MPL
MATHEMATICAL PROGRAMMING LANGUAGE

BY

RUDOLF BAYER
JAMES H. BIGelow
GEORGE B. DANTZIG
DAVID J. GRIES

MICHAEL B. MCCCRATH
PAUL D. PINsky
STEPHEN K. SCHUCK
CHRISTOPH WITZGALL

TECHNICAL REPORT NO. CS 119
MAY 15, 1968

COMPUTER SCIENCE DEPARTMENT
School of Humanities and Sciences
STANFORD UNIVERSITY

This document has been approved for public release and sale; its distribution is unlimited.
MPL

MATHEMATICAL PROGRAMMING LANGUAGE

by

Rudolf Bayer
James H. Bigelow
George B. Dantzig
David J. Gries

Michael B. McGrath
Paul D. Finsky
Stephen K. Schuck
Christoph Witzgall

Computer Science Department
Stanford University
Stanford, California

Research partially supported by National Science Foundation Grant GK-6431;
AT(04-3) 326 PA #18; National Institutes of Health Grant GM 14789-01 A1;
The purpose of MPL is to provide a language for writing mathematical programming algorithms that will be easier to write, to read, and to modify than those written in currently available computer languages. It is believed that the writing, testing, and modification of codes for solving large-scale linear programs will be a less formidable undertaking once MPL becomes available. It is hoped that by the Fall of 1968, work on a compiler for MPL will be well underway.

The language proposed is standard mathematical notation. This, at least, has been the goal. Whether or not there is such a thing as a standard notation and whether or not MPL has attained it, is up to the reader to decide.
Mathematical programming codes for solving linear programming problems in industry and government are very complex. Although the simplex algorithm (which is at the heart) might be stated in less than twenty instructions nevertheless error checks, re-inversion, product-form inverses for compactness, compacting of data, special restart procedures, sensitivity analysis, and parametric variation are necessary for practical implementation. Twenty thousand instructions are not uncommon. The cost to program such a system is several hundreds of thousands of dollars.

Recently, there has been much interest in extending mathematical programming codes into the large-scale, nonlinear, and integer programming areas. The large-scale mathematical programming applications are among the largest mathematical systems ever considered for practical solution by man. For example, a system of close to a million variables and thirty five thousand variables has already been solved using the decomposition principle.

If large-scale dynamic linear programs could be successfully solved it would have enormous potential for industrial, national, and international long-range planning.

For this reason, there is considerable interest in solving large-scale dynamic systems. Many papers have been written on this subject and the number of theoretical proposals now number in the hundreds. Very little in the way of empirical tests have been made. Occasionally, a "soft-ware" company has dared to go from a theoretical proposal to a commercial program with inclusive results. It is like going from a drawing board to a battleship when all that has been built before has been a rowboat.

The need then is to be able to write elaborate codes for solving mathematical programming systems; to test them out on sample problems; and to compare them with competitive and modified codes. Present day computer languages like FURTRAN, ALGOL, PL/I are not in the same world as machine language of 0 1 bits. Nevertheless, it is a formidable undertaking to read codes in these languages, particularly when they involve some twenty thousand instructions. The finding of
errors (debugging) is time consuming. It is often difficult for the author of a program to decipher his own hieroglyphics assuming he is available for consultation. This difficulty becomes ever more acute when extended to proposals for solving large-scale systems. It is one of the chief stumbling blocks to progress in getting practical large-scale system codes.

For this reason, the chief effort of MPL has been directed towards readability. The objective is not to invent a powerful new language but to have a highly readable language, hence one easy to read, correct, and modify.

The Iverson Language is an example of a powerful language. With a small amount of effort it could have been set up in standard mathematical notation and made readable (to a non-expert) as well. It is probably possible to implement MPL by using Iverson Language as a translator. This is not our plan.

It is possible to view MPL as nothing more than a beefed-up ALGOL or FORTRAN. The new programming language PL/1 is very powerful and could also be used to realize MPL. This is being considered. Moreover, recently there have become available excellent compilers for compilers that make easier the job of developing a compiler that would directly translate MPL into machine language. We are seriously considering this as our approach for implementing MPL.
COMPARATIVE MATH VS MPL NOTATION

The short introduction (Part I) that follows is not a formal description of the language. This is done in Part III; nor is it a general manual as Part II; rather our purpose is to motivate the need for MPL and to provide a short comparison with standard mathematical notation. MPL notation assumes that a standard key-punch or its equivalent is all that is generally available at present for program preparation. This limits the alphabet to Capital Roman and replaces $A_{ij}$ by its functional equivalent $A(I,J)$.

<table>
<thead>
<tr>
<th><strong>MATH</strong></th>
<th><strong>MPL</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>SUBSCRIPTS:</strong></td>
<td>$A_{ij}$</td>
</tr>
<tr>
<td><strong>SUPERSCRIPTR:</strong></td>
<td>$A^k$</td>
</tr>
<tr>
<td><strong>MATRICES:</strong></td>
<td><strong>A</strong></td>
</tr>
<tr>
<td>Matrix Addition</td>
<td>$A+B$</td>
</tr>
<tr>
<td>Matrix Product</td>
<td>$AB$ or $A\cdot B$</td>
</tr>
<tr>
<td>Transpose</td>
<td>$A'$ or $A^T$</td>
</tr>
<tr>
<td>Inverse</td>
<td>$A^{-1}$</td>
</tr>
<tr>
<td>$A=$Matrix, $k=$Scalar, $L=$Scalar</td>
<td>$A/K$</td>
</tr>
<tr>
<td></td>
<td>$A*K$</td>
</tr>
<tr>
<td></td>
<td>$K*A$</td>
</tr>
<tr>
<td></td>
<td>$K*L$</td>
</tr>
<tr>
<td>Composing a matrix $M$ from submatrices $A$, $B$, $C$, $D$</td>
<td>$M=\begin{bmatrix} A &amp; B \ C &amp; D \end{bmatrix}$</td>
</tr>
<tr>
<td></td>
<td>$M := (A,B,C)$</td>
</tr>
<tr>
<td>Column of a matrix $A$</td>
<td>$A_{ij}$</td>
</tr>
<tr>
<td>Row of a matrix</td>
<td>$A_{i,j}$</td>
</tr>
<tr>
<td>Determinant</td>
<td>$</td>
</tr>
<tr>
<td>Array of Consecutive Integers</td>
<td>$(k, k+1, \ldots, l)$</td>
</tr>
</tbody>
</table>
### OPERATORS:

#### Matrices or Scalars:

<table>
<thead>
<tr>
<th>Math</th>
<th>MPL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Addition, Subtraction, Multiplication</td>
<td>+, -, *</td>
</tr>
<tr>
<td>Division by Scalar</td>
<td>A/K</td>
</tr>
<tr>
<td>Exponent</td>
<td>A²</td>
</tr>
<tr>
<td></td>
<td>A⁻¹</td>
</tr>
<tr>
<td>Sign</td>
<td>2, +2, -2</td>
</tr>
<tr>
<td>Substitution Operator(=)</td>
<td>New value of A=value of B+C</td>
</tr>
<tr>
<td></td>
<td>A := B+C; (meaning: change the value of A on LHS to equal the value of B+C on RHS.)</td>
</tr>
</tbody>
</table>

#### Logical Operators

<table>
<thead>
<tr>
<th>Math</th>
<th>MPL</th>
</tr>
</thead>
<tbody>
<tr>
<td>If A &gt; B, C &gt; D, and not D = 0</td>
<td>THEN</td>
</tr>
<tr>
<td>If A &gt; B OR C &gt; D</td>
<td>THEN</td>
</tr>
</tbody>
</table>

#### Relational Operators

<table>
<thead>
<tr>
<th>Math</th>
<th>MPL</th>
</tr>
</thead>
<tbody>
<tr>
<td>=, &lt;, &gt;, ≥, ≤, ≠, “,”</td>
<td>=, &lt;, &gt;, ≥, ≤, “,”, ≠, “”</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### Set Operators

<table>
<thead>
<tr>
<th>Math</th>
<th>MPL</th>
</tr>
</thead>
<tbody>
<tr>
<td>A U B, A+B</td>
<td>A OR B</td>
</tr>
<tr>
<td>A N B, A-B</td>
<td>A AND B</td>
</tr>
<tr>
<td>A N (not B),</td>
<td>A AND NOT B</td>
</tr>
</tbody>
</table>

### MAPPINGS, PROCEDURES, SUBROUTINES:

<table>
<thead>
<tr>
<th>Math</th>
<th>MPL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Y = F(X)</td>
<td>Y := F(X);</td>
</tr>
<tr>
<td>Y = SIN(X)</td>
<td>Y := SIN(X);</td>
</tr>
<tr>
<td>Y = 2EX²</td>
<td>Y := 2eBe(X+a2);</td>
</tr>
<tr>
<td>Y = B⁻¹</td>
<td>Y := INVERSE(B);</td>
</tr>
</tbody>
</table>
SYMBOL REPLACEMENT:

Let \( W = f(x,y) \)

(meaning do not compute \( W \) but replace it by \( f(x,y) \) wherever \( W \) appears later on.)

SETS:

(any set of elements)

\( S = \{1,3,-2,5\} \)

\( S = \{1,\ldots,n\} \)

\( I \in S \)

\( I \in A \cup B \cup C \)

\( I \in A \cap B \cap C \)

\( I \in A \cap B \)

\( D = (A \cup B) \cap C \)

Index Set or Domain of a vector \( A \)

Domain of \( A \)

Index Set of a matrix \( A \)

Row Domain of \( A \)

Defining of Set where \( P(I) \) a Boolean Expression or property is true

\( \{I \in \mathcal{R} : P(I) \text{ is true}\} \)

OR

\( \{I \in \mathcal{R} : P(I)\} \)

OR

\( \{I \in \mathcal{R} : A_i > 0\} \)

\( \{I : A_i > 0\} \)

Empty Set

\( \emptyset, \text{Null, Empty} \)

SET FUNCTIONS:

Suppose \( S = (S_1,\ldots,S_M) \) is a 1-dimensional array of integers and we wish to pick out column vectors \( A_{S_1}, A_{S_2},\ldots,A_{S_M} \)

to form a matrix \( B \).

\( B = (A_{S_1},A_{S_2},\ldots,A_{S_M}) \)

\( B := A(S); \)

\( B := (A(J) \text{ FOR } J \text{ IN } S); \)

\( B := (A(S(I)) \text{ FOR } I \text{ IN } (1,\ldots,M)); \)

However,

\( B := (A(S(I)),\ldots, A(S(M))) \) is not correct because \( (P,\ldots,Q) \) means \( (P,P+1,P+2,\ldots,Q) \) in MPL.
SYMBOLS:

CAPS
A, B, ---

Lower Case
a, b, ---

Greek
α, β, ---

Integers
0, 1,...,99, ---

Multi-Character Symbol:

as function name:
PİVOT(M,R,S)
SIN(X)

as variable name:

Brackets
{} []

MATH

MPL

A, B, ---

(not available yet)

(0, 1,...,99, ---)

PIVOT(M,R,S)
SIN(X)

B2, BASIS, X_S

(not available yet)
SYNTAX

In general, a procedure has the form:

```
PROCEDURE F(X,Y,Z)
Statement;
---
Statement;
FINI;
```

Certain reserve words like FOR and IN can be interspersed in place of commas in 
F(X,Y,Z) as in the example given below.

**Example**  Given an array of integers R, we wish to write an algorithm, called SUM, 
that yields

\[ S = \sum_{j \in R} F(j) \]

```
PROCEDURE SUM(F)
"SET UP A STORAGE REGISTER S TO ACCUMULATE
THE SUM OF TERMS. INITIALLY,"
(1): S := 0;
"LET S' BE THE UPDATED VALUE OF S. WE WANT
TO STORE S' IN THE SAME PLACE AS S AND
THEREAFTER CALL IT S."
(2): SAME LOCATION(S,S');
(3): S' := S + F(I) FOR I IN DOM(F); "ITERATIVELY ADDS F(I) TO S."
(4): SUM := S; "SETS THE VALUE OF THE FUNCTION EQUAL TO S"
(5): RETURN; "'RETURN' MEANS: RETURN TO MAIN ROUTINE."
FINI;
"'FINI' MEANS: END OF WRITE-UP."
```

Once the \( \sum \) symbol, or rather SUM, is in the procedure library we can use it to write a statement like

\[ P = \sum_{i=1}^{n} i^2 \]

in MPL.

\[ P := \text{SUM}(I**2 \text{ FOR } I \text{ IN } T) \text{ WHERE } T := (1, \ldots, N); \]
The reference numbers like (1), (2),... on the left are called labels. They are not necessary in the above example and may be omitted. Labels can be a string of characters or numbers like (1), (2). If the latter, they need not be consecutive. Labels are used to locate a statement when a program branches.

A statement like the one with label (3) is called a substitution statement because $S' := S + F(I)$; means: Substitute for the current value of $S'$ on the left a new value equal to the current value of $S + F(I)$ on the right.

In general, $A := B$; means updated $A = \text{Current } B$. A statement $S := S + F(I)$; looks like nonsense but means: Updated $S = \text{Current } (S + F(I))$. Hence a programmer not interested in readability would probably boil down the procedure SUM to two lines.

PROCEDURE SUM(?)

\[ \text{SUM := 0; SUM := SUM + F(J) FOR J IN DOM(F); RETURN; FINI;} \]
There are several different types of statements that one can draw upon to write a procedure:

- **Procedure Name**
- **If**
- **Define**
- **Substitution**
- **For**
- **Release**
- **Let**
- **Same Location**
- **Fini**
- **Return**
- **Go to**

and some words like "then", "otherwise", "endif", "do", endfor" that indicate different parts of a compound "if" or "for" statement.

### Procedure Name Statement:

**PROCEDURE F(X)**  
**PROCEDURE F("IN" X, "OUT" Y)**  
where X, Y represents a list of one or more symbols.

**Examples:**  
**PROCEDURE SIN(X)**  
**PROCEDURE PIWT(A,R,S)**  
**PROCEDURE SIMPLEX(A,B,C,BV)**  
**PROCEDURE ARGMIN(F(I) FOR I IN T)**  
"where ARGMIN finds the first index or argument where the minimum occurs."

### Substitution Statement:

A := Arithmetic Expression;  
S := 0; M := ARGMIN(H(J) FOR J IN R);  
A := PIWTA(R,S); G := INVERSE(MATRIX) + H;  
S := ARGMIN(C(J) FOR J IN T) WHERE T := (1,...,N);  

**Examples:**  
S := 0; M := ARGMIN(H(J) FOR J IN R);  
A := PIWTA(R,S); G := INVERSE(MATRIX) + H;  
S := ARGMIN(C(J) FOR J IN T) WHERE T := (1,...,N);  

### Let Statement:

LET A := Arithmetic Expression;  
LET A := B;  
LET T := (I IN DOM(B) | A(I,S) > 0);  
LET R := ARGMIN(B(I)/A(I,S) FOR I IN T);
If LET is used to simplify only one statement, a WHERE can be used instead using inverse order.

\[ G := \text{INVERSE}(B) \text{ WHERE } B := \text{TRANSPOSE}(A); \]

Return Statement:

RETURN;

If this statement is reached during execution of the subroutine, the next step is to return to the main routine.

If Statement:

IF \( P \) THEN statement;...; statement;
OTHERWISE statement;...; statement;
ENDIF;

Example:

IF \( R \) = NULL THEN GO TO (21); OTHERWISE
\[ A := \text{PIVOT}(A,R,S); \text{ ENDIF;} \]

All statements up to "OTHERWISE" are executed if proposition \( p \) is true and then sequence control skips to the statement following ENDIF. However, as in the above example, there is a GO TO statement preceding the OTHERWISE then control skips to wherever GO TO directs. If \( p \) is not true, control skips to statements following "OTHERWISE". For the case of several parallel conditional statements OR IF statements are available - see Part II and III. OTHERWISE can be omitted if immediately followed by ENDIF.

For Statement:

FOR I IN T DO statement;...; statement; ENDFOR;

Example:

FOR I IN (1,...,M) DO
\[ S' := S + F(I); \]
\[ T' := S' + G(I); \]
ENDFOR;
Same Location Statement: SAME LOCATION(A,B);

A and B will be assigned the same set of storage locations in the computer. An alternative way to accomplish the same thing would be to write: LET A :: B; For psychological reasons, it seems best to separate the concept: "A is another symbol for B" from the concept "same storage location".

Go to Statement:

GO TO & (where & is a label). This means that control is to skip to the statement that has & as a label.

Define Statement:

Example: DEFINE B DIAGONAL M BY M;

Used to define the size of storage array needed for a symbol whose value will be computed piecemeal later on.

Release Statement:

To release a symbol and its storage assignment a release statement takes the form:

RELEASE A,B;

Its purpose is to conserve storage and permit re-use of the same symbol for some other purpose. A special type of automatic release is available that allows release of all symbols in a block of code.

Release occurs automatically when a procedure returns to a main routine; all symbols defined in the procedure and their storage are released except the output symbols, which are treated as part of the symbols of the main routine.

Symbols used as dummy as G in the statement: Z :: A+G WHERE G :: INVERSE(M);
are treated as local to the statement and are immediately released. The same applies
to the running index in a compound For statement and to a dummy parameter in a Let
statement as I in: LET G(I) := B(I)/A(I,J);.
EXAMPLE: SIMPLEX ALGORITHM

PROCEDURE SIMPLEX("IN" A,B,C,BV,"OUT" BV', Z', Z', CASE);

"WARNING: ALL INPUTS ARE MODIFIED IN THE COURSE OF CALCULATIONS."

"THE PROBLEM IS TO FIND MIN Z, X ≥ 0 SUCH THAT:

\[ AX = B, \quad CX = Z. \]

IT IS ASSUMED THAT:

- A IS IN CANONICAL FORM WITH RESPECT TO BV
- \( B > 0 \) ARE THE X VALUES OF BV, I.E. \( X(BV) = B \).
- THIS INITIAL BASIC SOLUTION IS REQUIRED TO BE FEASIBLE, I.E. \( B > 0 \).
- BV' IS THE OPTIMAL SET OF BASIC VARIABLES.
- B' ARE THE X VALUES OF BV', I.E. \( X(BV') = B' \).
- Z' = MIN Z
- CASE = FINITE OR UNBOUNDED.
- BV', B', Z' REFER TO LAST BASIC SOLUTION IN THE CASE THAT 'CASE = UNBOUNDED'."

"INITIALIZATION"

DEFINE CASE CHARACTER;

(1): Z := 0; "PRIMES WILL BE USED FOR UPDATED VALUES OF VARIOUS SYMBOLS.
THES WILL BE STORED IN THE SAME LOCATION."

