MPL

MATHEMATICAL PROGRAMMING LANGUAGE

Specification Manual for Committee Review

Prepared By:

Stanley Eisenstat
Thomas Magnanti
Steve Maier
Michael McGrath
Vincent Nicholson
Christiane Riedl

With Foreword By George B. Dantzig

STAN-CS-70-187
NOVEMBER 1970

COMPUTER SCIENCE DEPARTMENT
School of Humanities and Sciences
STANFORD UNIVERSITY

MPL

MATHEMAT I CAL PROGRAMM I NG LANGUAGE

Specification Manual for Committee Review

Prepared By:

Stanley Eisenstat

Thomas Magnant i

Steve Maier

Michael McGrath

Vincent Nicholson

Christiane Riedl

With Foreword By George B. Dantzig

Computer Science Department
Stanford University
Stanford, California

Research and reproduction of this report was supported by the National Science Foundation, Grant GJ 320.

Reproduction in whole or in part is permitted for any purpose of the United States Government. This document has been approved for public release and sale; its distribution is unlimited.

CONTENTS

ATTACHMENTS

FOREWORD

Abstract

Mathematical Programming Language (MPL) is intended as a highly readable, user oriented, programming tool for use in the writing and testing of mathematical algorithms, in particular experimental algorithms for solving large-scale linear programs. It combines the simplicity of standard mathematical notation with the power of complex data structures. Variables may be implicitly introduced into a program by their use in the statement in which they first appear. No formal defining statement is necessary. Statements of the "let" and "where" type are part of the language. Included within the allowable data structures of MPL are matrices, partitioned matrices, and multidimensional arrays. Ordered sets are included as vectors with their constructs closely paralleling those found in set theory. Allocation of storage is dynamic, thereby eliminating the need for a data manipulating subset of the language, as is characteristic of most high level scientific programming languages.

This report summarizes the progress that has been made to date in developing MPL. It contains a specification manual, examples of the application of the language, and the future directions and goals of the project.

A version of MPL, called MPL/70, has been implemented using PL/1 as a translator. This will be reported separately. Until fully implemented, MPL is expected to serve primarily as a highly readable communication language for mathematical algorithms.

Acknowl edgements

Professor David Gries, before he took a new position, laid out a substantial part of the general framework of the language and helped guide its detailed specification. Miss Christiane Riedl, his assistant and also mathematician at SLAC took on an active role after Gries left for Cornell. Important suggestions have been made by Dr. C. Witzgall and Dr. R. Bayer of Boeing Scientific Laboratories, Mr. Paul Davis of Union Carbide, and more recently Professor R. Floyd of Stanford. Professors Alan Manne and Richard Cottle are currently not listed as principal investigators (as earlier) because the coming phase will be concentrating on developing an efficient compi ler. The work group S. Eisenstat, T. Magnanti, S. Maier, M. McGrath, V. Nicholson, graduate students in Operations Research and Computer Science are expected to continue to contribute to the development.

THE NEED FOR MPL

The purpose of MPL (Mathematical Programming Language) is to provide a language for writing mathematical algorithms, expecially mathematical programming algorithms, that will be easier to write, to read, and to modify than those written in currently available languages (e.g. FORTRAN, ALGOL, PL/1, APL).

The need for a highly readable mathematically based computer language has been apparent for some time. Generally speaking, standard mathematical notation in a suitable algorithmic structure appears best for this purpose. The reason is that most researchers are famil ier with the "language" of mathematics having spent years going to school and taking many courses on this subject. For the mathematical programming application, the availability of such a tool is deemed essential.

Mathematical programming codes tend to be complex. (Some commercial codes have over a hundred thousand instructions.) They are developed by persons formally trained in mathematics using, for the most part, standard matrix and set notations. Recently, research has been directed toward structured large-scale systems. These systems have great practical potential especially for planning the growth of developing nations, the national economy, or industry.

To date many methods have been proposed for solving large-scale systems, but few have been experimentally tested and compared because of the high cost and the long time it takes to program them, and because it is difficult to debug and to modify them quickly after they are written. It is believed that highly readable programs would greatly facilitate experimentation with these proposed methods and would speed up the time when they can be used for finding optimal growth patterns of developing nations and industries. Moreover, experimentation is a valuable way to develop ones Intuition and test conjectures prior to developing theoretical proofs.

GENERAL FEATURES OF MPL

Research on MPL to date has been directed towards developing a highly readable language adhering as closely as possible to standard mathematical notation. Considerable attention has been given to keeping the definition structure of MPL as general as possible.

Matrix notation is required for the mathematical programming applications and this has been given special emphasis in MPL including partitioned matrices and matrices with special structure.

Set notation is universally used in mathematical proofs. However in statements of algorithms, as found in theoretical papers, one finds what appears to be set notation, but which turns out to be, on closer examination, an ordered set concept i.e. there is an assumed underlying ordering of the elements of a set. A convenient set-like notation is part of MPL. Typically it is used with the such-that construct which allows one to restrict or extend the definition of a set through logical expressions.

Other important features of mathematical notation are the "let" and "where" concepts. As commonly used, they serve as either a symbol substituter (macro) or as a short subroutine whose parameters are evaluated and the results substituted for the symbol. LET and WHERE constructs are also part of MPL.

Generally speaking, the literature of mathematics has been devoted to proofs of theorems. Algorithms as such, when they do appear, are often part of a constuctive proof and have an ad-hoc. organizational structure. MPL has adopted instead the formal block stucture of ALGOL with minor variatlons. Alternatives are provided for those who prefer not to see the words BEGIN and END used as -punctuation marks for blocks throughout a program. The user can optionally use less obtrusive special bracket symbols to conveniently group several statements forming a block or to group statements which follow and are subject to IF and FOR clauses. It is also possible in MPL to conveniently identify by labels parentheses pairs, complex statements and algebraic expressions and thereby greatly increase readabi 1 i ty.

In mathematics it is often desirable to change the meaning of symbols (e.g. variable names). In computer - languages a formal structure for **declaring" (defining) symbols is used and also for stating the "scope" (the set of instructions) where these definitions are to be applied. Fdr example, In ALGOL names of variables defined within a block cannot be used outside the block without redefinltton. In MPL, definition of a symbol can be made anywhere inside the block up to its first appearance in a statement; moreover, it can also be implicitly defined by the statement itself. Implicit definition is an important feature of MPL.

Provision is made for conveniently specifying the scope of a variable if it extends outside the block. Finally, it is possible to release the storage space assigned for the values of a symbol when no longer needed.

In defining a language it is natural to worry about whether or not it is possible to reasonably implement it. For example, the present form of MPL uses linear character strings for exponents, superscripts or summations in place of two dimensional notation like:

$$\sum_{j=1}^{m} A_{ij}B_{jk}$$

Thus, a_i is written a(i). However, one of the members of our task force group (V. Nicholson) has recently completed a Ph.D. dissertation on this subject and we plan to incorporate features of his already implemented two dimensional notation into MPL.

Except for special functions like sin(X), mathematicians avoid the use of multiple character variable names. The reason for this historically appears to be two-fold: First, it is easier to visualize algebraic manipulation of symbols when they appear as single characters. Second, it avoids possible confusion with implicit multiplication e.g. sin(x) meaning s·i·n·(x). However, by requiring in MPL the explicit use of the multiplication symbol, multiple character names are allowed

as in most computer languages.

Overal 1 status:

A draft of the MPL specifications in Backus Naur Form has recent 1 y been prepared under the general guidance of George B. Dantzig by our work group with Miss Riedl serving as general coordinator. This draft is now being readied for general review by a committee probably consisting of Rudolf Bayer, Paul Davis, David Gries, Robert Floyd, Donald Knuth, and Christoph Witzgall.

A preliminary test version of MPL, referred to as "MPL-McGra th," was at the suggestion of Paul Davis implemented in 1969 by Michael McGrath using PL/1 as a translator into PL/1 instructions. This version included those features of MPL that were easiest to translate into available PL/1 constructions.

DETAILED SPECIFICATION REVIEW

The first goal of the project was to specify the language in implementable form. The language outlined in the preliminary proposal to NSF as of May 1968 was systematically developed; the syntax was more closely aligned with standard mathematical notation and kept as general as possible. Many of the earlier constructs were extended and improved, for example:

- The vector construct was extended to include set notation in the form of ordered sets with logical qualifiers.
- More complex data structures were introduced, including multidimensional arrays, partitioned matrices and reference arrays.
- The domain of numeric constants was made the extended real numbers $\{-\infty, +\infty\}$
- In response to user requests, blocks were introduced as a primary means of defining scope of variables.
- The principle of dynamic allocation of storage was adopted for all non-scalar quantities.
- Both dynamic and static symbol substitution were introduced into the language.
- Subscripting was generalized to include subscripting of expressions.

- Function Variables which allow a general function name to be replaced by a specific name were introduced.
- Parameter passing for procedures was greatly extended by developing several different types of procedures. In particular, a function procedure was introduced which acts exactly as a function in mathematics, (i.e. without any side effects),

- This phase of the project is nearing completion. Concurrent with the submission of this 'proposal, the language specification will be given to the review committee. During the fall quarter the language will be used as a teaching tool to obtain feedback from potential users. By the end of calendar year 1970 it is hoped that the specifications can be frozen, so that implementation and use as a communication language can begin in earnest.

The second goal of the project was to implement these specifications. This involved development of a PL/1 translator and made possible evaluation of the language by Operations Research graduate students and researchers from bdth the academic and industrial communities.

Originally it was hoped that the compiler-compiler system under development by David Gries could be used to implement MPL on any installation on which his system was made available. Unfortunately, the compiler-compiler was never completed. Therefore, in order to produce an

why PL/1?

environment free compiler, a translator written in PL/1 was developed. It was felt that this would provide the widest possible circulation for MPL, since any installation with a PL/1 compiler could then be used.

The current version of the MPL/PL1 translator encompasses many of the unique constructs available within MPL. The translator was successfully used in a large scale systems optimization seminar with enthusiastic student response. Much valuable information was obtained from this exchange, and it is hoped that this practice can be continue& Of particular note, is that many students found the language easier to use and less tricky than either FORTRAN or ALGOL.

The MPL language was presented to the industrial community through the Stanford "Computer Forum" by M. McGrath in 1969 and C. Riedl in 1970; to the academic community through lectures by the proposer; and to the professional community by R. Bayer and C. Witzgall in talks on their matrix calculus which is expected to play a role in the generation and manipulation of special matrix structures. Some work was also done on using MPL as a tool in developing new algorithms and in presenting some of the existing algorithms in the field of Operations Research. It is hoped that this will become one area of future concentration in the further development of the MPL language. This has particular importance in gaining wider acceptance for the language.

RESEARCH PROGRAM

MPL as a Communication and Programming Language

To date the primary objective of the MPL project has been the formal language definition. The result of this effort is the Language Specification Manual written in Backus Naur Form which should function as a basis for an implementation. Since this manual is intended for computer specialists, it is not very suitable for an applied mathematician not trained in computer science. Accordingly, a next step for the project (and its proposed continuation) is the development of an MPL user's manual. This document would serve in two capacities:

(i) by giving an introduction to MPL for a wide spectrum of possible users, and

(ii) by expanding and interpreting the more involved features of the language found in the MPL specification manual,

To accommodate both of these objectives, the user's manual would endeavor to present MPL in a simplified form and at a level in which most of its constructs are explained. In this manner, the reader at the beginning or intermediate level, knowing only a subset of the language, would nevertheless be able to write MPL programs compatible with the full language.

