **V.**"

```
DEFINE product : =1;
FOR i IN V,
    product := product *i;
```

"**(2)** Find the product of the ***non-zero*** elements of the same vector."

```
DEFINE product : =1;
FOR I IN V:i¬=0,
    product := product * i;
```

4: PROCEDURES

Many times it behooves us to perform the same sequence of code several times, perhaps wi th slight variations, perhaps in different sections of the program. Loops (FOR and WHILE statements) do not provide all the power we need for this since, al though they al low us to repeat a section of code, they require all the repetitions to occur consecutively--nothing else can be interjected be tween them.

In the MPL sample way back on page 2 we needed to compute the 'Frobenius norms' of 3 matrices. We could have done this with the following loop:

```
FOR index IN {1,...,3}, "Compute 3 Frobenius norms"
|-    CASE index OF "Decide which matrix to work with this time"
     |_    DEFINE M = A;
           DEFINE M = B;
           DEFINE M = A*B
     _|;
     DEFINE sum := 0.0; "Initialize sum to (real) zero"
     FOR i IN {1,...,ROWSIZE(M)},
           FOR j IN {1,...,COLSIZE(M)},
                 sum := sum + M(i,j)*M(i,j); "Compute sum of squares"
     DEFINE F := sum ** 0.5; "Take square root--F is now Frobenius norm of M"
     CASE index OF "Decide where to save result"
     |-    DEFINE Norm_A = F;
           DEFINE Norm_B = F;
          ~DEFINE Norm_AB = F
     _|
-|
```

We cou ld then have used the 3 var iab les Norm_A, Norm_B, and Norm_AB whenever we needed them l a ter i n the program. This would work, but it is somewhat clumsy, and there are cases where this construct ion is not possible. For example, suppose we need to compute the sine function several times throughout a program, and we don't know ahead of time what al l of the arguments are going to be. The loop structure is clearly inadequate for this, yet we don't want to write into our code a separate copy of the sine routine at each place we need to use it. 'Ah!' you say, 'What about using GO TOs?'. Very well, we write one section of code which computes the sine of the value in 'X' and stores the result in the variable 'sin'. Then whenever we need to compute a sine we simply set X to the value of the argument and GO TO to the sine routine. It then computes the sine and, er, . . . how does i t get back to where we were? Well, we could set up some other variable 'sin_index' to a unique integer before going to the sine routine, and then the sine routine could end wi th a CASE statement doing one of several GO TOs to get back, but this is getting clumsy again.

What we want is a totally new program structure, something which allows us to say, in effect, 'Do that section of code over there, and then come back.' Many languages, including MPL, allow this to happen even in the middle of an expression, in which case the 'other code' can specify a value which is to be 'plugged in' as part of the expression. For instance, turning again to the sample on page 2, the IF-condition in

```
IF Norm (A*B) > Norm (A) * Norm (B)
```

causes us to transfer 3 times to the 'Norm' routine. The values computed b y the Norm function are then plugged in in place of the 'Norm (...)' for each respective case. We'll give more detailed examples once we've described things more thoroughly. Right now, let's just wrap up this fine-print section.

These 'subordinate' sections of code are called various things in different computer languages. Most common are the terms 'subroutine', 'function', and 'procedure'. They must be set off from the rest of the program in certain ways, and they may be used only in certain ways. So now it's time to tell you and our home audience exactly how our game is played.

Subroutines, functions, procedures--whatever you're used to calling them, in MPL we call them all 'procedures", and identify them by the keyword PROCEDURE, There is also a special type of procedure, identified by the keyword FUNCTION. Functions differ only subtly from procedures, so we won't bother discussing them until Section III.

There are only two significant variations among procedures--they may or may not have 'arguments', and they may or may not yield a result. (To say that a procedure has no result does not imply that it accomplishes nothing to use it. We'll get to this in a little while.) Let's start with the simplest case.

4.1: No Arguments, No Result

A procedure without arguments and which returns no result is written as follows:

PROCEDURE <name>: <statement>

where <name> is the name of the procedure and may be any legal identifier. The <statement> is the procedure body and may be any single statement. Usually the body is a compound statement, but this is not required. When the above construction is inserted into a program (probably followed by a semicolon to separate it from whatever comes next) it 'defines' the procedure and gives it the name specified. The procedure may then be invoked anywhere further on in the program by using its name in any of the three statements

CALL <name>

or

EXECUTE <name>

or simply

<name>

(The third form is just the name of the procedure written alone.) We shall be using the third form in most of our examples, but in order to be able to talk about these statements (the way we can talk about a FOR loop or a WRITE statement) we'll refer to them as CALL statements.

When the procedure is invoked, it causes the <statement> to be performed, after which the program continues execution in normal sequence following the CALL statement. The effect is to take the <statement>, which may be quite complex, and 'abbreviate' it into a single simple statement, namely the CALL.

The first two examples show typical no-argument-no-result procedures.

The third is a somewhat more contrived case, and shows a complete program using a procedure.

Examples:

"(1) Suppose you have a set of variables--foo, mumble, and "toad--which you wish to simultaneously reset to zero at "various places in the program."

```
PROCEDURE reset;
|_  foo := 0;
    mumble := 0;
    toad := 0
_|;
```

"Note that the above procedure implies you are able to "change, in a procedure, the values of variables outside "of that procedure. Thus you must be careful that you do "not use the same names for variables in the procedure as "for variables elsewhere in the program (unless you do it "with the deliberate intention of changing those variables* "values as in the example), lest the procedure destroy a "needed value. FOR indices within a procedure are, of "course, not subject to this restriction, since they are "automatically redefined and then restored after the loop.'

'On the other hand, any variable which is defined inside "a procedure body becomes 'undefined' upon returning from 'the procedure (in exactly the same fashion as a FOR index 'becomes undefined after the FOR loop), regaining whatever "attributes and values it had (if any) before the procedure "was called."

"(2) Several times throughout a program we may wish to perform "various tests to see if there is an error. E.g., is the 'matrix we are inverting singular, or is the scalar whose 'square root we need negative? If anything is ever found "to be wrong, whatever the problem happens to be, we wish "to print a standard error message, write out some pertinent variables, and stop. So we write a procedure, and "call it quits."

```
PROCEDURE quits;
|_  WRITE <<Error in phase>>, phase:
    WRITE <<Pertinent (and impertinent) data:>>;
    WRITE <<frog=>>, frog, <<glork=>>, glork;
    WRITE <<M=>>,M;
    STOP
_|;
```

"Notes:(a) The variables phase, frog, glork, and M must "all have been defined prior to the above code. This re- "quirement is *physical*; the definitions must *appear* ahead "of the procedure. It is not sufficient to define them "later, even though it may be prior to the use of the "procedure. The reason is that the MPL compiler is not "executing your program, but is reading it sequentially, 'and when it sees the above chunk of code it has to know "what types and structures to expect. (b) The above pro- "cedure, if executed, would not return to the section of

```
"code which invoked it, due to the STOP statement.  This
"is legal,"

"(3) Create a real vector V of size n such that V(i)= the
"first element of {(n-i)^i, i^n-n^i, (n+i)^0.75} which is ≥n.
"If none of the three is ≥n, then V(i)=0.  (For n=5, V
"would be {0., 9., 8., 399., 5.6234}.)  This could be
"done in various ways, but let's see one way to do it
"using a procedure."

PROGRAM

GIVEN n INTEGER;
DEFINE (V,V') VECTORS n;

v := 0.;

PROCEDURE overlay;  "1-statement procedure, so no |_ _| marks"
     FOR i IN {1,...,n},              "V, V', and n are
          IF V(i)=0 AND V'(i)>=n,     "already defined,
               V(i) := V'(i);         "as required"

V' := ~{FOR i IN {1,...,n}, (n-i)**i};
overlay:
V' := {FOR i IN {1,...,n}, ittn - n**i};
overlay:
V' := {FOR i IN {1,..., n}, (n+i) ** 0.75};
overlay:
WRITE <<n=>>, n, <<V=>>, V

END.
```

Note: Although procedures are most often used as a means of abbreviating a piece of code which would otherwise have to be duplicated several times throughout a program, it is often wise, especially in large programs, to have procedures which are invoked in only one place. The latter's purpose is to break up a program into logical components so as to make it more understandable, For instance, if you were writing a program to set up and solve a system of simultaneous equations, you might consider writing three procedures: one to set up the equations, one to solve them, and one to print the results. Most people find it easier to write and understand these smaller, separate routines, than to try merging them into a single long program. The program would now look something like this.

```
PROGRAM

PROCEDURE Set up:
|_   etc.
     etc.
_|;

PROCEDURE Solve;
|_   etc.
     etc.
```

```
_|;

PROCEDURE Print;
|_   etc.
     etc.
_|;

Setup:
Solve:
Print

END.
```

This also makes it easier to modify the program, since if you decide to change, say, the 'Solve' routine, it is much less likely to require any changes elsewhere in the program.

4.2: Arguments (Parameters)

The procedure: discussed in the preceding section were more or less autonomous; the only way to pass data into or out of them was using a 'shared' variable, such as V' in the last example. There is a more compact notation for passing data into a procedure. This is done in the form of one or more arguments (also called parameters). Rather than give the general form, which is beginning to look overly complicated, we'll use an example. Don't worry about the keyword VALUE: we'll explain that shortly.

Example:

```
PROCEDURE Swap (V,i,j)
     WHERE V IS VECTOR, (i,j) ARE SCALAR INTEGER VALUES:
"This procedure swaps V(i) with V(j)."
|_   DEFINE swap-temp := V(i);
     V(i) := V(j);
     V(j) := swap-temp
_|
```

In the above example, the procedure has 3 arguments, V, i, and j. The argument list must be enclosed in parentheses as shown. The type and structure of each argument must be specified after the keyword WHERE, unless the default attributes (REAL SCALAR) are desired. Note, however, that the size of V is not specified. Thus V may be different sizes at different invocations of Swap, but V must always be a real vector (in this particular example, that is.) The keywords IS and ARE are interchangeable and may in fact be omitted entirely if you feel they get in the way. **Note where the semicolon goes, after the WHERE portion. (The other semicolons are used, as usual, to separate the statements within the procedure body.) As before, another semicolon will normally be required after the '_|' in order to separate the procedure from whatever statement follows it.**

A typical invocation of the above procedure might be

```
Swap (X, 1, SIZE (X))
```

which would swap the first and last elements of the vector X. (If X is not a real vector an error will result.) Note that, using this construction, the variable X need not be defined before defining the procedure. This was required in the previous examples so that the compiler would know, when it read the procedure, what the types and structures of the variables were. Here, however, the procedure includes the information that V is going to be a real vector, which is all the compiler needs to know at this point. Of course, X must be defined before it can be used as an argument to the procedure.

4.2.1: The VALUE attribute

Now, as to the VALUE attribute. This has to do with 'parameter-passing conventions', which is a real can of worms, and which we don't intend to cover in detail until Section III. Unfortunately, if you're going to use procedures with arguments, you have to know about the 'VALUE' attribute. Let's see if we can explain what's happening, We'll warn at the outset that you probably won't understand all of this. If you get completely lost, try reading section 4.2.2 instead.

We'll start by mentioning that everything we're about to say applies only to scalar arguments. Non-scalar arguments are different and we'll explain them in a moment. For scalar arguments, the default (i.e., what happens if you don't say VALUE) is to treat them as follows. Suppose you call 'Swap' using

```
Swap (X, p, q)
```

to swap X(p) with X(q). What happens is that the values of p and q are copied, i.e. are assigned to the variables i and j of the procedure. (We should note that, like the index of a FOR loop, the arguments inside a procedure (in this case, V, i, and j) are 'local' to the procedure: they are temporarily redefined within the procedure, But that's another story.) Now we get to the situation which will eventually cause us trouble. Suppose our procedure were written differently, such that it changes the values of i and j? Well, since we made copies of p and q, the values of p and q are not affected during execution of the procedure. *But*, when you leave the procedure, the new values of i and j are copied back into p and q. Let's see if we can come up with an example demonstrating all of this.

Example;

```
DEFINE frog=7,toad=2.5;

PROCEDURE change (toad) WHERE toad IS INTEGER;
"Because 'toad' is the argument of a procedure, it is legal
"to 'redefine' it as an integer. The argument 'toad' bears
"no relation to the real variable 'toad' which was defined
"outside the procedure. On the other hand 'frog', which
"also appears in the procedure body below, is not defined
'within the procedure and thus is the same variable as that
"defined above. "
"We shall number the WRITE statements for future reference."
|_   WRITE frog, toad:   "1"
     toad := 9;
```

```
        WRITE frog, toad;     "2"
_1;

WRITE frog, toad:             "3"
CALL change (frog);
WRITE frog, toad              "4"
```

What does all this do? We first define two scalars: an integer (frog) and a real (toad) and assign values to them. Next we define a procedure named 'change': we don't actually execute it yet, so nothing is changed, We then reach statement "3" and print two numbers, 7 and 2.5. Now we call 'change'. 'frog' has the value 7, so this value is copied and the new variable 'toad' (distinct from the one just printed) is given the value 7. We do statement "1", printing 7 and 7. Now 'toad' is set to 9, but 'frog' is unchanged. Statement "2" prints 7 and 9. We finish executing 'change', whereupon the new value of 'toad' (9) is assigned back into 'frog'. The argument 'toad' disappears, leaving us with the old, real 'toad'. Statement "4" prints 3 and 2.5.

Now, if you claim to understand all of that then you're either a genius or a liar. This sort of thing is nearly impossible to explain even in an interactive classroom environment. So let us stress one particular point: The value of 'frog' was altered by the call to 'change'. So, consider what would happen if we said instead:

```
CALL change (3)
```

When we finished executing 'change', we would try to take the new value of 'toad', namely 9, and assign it to the constant 3! What happens if we now try to use the constant 3 somewhere else? Do we get 3, or 9? Something must be done to prevent this sort of shenanigans, and here's what we did.

> When you call a procedure, the values supplied (called the 'actual parameters') for all scalar arguments must be legal <left side>s, as defined for assignment statements.

Well, that certainly takes care of the problem, You can't use '3' as an 'actual parameter', so the problem can never come up. But wait a minute! This is too restrictive! Besides, the first example we gave used '1' and 'SIZE(X)' as arguments, and neither of those is a legal <left side>. This is where VALUE comes in, If an argument within a procedure (called a formal parameter to distinguish it from the 'actual parameters' supplied when the procedure is invoked) is defined in the WHERE clause using the special attribute VALUE, then the procedure is free to do anything at all to the formal parameter, and it will not affect the actual parameter. That's what VALUE does for you. Moreover

> If a formal parameter is given the VALUE attribute, the corresponding actual parameter may be any expression of the appropriate type (integer, real, etc.).

Note: It makes no difference whether or not the procedure actually does

assign new values to its arguments. If you want to be able to use arbitrary expressions as arguments you must use the VALUE attribute, as we did in 'Swap'.

As an aside, we should probably explain just how on earth it is even conceivable that a constant could have its value changed. After all, the very idea that a constant is, well, inconstant, must come as quite a surprise to most mathematicians. The reason is this: the computer does not actually know what we mean by a "constant". It deals solely with "numbers" (even MPL character values are represented internally by numbers), and by the construction of the machine all numbers in its "memory", its working space, are subject to change. Thus, when you write "7.3*X", the MPL compiler tells the computer to pick up the current value of X, then to multiply it by 7.3. But there is no instruction which tells the computer to multiply by 7.3. (There are several dozen simple instructions which the computer is capable of performing directly (the purpose of the MPL compiler is to rephrase your program in terms of these very simple instructions), but there is no way there could be a separate instruction for multiplying by every conceivable real constant!) So, when MPL wants the computer to multiply by 7.3, it creates the value 7.3 and stores it somewhere in the computer's memory. It can then tell the computer "multiply by that number over there", which is one of the 'simple instructions' available. So that's why constants have to be stored in the computer's memory. This being the case, MPL keeps track of which constants you use. If you use "7.3" several times, MPL notices this and creates only one copy of the constant, which is then used everywhere you need the constant 7.3. Thus, if you somehow change the value stored at that location in memory, you will 'change' the value of the constant 7.3 everywhere it is used.

Finally, remember that all of this applies only to scalar arguments. Because non-scalars can take up a lot of room, MPL does not make copies of non-scalar variables when they are used as arguments. Thus, in our original example (Swap), the value of the vector X is changed as soon as we change V inside the procedure. Furthermore, the actual non-scalar argument is not required to be a <leftside>. Thus the statement

```
Swap (2*X, 1, SIZE (X))
```

is accepted by HPL, though it has no net effect. (2*X is computed and stored somewhere temporarily, then 'Swap' exchanges two elements of this temporary vector, Then, since nothing else is being done with the vector, it is thrown away. X is unchanged.) On the other hand, the statement

```
Swap ({1.,2.,4.,8.}, 2, 3)
```

is also accepted, but is not recommended, It may be that a subsequent use of {1.,2.,4.,8.} will find the value {1.,4.,2.,8.} instead. It may even be that a subsequent use of the scalar 2. will find the value 4., and vice versa. But none of this is guaranteed. *Caveat emptor!*

4.2.2: Simpler version of section 4.2'1

In case you didn't understand anything we said in section 4.2.1, we'll now give a simple set of rules which, if obeyed blindly, should at least keep you out of trouble,

1) When describing formal parameters (in the WHERE clause), always include the attribute VALUE (or VALUES) for all scalar arguments.

2) If you assign a new value to a scalar parameter inside a procedure, it will not affect anything outside the procedure.

3) If you assign a new value to a non-scalar parameter, it will affect the actual parameter outside the procedure, maybe even if it is a constant. Since it is not a good idea to assign new values to constants, be careful.

4.2.3: Sample program

The examples which follow are in the form of a complete program which uses various procedures. Note in particular that all the program itself does is call one of the procedures with various arguments. This is effectively an alternative to the technique discussed in section 3.3 (in the fine print) for executing a program several times with minor variations. Instead of enclosing the whole program within one or more FOR loops, you can enclose it within a procedure body and call the procedure several times. This is slightly more flexible, but it is a moot point which method is simpler and/or more readable.