(2): SAME LOCATION (A, A'), (B, B'), (C, C'), (BV, BV'), (X, X'), (Z, Z');
"ITERATIVE LOOP"
"LET S BE COLUMN COMING INTO BASIS."
(3): \texttt{MIN}_1("IN" C, "OUT" S, c_S); \\
\texttt{"MIN}_1 \texttt{IS A FUNCTION THAT RETURNS THE INDEX AND THE}
\texttt{MINIMUM COMPONENT OF A VECTOR. IN THIS CASE VECTOR = C."}
\texttt{"WE NOW TEST WHETHER X(Bv) = B IS OPTIMAL."}

(4): IF c_S = 0 THEN CASE := 'FINITE'; RETURN; OTHERWISE
\texttt{"LET R BE THE INDEX OF THE BASIC VARIABLE DROPPING."}

(5): \texttt{MIN}_1("IN"(B(I)/A(I,S) FOR I IN \texttt{DOM}(B)|A(I,S) > 0), "OUT" R,Q); \\
\texttt{"IF ABOVE SET EMPTY, \texttt{MIN}_1 \texttt{RETURNS R = NULL, Q = 0:}
\texttt{OTHERWISE THE INDEX R AND THE MINIMUM RATIO, CALLED}
\texttt{Q, IS RETURNED."}

(6): IF R = NULL THEN CASE := 'UNBOUNDED'; RETURN; ENDIF; \\
\texttt{"UPDATE EVERYTHING BY PIVOTING ON A(R,S), PRIMES WILL
BE USED FOR UPDATED SYMBOLS. THESE ARE STORED IN SAME}
\texttt{LOCATION, SEE (2)."}

(7): B'(R) := Q;

(8): A'(R,) := A(R,*)/A(R,S);

\texttt{"ROW\_DOM(B) IS THE DOMAIN OF INDICES FOR B."}

(9) FOR I IN \texttt{ROW\_DOM}(B) | I ≠ R DO 

(10): B'(I) := B(I) - A(I,S) * Q;

(11): A'(I,*) := A(I,*) - A(I,S) * A'(R,*) ENDFOR;

(12): C' := C - C(S) * A'(R,*)

(13): Z' := Z + C(S) * Q ;
(14): \( BV'(R) := S; \)

"THE REMAINING COMPONENTS OF BV ARE UNCHANGED AND
SINCE BV AND BV' ARE STORED IN THE SAME LOCATION.
UPDATING IS COMPLETE, RECYCLE."

(15): GO TO (3); FINI;
The Manual to MPL comes in three parts

PART I: A SHORT INTRODUCTION
PART II: GENERAL DESCRIPTION
PART III: FORMAL DEFINITION
The objective is to develop a **readable** language for writing experimental codes to solve large-scale mathematical programming systems. Readability is defined as standard mathematical notation with minor adjustments reflecting current limitations of input-output equipment. Thus symbols are restricted to those found on a standard keypunch; subscripts (or superscripts) like $A_{i,j}$ appear as $A(i,j)$. Starting in the Spring of 1967, several test algorithms written in the proposed language gave evidence that readability was an achievable objective.

A task group in the latter part of 1967 began to define the proposed language in BACKUS Normal Form with the intent of using a special compiler's compiler to implement the language.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.0</td>
<td>INTRODUCTION</td>
<td>2/1</td>
</tr>
<tr>
<td>2.0</td>
<td>MPL LANGUAGE ELEMENTS</td>
<td>2/3</td>
</tr>
<tr>
<td>2.1</td>
<td>VARIABLES</td>
<td>2/3</td>
</tr>
<tr>
<td>2.2</td>
<td>CONSTANTS</td>
<td>2/5</td>
</tr>
<tr>
<td>2.3</td>
<td>OPERATORS</td>
<td>2/7</td>
</tr>
<tr>
<td>2.4</td>
<td>RESERVED WORDS</td>
<td>2/9</td>
</tr>
<tr>
<td>2.5</td>
<td>COMMENT STATEMENTS</td>
<td>2/9</td>
</tr>
<tr>
<td>3.0</td>
<td>EXPRESSIONS</td>
<td>2/11</td>
</tr>
<tr>
<td>3.1</td>
<td>LOGICAL EXPRESSIONS</td>
<td>2/11</td>
</tr>
<tr>
<td>3.2</td>
<td>ARITHMETIC EXPRESSIONS</td>
<td>2/12</td>
</tr>
<tr>
<td>3.2.1</td>
<td>COMPUTATIONAL EXPRESSIONS</td>
<td>2/12</td>
</tr>
<tr>
<td>3.2.2</td>
<td>FUNCTION REFERENCES</td>
<td>2/13</td>
</tr>
<tr>
<td>3.2.3</td>
<td>ARRAY BUILDERS</td>
<td>2/13</td>
</tr>
<tr>
<td>4.0</td>
<td>STATEMENTS</td>
<td>2/16</td>
</tr>
<tr>
<td>4.1</td>
<td>LABELED STATEMENTS</td>
<td>2/16</td>
</tr>
<tr>
<td>4.2</td>
<td>UNLABELED STATEMENTS</td>
<td>2/16</td>
</tr>
<tr>
<td>4.2.1</td>
<td>ASSIGNMENT STATEMENT</td>
<td>2/16</td>
</tr>
<tr>
<td>4.2.2</td>
<td>PROCEDURE CALL STATEMENT</td>
<td>2/17</td>
</tr>
<tr>
<td>4.2.3</td>
<td>KEYWORD STATEMENTS</td>
<td>2/17</td>
</tr>
<tr>
<td>4.2.3.1</td>
<td>GO TO STATEMENT</td>
<td>2/17</td>
</tr>
<tr>
<td>4.2.3.2</td>
<td>CONDITIONAL STATEMENTS</td>
<td>2/17</td>
</tr>
<tr>
<td>4.2.3.3</td>
<td>ITERATED STATEMENT</td>
<td>2/18</td>
</tr>
<tr>
<td>4.2.3.4</td>
<td>LET STATEMENT</td>
<td>2/19</td>
</tr>
<tr>
<td>4.2.3.5</td>
<td>DEFINE STATEMENT</td>
<td>2/20</td>
</tr>
</tbody>
</table>
5.0 STATEMENT BLOCKS
5.1 PROCEDURE BLOCKS
5.2 STORAGE ALLOCATION BLOCKS
5.3 ITERATION BLOCKS
5.4 CONDITIONED BLOCKS

6.0 EXAMPLES OF NFL PROCEDURES
1.0 INTRODUCTION

This paper describes recent work on a computer programming language for
the implementation of mathematical programming algorithms on a digital computer.
The objectives of the language are:

a) to facilitate programming an algorithm from
theoretical form to computer code in as short
a time as possible, and

b) to enable other mathematical programmers to
understand and modify an existing code with
a minimum of effort. The present efforts are
being directed toward the coding of experimental
mathematical programming algorithms rather than
commercial techniques. By and large, the first
report (Mathematical Programming Language, June
1967) represented the thinking of persons with
mathematical programming backgrounds. Since
then, several computer scientists contributing
to the project have brought the language much
closer to implementation.

The purpose of this report is to explain the use and the reasons for the
congepts being developed in MPL. This part of the Manual attempts to explain the
reasons for using the specific concepts of MPL while the third part developed
under the guidance of David Gries gives a formal definition of the language in a
modified form of BACKUS Normal Form. Part III is primarily the work of Stephen
Schuck, who, since joining the project last summer, has been a driving force behind
the implementation of MPL. His work in turn uses several concepts developed by
Rudolf Bayer and Christoph Wittigall of the Boeing Scientific Research Laboratories.
At present, the BACKUS Normal Form is used to describe the legal programs, not the
phrase structure of the language.

David Gries of Stanford University is currently developing a technique of
writing compilers, called the Komplir Implementation System (KIS), which, it is
planned will be used in the implementation of the Language. Many of the concepts
presented herein, are the same as or similar to those found in existing compiler languages (ALGOL, FORTRAN, COBOL, PL/1, etc.). One of the difficulties encountered thus far in writing a formal definition of MPL is that mathematical notation depends upon the context for its meaning. $(P_1, \ldots, P_M)$ may mean $(P_1, P_2, P_3, \ldots, P_M)$ or it may mean $(P_1, P_1 + 1, P_1 + 2, \ldots, P_M)$. This is defined in MPL to mean the latter.

There are certain concepts planned for MPL that have not yet been set down in BACKUS Normal Form. In particular, the representation of index sets has not been completely formalized; the ability to operate with matrices whose elements are matrices (useful for example in the decomposition principle) has not yet been fully developed. Procedure parameters need more work. Input-output statements have not yet been defined, nor storage commands that would reflect the variable size and speed of different memory locations.
2.0  **MPL LANGUAGE ELEMENTS**

The set of characters upon which MPL is built is the character set found on standard key-punches (such as the IBM 029 key-punch). For convenience, we shall group these characters into the categories of letters, digits, and special characters. The letters are A through Z, the digits are 0 through 9, and the special characters are as follows:

```
() <> , . +- * / : ; - = ' " # @ & % 
```

and a blank. Elements of MPL are defined to be one of the following four constructs—variable, constant, operator, or reserved word. Let us now delve more deeply into each of the above elements.

2.1  **VARIABLES**

Variables are symbols which represent those data values which may change during the execution of the program. There are several types of variables—arithmetic, logical, set and character.

For example, if C is a row vector and Q a scalar both previously defined then

```
D := (C, SIN(Q));
```

sets up a new row vector D with one more component than C. The function SIN(x) is a reserve word and "sin" cannot be used as symbol for a variable on the left hand side of an equation.

A variable may have zero, one, or two dimensions. A zero-dimensional variable is a scalar, a one-dimensional variable a vector, a two-dimensional variable a matrix.
In the remainder of this report, an array refers to any variable whose dimension is greater than zero. Each matrix has associated with it a structure shape commonly used in mathematical programming algorithms. These shapes are rectangular, diagonal, upper triangular, lower triangular, and sparse (meaning few non-zero elements). The concept of structure shape is useful in conserving memory space and execution time. An example of the use of shape matrices is in the storage and multiplication of two diagonal matrices of size \( n \times n \). Storing them as diagonal in the computer requires only \( n \) memory words for each (as opposed to \( n^2 \) for a rectangular matrix), and the multiplication of two diagonal matrices requires only \( n \) elementary multiplications as opposed to \( n^3 \) for rectangular matrices.

Vectors have the shape of row or column; this distinction is required for multiplying vectors by vectors or matrices. An additional feature of MPL is that the elements of an array may be arrays. This construct is helpful in coding algorithms such as the decomposition principle. Another variable allowed is an index set variable. This consists of an ordered set of integers. Examples of index sets are:

\[
(1, \ldots, M) \\
\text{SET}(1, 3, -4, 3, 12) \\
(I \text{ IN } (1, \ldots, M) \mid A(I, S) > 0)
\]

More will be said about how to define and use variables later on.

The symbols which constitute variables have two parts, the variable name and an optional subscript. The variable name alone completely identifies the variable under consideration if that variable is a scalar or an entire storage structure (vector, matrix, etc.). If the variable represents a subset (element, row, column, etc.) of a larger array, the variable-name part only identifies the larger array, subscripts being needed to specify the particular subset. Variable names always begin with a letter, but the characters which follow it may be any number of letters.
digits, or underscores. Reserved words (defined in Section 2.4) may not be used as variable names.

Examples of variable names are

A
OBJECTIVE
KEY_SET
BASIS_INVERSE

However, variable names with blanks like KEY_SET are not allowed. Subscripts are either scalar arithmetic expressions or the symbol * . Scalar arithmetic expressions (defined in Section 3.2) are automatically bounded to the nearest integer value when used as a subscript. The subscript * refers to an entire dimension of a storage structure. Thus

A(*, J) refers to the $j^{th}$ column while
A(I, *) refers to the $i^{th}$ row of the matrix A.

The following examples illustrate the use of subscripts:

M(B + C, 3)
B_INVERSE(1, *)
X_VALUE(BASIS_LIST(I)).

2.2 CONSTANTS

Constants are of four types—arithmetic, logical, set and character. The type of a constant determines how the number will be stored in the machine and used in calculations.
ARITHMETIC CONSTANTS may be either integer or real.

INTEGER ARITHMETIC CONSTANTS are written as a string of digits without a decimal point, examples 1, 10, 10090.

REAL ARITHMETIC CONSTANTS may or may not have an exponent. An exponentless real number is a sequence of digits containing a decimal point. Examples: 1., 1.0, .3925, 102.34. The exponent form of the real constant allows writing the constant in modified scientific notation. This form consists of an exponentless real number followed by an \( \times 10^p \) (meaning \( 10^{\text{the power}} \)) followed by an optionally signed string of digits.

Examples:

\[
\begin{align*}
2.5E02 & \quad (25 \times 10^2 = 2500.) \\
1.0E-02 & \quad (1.0 \times 10^{-2} = .01) \\
.8E03 & \quad (.8 \times 10^3 = 800.) \\
9.1E+05 & \quad (9.1 \times 10^5 = 910000.)
\end{align*}
\]

LOGICAL CONSTANTS are TRUE and FALSE.

A SET CONSTANT is NULL.

CHARACTER CONSTANTS are any string of characters enclosed by single quotes (').

Examples:

'TABLEAU'

'PRICES ARE'
2.3 OPERATORS

Operators are the connecting elements which allow the grouping of variables and constants into larger language phrases called expressions. Operators are of five classes:

a) arithmetic operators—unary: + and - ;
    and binary: + (addition), - (subtraction),
    * (multiplication), / (division), and
    ** (exponentation).

b) logical operators—unary: NOT ;
    and binary: AND, OR.

c) relational operators = (equal),
    ≠ (not equal), >= (greater than or equal),
    <= (less than or equal),
    > (strictly greater than),
    < (strictly less than).

d) concatenation operators (for building up matrices from elements) :
    a comma (,) is used for horizontal concatenation; a number sign (#) is used for vertical concatenation.

e) set operators − OR (union),
    AND (intersection), AND NOT (relative complement).

The use and meaning of the first three operators is quite similar to operators in existing languages (ALGOL) while the concatenation operator may be new to the
reader. This operator is used to build larger storage structures from smaller ones. For now an example of concatenation operators will be given; the detailed explanation of their use being presented in Section 3.2.3. Suppose A, B, C, and D are matrices of the same dimensions. Then \( M := (A, B) \# (C, D) \) represents a larger matrix of the following form: \( M = \begin{pmatrix} A & B \\ C & D \end{pmatrix} \). If the programmer writes \( M := (A, B) \# (C, D) \) partly on one punch card and partly on the next it takes the form \( M := (A, B) \)\( \# \) (C, D).

To resolve ambiguities which can develop in forming combinations of elements, each operator has an associated precedence. In the absence of parenthesis to dictate the meanings of such combinations, the meaning will be given by the precedence of the operators, with those having higher precedence being first. Operators of equal precedence will be performed from left to right as one would expect. Section 2.5.2 in Part III interprets the operator symbols in order of decreasing precedence. An \( \# \) before an operator indicates that its precedence is the same as the preceding operator. The following examples show the meaning of precedence.

\[
\begin{align*}
A - B/C + D & \quad \text{is interpreted as} \quad A - (B/C) + D \\
(A, B)\#C & \quad \text{is interpreted as} \quad \left[ (A, B) \right]_C \\
B + C/D \# B\#A & \quad \text{is interpreted as} \quad B + \left( (C/(D\#A))\#A \right)
\end{align*}
\]

Ambiguous notation in two of the examples can be avoided, of course, by use of parentheses.
2.4 Reserved Words

Reserved words in MPL fall into the categories of keyword symbols or standard function names such as sin(x) and procedure names. Recall that reserved words may not be used as variable names. Keyword symbols (such as FOR, IN, END, GO TO) will be discussed in Section 4.2.3.

Functions:

A standard function name identifies a standard function. It is hoped that extensive use of standard functions will lead to ease in programming and enhance the readability of the resulting codes. Presented in Section 5, Part III is a list of standard functions, which hopefully will grow as MPL develops. Reference to a standard function is of the form V := F(P) where V represents the value of the function, F represents the name of the function, and P represents one or more arguments which we will refer to as a parameter list. Depending upon the function, the value may be integer or real, scalar, vector, or matrix, and if matrix, it may have any structure shape. These properties as well as the properties of the parameter list are described in Part III. Following are a few examples of the use of standard functions. Let C and X be vectors, A a matrix, and T an index set all previously defined:

\[ Z := \text{SUM}(C(I) \times X(I) \text{ FOR I IN T}); \]
\[ R := \text{ARGMIN}(B(I)/A(I,S) \text{ FOR I IN T} | P(I,S) > 0); \]

2.5 Comment Statements (Quote Symbols)

In the algorithms coded thus far by the MPL group, it has been found that comments are essential for readability of computer codes. Comments may be placed between any two sentences and are separated from the program by quotes.
marks before and after the comment. Example:

SAME LOCATION (COUNT, COUNT');
A := B + C;
"A IS THE SUM OF B AND C"
FOR I IN SET_1, COUNT' := COUNT + 1;
"WHERE COUNT' IS THE UPDATED VALUE OF
COUNT WHICH IS STORED IN THE SAME
LOCATION AS COUNT AND REFERRED TO HERE-
AFTER AS COUNT."

The general objective of MPL is readability. It is however, doubtful that
a program will be readable unless liberally interlaced with comments statements
whereby the programmer explains to the reader why he is doing the various steps.
In experiments with mathematical programming routines, almost two lines of comments
are needed on the average to explain an executable line of code. Comment statements
can consist of one or several lines set off at the beginning and end by quote marks.

"PIVOTING WILL BE DONE ON THE FULL
 MATRIX D WHICH INCLUDES A, THE
 RHS B, AND COSTS C."

D := (A, B)#
   (C, 0);
"WE NOW INCREMENT COUNT AND RECYCLE."

COUNT' := COUNT + 1; GO TO (21);
3.0 **Expressions**

Variables, constants, and operators are combined into larger language phrases called expressions. Expressions are either arithmetic, logical, set or character. In addition, the value of an arithmetic expression has a shape (rectangular, diagonal, lower triangular, upper triangular, sparse). The following sections explain the use and meaning of some of the special features of MPL expressions.

3.1 **Logical Expressions**

A logical expression, having the value of TRUE or FALSE, is a comparison between two arithmetic expressions. Two arithmetic expressions which are compared by a relational operator must be identical in type, form and shape. Following are examples of logical expressions:

\[
A \rho B \\
\text{NOT } (X(I) \geq Y(I)) \\
(Z \geq M) \text{ AND } (B + C \leq A + D) \\
(H(I) = Z(I)) \text{ OR } (M = Q)
\]

When \( A \) and \( B \) are scalars and \( \rho \) is a relational operator, then the interpretation of \( A \rho B \) is clear. However, in the case of arrays, the meaning of \( A \geq B \) can differ by author. Table 1 below defines precisely what is meant by the relational operators in MPL.
Table 1

In this Table, A and B are arrays identical in type, form, and shape. 

A_i, B_i refer to elements of A and B.

<table>
<thead>
<tr>
<th>MPL Statement</th>
<th>Mathematical Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>A = B</td>
<td>A_i = B_i \forall_i</td>
</tr>
<tr>
<td>A &lt; B</td>
<td>A_i &lt; B_i \forall_i</td>
</tr>
<tr>
<td>A &lt; B</td>
<td>A_i &lt; B_i \forall_i</td>
</tr>
<tr>
<td>A = B</td>
<td>A_i = B_i \forall_i</td>
</tr>
<tr>
<td>A &gt; B</td>
<td>A_i &gt; B_i \forall_i</td>
</tr>
<tr>
<td>A -&gt; B</td>
<td>A_i \neq B_i \text{ for some } i</td>
</tr>
</tbody>
</table>

3.2 Arithmetic Expressions

Arithmetic expressions are any combination of the following types—
compositional expressions, function references, and array builders.