With a user's manual available, the project would proceed into a testing and evaluation phase. An important contribution to this phase would-be feedback from potential users. From this feedback we would be able to ascertain what modifications, if any, are required to give us the "best" language for the user. It is probable that MPL will be equally useful for statistical and numerical analysis applications, particularly in conjunction with special sub-routines useful in these fields. Though we would encourage investigation of MPL's use in other areas, we propose to concentrate primarily upon applications to Mathematical Programming.

Testing MPL as a language for Mathematical Programming would proceed along two fronts. First, standard algorithms, such as Generalized Upper Bounding (see attached), would be programmed using MPL. This would allow us not only to evaluate MPL as a programming tool but also to assemble a library of algorithms for use in further research. Second, MPL would be used to write and test new algorithms, consequently, evaluating its potential as a research tool. We believe that the language could have a great impact in this area - especially in academic research where the time and expense in programming for large scale systems has been prohibitive in other languages.

As a user's tool, MPL has been developed to parallel much of standard mathematical notation. Thus most

algorithms written in mathematics could almost as easily be written and read in MPL. This aspect of the language makes it attractive as a standard. communication language for algorithms. As one further phase of this proposal, we hope to explore this in greater depth. In particular, we would investigate whether it would be plausible to use MPL as a standard vehicle for presenting algorithms in journals especially for the newly proposed Mathematical Programming Journal. Not only would this have the beneficial effect of standardization, but it would also mean that published algorithms could be easily tested or implemented via MPL,

Some of the objectives outlined above can be partially met with the current version of the MPL translator. in order to fully test the language and implement it as a user's tool, however, the translator will have to be expanded or a compiler written. An investigation of these possibilities constitutes the next major task of the continuing project.

Implementation Considerations

A complete, "machine-independent" implementation seems essential in gaining broad acceptance of MPL as a mathematical programming language. Such an implementation could take two directions:

(i) Extending the current translator to encompass those

MPL concepts not presently handled (e.g. subscripting as an operator, partitioned data structures, concatenation, and set generators).

(ii) Writing a full-scale compiler into some ideal machine language (e.g. three address code or reverse Polish).

The translator would be less work but the more efficient code produced by a compiler would make the solution of large scale problems more practical. However, of -equal, if not greater, importance is a "How to Implement** manua 1, a compendium of suggestions on implementing some of the more powerful MPL constructs as well as techniques for handling large scale data structures and codes involving many thousands of instructions on a computer.

For the most part, the techniques would be machine independent, i.e., the method of implementation outlined in the manual should be of help in implementing any large-scale mathematical programming system.

Part of the manual would be concerned with the analysis of an MPL program. Items included would be parsing techniques, symbol table organization, a precedence grammar if possible, suggestions for the internal representation of the program after analysis, and an outline of code emitted for advanced features of MPL (e.g. function variables, indexing sets, dynamic LET statements).

Runtime organization which is essentially MPL independent would require a study of data structures

necessary for large scale systems, dope vectors, algorithms for handling the non-first-in-first-out data structures of MPL.

If an easily modifiable translator were written, experiments could be made with different runtime data structures, data handling algorithms, and computational algorithms (such as matrix expression evaluation).

MPL

LANGUAGE SPECIFICATION

TABLE OF CONTENTS

SECTION 0. NOTATION

In the following language specification, we use a modification of BACKUS-NAUR FORM (BNF) to describe the syntactic structure of MPL.

A syntactic rule or production consists of a LEFT PART (a syntactic class name), followed by a ::= (read "is defined as"), followed by a right part (a string of symbols-which define the left part). Syntactic class names are enclosed in angular brackets <,>; MPL symbols stand by themselves.

Notes:

(I) If a syntactic class is defined to be one of several strings of symbols, the alternates are separated by a | (read "or").

Example: <character> ::= <letter> | <digit> j <special character> reads A character is defined to be a letter, a digit, or a special character.

(II) If part of the right side of a production may be omitted, it is enclosed in square brackets {denoted by $,$ in this document).

Example: <number> ::= <number base> $<exponent>$ is equivalent to <number> ::= <number base> | <number base><exponent>

(III) A list of one or more symbols all belonging to the same syntactic class x is denoted by <<X> LIST>.

Example: <digit string> ::= <<digit> LIST> stands for <digit string> ::= <digit> | <digit string><digit>

If the symbols in the list must be separated by a delimiter, then the delimiter directly precedes the word LIST.

Example: <variable LIST> ::= <<variable>, LIST> is equivalent to <variable LIST> ::= <variable> | <variable LIST>, <variable>

(IV) The syntactic class <empty> represents the null string of symbols.

(V) The right side of a production may be partly described by a comment enclosed in quotes. The comment gives semantic restrictions on the right part.

Example: <VA expression> ::= <Vector valued" arithmetic expression>

(VI) Certain delimiters and reserved words may be substituted for other delimiters or reserved words. If y may be substituted for x, this is indicated by x <-- y at the first occurrence of x in the language definition. If x <-- y and y <-- x, this is indicated by x <--> y.

Example: := <-- = means that = is an alternate assignment symbol
IN <--> means that IN and are interchangeable

SECTION 1. BASIC CONCEPTS

1.1. The Character Set of MPL

The set of characters available in MPL will depend on the particular implementation. As this language specification is independent of any implementation, we here define a basic character set which will be used throughout this manual and suggest possible extensions to it.

<character> ::= <letter> | <digit> | <special character>

<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|
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> ::= +|-|*|/|**|<bar>|$|<|<=|=|¬=|>=|>|:=|(|)|00|¬|
.|,|;|:|"|<<|>>|_|'|<blank>|€|¢|
<opening rector bracket>|
<closing vector bracket>

<bar> ::= |

<blank> ::= "one blank space"

<opening vector bracket> ::= <|

<closing vector bracket> ::= |>

Notes:

(I) An implementation may allow the use of any other symbols (e.g., the Russian or Greek alphabet) in addition to the letters defined above.

(II) (,) <--> , ; 00 <--> INFINITY

1.2. Basic Elements of The Language

1.2.1. Identifiers and reserve words

<identifier> ::= <letter> $<<idchar> LIST>$ | <identifier>*

<idchar> ::= <letter> | <digit> | _

Identifiers bare no Inherent meaning, but are used to represent , simple variables (see 3.1.2), expressions (see 4.8), labels (see 4.4.1), and procedures (see 5.1). The scope of identifiers is controlled by the block structure of the 'program (see 4.1).

Identifiers must start with a letter, followed by any combination of letters, digits, and underscores: they ray end in one or more single , primes (apostrophes). Identifiers may not contain blanks. There is no restriction on the length of identifiers.

The following reserved words have special meaning and aay not be used as identifiers:

AND	ELSE	LET	RETURN
ANSWER	**EMPTY**	LOGICAL	ROW
ARITHMETIC	END	**LOWER**	SCALAR
ARRAY	EXECUTE	**MATRIX**	SPARSE
BEGIN	EXTERNAL	**MULT**	THEN
BLOCK	FALSE	**NAME**	TRIANGULAR
BY	FOR	NOT	TRUE
CHARACTER	FUNCTION	OR	UNDEFINED
COLUMN	GIVEN	**OTHERWISE**	UPPER
DEPENDENT	GO TO	PARTITION	VALUE
DEFINE	IF	PROCEDURE	VECTOR
DIAGONAL	IN	**PROGRAM**	**WHERE**
DO	INDEPENDENT	RECTANGULAR	WITH
DOMAIN	**INLINE**	RELEASE	
DIMENSIONAL	IS	RESULT	

Examples: **C5A, BASIC_VARIABLES, X, X', X''**

1.2.2. Digit strings

<digit string> **::=** <<digit> LIST>

Digit strings are used to form arithmetic constants (see 3.1.1.1) and (enclosed in patentheses) to represent labels (see **4.4.1).**

1.2.3. Deliaiters

The following special characters are used as operators, brackets, and separators:

<del imiter> ::= +|-|*|/||<bar>|#|<|<=|=|¬=|>=|>|:=|(|)|¬|**
,|;|:|"|<<|>>|<blank>|∈|∉|
<opening vector **bracket>|<closing** vector bracket>

1.2.4. Character strings

<character string> **::=** <<character> **LIST>**

Character 'strings are used in character constants (see **3.1.1.3).**

1.2.5. Blanks

A blank space is required after **an** identifier or **reserved** word which is **followed** by an identifier, reserved word, or number. Blanks are not permitted within identifers. Blank spaces are ignored, except within character strings.

1.2.6. Comments

Any sequence of characters (excluding a quote **(")**) enclosed in quotes **(",")** is treated as a blank space except within character strings. Such **comments** ray be used to insert remarks into the program.

1.3. The Structure Of A Program

1.3.1. Insertionsand insert. statements

A program submitted by a user consists of program text modified by insertions which yield the actual MPL program. The insertions and insert statements (which specify where insertions are to be made) are editing features and are therefore-; strictly speaking, not part of the language.

An insertion

```
$INSERTION <identifier>
"arbitrary text"
$END <"same" identifier>
```

will be deleted from the program text and the "arbitrary text" will replace the insert statement.

```
$INSERT <ident if ier>
```

wherever it appears in the program text. After each replacement, the resulting program text is searched for further insertions.

Example:

```
        PROGRAM SHORTIE;
        $INSERTION ALPHA
        ANSWER RESULT;  GO TO
        $END ALPHA
        DEFINE RESULT := 0;
LOOP:   IF RESULT>20 THEN BEGIN
        $INSERT ALPHA  EXIT END;
        RESULT := RESULT+1;
        $INSERT ALPHA LOOP;
EXIT:   END
```

will be transformed into

```
        PROGRAM SHORTIE;
        DEFINE RESULT := 0;
LOOP:   IF RESULT>20 THEN BEGIN
        ANSWER RESULT;  GO TO EXIT END;
        RESULT := RESULT+1;
        ANSWER RESULT;  GO TO LOOP;
EXIT:   END
```

1.3.2. Programs

```
<program> ::= PROGRAM $<"program" label>;$
              <<program unit>; LIST>
              END $<"same" label>$

<program unit> ::= <statement> | <procedure definition>
```

A program consists of a sequence of statements and procedure definitions. A program acts as a block (see 4.1) and the program label may be used in a defining statement to delimit scope (see 4.2.1). A transfer of control to the program label causes reexecution of the program.

SECTION 2. ATTRIBUTES and VALUES

The quantities on which the program operates are each characterized by a set of attributes and a (set of) **value(s).** A scalar (SCALAR) quantity represents a sinyle value; a non-scalar (VECTOR, MATRIX, ARRAY) quantity represents a set of values, a single value corresponding to each element of an underlying ordered **domain.** The range of possible values is specified by the type attribute, the underlying domain by the **domain** attribute,

Attributes are associated with variable names in defining statements (see 4.2.1). Values are assigned to variables (see 3.1.2) in defining statements and assignment statements (see 4.3). The attributes and value(s) associated with expressions {see 3) are determine? by the rules for operators (see 3.2).

