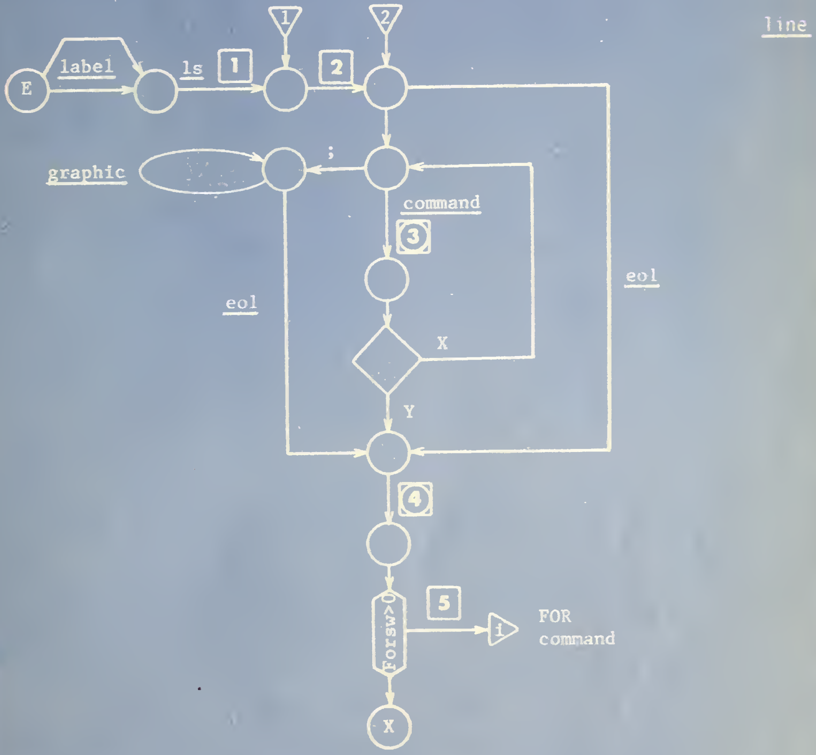
NBS Handbook 118

U.S. DEPARTMENT OF COMMERCE / National Bureau of Standards

MUMPS LANGUAGE STANDARD



NATIONAL BUREAU OF STANDARDS

The National Bureau of Standards[1] was established by an act of Congress March 3, 1901. The Bureau's overall goal is to strengthen and advance the Nation's science and technology and facilitate their effective application for public benefit. To this end, the Bureau conducts research and provides: (1) a basis for the Nation's physical measurement system, (2) scientific and technological services for industry and government, (3) a technical basis for equity in trade, and (4) technical services to promote public safety. The Bureau consists of the Institute for Basic Standards, the Institute for Materials Research, the Institute for Applied Technology, the Institute for Computer Sciences and Technology, and the Office for Information Programs.

THE INSTITUTE FOR BASIC STANDARDS provides the central basis within the United States of a complete and consistent system of physical measurement; coordinates that system with measurement systems of other nations; and furnishes essential services leading to accurate and uniform physical measurements throughout the Nation's scientific community, industry, and commerce. The Institute consists of the Office of Measurement Services, the Office of Radiation Measurement and the following Center and divisions:

Applied Mathematics — Electricity — Mechanics — Heat — Optical Physics — Center for Radiation Research: Nuclear Sciences; Applied Radiation — Laboratory Astrophysics[2] — Cryogenics[2] — Electromagnetics[2] — Time and Frequency[2].

THE INSTITUTE FOR MATERIALS RESEARCH conducts materials research leading to improved methods of measurement, standards, and data on the properties of well-characterized materials needed by industry, commerce, educational institutions, and Government; provides advisory and research services to other Government agencies; and develops, produces, and distributes standard reference materials. The Institute consists of the Office of Standard Reference Materials, the Office of Air and Water Measurement, and the following divisions:

Analytical Chemistry — Polymers — Metallurgy — Inorganic Materials — Reactor Radiation — Physical Chemistry.

THE INSTITUTE FOR APPLIED TECHNOLOGY provides technical services to promote the use of available technology and to facilitate technological innovation in industry and Government; cooperates with public and private organizations leading to the development of technological standards (including mandatory safety standards), codes and methods of test; and provides technical advice and services to Government agencies upon request. The Institute consists of the following divisions and Centers:

Standards Application and Analysis — Electronic Technology — Center for Consumer Product Technology: Product Systems Analysis; Product Engineering — Center for Building Technology: Structures, Materials, and Life Safety; Building Environment; Technical Evaluation and Application — Center for Fire Research: Fire Science; Fire Safety Engineering.