3.2.1 Computational Expressions

Computation expressions are of the structure 'left-operand'-'operator' 'right-operand'. If the left operand is missing, the operator is unary (one operand) — Example: -A, + (Q-Z/B). If both operands are present, they are connected by a binary operator (two operands) — Example: A+B, C*D. At execution time the expression will be evaluated to produce a result. In addition to being defined, an operation can only be performed if the operands conform to the conventional restrictions of matrix algebra (for example - M and N are matrices, then M*N has meaning if and only if the number of columns of M equals the number of rows in N). Section 2.5 of Part III describes these relationships in detail.
3.2.2 Function References

A function reference expression involves the use of predefined functions
as set forth in Section 2.4. Examples of function references used with
computational expressions to form new arithmetic expressions are given below.

\[ X \cdot \text{SUM}(Y) \]
\[ A \cdot \text{TRANSPOSE}(B) \]
\[ \text{BASIC}_COSTS \cdot \text{INVERSE}(\text{BASIS}) \]

We shall see further use of function references in array builders in the next
section.

3.2.3 Array Builders

There are two types of array builders—concatenators and array designators.

A concatenator is a notational device for constructing vectors and matrices
by concatenation. The rules for the use of a concatenator will be given followed
by several examples.

Operations within a concatenator are horizontal concatenation (denoted by
a comma) and vertical concatenation (denoted by a number sign). Horizontal
concatenation has precedence over vertical concatenation and is performed first
whenever both operations appear. Two structures being concatenated must conform,
i.e., have the same number of rows for horizontal concatenation and the same
number of columns for vertical concatenation. Both of the structures being
concatenated must be of the same type, all arrays must be rectangular and the
result is also rectangular. As an example of the use of array constructors,
consider the following:
A has M rows and N columns (matrix)
B has M rows and 1 column (column vector)
C has 1 row and N column (row vector)
(B, TRANSPOSE(C)) has M rows and 2 columns: (B C^T)
(A, b) or A, B has M rows and N+1 columns: (A B)
(A)%(C) has M+1 rows and N columns
((A,B)% (C, 0) has M+1 rows and N+1 columns

The above examples of correct usage of the array constructor while the following examples display incorrect usage because of the incompatibility of the rows and columns.

(A, C)
(A # B)

An array designator is used to horizontally concatenate several matrices
D(j) for J in some index set L. For example L might be a list of basic columns L(1), L(2), ..., L(M). Then the basis B is given by

B := (A(:, J) FOR J IN L);

Alternatively, it can be written

F := (A(:, L(I)) FOR I IN (1,...,M));

However, it should not be written

B := A(:, J) FOR J IN L;

because without the concatenation symbol it is equivalent to

FOR J IN L DO
  B := A(:, J);
ENDFOR;
which is quite different. Nor should it be written

\[ B := (A(*, L(1)), \ldots, A(*, L(M))); \]

because this does not define the running index and \((k, \ldots, \ell)\) in MPL means \((k, k+1, \ldots, \ell)\). Still simpler we can write

\[ B := A(*, L); \]
4.0 Statements

All statements in MPL are categorized first by whether or not they are preceded by a label. All statements are ended by the terminator semi-colon (;).

4.1 Labeled Statements

A label is a means of providing a specific location in a program to which execution control may be transferred. Labels are either a string of digits enclosed in parentheses or can have a name like a variable. A labeled statement consists of a label, followed by a colon followed by an "unlabeled statement" (defined in 4.2) and may be used only once as a label within each storage block. A label can only be referred to later in GO TO statements. Examples:

\[
\text{VAR :} \text{~} X + Y; \\
\text{UPDATING: ITERATIONS' :} \text{~} \text{ITERATIONS + 1; } \\
\text{GO TO UPDATING;}
\]

4.2 Unlabeled Statements

Unlabeled statements are of three types—assignment statements, procedure call statements, and keyword statements.

4.2.1 Assignment Statement

Assignment statements are used for transferring data values between data storage locations. The form of a substitution statement is \( V := \text{AE} \); where \( V \) is any variable as defined in Section 2.1 and \( \text{AE} \) is any arithmetic expression as defined in Section 3.2. Examples:

\[
A := B + C; \\
S := \text{ARGMIN}(V); \\
A(I, e) := B + C - 3eD;
\]
4.2.2 Procedure Call Statement

A procedure call statement transfers execution control to a procedure. When the execution of the procedure is completed, control returns to the statement following the procedure reference. More will be said about procedures in Section 5.1. Examples:

\[ \text{PIVOT}(M,R,S); \]
\[ \text{SIMPLEX("IN" A,B,C, "OUT" Z, BV, X,BV);} \]

4.2.3 Keyword Statements

Much of the power of MPL lies in the use of keyword statements. Formally, a keyword statement is one which begins with reserve words such as DEFINE, FOR, IF, GO TO, LET, ENDIF, RELEASE, RETURN. The complete list will be found in 3.2.4 in Part III. The keyword indicates to the computer and the programmer what type of action is desired. Some of the keyword statements will be discussed here, the remainder being discussed in Chapter 5 (Statement Blocks).

4.2.3.1 GO TO Statement

A GO TO statement is used to alter the normal sequential flow of control during the execution of a program. The form is \( \text{GO TO } l; \) where \( l \) is any label as defined in Section 4.1. Example:

\[
\begin{align*}
\text{ITERATE:} & \quad l' = l + 1; \\
& \quad \vdots \\
& \quad \text{GO TO ITERATE;}
\end{align*}
\]

4.2.3.2 Simple Conditional Statement (IF)

A simple conditional statement enables one to execute a single statement only
if certain conditions hold, and skip it otherwise. The form is

\[ s \text{ IF } le; \]

where \( le \) is any logical expression as defined in Section 3.1 and \( s \) is an assignment statement. Examples:

\[
\begin{align*}
S &:= 0 \text{ IF } A(x, J) = B; \\
R &:= S + T \text{ IF } Z = 0; \\
K &:= R \text{ IF } U = 0; \\
L &:= S \text{ IF } V \geq 0;
\end{align*}
\]

If the logical expressions \( le \) is true, the program is executed with \( s \) replacing the entire conditional statement. If not true, the program goes to the next statement.

In section 5.4 a compound conditional form is discussed. Its form is

\[
\text{IF } le \text{ THEN } s_1, \ldots, s_k \\
\text{OR IF } le \text{ THEN } s_{k+1}, \ldots, s_m \\
\text{OTHERWISE } s_{m+1}, \ldots, s_n \\
\text{ENDIF};
\]

4.2.3.3 Simple Iterated Statement (FOR)

A simple iterated statement is used to perform a given statement several times in such a manner that during each execution an iteration index is changed according to a predetermined pattern. The form is

\[ s \text{ FOR } v \text{ IN set; } \]
where \( v \) is any variable name as defined in Section 2.1, \( \text{set} \) is any index set variable as defined in Section 2.1 and \( \text{s} \) is a statement. \( \text{s} \) in general depends on \( v \). The first part of the conditioned statement (the \text{FOR} phrase) states that the values of an iteration index (\( v \) arc to range over \( \text{set} \)). The first cycle through \( \text{s} \) is executed with the first value of \( v \) in \( \text{set} \); the second cycle is executed, the second value of \( v \) in \( \text{set} \), and so forth until the last value of the iteration index has been used in the execution of \( \text{s} \). Then control is passed onto the next statement. Example:

\[
A(I) := B(I,J) \text{ FOR } I \text{ IN } (1,\ldots,M);
\]

In Section 5.3 a compound iterated statement is discussed. Its form is

\[
\text{FOR } V \text{ IN } \text{set} \text{ DO } S_1,\ldots,S_m \text{ ENDFOR;}
\]

4.2.3.4 \textit{Let Statement}

The \textit{let} statement enables one to represent one symbol by another and was introduced into MPL to enhance readability. This statement is similar to a \textit{MACRO}. It causes modification of the program at compiler time instead of execution time. The \textit{let} statement will be explained by showing several examples of its use.

a) \text{LET M := MATRIX;}  
\hspace{1cm} A := M*B; 
\hspace{1cm} \text{is equivalent to} \hspace{1cm} A := \text{MATRIX} * C;

b) \text{LET L(I) := RHS(I)/A(I,S); LET T := (1,\ldots,M);}  
\hspace{1cm} R := \text{ARGMIN} (L(T));  
\hspace{1cm} \text{is equivalent to} \hspace{1cm} R := \text{ARGMIN} (L(I)) \text{ FOR } J \text{ IN } T);  
\hspace{1cm} \text{or equivalent to} \hspace{1cm} R := \text{ARGMIN} (\text{RHS(I)/A(I,S)) FOR I \ IN (1,\ldots,M));}

c) \[ \text{LET } \text{BI} := \text{BASIS\_INVERSE}; \text{LET } \text{BC} := \text{BASIC\_COSTS}; \]
\[ \text{PI} := \text{BC} \times \text{BI}; \]

is equivalent to
\[ \text{PI} := \text{BASIC\_COSTS} \times \text{BASIS\_INVERSE}; \]

Note also in the first example that \( I \) is a dummy and that another symbol \( \mathcal{J} \) was used in its place later on. The form of a let statement is \[ \text{LET } v := e; \]
where \( v \) is a variable and \( e \) is an expression.

In the case that let is only used to simplify a single statement, an inverted let or \texttt{WHERE} form can be used.

\[ R := \text{ARGMIN}(L(J) \text{ FOR } J \text{ IN } T) \]
\[ \text{WHERE } T := (1, \ldots, M); \]

4.2.3.6 Define Statement

Before a variable name may be used in a program the type, structure and storage requirements of the values which it represents must be explicitly or implicitly defined. The only exception to this rule is that an undefined variable may be used as a dummy iteration index or as a dummy variable in a let or where situation. The declaration may be done in two ways. One is to define the variable but not give it any values:

\[ \text{DEFINE } V 1 \text{ BY } M; \]

The other is to define the variable and assign it values at the same time. In the example below \( V \) is a new variable while \( A \) and \( B \) have been previously defined.

\[ V := A + B; \]

Let us now explore the details and meaning of the define statement.
The form of an explicit DEFINE statement is:

<table>
<thead>
<tr>
<th>DEFINE</th>
<th>Variable</th>
<th>Type</th>
<th>Shape</th>
<th>Dimensions or Domain</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>name</td>
<td>ARITHMETIC</td>
<td>RECTANGULAR</td>
<td>m BY n</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>DIAGONAL</td>
<td>(m₁,...,m₂) BY (n₁,...,n₂)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>UPPER TRIANGULAR</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>LOWER TRIANGULAR</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>SPARSE WITH K NONZeros</td>
<td></td>
</tr>
<tr>
<td>name</td>
<td>logical</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>name</td>
<td>character</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>name</td>
<td>set</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Words "ARITHMETIC", "RECTANGULAR" will be understood if type, shape or size descriptors are omitted. Scalar is assured if size description is missing. Let symbols k, m, n, m₁, m₂, n₁, n₂ be any previously defined integers or integer expressions. A matrix "SPARSE WITH K NON-ZEROS" means the matrix has at most k non-zeros. It will be stored as a sparse matrix. A list which has neither row nor column interpretation may be indicated by "(m)" where m is the number of elements. Examples:

1. DEFINE E M BY N;
2. DEFINE D, E DIAGONAL P BY B;
3. DEFINE D (1,...,M) BY (K,...,1);
4. DEFINE J;
5. DEFINE M SPARSE WITH P NONZEROS;
6. DEFINE C I BY N;
7. DEFINE B M BY 1;
8. DEFINE L CHARACTER;
9. DEFINE S SET;
The form of a domain descriptor is SRL where SRL is a subscript range list, a series of subscript ranges separated by a BY. A subscript range is two arithmetic expressions separated by ,...,. Example of subscript range list: (1,...,M) BY (M+N,...,K). Each subscript range determines the minimum and maximum values of the array's subscripts. The number of subscript ranges in the subscript range list determines the number of dimensions of the storage structure. If the domain is of the form (1,...,M) BY (1,...,N) it is written in Dimension form M BY N or simply M for a one-dimensional list or set. The description shape and size descriptions may appear in any order in a define statement.

The second (and most used) method of defining a variable is implicitly. The form of an implicit define statement is vn := ae; where vn is a variable name as defined in Section 2.1 and ae is an arithmetic expression as defined in Section 3.2. In this version of the define statement the variable name being defined is given the same form, type, and structure as the value of the first arithmetic expression. Examples:

\[ M := (A, B)\quad (C, D); \]
\[ M := (A, B, C); \]
\[ B := (P(*, BL(I)) \text{ FOR } I \text{ IN } (1,...,M)); \]
\[ D := E + F*G; \quad \text{"WHERE } E \text{ AND } F \text{ ARE MATRICES"} \]
5.0 Statement Blocks

A program in MPL consists of a sequence of statements (defined in 4.0) and statement blocks. A statement block is a sequence of statements with special initiating and terminating statements. There are four kinds of statement blocks—procedure blocks, storage allocation blocks, conditional blocks and iteration blocks. The entire program is a procedure block. A block can have other blocks imbedded within it, or it may be imbedded in other blocks, but no two blocks partially overlap.

5.1 Procedure Blocks

A procedure is designed to carry out a specific sequence of operations which may be required over and over again. Rather than rewriting the sequence of steps each time, they may be written once in a form which can be utilized whenever needed. It is hoped that a library of procedures written in MPL will be developed, thereby enabling the work of one programmer to be available to others. This will not only speed up the writing of MPL codes, but will also enhance the readability. Later on we will say how to call a procedure in a program.

If one wants to write a procedure (which will later be called by some main routine), the procedure is initiated by a procedure statement, contains a statement sequence, and is terminated by a final statement. A procedure statement consists of the reserved word PROCEDURE followed by a procedure identifier. The procedure identifier specifies both the procedure name and the local names of the input-output parameters. The form of a procedure identifier is a variable name followed usually by a list of parameters enclosed in a pair of parentheses.
The *fini* statement is used to mark the end of a procedure write up. In contrast, *RETURN* is a signal during execution of a program that control is to be passed back to the main routine. This also terminates any storage allocation, iteration, or conditional blocks which were initiated but not explicitly or implicitly terminated within the procedure.

Control is passed to a procedure by either a function or a procedure reference call. A procedure may have several return statements, each one may cause termination during execution. Values are transferred to and from the procedure by means of substitution statements in the input-output section of the procedure identifier. In general, new variables for the main routine may be defined in the output section.

As an example of the use of the return statement in a procedure consider the following routine for checking whether two column vectors are equal.

COMPARE := 0 means \(A = B\).

```
PROCEDURE COMPARE(A,B)
(1): IF ROW_DIM(A) != ROW_DIM(B) THEN
    COMPARE := 1;
    RETURN;

    OTHERWISE

(2): FOR I IN ROW_DOM(A) DO
    IF A(I) != B(I) THEN
        COMPARE := 1;
        RETURN;
    ENDIF;
ENDFOR;
    COMPARE := 0;

(3): RETURN;
ENDIF;
FINI:
```
Next suppose that in a program we have the following sequence of statements:

\[
\text{IF \ COMPARE}(X,Y) = 0 \ \text{THEN} \ \text{GO TO}(21); \ \text{OTHERWISE} \ \text{GO TO} (23); \ \text{ENDIF};
\]

thus if the vector \( X \) equals the vector \( Y \) in each component, control is transferred to the statement (21), if not, it goes to (23).

5.2. **Storage Allocation Blocks, Release Statements**

Storage allocation blocks are required for the efficient use of memory core in a computer. To release a symbol and any storage for other use, the statement takes the form:

\[
\text{RELEASE} \ A, \ B;
\]

After much debate, it was decided that in writing mathematical programming codes, block storage allocation was preferable to continual re-allocation.

Release of symbols takes place automatically, however, with subprogram blocks and special release blocks.

All symbols and storage except outputs, generated within a procedure are released when the procedure returns to the main routine. Hence the same symbols outside the procedure can be used with entirely different meanings.

G in the statement

\[
Z := A + G \ \text{WHERE} \ G := \text{INVERSE}(M);
\]

is treated as a `dummy variable` locally defined within the block and immediately released. However, in the situation

\[
\text{LET} \ G := \text{INVERSE}(M);
\]

\[
Z := A + G;
\]

the release of \( G \) is not possible until the end of a procedure unless by a special
5.3 Iteration Block

An iteration block is a statement sequence which is repeated a number of times only with an iteration index changed between each execution. As such, this is a generalization of the iterated statement (Section 4.2.3.3). An iteration block is initiated by a for statement, contains a statement sequence, and is terminated by an endfor statement. The for statement (very similar to the for phrase of Section 4.2.3.3) governs the behavior of the iteration by specifying the values for the iteration index. Iteration blocks do not release symbols and storage like a subroutine blocks. Example: The form is

```
FOR v IN set DO
  s_1, ..., s_k
ENDFOR;
FOR I IN (1, ..., M) DO
  X(I) := Y(I);
  J' := J + 1;
  A(*, I) := B(I);
ENDFOR;
```

5.4 Conditional Blocks

Conditional blocks are constructions wherein the program selects between a set of mutually exclusive courses of action. A conditional block is initiated by an if statement and terminated by an endif statement. Or if and otherwise statements allow for the provision of multiple alternatives. This construct is a
generalization of the conditional statement (Section 4.2.3.2). Conditional blocks do not release symbols generated within them. The form is:

\[
\text{IF } \text{le THEN } s_1, \ldots, s_k \\
\text{OR IF } \text{le THEN } s_{k+1}, \ldots, s_m \\
\text{OTHERWISE } s_{m+1}, \ldots, s_n \\
\text{ENDIF;}
\]

\[
\text{IF } A = B \text{ THEN GO TO (7);} \\
\text{OR IF } A = C \text{ THEN GO TO (8);} \\
\text{OTHERWISE} \\
B := A; \\
\text{ENDIF;}
\]

The OR IF and OTHERWISE are optional in a conditional block. For example

\[
\text{IF } \text{le THEN } s_1, \ldots, s_2 \text{ ENDIF;}
\]
6.0 Examples of MPL Procedures

PROCEDURE SUM(F)

"SUMS A VECTOR F OVER ITS DOMAIN"
"ACUMULATE THE RUNNING SUM IN S."

(1): S := 0;
(2): SAME LOCATION (S', S);

"S' WILL BE THE UPDATED VALUE OF S TO BE STORED IN THE SAME
LOCATION AS S AND THEREAFTER REFERRED TO AS S."