Examples:

```
        PROGRAM

        "This program will print out a bunch of multiplication
        "tables of various sizes, and in various bases. For
        "example, it will compute that 7x3=21, then (for the
        "base-4 table, say) will convert 21 to 111. The 111
        'is printed as if it were a decimal value, but would
        'be interpreted by a human reader as base-4 notation.'

        'The program uses three procedures, and here they are.'

    "(1) This procedure prints out an error message. The two
        "arguments, a character vector and an integer vector,
        "are incorporated into the message. Unlike the earlier
        'example of an error-message procedure, this one does
        "'return to the caller' instead of arbitrarily stopping,
        "Note also the use of a set generator as yet another way
        "to get a vector of 100 asterisks."
        PROCEDURE oops (Message, Data) WHERE
            Message IS CHARACTER VECTOR,
            Data ARE INTEGER VECTOR;  "The word 'data' is plural!"
        |_  WRITE <<>>;  "Print a blank line"
            WRITE {FOR i=(1,...,100), <<*>>}; "Line of stars"
            WRITE Message;  'Separate WRITEs for separate lines"
            IF SIZE (Data) >0 THEN  "Don't print null vector"
                WRITE Data;
            WRITE <<>>
        _|;

   :"(2) This procedure prints an integer matrix. Instead of
        "printing as many numbers per line as will fit (which
        "is what you'd get using a simple WRITE statement), it
        'prints one row of the matrix per line, double-spaced.
        "It considers it an error if the number of columns is
        "≥9, since 3 integers won't fit on a line. (We'll see
        "a way to handle this is section 6.2.) In addition to
        "the matrix to be printed, there is second argument, a
```

```
"logical value which if true causes the margins above the
"rows and to the left of the columns to be numbered, the
"numbering separated from the matrix by dotted lines. In
"this case there is a limit of 7 columns.  If you wonder
"just what all this output looks like, feel free to try
"out this program.  In case you're lazy, however, we'll
"include as a sample the output which would be produced
"by the second call."
PROCEDURE matrix_print(M, flag) WHERE
    M IS INTEGER MATRIX, flag IS LOGICAL VALUE:
    IF COLSIZE(M) > 8 OR (flag AND COLSIZE (M)=8),
    " 'AND' has precedence over 'OR'; the parentheses
    "are included solely for clarity."
        oops (<<Matrix too wide.  Dimensions are:>>,
            (ROWSIZE(M), COLSIZE (M)))
    ELSE  "The entire rest of the procedure is 'else'"
    |-  IF flag,  "Put out heading if requested"
        |-  WRITE (FOR i=(1,...,17),<<>>),
                (1,...,COLSIZE (M));
            WRITE (FOR i=(1,...,16),<<>>),<<+>>,
                (FOR i=(1,...,16 * COLSIZE (M)),<<->>)
        _|;
        "The headings are always to be read as decimal.
        "Thus the base-4 table will indicate that 7x3=111,
        "meaning 7₁₀×3₁₀=111₄, instead of showing that
        "13x3=111."
        FOR p:=(1,...,ROWSIZE(M)),
            IF flag,
            |_  WRITE p, <<|>>,M(p,*);
                WRITE (FOR i=(1,...,16),<<>>),<<|>>
            _|
            ELSE
            |_  WRITE M(p,*);
                WRITE <<>>
            _|
    _|;
"Note that the above procedure body is not surrounded by
"|_ ... _|, since it is a single IF statement.  Also
"note that it would be shorter and faster to use the form
"<<                >> instead of (FOR i=(1,...,16),<<>>),
"and it even looks clearer.  But it isn't.  It does make
"it clearer that what's being printed is a bunch of
"blanks, but the set generator makes it easier to see
"exactly how many blanks are being printed."

"(3) This procedure, as has already been noted, is in effect
    "the 'main program',  It has 4 arguments, x, y, b, and
    "flag, and uses the preceding procedure to print a
    "multiplication table of x rows and y columns in base b.
    "x y, and b are integer scalars, and an error message is
    "printed unless 2≤b≤10.  The 'matrix-print routine
    "places restrictions on the value of y, but they are not
    "enforced in the following procedure.  (Thus if we later
    "rewrite 'matrix-print' it does not necessitate changing
    "this procedure as well.)  The remaining argument is a
```

```
"logical flag which, if true, causes headings to be
"printed above and alongside the table.  It is not used
"In this procedure, but is merely passed along as an
"argument to matrix_print.  A general heading, perhaps
"more properly called a title, is printed regardless of
"the value of 'flag'."

"As for the method used to convert from decimal to base b,
"it is the standard technique of repeatedly dividing by b
"(integer- division) until the quotient is zero, whereupon
"the remainders are the b-ary digits in order from right
"to left.  By using'scalar multiplication and division,
"all elements of the table are converted simultaneously,
"'in parallel'."+
PROCEDURE table ix, y, b, flag) WHERE
     (x,y,b) ARE INTEGER VALUES, f lag IS LOGICAL VALUE;
     IF b<2 OR b>10,
          oops (<<Invalid base for 'table'.  x, y , & b are:>>,
               (x, y, b))
     ELSE   "Again, the rest of the procedure is ail 'else"'
     |-   DEFINE T  := COLUMN ( 11 ,...,x)) * ROW ((1,...,y)),
               digit := 1 ,  "initial digit in units place"
               zeroes := 0*T,  "Ha tr i x of zeroes"
               TbaseB  := zeroes:  "Stores final result"
          WHILE NOT T = zeroes "not the same as T ¬= zeroes ! ! ",
          |_   DEFINE ToverB:= T/b;  "Compute & add next digit"
               TbaseB  := TbaseB + (T-b*ToverB) * digit;
               T  := ToverB;  "Save quotient"
               digit  := digit * 1 0   "Advance to next digit"
          _|;
          WRITE <<Multiplication Table in base>>, b;  "Title"
          FOR i IN 11, 2),
               WRITE <<>>;  "Put out a couple blank lines"
          matrix-print (TbaseB, flag);  "Print table"
          FOR i IN (1,...,6),
               WRITE <<>>   "Put out a few more blank lines"
     _|;

"And final ly, the 'program' itself."
EXECUTE table (27, 7, 3, TRUE) ;
EXECUTE table (16, 5, 4, TRUE);
FOR i IN (2,...,8),
     EXECUTE table (i, i, i, FALSE)
"That ' s enough. "

END.
```

†Most computers nowadays do not actually do operations in pat-al lel, but would instead perform scalar multiplication by multiplying each element of the matrix in turn. One of the nice things about 'high-level' computer languages such as MPL is that they al low us to ignore the drudgery involved in how computers actually work, But we digress.

Here is the output which would be produced by the second EXECUTE statement above,

Multiplication Table in base 4

	1	2	3	4	5
1	1	2	3	10	11
2	2	10	12	20	22
3	3	12	21	30	33
4	10	20	30	100	110
5	11	22	33	110	121
6	12	30	102	120	132
7	13	32	111	130	203
8	20	100	120	200	220
9	21	102	123	210	231
10	22	110	132	220	302
11	23	112	201	230	313
12	30	120	210	300	330
13	31	122	213	310	1001
14	32	130	222	320	1012
15	33	132	231	330	1023
16	100	200	300	1000	1100

4.3: Procedures with Results

In the previous section we showed you how to pass information into a procedure, via arguments. But the only 'safe' way for a procedure to pass information back was by using 'global' variables, as in the examples 'reset' and 'overlay' in section 4.1. There is a more straightforward method, wherein a procedure can yield a specific result, which is then plugged directly into an expression. When it is necessary to distinguish these procedures from the forms already discussed, we will refer to these new ones as result procedures. (In FORTRAN these are called 'functions', while our earlier procedures would be 'subroutines'.)

This is done by naming a result variable when the procedure is defined. Whenever the procedure is used, the value it yields is equal to the last value assigned to the result variable, As a typical example, here

is a procedure called 'sin' which yields the sine of its argument. The result variable in this example is 'res'.

```
PROCEDURE res := sin(X)    "'=' can replace ':='"
    WHERE X IS REAL VALUE, res IS REAL:
|_  DEFINE term := X       "This is essentially the same old
    res := X;              "sin routine we've been using all
    DEFINE old := 0.0;     "along, with 'res' replacing 'sin'
    DEFINE i := 1;         "(we couldn't use 'sin' for the
    WHILE old ~= res,      'result variable since that's the
    |-  old := res;        "name of the procedure."
        i := i+2;
        term := -term*X*X/((i-1)*i);
        res := ree + term
    _|
_|
```

Several remarks should be made here. First, notice that 'res', in addition to being included ahead of the name 'sin', is also listed in the WHERE clause. In this particular case it wasn't necessary, since the default attributes (REAL SCALAR) apply. We could in fact omit the WHERE clause entirely, since X is also a real scalar, except we need to specify VALUE for X (see section 4.2.1 or section 4.2.2). Second, we need not worry about the newly-defined variables (old, term, and i) conflicting with other definitions outside the procedure. As we stated in the examples in section 4.1, any variables defined within a procedure act like FOR indices; the new definitions temporarily 'override' any previous ones and disappear after the procedure has finished. Thus, for instance, a program using the 'sin' procedure is free to have an integer matrix called 'old', and the matrix will not affect the procedure, nor vice versa. Third, it is possible (and occasionally even useful) for a procedure to have a result even though it has no arguments. We will include an example of this later on.

You'd probably like to know how to use these procedures. After all, if you say

```
EXECUTE sin(1.3)
```

the procedure might well compute the sine of 1.3, but what does it do with the result? The answer is that this isn't how it's done at all. To use a result procedure, you simply write its name, followed by a parenthesized list of arguments, anywhere in an expression. The value returned by the procedure is plugged into the expression at that point.

Examples:

```
DEFINE pi := 3.14159265358979, "approximately"
    s := sin(pi/4);    "Calls 'sin' with an argument of
                       "pi/4; assigns s the result. s is
                       "thus .5**.5, or about 0.70710678"
s := s + sin(s)*s + 1 ; "sin(s)*s = s*(sin(s)), not
                       "sin(s*s). s is now 2.16647"
DEFINE t := sin(sin(sin(1.)));
```

```
"Cal Is sin with an argument of 1., then cal Is sin again,
"giving it as an argument the result from the first call.
"The result from the second call is the argument for yet
"a third cal I, and the third result is assigned to t,
"This final result turns out to be about 0.67843"
DEFINE V := {sin(pi), sin(pi/2), sin(pi/4), sin(pi/6)};
"V is the real vector {0.0, 1 1 0, 0.70710678, 0.5}"
I F sin(s) < sin(t),   "sin(2.1665)=.8278, sin(.6784)=.6276,
     STOP:             "so this STOP is not done"
DEFINE V' := {F O R  i  IN {1,2,4,6}, sin(pi/i)}; "V'=V"
```

What sort of procedure would use no arguments but return a useful result? Wel I, one such might be a 'random number generator', which returns a uniformly selected random real scalar between 0 and 1 . When you invoke such a procedure, since there are no arguments, you may leave off the usual parentheses. L e . , instead of writing "random()", just write "random".)

Example;

```
PROGRAM

"This program performs simple tests on an equally simple
"random number generator (RNG). The RNG assumes an integer
"scalar variable, 'seed', exists, and that its value is an
"odd number between 0 and 1048576 (2^20). The value of the
"seed uniquely determines the number produced, and the RNG
'modifies the seed such that successive uses of the RNG
"produce a pseudo-random sequence of values, Unless the
"seed is altered by another part of the program, the se-
"quence repeats after 2^18 (262144) values. The 'random
"numbers' are uniformly distributed over the range 0 to 1."

"The tests being performed are, as mentioned already, quite
"trivial. First, 1000 random numbers are averaged. This
"result should be close to one-half. Next, the squares of
"another 1000 random numbers are averaged, This result
"should be about one-third. Last, another 2000 random
"numbers are multiplied in pairs, and the 1000 products are
"averaged. I f consecutive values in the random sequence
"are i ndeed i ndependen t, the average should be one-fourth."

GIVEN seed INTEGER; "Must define before used in procedure"

PROCEDURE r:= random; "No WHERE needed (r i s rea l sca l ar)"
"Random number generators tend to be ful l of 'magic'
"numbers.  This one is no exception."
|_    seed := seed * 1027;
     seed := seed - (seed/1048576)*1048576;
     r := seed / 1048576.0  "Don't want integer division"
_|;

DEFINE sum := 0.0;
FOR i IN {1,...,1000}, sum := sum t random;
WRITE <<Average: >>,sum/1000;
```

```
sum := 0.0:
FOR i IN {1,. . . , 1000}, sum := sum + random**2;
WRITE <<Average square:>>,sum/1000;

"Compare the preceding loop to the next one. You see,
"'random**2' is not the same as 'randomxrandom', because
"'random' has a different value each time it appears!"

sum := 0.0;
FOR i IN {1,. . . ,1000}, sum := sum + random*random;
WRITE <<Average product: >>,sum/1000;

"The preceding 3 loops are so similar that we are tempted
"to try to combine them-using loops and/or procedures.
'But they are so short anyway, how much could we hope to
"save? Anyway, that's if."

END.
```

Incidentally, when the above program was run with an input value of 79 for the seed, it produced the numbers .4954, .3344, and .2437, which look reasonable.

As a final example, consider the following procedure, which would have been useful in the multiplication-table program (section 4.2.3).

```
PROCEDURE string := repeat (char, number)
     WHERE char IS CHARACTER VALUE,
           number IS INTEGER VALUE,
           string IS CHARACTER VECTOR:
'Create a string consisting of 'number' repetitions of
'the single character stored in the scalar 'char'.
"Use DEFINE to set size of vector result."
     DEFINE string := {FOR i := {1,...,number}, char)
```

If we had had this in that earlier program, we could have replaced such statement9 as

```
WRITE {FOR i = {1,...,16}, << >>}, <<+>>,
     {FOR i = {1,...,16 * COLSIZE (M)}, <<->>}
```

with the more readable form

```
WRITE repeat (<< >>,16),<<+>>, repeat (<<->>,16*COLSIZE(M))
```

4.4: RETURN Statement

RETURN is to a procedure what STOP is to a program. If, in a procedure, the statement

```
RETURN
```

is executed, the procedure terminates at once and returns to the caller. If the procedure has a result, it is the value of the result variable at the time the RETURN is performed.

RETURN statements are rarely necessary, but they are sometimes convenient. As an example, we'll rewrite the 'matrix-print' procedure of section 4.2.3 using a RETURN, While we're at it, we'll assume the existence of the 'repeat' procedure for character strings,

```
PROCEDURE matrix-print (M, flag) WHERE
    M IS INTEGER MATRIX, flag IS LOGICAL VALUE:
|-  IF COLSIZE (M) > 8 OR (flag AND COLSIZE (M) = 8),
    |_  oops (<<Matrix too wide.  Dimensions are:>>,
              (ROWSIZE(M), COLSIZE (M)));
        RETURN
    |;
    "If we get this far, the IF must have been false."
    "The rest of the procedure is largely unchanged,
    "except it is no longer inside an ELSE-clause."
    IF flag,
    |_  WRITE repeat(<<>>,17),(1,...,COLSIZE(M));
        WRITE repeat (<<>>,16), <<+>>,
              repeat(<<->>,16*COLSIZE(M))
    _|;
    FOR p := (1,...,ROWSIZE(M)),
        IF flag,
        |_  WRITE p,<<|>>,M(p,*);
            WRITE repeat (<<>>,16),<<|>>
        _|
        ELSE
        |_  WRITE M(p,*);
            WRITE <<>>
        _|
_|;
```

4.5: Library Functions

HPL has several 'pre-defined' procedures available for you to use, which do things like compute the inverse of a matrix, or the transpose, or find the maximum element of a vector. These procedures, called library functions, are used like any other procedure. In fact, SIZE, ROWSIZE, VECTOR, etc., are really library functions, This being the case, we've had several examples showing how to use them, so we won't give any examples here. A complete list of available functions is in Appendix A.

Many more functions are available in the standard FORTRAN library, including sin, cos, log, and dozens more. It is possible to use FORTRAN functions in an HPL program, We'll cover this in Section III.

Note that, while certain of the HPL library functions are also

keywords, most are not. This means you can use them as identifiers, overriding their pre-defined meanings. Thi s i a not recommended, however, since strange things can occur. Consider this sequence:

```
DEFINE X =(2,1,4);
WRITE SIZE (X);    "Outputs '3' "
DEFINE SIZE =(4,2,3,5,1);
WRITE SIZE (X);    "Outputs ' 2  4 5'"
```

5: BLOCK STRUCTURE

In Section I we cautioned you that it is illegal to redefine the type or structure of a variable. We also said there was one particular exception to this rule, which would be discussed under 'block structure'. Since then we've seen some specific exceptions, such as **FOR** indices and arguments to procedures. We shall now show that, with suitable **hand-waving**, most of these exceptions (all but the FOR indices) can be explained as an aspect of **block** structure.

5.1: **Block-Structured** Programs

The basic idea is that a program consists of blocks of code. Little blocks go inside bigger blocks, which in turn go inside still bigger blocks. Just as with physical building blocks, program blocks cannot 'partially overlap'. That is, if two blocks overlap at all it is because one of them is entirely contained within the other. In an HPL program the **biggest block** is the entire program. Each procedure is also a block, contained within the program's block. In a moment **we'll** specify exactly what defines a block, but first we want to explain how block structuring affects the redefinition of variables, We shal l use the terms inner block and outer block with the interpretation that the inner block is any block contained in some other block, which is the outer block.

When you enter an inner block, all of the variables, labels (for GO **TOs**), and procedures which have been defined in the outer block are sti l l defined, They may thus be used within the inner block. However, if you wi sh, you may redefine any or al l of them without regard for their previous clef **initions.** If you do this, then the external definitions (i.e., the meanings which the identifiers had in the outer block) are no longer accessible to the inner block,

When you leave an inner block, any definitions which took place inside that block are 'undone'. If the identifiers in question had been defined in the outer block as well, then they are restored to their previous attributes and values; otherwise they become undefined.