THE INSTITUTE FOR COMPUTER SCIENCES AND TECHNOLOGY conducts research and provides technical services designed to aid Government agencies in improving cost effectiveness in the conduct of their programs through the selection, acquisition, and effective utilization of automatic data processing equipment; and serves as the principal focus within the executive branch for the development of Federal standards for automatic data processing equipment, techniques, and computer languages. The Institute consists of the following divisions:

Computer Services — Systems and Software — Computer Systems Engineering — Information Technology.

THE OFFICE FOR INFORMATION PROGRAMS promotes optimum dissemination and accessibility of scientific information generated within NBS and other agencies of the Federal Government; promotes the development of the National Standard Reference Data System and a system of information analysis centers dealing with the broader aspects of the National Measurement System; provides appropriate services to ensure that the NBS staff has optimum accessibility to the scientific information of the world. The Office consists of the following organizational units:

Office of Standard Reference Data — Office of Information Activities — Office of Technical Publications — Library — Office of International Relations — Office of International Standards.

[1] Headquarters and Laboratories at Gaithersburg, Maryland, unless otherwise noted; mailing address Washington, D.C. 20234.

[2] Located at Boulder, Colorado 80302.

MUMPS Language Standard

t. Handbook, no. 118,

Edited by

Joseph T. O'Neill

Systems and Software Division
Institute for Computer Sciences and Technology
National Bureau of Standards
Washington, D.C. 20234

Sponsored by

National Center for Health Services Research
Health Resources Administration
Department of Health, Education, and Welfare
Rockville, Maryland 20852

U.S. DEPARTMENT OF COMMERCE, Rogers C. B. Morton, *Secretary*

James A. Baker, III, *Under Secretary*

Dr. Betsy Ancker-Johnson, *Assistant Secretary for Science and Technology*

US
111
NATIONAL BUREAU OF STANDARDS, Ernest Ambler, *Acting Director*

Issued January 1976

Library of Congress Cataloging in Publication Data

MUMPS Development Committee.
MUMPS Language Standard.

(National Bureau of Standards Handbook; 118)
Supt. of Docs. No.: C13.11.118.
1. MUMPS (Computer program Language) I. Institute for Computer Sciences and Technology. Systems and Software Division. II. National Center for Health Services Research. III. Title. IV. Series: United States. National Bureau of Standards. Handbook; 118.
QA76.73.M85M85 1975 001.6′424 75-619261

National Bureau of Standards Handbook 118

Nat. Bur. Stand. (U.S.), Handb. 118,144 pages (Jan. 1976)
CODEN: NBSHAP

U.S. GOVERNMENT PRINTING OFFICE
WASHINGTON: 1976

For sale by the Superintendent of Documents, U.S. Government Printing Office, Washington, D.C. 20402
(Order by SD Catalog No. C13.11 :118). Price $2.70 (Add 25 percent additional for other than U.S. mailing).

FOREWORD

MUMPS, an acronym for Massachusetts General Hospital Utility Multi-Programming System, is a high-level, interactive computer programming language developed for use in complex data handling operations.

Part I of this Standard, the MUMPS Language Specification, MDC/28, contains the narrative description of the MUMPS language which was adopted and approved for publication as a Type A release of the MUMPS Development Committee on March 12, 1975; it supersedes the Partial MUMPS Language Standard, MDC/25, of October 14, 1974 and the more recent marked proposal, titled Interim MUMPS Language Specification, MDC 1/8, of February 10, 1975.

Part II, the MUMPS Transition Diagrams, MDC/33, consists of a formal definition of the language described in Part I. It was adopted and approved for publication as a Type A release of the MUMPS Development Committee on September 17, 1975; it supersedes the earlier publications with the same title, numbered MDC 1/9 and dated March 12, 1975 and August 8, 1975.

Part III, the MUMPS Portability Requirements, MDC/34, identifies constraints on MUMPS programs and implementations required for portability of MUMPS application code. It was adopted and approved for publication as a Type A release of the MUMPS Development Committee on September 17, 1975; it supersedes the earlier draft proposal with the same title, numbered MDC 1/10 and dated May 23, 1975.