(3): S' := S + F(I) FOR I IN DOM(F);

"ITERATIVELY ADDS F(I) TO S"

(4): SUM := S;
(5): RETURN; FINI;

PROCEDURE MIN_1("IN" F, "OUT" K, M)

"K IS THE FIRST INDEX I WHERE F(I) TAKES ON ITS MINIMUM
VALUE M OVER DOMAIN OF F."
"INITIALIZE K AND M"

(1): K := DOM(F)(1); "I.E. THE FIRST COMPONENT OF THE SET DOM(F)"
(2): M := F(K);
(3): SAME LOCATION (K, K'), (M, M');

"K', M', ARE UPDATED VALUES OF K, M"
(4): FOR I IN DOM(F) DO
    IF F(I) < M THEN
        K' := I;
        M' := F(I);
    ENDIF;
ENDFOR;
(5): RETURN; FINI;

PROCEDURE COL_PIVOT (A,P,R);
"WARNING - MODIFIES A AND STORES THE RESULT A' IN THE
SAME LOCATION AS A."
"PIVOTS (A, P) ON P(R) WHERE A IS A MATRIX AND P A
COLUMN VECTOR, AND RETURNS A', THE MODIFIED A PART ONLY."

(1): SAME LOCATION (A', A);
(2): M := ROW_DIM(A);
(3): LET T := (1,...,M) AND NOT R;
(4): A'(R, *) := A(R, *)/P(R);
(5): A'(I, *) := A(I, *) - A'(R, *) * P(I) FOR T IN T;
(6): COL_PIVOT := A';
(7): RETURN; FINI;

PROCEDURE REVISED_SIMPLEX_2("IN" A,D,C,BV, "OUT" STATUS, X,Z,K);
"REVISED_SIMPLEX_2 IS JUST PHASE 2.
A = MATRIX, C = COSTS, D = RHS, BV = BASIC VARIABLES,
X = BV VALUES, Z = OBJECTIVE VALUE, K = ITERATIONS"
"THE PROBLEM IS TO FIND MIN Z, X \geq 0, AX = D, CX = Z.
IF MIN Z ISFINITE, STATUS = FINITE, OTHERWISE STATUS =
INFINITE. IT IS ASSUMED THAT BV IS A BASIC FEASIBLE SET
OF VARIABLES."
"INITIALIZATION"

(1): \( K := 0 \),
(2): \( \text{STATUS} := '\text{FINITE}' \);

"The first step is to set up the initial basis which consists of the set of basic variable columns, \( BV \), of \( A \). Thus \( \text{BASIS} := A(BV) \). Let \( G \) be the inverse of the basis. We are interested in computing \( G \) and later updating it."

(3): \( G := \text{INVERSE(\text{BASIS})} \) where \( \text{BASIS} := A(BV) \);

"Also \( X \), the values of the basic variables, are initially"

(4): \( X := G \times D \);

"ITERATIVE LOOP"

"The costs associated with basic columns are \( C(BV) \) - hence the simplex multipliers \( P \) are given by"

(5): \( P := C(BV) \times G \);

"Let \( S \) denote the index of the column of \( A \) coming into the basis and \( ^{-1}S = C(S) \)."
(6): \texttt{MIN}("IN" C \cdot P \ast A, "OUT" S, C_S);

"Which is the index (argument) of the smallest component of the vector of relative costs C \cdot P \ast A."

"Test for finite \textit{MIN} Z"

(7): \texttt{GO TO (16)} IF C_S > 0;

"Let \( Y \) be the representation: terms of the basis."

(8): \texttt{Y := G} \ast A(*, S);

"Let \( R \) denote the index of the column in the basis to be removed"

\texttt{LET T := (I IN DOM(Y) | Y(I) > 0)};

\texttt{IF T = NULL THEN}

\texttt{STATUS := 'INFINITE'};

\texttt{GO TO (16)};

\texttt{ENDIF};

(9): \texttt{MIN}("IN" (X(I)/Y(I) FOR I IN T), "OUT" R, Q);

"Update \( X, G, K, BV \) denoted by \( X', G', K', BV' \)"

(10): \texttt{SAME LOCATION} (X, X'), (C, G'), (K, K'), (BV, BV');

(11): \texttt{K' := K + 1;}
(12): \( X' := X - Y \times Q; \)
\( X'(R) := Q; \)
(13): \( G' := \text{COL\_PIVOT}(G,Y,R); \)

"\text{COL\_PIVOT} pivots \( (G,Y) \) on \( Y(R) \) and returns modified \( G \) part."

(14): \( BV'(R) := S; \)

"Change \( R\)-th basic variable to \( S \)."
"Updating complete, recycle"

(15): \( \text{GO TO (5)}; \)

"Termination"

(16): \( Z := C(BV) \times X; \)
(17): \( \text{RETURN}; \)
(18): \( \text{FINI}; \)
MPL

MATHEMATICAL PROGRAMMING LANGUAGE

PART III

A FORMAL DEFINITION OF MPL

PREPARED BY STEPHEN K. SCHUCK
APRIL 1968

COMMITTEE MEMBERS
RUDOLF RAYER
JAMES RIGELOW
GEORGE DANTZIG
DAVID GRIES

MICHAEL MCGRAH
PAUL PINSKY
STEPHEN SCHUCK
CHRISTOPH WITZGALL

THIS IS THE THIRD OF THREE PARTS:

PART I A SHORT INTRODUCTION
PART II A GENERAL DESCRIPTION
PART III A FORMAL DEFINITION

NUT: BECAUSE THE DEVELOPMENT OF PARTS I AND II WAS SLIGHTLY OUT OF PHASE WITH THE DEVELOPMENT OF PART III, THE READER MAY OBSERVE SOME NOTICEABLE, ALTHOUGH NOT SIGNIFICANT, DISCREPANCIES BETWEEN THEM. THESE DISCREPANCIES ARE DUE TO THE FACT THAT MPL IS NOT YET FULLY DEVELOPED AND MANY IDEAS ARE STILL EXPERIMENTAL.
ABSTRACT

COMMUNICATION WITH A DIGITAL COMPUTER IS A PROBLEM WHICH HAS OCCUPIED MANY PEOPLE FOR A LONG TIME. IN ORDER TO ALLOW THE COMPUTER TO BE MORE WIDELY USED AS A COMPUTATIONAL TOOL MUCH OF THIS EFFORT HAS GONE INTO DEVELOPING SYSTEMS THROUGH WHICH A PERSON MAY COMMUNICATE HIS DESIRES EVEN THROUGH HE IS NOT FAMILIAR WITH THE SOPHISTICATED AND HIGHLY DETAILED PROGRAMMING LANGUAGES AVAILABLE. THE MATHEMATICAL PROGRAMMING LANGUAGE IS ANOTHER ATTEMPT TO PROVIDE A LANGUAGE IN WHICH THE NON-PROGRAMMER MAY WRITE PROGRAMS. THE VALUE OF THIS WORK LIES IN THE FACT THAT IT IS ORIENTED DIRECTLY TOWARD MATHEMATICAL PROGRAMMING. CONSEQUENTLY CONSIDERABLE EFFORT HAS BEEN MADE TO MAKE MPL LOOK AS MUCH LIKE STANDARD MATHEMATICAL NOTATION AS POSSIBLE.

IT IS HOPE THAT THIS WORK WILL PRODUCE A RIGOROUSLY DEFINED LANGUAGE IN WHICH MATHEMATICAL PROGRAMMERS CAN DESCRIBE ALGORITHMS WHICH WILL AT THE SAME TIME BE EASILY UNDERSTOOD BY OTHER MATHEMATICAL PROGRAMMERS AND MEANINGFUL AND VALID COMPUTER PROGRAMS.

SINCE MPL IS A LANGUAGE INTENDED FOR COMMUNICATION BOTH WITH OTHER INDIVIDUALS AND WITH COMPUTERS, ITS DEVELOPMENT IS AN EFFORT TO PROVIDE A 'READABLE' PROGRAMMING LANGUAGE. HOWEVER, FOR A PROGRAM TO BE READABLE (AN EASY TO USE AND RAPID METHOD FOR TRANSFERRING INFORMATION) IT MUST BE BOTH 'UNDERSTANDABLE' (THE NOTATION IS FAMILIAR OR SELF-EXPLANATORY WITHIN ITS CONTEXT) AND 'COMPREHENDABLE' (THE PARTS OF A PROGRAM MUST INTERRELATE IN A MEANINGFUL MANNER FOR THE PROGRAM READER). IN THIS RESPECT THE EMPHASIS OF MPL IS UPON PROVIDING AN UNDERSTANDABLE LANGUAGE. COMPREHENDABILITY WILL STILL BE THE USER'S RESPONSIBILITY.
TABLE OF CONTENTS

0

INTRODUCTION

0-1

ABSTRACT

0-2

TABLE OF CONTENTS

0-3

MPL LANGUAGE DESIGN PHILOSOPHY

0-4

USE OF THE MANUAL

1

BASIC LANGUAGE STRUCTURE

1-1

AN ORGANIZATION OVERVIEW

1-2

THE MPL CHARACTER SET

1-3

SOME ELEMENTARY PHRASES

2

EXPRESSIONS

2-1

ATTRIBUTES OF EXPRESSIONS

2-2

CONSTANTS

2-2-1

NUMBERS

2-2-2

LOGICAL CONSTANTS

2-2-3

SET CONSTANTS

2-2-4

CHARACTER CONSTANTS

2-3

VARIABLES

2-3-1

VARIABLE NAMES

2-3-2

SUBSCRIPTS

2-4

PROCEDURE CALLS

2-4-1

PROCEDURE NAMES

2-4-2

PARAMETER LISTS

2-5

COMPUTATIONAL EXPRESSIONS

2-5-1

OPERATOR CLASSES AND ALLOWABLE CONFIGURATIONS

2-5-2

OPERATOR DEFINITIONS AND PRECEDENCES

2-5-3

SEMANTICS

2-6

OTHER EXPRESSIONS

2-6-1

DOMAIN ITEM

2-6-2

CONCATENATOR

2-6-3

ARRAY CONSTRUCTOR

2-6-4

SUBSET SPECIFIER

3

PROGRAM CONSTRUCTION (PROCEDURES)

3-1

STATEMENT SEQUENCES

3-2

STATEMENTS

3-2-1

LABELS

3-2-2

ASSIGNMENT STATEMENTS

3-2-3

PROCEDURE CALL STATEMENTS

3-2-4

SIMPLE KEYWORD STATEMENTS

3-2-4-1

LET STATEMENT

3-2-4-2

GO TO STATEMENT

3-2-4-3

RETURN STATEMENT

3-2-4-4

OFFLINE STATEMENT

3-2-4-5

RELEASE STATEMENT

3-2-5

COMPLEX KEYWORD STATEMENTS

3-2-5-1

CONDITIONED STATEMENT

3-2-5-2

ITERATED STATEMENT

3-2-5-3

BLOCK STATEMENT
TABLE OF CONTENTS (CONTINUED)

4  INPUT/OUTPUT STATEMENTS
5  LIBRARY PROCEDURES
6  PROGRAM FORMATION MECHANICS
   6-1  CARD FORMAT
   6-2  USE OF BLANKS
   6-3  COMMENTS
7  RESUME OF DEFINITIONS
8  SAMPLE PROGRAM

0-3  MPL LANGUAGE DESIGN PHILOSOPHY

The philosophy behind the design of the Mathematical Programming Language (hereafter called MPL) is to provide a maximum of readability to the uninitiated. Thus it can hopefully be assumed that the user has only a familiarity with the notation of current mathematical literature. As a result the language definition attempts to avoid abbreviations which may be obscure, to keep the number of special symbols to a minimum, and to provide the most familiar notation and formation.

As MPL developed it became obvious that many useful structures were available in existing languages. As a result the reader who is familiar with ALGOL, FORTRAN, PL/1, etc., will encounter familiar forms and philosophies. No attempt has been made to parallel any single such language, but where applicable to develop the best that was available.
THE FOLLOWING DISCUSSION IS ORGANIZED SO THAT THE READER MAY FOLLOW THE CONSTRUCTION OF MPL FROM THE MOST ELEMENTARY UP TO THE WIDEST CONCEPTS. THE FINAL SECTION IS A SUMMARY OF THE FORMAL DEFINITIONS SO THAT THIS PAPER MAY BE USED BOTH FOR INSTRUCTION AND AS A REFERENCE MANUAL. EXAMPLES WILL BE LIBERALLY SPRINKLED AMONG THE DESCRIPTIONS.

THE DEFINITION OF MPL WHICH APPEARS HERE IS AIDED BY THE USE OF A METALINGUISTIC OR LANGUAGE-DESCRIBING LANGUAGE WHICH HAS SEVERAL SPECIAL SYMBOLS.

\(<\) \quad \text{A PAIR OF BROKEN BRACKETS DELIMITS A PHRASE NAME.}

\(''\) \quad \text{A PAIR OF PRIMES DELIMITS A CHARACTER STRING WHICH APPEARS IN A PHRASE EXACTLY AS IT APPEARS IN THE PRIMES.}

\(::=\) \quad \text{READ THIS SYMBOL "IS DEFINED AS". IT SEPARATES THE PHRASE NAME ON THE LEFT FROM THE PHRASE DESCRIPTION ON THE RIGHT.}

\(\mid\) \quad \text{READ THIS SYMBOL "OR". IT SEPARATES MUTUALLY EXCLUSIVE DESCRIPTIONS.}

EXAMPLE METALINGUISTIC STATEMENTS:

\(<\text{CHARACTER}>::=\text{<LETTER>|<DIGIT>|<SPECIAL CHARACTER>}\)

This metalinguistic statement reads "A character is defined as a letter or a digit or a special character."

\(<\text{ITERATED STATEMENT}>::=\text{<IF|<EXPRESSION>|,<STATEMENT>}\)

This reads "An iterated statement is defined as the characters 'if' followed by an expression followed by a comma followed by a statement."
1-1 AN ORGANIZATIONAL OVERVIEW

THE MPL LANGUAGE IS DESIGNED TO FACILITATE THE COMMUNICATION OF MATHEMATICAL PROGRAMMING ALGORITHMS. THE COMPLETE STATEMENT OF AN ALGORITHM IN MPL IS A 'PROGRAM'. A PROGRAM IS COMPOSED ONE OR MORE 'PROCEDURES', EACH OF WHICH IS A SEQUENCE OF SEVERAL 'STATEMENTS'. EACH STATEMENT IS MADE UP OF 'RESERVED WORDS' AND 'EXPRESSIONS', THE BASIC BUILDING BLOCKS OF MPL. THESE, FINALLY, ARE COMPOSED OF 'CHARACTERS'.

1-2 THE MPL CHARACTER SET

THE CURRENT VERSION OF MPL IS BASED UPON THE CHARACTER SET OF THE IBM 029 KEYPUNCH. FOR CONVENIENCE THESE CHARACTERS ARE GROUPED INTO THE CATEGORIES OF LETTERS, DIGITS, AND SPECIAL CHARACTERS.

<Character>::=<Letter>|<Digit>|<Special Character>

WHERE THE SPECIFIC CHARACTERS IN EACH CATEGORY ARE GIVEN BY:

<Letter>::= 'A'|'B'|'C'|'D'|'E'|'F'|'G'|'H'|'I'|'J'|'K'|'L'
|'M'|'N'|'O'|'P'|'Q'|'R'|'S'|'T'|'U'|'V'|'W'|'X'|'Y'|'Z'

<Digit>::= '0'|'1'|'2'|'3'|'4'|'5'|'6'|'7'|'8'|'9'

<Special Character>::= '('|')'|'*'|'+|'-'|'/'
|'='|'|'|'|'|'|'|'|'|'|'|'|'|'|'|'|'
|'|'|'|'|'|'|'|'|'|'|'|'|'|'|'|'

TWO OTHER CHARACTERS ARE AVAILABLE ON THE 029 KEYPUNCH, BUT ARE NOT INCLUDED IN THE ABOVE CATEGORIES DUE TO THEIR SPECIAL USAGE IN MPL. THESE CHARACTERS ARE

* Statement Terminator
* Comment Delimiter

1-3 SOME ELEMENTARY PHRASES

<Character String>::= ''|<Character String><Character>

<Digit String>::=<Digit>|<Digit String><Digit>

<Null Phrase>::= ''|<Null Phrase>

These phrases are used in several places throughout the manual. The character and digit strings are just strings of characters or digits as their names imply. The null phrase indicates that the phrase which it describes may be omitted.
2  EXPRESSIONS

\[
\text{<EXPRESSION>} ::= (\text{<EXPRESSION>})^* \\
\text{<NUMBER>}
\text{'\text{'TRUE'}|'\text{'FALSE'}
\text{'\text{'NULL'}
\text{''<CHARACTER STRING>''}
\text{<VARIABLE>}
\text{<PROCEDURE CALL>}
\text{<COMPUTATIONAL EXPRESSION>}
\text{<DOMAIN ITEM>}
\text{<CONCATENATOR>}
\text{<ARRAY CONSTRUCTOR>}
\text{<SUBSET SPECIFIER>}
\]

EXPRESSIONS ARE ELEMENTS OF MPL WHICH HAVE "VALUE". THEY USUALLY
DERIVE THEIR VALUES FROM MANIPULATIONS OF VALUES OF CONSTITUENT
PARTS. THE MOST BASIC EXPRESSIONS ARE CONSTANTS WITH FIXED
VALUES AND VARIABLES WITH VALUES WHICH MAY CHANGE DURING PROGRAM
OPERATION. EACH CONSTANT AND VARIABLE, AND CONSEQUENTLY EACH
EXPRESSION, HAS AN ASSOCIATED SET OF ATTRIBUTES WHICH DESCRIBE THE
PROPERTIES OF THE VALUE OF THE EXPRESSION.

2-1  EXPRESSION ATTRIBUTES

*TYPE*  MPL ALLOWS THE USER TO MANIPULATE VALUES WHICH ARE ARITHMETIC
QUANTITIES, LOGICAL OR BOOLEAN QUANTITIES, SETS, OR CHARACTER STRINGS.
CONSEQUENTLY THE POSSIBLE VALUES FOR THE TYPE ATTRIBUTE ARE ARITHMETIC,
LOGICAL, SET, AND CHARACTER. INITIALLY NO ATTEMPT IS BEING
MADE TO IMPOSE THE "FLOATING POINT" AND "INTEGER" SUB-CLASSIFICATIONS
OF THE ARITHMETIC TYPE ON MPL USERS. INSTEAD IT IS HOPED, PERHAPS
IN VAIN, THAT THESE HARDWARE IMPOSED CONVENTIONS MAY BE BYPASSED.

*FORM*  IF A VALUE HAS TYPE ARITHMETIC, THEN IT MAY BE EITHER A SCALAR
QUANTITY, A VECTOR QUANTITY, OR A MATRIX QUANTITY. CONSEQUENTLY THE
POSSIBLE VALUES FOR THE FORM ATTRIBUTE ARE SCALAR, VECTOR, AND MATRIX.

*SHAPE*  IF A VALUE HAS TYPE ARITHMETIC, ITS FORM USUALLY HAS A RELATED
SHAPE ATTRIBUTE WHICH PROVIDES ADDITIONAL INFORMATION ABOUT THE VALUE'S
ORGANIZATION. A SCALAR FORM HAS NO SHAPE ATTRIBUTE. A VECTOR MAY
BE EITHER A ROW VECTOR OR A COLUMN VECTOR SO ITS POSSIBLE SHAPES ARE
ROW AND COLUMN. MATRICES, NORMALLY RECTANGULAR, ARE GIVEN SHAPES TO
CONSERVE STORAGE SPACE BY STORING ONLY SUBSETS OF ELEMENTS. POSSIBLE
MATRIX SHAPES ARE RECTANGULAR, UPPER TRIANGULAR, LOWER TRIANGULAR,
DIAGONAL, AND SPARSE.
2-2  CONSTANTS

A constant is an expression which has a fixed value determined by the name of the constant. There are constants of each type.