Example:

```
PROGRAM

DEFINE (a, b) INTEGER SCALARS,
a := 7;
b := 9;

PROCEDURE result1 := f1(k) WHERE k IS INTEGER VALUE,
                                  result1 IS INTEGER;
     result1 := kxkt

PROCEDURE result2 := f2(k) WHERE k IS INTEGER VALUE,
                                  result2 IS INTEGER;
     result2 := -3*k;

WRITE a, b, f1(2), f2(2);
'So far this is nothing new. We've just written out 4
"integers, namely 7, 9, 4, and -6."

PROCEDURE example;
'Since-procedures are sub-blocks within the main program,
"we've just entered an 'inner block', Therefore we may
"now redefine anything we wish."
|_   DEFINE (b,c) REAL SCALARS:
     a := 1.61803;
     b := 2.71828;
     c := 3.14159;
     "Since we didn't redefine 'a', the only ,'a' we've
     'got is the one we defined back at the beginning.
     'Therefore, when this procedure gets executed, we
     'will change the value of that 'a'. On the other
     "hand, since we redefined 'b', the one defined at
     "the beginning will be unaffected, When we leave
     "this procedure, 'a' will have the value 1, since
     '1.61803 gets rounded down, but 'b' will again be
     "9. However, within the procedure, 'b' is a real
     "scalar with value 2.71828."
     DEFINE f2 := (5,10,...,100);
     "f2 used to be a procedure, Within this 'example'
     "procedure, f2 is redefined as an integer vector.
     "We haven't done anything with f1, so it remains
     "a procedure. "
     WRITE a, b, c, f1(2), f2(2)
     'This will write 1, 2.71828, 3.14159, 4, and 10.
     "Note that 'f1(2)' says to execute f1 with k=2,
     "while 'f2(2)' now means the second element of
     "the vector f2."
_|;

EXECUTE example; "Perform the above actions"

WRITE a, b, f1(2), f2(2);
"Since b and f2 were redefined within the inner block of
"'example', they have now reverted to their old defini-
"tions, namely b is 9 and f2 is a procedure. Since 'a'
```

```
"was not redefined in 'example', its value in the outer
"block has been changed.  So this WRITE statement prints
"the values 1, 9, 4, and -6."

WRITE c    "I I legal (c is no longer defined)"

END.
```

Hopefully the preceding example has given you some idea of the effects of block structure. Effectively what it allows you to do is write procedures which can be used in several different programs wi thout your having to worry about whether the variable names in the procedures conflict with those in the programs.

5.2: Definition of a Block

Now it's time for us to state explicitly what a block is. There are real ly only 3 kinds, and we've already mentioned two of them. The whole program is a block, and *outside* that block is where the library functions are defined for you, (This explains why you can use the library functions, but are al lowed to redefine them if you like. The main program is like an inner block within the MPL universe.) The second type of block is a procedure. Every procedure is a block, within which the formal parameters and result variable (if any) are defined automatically for you by MPL. (This explains why a formal parameter can have the same name as a variable in the main program, as in the example in section 4.2.1.) The third type of block is one which you explicitly designate. It looks exactly l ike a compound statement, except instead of BEGIN and END (|_ and _|) you use BLOCK and END,

Example:

```
"This program does some sort of bizarre calculation, The
"only important things going on are in the comments. if
"you get turned on by strange mathematical formulae, you
"can try to figure out just what gets computed by al l this."

"The problem is, there's hardly ever a need for explicit
"BLOCK statements; certainly never in smal l programs.
"So we just put together something moderately complex,'

DEFINE x-10000, y-l ;
WHILE y <x,
     BLOCK     "Block used just as if it were '|_'"
     DEFINE x = y**.5;  "This doesn't affect the x in the
                        "WHILE statement, since that is
                        '*outside this block."
     WRITE y,x**x;
     y = (0,...,y) * (1,...,y+1)
     END;    "End of special block"
WRITE y , x   "This x is still equal to 10000
```

"(y happens to be 4665760)"

"For the insatiably curious, the garbage that gets printed
"by this useless program is

1	1.0
2	1.6325
8	18.93 --
240	2.7355E+18
4665760	10000

In case you hadn't guessed, explicit blocks (BLOCK . . . END) are not particularly useful, but they are there if you need them. One possible use is when you've got a big program in which, by mistake, you used the same identifier for different types of variables in different sections, resulting in an error (illegal redefinition). Rather than going through half your program changing variable names, you might (if you're lucky) be able to get away with just slapping a BLOCK . . . END around one chunk of the program, such that the redefinition is permitted. By 'lucky' we mean that this won't work unless your program is easily broken into independent chunks, since when you leave the block all of the variables defined within it 'go away'. (Of course, if your program can be broken up this way, you should have written it as a bunch of procedures, as suggested in section 4.1. This way you would have avoided all this trouble in the first place, since procedures are automatically blocks.)

6: INPUT/OUTPUT (REVISITED)

The I/O facilities described in Section I were more or less uncontrolled. The input data had to be free-format; there was no way to treat a string of digits as distinct one-digit numbers. The output always used an enormous amount of space to print each number; there was no way to fit more than 8 integers or 5 reals on a single line of output.

NPL provides for three levels of control over your I/O. The first, unformatted I/O, was described in Section I. The other forms we shall term semi-formatted and formatted.

6.1 : Semi-formatted Output

There is no such thing as semi-formatting Input. Semi-formatting allows you to specify to MPL how much space should be used to print integer and real values. This is accomplished using the special, pre-defined variables, INTEGERSIZE and REALSIZE. Like library functions, these two variables have special meanings, which you can override if you wish by defining them to be something else. If you do so, however, you can no longer take advantage of the special pre-defined meanings.

Whenever an unformatted WRITE statement prints out an integer, it uses 16 characters (many of which are blanks). It prints the integer right-justified on a field of 14 characters, then tacks 2 blanks on the end. NOW, as it happens, the special variable INTEGERSIZE starts out as an integer scalar uhose value is 14. If you assign a new value to INTEGERSIZE, then any subsequent unformatted WRITEs will use the new field width. You will still get the 2 trailing blanks, which are not counted as part of the field width. If an integer is too large to be printed in the room allotted, it will be printed as a bunch of asterisks.

Example:

```
DEFINE V = [1,-10,100,-100,100000];
WRITE V;
FOR i IN [10,6,4,1],
|_   INTEGERSIZE := i;
     WRITE V
,_|
```

This produces 5 lines of output, as follows,

```
             1            -10            100           -100          100000
         1          -10          100          -100       100000
     |      -10      100     -100  100000
   1   -10    100  -100 -****
1  *   *   *   *
```

If you set INTEGERSIZE to be zero or negative, the default field width of 14 (plus 2 trailing blanks) is used instead.

REALSIZE is the analogous variable for controlling how much room to use for real values. It starts out equal to 22 (again, with 2 trailing blanks).

Although semi-formatting gives you some amount of control over the appearance of your output, it is still very limited. You can't change INTEGERSIZE in the middle of a WRITE statement, for example. To get full control, you need to use formatted I/O.

6.2: Formatted I/O

Well, we hate to have to say this, but we're going to cop out on this one. MPL formatted I/O makes use of FORTRAN formats, and describing FORTRAN formats would take several pages. If you want to be able to do all sorts of fancy things, we suggest you find a good FORTRAN text and study the section on FORMAT statements, What we shall do here is describe some of the simpler forms of FORTRAN formats, explain how they are used in MPL, and give a few examples.

First, the syntax. An MPL formatted READ or WRITE statement looks just like an unformatted one, except the first thing in the list of expressions must be a character vector and must be followed by a colon

instead of a comma. This character vector is then taken to be the format and is not printed,

Examples:

```
DEFINE (V,V') VECTORS 10,
     Fmt:= << (5F10.4) >>;
READ <<(10F5.0)>>: V   , V';
WRITE Fmt :V+V';
```

This would read 10 real values from one card and assign them to V, then read 10 more real values from the next card (even if there is more data on the first card) and assign them to V'. It would then print the vector sum in two lines of five numbers each. (We'll explain this a little more later on. Right now we just wanted to show you what it looked like.)

Now, as to the FORTRAN formats, First off, they must always begin with a left parenthesis and end with a right parenthesis. As for what goes in-between, well,, as we've said, it can get quite complex. We shall discuss the following types: Iw (for integers), Ew.d and Fw.d (for reals), Lw (for logicals), A w (for characters), 'xxx' (text constants), X, and slash ('/' to end a line). Throughout these discussions, we shall use 'w' and 'd' to mean any non-negative integer constants. For instance, when we say 'Fw.d' we mean anything like F9.4, F10.0, F13.10, etc.

The general form of a format is a series of format items, such as 17 or E10.3 or 'frog', separated by commas. A slash need not be separated from adjacent items. Blanks between items are ignored, Thus a typical format might look like

```
(I3,I3,I3,F9.0,'glork'/A10 , 'glork' /A10/////// I5)
```

If you want to repeat an I, E, F, L, A, or X item, you may do so by prefacing it with a number indicating the number of repetitions desired. Thus the above format is equivalent to

```
(3I3,F9.0,'glork'/A10,'glork'/A10///////I5)
```

(We got rid of the excess blanks while we were at it.) Whole sections can be repeated by enclosing them in parentheses and putting the repetition count ahead of the left parenthesis. Thus the above could be rewritten

```
(3I3,F9.0,2('glork'/A10),7(/),I5)
```

Note that, when the 7 slashes became 7(/), it was necessary to include commas to separate them from the adjacent items,

We shall now discuss what each of the different types of format item means. The formats behave slightly differently for input than for output, so we'll discuss the two separately.

6.2.1: Output formats

Integers may be written only using the Iw format Item. The value is printed right-justified on a field of 'w' characters. You do *not* get any

trai l ing blanks uhen you use a formatted WRITE (this is true for real item types as well). If the field is not wide enough to contain the number, a bunch of asterisks Is printed out instead.

Example:

```
DEFINE M INTEGER MATRIX 2 BY 5;
M(1,*) := (1, -1, 0, 79, -79);
M(2,*) := (4444, 55555, 666666, 7777777, 888888881;
WRITE <<(313, 415, 216, 110, 112)>>: M;
```

This would produce:

```
 1 -1 0   79  -73 444455555666666****** 88888888
```

Notice that some of the numbers got run together because there was not enough room to include a leading blank.† Notice also that the final item, 112, happens not to be used. Lastly, notice that the elements of the matrix are output in the same order as for unformatted I/O, row-major order.

Reals may be uritten using either Ew.d or Fw.d items. Both forms specify a field of 'w' characters. Ew.d uses 'scientific notation', so that 55 prints as "0.550 02", and 'd' significant digits are printed. Fu.d uses ordinary notation, and prints 'd' digits after the decimal point. Cont inuing the previous example:

```
DEFINE tl' = M * 0.001;
WRITE <<(5E10.3)>>: M'(1,*);
WRITE <<(4E11.2,E11.4)>>: M'(2,*);
WRITE <<(F5.2,F7.3,3F9.4)>>: M'(1,*);
WRITE <<(5F7.2)>>: M'(2,*);
```

Thi s produces:

```
 0.1000-02-0.1000-02  0.0        0.790D-01-0.790D-01
   0.440 01    0.560 02    0.670 03    0.780 04 0.88890 05
 0.00 -0.001  0.0       0.0790 -0.0730
   4.44 55.56 666.677777.78*******
```

In the last line of output, the values 666.67 and 7777.78 got run together. Note that 0.0 seems to be a special case. This may not be the case with ail versions of MPL, since it depends on the local dialect of FORTRAN.

Logicals are uritten using the Lw item. A logical value prints as a 'T' or 'F' preceded by (w-1) blanks. Thus

```
WRITE <<(2L2,4L1,3L3,L5)>>: (FOR i IN (1,...,10), i<5 OR i=8);
```

produces

†Some versions of MPL may not allow this, and uiii treat these cases as if there were insufficient room to fit the number. Also, some versions may do other things when a number won't fit, instead of printing asterisks. But these are ail fairly minor details,

T TTTFF F T F F

Characters are written using the Au format item. Each such item prints a single character preceded by (w-1) blanks. Thus, to print a character vector of size 10, you should use 10A1, not A10. In addition, if you want to include a character constant in your output, you can do it either by putting it in the list of things to be output and printing it with Al format items, or you can use the 'xxx' format item. The 'xxx' item does not use any expressions from the output list, but simply prints whatever is between the apostrophes. (If you want to print an apostrophe, write it as two apostrophes,) Thus the two statements

```
WRITE <<(I4,10A1,I4)>>: 79, <<I t's easy!>>, 79;
WRITE <<(I4,'I t''s easy!',I4)>>;79,79;
```

produce the same output, namely:

```
 79I t's easy! 79
```

Continuing the earlier examples:

```
WRITE (<<(I3,':',4(I5,','))>>: M(1,*));
```

This time we decided to include the optional parentheses around the whole output list. This statement produces:

```
1:  -1,   0,   79'  -79,
```

Notice that the text items (':' and ',') are printed interspersed among the elements of M. Items in the format are used in the order in which they are encountered. (See fine print for more detailed explanation.)

Going through this more slowly, here's what happens. We start going through the format, and the first thing we find is an I3. So we need an integer to print. Since M(1,*) is 5 integers, we print the first element, 1. The next thing in the format is ':', so we print a colon. Now we hit a repetition count, which we skip over for now. The I5 requires another integer to be printed. Since we haven't finished with M(1,*) yet, we print its second element, -1. We come to the ',' item and so print a comma. Now, repeating the parenthetical group, we hit I5 again, and print M(1,3). And so on.

The X format item simply causes a blank to be printed. To get more than one blank, use a repetition count (i.e. use 10X, not X10). Strictly speaking, you never need to use X items, but they can make formats a bit clearer. For example, the two statements

```
WRITE <<(2X,5I2,3X,'frog',2X,4A1)>>: (1,...,5), <<toad>>;
WRITE <<(I4,4I2,'   frog',A3,3A1)>>:(1,...,5), <<toad>>;
```

produce the same output, namely:

```
 1 2 3 4 5    frog toad
```

Lastly, the slash (/) format item is used to go to the next output line prematurely'

Example:

```
WRITE <<(10I3)>>:(1,...,10);
WRITE <<(4I3/5I3/I3)>>:(1,...,10);
```

produces

```
  1  2  3  4  5  6  7  8  9  1  0
  1  2  3  4
  5  6  7  8  9
 10
```

In addition, if there are more things to be written than there are format items to correspond to them (this therefore doesn't count 'text', X, or / format items), then an extra / is automatically inserted, after which the format is restarted starting from the last left parenthesis. (Don't complain to us: this is all standard FORTRAN.) If the left parenthesis is preceded by a repetition count, the count is also included in the restart.

Example:

```
WRITE <<(I4,':',3I3)>>:(1,...,10);
WRITE <<(I4,':',3(I3))>>:(1,...,10);
```

produces

```
   1:  2  3  4
   5:  6  7  8
   9: 10
   1:  2  3  4
 5  6  7
 8  9 10
```

6.2.2: Formatted input

Using unformatted input, you could scatter your input data as widely as you wished--extra spaces or even blank cards betueen numbers made no difference. This is not the case with formatted input. If you read a number using an I5 format item, MPL will go out and grab the next 5 characters of input, be they digits, blanks, or pure garbage. It's up to you to make sure those 5 characters represent an Integer, or the results are unspecified. So, formatted input is generally used only when (a) your input data is in a fairly regular form (often having been supplied to you by someone else) and (b) unformatted input won't handle it. There are three main reasons why unformatted input won't handle certain data. First, there may be numbers with no intervening spaces, such as streams of digits intended to be taken as individual one-digit numbers. Second, you may wish to read character or logical data, which you cannot do using an unformatted READ or GIVEN statement, (Note: There is no such thing as a formatted GIVEN statement,) Third, there may be useful data in columns 73-80 of the data cards. Unformatted input ignores these columns, but formatted READ statements may use them,

Another important difference between formatted and unformatted READs

is that, whereas an unformatted READ could read half of an input card and leave the rest to be read later, a formatted READ throws away any unused portion of an input line that may be left over when the READ is completed. A subsequent READ,. formatted or unformatted, will start reading at the beginning of the next input card.

Integer values are read using Iw format items. 'w' input characters are read and interpreted as an integer value. (In all formatted input, the results are unspecified if the input data cannot be interpreted in the required way, such as if you use an 15 item and the input card says "1234X", or even "12.00". Leading blanks are all right,)

Example:

```
DEFINE M INTEGER MATRIX 2 BY 5;
READ <<(3I3,5I1,I4,I2)>>: M;
```

If the input data is

```
123-5678901234+678 012
```

then M becomes

```
123 -56 789    0    1
  2    3    4 678    0
```

The last two data characters ("12") are not read and are discarded.

Real values are read using Ew.d or Fw.d items. It does not matter which you use; both ".35E1" and "3.5" may be read by both E and F format items. 'w' input characters are read and interpreted as a real value. If the input data includes a decimal point, then the 'd' part of the format item is ignored. Otherwise a decimal point is assumed 'd' places from the right of the units digit; i.e. the number is multiplied by 10^{-d}. (If this confuses you, always use zero for 'd' (as in F7.0), whereupon the 'd' part will never have any effect whatever.)

Example:

```
DEFINE V REAL VECTOR 6;
READ <<(F7.0,4F7.2,E5.1)>>: V;
```

If the input data is

```
-12345 -12345 12.345 12E-31.2E-3 123,
```

then V becomes {-12345., -123.45, 12.345, 0.00012, 0.0012, 123.}.

We shall now explain the 3rd and 4th elements in more detail. For V(3), the format was F7.2 (the second of four repetitions), and the 7 characters read were " 12.345". Since the 7 characters included a decimal point, the value was taken as is, and V(3) = 12.345. For V(4), the format was again F7.2, and the 7 characters were " 12E-3". No decimal point was given (even though an exponent (E-3) was specified) so a decimal point was assumed 2 places from the right of the units position, yielding .12E-3, or .00012.

Logical values are read using Lw items. 'w' characters are read, and

the first non-blank character is examined. If it is "T", the value is true; if it is "F", the value is false. If it is neither, -or if all 'w' characters are b l anks, the result is unspecified, -

Example;

```
DEFINE L LOGICAL VECTOR 8;
READ <<(3L7,5L1)>>:L;
```

If the input data is

FALSE TRUE FROG TFFTF

then L becomes (FALSE, TRUE, FALSE, TRUE, FALSE, FALSE, TRUE, FALSE).