<attribute> : := <type attribute> | **<dimensionality** attribute> |
<domain attribute) | <shape attribute>

<type attribute> ::= ARITHMETIC | LOGICAL | CHARACTER |
<reference variable attribute> |
<procedure variable attribute>

<reference variable attribute> ::= (<<attribute> LIST) |
$<type attribute>$ (PARTITION $<(<<span>,LIST>) BY LIST>$)

<procedure variable **attribute>** ::= (<procedure **head>)**

<dimensionality attribute> ::= SCALAR | VECTOR | <matrix attribute> |
$<digit string>-DIMENSIONAL$ ARRAY

<matrix attribute> : := MATRIX | ROW VECTOR | COLUMN VECTOR

<domain attribute> ::= **$WITH DOMAIN$** < BY LIST>

 ::= EMPTY | **<SA** expression> | **<VA** expression> |
DOMAIN (<expression>) | **<"(VECTOR) VECTOR"** expression>

<shape **attribute>** ::= DIAGONAL | RECTANGULAR | UPPER TRIANGULAR |
LOWER TRIANGULAR | SPARSE

2.1 The Type Attribute

The type attribute specifies the range of possible values. The **value** UNDEFINED used in initialization is of universal type.

2.1.1. Simple types (ARITHMETIC, LOGICAL, CHARACTER)

TYPE	VALUE CAN BE
ARITHMETIC	numeric, +00, -00
LOGICAL	**TRUE,** FALSE (truth values)
CHARACTER	any character provided by an implementation

2.1.2. Reference types

The reference variable attribute specifies the range of possible values as the collection of all quantities whose attributes are consistent with the attributes specified (or defaulted (see **4.2.1**)) in the attribute list.

2.1.3. Procedure types

The procedure variable attribute specifies the range of possible values as the collection of all **procedure** definitions (see 5.1) with procedure head compatible with the specified procedure head. Two procedure heads are compatible if their formal input (return) parameter(s) agree in number and specified attributes.

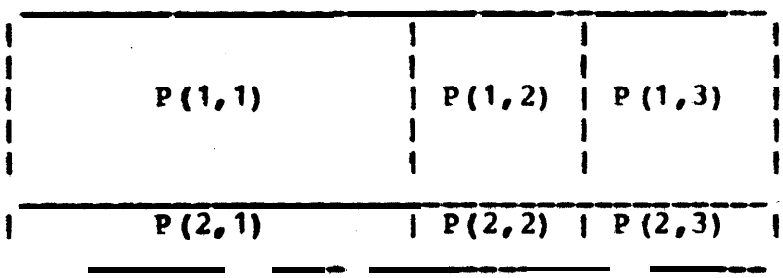
-2.2. The **Dimensionality** Attribute

The dimensionality attribute specifies the dimension (number of component domains) of the associated domain for non-scalar quantities.

An ARRAY may have any number of 'dimensions. A VECTOR is a **1-DIMENSIONAL** ARRAY; a MATRIX IS A **2-DIMENSIONAL** ARRAY. A **ROW** (COLUMN) VECTOR is treated as a special kind of **MATRIX.** For completeness, ve define a SCALAR as a **0-DIMENSIONAL** ARRAY.

The components of a REFERENCE **ARRAY** are themselves arrays. All components of a reference array must have the same type and dinensionality but they can differ in size. In order to access the scalar elements of the components of a reference array **two** sets of subscripts must be used **(see 3.1.2.1).**

A **PARTITIONED MATRIX** is a two-dimensional reference array whose components are matrices. All **components** in a row of a partitioned matrix must have the same number of rows and all components in a column must have the same number of columns, so that a 2 by 3 partitioned matrix can be represented by a diagram:



2.3. The Domain Attribute

The domain attribute specifies the associated domain for non-scalar quantities, Domains are restricted to Cartesian products of component

domains. A component domain may he any finite (possibly empty), strictly increasing sequence of integers. A component domain of the form <|1. ⊠⊠ ,n|> is said to be canonical. A domain is canonical if each component domain is canonical.

The span EMPTY specifies the empty component domain. The span <SA expression> specifies the component domain <|1,. ..,<SA expression>|>. The span <VA expression> specifies the component domain <VA expression>. The span DOMAIN (<expression>) specifies a sequence of component domains, namely the component domains associated with the expression. The span <"(VECTOR) VECTOR" expression> specifies a sequence of component Romains, namely the vector-valued components of the <"(VECTOR) VECTOR" expression>.

2.4. The Shape Attribute

The shape attribute is used to economize on the space required to store large data structures and to produce more efficient code to handle then.-

SECTION 3. EXPRESSIONS

An expression is a rule for computing a (set of) value(s) by executing the indicated operations on the values represented by the operands of the expression. In this section, we shall describe the basic operands of expressions, the allowable operations on them, and finally the syntax and manner of evaluation of expressions.

We shall use the following abbreviations to denote special classes of expressions:

A	for arithmetic
L	for logical
C	for character
R	for reference type
P	for procedure type
S	for "scalar valued"
V	for "vector valued"

Thus, the symbol <SA expression> used in the preceding section abbreviates <"scalar valued" arithmetic expression>.

3.1. Operands

3.1.1. Constants

3.1.1.1. Arithmetic constants

<number> ::= <number base> $<exponent>$

<number base> ::= <digit string> | <digit string>. | .<digit string> | <digit string>.<digit string>

<exponent> ::= E Wadding operator>$ <digit string>

<adding operator> ::= +|-

Examples: 1970, 3.1415926536, 6.0247E+23, 6.6254E-27

3.1.1.2. Logical constants

<logical value> ::= TRUE | FALSE

3.1.1.3. Character constants

<character constant> ::= <opening character quote>
<character string>
<closing character quote>

<opening character quote> ::= <<

<closing character quote> ::= >>

Example: <<NOW IS THE TIME FOR ALL PARTIES TO COME TO THE AID OF MAN>>

A character constant is a CHARACTER VECTOR (with canonical domain) whose component values are the characters in the string. The character string may not contain the sequences << or >>. Blank spaces are valid characters.

3.1.2. Variables

<variable> ::= <simple variable> | <subscripted variable>

<simple variable> ::= <identifier> | <variable synonym>

<subscripted variable> ::= <variable> (<<range>, LIST>)

<range> ::= <SA expression> | <VA expression> | *

Examples: A13_B, subvector (*), X (3-A),
SUBMAT (*,<|3,4,5|>),ARRAY_EL (6,B/5,4**X,8),
SUBARRAY (*,*,*,3), REFARR (3) (*,5,6)
PART-MATRIX (5,6) (*,<|2,8,9,|>)

Variables represent storage locations where values are stored which may change during execution of the program. At any given time the value (or the ordered set of values) associated with the variable is the last value(s) assigned to the variable.

Note that a variable may be a scalar, vector, matrix, or array.

3.1.3. Vector generators

<vector generator> ::= <N-tuplet> | <index range> | <set generator>

<N-tuplet> ::= <opening vector bracket>
<<expression>, LIST>
<closing vector bracket>

<index range> ::= <opening vector bracket>
<SA expression>, $<SA expression>,$...,<SA expression>
<closing vector bracket>

<set generator> ::= <opening vector bracket>
<expression><FOR phrase>
<closing vector bracket.>

3.1.3.1. N-tuplets

The expressions listed must be scalars or vectors all of the same type. The n-tuplet is a vector (with canonical domain) of that type whose set of values is the concatenation of the values of the scalars in the list and the sets of values of the vectors in the list.

Examples:
<|TRUE, A OR B, ¬D, X<Y|>
<|<<JOE>>,<<KIM>>,<<AGE20>>|>

<|1,a-b,-3.5,<|.67E-3,e,12.5|>|> is the same as
<|<|1,a-b,-3.5|>,.67E-3 ,e,12.5|> which is the same as
<| 1,a-b,-3.5,.67E-3,e,12.5|>

3.1.3.2. Index ranges

An index 'range is an arithmetic vector (with canonical domain). The set of values generated by the index range

<| VFIRST, VSECOND, ..., VLAST |>

is the sequence

VFIRST, VFIRST+ 1*VSTEP, VFIRST+2*VSTEP, ... , VFIRST+N*VSTEP

where VSTEP = VSECOND-VFIRST (if VSECOND is omitted, then VSTEP is taken to be one) and

N = sup | n | n>=0 and (VFIRST+n*VSTEP-VLAST)*VSTEP <= 0 |.

If (VFIRST-VLAST) *VSTEP > 0, then the set of values generated is empty.

Examples: <|71,...,75|> = <|71,72,73,74,75|>
<|0.1.0.3,...,0.8|> = <|0.1,0.3,0.5,0.7|>
<|7,5,...,-2|> = <|7,5,3,1,-1|>
<| 3,...,0|> is EMPTY.

3.1.3.3. Set generators

A set generator yields a vector having as many components as are determined by the <FOR phrase> (see 4.7). The values of the components are given by the value of the <SA expression> as modified by the successive values of the controlled variable of the <FOR phrase>. The identifier denoting this controlled variable is local to the set generator.

Example: <| i**2 FOR i=<|1,...,6|> : i ¬= 4 |>
gives <| 1,4,9,25,36 |> with domain <|1,2,3,5,6|>

3.1.4. Procedure calls

A procedure call (see 5.2) may be used as an operand in an expression provided that the procedure has precisely one formal return parameter. The attributes and value(s) are taken from the actual return parameter.

3.1.5. Relations

<relation> ::= <A expression><relational operator><& expression> |
<C expression><equality operator><C expression> |
<S expression> IN <V expression> |
<S expression> NOT IN <V expression> |
<V expression> IS EMPTY |
<variable> IS UNDEFINED

<relational operator> ::= <equality operator> | < | <= | >= | >

<equality operator> ::= ¬= | =

*) IN, NOT IN <-->∈ ,∉

Examples: 3.5< A+F(B+C), <<HUGO>> ¬= <<HUGO >>,
2 TN <|8,6,3,-1,2|>, X IS UNDEFINED,
<| 10,...,A|> IS EMPTY

Relations are operands of logical expressions. Their values are determined as follows:

(I) The two operands OP1,OP2 of an equality operator may differ in kind and size.
OP1=OP2 is TRUE if OP1 and OP2 have the same dimensionality and size and every pair of corresponding components of them is equal, and FALSE otherwise.
OP1¬=OP2 is TRUE if and only if OP1=OP2 is false.

Note that this implies for character tests that in the example above, the value of <<HUGO>> ¬= <<HUGO >> is FALSE because of the unequal length of the character strings.

(II) If the relational operator is not an equality operator, then the two operands must have the same dimensionality and size. The result is TRUE if the relation holds for every pair of corresponding component-s and FALSE otherwise.

(III) <S expression> TN <V expression> is TRUE if the value of the S expression is the value of at least one of the components of the V expression, otherwise it is FALSE.
Note that the result is FALSE if the V expression is EMPTY.

('IV) <V expression> IS EMPTY is TRUE if the result of the V expression is a vector with no components (i.e, it has the domain attribute EMPTY (see 2.3)) and FALSE otherwise.

(V) <variable> IS UNDEFINED is TRUE if and only if not all of the components of the variable have been assigned a value yet,

3.1.6. Synonyms

<synonym> ::= <identifier> $(<<argument>, LIST>)$

<argument> ::= <expression>

<variable synonym> ::= <synonym> "defined by a let statement to stand for a variable"

<expression synonym> ::= <synonym> "defined by a let statement to stand for an expression which is not a variable"

A synonym must be defined by a symbol suhstitutar in a LET statement or WHERE phrase (see 4.8) before it can be used, and is not defined outside the block containing the LET statement (see also 4.1).