2-2-1  NUMBERS

\[
\langle \text{NUMBER} \rangle ::= \langle \text{NUMBER BASE} \rangle | \langle \text{NUMBER BASE} \rangle \langle \text{EXponent} \rangle
\]

\[
\langle \text{NUMBER BASE} \rangle ::= \langle \text{DIGIT STRING} \rangle
\]

\[
\langle \text{EXponent} \rangle ::= \langle \text{DIGIT STRING} \rangle
\]

Essentially a number is a digit string (1-31), possibly containing a single decimal point. If the number has a very large or a very small value so that writing it requires many zeros, it becomes worthwhile to use the abbreviated 'scientific notation' provided by the exponent. Here 'E' means 'times ten to the power'. The symbol '' indicates that the sign following the 'E' is optional.

Example numbers

2 13.6 2.54 16325 15.6E-03 2E5 .006

2-2-2  LOGICAL CONSTANTS

Logical, boolean, or truth valued expressions result mostly from tests on other quantities which yield the values true or false. Since there are only two possible values for any logical expression, there are only two possible logical constants, 'true' and 'false'.

2-2-3  SET CONSTANTS

Sets in MPL are intended primarily for indexing over rows or columns of matrices, iteration loops, etc. As a result, set elements have whole number values. There are no duplicate element values in sets. However, since sets may contain a variable number of elements, they have an associated size or number of elements. The single most important test on a set is therefore whether it is empty. Thus the set constant 'null' is provided to facilitate these tests and for other uses.
CHARACTER CONSTANTS

CHARACTER CONSTANTS HAVE THE FORM **"<CHARACTER STRING>"**.

CHARACTER CONSTANTS WERE ORIGINALLY PROVIDED IN MPL FOR CONVEYING FORMAT INFORMATION TO THE INPUT AND OUTPUT ROUTINES. HOWEVER, WITH ONLY SLIGHT DEVELOPMENT A VERY POWERFUL MANIPULATING CAPABILITY APPEARED. A CHARACTER CONSTANT IS ANY STRING OF CHARACTERS DELINEATED BY A PRIME (SINGLE QUOTE) ON EACH END. A PRIME WITHIN A CHARACTER STRING MUST BE REPRESENTED BY TWO ADJACENT PRIMES, I.E. **" (AS OPPOSED TO A DOUBLE QUOTE **)

EXAMPLE CHARACTER CONSTANTS

"1H-,25E13.6"
"HLP,HELP"
"THIS IS THE JONES" HOUSE"

VARIABLES

**<VARIABLE>::=<VARIABLE NAME>|<VARIABLE>'("<SUBSCRIPT LIST>")**

VARIABLES REPRESENT VALUES. JUST AS A VARIABLE NAME IS USED TO REPRESENT AN ENTIRE MATRIX OR VECTOR, VARIABLE NAMES WITH SUBSCRIPTS REPRESENT SPECIFIC ELEMENTS OR SETS OF ELEMENTS OF THESE FORMS. MPL VARIABLES CAN REPRESENT VALUES INDIRECTLY. FOR INSTANCE, IF A REPRESENTS A MATRIX, THE ELEMENTS OF THE MATRIX COULD BE NUMBERS, OR THEY COULD BE POINTERS TO OTHER MATRICES. IN THE LATTER MANNER A(I,J)(K,L) WOULD PICK FROM A(I,J) THE POINTER TO SOME MATRIX FROM WHICH THE (K,L)TH ELEMENT WAS ACTUALLY DESIRED. THE POWER HERE IS THAT THE ELEMENTS OF AN ARITHMETIC MATRIX OR VECTOR NOW MAY BE OTHER ARITHMETIC QUANTITIES, LOGICAL QUANTITIES, SETS, OR CHARACTER STRINGS.

VARIABLE NAMES

**<VARIABLE NAME>::=<LETTER>
 |<VARIABLE NAME><LETTER>
 |<VARIABLE NAME><DIGIT>
 |<VARIABLE NAME>**

A VARIABLE NAME NAMES A STORAGE STRUCTURE AND THEREBY HAS ALL OF THE ASSOCIATED PROPERTIES OF THE STRUCTURE. IF THE STRUCTURE HAS TYPE ARITHMETIC ITS ELEMENTS MAY BE POINTERS TO OTHER STRUCTURES HAVING OTHER TYPES. A VARIABLE NAME ALWAYS BEGINS WITH A LETTER WHICH MAY BE FOLLOWED BY ANY NUMBER OF LETTERS, DIGITS, UNDERSCORES, OR PRIMES.

EXAMPLE: VARIABLE NAMES

A A* ALPHA36 THIS IS A VARIABLE NAME OBJECTIVE_FUNCTION
2-3-2  

**SUBSCRIPTS**

SUBSCRIPTS ARE SUBSCRIPT LISTS ENCLOSED IN PARENTHESES.

<SUBSCRIPT LIST>::=<SUBSCRIPT ELEMENT><SUBSCRIPT LIST>'<SUBSCRIPT ELEMENT>

<SUBSCRIPT ELEMENT>::=*<EXPRESSION>

SUBSCRIPTS ARE USED TO ACCESS SUBSETS OF ELEMENTS OF ARITHMETIC DATA STRUCTURES. THE NUMBER OF SUBSCRIPT ELEMENTS IN A SUBSCRIPT LIST MUST BE EQUAL TO THE NUMBER OF DIMENSIONS OF THE DATA STRUCTURE. THE * USED AS A SUBSCRIPT ELEMENT REFERENCES AN ENTIRE ROW OR COLUMN OF AN ARRAY. THUS A(*,*) = A AND B(*) = B WHERE A AND B ARE A MATRIX AND A VECTOR RESPECTIVELY. VALUES OF EXPRESSIONS USED AS SUBSCRIPT ELEMENTS MUST HAVE EITHER ARITHMETIC OR SET TYPE. IF THE EXPRESSION IS ARITHMETIC IT MUST BE EITHER A SCALAR OR A VECTOR. A SCALAR ACCSESSES A SINGLE ELEMENT WHILE A VECTOR ACCSESSES A SET OF ELEMENTS. ANY FRACTIONAL PART OF A VECTOR OR SCALAR ELEMENT VALUES IS DROPPED AND ANY VALUES OUTSIDE THE RANGE OF THE SUBSCRIPT ELEMENT ARE IGNORED.

**EXAMPLE VARIABLES**

A(3*A+3,C)  A*(I,J)  B(I)  A*(1,*)  FOR_ROW_SET, COL_SET

AS MENTIONED IN (2-3) THE ELEMENTS OF AN ARITHMETIC DATA STRUCTURE (VECTOR OR MATRIX) MAY ALSO POINT TO OTHER SUCH QUANTITIES. HENCE "MATRIX_LIST(K)(I,J)" ACCESSES THE (I,J)TH ELEMENT IN THE MATRIX INDICATED BY THE (K)TH ELEMENT IN "MATRIX_LIST". THIS PROCESS MAY BE CONTINUED TO ANY LEVEL, BUT WITH CARE.

2-4  

**PROCEDURE CALLS**

<PROCEDURE CALL>::=<VARIABLE NAME>

|<VARIABLE NAME>='('<EXPRESSION LIST>')'

<EXPRESSION LIST>::=<EXPRESSION><EXPRESSION LIST>'<EXPRESSION>

A PROCEDURE CALLS A PROCEDURE FROM WITHIN AN EXPRESSION. IT IS ASSUMED THAT THE CALLED PROCEDURE RETURNS A VALUE WHICH CAN BE USED TO EVALUATE THE EXPRESSION IN THE CALLING PROCEDURE.

WHEN A PROCEDURE IS DEFINED (3) ANY VALUES WHICH WILL BE PASSED FROM THE CALLING PROCEDURE AT THE TIME OF THE CALL ARE REPRESENTED BY VARIABLE NAMES IN THE VARIABLE NAME LIST FOLLOWING THE PROCEDURE NAME IN THE DEFINITION. THESE VARIABLES TAKE THE VALUES OF THE EXPRESSIONS IN THE PROCEDURE CALL EXPRESSION LIST IN THE ORDER IN WHICH THEY OCCUR.

THE VALUE OF A PROCEDURE IS DETERMINED IN AN ASSIGNMENT STATEMENT WITHIN THE PROCEDURE IN WHICH THE NAME OF THE PROCEDURE APPEARS ON THE LEFT OF THE ASSIGNMENT SYMBOL (3-2-2).

**EXAMPLE PROCEDURE CALLS**

PIVOT(A*A',B',I+7,J+K-3)
SUB(A)
2-5  COMPUTATIONAL EXPRESSIONS

<COMPUTATIONAL EXPRESSION>: = '<EXPRESSI0N> + '<EXPRESSI0N>
| '<EXPRESSI0N> - '<EXPRESSI0N>
| '<EXPRESSI0N> * '<EXPRESSI0N>
| '<EXPRESSI0N> / '<EXPRESSI0N>
| '<EXPRESSI0N> ** '<EXPRESSI0N>
| '<EXPRESSI0N> *** '<EXPRESSI0N>
| '<EXPRESSI0N> AND '<EXPRESSI0N>
| '<EXPRESSI0N> OR '<EXPRESSI0N>
| '<EXPRESSI0N> IN '<EXPRESSI0N>
| '<EXPRESSI0N> NOT '<EXPRESSI0N>
| '<EXPRESSI0N> AND NOT '<EXPRESSI0N>
| '<EXPRESSI0N> ** '<EXPRESSI0N>
| '<EXPRESSI0N> -= '<EXPRESSI0N>
| '<EXPRESSI0N> '<EXPRESSI0N>
| '<EXPRESSI0N> >= '<EXPRESSI0N>
| '<EXPRESSI0N> <= '<EXPRESSI0N>
| '<EXPRESSI0N> != '<EXPRESSI0N>

'OPERATORS' MODIFY OR CONNECT 'OPERAND' EXPRESSIONS IN COMPUTATIONAL EXPRESSIONS. ALL COMPUTATIONAL EXPRESSIONS HAVE ONE OF TWO GENERAL FORMS:

UNARY  <OPERATOR><R-OPERAND>
BINARY <L-OPERAND><OPERATOR><R-OPERAND>

2-5-1  OPERATOR CLASSES AND ALLOWABLE CONFIGURATIONS

EACH OPERATOR HAS A UNIQUE CONTEXT IN WHICH IT MAY BE USED. THE CONTEXT IS DETERMINED BY THE TYPES OF THE ASSOCIATED OPERANDS. AS A RESULT 'OPERATORS ARE CLASSED AS 'ARITHMETIC', 'SET', 'ARITHMETIC TEST', 'SET TEST', AND 'LOGICAL'.

THE FOLLOWING TABLE DETERMINES THE TYPES OF OPERANDS ALLOWABLE WITH EACH CLASS OF OPERANDS.

<table>
<thead>
<tr>
<th>L-OPERAND TYPE</th>
<th>OPERATOR CLASS</th>
<th>R-OPERAND TYPE</th>
<th>RESULT TYPE</th>
</tr>
</thead>
<tbody>
<tr>
<td>ARITHMETIC</td>
<td>ARITHMETIC</td>
<td>ARITHMETIC</td>
<td>ARITHMETIC</td>
</tr>
<tr>
<td>SET</td>
<td>SET</td>
<td>SET</td>
<td>SET</td>
</tr>
<tr>
<td>ARITHMETIC</td>
<td>ARITHMETIC TEST</td>
<td>ARITHMETIC</td>
<td>LOGICAL</td>
</tr>
<tr>
<td>SET</td>
<td>SET TEST</td>
<td>SET</td>
<td>LOGICAL</td>
</tr>
<tr>
<td>LOGICAL</td>
<td>LOGICAL</td>
<td>LOGICAL</td>
<td>LOGICAL</td>
</tr>
</tbody>
</table>
OPERATOR DEFINITIONS AND PRECEDENCES

The operators which fall into these classes and their meanings are shown in the following table so that the order of computation in any complicated expression will be unambiguous. Each operator has a precedence (indicated by a precedence number) and operations with the highest precedence (number) are performed first. Operators with the same precedence number have equal precedence and are performed from left to right.

OPERATOR DEFINITION TABLE

<table>
<thead>
<tr>
<th>OPERATOR PRECEDENCE</th>
<th>USE</th>
<th>INTERPRETATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>** **</td>
<td>70</td>
<td>BINARY</td>
</tr>
<tr>
<td>* *</td>
<td>65</td>
<td>UNARY</td>
</tr>
<tr>
<td>* -</td>
<td>65</td>
<td>UNARY</td>
</tr>
<tr>
<td>** **</td>
<td>60</td>
<td>BINARY</td>
</tr>
<tr>
<td>**</td>
<td>55</td>
<td>BINARY</td>
</tr>
<tr>
<td>/</td>
<td>50</td>
<td>BINARY</td>
</tr>
<tr>
<td>+</td>
<td>45</td>
<td>BINARY</td>
</tr>
<tr>
<td>-</td>
<td>45</td>
<td>BINARY</td>
</tr>
<tr>
<td>AND</td>
<td>40</td>
<td>BINARY</td>
</tr>
<tr>
<td>OR</td>
<td>35</td>
<td>BINARY</td>
</tr>
<tr>
<td>AND NOT</td>
<td>30</td>
<td>BINARY</td>
</tr>
<tr>
<td>= =</td>
<td>25</td>
<td>BINARY</td>
</tr>
<tr>
<td>=</td>
<td>25</td>
<td>BINARY</td>
</tr>
<tr>
<td>&gt;=</td>
<td>25</td>
<td>BINARY</td>
</tr>
<tr>
<td>&lt;=</td>
<td>25</td>
<td>BINARY</td>
</tr>
<tr>
<td>&gt;</td>
<td>25</td>
<td>BINARY</td>
</tr>
<tr>
<td>&lt;</td>
<td>25</td>
<td>BINARY</td>
</tr>
<tr>
<td>IN</td>
<td>20</td>
<td>BINARY</td>
</tr>
<tr>
<td>NOT</td>
<td>15</td>
<td>UNARY</td>
</tr>
<tr>
<td>AND</td>
<td>10</td>
<td>BINARY</td>
</tr>
<tr>
<td>OR</td>
<td>9</td>
<td>BINARY</td>
</tr>
</tbody>
</table>
EACH COMPUTATIONAL EXPRESSION HAS THE FORM
<(L-OPERAND)OPERATOR>R-OPERAND>

THIS SECTION DESCRIBES THE RESTRICTIONS PLACED UPON EACH
OPERAND AND SOME ADDITIONAL PROPERTIES OF THE RESULTS.

ARITHMETIC OPERATORS

THE CURRENT VERSION OF MPL RESTRICTS ARITHMETIC DATA STRUCTURES
TO TWO DIMENSIONS. THIS RESTRICTION ALLOWS CONSIDERABLE IMPLICIT
COMPUTING POWER WITHOUT BEING OVERLY RESTRICTIVE FOR MATHEMATICAL
PROGRAMMING APPLICATIONS. THUS ALL ARITHMETIC DATA STRUCTURES
(EVEN THE CONSTANT 15) CAN BE VISUALIZED AS MATRICES.

<table>
<thead>
<tr>
<th>OPERATOR PART</th>
<th>CHARACTERISTICS</th>
</tr>
</thead>
<tbody>
<tr>
<td>L-OPERAND</td>
<td>ANY ARITHMETIC QUANTITY.</td>
</tr>
<tr>
<td>R-OPERAND</td>
<td>AN ARITHMETIC QUANTITY WITH THE SAME NUMBER OF COLUMNS AS THE L-OPERAND.</td>
</tr>
<tr>
<td>RESULT</td>
<td>THE VERTICAL CONCATENATION OF THE TWO OPERANDS. IT HAS THE SAME NUMBER OF COLUMNS AS EACH OPERAND AND THE NUMBER OF ROWS EQUAL TO THE SUM OF THE NUMBERS OF ROWS IN EACH OPERAND.</td>
</tr>
<tr>
<td>L-OPERAND</td>
<td>NONE.</td>
</tr>
<tr>
<td>R-OPERAND</td>
<td>ANY ARITHMETIC QUANTITY.</td>
</tr>
<tr>
<td>RESULT</td>
<td>SAME AS R-OPERAND.</td>
</tr>
<tr>
<td>L-OPERAND</td>
<td>NONE.</td>
</tr>
<tr>
<td>R-OPERAND</td>
<td>ANY ARITHMETIC QUANTITY.</td>
</tr>
<tr>
<td>RESULT</td>
<td>THE R-OPERAND WITH ALL ELEMENT VALUE SIGNS REVERSED.</td>
</tr>
<tr>
<td>L-OPERAND</td>
<td>ANY ARITHMETIC QUANTITY WITH THE SAME NUMBER OF ROWS AND COLUMNS. THUS THE L-OPERAND MAY BE EITHER A SQUARE MATRIX OR A 'SCALAR'.</td>
</tr>
<tr>
<td>R-OPERAND</td>
<td>MUST BE A SCALAR (ONE ROW AND ONE COLUMN) WITH A NON-NEGATIVE VALUE.</td>
</tr>
<tr>
<td>RESULT</td>
<td>THE L-OPERAND MULTIPLIED BY ITSELF THE NUMBER OF TIMES SPECIFIED BY THE R-OPERAND. IF THE L-OPERAND HAS MORE THAN ONE ROW AND COLUMN ANY FRACTIONAL PARTITION OF THE R-OPERAND WILL BE DROPPED. OTHERWISE THE L-OPERAND IS A SCALAR AND ANY POSITIVE VALUES FOR THE R-OPERAND ARE ALLOWED.</td>
</tr>
</tbody>
</table>
2-5-3  SEMANTICS (CONTINUED)

OPERATOR PART  CHARACTERISTICS

*+  L-OPERAND  ANY ARITHMETIC QUANTITY.
R-OPERAND  ANY ARITHMETIC QUANTITY WITH THE SAME NUMBER OF
           ROWS AS THE L-OPERAND HAS COLUMNS EXCEPT THAT
           EITHER OPERAND MAY BE A SCALAR.
RESULT  AN ARITHMETIC QUANTITY WITH THE SAME NUMBER
        OF ROWS AS THE L-OPERAND AND THE SAME NUMBER
        OF COLUMNS AS THE R-OPERAND. ELEMENT VALUES ARE
        THE RESULT OF CONVENTIONAL MATRIX MULTIPLICATION.
        IF EITHER OPERAND IS A SCALAR THE RESULT HAS
        THE SAME NUMBER OF ROWS AND COLUMNS AS THE OTHER
        OPERAND.

*+  L-OPERAND  ANY ARITHMETIC QUANTITY.
R-OPERAND  ANY SCALAR ARITHMETIC QUANTITY.
RESULT  HAS ALL THE PROPERTIES OF THE L-OPERAND
        EXCEPT THAT ALL ELEMENT VALUES HAVE BEEN
        DIVIDED BY THE R-OPERAND.