Character values are read using Au items. (Note: The 'xxx' format item must not be used for input.) 'w' characters are read, and the last one is used as the value read.

Exampl e:

```
DEFINE C CHARACTER VECTOR 13;
READ <<(A5,3A7,7A1,A4,A5)>>:C;
```

If the input data is

ANYTHING THAT CAN GO WRONG, WILL GO WRONG.

then C becomes the character vector <<HAGG, WILL W.>>. (Don't ask us who Will W. Hagg is; we're still trying to figure out how he managed to get his name into this manual, Something apparently uent wrong.)

The X format item causes a single data character to be skipped. As with output formats, the form Xw is illegal: to skip 10 characters, use 10X.

Example:

```
DEFINE (V1,V2) INTEGER VECTORS 6;
READ <<(I3,I4,2I2,I5,I4)>>:V1;
READ <<(I3,2X,3I2,X,2I4)>>:V2;
"Remember that, since the first READ discards any leftover
"portion of its input l ine, the second READ will read on
"the second l ine of input."
```

If the input data is

```
12345678901234567890
12345678901234567890
```

then V 1 becomes (123,4567,89,1,23456,7890) uhi le V2 becomes (123,67,89,1,3456,7890).

Lastly, the slash (/) format item cause6 the rest of the current input card to be thrown away. The READ continue6 on the next card. As with output, If a format 'run6 out' and has to be repeated, an extra '/' is assumed.

Example;

```
DEFINE V INTEGER VECTOR 10;
READ <<(213/I4,212)>>;V;
```

If the input data is

```
12345678
90123456
78901234
56789012
34567830
```

then V becomes (123,456,9012,34,56,789,12,5678,90,12).

The '213' reads 123 and 456, then the slash causes the rest of the line to be discarded. The '14,212' reads 9012, 34, and 56. The end of the format is reached, but we still have more elements of V to read, so a slash is assumed, discarding a bunch of blanks left over on the second line. As it would with output, the format now repeats starting from the last left parenthesis, which happens to be at the beginning. '213' reads 789 and 012, then '14,212' reads 5678, 90, and 12 from the fourth line. The last line is unused and would be read by the next READ statement.

6.2.3: Carriage control

MPL formatted output uses the first character of each output line for carriage control, the same as in FORTRAN. If you know what this means, you can skip this section.

We haven't been completely honest with you about formatted output. Everything gets printed just the way we said it does, except that the *first* character of each line is not printed. Instead it is used to specify such things as double-spacing, triple-spacing, etc. These things are called 'carriage control' (because they control the motion of the 'carriage' of the line printer), and the first character of each line is referred to as the 'carriage control character'. This applies only to formatted output, not to semi-formatted nor unformatted output.

The interpretations of the carriage control character are as follows.

Character	Interpretation
blank	Normal (single-spacing)
0	Leave a blank line ahead of this one (double-spacing)
	Leave two blank lines ahead of this one (triple-spacing)
+	Print this line on top of the preceding one
1	Print this line at the top of the next page

Any other characters may have unpredictable effects, although usually they act as if they were blanks.

Example:

```
WRITE <<('frog'/'0toad'/'+////'/I5)>>; 79
```

Ignoring carriage control for the moment, this would print

```
 frog
0toad
+////
   79
```

Now, taking into account the carriage control characters, the first line print6 normally. The second line is double-spaced with respect to the first, and the third I ins print6 on top of it. The last line prints normally, below the preceding one. The net effect is

```
frog

toad
  79
```

If you don't want to use these features, just be sure every I ine of formatted output starts with a blank. Much of the time you don't even have to think about it: if the first thing on the line is a numeric value then it almost always starts with a blank. But be careful. Suppose you wr i te an integer vector V using

```
WRITE <<(10I7)>>: V   "Write V in lines of 10 values each"
```

This is fine as long as the values in V are suitably small. But if V(1) equa l s -100000 then you' l l get a '-' in the Carriage control position, 60 the l ine wi l l print triple-spaced, and V(1) wi l l appear to be positive 100000. Or suppose V(1) is 1234567, Then the line wi l l print on a fresh page, and V(1) wi ll appear to be 234567. So If you are not explicitly including a carriage control character, be sure your format items are large enough, or yaur number6 smal l enough, such that a l l your 'numbers are guaranteed to print with leading blanks.

7: THE LET STATEMENT

7.1: What It Does

The LET statement defines a synonym which you can use at compile-time. Just about anything we might say to describe it formal ly would use terminology with which you may not be familiar, 60 we' l l at tempt to exp l a in it using examples.

Examp le:

```
LET F(x) := M(*,x) + x**B;
WRITE F(2)
```

The first statement defines a synonym, F, which is very much I ike a 'resul t' procedure. The second statement is effectively changed by MPL into the statement

```
WRITE M(*,2)+2*B
```

In this particular instance we could have got ten the same effect using an ordi nary procedure, thust

```
PROCEDURE R :=F(x) WHERE x IS INTEGER VALUE,
                         R IS COLUMN;
    DEFINE R := M(*,x) + x*B;
"Must use DEFINE in order to establish size of R."
WRITE F(2)
```

The LET notation has three main advantages. First, it is shorter and thus more natural. Second, it is substituted direct ly into your program wherever it is used. Let's look more closely at this second difference. A procedure is created once: the code is set up somewhere. When you use it, your program branches over to that code, computes whatever i t' 6 supposed to, and comes back. When you use a LET synonytn, the code i s dup l i ca ted every time you use it, This is faster, s i nce you no l anger have to branch back and forth, but it uses more storage space in the computer. However, since a LET synonym has to be a single expression, it is unlikely to take up much space. The main consideration in using LETs is thus convenience.

The third advantage is also one of convenience--you do not have to define the type6 or structures of the arguments or result, the way you do for procedures. In fact, you couldn't even if you wanted to: MPL won't accept a WHERE clause in a LET statement. Thus if you set up the synonym

```
LET  add  (x,y)=x+y;
```

and define the fol lowing variables:

```
DEFINE RV REAL VECTOR 10,
       IV INTEGER VECTOR 10,
       (a,b) REAL SCALARS,
       M MATRIX 7 BY 9;
```

then all of the following form6 (and many others) are valid:

```
add (a,b)      yields atb, a real scalar
add (RV,b)     yields RV+b, a real vector
add (IV,IV)    yields IV+IV (i.e. 2*IV), an integer vector
add (a,M)      yield6 a+M, a real matrix
```

Perhaps it's time we gave the general form of a LET statement.

```
LET <identifier> (<argument list>):=<expression>;
```

The semicolon is required. The keyword $LET may replace LET (don' t ask why), and as usual '=' may replace ':='. If there are no arguments, the

parentheses may be omitted. Like other definitions, synonyms are subject to block structure (section 5) and thus vanish when you exit from the blocks in which the LET statements occur.

7.2; How It Does It

In case you are interested, we wi l l now describe exactly how a LET synonym is treated by MPL. Even if you're not interested, you should skim through the rest of this section, because at one point we'll be mentioning a 'bug' which you might run into, and which appears terribly mysterious unless you know what's going on.

Once you've defined a synonym, you may invoke it the same way you would use a 'resul t procedure', as shown in the earlier examples. So here comes MPL, trundl ing through your program, and i t comes to a synonym name fol lowed by a list of 'actual parameters'. Here's what happens, broken clown into steps. Some of these step6 may appear strange; we' l l soon explain the reasons behind them.

ill MPL rummages--around and finds the <expression> given in the relevant LET statement.

[2] The actual parameters are substituted for the corresponding formal parameters in the expression, with an extra set of parentheses automatically put around each one,

[3] The entire expression is put inside parentheses.

[4] MPL takes the resulting expression and pretends you wrote it instead of the synonym cal l. (This leads to the problem we mentioned. We're getting to it.)

By these rules, using the 'add' synonym given earl ier, if you write

```
x := add (y,z)
```

then MPL acts as if you had written

```
x := ((y) + (z))
```

You might well wonder why on earth all those parentheses are being tossed in. They are there 60 that synonyms will behave like procedures. Suppose we. have the 'add' synonym, and al so a 'times' synonym

```
LET times (x,y) = x * y;
```

Then consider the possibi lities. If we didn't insert parentheses around each argument (step 2), then

```
time6 (3 + 5,7 + 9)
```

(which we would like to be 8*16, or 128) would yield

```
(3 + 5 * 7 + 9)
```

which is 3 + 35 + 3, or 47. If we left out the extra set of parentheses around the entire expression (step 3), then

```
add(3,5) * add(7,9)
```

(which again we'd like to be 128) would yield

```
(3) + (5) * (7) + (9)
```

which is also 47. By the rules actually used, these two examples yield

```
((3 + 5) * (7 + 9))  and  ((3) + (5)) * ((7) + (9))
```

both of which produce 8*16, or 128.

Okay, now that we've explained the situation and the rationale behind it, here's the problem, Suppose you say

```
WRITE add(3,5) * add(7,9)
```

which is interpreted as

```
WRITE ((3) + (5)) * ((7) + (9))
```

This runs into the problem uith WRITE statements mentioned back in Section I (subsection 7.2.2), namely, the compiler does not know how to distinguish the parentheses used to enclose the list of items to be written out from the parentheses enclosing parts of the expression.

To get around this, you must include the 'optional' parentheses in a WRITE statement uhenever the first thing being written starts with a synonym. In this example, you'd write

```
WRITE (add(3,5) * add(7,9))
```

Note: It turns out you don't need these parentheses if the only thing being written is a single synonym call. You might try applying the 4 steps given above to our first example, F(2), to see why this is so.

7.3: What It Can Do (But Don't)

Don't read this section if you get confused easily.

If you have a devi ous mind (as we do) you may have thought of some bizarre ways of actually *taking advantage* of the extra parentheses being forced upon you.

Example:

```
LET add (x,y) = x + y;   "Same as before, but now., ."
LET weird (x,subscript,y) = x subscript + y;
DEFINE Q:= weird ({1,...,10} + {100,200,...,1000}, 7,
          {1000,2000,...,20000}) add (7,9)
```

The last two statements are not strictly kosher, since in both we have two things placed side-by-side with no operator between them. But if we go ahead and apply the 4-step rules to the assignment statement, it turns out to be

```
DEFINE Q:=(({1,...,10} + {100,200,...,1000})(7)
          + ({1000,2000,...,20000}))((7) + (9))
```

which is a valid statement after all (the (7) and ((7) t (9)) are both used as subscripts). Q is assigned the integer scalar 16707,

Now, the only reason we mentioned this is because, if we didn't, some clever person would try it anyway. By bringing it up ourselves, we can take this opportunity to tell you: DON'T DO *THIS!* We can't promise that MPL will always work this way, so you'd only be asking for trouble. Furthermore, you can accomplish the same things legally by using normal notation:

```
LET add (x,y) = x t y;
LET weird (x,subscript,y) = x(subscript) t y;
DEFINE Q:= weird ({1,...,10} t {100,200,...,1000}, 7,
          {1000,2000,...,20000}) (add (7,9))
```

8: HOW TO USE MPL (REVISITED)

There are several useful variations you can make on the basic method of running an MPL program as shown in Section I. The additional features which we are about to discuss fall into two categories. Some of them are used for creating external 'libraries' of MPL functions. We will see in Section III how you can make use of such libraries, but we feel we might as well show you now how to create them. The other new features are 'compilation options', which allow you to do such things as suppress the listing of your MPL program, or produce an 'object deck'. (If you're a computer buff you know what an object deck is; if you're not you probably don't care about how to get one,)

8.1: Creating MPL Libraries

By 'library' we mean a collection of programs and/or procedures which have already been compiled (this is what we're about to show you how to do)

and which can then be used by other programs without having to include the the 'I ibrary' procedures explicitly in the later programs. We'll see how to use iibrary procedures in Section III. But ue felt that, as long as we were going to be talking some more about how to use MPL, we ought to include everything.

Every program or procedure you put into the library has got to have a name. For procedures the name is simply the name of the procedure. For programs, you must specify a name by putting it on the PROGRAM card. The name must be fol lowed by a semicolon. For example,

```
PROGRAM TRAFFIC;
```

might be the first line of a program for doing traffic flow analysis. We might anticipate using the program several times, so we could put it into a I ibrary under the name "TRAFFIC".

When a procedure is being put into a library, it is written all by itself (as opposed to being inside a program). For example, i f we had wr i t ten a procedure which returned the reversal of its real vector argument, and we wished to put this procedure into a I ibrary under the name REVERSE, ue'd wr i te:

```
//COMP.SYSIN D  D *
PROCEDURE V' := REVERSE(V) WHERE (V, V') ARE VECTORS:
     DEFINE V' := V((SIZE(V), SIZE(V)-1,..., 1));
/*
```

Next, ue'd like to be able to compile several procedures at once and store them al I in the same library. To do this, just put them al I one right after another, ui th an extra card between each pair of procedures, The extra card should read

```
%MPL
```

We'l l show an example of this in a moment, The only other thing to worry about is how we specify where the library is going to be. This is done in the standard way (i.e., standard among the major language processors on the IBM system where MPL currently resides). I n case you don' t know what the standard way is, it involves putting this card ahead of your "COMP.SYSIN" card:

```
//COMP.SYSLIN D  D DSN=<dsname>,DISP=(NEW,KEEP),UNIT=<disk>,VOL=<volume>
```

where <dsname> is the "data-set name" of your I ibrary, <disk> is whatever a disk is called at your installation, and <volume> is the volume where you want the library to be created (i.e. the particular disk). This is all standard IBM JCL; in other words, incomprehensible, We are even less inclined to teach JCL than we were to teach FORTRAN formats, so don' t expect any more details.

When you wish to use these routines in a later program you wi l l have to tell MPL what libraries to look in and where to find them. This is done by preceding your "GO.SYSIN" card with the cards:

```
//GO.SYSLIN D D
//          DD  DSN=<dsname>,DISP=SHR,UNIT=...,VOL=...
//          D D DSN=<dsname>,DISP=SHR,UNIT=...,VOL=...
```

The first card must be included exactly as shown. The others consist of one card per I ibrary, where each <dsname> is the nam8 of a library, and the UNIT and VOL parameters must be filled in with the appropriate information, the same as they were when the library was created. (If you've "catalogued" the I ibrary, these parameters may be omi tted. The term "catalogued" refers to another feature of JCL, and we won't be covering it.)

In order to give an example of all this, we will of course have to include something you haven't learned yet, namely the use of separately-compiled procedures. Hopefully the syntax will be self-explanatory. While we're at it, we will include the use of the FORTRAN sine routine, 'DSIN'.

Example:

Here are -two complete jobs. They are printed here exactly as they looked when ue ran them, except for keyword cards! The first creates a library with two procedures in it; the second uses this library, and also a FORTRAN routine, DSIN. Notice in the second job that, when the I ibrary procedures are used, they are referred to by different names than the ones by which they are known in the library.

```
//MAKELIB JOB (J88$D2,302,.24),'MPL PROJECT'
// EXEC MPL
//COMP.SYSLIN D D DSN=WYL.D2.J88.LIBRARY,UNIT=DISK,VOL=SER=PUB002,
//  DISP=(NEW,KEEP)
//COMP.SYSIN D D *
PROCEDURE R := SQUARE (X) WHERE X I S REAL VALUE;
     R := X*X;
%MPL
PROCEDURE FROG:
|_   WRITE <<RIBBIT>>;
     WRI TE <<BREE-DEEP>>
_|.
/*
```

Some remarks before we present the second job. First, due to the limitations of JCL, and the length of our <dsname>, we had to put the DISP on a JCL "continuation" card. This is standard JCL, 50 as usual we wi I I not discuss it. Also note that we terminated the second procedure with a period from force of habit. This is acceptable,

```
//USELIB JOB (J88$D2,302,.24), 'MPL PROJECT'
// EXEC MPL
//COMP.SYSIN D D *
PROGRAM FROGGY; "It doesn't hurt to have a name on if."
```

```
PROCEDURE TOAD;        "Tell MPL that TOAD should refer to a library
     MPL <<FROG>>;     "'routine which is called FROG in the library,"

PROCEDURE S := SQ(Z) WHERE Z IS REAL VALUE:
     MPL <<SQUARE>>;

PROCEOURE S := SIN(X) WHERE X IS REAL-VALUE,
     FORTRAN <<DSIN>>;

LET PI := 3.14159265358979;
WRITE SIN(PI),SIN(PI/2),SIN(PI/4);
WRITE SQ(SIN(PI/4)); "Should produce 0.5"
TOAD:
WRITE SQ(.5)

END.
/*
//GO.SYSLIN  DD
//           DD  DSN=WYL.D2.J88.LIBRARY,DISP=SHR,UNIT=DISK,VOL=SER=PUB002
```

The GO.SYSIN card itself is not needed since no input data is called for. The output actually produced by the second program is shown here:

```
 3.487868498008863E-57          1.00000000000000          0.707106781186547
    0.499999999999939
RIBBIT
BREE-DEEP
     0.250000000000000
```

8.2: Compilation Parameters

This section should be somewhat easier to swallow than the preceding one, but everything said here is also for the most part specific to this particular MPL compiler and this particular IBM system.

To invoke any of the following options, modify your EXEC card to look like

```
// EXEC MPL,PARM.COMP='NOWARN,DECK'
```

where the "NOWARN,DECK" can be any number of parameters from the set given below. Each parameter comes in two flavors, on or off. For example, LIST causes your program to be printed as it gets compiled (this is the usual state of things), whereas NOLIST would suppress this listing, Each pair of parameter8 is discussed briefly in the paragraphs that follow. The default parameter (the one which gets assumed if you don't specify either one) in each pair is underlined,

WARN NOWARN

Selecting NOWARN will cause the compiler to suppress any warning messages, Error messages are st i ll printed. (Refer to Appendix B for a complete list of warning and error messages.)