The synonym is replaced by the corresponding expression (enclosed in parentheses). Variable synonyms," i.e., synonyms that stand for a variable may also be used on the left side of an assignment statement (see 4.3). In this case they are not enclosed in parentheses when the substitution is made.

If the synonym depends on arguments, they will be substituted (enclosed in parentheses) for the dummies occuring in the expression. The modified expression will then replace the synonym.

The result of this replacement must be an allowable operand for the expression containing the synonym.

Examples : .

```
LET A(j) := 2*j+loop(j) ; C = D - A(i+3) ;
is the same as C = D - (2* (i+3) +loop(i+3)) ;

LET S := <| f(j) FOR j IN T : j ¬= 2|> ;
C = <|2*i+3 FOR i IN S|> ;
is the same as
C = <| 2*i+3 FOR i IN <| f(j) FOR j IN T : j¬=2 |>|> ;
```

3.2. Operators

3.2.1. Arithmetic operators

<arithmetic operator> ::= +|-|*|/|**

The basic arithmetic operators (+,-,*,/,**) are defined for ARITHMETIC SCALAR operands and have the conventional meaning (addition, subtract ion, multiplication, division, and exponentiation). The arithmetic operators defined for non-scalar arithmetic operands may be described in terms of these basic arithmetic operators.

3.2.1.1. Unary operators (+ and -)

The unary operators + and - are defined for all non-scalar arithmetic operands and are performed componentwise, leaving the dimensionality and domain of the operand unchanged.

3.2.1.2. Rinary operators

3.2.1.2.1. Addition (+) and Subtraction (-)

The binary operators + and - are defined for all pairs of non-scalar arithmetic operands with the same dimensionality and domain. The indicated operation is performed componentwise, leaving dimensionality and domain unchanged.

3.2.1.2.2. Scalar multiplication(*) and division(/)

One operand (the first for division) may be any arithmetic operand, whereas the other must be a scalar and in the case of division not zero.

The dimensionality and domain 04 the result are taken from the (dimensioned) operand and the value(s) obtained by multiplying {dividing) each of its elements by the scalar.

3.2.1.2.3. Exponentiation (**)

If OP1 and OP2 are scalars, then OP1**OP2 is defined if
(1) OP1 > 0
(2) OP1 = 0 and OP2 > 0
(3) OP1 < 0 and OP2 is an integer.

Tf OP1 is a square matrix, then OP1**OP2 is defined if OP2 is a positive integer N and denotes the result of N multiplications of OP1 by itself; OP1**0 denotes the identity matrix with the same number rows anti columns.

3.2.1.2.4. Inner product (*)

The inner product of two non-scalar operands is algebraically a generalization of matrix multiplication an? the vector inner product. Let

A p-DIMENSIONAL ARRAY M(1) BY ... BY M(p)
B q-DIMENSIONAL ARRAY N(1) BY ... BY N(q)

denote the two operands. Then the result C = A*B is defined if and only if M(p) = N(1):

C (p+q-2)-DIMENSIONAL ARRAY
M(1) BY ... RY M(p-1) BY N (2) BY ... BY N (q)

C(I(1),...,I(p-1),J(2),...,J(q))
= SUM (A(I(1),...,I(p-1),k)*B(k,J (2),...,J(q)) FOR k IN M(p))

for all I(i) IN M(i), i = 1,...,p-1
J(j) IN N(j), j = 2,...,q

In -particular, the inner product of two vectors (matrices) reduces to the vector inner product (matrix multiplication). The inner product of a matrix and a vector or a vector and a matrix is a vector.

3.2.2. Concatenation

<concatenating operator> ::= <horizontal concatenating operator> |
<vertical concatenating operator>

<horizontal concatenating operator> ::= |

<vertical concatenating operator> ::= &

Concatenation is defined for operands of all three types. Both operands of a concatenating operator must have the same type which will be the type of the result.

The operands of horizontal and vertical concatenation can be scalars (interpreted as 1 by 1 matrices) or matrices (including row and column vectors). The result is the matrix (with canonical domain) obtained by appending the elements of the second operand at the right side (or in the case of vertical concatenation at the bottom) of the first operand. The two operands of | must have the same number of rows, the two operands of ! must have the same number of columns.

Examples: Let A be the 2 by 3 matrix |1 3 4|
|6 -2 4|

B the 2 by 4 matrix |5 7 9 11|
|8 6 4 2|

C the row vector (0 1 0 1)

then A|B is the 2 by 7 matrix |1 3 4 5 7 9 11|
|6 -2 4 8 6 4 2|

B!C is the 3 by 4 matrix |5 7 9 11|
|8 6 4 2|
|0 1 0 1|

6|C|8 is the row vector (6 0 1 0 1 8)

3.2.3. Logical operators

The basic logical operators (¬, AND, OR) are defined for LOGICAL SCALAR operands and have the conventional meaning (negation, con junction, and disjunction). The logical operators defined for non-scalar logical operands may be described in terms of these basic logical operators.

3.2.3.1. Negation (¬)

The unary operator NOT is defined for all non-scalar logical operands and is performed componentvise, leaving the dimensionality and domain of the operand unchanged.

*) ¬ <--> NOT

3.2.3.2. Binary logical operators

<logical operator> ::= AND | OR | NULT

3.2.3.2.1. AND and OR

The binary operators AND and OR are defined for all pairs of non-scalar logical operands with the same dimensionality and domain. The indicated operation is performed componentwise, leaving the

dimensionality and domain unchanged.

Example:
(TRUE, A OR B) AND (FALSE,TRUE)
results TN (FALSE, A or B)

X2.3.2.2. **Logical** inner product **(MULT)**

The logical inner product of two non-scalar logical operands is defined analogously to the (arithmetic) inner product except that multiplication is replaced by **AND** and summation is replaced by OR.

Examples: Suppose A is |TRUE TRUE |
|FALSE FALSE|

and B **IS (FALSE,TRUE)**

then A **MULT** B is the vector **(TRUE,FALSE)**

and B **MULT** A is the vector **(FALSE,FALSE).**

3.2.4. Subscripting

Subscripting as an operator is defined for all non-scalar operands (a scalar operand with a non-scalar reference value is treated as a non-scalar operand with the **attributes** and set of values associated with the non-scalar reference value). The number of subscripts must agree with the diaensionality of the operand. A **<SA** expression> subscript specifies one element of the corresponding component domain. A <VA **EXPRESSION>** subscript specifies a subdomain of the corresponding component domain. A * subscript specifies the entire corresponding component domain. The type of the result is the type of the operand. The domain of the result is the Cartesian product of the domains of **the** vector subscripts and the component domains corresponding to * subscripts (if all subscripts are scalar, then the domain is &dimensional and the result is scalar). The (set of) value(s) associated with the result is the specified (subset of) component(s) of the operand.

3.3. Expressions

<expression> **::=** <arithmetic expression> j <logical expression> |
<character expression> | <reference expression> |
<procedure expression>

<arithmetic expression> **::= $<adding operator>$<A** operand> |
<A expression><arithmetic operator><A operand> |
<A **expression><concatenating operator><A** operand>

<A operand> **::=** <number> | <simple variable> | <procedure call> |
<vector generator> | <expression synonym> |
(<A expression>) | <A operand> (<<range>, **LIST>)**

<logical expression> ::= <L operand> |
<L expression><logical operator><L operand) |
<L expression><concatenating operator><L operand>

<L operand> ::= <logical value> | <simple variable> | <procedure call> |
<vector generator> | <relation> | ¬<L operand> |
<expression synonym> | (<L expression>) |
<L operand> (<<range>,LIST>)

<character expression> ::= <C operand> |
<C expression><concatenating operator><C operand>

<C operand> ::= <character constant> | <simple variable> |
<procedure call> | <vector generator> |
<expression synonym> | (<character expression>) |
CC operand> (<<range>, LIST>)

<reference expression> ::= Wadding operator>3 <R operand> |
<R expression><arithmetic operator><R operand> |
<R expression><logical operator><R operand> |
<R expression><concatenating operator><R operand>

<R operand> ::= <simple variable> | <procedure call> |
<vector generator> | <relation> | ¬<R operand> |
<expression synonym> | (<R operand>) |
<R operand> (<<range>, LIST>)

<procedure expression> ::= <P operand> |
<P expression><concatenating operator><P operand>

<P operand> ::= <procedure identifier> | <simple variable> |
<procedure call> | <'vector generator> |
<expression synonym> | (<P expression>) |
<P operand>(<<range>, LIST>)

Examples: (<|3,-7,5|>*<|6,8,4|>)**(FU(a,b)-3)
¬VAR IS UNDEFINED AND C>0 OR D
<| <<3 LITTLE BEARS>>,<<IN>>,<<THE>>,<<WOODS>> |>

Any simple variable, procedure call, vector generator, or expression synonym used as an

L operand		logical type
C operand		character type
A operand	must have	arithmetic or logical type
R operand		reference type
P operand		procedure type

If a logical quantity is used as an arithmetic operand, then TRUE is interpretted as 1 and FALSE as 0.

The sequence of operations within an expression is generally executed from left to right, but the order of evaluation is modified by the following precedence rules:

Each operator has an associated precedence number indicating its binding power. Operators with low precedence numbers take priority over operators with high precedence numbers:

Operator	Precedence
subscripting	first
#,!	second
**	third
*,/	fourth
+,-	fifth
<,<=,=,¬=,>=,>,IN,NOT IN,IS	sixth
MULT	seventh
¬	eighth
AND	ninth
OR	tenth

The expression between matching left and right parentheses is evaluated and the value(s) used in subsequent operations. Thus any order of execution of operations within an expression can be specified by appropriate parenthesizing.

SECTION 4. STATEMENTS

The units of operation in MPL are called statements. Statements are executed in sequence, as written, except when this sequence is modified by sequence control statements or conditional statements.

```
<statement> ::= <empty> | <label>: <statement> |
                <block> | <compound statement> |
                <variable control statement> |
                <assignment statement) |
                <sequence control statement) |
                <procedure statement> $<WHERE phrase>$ |
                <conditional statement> |
                <iteration statement> |
                <static let statement) |
                <dynamic let statement) |
                <input statement> $<WHERE phrase>$ |
                <output statement> $<WHERE phrase>$

<variable control statement> ::= <DEFINE statement> |
                                 <defining assignment statement> |
                                 <RELEASE statement>

<sequence control statement> ::= <GO TO statement> | <RETURN statement>
```

4.1. Blocks and the Scope of Identifiers

```
<block> ::= BLOCK $<label>;$
            <<program unit>; LIST>
            END $<"same" label>$

<compound statement> ::= BEGIN $<label>;$
                         <<program unit>; LIST>
                         END $<"same" label>$
```

*) BLOCK,END <--> [,] ; BEGIN,END <--> L,J

BLOCK <label>; <--> <label>: BLOCK

BEGIN <label>; <--> <label>: BEGIN

END <label> <--> ;<label> END

Blocks control the scope of identifiers by introducing new levels of nomenclature: an identifier declared in (local to) a block represents a unique entity within that block but does not represent that entity outside the block. An identifier declared in an embracing block is said to be global to the block. If an identifier is both local and global to a block, the global meaning can not be used within the block.