*+  L-OPERAND  ANY ARITHMETIC QUANTITY.
R-OPERAND  ANY ARITHMETIC QUANTITY WITH THE SAME NUMBER
           OF ROWS AND COLUMNS AS THE L-OPERAND.
RESULT  AN ARITHMETIC QUANTITY WITH THE PROPERTIES
        OF THE L-OPERAND. ALL POINTERS ARE SET TO ZERO.

SAME AS *++ (BINARY)

SET OPERATORS

OPERATOR PART  CHARACTERISTICS

*  AND  L-OPERAND  ANY SET.
R-OPERAND  ANY SET.
RESULT  A SET CONTAINING ONLY THOSE ELEMENTS WHICH
        AppeARED IN BOTH THE L-OPERAND AND THE R-OPERAND.

*  EX  L-OPERAND  ANY SET.
R-OPERAND  ANY SET.
RESULT  A SET CONTAINING ALL ELEMENTS WHICH AppeARED
        IN EITHER THE L-OPERAND, THE R-OPERAND OR BOTH.

*  OR  L-OPERAND  ANY SET.
R-OPERAND  ANY SET.
RESULT  A SET CONTAINING ALL ELEMENTS WHICH AppeARED
        IN THE L-OPERAND BUT NOT IN THE R-OPERAND.
**ARITHMETIC TEST OPERATORS**

ARITHMETIC TEST OPERATORS IMPOSE THREE DIFFERENT REQUIREMENTS ON THEIR TWO OPERANDS. TO SATISFY THESE REQUIREMENTS BOTH OPERANDS ARE TREATED AS MATRICES. THESE REQUIREMENTS ARE:

1. THE TWO OPERANDS HAVE THE SAME NUMBER OF ROWS.
2. THE TWO OPERANDS HAVE THE SAME NUMBER OF COLUMNS.
3. THE SPECIFIED RELATIONSHIP HOLDS WITHIN EACH PAIR OF CORRESPONDING (L-OPERAND, R-OPERAND) ELEMENTS.

<table>
<thead>
<tr>
<th>OPERATOR PART</th>
<th>CHARACTERISTICS</th>
</tr>
</thead>
<tbody>
<tr>
<td>L-OPERAND</td>
<td>ANY ARITHMETIC QUANTITY.</td>
</tr>
<tr>
<td>R-OPERAND</td>
<td>ANY ARITHMETIC QUANTITY.</td>
</tr>
<tr>
<td>RESULT</td>
<td>A LOGICAL QUANTITY WHICH IS TRUE ONLY IF REQUIREMENTS 1), 2), AND 3) ARE SATISFIED USING THE EQUALITY RELATIONSHIP.</td>
</tr>
</tbody>
</table>

| L-OPERAND     | ANY ARITHMETIC QUANTITY. |
| R-OPERAND     | ANY ARITHMETIC QUANTITY. |
| RESULT        | A LOGICAL QUANTITY WHICH IS FALSE ONLY IF REQUIREMENTS 1), 2), AND 3) ARE SATISFIED USING THE EQUALITY RELATIONSHIP. |

| L-OPERAND     | ANY ARITHMETIC QUANTITY. |
| R-OPERAND     | ANY ARITHMETIC QUANTITY. |
| RESULT        | A LOGICAL QUANTITY WHICH IS TRUE ONLY IF REQUIREMENTS 1), 2), AND 3) ARE SATISFIED USING THE GREATER THAN OR EQUAL RELATIONSHIP. AN ERROR CONDITION EXISTS IF EITHER OF REQUIREMENTS 1) AND 2) IS NOT SATISFIED. |

SAME AS '>=1' EXCEPT THAT THE RELATIONSHIP FOR REQUIREMENT 3) IS LESS THAN OR EQUAL.

SAME AS '>=1' EXCEPT THAT THE RELATIONSHIP FOR REQUIREMENT 3) IS STRICTLY GREATER THAN.

SAME AS '>=1' EXCEPT THAT THE RELATIONSHIP FOR REQUIREMENT 3) IS STRICTLY LESS THAN.
### SET TEST OPERATORS

<table>
<thead>
<tr>
<th>OPERATOR PART</th>
<th>CHARACTERISTICS</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>IN</strong></td>
<td></td>
</tr>
<tr>
<td><strong>L-OPERAND</strong></td>
<td>ANY SFT.</td>
</tr>
<tr>
<td><strong>R-OPERAND</strong></td>
<td>ANY SFT.</td>
</tr>
<tr>
<td><strong>RESULT</strong></td>
<td>A LOGICAL QUANTITY WHICH IS TRUE ONLY IF ALL ELEMENTS OF THE L-OPERAND ARE ALSO ELEMENTS OF THE R-OPERAND.</td>
</tr>
</tbody>
</table>

### LOGICAL OPERATORS

<table>
<thead>
<tr>
<th>OPERATOR PART</th>
<th>CHARACTERISTICS</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>NOT</strong></td>
<td></td>
</tr>
<tr>
<td><strong>L-OPERAND</strong></td>
<td>NONE.</td>
</tr>
<tr>
<td><strong>R-OPERAND</strong></td>
<td>ANY LOGICAL QUANTITY.</td>
</tr>
<tr>
<td><strong>RESULT</strong></td>
<td>A LOGICAL QUANTITY WHICH IS FALSE IF THE R-OPERAND IS TRUE AND IS TRUE IF THE R-OPERAND IS FALSE.</td>
</tr>
</tbody>
</table>

| **AND**       |                 |
| **L-OPERAND** | ANY LOGICAL QUANTITY. |
| **R-OPERAND** | ANY LOGICAL QUANTITY. |
| **RESULT**    | A LOGICAL QUANTITY WHICH IS TRUE ONLY IF BOTH THE L-OPERAND AND THE R-OPERAND VALUES ARE TRUE. |

| **OR**        |                 |
| **L-OPERAND** | ANY LOGICAL QUANTITY. |
| **R-OPERAND** | ANY LOGICAL QUANTITY. |
| **RESULT**    | A LOGICAL QUANTITY WHICH IS FALSE ONLY IF BOTH THE L-OPERAND AND THE R-OPERAND VALUES ARE FALSE. |
MPL contains constructions which are not properly classed as computational expressions, but which are used to combine variables, constants, or more complicated expressions into larger expressions.

2-5-1 Domain Items

\[ \langle \text{DOMAIN ITEM} \rangle ::= \langle \text{EXPRESSION} \rangle, \ldots, \langle \text{EXPRESSION} \rangle \]

Domain items have values which are sets. The sets are specified by specifying the lowest and highest valued elements and assuming that all intermediate valued elements are in the set. Both expressions should have scalar arithmetic values and only the whole number parts of these will be used. The value of the first expression should be less than the second. If the expression values are equal the set will contain one element. If the first expression is greater than the second the set will be empty.

Example domain items:

\( [1, \ldots, 10] \)
\( [10, \ldots, L-1] \)
\( [\text{HERE}, \ldots, \text{THEPE}] \)

2-5-2 Concatenator

\[ \langle \text{CONCATENATOR} \rangle ::= \langle \text{EXPRESSION LIST} \rangle \]

A concatenator has an arithmetic value. It allows the construction of arithmetic data structures by the explicit horizontal concatenation (adjacent placement) of several smaller structures with the same number of rows. The indices of the resulting structure begin at one. Vertical concatenation is accomplished using the operator \( \# \).

Example concatenators:

\( [1, 3, 4, 8, 16] \)
\( [3*1, 5*1, 2*1+3, 1+1, 13, 69] \)
\( [4, 8] \)
2-6-3  ARRAY CONSTRUCTOR

<Array Constructor>::=("<Expression>" "<For Phrase>")

An array constructor has an arithmetic value. It allows the construction of arithmetic data structures by the implicit horizontal concatenation of several expression values. Thus all expressions being concatenated must have the same number of rows. The for-phrase (3-2-5-2) governs the iterative process which provides values to be concatenated.

Example array constructors:
(A(*I)+B FOR I IN S)
(B(I) FOR I IN (1,...,N))
(C(J) FOR J IN S|F(J) >= D)

2-6-4  SUBSET SPECIFIER

<Subset Specifier>::=("<Variable Name>" IN "<Expression>"
                      |"<Expression>")

Subset specifiers produce sets. They form sets from larger sets by selecting elements with a given property. The variable name represents elements selected from the parent set so that they may be tested for the property. The first expression determines the parent set and must be set valued. The second expression tests the property and must be logical valued. Only those elements in the parent set for which the logical expression is true are included in the new set.

Example subset specifiers:
(J IN S | A(J,K) <= R)
(J IN S | J >= D AND J <= Y)
PROGRAM CONSTRUCTION

PROGRAM ::= 'PROCEDURE 'PROCEDURE IDENTIFIER' <STATEMENT SEQUENCE>'FINI'';
| PROGRAM 'PROCEDURE 'PROCEDURE IDENTIFIER' <STATEMENT SEQUENCE>'FINI'';

PROCEDURE IDENTIFIER ::= '<VARIABLE NAME>
| '<VARIABLE NAME LIST>'

VARIABLE NAME LIST ::= '<VARIABLE NAME>
| '<VARIABLE NAME LIST>', '<VARIABLE NAME>

A PROGRAM IN MPL IS A COMPLETE STATEMENT OF AN ALGORITHM AND IS MADE UP OF ONE OR MORE PROCEDURE DEFINITIONS. IT IS ASSUMED THAT THE PROGRAM BEGINS WITH THE FIRST PROCEDURE SO DEFINED. IN THE CURRENT VERSION OF THE LANGUAGE PROCEDURE DEFINITIONS MAY NOT BE NESTED (APPEAR WITHIN OTHER PROCEDURE DEFINITIONS) ALTHOUGH PROCEDURE CALLS MAY BE NESTED TO ANY DEPTH (PROCEDURE A CALLS PROCEDURE B WHICH CALLS PROCEDURE C, ETC.).

PROCEDURE DEFINITIONS BEGIN WITH THE KEYWORD 'PROCEDURE' AND END WITH THE KEYWORD 'FINI'. NOTE THAT PROCEDURE DEFINITIONS HAVE THE SAME GENERAL FORM AS A COMPLEX KEYWORD STATEMENT (3-2-5).

THE PROCEDURE IDENTIFIER PROVIDES NAMES FOR THE PROCEDURE AS WELL AS FOR THE INFORMATION WHICH WILL BE PASSED TO THE PROCEDURE BY A CALLING PROGRAM. WHEN THE PROCEDURE IS CALLED THE PARAMETER EXPRESSIONS (SEE PROCEDURE CALLS (2-4)) ARE EVALUATED AND THESE VALUES ARE USED IN THE CALLED PROCEDURE WHEREVER THEIR REPRESENTATIVE NAMES OCCUR.

EXAMPLE PROGRAM COMPOSED OF TWO PROCEDURES

PROCEDURE PROG
  ***
  SUB(J,K);
  ***
  FINI;
PROCEDURE SUB(E,F)
  ***
  RETURN;
  ***
  FINI;

STATEMENT SEQUENCES

STATEMENT SEQUENCE ::= '<STATEMENT>', '<STATEMENT SEQUENCE>', '<STATEMENT>

A STATEMENT SEQUENCE IS A SEQUENCE OF ONE OR MORE STATEMENTS. THIS CONCEPT IS USEFUL FOR DEFINING PROGRAMS (3) AND COMPLEX KEYWORD STATEMENTS (3-2-5).
3-2 STATEMENTS

<STATEMENT>::=<LABEL>**<STATEMENT>
|<ASSIGNMENT STATEMENT>
|<PROCEDURE CALL STATEMENT>
|<KEYWORD STATEMENT>

STATEMENTS IN MPL DETERMINE THE SEQUENCE OF OPERATIONS WHICH MAKES A PROGRAM MEANINGFUL.

3-2-1 LABELS

<LABEL>::=<VARIABLE NAME>|<DIGIT STRING>|

LABELS ARE EITHER VARIABLE NAMES OR STRINGS OF DIGITS ENCLOSED IN PARENTHESES. SINCE MPL IS WRITTEN IN A FREE FORMAT, A LABEL MUST BE SEPARATED FROM THE FOLLOWING STATEMENT BY A COLON ':'. LABELS MAY ONLY BE REFERENCED BY 'GO TO' STATEMENTS (3-2-4-2).

EXAMPLE LABELED STATEMENTS

LABEL: VAR:=EXP;
LOCATION_B: VAR2:=EXP2;
(13): VAR3:=EXP3;

3-2-2 ASSIGNMENT STATEMENTS

<ASSIGNMENT STATEMENT>::=<VARIABLE>**<EXPRESSION>**;
|<VARIABLE>**<EXPRESSION>**<FOR PHRASE>**;
|<VARIABLE>**<EXPRESSION>**<IF<EXPRESSION>**;
|<VARIABLE>**<EXPRESSION>**WHERE<SYMBOL SUBSTITUTE>**;


EXAMPLE: ASSIGNMENT STATEMENTS

A:=9;
MATRX:=(A,H)
(C,C);
YES OR NO:=MATRX:=INVERSE(A)
SET1:=SET2 AND SET3 OR SET4;
THE ASSIGNMENT STATEMENT HAS SEVERAL MODIFIED FORMS WHICH ARE PROVIDED TO MAKE MPL A MORE 'NATURAL' LANGUAGE.

THE ITERATED ASSIGNMENT STATEMENT

THE ITERATED ASSIGNMENT STATEMENT PROVIDES A METHOD FOR ITERATIVELY PERFORMING AN ASSIGNMENT. THIS FORM IS EQUIVALENT TO THE SHORT FORM ITERATED STATEMENT (3-2-5-2). FOR PHRASES ARE ALSO DISCUSSED IN (3-2-5-2).

EXAMPLE ITERATED ASSIGNMENT STATEMENTS

\[
A(P\_ROW,J) := A(P\_ROW,J)/A(P\_ROW,P\_COL) \text{ FOR } J \text{ IN COLDOM(A)};
\]

\[
A(I,*):=A(I,*)-A(I,P\_COL)*A(P\_ROW,*) \text{ FOR } I \text{ IN ROWNUM(A)};
\]

THE CONDITIONED ASSIGNMENT STATEMENT

THE CONDITIONED ASSIGNMENT STATEMENT ALLOWS THE SPECIFICATION OF A CONDITION UNDER WHICH AN ASSIGNMENT WILL OCCUR. THIS FORM IS EQUIVALENT TO THE SHORT FORM OF THE CONDITIONED STATEMENT (3-2-5-1).

EXAMPLE CONDITIONED ASSIGNMENT STATEMENTS

\[
x(I):=B-A(I,J) \text{ IF } X(J)=1;
\]

\[
x(I):=R(I) \text{ IF } R(I)>0;
\]

THE ASSIGNMENT STATEMENT WITH SYMBOL SUBSTITUTION

THE ASSIGNMENT STATEMENT WITH SYMBOL SUBSTITUTION ALLOWS THE USER TO REDUCE THE APPARENT COMPLEXITY OF EXPRESSIONS BY USING A SINGLE SYMBOL TO REPRESENT A LARGE AND COMPLEX STRING OF CHARACTERS AS DEFINED BY THE SYMBOL SUBSTITUTOR FOLLOWING THE 'WHERE' (SEE (3-2-4-1) FOR A DEFINITION OF SYMBOL SUBSTITUTORS). ONLY A SINGLE SUBSTITUTION IS ALLOWED SINCE THE 'LET' STATEMENT TERMINATOR ALSO TERMINATES THE STRING TO BE SUBSTITUTED. THIS FORM IS SIMILAR TO USING A 'LET' STATEMENT EXCEPT THAT THE (SYMBOL,CHARACTER STRING) EQUIVALENCE ONLY HOLDS WITHIN THE ASSIGNMENT STATEMENT DEFINING IT.

EXAMPLE ASSIGNMENT STATEMENTS WITH SYMBOL SUBSTITUTION

\[
R:=P+Q \text{ WHERE P:=INVERSE((A,B,C,D))};
\]

IMPLICIT DEFINE STATEMENT

IF A VARIABLE FIRST APPEARS AS LEFT MEMBER OF AN ASSIGNMENT STATEMENT WITHOUT ITS TYPE STRUCTURE AND STORAGE REQUIREMENTS HAVING BEEN PREVIOUSLY DECLARED BY A DEFINE STATEMENT (3-2-4-4) THESE REQUIREMENTS ARE DETERMINED BY THE EXPRESSION THAT APPEARS AS RIGHT MEMBER. THE IMPLICIT DEFINE CONCEPT IS UNDER DEVELOPMENT AND WILL NOT BE DISCUSSED FURTHER.
3-2-3 PROCEDURE CALL STATEMENT

\[\text{PROCEDURE CALL STATEMENT} ::= \text{PROCEDURE CALL} \}

A PROCEDURE CALL STATEMENT CALLS A PROCEDURE WHICH DOES NOT RETURN
A VALUE (VS. THE PROCEDURE CALL WHICH CALLS A PROCEDURE FROM WITHIN
AN EXPRESSION). SINCE THE PROCEDURE CALL STATEMENT APPEARS
ALONE (NOT IN AN EXPRESSION), ANY VALUE RETURNED BY THE PROCEDURE
IS LOST.

EXAMPLE PROCEDURE CALL STATEMENTS

\[\text{PIVOT(A,P_Row,P_Col)};\]
\[\text{PROC1(A,B,C,D)};\]
\[\text{PROC2(I+J-3*K,J-2,WHAT_NOW,A,R,C)};\]

3-2-4 KEYWORD STATEMENTS

\[\text{KEYWORD STATEMENT} ::= \text{LET STATEMENT} \}

\[\langle \text{GO TO STATEMENT} \rangle\]
\[\langle \text{RETURN STATEMENT} \rangle\]
\[\langle \text{DEFINE STATEMENT} \rangle\]
\[\langle \text{REDEFINE STATEMENT} \rangle\]
\[\langle \text{CONDITIONED STATEMENT} \rangle\]
\[\langle \text{ITERATED STATEMENT} \rangle\]
\[\langle \text{BLOCK STATEMENT} \rangle\]

EACH KEYWORD STATEMENT BEGINS WITH AN MPL KEYWORD. THESE
STATEMENTS ARE DIVIDED INTO SIMPLE AND COMPLEX STATEMENTS. COMPLEX
STATEMENTS HAVE SPECIAL BEGINNING AND ENDING SYMBOLS AND CONTAIN
OTHER STATEMENTS WITHIN THEM. THIS SECTION DISCUSSES ONLY THE
SIMPLE KEYWORD STATEMENTS.

3-2-4-1 LET STATEMENT

\[\text{LET STATEMENT} ::= \langle \text{SYMBOIL SUBSTITUTER} \rangle \}

\[\langle \text{SAME LOCATION} \rangle \langle \text{VARIABLE NAME} \rangle \langle \text{VARIABLE NAME} \rangle ;\]
\[\langle \text{CHARACTER STRING} \rangle \]

\[\text{SYMBOIL SUBSTITUTER} ::= \langle \text{VARIABLE NAME} \rangle \langle \text{CHARACTER STRING} \rangle \]
\[\langle \text{VARIABLE NAME} \rangle \langle \text{VARIABLE NAME LIST} \rangle ;\]
\[\langle \text{CHARACTER STRING} \rangle \]

LET STATEMENTS DIFFER FROM OTHER MPL STATEMENTS BY MODIFYING
THE PROGRAM AT TRANSLATION TIME INSTEAD OF EXECUTION TIME. THEY
CAN MAKE A PROGRAM EASIER TO WRITE AND/OR MORE READABLE BY
ALLOWING THE PROGRAMMER TO REPRESENT CHARACTER STRINGS BY SYMBOLS.