LOAD NOLOAO

Selecting NOLOAD ui ll cause the 'object program', which is normally directed to the SYSLIN data set, not to be. This results in the program not being run, nor being written in the l ibrary. The NOLOAO opt ion therefore is not terribly useful, unless it is used in conjunction with the DECK parameter.

DECK NODECK

If DECK is selected, a copy of the object pro gam is punched onto cards via the SYSPUNCH data set, for uhlch a suitable J C_ card must be included.

NOLIST

If NOLIST is selected, the source program is not listed by the compiler.

NOSUBS

Selecting NOSUBS will cause MPL to forego al l subscript range-checking at run-t ime. This results in an increase in speed (how much of an increase depends on hou much subscripting you're doing) at the risk of having errors go possibly undetected. If you are doing a lot of subscripting, you might want to specify NOSUBS once your program has been completely debugged,

TEXT NOTEXT

Specifying NOTEXT wi l l cause the output from the parser (one of the intermediate stages of compilation) to be printed. This is usually useful only to the MPL project people (i.e., us) as an aid in debugging the compiler itself,

CODE NOCODE

Similar to the TEXT/NOTEXT options, except it is the object program which is printed.

SECTION III

In this Section we intend to describe everything about MPL which we have not previously discussed. Some of these feature9 have **not actual ly** been implemented (we'll mention which ones as we come to them) and are included here merely for completeness, since later versions of MPL may provide them.

The features presented in this Section are all highly advanced and specialized. In describing them to you, we will usually assume you already understand the underlying mathematical and/or programming concepts. If for some features you don't, just ignore those features. The thing to remember here in Section III is that none of these things are really necessary for you to be able to use NPL for large-scale problems. It's just that they are here in case you want them. If you don't happen to know what a partition matrix is, you're unlikely to care about how NPL represents one, so forget about it.

In addition, unlike the previous Sections, the various pieces of Section III are independent. You can read any part of it without having read the rest. (We do assume you've got a firm understanding of Sections I and II.) What we recommend you do is skim through Section III, just to see what is available. Then any time you decide you could make use of one of these features, go back and study the relevant portion.

1: SPECIAL DATA STRUCTURES

1.1 : Matrix Sets

The matrix set dimensionality refers to a vector or matrix each of whose components is itself a matrix, A matrix set may be subscripted in the usual manner to yield one of its submatrices, which may in turn be subscripted. A matrix set may *not* be subscripted using anything other than a scalar.

A one-dimensional matrix set is dimensioned (we're talking now about how you define it, not how you subscript it) using 2 positive integer vectors, which must be the same size. The i^{th} component matrix has a row size equal to the i^{th} element of the first vector and a column size equal to the i^{th} element of the second vector. Tuo-dimensional matrix sets are dimeneioned in a simi lar manner using 2' positive integer matrices of matching size.

All matrices In a given matrix set must have the same type. As always, the default type (if none is specified) is REAL,

Because each element of a matrix set is an individual matrix, it is illegal to have one appear on the left side of a defining assignment statement. This changes the dimensions of that element matrix, without affecting the other matrices in the matrix set, We'll include a case of this in the example below.

Example:

```
"Define a set of four real matrices with sizes 2 by 3,
"3 by 5, 4 by 2, and 5 by 2, respectively."
DEFINE MS MATRIX SET (2,...,5) BY (3,5,2,2);
READ MS(2); "Read9 a 3 by 5 real matrix"
MS(1) := MS(2)((1,3),(2,3,5));
"Sets up 2 by 3 submatrix of the 3 by 5 matrix."
MS(4) := TRANSPOSE (MS(1)*MS(2));
"The dimensions work out right; this is legal,"
MS(3) := MS(2) * MS(4); "Illegal (MS(3) is 4 by 2 but
                         "MS(2)*MS(4) is 3 by 2)"
DEFINE MS (3):= MS(2)*MS(4) "Legal, MS(3) is now
                             "3 by 2. MS(1), MS(2),
                             "& MS(4) are unchanged, "
```

There is no such thing as a 'vector set', Each component of a matrix set must be a two-dimensional object, although the matrix set itself may be one-dimensional.

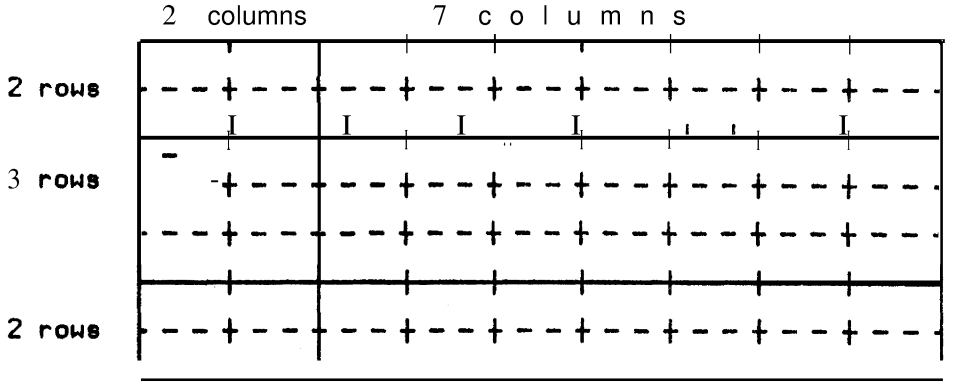
I.2: Partition Matrices

A partition matrix (also called a partition& matrix) is a matrix structure which is divided into a set of partitions both along rows and along columns. In the DEFINE statement, the dimensions of the partitions are given as two integer vectors. If we call these two vectors R and C, then the total number of partition9 in the partitioned matrix is SIZE(R)*SIZE(C), and the total size of the partitioned matrix is SUM(R) by SUN(C).

For example, the statement

```
DEFINE PM PARTITION MATRIX (2,3,2) BY (2,7)
```

would set up a 7 by 9 real matrix, partitioned thusly:



A partition matrix is subscripted using two subscripts, yielding a submatrix (not a scalar). In the above example, PM(1,2) would be a 2 by 7 real matrix, and PM(1,2)(1,1) would be a real scalar at the 1st row and 3rd column of PM.

A partition matrix may not be subscripted using vector subscripts. It may be subscripted using the * to yield an entire row or column partition. Thus PM(1,*) would be a 2 by 9 matrix, and PM(*,*) would be the entire structure as a 7 by 9 real matrix.

A partition matrix must not appear unsubscripted. To assign (for example) the values i*j to all components (i,j) of the whole 7 by 9 matrix above, we would use something like

```
PM(*,*) := COLUMN ((1,...,7)) * ROW ((1,...,9))
```

or

```
FOR i=(1,...,7), FOR j=(1,...,9), PM(*,*)(i, j) = i*j
```

1.3: Shape Attributes

Shape attributes have not been implemented in the current HPL compiler.

Only matrices may have shape attributes. Like the type and structure attributes, shape attributes are specified when a variable is defined, and may not be changed by later definitions, Mathematically, the shape attributes have no effect. However, by declaring a matrix to be, say, upper triangular, you are letting MPL know that it can save a lot of uork by assuming the lower portion of the matrix is zeroes. The attributes would carry over automatically where appropriate, so that if M1 and M2 were both defined to be upper triangular, then

```
DEFINE M3 = M1 * M2
```

would define M3 to be upper triangular as well. This sort of thing can lead to problems of demarcation, which is one reason why these features have not yet been implemented. For instance, is the sum of two sparse matrices sparse? The product? How about the inverse of a sparse matrix? Oh well, in case they ever become available, the shape attributes are:

```
RECTANGULAR
UPPER TRIANGULAR
LOWER TRIANGULAR
DIAGONAL
SPARSE
```

The default is RECTANGULAR. Remember that only matrices may be given shape attributes. Also note that, at present, attempting to use these features may lead to errors.

2: SPECIAL OPERATORS

2.1: MULT

Although logical matrices cannot be multiplied using the '*' operator, it is possible to perform the boolean equivalent of matrix multiplication, with logical 'and' taking the place of multiplication and modulo-2 addition† taking the place of normal addition. (Mathematically, this corresponds to GF(2) matrix multiplication.) This operation is represented by the keyword MULT, and is permitted only between two logical matrices of compatible sizes.

Example:

```
DEFINE L1 LOGICAL MATRIX 3 BY 4,
       L2 LOGICAL MATRIX 4 BY 5;

FOR i IN (1,...,3),
    FOR j IN (1,...,4),
        L1(i,j) := (j = 2) OR ((i+j)/2)*2 = (i+j);
        "I.e., either j is 2 or i+j is even."

FOR i IN (1,...,4),
    FOR j IN (1,...,5),
      L2(i,j) := (i+j) > 4 AND (i+j) < 8 ;
```

†Modulo-2 addition between two logical values p and q may be thought of as the 'exclusive-or' operation, which is defined as "(p AND NOT q) OR (q AND NOT p)", which is to say 'p or q but not both'.

```
DEFINE L3 := L1 MULT L2;  "L3 is now defined to be a
                           "3 by 5 logical matrix"

WRITE <<(3(4L2/)/4(5L2/)/3(5L2/))>>:L1, L2, L3
```

This would print three logical matrices, namely:

```
T T T F
F T F T
T T T F

F F F T T
F F T T T
F T T T F
T T T F F

F T F T F
T T F T T
F T F T F
```

2.2: IS NULL--'

The operator IS NULL may be used to test whether a vector is null, meaning it has a size of zero. Thus, "V IS NULL" is equivalent to "SIZE(V) = 0". It is simply an alternative and perhaps more natural way to write it. The result of the operation is a scalar logical value.

Examples:

```
{1,...,0} IS NULL   yields TRUE
{1,...,2} IS NULL   yields FALSE
{1,...,n} IS NULL   yields TRUE if and only if n ≤ 0
{0,...,-5} IS NULL  yields TRUE
```

2.3: Precedence

The complete precedence table for MPL operators, including those just described, is:

First; subscripting, result procedures
+ - (unary)
# |
**
* /
+ - (binary)
reiationele, IN, NOT IN, IS NULL
MULT
NOT
AND
Last: OR

3 : PROCEDURES (REVISITED)

There are all sorts of fancy features relating to procedures which we did not want to force upon you in Section II. They can be split into four more or less distinct categories. First, there's al I the stuff about parameter-passing conventions, of which the VALUE attribute is but one aspect. Second, there's the use of FORTRAN subroutines and separately-compiled I i brary procedures, which we al luded to at the end of Section II, Third, there is the concept of recursion, a powerful programming technique with which you may already be familiar. Last, there is the use of 'parametric procedures', meaning the use of the name of a procedure as an argument to another procedure, Not too surprisingly, we shall discuss these four categories 'in four separate sections.

3.1 : Parameter-Passing Conventions

We ui I I assume, since you have dared to begin reading this section, that you understand most if not al I of what we said about the VALUE attribute in Section II. In this section we will explain the alternatives to VALUE, what they are and what they do. I f you have already I earned this from some other programming language, such as ALGOL, you shou I d have very I i ttle trouble here.

In addition to the usual attributes (REAL, INTEGER, VECTOR, etc.), formal parameters being defined in a WHERE c I ause may be g i ven an attribute describing in some sense how the argument is to be treated. This extra attribute may be any one of the following:

VALUE
RESULT
VALUE RESULT
REFERENCE

If you don't specify any of these four attributes, the default is VALUE **RESULT** for scalar arguments and REFERENCE for non-scalars, We will now describe the effects of each of these four attributes.

3.1.1 : VALUE

We've described this before, so we'll just summarize the important aspects. If an argument has the VALUE attribute then, when you call the procedure, a copy is made of the actual parameter given in the call. If the procedure changes the value of the formal parameter, the actual parameter is unaffected. Thus the actual parameter is unchanged when you leave the procedure. (Of course, within the procedure, assigning a new value to the formal parameter does change the value of the formal parameter, It's just that, since the formal parameter is a copy of the actual parameter, the actual parameter is not affected,)

3.1.2: RESULT

This attribute indicates that the argument is being used only to pass a result back to the caller. This is not to be confused with the result variable of a procedure. The effect of using the **RESULT** attribute is that, when the procedure is called, the formal parameter is assigned no value whatsoever. That is, the actual parameter, whatever it may be, **is ignored** completely when the procedure is entered. When you exit from the procedure, then whatever value has been assigned to the formal parameter gets copied over into the actual parameter.

In order to prevent you from changing the values of constants, it is i i legal to use anything other than a legal <left side> as an actual parameter for a RESULT argument.

3.1.3: VALUE RESULT

The VALUE RESULT attribute combines the features of the VALUE and RESULT attributes. Thus, when the procedure is cai led, a copy is made of the actual parameter, Any new values assigned to the formal parameter ui thin the procedure affect only the formal parameter. But then, when the procedure f inishes, the current value of the formal parameter is copied back into the actual parameter. This corresponds exactly to what we said (back in Section II) would happen if you didn't specify VALUE. This shouidn' t come as much of a surprise to you, since we mentioned just a I ittle while ago that VALUERESULT is the default,

Because the VALUE RESULT attribute 'includes' the RESULT attribute, the same reetr ict ions apply to the actual parameters permi t ted to 'correspond to such an argument, namely only <left side>s.

3.1.4: REFERENCE

Arguments passed by REFERENCE are another story entirely. With all three of the other forms discussed above, the formal parameter was kept distinct from the actual parameter. The only differences among the three forms involved when values were copied back and forth between the two.

When an argument has the REFERENCE attribute, however, it is not copied. Any time the procedure uses or changes the formal parameter, it deals directly with actual parameter. (The term "reference" is used because the formal parameter, instead of being distinct from the actual parameter, merely refers to it.)

Example:

```
DEFINE k := 79;
INTEGERSIZE := 4;

PROCEDURE P1(x) WHERE x IS INTEGER VALUE;
|_   WRITE <<P1:>>, x, k;
     x := 1;
     WRITE << >>, x,   k
_|;

PROCEDURE P2(x) WHERE x IS INTEGER VALUE RESULT;
|-   WRITE <<P2:>>, x, k;
     x := 2;
     WRITE << >>,  x,  k
_|;

PROCEDURE P3(x) WHERE x IS INTEGER REFERENCE;
|_   WRITE <<P3:>>, x, k;
     x := 3;
     WRITE << >>, x,   k
_|;

WRITE <<*1*>>, k;
P1 (k);
WRITE <<*2*>>, k;
P2 (k);
WRITE <<*3*>>, k;
P3 (k);
WRITE <<*4*>>, k;
```

The character strings in the WRITE statements are there to make it easier to associate the various pieces of the output with the appropriate statements. The output is:

```
*1*  73
P1:  73    79
      1    73
*2*  73
P2:  79    73
      2    79
*3*   2
P3:   2     2
      3     3
*4*   3
```

In P1, the new value assigned to its argument, x, never causes a change in the value of the actual parameter, k. Thus k is still 79 at *2*. In P2, when x is assigned the value 2, it does not affect k, so the second WRITE

in P2 prints out 2 (for x) and 79 (for k). But when P2 finishes, k gets assigned the currentvalue of x, so 2 gets printed at *3*. Finally, in P3, when x is assigned the value 3, it immediately affects k, since x 'refers' directly to k. So the second WRITE in P3 prints 3 for both values. When P3 finishes, k is still 3.

MPL does not place any restrictions on what may be used as the actual parameter for a REFERENCE argument. Thus it is possible for you to destroy yourself, or at least your program, by passing constants to a procedure which declares the argument to be REFERENCE and then assigns it a new value.

Example:

```
PROCEDURE five (x) WHERE x IS INTEGER REFERENCE;
    x := 5;

CALL five (3); "Suddenly, '3' has the value 5!"
FOR k IN (1,...,3), WRITE 3*k;
```

This would print the sequence of numbers: 5, 10, 15, 20, 25, If you do this sort of thing deliberately, you forfeit all right to sympathy.

All non-scalar arguments default to REFERENCE, because making copies of them could significantly slow down the procedure. Thus, as we warned in Section II, you should be careful about using constants or expressions as non-scalar arguments in procedure calls, if the procedure is expected to assign new values to those arguments.

3.1.5: Functions

Functions are almost exactly the same as procedures. They are written the same way as are procedures, except the keyword PROCEDURE gets replaced by FUNCTION. The intention is that functions should be self-contained, just as mathematical functions have values dependent only upon their argument(s). A procedure is allowed to use and/or change the values of any variables in the program, including those defined outside of itself. Functions are allowed to use these 'global' values, although it is bad practice. But functions are *not* allowed to *change* the value of any variable which was defined outside the function, Furthermore, functions are not allowed to have RESULT or VALUE RESULT arguments. The default attribute for scalar arguments to functions is VALUE, unlike for procedures where it is VALUE RESULT.

The effect of all this is to insure that a function cannot affect the 'outside environment' in any way, except by returning a result (via a result variable, not via a RESULT argument), Actually, functions can also affect the calling program by doing I/O, since if it reads some input data then that data can no longer be read by the main program.

If a function has a REFERENCE argument which is the subject of a DEFINE or assignment statement within the function, you will get a warning at compile-time.

3.2: Separately Compiled Procedures

At the end of Section II, when we were showing you how to use MPL, we explained how you could set up libraries of routines, similar to the 'library functions' listed in Appendix A, which you could then invoke without having to include them in the calling programs. It is now time to show you how to write programs which use these library procedures. (We will call them library procedures to distinguish them from the library functions, although there is really no reason why you couldn't put functions into a library, too.)

3.2.1: General usage

In Section II we shoued you the 'JCL' which had to go outside your program in order to tell MPL where to find the libraries. Here we are going to show you what goes into the calling program, Essentially, you just have to define the procedure the same way you would if it were being included as part of the program, but instead of a 'procedure body' you use one of the statements

```
FORTRAN <<NAME>>
```

or

```
MPL <<NAME>>
```

where NAME is the name of the library procedure to be used. NAME need not be the same as the name of the procedure currently being defined, although in most cases it is less confusing if the names correspond. For example, there is a standard FORTRAN function called OSIN, which takes a double-precision real argument and returns its sine as a double-precision real value. (All REAL items in MPL are double-precision.) If ue wanted to use this function in an MPL program, we'd include this in our program:

```
FUNCTION r := sin(x) WHERE (r,x) ARE REAL SCALARS;
    FORTRAN <<DSIN>>
```

We could then use 'sin' like any other function, Two notes: First, the WHERE clause in the preceding example could have been omitted, since REAL SCALAR is the default. We do not need to specify x to be VALUE since that is the default for FUNCTIONs. Second, HPL automatically looks in the FORTRAN system library, so we wouldn't have to do anything special in the JCL in order to use the OSIN function.