Identifiers may be declared explicitly by defining statements (see 4.2.1) or let statements (see 4.8); or implicitly by their appearance as labels (see 4.4.1) or procedure identifiers (see 5.1). Certain other syntactic units also implicitly delimit the scope of some or all of the identifiers declared within them in the same way as blocks:

(1) The conditioned (alternative) statement of a conditional statement acts as a block with respect to labels (see 4.4.1).

(2) An iteration statement acts as a block with respect to the control variable (see 4.7) and labels (see 4.4.1).

(3) A procedure definition acts as a block with respect to identifiers used as (part of) formal parameters in the procedure head (see 5.1).

(4) The identifier denoting the control variable in a set generator (see 3.1.3.3) or a serial actual parameter (see 5.2) is local to the set generator or serial actual parameter.

(5) Identifiers denoting dummy arguments in a symbol substituter are local to the symbol substituter (see 4.8).

(6) Identifiers denoting synonym names in a WHERE phrase are local to the statement qualified hy the where phrase.

If a block is labelled, then the block label may optionally follow the closing END and may be used in defining statements to delimit scope (see 4.2.1).

A compound statement is used to group together a sequence of statements and procedure definitions. If the compound statement is labelled, then that label may optionally follow the closing END.

4.2. Variable Control Statements

4.2.1. DEFINE statements

<DEFINE statement> ::= <DEFINE phrase><<defining phrase!>, LIST> |
<DEFINE phrase><defining phrase><qualifier>

<DEFINE phrase> ::= DEFINE $IN <"block" label>$

<defining phrase> ::= <<variable name>, LIST> <<attribute> LIST>

<variable name> ::= <identifier> | <reference variable name>

<reference variable name> ::= <variable name>
$<(<subset specification>)<blank> LIST>$
(<simple subscript>)

<subset specification> ::= <subspan> | <simple subscript>,<subspan> |
<subset specification>,<range>

<subspan> ::= <VA expression> | *

<simple subscript> ::= <<SA expression>, LIST>

Examples: DEFINE A LOGICAL 3 BY 5, B SCALAR
DEFINE C(1,6) 8 BY 8 DIAGONAL

DEFINE statements, defining assignment statements (see **4.2.2),** and **GIVEN** statements (see 4.9) are all defining statements in the sense that **they** delimit the scope of identifiers, assign attributes to identifiers and reference variable names, and allocate storage. Defining statements are executable; an identifier must be defined before it is referenced.

The scope of an identifier is either the innermost block containing the defining statement or the embracing block whose block label appears in the defining statement.

A variable name is either an identifier or a reference variable name (a single component of a reference variable; e.g., a submatrix of a partitioned matrix). The attributes to be associated with the variable name may be listed in any order in the defining phrase. The type and dinensionality attributes must be consistently defined throughout the scope; the domain and shape **may** change. Hissing attributes are defaulted in the following **manner:**

type: unchanged, if specified in another defining statement
ARITHMETIC, otherwise

dinensionality:
unchanged, if specified in another defining statement
SCALAR, if a **domain** is not specified
ARRAY (with appropriate number of **dimensions),otherwise**

domain: **EMPTY** BY **...** BY **EMPTY,** for arrays
unspecified, otherwise

shape: ROY **(COLUMN),** f o r **ROW (COLUMN) VECTORs**
RECTANGULAR, otherwise

When a variable name(s) is defined, all expressions in the defining phrase are evaluated, the variable **name(s)** is released (thus the associated value(s) are lost) (see **4.2.3),** and storage is allocated. Scalars are initialized **with value** UNDEFINED; arrays are initialized conponentuise with the value **UNDEFINED.**

When program control leaves a block, all identifiers defined local to the block are released (see 4.2.3) and **"un-defined"** (lose **definition).**

DEFINE STATEMENTS BAY BE **QUALIFIED** BY FOR phrases, IF phrases, and **WHERE** phrases (see **4.3).**

4.2.2. Defining assignment staterents

<defining assignment **statement> ::=**
<DEFINE phrase><simple defining assignment **statement>,** LIST> |
<DEFINE phrase><simple defining assignment **statement><qualifier>**

<simple defining assignment statement> **::=**
<variable name> **:=** <expression> **$<domain specification>$ |**
(<<left side element **name>,** LIST>) **:=** <procedure call>

<left side element. name> ::= <variable name> | _

Defining assignment statements serve as defining statements (see 4.2.1) as well as assignment statements (see 4.3) .

The first. form of the (simple) defining assignment statement causes evaluation of an expression. The variable name is defined with the attributes associated with the expression and assigned the value(s) of the expression. The domain of the expression may be redefined in a domain specification (see 4.3).

The second form of the (simple) defining assignment statement causes execution of a procedure. The left side element name(s) is defined with the attributes associated with the corresponding formal return parameter(s) and assigned the value(s) of the corresponding actual return parameter(s). An underscore appearing on the left side means that. the corresponding definition and assignment should be omitted.

Defining assignment statements may be qualified by FOR phrases, IF phrases, and WHERE phrases (see 4.3).

4.2.3. Release Statement

<RELEASE statement> ::= RELEASE <<variable name>, LIST>

Example: RELEASE MAT , A, B(3,6,7)

RELEASE statements serve to deallocate storage. Scalars are assigned the value UNDEFINED; arrays are redefined with domain EMPTY BY ... BY EMPTY (with the appropriate number of dimensions) and shape RECTANGULAR. The value(s) associated with the variable *name(s)* is lost.

No RELEASE statement is required before the terminating END of a block since the END acts as an implicit RELEASE statement for all identifiers defined local to the block.

4.3. Assignment Statements

<assignment statement> ::= <simple assignment statement> $<qualifier>$

<simple assignment statement> ::=
<variable> := <expression> $<domain specification>$ |
(<<left side element>, LIST>) := <procedure call>

<left side element.> ::= <variable> | _

<domain specification> ::= WITHDOMAIN < BY LIST>

<qualifier> ::= <WHERE phrase> |
<<qualifying phrase>, LIST> $,<WHERE phrase>$

<qualifying phrase> ::= CPOR phrase> | <IF phrase>

*) := <-- =

Examples:
```
(a,b,c) := f(x)
LOGIC (i) := a=e(i) OR i>20 FOR i IN <|10,...,30|>
PART(4,5) (*,4) := A*? & MAT
A := B*C - 3.37 * E + <|2,3,5|>
X := A<B OR F IN <|6,8,9|>
```

Assignment statements serve to assign the value(s) of the expression or procedure call on the right side of the assignment symbol to the variable(s) on the left side. The variable(s) and the corresponding value(s) must be assignment compatible:

(1) If the variable is character (logical), then the corresponding value must be character (logical). If the variable is arithmetic, then the corresponding value may be arithmetic or logical (see 2.1).

(2) The variable and the corresponding value must have the same dimensionality, except that a vector may be assigned to a ROY (COLUMN) VECTOR and vice versa.

(3) The variable and the corresponding value must have the same domain. If their domains differ, a defining assignment statement (see 4.2.2) must be used.

(4) The variable and corresponding value need not agree in shape.

The first form of the assignment statement causes the evaluation of an expression and assigns the value(s) to the variable on the left side. The domain of the expression may be redefined in a domain specification. The specified domain must be homeonorphic (corresponding component domains have equal numbers of elements) to the domain of the expression and compatible with the domain of the left side variable.

The second form of the assignment statement causes the execution of a procedure and assigns the value(s) of the actual return parameter(s) to the corresponding (in order from left to right) left side element(s). The number of left side elements must be the same as the number of formal return parameters in the procedure definition (see 5.1). All variable(s) on the left side must be assignment compatible with the corresponding formal return parameter(s) . An underscore appearing on the left side means that the corresponding assignment should be omitted.

Examples:
```
(Ret 1, _ , Ret 3, _ ) := FU(x-y)
(FU has 4 return parameters, but only the first and third of
 these are of interest).

(Objective value, Basic-variables, Optimal-x,Peasibility) :=
SIMPLEX (Matrix, Costs, RAS, BASIC_VARIABLES)
```

The effect of a simple assignment statement (DEFINE statement, defining assignment statement) modified by FOR phrases, IF phrases, and symbol substituters

<simple assignaent statement> <qualifying phrase "1"> ...
<qualifying phrase "n">, WHERE <<symbol substituter>, LIST>

can be described by the following sequence of MPL statements:

```
LET <<symbol substituter>, LIST>;
<qualifying phrase "n">,
  . . .
    <qualifying phrase "1">,
      <simple assignment statement);
```

except that identifiers denoting synonym names in the let statement are defined locally.

```
Examples: E := D* A    WHERE A = (B|C) & (C|D)
          X := X'+1    WHERE X = X'
          INCIDENCE (i,y) := TRUE IF i IN ARCS(y)
          A (P_ROW,y) := A(P_ROW,y) / A(P_ROW,P_COL)
                         FOR y IN COL_DIM (A)
          (Y (i) , Z (i) ) := FUNCTION(i) FOR i IN S
```

4.4. Sequence Control Statements

4.4.1. Labels and GO TO statements

<label> ::= <identifier> | (<digit string>)

<GO TO statement) ::= GO TO <label>

A GO TO Statement causes a transfer of control to the statement immediately following the label. Since labels are inherently local (see 4.1), NO CO TO Statement can lead into a block, a procedure definition, the conditioned (alternative) statement of a conditional statement, or an iteration statement.

4.4.2. RETURN statements

<RETURN statement) ::= RETURN

A RETURN statement causes a transfer of control from a procedure back to the main program or procedure calling that procedure. A RETURN statement may not occur outside a procedure definition.

NO RETURN statement is required before the terminating END of a procedure definition {see 5.1) since the END acts as an implicit RETURN statement.

4.5. Procedure Stateaents

<procedure statement> ::= EXECUTE <procedure call>

Example: EXECUTE GENERALIZED-UPPER-BOUND (m,n,l,A,G,b)

A procedure statement causes the execution of a procedure (that specified in the procedure call (see 5.2)) with no return parameters. A procedure with return parameters may be called in an assignment statement (see 4.3).

4.6. Conditional Statements

<conditional statement> ::= <simple conditional statement>
$OTHERWISE <alternatioe statement>$

<simple conditional statement> ::= <IF phrase> THEN
<conditioned statement>

<IF phrase> ::= IF <SL expression>

<conditioned statement> ::= <statement>

(alternative statement> ::= <statement>

*) THEN <-- , ; OTHERWISE <--> ELSE

Example:

```
IF X(i) = LOWER_BOUND(i)  THEN
   |_IF GRADIENT(i) > 0, MODIFIED_GRADIENT(i) := 0_|
ELSE IF X(i) = UPPER_BOUND(i)  THEN
   |_IF GRADIENT(i) < 0, MODIFIED_GRADIENT(i) := 0_|
ELSE MODIFIED_GRADIENT(i) := GRADIENT(i)
```

A simple conditional stateaent is executed as follows:

(1) The <SL expression> is evaluated.

(2) If the value of the <SL expression> is TRUE, then the conditioned statement is executed; otherwise the conditioned statement is skipped and the next statement is execute&

The effect of a conditional statement of the form

IF <SL expression> THEN <conditioned statement>
ELSE <alternative statement>

can be described by the following sequence of MPL statements:

```
IF ¬<SL expression> THEM GO TO ELSE_LABEL;
<conditioned statement>;  GO TO NEXT_STATEMENT;
ELSE-LABEL: <alternative statement>;
NEXT_STATEMENT:
```

'each ELSE <alternative statement) is to be paired with the innermost unpaired <simple conditional statement). The resulting syntactic ambiguity, known as the dangling ELSE problem, can be resolved.