THE TWO PARTS OF A SYMBOL SUBSTITUTER ARE THE CHARACTER STRING (1-3) TO THE
RIGHT IF THE ASSIGNMENT SYMBOL AND THE IDENTIFIER TO THE LEFT.
THE IDENTIFIER PROVIDES A NAME FOR THE CHARACTER STRING AND,
OPTIONALLY, NAMES FOR PARAMETERS. IF THE STRING NAME IS DEFINED
WITHOUT PARAMETERS EVERY OCCURRENCE OF THE NAME IN THE FOLLOWING
TEXT WILL BE REPLACED BY THE CHARACTER STRING. THE PARAMETERS
ALLOW MODIFICATION OF THE CHARACTER STRING AT THE TIME OF REPLACEMENT WHEN OCCURRENCES OF THE PARAMETER NAMES IN THE CHARACTER STRING ARE REPLACED WITH THE CHARACTER STRINGS PROVIDED AS PARAMETERS WITH THE STRING NAME. IF COMMAS MUST APPEAR WITHIN THESE PARAMETER CHARACTER STRINGS, TWO MUST BE USED FOR EVERY INTENDED SINGLE OCCURRENCE. Thus \((a,b)\) AS A PARAMETER CHARACTER STRING IN A LET STATEMENT MUST BE WRITTEN \((a,,b)\), WHICH IS TO AVOID HAVING THE COMMA TREATED AS A PARAMETER SEPARATOR. THE SEMICOLON TERMINATES THE CHARACTER STRING AND SO MAY NOT OCCUR WITHIN IT.

AS A RATHER EXTREME EXAMPLE, THE STATEMENT

\[
\text{LET } A(C,1) := B(11*C(1));
\]

FOLLOWED BY

\[
D(K) := A(R+F,N);
\]

YIELDS

\[
D(K) := B(N)*R+F(J);
\]

WHILE THE STATEMENT

\[
\text{LET LOOP(VAR,START,INC,STOP) := FOR VAR:=START STEP INC UNTIL STOP DO;}
\]

FOLLOWED BY

\[
\text{LOOP(I,3*4+K,15,N) A(I) := B(I); ENDFOR;}
\]

YIELDS

\[
\text{FOR I :=3*J+K STEP 15 UNTIL N DO A(I) := B(I); ENDFOR;}
\]

CERTAINLY THESE ARE RATHER OBSCURE USES IN A MATHEMATICAL PROGRAMMING LANGUAGE, BUT THEY ARE INCLUDED TO GIVE THE READER IN IDEA OF THE POWER WHICH IS INHERENT IN THIS CONCEPT.

IN A MORE CONVENTIONAL USAGE THE STATEMENT

\[
\text{LET } A(T) := A(T,2*X);
\]

FOLLOWED BY

IF \(A(T)>0\), GO TO (5);

YIELDS

IF \(A(T,2)*X>0\), GO TO (5);

THE FORM USING THE KEYWORD *SAME LOCATION* INDICATES AN EQUIVALENCE BETWEEN THE TWO SYMBOLS WITHIN THE PARENTHESES.

A SHORT FORM OF LET STATEMENT USING INVERTED WORD ORDER WITH 'WHERE' INSTEAD OF 'LET', IS DISCUSSED UNDER (3-2-2).

3-2-4-2

GO TO STATEMENT

\[
\text{GO TO STATEMENT} := \text{GO TO '<LABEL>';}
\]

GO TO STATEMENTS ALTER THE NORMAL SEQUENTIAL FLOW OF PROGRAM EXECUTION BY TRANSFERRING CONTROL TO THE POINT IN THE PROGRAM INDICATED BY THE LABEL (3-2-1).

EXAMPLE - GO TO STATEMENTS

\[
\text{GO TO LUC3;}
\]

\[
\text{GO TO 1231;}
\]
3-2-4-3 RETURN STATEMENT

<RETURN STATEMENT> ::= 'RETURN''

THE RETURN STATEMENT RETURNS CONTROL FROM A CALLED PROCEDURE TO ITS CALLING PROCEDURE.

EXAMPLE USE OF THE RETURN STATEMENT IN A PROCEDURE

PROCEDURE EQUAL(I,A,I)
    IF DOM(A) = DOM(I) THEN
        EQUAL = FALSE;
        RETURN;
    ENDF;
    FOR I IN DOM(A),
            EQUAL = FALSE;
            RETURN;
        ENDF;
        EQUAL = TRUE;
        RETURN;
    FIN;

3-2-4-4 DEFINE STATEMENT

<DEFINE STATEMENT> ::= 'DEFINE' <VARIABLE NAME LIST><TYPE PHRASE><SHAPE PHRASE><SIZE PHRASE>

<TYPE PHRASE> ::= "ARITHMETIC" | "LOGICAL" | "SET" | "CHARACTER"
                | <NULL PHRASE>

<SHAPE PHRASE> ::= "RECTANGULAR" | "DIAGONAL" | "UPPER TRIANGULAR"
                   | "LOWER TRiANGULAR" | "ROW" | "COLUMN" | "SPARSE WITH"
                   | <EXPRESSION> | "NONZeros" | <NULL PHRASE>

<SIZE PHRASE> ::= <EXPRESSION> | <EXPRESSION> | <NULL PHRASE> | <NULL PHRASE> | <NULL PHRASE>

A VARIABLE NAME MAY BE USED IN A PROGRAM THE TYPE, STRUCTURE, AND STORAGE REQUIREMENTS OF THE VALUES WHICH IT REPRESENTS MUST BE DECLARED. THE ONLY EXCEPTIONS ARE THE VARIABLES USED IN ITERATED STATEMENTS (3-2-5-2) AND ARRAY CONSTRUCTORS (7-6-31), AND SET ELEMENT REPRESENTATIVES USED IN SUBSET SPECIFIERS (2-6-4).

SEE IMPLICIT DEFINE ASSIGNMENT STATEMENT UNDER 3-2-2.

A VARIABLE NAME LIST IS DEFINED UNDER PROGRAMS (3).

THE TYPE PHRASE DETERMINES WHETHER THE VALUE OF THE VARIABLE IS TREATED AS AN ARITHMETIC, LOGICAL, SET, OR CHARACTER QUANTITY. IF THIS PHRASE IS OMITTED THE VALUE IS ASSUMED TO BE ARITHMETIC.

THE SHAPE PHRASE MAY ONLY BE USED WHEN DEFINING ARITHMETIC QUANTITIES AND DETERMINES THE STRUCTURE OF SPACE REQUIRED FOR STORING THE DATA AS WELL AS ITS ORGANIZATION. IF THE SHAPE
PHRASE IS OMITTED THE DEFAULT ASSUMPTIONS ARE:

<table>
<thead>
<tr>
<th>DIMENSION</th>
<th>DEFAULT SHAPE</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>RECTANGULAR</td>
</tr>
<tr>
<td>1</td>
<td>COLUMN</td>
</tr>
<tr>
<td>0</td>
<td>NONE</td>
</tr>
</tbody>
</table>

THE MODIFIERS 'RECTANGULAR', 'DIAGONAL', 'UPPER TRIANGULAR', AND 'LOWER TRIANGULAR' ARE ONLY MEANINGFUL WHEN DEFINING TWO DIMENSIONAL QUANTITIES (MATRICES) WHILE THE MODIFIERS 'ROW' AND 'COLUMN' ARE MEANINGFUL ONLY WHEN DEFINING ONE DIMENSIONAL QUANTITIES (VECTORS). THE MODIFIER 'SPARSE' CAN CONSERVE STORAGE WHEN THERE IS A PREDOMINANCE OF ZERO ELEMENTS IN THE ARRAY. THE EXPRESSION IN THE SPARSE MODIFIER MUST BE A SCALAR VALUED ARITHMETIC EXPRESSION IN THAT IT INDICATES THE NUMBER OF ELEMENTS OF THE SPARSE ARRAY WHICH ARE ACTUALLY TO BE KEPT.

THE SIZE PHRASE SPECIFIES THE NUMBER OF DIMENSIONS OF THE VARIABLE AS WELL AS THE RANGES OF THE INDICES ON EACH OF THESE DIMENSIONS. THE EXPRESSIONS IN THE SIZE PHRASE MUST BE EITHER DOMAIN ITEMS (2-6-1) OR SCALAR ARITHMETIC EXPRESSIONS. DOMAIN ITEMS GIVE BOTH THE UPPER AND LOWER BOUND ON THE RANGE OF THE SUBSCRIPT WHILE SCALAR ARITHMETIC EXPRESSIONS DETERMINE ONLY THE UPPER BOUND ON THE SUBSCRIPT RANGE AND A LOWER BOUND OF ONE IS ASSUMED. THE TYPE PHRASE, SHAPE PHRASE, AND SIZE PHRASE MAY APPEAR IN ANY ORDER IN A DEFINE STATEMENT.

EXAMPLE DEFINE STATEMENTS

```
DEFINE J,K ARITHMETIC;
DEFINE SET1,SET2,SET3 SET;
DEFINE STRING1 CHARACTER;
DEFINE A (1,...,M) BY (1,...,N);
DEFINE A M BY N;
DEFINE C N ROW;
DEFINE SPARSE A M BY N SPARSE WITH I*NONZEROS;
```

RELEASE STATEMENT

```
<RELEASE STATEMENT>;**RELEASE <VARIABLE NAME LIST>**;
```

THE RELEASE STATEMENT EXPLICITLY RELEASES THE STORAGE ALLOCATED BY OR AFTER THE CORRESPONDING DEFINE STATEMENT (3-2-4-4). IT IS IMPOSSIBLE TO RELEASE A VARIABLE WHICH WAS ALLOCATED OUTSIDE OF THE CURRENT BLOCK (3-2-4-3). RELEASE STATEMENTS REPEATING VARIABLE NAMES WHICH HAVE NOT BEEN DEFINED OR HAVE ALREADY BEEN RELEASED ARE IGNORED. THE RELEASE STATEMENT ALSO IMPLICITLY RELEASES ALL STORAGE WHICH WAS DEFINED AFTER ANY VARIABLE IN THE NAME LIST (56F (3-2-5-3) FOR AN EXAMPLE).

EXAMPLE RELEASE STATEMENTS

```
RELEASE A;
RELEASE A,B,C,D,P,TI
```
3-2-5

COMPLEX KEYWORD STATEMENTS

THE FOLLOWING SECTION DISCUSSES COMPLEX KEYWORD STATEMENTS. THESE STATEMENTS ALL HAVE THE FORM

<INTRODUCTION><STATEMENT SEQUENCE><TERMINATION>

3-2-5-1

CONDITIONED STATEMENT

<CONDITIONED STATEMENT>::=<IF 'EXPRESSION' THEN <STATEMENT SEQUENCE> OR IF SEQUENCE><OTHERWISE PHRASE> ENDIF ';

<OR IF SEQUENCE>::=<NULL PHRASE> OR IF 'EXPRESSION' THEN <STATEMENT SEQUENCE>

<OTHERWISE PHRASE>::=OTHERWISE <STATEMENT SEQUENCE> <NULL PHRASE>

A CONDITIONED STATEMENT ALLOWS THE USER TO SELECT CONDITIONS UNDER WHICH STATEMENT(S) WILL BE EXECUTED. THE SHORT FORM IS USED ONLY WHEN A CONDITION GOVERNS THE EXECUTION OF A SINGLE STATEMENT. THE LONG FORM ALLOWS THE TESTING OF SEVERAL MUTUALLY EXCLUSIVE CONDITIONS. WHEN A CONDITION IS SATISFIED THE STATEMENTS FOLLOWING THE TEST ARE EXECUTED AND CONTROL PASSES TO THE END OF THE STATEMENT. THE EXPRESSIONS FOLLOWING THE "IF" AND THE KEYWORD "OR IF" ARE LOGICAL VALUED SPECIFICALLY THE LOGICAL EXPRESSIONS FOLLOWING THE "IF" IS EVALUATED AND IF TRUE THE FOLLOWING STATEMENT SEQUENCE IS EXECUTED AND CONTROL THEN PASSES TO THE "ENDIF". IF THE EXPRESSION IS FALSE THE EXPRESSION IN THE NEXT FOLLOWING "OR IF" IS EVALUATED WITH THE SAME ACTIONS. IF AN "OTHERWISE" IS ENCOUNTERED ALL STATEMENTS IMMEDIATELY FOLLOWING THE "OTHERWISE" ARE EXECUTED.

EXAMPLE CONDITIONED STATEMENTS

IF I=0, GO TO NON_ZERO;
IF A(i,j)=R, A(i,j)=A(*)+1;
IF A>R THEN
GO TO A_EQUAL_D;
ON IF A=C THEN
V0 TO A_NE_B BUT_EQ_C;
ON IF J->K AND K>3*R THEN
A=K;
OTHERWISE
A=A;
C=I;
GO TO NO_GOOD;
ENDIF;

SEE ALSO CONDITIONED ASSIGNED STATEMENT UNDER (3-2-2) WHERE A SHORT-IF FORM IN INVERTED ORDER IS DISCUSSED.
ITERATED STATEMENT

<ITERATED STATEMENT>::=<FOR PHRASE>::=<STATEMENT>

|<FOR PHRASE>::|<STATEMENT SEQUENCE>::|ENDFOR::|

<FOR PHRASE>:::=FOR "<VARIABLE NAME>" IN "<EXPRESSION>

|FOR "<VARIABLE NAME>" IN "<EXPRESSION>"

|FOR "<VARIABLE NAME>" IN "<EXPRESSION>" STEP 

<EXPRESSION> UNTIL "<EXPRESSION>"

THE FOR PHRASE GOVERNS THE INDEXING OF AN ITERATION. ONE OF THE
TWO FORMS INDICATES AN INDEXING OVER ELEMENTS OF A SET, NAMES THE
INDEX, SPECIFIES THE SET, AND ALLOWS ELEMENTS OF THE SET TO BE
SELECTIVELY DISCARDED. ON EACH CYCLE OF THE ITERATION THE INDEX
TAKES ON A NEW VALUE FROM THE SET. THIS INDEX MAY BE USED TO
AFFECT STATEMENTS WITHIN THE SCOPE OF THE ITERATION. SELECTIVE
DISCARDING OF ELEMENTS IS PERFORMED BY THE OPTIONAL EXPRESSION
FOLLOWING THE "SUCH THAT" SYMBOL ("|"), HENCE THE INDEX VARIABLE
AND FIRST EXPRESSION MUST BE SCALAR ARITHMETIC QUANTITIES, THE
SECOND EXPRESSION MUST BE SET VALUED, AND THE OPTIONAL THIRD
EXPRESSION MUST BE LOGICAL VALUED.

THE SECOND FORM SPECIFIES THE INDEXING IN A MORE CONVENTIONAL
MANNER IN WHICH THE INDEX IS GIVEN A STARTING VALUE FOR THE FIRST
CYCLE AND THAT VALUE IS INCREMENTED BY THE STEP ON EACH SUCCESSIVE
CYCLE. THE TERMINAL CONDITION IS TESTED ON EVERY CYCLE BEFORE
ANY ENCLOSED STATEMENTS ARE EXECUTED. EXECUTION OF THESE STATEMENTS
OCCURS AS LONG AS THE CONDITION IS NOT SATISFIED. THUS THE VARIABLE
NAME AND THE FIRST TWO EXPRESSIONS MUST BE SCALAR ARITHMETIC
QUANTITIES WHILE THE TERMINAL CONDITION EXPRESSION MUST BE LOGICAL
VALUED. THIS SECOND FORM DOES NOT PROVIDE AN ADDITIONAL TEST FOR
SCREENING INDICES.

EXAMPLE: ITERATED STATEMENTS

FOR I IN (1:*:*:M), A(I):=B(I,J);  
FOR I IN SET1 | I->P, FOR J IN SET2, A(I,J):=C;  
FOR I IN SET3 OR SET4 | B(I)>=0 ON  
9(I):=g(I);  
K:=K+1;  
ENDFOR;  
FOR K:=1 STEP 2 UNTIL K>=N, A(K):=B(K);

SEE ALSO ITERATED ASSIGNMENT STATEMENT UNDER (3-2-2) WHERE THE ABOVE FIRST (SHORT)
FORM IS DISCUSSED IN INVERTED ORDER.
3-2-5-3  BLOCK STATEMENT

<BLOCK STATEMENT>::= BLOCK ' <STATEMENT SEQUENCE> ' END BLOCK'';

Allocation and handling of storage is a major problem in MPL since it will be used to solve problems involving large amounts of data. The block statement allows the programmer to divide his procedures into blocks within which he can allocate (define (3-2-4-4)) storage. This space is automatically released when control leaves the block. In addition storage may be explicitly released (3-2-4-5) elsewhere in the block in which it was defined, but in no other block. In this case storage is released in an order opposite that of definition. Thus the sequence

DEFINE A;
DEFINE B;

RELEASE A;

causes both A and B to be released in that order. Notice that a procedure is an implied block statement.

EXAMPLE BLOCK STATEMENTS

BLOCK
DEFINE MATRIX M+1 BY N+1:
MATRIX:=(A,B,R,E); (C+2);
END BLOCK:  "Even though it is assumed that A, B, C, and Z are defined outside the block, this statement produces no usable results."
INPUT / OUTPUT

Very little work has yet been done on this section. It is currently thought that many ideas will be adopted from languages such as ALGOL, FORTRAN, or PL/I.
THIS SECTION DESCRIBES THE USE OF SEVERAL PROCEDURES WHICH ARE PROVIDED IN THE MPL LIBRARY. REFERENCES TO THESE PROCEDURES ALL HAVE THE FORM \texttt{F(P)} WHERE \texttt{F} REPRESENTS THE NAME OF THE PROCEDURE AND \texttt{P} REPRESENTS A LIST OF PARAMETERS. WHERE INDICATED THESE PROCEDURES RETURN VALUES WITH TYPE, SHAPE, AND FORM AS DESCRIBED BELOW.

\textbf{ARGMAX(VECTOR)}  
VECTOR AN ARITHMETIC EXPRESSION WITH A VECTOR VALUE.  
VALUE THE SCALAR ARITHMETIC INDEX OF THE FIRST OCCURRING MAXIMUM VALUED ELEMENT OF \texttt{VECTOR}.

\textbf{ARGMIN(VECTOR)}  
VECTOR ANY VECTOR VALUED ARITHMETIC EXPRESSION.  
VALUE THE SCALAR ARITHMETIC INDEX OF THE FIRST OCCURRING MINIMUM VALUED ELEMENT OF \texttt{VECTOR}.