Refer to Section II for more examples of the use of separately-compiled procedures,

3.2.2: Notes on FORTRAN linkage

Certain special considerations must be taken into account when using a FORTRAN library routine, due to the fact that FORTRAN and MPL have different internal representations for some forms of data. First off,

there is the question of what the MPL data types correspond to in FORTRAN, since in FORTRAN (at least, in the version of FORTRAN at our installation) there are different types of integer, real, and logical values. The following table shows the correspondence between MPL and FORTRAN data types, and also the internal form used by the two languages as implemented on the IBM system.

MPL	FORTRAN	Internal
INTEGER	INTEGER*4	Full-word (32-bit) fixed-point
REAL	REAL*8	Double-precision floating-point
LOGICAL	LOGICAL*1	One-byte (8-bit), = 0 or 1
CHARACTER	LOGICAL*1	One-byte EBCDIC character

The next table shows the permissible' correspondence between MPL and FORTRAN data structures. Note that there are no FORTRAN structures compatible with MPL matrix sets or partition matrices.

MPL	FORTRAN
SCALAR	scalar
VECTOR	1-dimensional array
ROW, COLUMN, MATRIX	2-dimensional array
ARRAY	3-dimensional array

If arrays are passed to FORTRAN routines, care should be taken in using the asterisk or index sets as subscripts. These may produce an array which is not stored in consecutive memory locations, a phenomenon which NPL is prepared for but which will usually cause a run-time error in a FORTRAN routine. If an array expression of this type is required, you should define a temporary variable to hold the value, and use the temporary variable in the FORTRAN call.

3.3: Recursion

There's a good chance you are already familiar with the technique of recursion, in which case all we have to tell you is that MPL procedures are automatically recursive. (Some languages, such as PL/1, require that you explicitly state when a procedure is intended to be used recursively.) If you don't already know what recursion is, read on.

3.3.1: Ordinary recursion

Essentially, a recursive procedure is simply a procedure which calls itself. This may sound like an undesirable state of affairs, since if a procedure calls itself, then this second call will eventually call itself again, and the third call will do a fourth call, and , . . it sounds as if we'd never return from any of these calls. Indeed, this problem sometimes occurs when a recursive procedure fails to work correctly, and the resultant 'runaway recursion is very much like an 'infinite loop'. However, when a recursive routine is working correctly, it does the

recursive call *conditionally*, so that sooner or later the recursion stops and things begin to return.

Although anything that can be done with recursion can also be done with ordinary loops, and vice versa, recursion is often a more natural way to represent what is going on. The traditional example of a recursive function is the computation of n factorial (1·2·3·....·n) using a recursive function. This seems a bit absurd, since we've already used this as an example of how to use loops, and loops seem to do the job quite nicely. However, to keep the purists happy, we present here a typical function for computing n factorial recursively. (Actually, by the ground rules for Section III, we can't assume you've read the section about functions, so we'll do it as a procedure.)

```
PROCEDURE nfact := factorial (n)
    WHERE n IS INTEGER VALUE, nfact IS INTEGER;
"We will assume n≥0. 0 factorial = 1 factorial = 1."
    IF n < 2,
        nfact := 1
    ELSE
        nfact := n * factorial (n-1);
```

To see how this works, let's trace through what happens when factorial is called with an argument of 3. Since n≥2, it tries to compute 3 * factorial (2). So all of a sudden we've called factorial a second time, this time with n=2. Again the ELSE-clause is performed, so we need to compute 2 * factorial (1). Here we go again; we've just called factorial a third time, with n=1. But this time n<2, so we merely set nfact=1. This result is then returned as the result of "factorial (1)", so the second call to factorial ends up setting nfact to be 2 * 1, or 2. This result is then returned as the value of "factorial (2)", whereupon the original call to factorial can use this value to compute its final result, 3 * 2, or 6.

Now that we've kept the purists happy, we're going to keep *us* happy by giving a more reasonable, if slightly more complicated, use of recursion. The application we're going to use is that of sorting a real vector such that the elements are in ascending order. There are myriad ways to do this, but one of the simpler (yet efficient) methods involves recursion. The main idea is that, if you have two vectors already sorted, it is very easy to combine them into a single sorted vector. You simply step through them simultaneously, at each step taking the smaller value and copying it from its vector into the combined one. (Actually, it is slightly misleading to say we step through the two vectors simultaneously, since at each step we move past the smaller number, while staying in the same place in the other vector. But the idea should be fairly obvious.) So what we intend to do is break the original vector into two roughly equal parts, sort each part using a recursive call, and finally merge the two sorted halves.

Here we have a procedure which sorts its argument. The sorting is done 'in place', in the sense that the actual parameter is changed. The condition for terminating the recursion is the vector having a size of 1, in which case it is already sorted. (Note: This algorithm is often written

such that it uill only sort vector6 whose sizes are powers of two, which makes breaking the vector into tuo part8 fairly straightforward. Due to the power of MPL's subscripting forms, we will not require this restriction.)

```
PROCEDURE sort (V) WHERE V IS REAL VECTOR;
"Since V is passed by reference, any change6 made to it
"will affect the actual parameter used in the call, The
"sorted vector is stored back in V."
|_  IF SIZE (V) = 1, RETURN;  "V is already sorted"
    DEFINE V' := V({1,3,...,SIZE(V)}),
         V'' := V({2,4,...,SIZE(V)});
    "V' and V'' partition the elements of V."
    sort (V');
    sort (V'');
    "The sublists are sorted. Now to merge them.
    "PV' will be an index for V', PV" for V''."
    DEFINE PV' := 1, PV'' := 1, flag LOGICAL;
    "We're going to step through the two smaller vectors,
    "copying elements into V.  We'll be done once we've
    "copied as many element9 as we had when we started."
    FOR PV := {1,...,SIZE(V)},
    |_  "We will set 'flag' to true iff the next element
        "to copy is coming from V'."
        IF PV' > SIZE(V'),
             flag := FALSE    "Nothing left in V'."
        ELSE IF PV'' > SIZE(V''),
             flag := TRUE     "Nothing left in V''."
        ELSE
             flag := (V'(PV') <= V''(PV''));
        IF flag,
        |_   V(PV) := V'(PV');
             PV' := PV'+1
        _|
        ELSE
        |_   V(PV) := V''(PV'');
             PV'' := PV''+1
        _|
    _|
_|;
```

You might want to try stepping through this procedure to see how it works on a small vector, perhaps about six elements,

3.3.2: Foruard procedures

There is a subtle problem which can arise uhen you use recursive procedures. As long as the only recursion involve9 procedures calling themselves, you're safe. But suppose you have two procedures, each of which calls the other? For example, suppose you want to program the following two logical functions over the non-negative integers:

```
p(k) = TRUE          if k=0
     = p(k/2)        if k>0 is even
     = q((k-1)/2)    if k>0 is odd

q(k) = FALSE         if k=0
     = q(k/2)        if k>0 is even
     = p((k-1)/2)    if k>0 is odd
```

(Another way to define these functions is: p(k) is true if and only if the binary representation of k has an even number of 1's, and q(k) is -p(k). But the first definition is more useful for this example.)

It is a fairly simple matter to write the above functions in MPL. A typical pair of procedure6 might be:

```
PROCEDURE res := p(k) WHERE k IS INTEGER VALUE, res LOGICAL:
"Assume k is non-negative"
    IF k=0,
        rss := TRUE
    ELSE IF k=(k/2)*2,
        res := p(k/2)
    ELSE
        res := q((k-1)/2);   "Could use q(k/2), since k is
                             "odd, hut this is clearer"

PROCEDURE res:=q(k) WHERE k IS INTEGER VALUE, res LOGICAL:
"Assume k i 9 non-negative"
    IF k-0,
        res := FALSE
    ELSE IF k=(k/2)*2,
        res := q(k/2)
    ELSE
        res := p((k-1)/2);
```

So what is the problem? The problem is that, when the MPL compiler encounters the first procedure, it sees a call to 'q', but it has not yet seen the definition of 'q'. Normally you would take care of this by putting the procedure 'q' ahead of procedure 'p', but this doesn't work here because 'q' uses 'p'. In effect, each of these two procedures must be defined ahead of the other,

To take care of situations like that just shown, MPL allows you to define what is known in the trade as a forward procedure. What it amounts to is that you may specify an identifier *which will later be defined as* a procedure, You may then use the identifier as though the procedure definition had already been given, and then at a later point in the block give the actual definition. (MPL will give an error message if the definition is never supplied,)

To define a forward procedure, use a DEFINE statement and give the identifier the attribute PROCEDURE (if the procedure will not have a result) or RESULT PROCEDURE (if it will), In the latter case, you should

also include **type** and structure attributes specifying the **type/dimensionality** of the anticipated **result.** (As always, the default is REAL SCALAR.) For example, the pair of procedures given above would be preceded by the line

```
DEFINE q LOGICAL RESULT PROCEDURE:
```

Note that the forward definition does not include any information about the arguments of the procedure.

3.4: Parametric Procedures

The idea behind parametric procedures is as simple as that underlying recursion, namely we wish to be able to make the name of a procedure be the argument to another procedure. This sort of thing can often be useful in writing procedures which do some sort of analysis of an arbitrary function. For example, suppose we were writing a numerical integration routine. It would be reasonable for the routine to have four arguments--the lower and upper limits of integration, the accuracy desired, and the function being integrated.

To make an argument be a parametric procedure, define it in the WHERE clause using one of the attributes PROCEDURE or RESULT PROCEDURE. Do not give it any other attributes (such as structure or parameter-passing type) unless it is a result procedure, in which case you must also specify the type of the result. You may then use the formal parameter exactly as you would any other procedure. If you specify RESULT PROCEDURE in the WHERE clause, the parameter is assumed to be a procedure which returns a result of the specified type, and it is called by its appearance in an expression. If you specify just PROCEDURE for the argument, it is called by a CALL statement (or any of the equivalent forms).

A numerical integration routine would be too complex to use as an example here. instead, we'll show how to write a procedure which gives tables of 'values of an arbitrary real function.

Example:

```
PROCEDURE tabulate (func, list)
     WHERE func IS REAL RESULT PROCEDURE,
          list IS REAL VECTOR:
"Produces a table giving the value of 'func' for each
"element of the 'list' vector, Since 'list' is real,
"we can't use a FOR loop on it directly. Also, while
"we're at it, we might as well print a fancy heading."
|_   WRITE <<(8X,A1,13X,4A1/2X,2(13('='),3X))>>;<<X>>,<<f (XI>>;
     FOR i IN [1,...,SIZE(list)],
          WRITE <<(E15.7,E16.7)>>; list(i), func(list(i))
_|;
```

A typical use of this procedure might look like this.

```
PROCEDURE r := sin(x) WHERE x IS REAL VALUE;
    FORTRAN <<DSIN>>;

tabulate (sin, {0,...,10}*0.1)
```

which would produce the output:

X	f(X)
0.0	0.0
0.1000000D 00	0.9983342D-01
0.2000000D 00	0.19866930 00
0.38000000 00	0.29552020 00
0.40000000 00	0.38941830 00
0.50000000 00	0.47942550 00
0.6000000D 00	0.56464250 00
0.70000000 00	0.64421770 00
0.80000000 00	0.71735610 00
0.90000000 00	0.78332690 00
0.1000000D 01	0.84147100 00

(The formats E15.7 and E16.7 were chosen to make the columns line up the way they do. If you're wondering why there is only one blank at the front of each line, you're probably forgetting about carriage control.)

Note: MPL library functions (as listed in Appendix A) may not be used as parametric procedures. You can get around this restriction if necessary by defining simple, procedures, such as

```
PROCEDURE M' := INVRS(M) WHERE (M,M') ARE MATRICES:
    M' := INVERSE(M);
```

You could then use INVRS as a parametric procedure.

4: MISCELLANEOUS FEATURES

In this final section we intend to describe a small pot-pourri of features which did not seem important enough to put in sections by themselves.

4.1: The EMPTY Specification

Suppose you have written a procedure which uses a global variable, M, and assumes it to be a real matrix, Suppose further that you have put this procedure near the front of your program, but it is not called until much later. Since you don't really need to have a matrix M until the procedure

is cal led, you'd like not to have to create the matrix unti I it is that time. Unfortunately, MPL insists that the matrix at least be defined when the procedure is being defined, so that MPL can tel I what sort of operations are being done.

What is needed here (or would at least be useful) is a means for clef ining a matrix without actual ly creating it, since creating it might take up lots of space in the computer. This can be done by using the special EMPTY keyword. It is used in place of the dimensioning information in the define statement, thusly:

```
DEFINE M REAL MATRIX EMPTY BY EMPTY
```

There is nothing unique about matrices in this regard. The EMPTY dimension may also be used uhen defining vectors or arrays. The effect of such a definition is to let MPL know what the type and structure of the variable will be, without actually creating the object or even setting aside space for it, Of course, before the variable is actually used in the course of program execution, it had better have been assigned a value.

4.2: The DYNAMIC Attribute

Specifying the attribute DYNAMIC for a non-scalar variable tells MPL that the dimensions of that variable are subject to change. MPL automatical ly assumes a variable is dynamic if it is defined using a defining assignment statement, or if it appears in a RELEASE statement (see section 4.4).

The only time you have to uorry about this feature is if you have a non-sea I ar var i able which i s being passed by REFERENCE to a procedure, and which w i I I have i ts size changed by the procedure. I f you have such a variable, and it does not appear in a defining assignment nor a RELEASE statement, you must define it with the attribute DYNAMIC (along with all the usual attributes).

4.3: The ABEND Statement

Executing an ABEND statement wi I I cause execution to terminate (as for a STOP statement) with a user abend code 63 and a core dump. This is not intended for normal use and is included here only for completeness of documentation, The ABEND statement has the form:

```
ABEND
```

Notes: As currently implemented, the FORTRAN monitor wi I I intercept the abend and ui I I terminate instead with FORTRAN error message IHC2401. Also, if the run-time option DUMP Is specified in the JCL and any error occurs, then the job will abend with code 101,

4.43 The RELEASE Statement

Normally, storage for variables defined within a block is released by exiting from the block, whereupon the storage may be used for other things. (Bear in mind that, for the foreseeable future, computers have only a finite amount of storage space!) It is possible to release storage explicitly in situations where block structure may prove inadequate.

The RELEASE statement is used to free the storage used by non-scalar variables without having to rely on block structure. (Scalar variables take up so little space there is no point in ever releasing their storage.) Each Variable specified in the RELEASE statement is made 'empty' and its storage is released for general use. The variable may not be used again without first defining it, such that it will have an explicit size and value,

The form of a RELEASE statement is

RELEASE <list of variables>

where each variable in the list may be any unsubscripted non-scalar variable. The items in the list are separated by commas. Matrix sets may appear either unsubscripted, in which case the entire matrix set is released, or subscripted with scalars, in which case only the selected element matrix is freed, i.e., the sizes of its dimensions are set equal to zero. Partition matrices may not appear at all in RELEASE statements.

A DEFINE statement, when executed, always implicitly releases any old values (of the pertinent variables) created by previous execution of it or any other DEFINE statement contained within the same block. Definitions outside the block will 'reappear' when the block is exited.

4.5: Program Efficiency

In the current version of the MPL compiler, certain constructs run significantly faster than others. Advanced programmers may wish to at least be aware of the more and less efficient constructs in MPL 80 as to be able to write programs which run as fast as possible. There is generally no sacrifice in readability, since the faster constructs tend to favor matrix operations and set generators.

4.5.1: Index sets

In FOR statements, set generators, and subscripts, index sets produce much faster code than do general vectors.

4.5.2: Storage allocation

The RELEASE statement and the DYNAMIC attribute are less efficient than the stack-oriented allocation method used by block structure.

4.5.3: Non-scalar operations

The use of bui l t-in array operations,' using index sets to s i m u l a t e l o o p s if necessary, is more efficient than coding FOR loops which manipulate scalar elements of non-scalar structures.

Example:

```
"To negate every other rou of a matrix M, do this;"
M({1,3,...,ROWSIZE(M)}, *) := -M({1,3,...,ROWSIZE(M)}, *);

"Don't use this (it's slower):"
FOR i IN {1,3,...,ROWSIZE(M)}, M(i,*) := -M(i,*);

"This is slowest of all:"
FOR i IN {1,3,...,ROWSIZE(M)}, FOR j IN {1,...,COLSIZE(M)},
    M(i,j) := -M(i,j);
```

Use common sense, though. If you have to bend over backwards just to use a matrix operation, it's probably not worth it. For example, any one of the following would compute the sum of the elements of a diagonal (square) matrix M, but the first method is the most efficient.

```
DEFINE total := 0.0;
FOR i IN {1,...,ROWSIZE(M)}, total := total + M(i,i);

"or.. . "

DEFINE total := SUM (M *
        COLUMN ({FOR i IN {1,...,ROWSIZE(M)}, 1.0}));

"or..,"

DEFINE total : = (ROW ({FOR i IN {1,...,ROWSIZE(M)}, 1.01)
        * (M * ONES (ROWSIZE(M), ROWSIZE(M)))) (1);
```

APPENDIX A

The following library functions are predefined outside the main block of a MPL program.