Reference: Paul W, Abrahars, "A Final Solution to the Dangling ELSE of ALGOL 60 and Related Languages," Comm. ACM 9 (Sept. 1966), 679-682.

4.7. Iteration Statements

<iteration statement> ::= <FOR phrase> DO <iterated statement>

<FOR phrase> ::= FOR <control variable> IN <VA expression>
$: <SL expression>$

<control variable> ::= <identifier>

<iterated statement> ::= <statement>

*) DO <-- ,

Examples: FOR i IN <|1,...,N|> : i IN BASIS DO
CMIN := MIN(CMIN , C(i) - PRICES * A(*,i))
FOR x IN NODES, FOR y IN NODES : y IN SUCCESSOR(x),
CONNECTION(x,y) := TRUE

An iteration statement causes the iterated statement to be repeatedly executed for a sequence of zero or more values of the control variable in the FOR phrase. The control variable is implicitly declared as an ARITHMETIC SCALAR local to the iteration statement; thus its value is lost on exit unless it is assigned to a globally define? variable. The control variable may not be changed by assignment within the iterated statement.

The sequence of values of the control variable is evaluated before the iterated statement is executed. The effect of an iteration statement can be described by the following sequence of MPL statements:

```
LET i := <control variable>;
DEFINE S := <| <|i FOR i TN <VA expression> $:<SL expression>$|> |>;
IF S IS EMPTY, GO TO NEXT-STATEMENT;
DEFINE COUNT := 1;
LOOP:  DEFINE i := S(COUNT);  <iterated statement>;
COUNT := COUNT + 1;
IF COUNT <= LENGTH(S), GO TO LOOP;
NEXT-STATEMENT:
```

4.8. Let Statements

<dynamic let statement> ::= LET $IN <"block" label>$ <substitution list>

<static let statement) ::= $LET $IN <"block" label>$ <substitution list>

<substitution list> ::= <<symbol substituter>, LIST> |
<symbol substitute0 <WHERE phrase>

<WHERE phrase> ::= WHERE <<symbol substituter>, LIST>

<symbol substituter> ::= <synonym name> := <expression>

<synonym name> ::= <synonym identifer> $ (<<dummy argument>, LIST>) $

<synonym identifier> ::= <identifier>

<dummy argument> ::= <identifier>

Examples: LET CBV := COST(BASIC-VARIABLES)
LET GUB(k) := <|G(k),...,G(k+1)-1|>

A (static/dynamic) let statement serves to declare a (static/dynamic), synonym name. The scope of the synonym identifier is either the innermost block containing the let statement or the embracing block whose block label appears in the let statement.

A static synonym name acts as a compile-time macro: the erpressian replaces (see 3.1.6) the synonym name in the source text between the let statement and the end of the block or another let statement redeclaring the synonym name.

A dynamic synonym name acts as an expression variable: the value (an expression) replaces (see 3.7.6) the synonym nane at each run-time reference within the scope of the synonym identifier. The dynamic let statement serves as an assigneent statement for dynamic synonym names.

4.9. INPUT / OUTPUT Statements

<input statement> ::= GIVEN $IN <"block" label>$
<<defining phrase>, LIST>

<output statement) ::= ANSWER <<expression>, LIST>

Examples: GIVER ● ,n,l SCALAR, A MATRIX a by n, G VECTOR 1+1,
b COLUMN VECTOR m
ANSWER Status, BV, XBV, YBP, s, KV, GUB_BV

The INPUT/OUTPUT provided in MPL at present is rudimentary and intended merely as a first step toward more powerful concepts.

The GIVEN statement serves as a defining stateaent (see 4.2.1) as well as an input staterent. The variable name(s) is defined and assigned the corresponding input value(s). The type and dimensionality of the variable name(s) must be specified in the defining phrase of the GIVEN statement;: the domain and shape ray be specified in the defining phrase or vi.11 be taken from the data. The data is assumed to be labelled with the variable name(s) and to contain information on type, dimensionality, domain, and shape. These data attributes must be consistent with those specified in the defining phrase.

The ANSWER statement produces labelled printed output; i.e., the "name" of the expression precedes the value. Thus the statement

ANSWER X, SIN(2*PI*X)

will produce the printed output

X = **********, SIN(2*PI*X) = ********** .

SECTION 5. PROCEDURES

Procedures are subprograms that can be defined anywhere in the program and which are activated each time a procedure call is executed. When the procedure is called the formal parameters are replaced by actual parameters.

5.1. Procedure Definitions

```
<procedure definition> ::= EXTERNAL <procedure head> |
                           $INLINE$ (procedure head>; <procedure body> |
                           <one-line procedure definition>

<procedure head> ::= $<procedure attribute>$ PROCEDURE
                     $<formal return parameters> :=$ <procedure identifier>
                     $(<<formal input parameter>, LIST>)$
                     $WHERE <<parameter specification>, LIST>$

<procedure attribute> ::= INDEPENDENT | DEPENDENT | FUNCTION

<procedure identifier> ::= <identifier>

<formal return parameters) ::= <identifier> | {<<identifier>, LIST>)

<formal input parameter> ::= <identifier> | <serial formal parameter>

<Serial formal parameter> ::= <"E"identifier> FOR <bound identifier>
                  IN <"VA"identifier> $:<"SL"identif ier>$

<bound identifier> ::= <identifier>

<parameter specification> ::=
                      <<identifier>, LIST><<attribute> LIST><parameter type> |
                      <<identifier>, LIST> $<procedure attribute>$ PROCEDURE

<parameter type> ::= VALUE | NAME | VALUE RESULT | RESULT | <empty>

<procedure body> ::= <statement>

<one-line procedure definition> ::= <<attribute> LIST> $INLINE$
                 $<procedure attribute>$ PROCEDURE
                 <procedure identifier> $(<<formal input parameter>, LIST>)$
                 :=<expression> $WHERE <<parameter specification>, LIST>$

*)  FUNCTION PROCEDURE <-- FUNCTION
```

5.1.1. EXTERNAL procedures

The attribute EXTERNAL denotes a procedure that has to be fetched from a library outside the program; the procedure head in the program supplies only the neccessary information on its parameters.

5.1.2. INLINE procedures

A procedure can be specified INLINE so that if the implementation permits, the statements corresponding to the procedure as modified by the actual parameters (see 5.2.) will be inserted at the place where the procedure call occurs in the program. This enables the programmer to avoid the overhead associated with procedure calls if he desires. This process should, of course, Plot lead to a recursive situation.

5.1.3. The procedure head

The procedure head contains the name of the procedure and information (at least type and dimensionality) about its parareters. In order to conform to mathematical function notation, the procedure head is written as an explicit assignment to the return parameters. The input and return parameters are formal parameters i.e. the identifiers used here do not represent actual quantities and have to be replaced by actual parameters each time the procedure is called.

Each formal parameter must be specified in the parameter specification list. If a formal parameter represents a variable, at least its type and dirensionality must be indicated. The effect of the attribute VALUE is explained in (5.2.1). A serial formal paraaeter represents a list of actual input parameters depending on a POR phrase. The <bound identifier> represents the control variable of the FOR phrase. In a serial formal parameter only the attributes of the first identifier need be specified; the attribotes of the cthers are implied by their position in the FOR phrase(see 5.2.). The scope of the formal parameters in the procedure head is the procedure definition. The default parameter type is NAME independent and independent procedures.

5.1.4. The procedure body

The procedure body consists of the actual statements to be executed vhen the procedure is called (see 5.2.).

The procedure body is the scope for all identifiers defined within the procedure. The scope of identifiers can not be extended outside a procedure body. Procedure definitions may be nested, i.e. a procedure body may contain procedure definitions.

S.t.5. The procedure attributes

The procedure attributes specify subclasses of procedures with certain restrictions as to their parameters and to the way they are handled.

The most general case is a DEPENDENT procedure. A DEPENDENT procedure has free access to its environment and can use or change every quantity defined in the block containing .the procedure definition. DEPENDENT is the default value for the procedure attribute.

5.1.5.1. FUNCTION procedures

FUNCTION procedures are closely related to the mathematical concept of a function, i.e. they compute one or more return values from one or

more input parameters.

(i) They may not refer to any variable or label whose definition occurs outside the function procedure.

(ii) They may reference function procedures only, i.e. all procedure parameters of a function and all procedures defined inside a function or called from within a function must themselves be function procedures.

(iii) They may not contain any input/output statement except ANSWER statements (4.9).

(iv) No input parameters may be changed within the body of a function procedure. All input parameters are treated as if they were VALUE parameters. Assignment of values to input parameters is illegal.

(v) FUNCTION procedures may not have serial formal parameters.

5.1.5.2. INDEPENDENT procedures

INDEPENDENT procedures are identical to FUNCTION procedures except that they may have NAME parameters and serial formal parameters and may refer to both FUNCTION and INDEPENDENT procedures.

5J.2.3 One-line procedures

The one-line procedure is provided as a short way of witting a procedure which computes an expression (see 3). The value of this expression is returned to be used as an operand or in an assignment statement. The <<attribute> LIST> specifies the attributes of the computed expression.

5.1.6. Examples of procedure heads

```
PROCEDURE (OPT-X, OPT-Z)  := SIMPLEX(A,RHS,COSTS) where A
                             MATRIX, OPT_X,RHS COLUMN VECTOR,
                             COSTS ROW VECTOR, OPT_Z SCALAR

INLI N-E   PROCEDURE   RSLT :=  MAXIMUM (Y(i) FOR i IN S :L)
                                where Y(i), RSLT SCALAR

FUNCTION   PROCEDURE A= EXP(B) WHERE A,B MATRIX
```

5.2. Procedure Calls

```
<procedure call> ::= <procedure identifier>
                     $ ((<actual parameter>, LIST)$

<actual parameter> ::= <expression> | <procedure identifier> |
                       <serial actual parameter) |
                         "meaning the corresponding actual parameter
                          is omitted"
```

<serial actual parameter> ::= <expression><FOR phrase>

A procedure call causes the execution of the statement of the corresponding procedare body after these statements have been modified by actual parameters. Por a one-line procedure the expression is considered the body.

Procedure calls may occur in the following contexts:

(I) A procedure without formal return parameters may be called in a procedare statement (4.5).

(II) A procedure with one return parameter may be used as an expression operand (3.1.4). In this case a dummy variable is created having the attributes of the return parameter, and it is used as an expression operand after the completion of the procedure call.

(II?) Procedures with one or more return parameters can be called in an assignment statement (4.3). In this case the actual return parameters must he assignment compatible with the corresponding formal return parameter. Upon completion of the procedure call, the actual return parameters are assigned the final values of the corresponding formal parameters. If an actual return parameter is omitted (represented by"_"), then no assignment is made. If no value has been computed for the formal return parameter, then the actual return parameter will be UNDEFINED.

The number of actual parameters (both input and return parameters) must be the same as the number of formal parameters specified in the procedure head. Each actual parameter "replaces" the formal parameter in the same position and must be compatible with it in terms of its attributes (see 5.2.1, 5.2.2, 5.2.3 for details).

The effect of the execution of a procedure call is equivalent to the effect of executing the statements of the procedure body modified as illustrated in the rest of this section ("equivalent" because an implementation may choose any optimization strategy yielding the same result). After all modifications are completed, the procedure body must yield a sequence of valid MPL statements.