\textbf{COLODIM(MATRIX)}  
MATRIX ANY ARITHMETIC EXPRESSION.  
VALUE THE SCALAR ARITHMETIC NUMBER OF ELEMENTS IN THE RANGE OF THE FIRST OR ONLY SUBSCRIPT OF \texttt{MATRIX}. THIS FUNCTION IS INTENDED FOR FINDING THE NUMBER OF COLUMNS IN A MATRIX. SO IF \texttt{MATRIX} IS A VECTOR OR SCALAR EXPRESSION, \texttt{V} := 1.

\textbf{DIM(VECTOR)}  
VECTOR ANY ARITHMETIC EXPRESSION.  
VALUE THE SCALAR ARITHMETIC NUMBER OF ELEMENTS IN THE RANGE OF THE FIRST OR ONLY SUBSCRIPT OF \texttt{VECTOR}. IF \texttt{VECTOR} IS MATRIX VALUED THIS PROCEDURE IS EQUIVALENT TO ROWDIM. IF \texttt{VECTOR} IS SCALAR VALUED, \texttt{V} := 1.

\textbf{IDENTITY(RANK)}  
\texttt{RANK} THE SCALAR ARITHMETIC RANK OF THE SQUARE IDENTITY MATRIX WHICH IS THE VALUE OF THE PROCEDURE.  
VALUE AN IDENTITY MATRIX WITH \texttt{RANK} ROWS AND COLUMNS.

\textbf{INVERSE(MATRIX)}  
\texttt{MATRIX} A SQUARE, NON-SINGULAR, MATRIX VALUED ARITHMETIC EXPRESSION.  
VALUE THE INVERSE OF \texttt{MATRIX}.

\textbf{MAX(VECTOR)}  
VECTOR A VECTOR VALUED ARITHMETIC EXPRESSION.  
VALUE THE SCALAR ARITHMETIC VALUE OF THE MAXIMUM VALUED ELEMENT OF \texttt{VECTOR}.

\textbf{MIN(VECTOR)}  
VECTOR ANY VECTOR VALUED ARITHMETIC EXPRESSION.  
VALUE THE SCALAR ARITHMETIC VALUE OF THE MINIMUM VALUED ELEMENT OF \texttt{MATRIX}. ALL POINTERS ARE IGNORED.
ONES(ROWS, COLUMNS)
ROWS THE SCALAR ARITHMETIC NUMBER OF ROWS IN V.
COLUMNS THE SCALAR ARITHMETIC NUMBER OF COLUMNS IN V.
VALUE A MATRIX OF ONES WITH ROWSROWS AND COLUMNS COLUMNS.

ROWSDIM(MATRIX)
MATRIX ANY ARITHMETIC EXPRESSION.
VALUE THE SCALAR ARITHMETIC NUMBER OF ELEMENTS IN THE RANGE
OF THE FIRST SUBSCRIPT OF MATRIX. THIS PROCEDURE IS
INTENDED FOR FINDING THE NUMBER OF ROWS IN A MATRIX,
BUT IS EQUIVALENT TO DIM(VECTOR) IF MATRIX IS ACTUALLY
VECTOR VALUED. IF MATRIX IS SCALAR VALUED, V := 1.

SUM(VECTOR)
VECTOR A VECTOR VALUED ARITHMETIC EXPRESSION.
VALUE THE SCALAR ARITHMETIC SUM OF THE ELEMENTS OF VECTOR.

TRANSPOSE(MATRIX)
MATRIX ANY ARITHMETIC EXPRESSION.
VALUE THE TRANSPOSE OF MATRIX. IF MATRIX HAS M ROWS AND
N COLUMNS THEN V HAS N ROWS AND M COLUMNS.

UNIT(SIZE, INDEX)
SIZE THE SCALAR ARITHMETIC NUMBER OF ELEMENTS IN VECTOR V.
INDEX THE SCALAR ARITHMETIC SUBSCRIPT OF THE SINGLE ONE VALUED
ELEMENT IN V, WHERE 1 <= INDEX <= SIZE.
VALUE AN ARITHMETIC COLUMN VECTOR WITH SUBSCRIPT RANGE
[1, ..., SIZE] WHICH HAS ALL ZERO ELEMENTS EXCEPT FOR THE
SINGLE ONE ELEMENT IN THE INDEXTH POSITION.

ZEROS(ROWS, COLUMNS)
ROWS THE SCALAR ARITHMETIC NUMBER OF ROWS IN V.
COLUMNS THE INTEGER SCALAR NUMBER OF COLUMNS IN V.
VALUE A MATRIX OF ZEROS WITH ROWS ROWS AND COLUMNS COLUMNS.

ALSO
SIZE...SCALAR ARITHMETIC VALUED PROCEDURE FOR FINDING THE
NUMBER OF ELEMENTS IN A SET.
SET...SET VALUED PROCEDURE FOR CONVERTING ARITHMETIC
QUANTITIES TO SETS.
DOM...SET VALUED PROCEDURE FOR INDEXING OVER VECTOR ELEMENTS.
ROWDOM...SET VALUED PROCEDURE FOR INDEXING OVER MATRIX ROWS.
COLDOM...SET VALUED PROCEDURE FOR INDEXING OVER MATRIX COLUMNS.
6-1 CARD FORMAT

MPL uses a 'FREE FORMAT' style which means that statements may be strung one immediately after the other, only separated by the ';' terminators. Thus much of the responsibility for an aesthetic and readable program rests on the writer.

When communicating the program to the computer on punch cards the program 'TEXT' must be confined to columns 1 through 72. Columns 73 through 80 may be used for identification since they will be ignored. This is a common programming convention.

6-2 USE OF BLANKS

Blanks are used as delimiters in MPL and are required where specified in the various definitions. In addition they may be inserted between any two symbols (items enclosed in primes in the metalanguage definition) but may not appear within variable names or key words except where specified.

Wherever a blank is allowed or required any number of multiple blanks is allowed.

6-3 COMMENTS

Comments may be placed anywhere in an MPL program since they are completely ignored by the computer. They are delimited on both ends by a quote ("THIS IS NOT A DOUBLE PRIME ('')"). Obvious care must be taken to insure that the terminal quote appears in its proper place.
7.1 RESUME OF DEFINITIONS

<ARRAY CONSTRUCTOR>::= ("<EXPRESSION>" <FOR PHRASE>);

<ASSIGNMENT STATEMENT>::=<VARIABLE>:=<EXPRESSION>;<FOR PHRASE>;

|<VARIABLE>:=<EXPRESSION> <FOR PHRASE>|
|<VARIABLE>:=<EXPRESSION> <IF <EXPRESSION>;
|<VARIABLE>:=<EXPRESSION> WHERE <SYMBOL SUBSTITUTER>;

<BLOCK STATEMENT>::=BLOCK <STATEMENT SEQUENCE> ENDBLOCK ;

<CHARACTER>::=<LETTER>|<DIGIT>|<SPECIAL CHARACTER>

1-2

<CHARACTER STRING>::=<"<CHARACTER STRING><CHARACTER>

1-3

<COMPUTATIONAL EXPRESSION>::=**<EXPRESSION>
|'*'<EXPRESSION>

|'<EXPRESSION>' <FOR PHRASE>
|'<EXPRESSION>'**<EXPRESSION>
|'<EXPRESSION>'/*<EXPRESSION>
|'<EXPRESSION>'/<<EXPRESSION>
|'<EXPRESSION>'**<EXPRESSION>
|'<EXPRESSION>'<EXPRESSION>
|'<EXPRESSION>' <FOR PHRASE>
|'<EXPRESSION>'=*<EXPRESSION>
|'<EXPRESSION>'**<EXPRESSION>
|'<EXPRESSION>'/*<EXPRESSION>
|'<EXPRESSION>'/<<EXPRESSION>
|'<EXPRESSION>'**<EXPRESSION>
|'<EXPRESSION>'<EXPRESSION>
|'<EXPRESSION>' <FOR PHRASE>

<CUNCATENATOR>::=('<EXPRESSION LIST>**')

2-5

<CONDITIONED STATEMENT>::='IF '<EXPRESSION>' THEN '<STATEMENT SEQUENCE>
|<IF <STATEMENT SEQUENCE> OTHERWISE PHRASE> ENDIF';

3-2-5-1

<DEFINITE STATEMENT>::='DEFINE <VARIABLE NAME LIST><TYPE PHRASE>

1-4-4

<DIGIT>::=0|1|2|3|4|5|6|7|8|9

1-2

<DIGIT STRING>::=<DIGIT>|<DIGIT STRING><DIGIT>

1-3

<NUMERIC LIT>::=<"<EXPRESSION>".....<EXPRESSION>'1

2-6-1

<EXPRESSION>::=<DIGIT STRING>

1-0-<DIGIT STRING>

2-2-1
7

RESUME OF DEFINITIONS (CONTINUED)

<EXPRESSION> ::= (*<EXPRESSION>*
|<NUMBER>
|*TRUE*|*FALSE*
|*NULL*
|***<CHARACTER STRING>***
|<VARIABLE>
|<PROCEDURE CALL>
|<COMPUTATIONAL EXPRESSION>
|<DOMAIN ITEM>
|<CONCATENATOR>
|<ARRY CONSTRUCTOR>
|<SUBSET SPECIFIER>

<EXPRESSION LIST> ::= <EXPRESSION> |<EXPRESSION LIST>*<EXPRESSION>

<FOR PHRASE> ::= *FOR* '<VARIABLE NAME>' IN '<EXPRESSION>'
|*FOR* '<VARIABLE NAME>' IN '<EXPRESSION>' |*<EXPRESSION>*<EXPRESSION>
|*FOR* '<VARIABLE NAME>' =<EXPRESSION>*<EXPRESSION>
|STEP *<EXPRESSION>* UNTIL '<EXPRESSION>'

<GO TO STATEMENT> ::= *GO TO* '<LABEL>'

<ITERATED STATEMENT> ::= *<FOR PHRASE>* DO '<STATEMENT SEQUENCE>*ENDFOR**

<KEYWORD STATEMENT> ::= *LET STATEMENT>
|<RETURN STATEMENT>
|<DEFINE STATEMENT>
|<RELEASE STATEMENT>
|<CONDITIONED STATEMENT>
|<ITERATED STATEMENT>
|<BLOCK STATEMENT>

<LET> ::= <VARIABLE NAME> |('<DIGIT STRING>**

<NULL PHRASE> ::= *<NULL PHRASE>*

<NUMBER> ::= (<NUMBER BASE><NUMBER BASE><EXONENT>

<NUMBER BASE> ::= <DIGIT STRING>
|<DIGIT STRING>*
|*<DIGIT STRING>*
|*<DIGIT STRING>*<DIGIT STRING>
I
7(3)

RESUME OF DEFINITIONS (CONTINUED)

<OR IF SEQUENCE>::=<NULL PHRASE>
|<OR IF SEQUENCE>|OR IF *<EXPRESSION>" THEN "
|<STATEMENT SEQUENCE>

<OTHERWISE PHRASE>::=*OTHERWISE "<STATEMENT SEQUENCE">
|<NULL PHRASE>

<PROCEDURE CALL>::=<VARIABLE NAME>
|<VARIABLE NAME>"(<"EXPRESSION LIST">)"

<PROCEDURE CALL STATEMENT>::=<PROCEDURE CALL>

<PROCEDURE IDENTIFIER>::=<VARIABLE NAME>
|<VARIABLE NAME>"(<"VARIABLE NAME LIST">)"

<PROGRAM>::=*"<PROCEDURE IDENTIFIER> STATEMENT SEQUENCE">FINI"
|<PROGRAM>"<PROCEDURE IDENTIFIER> STATEMENT SEQUENCE">FINI"

<RELEASE STATEMENT>::="RELEASE "<VARIABLE NAME LIST>";

<RETURN STATEMENT>::="RETURN";

<SHAPE PHRASE>::="RECTANGULAR"|"DIAGONAL"|"UPPER TRIANGULAR"
|"LOWER TRIANGULAR"|"ROW"|"COLUMN"|"SPARSE WITH"
|"<EXPRESSION>""NONZEROS"|<NULL PHRASE>

<SIZE PHRASE>::=<EXPRESSION>"BY"<EXPRESSION>
|<EXPRESSION>"|<NULL PHRASE>

<SPECIAL CHARACTER>::=\"*|\"0\"|\"1\"|\"2\"|\"3\"|\"4\"|\"5\"|\"6\"|\"7\"|\"8\"|\"9\"

<STATEMENT>::="LABEL"":<STATEMENT>
|<ASSIGNMENT STATEMENT>
|<PROCEDURE CALL STATEMENT>
|<KEYWORD STATEMENT>

<STATEMENT SEQUENCE>::=<STATEMENT>|<STATEMENT SEQUENCE>"STATEMENT>";

<SUBSCRIPT ELEMENT>::=""|<EXPRESSION>

<SUBSCRIPT LIST>::=<SUBSCRIPT ELEMENT>
|<SUBSCRIPT LIST>""|<SUBSCRIPT ELEMENT>

<SUBSET SPECIFIER>::="<"VARIABLE NAME>" IN "<EXPRESSION>
|"<EXPRESSION>

<SYMBOL SUBSTITUTE>::="<VARIABLE NAME>"="""CHARACTER STRING"
|<VARIABLE NAME>"("VARIABLE NAME LIST")"="""CHARACTER STRING>

<TYPE PHRASE>::="""ARITHMETIC""|"LOGICAL"|"SET"|"CHARACTER"
|<NULL PHRASE>

3-2-4-4
RESUME OF DEFINITIONS (CONTINUED)

<VARIABLE>::=<VARIABLE NAME> | <VARIABLE>*(<"SUBSCRIPT LIST">)*

<VARIABLE NAME>::=<LETTER>
    | <VARIABLE NAME><LETTER>
    | <VARIABLE NAME><DIGIT>
    | <VARIABLE NAME>-.'
    | <VARIABLE NAME>@'

<VARIABLE NAME LIST>::=<VARIABLE NAME>
    <VARIABLE NAME LIST>*,<VARIABLE NAME>

THIS STATEMENT IS NOT PART OF THE FORMAL DEFINITION, BUT IS INCLUDED FOR REFERENCE.

<KEYWORD>::=' ARITHMETIC'
    | 'BLOCK'
    | 'BY'
    | 'CHARACTER'
    | 'COLUMN'
    | 'DEFINE'
    | 'DIAGONAL'
    | 'DO'
    | 'END BLOCK'
    | 'ENDIF'
    | 'END FOR'
    | 'FALSE'
    | 'FINI'
    | 'FOR'
    | 'GO TO'
    | 'IF'
    | 'IN'
    | 'LET'
    | 'LOGICAL'
    | 'LOWER TRiANGULAR'
    | 'NULL'
    | 'NONZEROS'
    | 'OR IF'
    | 'OTHERWISE'
    | 'RECOGNIZE'
    | 'RECTANGULAR'
    | 'RELEAF'
    | 'ROW'
    | 'SAME LOCATION'
    | 'SET'
    | 'SPARSE WITH'
    | 'STEP'
    | 'THEN'
    | 'TRUE'
    | 'UNTIL'
    | 'UPPER TRiANGULAR'
    | 'WHERE'
SAMPLE MPL PROGRAMS

PROCEDURE REVISED SIMPLEX(MATRIX, COSTS, RHS, BASIC_VARIABLES, UNBOUNDED, OBJECTIVE_VALUE, ITERATIONS)
DEFINE I, J: "THESE ARE INDICES LATER ON"
UNBOUNDED := FALSE; ITERATIONS := C;
LET P := MATRIX;
LET C := COSTS;
LET Q := RHS;
LET BV := BASIC_VARIABLES;
LET M := ROWDIM(P);
LET N := COLDIM(P);

"WE ASSUME THAT BV CONSTITUTES A FEASIBLE SET OF BASIC VARIABLES GIVEN BY THEIR INDICES."
"WE WISH TO FIND X >= 0 SUCH THAT P*X = Q"
"WHICH MINIMIZES C*X = OBJECTIVE_VALUE. FIRST WE CALCULATE THE INVERSE OF THE RACES."
DEFINE INV_B M BY M;
INV_B := INVERSE(P(*P:*Q));

"THE CURRENT RIGHT HAND SIDE IS"
Q := INV_B * Q;

"THE CORRESPONDING COST VECTOR IS"
DEFINE CB M ROW;
CB := C(R:Q);

"S IS THE INDEX OF THE INCOMING COLUMN"
"R IS THE INDEX OF THE OUTGOING COLUMN."
DEFINE S, R;

PRICING BLOCK
ITERATIONS := ITERATIONS + 1;

"FIND THE SIMPLEX MULTIPLIERS *SM*"
DEFINE SM M ROW;
SM := CB * INV_B;

"AND THE SMALLEST RELATIVE COST FACTOR"
S := ARGMINIC(SM:*P);

"TEST FOR OPTIMALITY OF THE CURRENT BASIS"
IF C(S) > SM(P, S) THEN
"WE HAVE FOUND THE OPTIMAL BASIS"
OBJECTIVE_VALUE := C*B*Q;
RETURN;
END IF;
END BLOCK;

"NOW COLUMN S IS INTRODUCED INTO THE BASIS."
"PB IS THE REPRESENTATION OF P(*S) IN TERMS OF THE CURRENT BASIS"
DEFINE PR M COLUMN;
PR := INV_A*PB(*S);
R := Q;
R := ARGMIN(Q/PR P*S) FOR I IN (1, ..., M | P(I, S) > 0);
SAMPLE MPL PROGRAM (CONTINUED)

"IF ALL P(i,s)<0, THEN WE STILL HAVE R=0 AND
A CLASS OF SOLUTIONS APPROACHING MINUS INFINITY
EXISTS"

IF R=0 THEN
  UNRounded := TRUE;
  RETURN;
ENDIF;

"NOW UPDATE THE BASIC VARIABLE LIST BV, THE COST
ASSOCIATED WITH THE BASIS L
VECTOR CB ASSOCIATED WITH THE BASIS, THE VALUES
Q OF THE BASIC VARIABLES, AND THE INVERSE
INV_B OF THE BASIS."

BV(R):=S;
CB(R):=C(s);

"UPDATE Q"
FOR J IN (1,...,M) | J!=R, Q(J):= Q(J)-PB*(Q(R)/P(R,S));
Q(R):=Q(R)/PB(R,S);

"NOW UPDATE THE BASIS INVERSE"
PIVOT(INV_B, PB, R);

"NOW THE CYCLE IS COMPLETE AND WE RETURN TO
CHECK THE OPTIMALITY OF THE NEW BASIS."
GO TO PRICING;
FINIS;

PROCEDURE PIVOT(MATRIX, PIVOT_COL, PIVOT_ROW)
LET M := MATRIX;
LET P := PIVOT_COL;
LET R := PIVOT_ROW;
FOR I IN ROWNDER(M) | I!=R, M(I,:)=M(R,:)*P(I)/P(R);
M(R,:)=M(R,:)/P(R);
RETURN;
FINIS:
The purpose of MPL is to provide a language for writing mathematical programming algorithms that will be easier to write, to read, and to modify than those written in currently available computer languages. It is believed that the writing, testing, and modification of codes for solving large-scale linear programs will be a less formidable undertaking once MPL becomes available. It is hoped that by the Fall of 1968, work on a compiler for MPL will be well underway.
Mathematical Programming
Large-Scale Linear Programs