Function	Parameter (s)	Result
ABS (S)	Real or int. scalar	Absolute value (of same type)
TRUNCATE (X)	Real scalar	Integer obtained by truncating X toward zero
SUM (V)	Real or int. vector	Sum of the elements
MIN (V)	Real or int. vector	Smallest element
ARGMIN (V)	Real or int, vector	Index (subscript) of first occurrence of smallest element†
MAX (V)	Real or int. vector	Largest element
ARGMAX (V)	Real or int. vector	Index (subscript) of first occurrence of largest element†
TRANSPOSE (M)	Any type matrix	Transpose of matrix
INVERSE (M)	Real matrix	Inverse of matrix (error if M is non-square or singular)
IDENTITY (J)	Integer scalar	Real identity matrix of rank J
ONES (J,K)	Two integer scalars	Real J by K matrix of all ones
ZEROES (J,K)	Two integer scalars	Real J by K matrix of all zeroes
SIZE (V)	Any type vector	Integer number of elements
ROWSIZE (M)	Matrix or column	Number of rous
COLSIZE (M)	Matrix or row	Number of columns
VECTOR (RC)	Row or column	Vector with same type, size, and value as RC
ROW (V)	Any type vector	Equivalent row vector
COLUMN (V)	Any type vector	Equivalent column vector

†The functions ARGMIN and ARGMAX are a bit strange in that they return the index where the minimum or maximum component of a vector first occurs, except if the vector is generated by a set generator, in which case the

function returns the first index of the set generator loop which **generated** the component. If the argument to ARGMIN or ARGMAX is empty (null) then the result is zero. A null argument to MIN or MAX will yield an **arbitrary** garbage result.

Due to a compiler restriction, if the argument to ARGMIN or ARGMAX is a set generator, then the FOR-vector within the set **generator must contain** only strictly positive elements.

4. Examples:

```
WRITE ARGNIN ({FOR i IN {14,12,11,9},(i-10)**2});
"The above outputs '11', whereas;"
DEFINE V:= {FOR i IN {14,12,11,9}, (i-10)**2};
WRITE ARGNIN (V); "Outputs '3'"

LET N ={1,...,5}; LET f(i)= 2*i+1;
WRITE ARGNAX ({FOR i IN N: f(i)=2,i**2})    "Outputs '0'"
```

APPENDIX B

Much of what will be said in this Appendix is specific to the current MPL compiler. Future versions may vary in their detection and handling of error conditions.

Errors in your program may be detected at any one of three stages. If you write an invalid MPL statement, it will generally be detected when you try to compile your program. If this happens, NPL will not attempt to actually execute the program, but will merely scan it for further errors. Other types of errors, such as mismatched array sizes or division by zero, cannot be detected until the program is executed, since they may depend on data read in at that time. If such an error occurs, MPL reports it and immediately ceases execution, as though a STOP had been executed. In between these two phases is the time when MPL, having read your program, generates from it an equivalent sequence of 'machine-language' instructions, i.e. instructions simple enough for the computer to perform directly. Certain problems can occur here if your program is too large. We will discuss these three areas as they would occur in chronological order.

1: COMPILE-TIME ERRORS

Syntactic errors (errors in form), as well as most semantic errors (errors in meaning), are detected when the program is compiled, Compile-time errors messages are of the form:

```
***ERROR***   <error number>   <error message>
  NEAR COORDINATE   <coord>   FOUND NEAR "<tokens>"
```

The <error message> indicates the nature of the error and the <error number> provides an index for reference to the more detailed explanations given later in this Appendix. <coord> is the nearest source statement coordinate to the error (the coordinates are listed alongside your statements when the program is compiled), and <tokens> are the last two and current token being scanned by the compiler when the error occurred. (A <token> is anything which is a single 'entity' to MPL, such as a single scalar constant, or an identifier, or a semicolon,) If an error occurs inside a complex expression, the <tokens> displayed may be beyond the actual occurrence of the error. If the error is a scanner error (241 through 248) then the tokens listed will not include the item in error, but rather will be the tokens immediately preceding it in the program,

Compile-time error messages in NPL give the location of the error and the tokens being scanned at the time the error was detected. Most often, though, the error message will be printed after an intervening line of source program which itself contains no errors, The location of the error must be located by comparing the coordinate given in the error message with

the coordinates in the left-hand column of the listing, rather than assuming that the error always occurred in the statement above the error message. Also, the tokens printed out wi ll of ten be 're-worded' 'by NPL, such that they do not necessarily reflect the actual program listing. For example, MPL might say WRITE when your program said ANSWER, or it might print ".350000 E+01" when you wrote '3.5".

We ui l l now present a sample program listing, showing what some of these things actually look like. First, let's show you the program.

```
PROGRAM

FOR I IN (1.0, 1.5, 3.0, 7.9),
|_   DEFINE VECT := (I, I+1, I-3),
           X := VECT * VECT
     WRITE I;
     WRITE VEC, X
_|

END.
```

The above program has 3 errors. (You might want to try to find them yourself before we point them out. Go ahead; we'll wait,) First off, it attempts to use a real vector in a FOR statement. Second, the semicolon is missing from the end of the DEFINE statement (after the "VECT * VECT"). Third, VECT is misspelled in the second WRITE statement.

The next page shows what the program listing would look like, This is copied verbatim from an actual listing. Since you don't get bold-face in an actual output, the keywords below are printed in normal type.

You can see examples of the re-wording we mentioned. In the first error message, the error was detected when the comma was read, so the last few tokens were "7.9","}" , and ",". They were reported as ".789999CE+01", ")>" (equivalent to '}'), and ",". Don't worry about the 'C' in the real constant; random letters may turn up at times between the last digit and the 'E', but you can ignore them and you'll be all right.

A remark concerning the "NEST" column in the listing, As in ALGOL-W, i t keeps track of the number of unmatched BEGINs or BLOCKs you've, gone past. Thus, if you have what you thought were a matching pair of '|_ and '_|' markers, then the NEST value just ahead of the '|_' should equal the NEST value just after the '_|'. I f this i sn' t the case, i t means you've probably got an unmatched '|_' or '_|' somewhere between the ones in quest ion.

```
COORD NEST     SOURCE STATEMENT

00000  00 PROGRAM
00001  01
00001  01   FOR I IN {1.0, 1.5, 3.0, 7.9},
00001  01   |_   DEFINE VECT := {I, I+1, I-3},
       ***ERROR***    315    <RANGE VECTOR> NOT INTEGER VECTOR IN 'FOR'
         NEAR COORDINATE 1    FOUND NEAR "  .789999CE+01 )>  ,  "
00002  02             X := VECT * VECT
00002  02         WRITE I;
       ***ERROR***    368    SEMICOLON MISSING
         NEAR COORDINATE 2    FOUND NEAR "  * VECT WRITE "
00003  02  WRITE  VEC, X
       ***ERROR***    322    VEC IS UNDEFINED
         N E A R COORDINATE 3    FOUND NEAR "; WRITE VEC "
00003  02   _|
00004  01
00004  01   END.

SYMTAB    33 / 1742   QUADS    34 / 4573   BLKTAB   1 / 45

    3 ERRORS DETECTED

*** CODE GENERATION SUPPRESSEO DUE TO ERRORS ***

000.07 SECONDS COMPILE TINE
```

Certain types of errors' are particularly likely to lead to several other errors. Specifically, error8 in DEFINE statements will usually produce other errors later on because the compiler does not have correct information about the attributes of a variable. Likewise, errors in block structure or the grouping of statements (|_ . . . _|) may, as in ALGOL, result in incorrect variable scope and produce spurious error messages for undefined variables and bad attributes.

Example:

```
"Create a 10 by 10 integer identity matrix and a vector
"containing the first 10 squares.  This is a somewhat
"inefficient method, but it'll do for this example."

DEFINE M INTEGER MATRIX 10 BY 10,
     V INTEGER VECTOR 10;

FOR i IN {1,...,10},
|_   FOR j IN {1,...,10},
          IF i = j THEN
               M(i, j) := 1
          ELSE
               M(i, j) := 0;
     V(i) := i
_|
```

Because the IF statement is the only thing being done by the inner FOR loop, there was no need to enclose it in |_...._| marks, But suppose that, as you were keypunching the program, you forgot this and included the '_|' at the end, but didn't include a '|_' at the beginning, resulting in:

```
DEFINE M INTEGER MATRIX 10 BY 10,
       V INTEGER VECTOR 10;

FOR i IN (1,...,10),
|_   FOR j IN (1,...,10),
          IF i = j THEN
               M(i,j) := 1
          ELSE
               M(i,j) := 0
     _|;
     V(i) := i
_|
```

The extra '_|' now acts to terminate the *outer* FOR loop, so that when the statement

```
V(i) := i
```

is encountered, you wi ll get an error message due to "i" being undefined. Then the final '_|' will be reported as extraneous, whereas in reality the only actual error occurred somewhat earlier. Finding this sort of thing can be tricky sometimes, but you'l l get used to it. (Better yet, after a while you'll stop making this sort of error.)

1 f you run into one of these 'propagating' errors, you may want to try fixing the one major error and then re-running your program to see if there were any other errors which weren't caused by the blatant one. But don' t carry this practice too far! Many novice programmers get into the habit of fixing the first error they find, then running the program again. In genera l, this is a waste of time and money, since one listing wil l often reveal several errors. Unless you have reason to believe that most of your errors are being caused by one of the problems mentioned above, you should try to account for (and fix) every error before running your program again.

2: CODE-GENERATOR ERRORS

After the MPL compi ler has read through your program, it generates -'machine-language code which the computer will then execute. During this 'code-generation phase certain compi ler-specif ic limitations may be encountered, resulting in errors which are not real ly your fault. Since these errors are a bit more mysterious than those reported during the previous phase, us will explain them separately.

The errors under discussion are numbers 901 through 1004 (don' t worry, there are a lot of unused number8 in between).

Message 301 is caused by too many different constants appearing in a single block of your program (if you run into this error, refer to Section II for a description of block structure). To fix the problem, more blocks must be inserted to break up the one that is too large, This may be possible by simply changing a compound statement to a block, if this action does not destroy the scope of variables. (Remember that, when you exit from a block, all variables defined uithin that block are destroyed.)

Message 902 is similar to 901 and is caused by too many variables being defined within a single block.

Message 903, on the other hand, is caused by nesting blocks and procedures too deeply within other blocks and procedures. That is, if you define a procedure inside another procedure which is defined within yet another procedure which..., well, at some point the compiler can no longer keep track of how far down you've dived. Thus, when breaking a program into blocks and/or procedures (which are themselves blocks), it is better if you break it into a sequence of independent blocks, rather than setting up a complex hierarchy.

Message 904 is similar to 901 and is caused by too much code inside a single block.

Message 905 is caused by FOR statements being nested greater than 10 deep. That is, you have a FOR loop inside a FOR loop inside a FOR loop... If you *really* need this many loops inside one another, you can probably manage it by having 10 of them, where the innermost loop calls a procedure, The procedure can then do the remaining loops.

Message 306 is similar to 901 and is caused by too many statements in a single block. This is not quite the same as 904, since the latter depends on the complexity of the statements. 906 is caused by a fixed limit on the actual number of statements in a single block.

Messages 1001 through 1004 are compiler errors. That is to say, they should n' t happen. If one of them happens to you, please rerun the job with TEXT and CODE parameters (see Section 11.8 if you don't know how to do this), and specify a large line estimate on your JOB card. Then send the listing to the HPL project for analysis.

3: LIST OF ERROR MESSAGES

In this section we present a list of all MPL error messages, with the exception of those encountered at run-time. The messages are ordered according to their error numbers, and suggested corrective actions are given wherever the error message might not be explicit enough to isolate the problem.

A ⊗ after the error number indicates that the error is 'fatal', If such an error occurs, the compilation is terminated immediately after the error is reported.

A ⊖ means the expression being examined is flushed ui thout further scanning, Compilation continues following the end of the expression,

A • means the condition being reported is not an error, but merely a warning, Your program may run despite these problems, but you should look closely to make certain you intended to do whatever it was you did. One **warning which tends to** come up with annoying frequency is number 358, which occur6 whenever you convert a non-scalar integer value to real, or vice versa.

Port ions of error messages shown here in lower-case represent thi ngs which wi l l be filled in differently for different errors. **For** example, in error 306, the <token> will be whatever token was found at the beginning of the erroneous statement.

Messages from the input scanner:

241 ILLEGAL CONSTANT
242 INTEGER CONSTANT TOO LARGE
243 ILLEGAL IDENTIFIER
244 STRING CONSTANT LONGER THAN 256 CHARS
245 ⊗ HASH TABLE OVERFLOW (TOO MANY I DENTIFIERS)
You have too many different names in your program, **This has** nothing to do with block structure (error 902) and breaking your program into **smaller** blocks will not help, This error is unlikely to occur, since the l imit on the total number of identifier6 in a single progam is quite large, but if you run into i t you should try putting parts of your program into procedure6 and compi l ing them separate l y. See Section III **for** the use of separately-compiled procedures.
246 • **ILLEGAL CHARACTER ENCOUNTERED**
247 ILLEGAL REAL CONSTANT (TOO LARGE OR TOO SMALL)
248 ⊗ SYMBOL TABLE STRING SPACE OVERFLOW

Messages from LET processing:

250 **ILLEGAL LET DEFINITION SYNTAX**
251 ILLEGAL LET REFERENCE ARGUMENT LIST
252 **LET VARIABLE** PREVIOUSLY DEFINED
253 LET BODY SPACE OVERFLOW
A LET reference expanded to something too large for MPL to handle in one piece, Sorry,
254 LET RECURSION STACK **OVERFLOW**
A LET reference expanded into something which contained a LET reference which expanded into something which... If this goes on too far, MPL lose6 track of what it's supposed to be doing, and gives up, producing this error.
255 LET DEFINITION IS RECURSIVE
The <expression> in a LET statement included a use of the **LET** synonym being defined. Since, i f **this synonym** were ever used, i t could never be resolved, this error is reported.
256 MORE THAN 30 DUMMY PARAMETERS IN **LET DEF**
This is an arbitrary compiler-specific **restriction,** simi lar to that mentioned in Sectlon I.7 uith regard to I/O **lists** and DEFINE statements.

257 WRONG NUMBER OF ACTUAL PARMS IN LET CALL
258 LET APPEARS IN LET CALL OR INSIDE LET DEF
Although LET expressions are allowed to involve other LET synonyms, they must not have a right-hand side which contains the word LET.
253 ⊖ LET EXPANSION ERROR - EXPRESSION FLUSHED

Messages from DEFINE processing:

260 ATTRIBUTE NOT YET IMPLEMENTED
261 ILLEGAL ATTRIBUTE LIST SYNTAX
262 MULTIPLE DEFINITION
You tried-to redefine the type or structure of an identifier.
263 ILLEGAL OPERATOR/KEYWORD IN DEFINITION
264 'BY' MISSING AFTER MATRIX DOMAIN SPEC.
265 DOMAIN SPEC. MUST BE INTEGER SCALAR
266 NON-IDENTIFIER IN MULTIPLE DEFINE LIST
267 ILLEGAL MULTIPLE DEFINE LIST
268 ILLEGAL DEFINITION SYNTAX
269 MORE THAN 30 ITEMS IN MULTIPLE DEFINE LIST
270 ILLEGAL ASSIGNMENT DEFINE
271 DOMAIN OMITTED FROM ARRAY DEFINITION
272 DOMAIN SPECIFIED IN 'WHERE' CLAUSE OF PROC.
You mustn't try to specify the sizes of non-scalar arguments in a procedure's WHERE clause.
273 DOMAIN GIVEN IN FORMAL VALUE PROCEDURE DEFINE
The same as 272 but for the result variable.
274 • DEFINE OF REFERENCE PARAMETER IN FUNCTION
A parameter to a function (as opposed to a procedure, see Section III) was declared within the function as being passed by reference, making it possible for the function to change the value of the actual parameter. The function has now done a DEFINE involving that parameter, thereby changing its value in the calling program. Functions aren't supposed to change anything outside themselves, so this is flagged as a warning.

Messages from PROCEDURE processing:

280 NAME IN 'WHERE' CLAUSE NOT IN PROC HEAD
A procedure WHERE clause included. an identifier which was neither a formal parameter nor the result variable.
281 PRDCEDURE NAME MISSING
282 NAME APPEARS TWICE IN DUMMY PARM LIST
283 PROCEDURE RESULT VARIABLE NOT DEFINED
If the result variable of a procedure is non-scalar, its first use within the procedure must be in a DEFINE or assignment DEFINE statement, in order to establish the size of the variable.
284 ILLEGAL DUMMY PARAMETER LIST
285 SEMICOLON MISSING AFTER PROC HEAD
286 'END' EXPECTED
287 PROCEDURE NAME MULTIPLY DEFINED
288 ATTRIBUTES DO NOT BATCH THOSE FORWARD DEFINED
In a forward result procedure (Section III) the result variable

has different attributes from those given in the forward definition. Another possibility is that the forward definition did not specify RESULT and the actual definition has a result variable, or the other way around,

230 • ARRAY PARAMETER NOT SPECIFIED AS REFERENCE

Here the term 'array' refers. to any non-scalar parameter. If such a parameter is not passed by reference, MPL wi l l have to create a copy of it uhen the procedure is called, which wastes time and space. Always pass non-scalar arguments by reference (this is the default) unless you explicitly desire to change the formal parameter within the procedure without affecting the actual parameter included in the call.

291 • ASSIGNMENT TO REFERENCE PARAMETER IN FUNCTION

Similar to message number 274.

292 DUMMY RESULT NAME MISSING IN VALUE PROC DEF

233 ILLEGAL FORTRAN EXTERNAL PROCEDURE HEADER

294 FORTRAN FUNCTION MUST HAVE SCALAR RETURN VAL

FORTRAN functions invoked from an MPL program are not al lowed to return non-scalar results, since the internal representat ions of the structures are not compatible between the two languages.

295 ILLEGAL MPL EXTERNAL PROCEDURE HEADER

Messages from statement processing:

296 • EXTRA RIGHT PAREN - DELETED

297 ⊖ MORE THAN 30 ITEMS IN READ / WRITE LIST

298 ⊖ ILLEGAL CASE STATEMENT SYNTAX

299 CASE STMT INDEX EXPR NOT INTEGER SCALAR

300 ILLEGAL ITEM IN RELEASE STATEMENT LIST

301 MULTIPLY DEFINED LABEL

That ' s pronounced "mu l t i *plee*", not "mu l t i *plie*".

302 ILLEGAL LEFT SIDE OF ASSIGNMENT STATEMENT

303 ⊖ := EXPECTED AFTER IDENTIFIER

304 ILLEGAL DIMENSIONALITIES FOR ASSIGNMENT

305 ILLEGAL MIXED TYPES IN ASSIGNMENT

306 <token> IS ILLEGAL TO BEGIN STATEMENT

307 ILLEGAL TYPE/DIMEN IN <COND EXPR> / 'IF' STMT

The <condi t ion> in an IF statement was not a scalar logical express i on.