5.2.1. VALUE parameters

Input parameters representing a variable may be specified VALUE in the procedare head.

In general, for each VALUE parameter there is an internal procedure variable having the same attributes as the corresponding formal parameter. (In certain cases, in particular for FUNCTION PROCEDURES, this may be implemented differently).

Before execution of the procedure body begins, each actual parameter is evaluated-and the result is assigned to the corresponding

internal variable. Note, this implies that the internal variable (whose attributes are given in the procedure head) and the actual parameter must be assignment compatible as defined in 4.2.

If the actual parameter is omitted (i.e. it is "_", the underscore) no assignment is made and the formal parameter will be UNDEFINED.

Any occurrence of the formal parameter identifier inside the procedure body will then he replaced by the corresponding internal parameter.

If the parameter were to be changed within the procedure body by assignment, this will affect the internal variable only and not the actual parameter itself.

5.2.2. NAME parameters

All procedure parameters and those parameters denoting variables that have not been specified VALUE are called NAME parameters. The way in which actual name parameters replace any occurrence of the corresponding formal parameters is best described as textual substitution, i.e. the "name" of the actual parameter (enclosed in parenthesis wherever this is syntactically necessary) replaces the formal parameter,

Any change made to the formal parameter within the procedure body is reflected by the same change occurring to the actual parameter (note the difference from VALUE parameters).

Actual parameters called by name must have their attributes specified in the parameter specification. Actual procedure parameters must agree in their procedure attributes with the corresponding formal parameters.

It is possible, but not advisable, to omit a NAME parameter. If program control reaches a reference to an omitted NAME parameter, then an execution error will result.

5.2.3. Serial actual parameters and serial formal parameters

A serial actual parameter (SAP) is of the form

<expression><FOR phrase>

A SAP does not stand for a vector, but rather for a list of expressions controlled by the FOR phrase. The control variable (CV) of the FOR phrase has as its scope the SAP itself, i.e. any occurrence of the CV within the expression represents only the CV and not any other identifier defined in the embracing block.

The CV, as well as the other components of the FOR phrase (the VA expression and the SL expression, see 4.7) are available to the procedure individually and are accessed by means of the serial formal parameter. The CV is passed by NAME to the <bound identifier>; the

<expression> is passed by NAME to the <"E"identifier>; the <"VA"expression> of the FOR phrase is passed by NAME to the <"VA"identifier> ; and the <"SL"expression> of the FOR phrase is passed by NAME to the <"SL"idientifier>.

5.2.4. The return parameters

All formal return parameters must denote variables, and must be specified by a defining phrase in the procedure head. (VALUE is not meaningful for return parameters).

For each formal return parameter there is a internal variable having the attributes specified for the return parameter in the procedure head. This internal variable replaces any occurrence of the formal return parameter within the procedure body.

After execution of the procedure body the value of the internal variable is assigned to the actual return parameter if one exists (see 4.2.2) or is used as an operand.

5.2.5. The RESULT parameters

The RESULT parameter is handled in exactly the same way as a return parameter.

S.2.6. The VALUE RESULT parameter

VALUE RESULT parameters are handled like VALUE parameters before execution of the procedure body and like RESULT parameters, after execution of the procedure body.

5.3. Library Procedures

This section describes the use of several procedures which are provided in the MPL library. References to these procedures all have the form P(P) where F represents the name of the procedure and P represents a list of parameters. Where indicated, these procedures return values with attributes as described below.

ABS(SCALAR)

SCALAR	Any scalar valued arithmetic expression.
VALUE	The absolute value of 'SCALAR'.

ARGMAX(VECTOR)

VECTOR	Any vector valued arithmetic expression.
VALUE	The scalar arithmetic index of the first occurring maximum valued element of 'VECTOR'.

ARGMIN(VECTOR)

VECTOR	Any vector valued arithmetic expression.
VALUE	The scalar arithmetic index 'of the first occurring minimum valued element of *VECTOR@.

COLDIM(MATRIX)
MATRIX Any matrix valued expression.
VALUE The scalar arithmetic number of elements in the range of the second subscript of 'MATRIX'. This function is intended for finding the number of columns in a matrix, so if 'MATRIX' is a column vector, 'VECTOR' := 1.

DTM (VECTOR)
VECTOR Any vector valued arithmetic expression.
VALUE The scalar arithmetic number of elements in the range of 'VECTOR',

IDENTITY (RANK)
RANK The scalar arithmetic rank of the square identity matrix which is the 'VALUE' of the function.
VALUE An identity matrix with 'RANK' rows and columns.

INVERSE(MATRIX)
MATRIX A square, non-singular, matrix valued arithmetic expression.
VALUE The inverse of 'MATRIX'.

MAX(VECTOR)
VECTOR Any vector valued arithmetic expression.
VALUE The scalar arithmetic value of the minimum value4 element of 'VECTOR'.

MIN(VECTOR)
VECTOR Any vector valued arithmetic expression.
VALUE The scalar arithmetic value of the minimum valued element of 'VECTOR'.

ONES(ROWS,COLUMNS)
ROWS The integer scalar number of rows in 'VALUE'.
COLWMNS The integer scalar number of columns in 'VALUE'.
VALUE A matrix of ones with 'ROWS' rows and 'COLUMNS' columns.

ROWDIM (MATRIX)
MATRIX Any matrix valued arithmetic expression.
VALUE The scalar arithmetic number of elements in the range of the first subscript of 'MATRIX'. This function is intended for finding the number of rows in a matrix, so if 'MATRIX' is a row vector, 'VALUE' := 1.

SUM (VECTOR)
VECTOR A vector valued arithmetic expression.
VALUE The scalar arithmetic sum of the elements of 'VECTOR'.

TRANSPOSE(MATRIX)
MATRIX Any matrix valued arithmetic expression.
VALUE The transpose of 'MATRIX'. If 'MATRIX' has m rows and n columns, then 'VALUE' has n rows and m columns.

TRUNCATE(SCALAR)
SCALAR Any scalar valued arithmetic expression.
VALUE sign of 'SCALAR' times largest integer < ABS('SCALAR').

ZEROES(ROWS,COLUMNS)
ROWS The integer scalar number of rows in 'VALUE'.
COLUMNS The integer scalar number of columns in 'VALUE'.
VALVE A matrix of zeroes with 'ROWS' rows and 'COLUMNS' columns.

Two Examples of Mathematical Programming Algorithms Written In MFL

SIMPLEX ALGORITHM

GENERALIZED UPPER BOUND

The latter represents an algorithm for solving certain large scale problems. Systems of this type, encountered in practice, have run over 30,000 equations and half-million variables. In order to develop efficient codes, it is necessary that experimental programs be highly readable, easy to debug, so that various versions can be quickly tested and compared.

PROGRAM SIMPLEX-ALGORITHM:

"Find Max x(1), x(j) > 0 *for* j in {2,...,n}:

. A*x = b, b ≥ 0

where V = {V(1), V(2),...,V(m)}, *the index set of the initial basis, is given with* V(1) = 1 *corresponding to objective* x(1). *It is assumed that* A(v) = Identity(m)."

Given m, n scalars, A matrix m by n, b column vector m,

V vector m;

Let I = {1,...,m}; Let J = {1,...,n};

RECYCLE : Define *"incoming variable* s *and* δ *the minimum relative cost"*

(s,δ) = ARG MIN [A(l,j) *for* j ε J:j ≠ 1];

If δ > 0, [Answer -Bounded-, V, b; Go to FINI];

"otherwise" define *"Pivot row* r *and level* θ *of incoming variable* x(s)."

(r,θ) = ARG MIN [b(i)/A(i,s) *for* i ε I:A(i,s) > 0];

If θ = +∞, [Answer <<Unbounded>>, V, b, s , A(*,s) ; Go to FINI];

"Update" A = PIVOT [A,A,(*,s),r]; b = PIVOT [b,A(*,s),r]; V(r) = s ;

'where PIVOT *pivots matrix* A *on* A(r,s) *and* **returns** *modified* A."

Go to RECYCLE;

FINI: End *"program"*

GENERALIZED UPPER BOUND

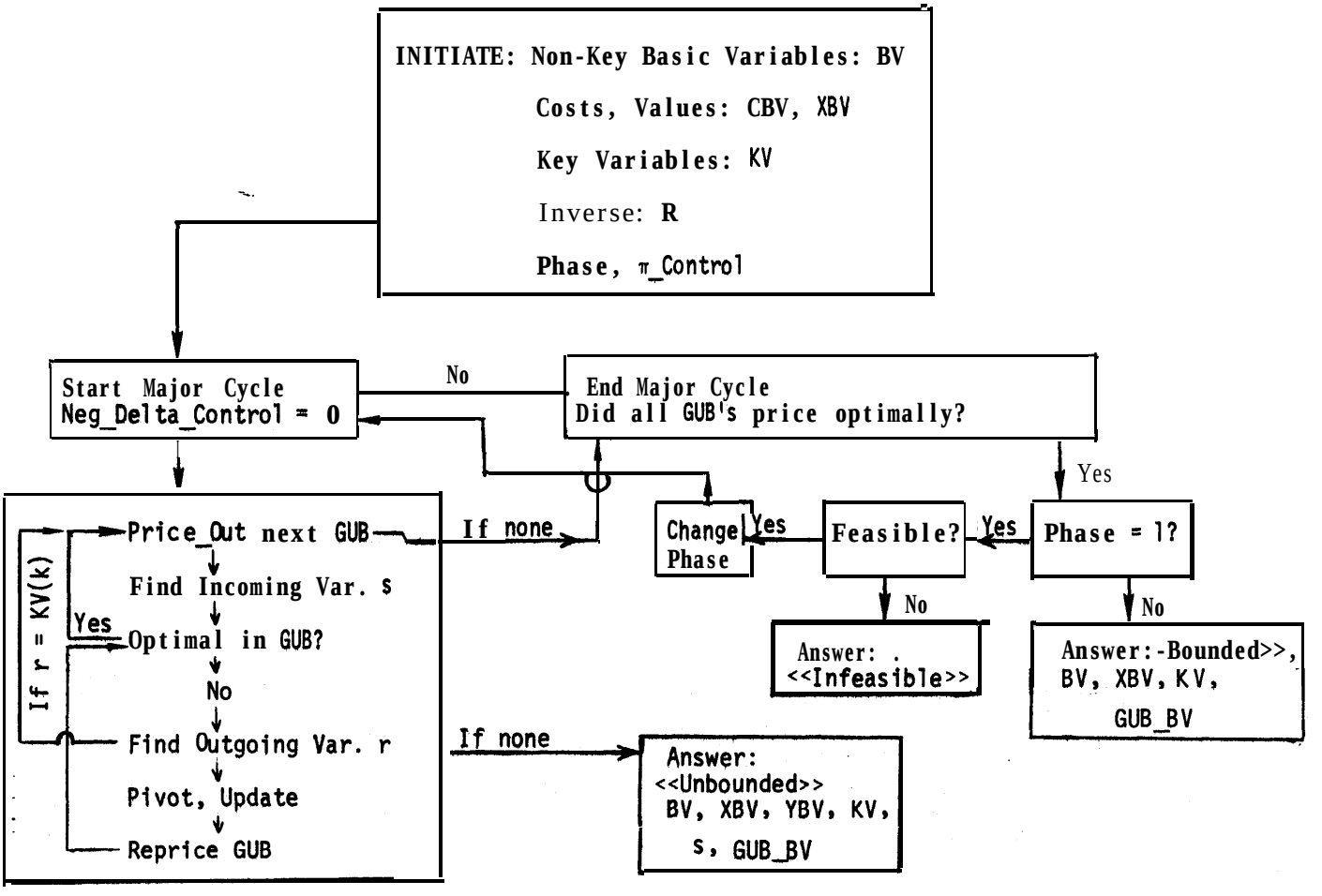
GIVEN **A**, **m** × **n**, FIND MAX xn:

$Ax = b, x_j \geq 0$ **for** $j = \{1,\ldots,n-1\}$,

$$\sum_{j \in GUB(k)} x_j = 1, \quad k = \{1,\ldots,\ell-1\}$$

where

$$GUB(k) = \left\{ j \mid G(k) \leq j \leq G(k+1)-1 \right\}$$



PROGRAM GENERALIZED-UPPER-BOUND:

"The description of this algorithm is written in M.P. L. *(Mathematical Programming Language). Commentary such as this on the algorithm are* **enclosed** *in quotes. Underlines, not part of the Zanguage, are to help the reader identify the special (reserved) words of the Zanguage. "*

GIVEN m,n,l scalar, A matrix m by n, G vector l+1,b column vector m;

"The problem is to find

Max x(n), x(j) ≥ 0 for j in {1,...,n-1}

subject to

(i) A * x = b

(ii) SUM[x(i) for i in GUB(k)] = 1

for k *in* Cl,. . . J-13 *where we"*

LET GUB(k) = {G(k),...,G(k+1)-1} ;

'which we call a' ' GUB' ' *set. The last variable* x(n), is *to be maximized. The set* {x(n-m+1),...,x(n-1)} *contains the artificial variables and the matrix* [A(*,n-m+1),...,A(*,n)] **forms an** *identity matrix. We also assume that*

G(1) = 1, G(i) ≤ G(i+1), G(l+1) = n-m+1

Note that GUB(l) = {G(l),...,n-m} *are the variables which have no partial sum* conditions *(ii) associated with them. "*

LET L = {1,...,l} ; LET I = {1,...,m} ;

"Initiate Non *key Basic Variables* BV. *Let the* GUB *set* **of** *the* i-*th basic variable* BV(i) *be denoted by* GUB_BV(i) ."

DEFINE BV vector m, GUBBV vector m;

For i in I do [BV(i) = n-m+i; GUB BV(i) = l+1]

"Key basic variables are denoted by KV. *In each* GUB *we initially select for the key variable the one with lowest cost coef."*

DEFINE KV vector Z-1;

KV(k) = Index_Min [A(m,j) for j in GUB(k)]

for k in L : k ≠ 1;

"where Index_Min is the name of a function that yields the index (or argument) where the minimum is attained."

"The Inverse of the columns corresponding to BV , as modified by subtracting off their key columns, will be denoted by R. Initially R is given by:"

DEFINE R = Identity (m);

"The modified RHS is denoted by b' . It is formed by setting key variables = 1, substituting into (i) and subtracting from b."

DEFINE b' = b - SUM (A(*,j) for j in KV);

For i in I: b'(i) < 0 and i ≠ m, ⌊ b'(i) = -b'(i); R(i,i) = -1 ⌋;

"where we have corrected R and b' so that the initial basic sotution is feasible."

"Because R is an adjusted identity, the values of BV in the initial basic solution XBV are"

DEFINE XBV = b';

"In Phase = 1 the cost coefficients are all zero except for j in {n-m+1 , . . , ,n-1} where the coef. are each one. For j in {1,...,n-m} the oost coef. remain zero after subtraction of their key columns. Non-key basic cost coef. are denoted by CBV. Initially"

DEFINE CBVRow_of_ones(m); CBV(m) = 0;

"Finally we set up two scalar control parameters, the Phase control and the π_Control where the tatter, if 1, is a signal to

compute new π values for the next major or minor cycle. "

DEFINE Phase = 1; DEFINE π_Control = 1;

"START MAJOR CYCLE"

MAJOR-CYCLE: **DEFINE** Neg_Delta_Control = 0;

"where the latter **counts** *the* **number of** *columns that* **price-out** *negative. We* **now** *get ready to price out the various* GUB's k. *"*

FOR k in L do

"*The* **FOR** *loop ends just after* RECYCLE *label"*

MINOR CYCLE: If π Control = 1 and Phase = 1,

DEFINE π = -CBV*R;

If π Control = 1 and Phase = 2,

DEFINE π = R(m,*);

"i.e. the above computes the price vector π. If π_Control = 0, *it is not necessary to compute π and the above steps are skipped. "*

"We are now ready to price out next **GUB** **(or** *reprice the same* GUB*). But first we reset"*

π_Control = 0;

"PRICE OUT GUB"

DEFINE (s,d) = ARG MIN[π*A(*,j) for j in GUB(k)];

"where **ARG MIN is a** *function that returns* **s** *and* d. **s** *is the smallest argument (index) for which the minimum value* d *is attained. Let δ be the priced-out value of column* **s** *after it is corrected for the price on its* GUB *equation* (for k<l)*."*

If k < l, define δ = d-π*A(*,KV(k));

If k = l, define δ = d;

"PRICE OUT NEXT GUB"

If δ ≥ 0, go to RECYCLE;

'Where recycle is the label at the end of **for** *loop of the minor*

cycle, so that **k is** *incremented to* **k+1 and** *pricing starts again with next* **GUB.** *However* if $\delta < 0$, *then we want to introduce column s into the basis and* find *(if possible) column* **r** to *drop from the basis."*

"REPRESENT THE INCOMING COLUMN s IN TERMS OF BASIS"

If k = 1, define YBV = R*A(*,s);

If k < 1, define YBV = R*[A(*,s)-A(*,KV(k))];

"DETERMINE COLUMN r TO DROP FROM BASIS"

"We first apply the usual ratio test **for** *the non-key basic columns* **BY.** *"*

DEFINE (r,θ) = ARG MIN [[XBV(i)/YBV(i)] for i in I: ifm and YBV(i)>0];

"If $\theta = +\infty$ *above, it means no pivot oan be found among the variables* **BV(i).** *The basic variable corresponding to* **r** *is* **BV(r) .** *We are now interested in discovering if a key-basso variable* **j** *witt drop because it has a lower ratio than those* of **BV(i).** *If yes we reset* **r** *equal to this* **j** *and denote Cts* **GUB** *set as* **GUB_r .** *Initially we set* **GUB_r = 0;** *as tong as it remains zero it means* **r** above **is** *stilt the winner. If* **GUB_r** > *0 then by definition* **KV(GUB_r) = r."**

DEFINE GUB_r = 0;

"TEST RATIO FOR KEY **VARIABLES KV**"

"At thts point we need to know the values of **KV(i)** *which we denote* **XKV** *and the corresponding representation of column* s *denoted* **YKV.** *Since* **GUB** *'s with a unique basic variable don't drop their basic variable under a basis change (except* **i = k** *possibly) we need to consider only those* **GUB's** *that have non-key basic variables. We* **now** *Zook for such* **GUB** *'s."*

DEFINE a = 0;

f-LOOP: DEFINE f = MIN [GUB_BV(i) for i in I:GUB_BV(i)>a];

If f ≥ 1, go to LOOP_EXIT else a = f;

"The above if iterated will pick up successively the next higher index f of *the* GUB's of BV(i)."

LET F = {j for j in I: GUB_BV(j) = f} ;

DEFINE XKV = 1-SUM[XBV(i) for i in F];

DEFINE YKV = -SUM[YBV(i) for i in F];

YKV = 1+YKV if f = k;

"where the latter corrects YKV if *the incoming column s is* **in** GUB *set* f *i.e.* if f = k. *Note that the above states that we can obtain the values of the key variab Zes by* **plugging in** *the* **values of** XBV(i) *into the set* **equations (ii).** **We** *now do the ratio test on* (XKV/YKV)."

If YKV ≤ 0 or (XKV/YKV) ≥ 8, go_to f_LOOP;

"otherwise" r = KV(f); θ = XKV/YKV; GUB r = f;

"We temporarily store the winning XKV *and* YKV;

DEFINE XTEMP = XKV; DEFINE YTEMP = YKV; Go_to f_LOOP;

'This completes all the ratio tests except for the possibility that s *reaches its upper bound and becomes the key_variable in place* **of** KV(k) **for** k 4 1.

LOOP-EXIT: If k = 1 and θ = +∞, go to UNBOUNDED;

If k < 1 and θ > 1, [KV(k)=s;XBV=XBV-YBV; Go to_RECYCLE];

"In the latter case the incoming variable s reaches its upper bound and replaces the key variable in the same GUB. *Accordingly we reset* KV(k) = *s and adjust* XBV, *Then by recycling we go* **on** *to price out* **next** GUB. *Otherwise we are ready for the pivot."*

π_Control = 1; Neg_Delta_Control = 1;

"SWAP KEY AND NON-KEYBASIC VARIABLES"

"*Swapping is not needed in the* following *case.* "

If GUB_r = 0, go to UPDATE;'

"*If* GUB_r > *0, then it is necessary to interchange key variable* r *with another basic variable* t *in the same* GUB *set as* r. "

DEFINE t = MIN[i for i in I: GUB BV(i) = GUB r];

"*We now swap* r *and* t."

r = t; KV(GUB r) = BV(t) ; XBV(r) = XTEMP; YBV(r) = YTEMP;

"*We must now fix up the inverse.* "

R(r,*) = -SUM[R(i,*) for i in I :GUB BV(i) = GUB BV(r)]

"UPDATE BASIS, PIVOT'

XBV = XBV-YBV*θ; XBV(r) = θ;

UPDATE: BV(r) = s; CBV(r) = 0; GUB_BV(r) = k;

R = PIVOT (R,YBV,r); XBV = PIVOT (XBV,YBV,r);

"*The function* PIVOT *pivots in the last column of the matrix* (R,YBV) *on row* r *and returns the modified* R *columns.* , *Having pivoted we now go back and reprice the* GUB *and* continue *doing this until the* GUB *prices out optimally before going on to next* GUB."

Go to MINOR-CYCLE ;

"*When the* GUB *prices out optimally* or s goes *to its upper bound* we *price out next* GUB *by going to the recycle label at end of the for loop which now follows.* "

RECYCLE:]; "*End for k in L do loop*"

"*After pricing out all* GUB's *the for statement reaches its end designated above by*] *and control moves to the next statement below?*"

If Neg_Del ta_Control > 0 , go to MAJOR-CYCLE ;

"*i. e. starting the pricing over again beginning with the first* GUB."

"*If* Neg_Del ta_Control = 0, *it means we are optimal and we either initiate* Phase = 2 *or terminate with the optimal solution.*"

"TERMINATE"

If Phase = 1 ,⌊2 if SUM[XBV(i) for i in I:BV(i)>n-m and i≠m]>0 then

⌊3 Answer -Infeasible=; Go to FIN1 3⌋

else "set" Phase = 2; π_Control = 1; Go to MAJOR CYCLE 2⌋;

If Phase = 2,

⌊4 Answer <<Bounded>>, GUB BV, BV, XBV, KY; Go to FIN1 4⌋ ;

UNBOUNDED; "Phase = 2 and θ = +∞"

DEFINE

Answer <<Unbounded>>, BV, XBV, YBV, s, KV, GUB_BV;

FIN1 : END *"Program"*