308 'THEN' IS MISSING AFTER 'IF'

309 <identifier> IS AN UNDEFINED 'GO TO' LABEL

310 GO TO STATEMENT DOES NOT REFER TO A LABEL

311 ILLEGAL FORMAT IN I/O STATEMENT

312 ILLEGAL READ / WRITE STATEMENT

313 ⊖ 'FOR' STATEMENT INDEX VARIABLE NOT IDENTIFIER

314 'IN' MISSING IN 'FOR' STMT

315 <RANGE VECTOR> NOT INTEGER VECTOR IN 'FOR'

316 ILLEGAL TYPE/DIMEN IN : CLAUSE AFTER 'FOR'

The condition given following the colon in a FOR statement (Section II) was not a scalar logical expression.

317 '00' MISSING IN 'FOR' / 'WHILE' STATEMENT

318 ASSIGNMENT OF PARTITION MATRIX OR ARRAY

319 <COND EXPR> NOT LOGICAL SCALAR IN 'WHILE'

320 ⊖ ILLEGAL RELEASE STATEMENT SYNTAX

Messages from **expression** processing:

321 OPERATOR HISSING
You wrote two operands with no operator **between** them.

322 <identifier> IS UNDEFINED

323 **<IDENTIFIER>(** WHERE <IDENTIFIER> IS-SCALAR
You tried to subscript a scalar. Just what did you have **in mind?**

324 **<operator>** IS ILLEGAL OPERATOR IN EXPRESSION

325 <operator> IS ILLEGAL BINARY OPERATOR

326 **<operator>** IS ILLEGAL UNARY OPERATOR

327 ILLEGAL DIMENSIONALITY FOR SUBSCRIPT LIST

328 ILLEGAL TYPE FOR SUBSCRIPT LIST

329 <operator> **:** ILLEGAL TYPE

330 <operator> **:** ILLEGAL DIMENSIONALITY

331 ILLEGAL **MPLBUILTIN** FUNCTION ARGUMENT
A library function was used with arguments **which** did not meet the criteria given in Appendix A.

332 EXPRESSION APPEARS IN READ LIST

333 ROW/COLUMN **USED** AS **1** BY 1 MATRIX FOR ***/** MULT

334 DIMENSIONALITIES DIFFER IN A COMPARISON

335 UNFORMATTED READ OF NON-NUMERIC VARIABLE

336 WRONG NUMBER OF SUBSCRIPTS IN LIST

337 ILLEGAL **'IS-NULL'/'** IS UNDEFINED'
The keyword UNDEFINED refers to a feature which never got implemented. Don't worry about it.

338 ILLEGAL TRANSFER FUNCTION ARGUMENT
You **gave** a non-vector as the argument to the ROW or COLUMN library function, or a non-row non-column to the VECTOR **function,**

339 ILLEGAL ITERATED VECTOR GENERATOR

340 ILLEGAL **TYPE/DIMEN** FOR UNARY -

341 ILLEGAL FUNCTION/PROCEDURE CALL

343 NULL EXPRESSION

344 **<name>:** FORWARD PROCEDURE NEVER DEFINED

346 <number> ACTUAL PARAMETER! MISMATCHED ACTUAL / FORMAL
The n^{th} actual parameter in a procedure call did not have the same type and structure as the **formal** parameter in the procedure.

347 ILLEGAL EXPRESSION SYNTAX

348 **ILLEGAL** COMMA

349 **ILLEGAL** **,...,** IN VECTOR EXPRESSION

350 ILLEGAL **TYPE/DIMEN** IN VECTOR EXPRESSION.
This is the error you'll get if you try to mix integer and **real** constants in a single N-tuple.

351 ILLEGAL ITERATIVE **VECTOR** EXPRESSION

352 ILLEGAL USE OF PARTITION MATRIX/ARRAY

353 EXPRESSION ENDS BADLY

354 OPERAND STACK OVERFLOW
The expression is too complicated for MPL to figure out in one piece. Break it up into two or more smaller expressions, using temporary variables if necessary.

355 OPERATOR STACK OVERFLOW
Similar to 354.

356 ILLEGAL **TYPE/DIMEN** FOR 'IN' / 'NOT IN'

357 PROPER PROCEDURE APPEARS IN EXPRESSION
You **tried** to call a non-result procedure as if it were a **result** procedure.

358 ∘ ARRAY TYPE CONVERSION REQUIRE0
You performed some operation, perhaps assignment, which requires treating an integer value as if it were real, or vice versa. MPL is merely warning you that it will have to convert every element of the non-scalar structure from integer to real, or vice versa, and that this will slow down your program, You may want to redefine some of your variables and/or change some constants from integer to real (or vice versa) in order to cut down on this source of inefficiency,

359 RIGHT PAREN MISSING

360 <number> ACTUAL PARAMETER: EXPR IS RESULT FORMAL PARM
This is the error you'll get If you fail to define a scalar argument with the VALUE attribute, and then cal I the procedure with an expression as the actual argument,

Messages from overall program processing:

361 'PROGRAM MISSING AT START

362 ∘ '.' ,MISSING AT END

363 'END' OMI TTED

364 ⊗ SYMBOL TABLE OVERFLOW
Similar to 245. Same measures should be taken.

366 ⊗ QUADRUPLE SPACE OVERFLOW
(MPL calls its intermediate form of code 'quadruples'.) This condition is similar to that of messages 364 and 245. The same corrective measures apply.

367 PROGRAM IS DUMMY EXTERNAL MPL PROCEDURE

368 SEMICOLON MISSING

369 SEGMENT TABLE OVERFLOW

370 BLOCKS NESTED TOO DEEP
Must be 7 levels or less, including the main program. Similar to message 903.

Code generator error messages:

See previous section for explanations,

901 ⊗ SEGMENT PROLOGUE > 4K

902 ⊗ DATA AREA OVERFLOW

903 ⊗ FUNCTION / PROCEDURE CALLS AND BLOCKS NESTED TOO OEEP

904 ⊗ PROGRAM SEGMENT > 8K

905 ⊗ FOR'S NESTED > 3.0 DEEP

906 ⊗ COORDINATE TABLE OVERFLOW

1001 @REGISTER ALLOCATION HANGUP (COMPILER ERROR SHOULD NEVER OCCUR)

1002 @REGISTER ALLOCATION HANGUP (COMPILER ERROR SHOULD NEVER OCCUR)

1003 @REGISTER ALLOCATION HANGUP (COMPILER ERROR SHOULO NEVER OCCUR)

1004 @REGISTER ALLOCATION HANGUP (COMPILER ERROR SHOULO NEVER OCCUR)

4: RUN-TIME ERRORS

Run-time errors in MPL may result from a variety of conditions which cannot be detected at compile-time, such as unequal sizes in a non-scalar assignment.

The information provided when a run-time error occurs includes the source coordinate of the error, the error message, and a traceback of procedure and block calls. Each location referred to in the traceback gives the source coordinate of the location and the name of the procedure (or "*BLOCK*" if it is contained by an explicit block, or "*MAIN*" if it is contained in the outer level of the main MPL program) that contains the location.

The following is hopefully a complete list of MPL run-time error messages and explanations. Note that the error messages tend to use the term "array" to mean any non-scalar object.

ARRAY SUBSCRIPTING

A subscript for a non-scalar object is out of range.

CASE SELECTION INDEXING

The integer expression of a CASE statement was outside the range {1,...,n}, where n is the number of statements among which the CASE statement was to have selected.

FORTRAN DETECTED ERROR

An error was found by the FORTRAN run-time monitor, which MPL uses for various tasks. The line immediately preceding the MPL error message should be an error message from the FORTRAN routine which detected the error condition. Refer to standard FORTRAN documentation if necessary to analyze the error,

ILLEGAL ARGUMENT FOR INVERSE

The argument matrix for the library function INVERSE was not square.

ILLEGAL ARRAY FOR FORTRAN

An array passed to an external FORTRAN subroutine was not in column-major order.

ILLEGAL ARRAY SIZE DEFINED

In a non-assignment DEFINE statement the size(s) were non-positive.

ILLEGAL DIMEN ARG MATRIX SET

In the definition of a matrix set the dimension arrays contained non-positive elements, were null, or differed in size.

ILLEGAL ITER VECTOR IN SUBARRAY

An index set used as a subscript had components that were out of range as legal subscripts. Also, if the object being subscripted is not a vector, the error may have been caused by the index set being nul l. (Only vectors may be subscripted by a nuli vector, yielding a nul l-vector result,)

ILLEGAL PART MATRIX DIM VECT

In the definition of a partition matrix one of the dimension vectors had non-positive elements or was null,

ILLEGAL SIZES FOR CONCAT

The row sizes for horizontal concatenation, or the column sizes for vertical concatsnation, did not match.

ILLEGAL SIZES IN MATRIX MULT

In a matrix multiplication or logical MULT operation the column size of the first operand did not equal the row size of the second.

INTEGER CONVERSION

An attempt was made to convert a real value to an integer, where the magnitude was greater than the largest integer $(2^{31}-1)$.

REDEFINE OF NON-DYNAMIC PARM

An actual parameter corresponding to a non-scalar parameter in a procedure, and which was assigned a new size inside the procedure, did not have the attribute DYNAMIC.

SINGULAR ARGUMENT FOR INVERSE

The matrix argument to the library function INVERSE was singular.

STORAGE OVERFLOW

The MPL program requires more main storage than is available for your job on the computer. You might be able to fix this by judicious use of block structure or RELEASE statements to free up storage once you're done with it.

UNDEFINED ARRAY

A non-scalar variable was referenced that had not been defined. This may occur when the result variable for a procedure is non-scalar and its first use inside the procedure is not in a DEFINE statement (to specify its size). Such an error may get detected at compile-time. Null vectors may also be reported as "undefined arrays" under certain circumstances.

UNEQUAL SIZE ARRAYS ASSIGNED

The size attributes of the left side of an assignment statement do not match those of the expression on the right. Note that in an assignment DEFINE the right side size(s) are copied to the left side, so that mismatching of this sort cannot occur.

UNEQUAL SIZE ARRAYS COMPARE0

The sizes of two non-scalars being compared differ.

UNEQUAL SIZE ARRAYS IN ARI TH

In an ari thmeti c operat i on for non-scalars that requires operands of equal size, the operands were not of equal size.

UNFORMATTED INPUT

The input cards for an unformatted READ or GIVEN statement contained something other than legal arithmetic constants, or a real value was encountered when an integer uas called for,

ZERO INCRMT IN ITER VECTOR

In an iterative vector (index set) the second element equal led the first.

APPENDIX C

The following tables indicate what types and structures are allowed with each binary operator. To use these tables to determine whether a given use of an operator is legal, first find the table for that operator. Then check the list of legal data types to make certain the data types you're using are permitted, Next, find the row corresponding to the data structure of the. left operand and the column corresponding to the right operand. The table entry for that row and column indicates the structure of the result.

If the entry is ⊗ it means that the operation is illegal for that combination of structures. There may also be a reference to one of the footnotes at the end of this Appendix.

Addition and Subtraction: + -

Legal types: integer, real

+ -	scalar	vector	row	column	matrix	array
scalar	scalar	vector	row	column	matrix	array
vector	vector	vector'	⊗	⊗	⊗	⊗
row	row	⊗	rou'	row'	row'	⊗
column	column	⊗	column'	column'	matrix'	⊗
matrix	matrix	⊗	matrix'	matrix*	matrix'	⊗
array	array	⊗	⊗	⊗	⊗	array'

Multiplication: *

Legal types: integer, real

*	scalar	vector	row	column	matrix	array
scalar	scalar	vector	row	column	matrix	array
vector	vector	scalar'	⊗	⊗	⊗	⊗
row	row	⊗	matrix[2]	scalar[2]	row[2]	⊗
column	column	⊗	matrix	matrix[2]	matrix[2]	⊗
matrix	matrix	⊗	matrix[2]	column[2]	matrix[2]	⊗
array	array	⊗	⊗	⊗	⊗	⊗

Division: /

Legal types: integer, real

/	scalar	vector	row	column	matrix	array
scalar	scalar	⊗	⊗	⊗	⊗	⊗
vector	vector	⊗	⊗	⊗	⊗	⊗
row	row	⊗	⊗	⊗	⊗	⊗
column	column	⊗	⊗	⊗	⊗	⊗
matrix	matrix	⊗	⊗	⊗	⊗	⊗
array	array	⊗	⊗	⊗	⊗	⊗

Horizontal Concatenation: |

Legal types: integer, real, logical, character

|	scalar	vector	row	column	matrix	array
scalar	⊗	⊗	⊗	⊗	⊗	⊗
vector	⊗	vector	matrix	matrix[3]	matrix[3]	⊗
row	⊗	matrix	matrix	matrix[3]	matrix[3]	⊗
co l umn	⊗	matrix[3]	matrix"	matrix[3]	matrix[3]	⊗
matrix	⊗	matrix[3]	matrix[3]	matrix[3]	matrix[3]	⊗
array	⊗	⊗	⊗	⊗	⊗	⊗

Vertical Concatenation: #

Legal types: integer, real, logical, character

#	scalar	vector	row	column	matrix	array
scalar	⊗	⊗	⊗	⊗	⊗	⊗
vector	⊗	matrix[1]	matrix[4]	matrix[4]	matrix[4]	⊗
row	⊗	matrix[4]	matrix[4]	matrix[4]	matrix[4]	⊗
co l umn	⊗	matrix[4]	matrix[4]	matrix	matrix[4]	⊗
matrix	⊗	matrix[4]	matrix[4]	matrix[4]	matrix[4]	⊗
array	⊗	⊗	⊗	⊗	⊗	⊗

Exponentistion: *

Legal types: Integer, real

*	scalar	vector	row	column	matrix	array
scalar	scalar	⊗	⊗	⊗	⊗	⊗
vector	⊗	⊗	⊗	⊗	⊗	⊗
row	⊗	⊗	⊗	⊗	⊗	⊗
column	⊗	⊗	⊗	⊗	⊗	⊗
matrix	⊗	⊗	⊗	⊗	⊗	⊗
array	⊗	⊗	⊗	⊗	⊗	⊗

Membership: IN, NOT IN

Legal types: integer, real (result always logical)

IN	scalar	vector	row	column	matrix	array
scalar	⊗	scalar	⊗	⊗	⊗	⊗
vector	⊗	⊗	⊗	⊗	⊗	⊗
row	⊗	⊗	⊗	⊗	⊗	⊗
column	⊗	⊗	⊗	⊗	⊗	⊗
matrix	⊗	⊗	⊗	⊗	⊗	⊗
array	⊗	⊗	⊗	⊗	⊗	⊗

Logical Multiplication: MULT

Legal types: logical

MULT	scalar	vector	row	column	matrix	array
scalar	scalar[5]	⊗	⊗	⊗	⊗	⊗
vector	⊗	⊗	⊗	⊗	⊗	⊗
row	⊗	⊗	matrix[2]	scalar[2]	row[2]	⊗
column	⊗	⊗	matrix	matrix'	matrix[2]	⊗
matrix	⊗	⊗	matrix[2]	column'	matrix[2]	⊗
array	⊗	⊗	⊗	⊗	⊗	⊗

Logical **Operations**: AND, OR

Legal types: **logical**

AND, OR	scalar	vector	row	column	matrix	array
scalar	scalar	⊗	⊗	⊗	⊗	⊗
vector	⊗	⊗	⊗	⊗	⊗	⊗
row	⊗	⊗	row[1]	⊗	⊗	⊗
column	⊗	⊗	⊗	column[1]	⊗	⊗
matrix	⊗	⊗	⊗	⊗	matrix[1]	⊗
array	⊗	⊗	⊗	⊗	⊗	⊗

Assignment: := (or =)

Legal types: integer, real, logical, character

:=	scalar	vector	row	column	matrix	array
scalar	legal	⊗	⊗	⊗	⊗	⊗
vector	legal	legal'	legal'	legal'	⊗	⊗
row	legal	legal'	legal'	legal'	legal'	⊗
column	legal	legal'	legal'	legal'	legal'	⊗
matrix	legal	⊗	legal'	legal'	legal'	⊗
array	legal	⊗	⊗	⊗	⊗	legal'

Notes:

'Size must match to be legal. E.g., row + column is legal only if both the row and column have size 1.

[2]Column size of first must equal row size of second,

[3]Row sizes must match.

[4]Column sizes must match.

[5]Equivalent to AND operation.

APPENDIX D

This is a complete list of MPL keywords, together with references to where in this manual they are described.

Notes:

'Reserved for feature not yet implemented.

[2]Reserved for feature not yet defined and thus not included in this manual.

[3]Obsolete; reserved to be compatible with previous versions of MPL.

HISTORICAL NOTE

Contributors to MPL

The development of the Mathematical Programming Language started in 1967 and went through a number of stage6 from initial specification to implementation via translators and compilers.

Those responsible for (or who participated in) the first specification of MPL were

Rudolf Bayer	Paul Davies	Stephen K. Schuch
James tl. Bigelow	David J. Gries	Christoph Witzgall
George B. Dantzig	Paul P. Pinsky	

The second specification of MPL was prepared by

Stanley Eisenstat	Vincent Nicholson
Thomas L. Magnanti	Christiana Riedl
Steven F. Maier	

A PL/1 translator of MPL to PL/1 was prepared by

Michael McGrath

Compiler specifications were prepared by

Theodore C. Tenny and Makoto Arieawa

This User's Guide is based on the MPL compiler designed and implemented by

Thomas S. Hedges

Some general suggestions with respect to the Hedges compiler were made by John Tom l in and James Kalan, and some of the final stages of debugging were done with the assistance of Donald R. Woods,

The User's Guide was written and produced by

Donald R. Woods

with technical advice from Thomas S. Hedges and editorial advice from George B. Dantzig, It was produced using the PUB document compiler and the Xerox Graphics Printer at the Stanford Artificial intelligence Laboratory.

Research supported by NSF grant DCR74-23014.