As a MUMPS user and charter member of the MUMPS Development Committee, the National Bureau of Standards is pleased to have the opportunity to make this information available through publication of this NBS Handbook.

Ruth M. Davis, Director
Institute for Computer
Sciences and Technology
National Bureau of Standards
Washington, D.C. 20234

October 14, 1975

ABSTRACT

This NBS Handbook contains a three-part description of various aspects of the MUMPS computer programming language. Part I, the MUMPS Language Specification, consists of a stylized English narrative definition of the MUMPS language which was adopted and approved for publication as a Type A release of the MUMPS Development Committee on March 12, 1975. Part II, the MUMPS Transition Diagrams, represents a formal definition of the language described in Part I, employing a form of line drawings to illustrate syntactic and semantic rules governing each of the language elements; it was adopted and approved for publication as a Type A release of the MUMPS Development Committee on September 17, 1975. Part III, the MUMPS Portability Requirements, identifies constraints on the implementation and use of the language for the benefit of parties interested in achieving MUMPS application code portability; it was adopted and approved for publication as a Type A release of the MUMPS Development Committee on September 17, 1975.

A bibliography of other MUMPS Development Committee documents is included.

Key words: Data handling language; interactive computing; interpretive computer programming language and operating system; medical automation; minicomputer-based systems; MUMPS Development Committee; MUMPS Language Standard.

PREFACE

The reader is hereby notified that the language specifications contained in this Standard have been approved by the MUMPS Development Committee but that they may be partial specifications which rely on information appearing in many parts of the MUMPS specifications. The specifications are dynamic in nature, and the changes reflected by these approved releases may not correspond with the latest specifications available.

Because of the evolutionary nature of MUMPS specifications, the reader is further reminded that changes are likely to occur in the specifications released herein prior to a complete republication of MUMPS specifications.

This document may be reproduced in any form so long as acknowledgment of the source is made. Anyone reproducing it is requested to include this preface.

R. Peter Ericson, Chairman
MUMPS Development Committee
The Institute of Living
Hartford, Connecticut 06106

September 17, 1975

ACKNOWLEDGMENTS

This document is the result of a cooperative effort on the part of the MUMPS Development Committee, chaired by R. Peter Ericson of the Institute of Living, its Subcommittee on Standard Specifications and Language Development, chaired by David A. Bridger of Artronix, Inc., and that Subcommittee's Task Group on MUMPS Standardization, chaired by Jack Bowie of the Massachusetts General Hospital. Part I was prepared by Melvin E. Conway under Contract No. 5-35770 with the National Bureau of Standards. Part II was prepared by David D. Sherertz and Anthony I. Wasserman under Contract No. 4-35840 with the National Bureau of Standards. Part III was prepared by Edward A. Gardner and Carl B. Lazarus under Purchase Orders No. 409767 and 512576, respectively, from the National Bureau of Standards. The work was authorized under the terms of an interagency agreement between the National Center for Health Services Research, Health Resources Administration, U.S. Department of Health, Education, and Welfare and the Institute for Computer Sciences and Technology, National Bureau of Standards, U.S. Department of Commerce.

MDC Participants

Institution	Participant(s)
Advanced Medical Systems Corporation	Stanley M. Rose
Artronix, Inc	David A. Bridger
B-D Spear	Charles M. Smith
Baylor College of Medicine	David B. Brown, Rudolph F. Trost
Beth Israel Hospital	Robert F. Beckley III
Community EKG Interpretive Service	James Cruce
Dartmouth College	William Campbell
Department of Health, Education, and Welfare (HRA)	Donald R. Barnes, William V. Glenn, Jr.
Department of Health, Education, and Welfare (NIH)	David B. Swedlow
Department of Health, Education, and Welfare (NLM)	David C. Hartmann
Digital Equipment Corporation	Roger S. Gourd, Robert D. Shear, Elisabeth Sopka
Georgetown University	David L. Williams
Georgia Institute of Technology	Fred R. Sias
Health Care Management Systems, Inc	Leonard L. Hurst
Institute of Living	R. Peter Ericson
Interpretive Data Systems, Inc	Paul L. Egerman, Carl B. Lazarus, Phillip T. Ragon
Jefferson Medical College	Robert F. Curley
Massachusetts General Hospital	G. Octo Barnett, Jack Bowie, Robert L. Rees, Craig J. Richardson
Meditech, Inc	A. Neil Pappalardo, Richard J. Pietravalle
MITRE Corporation	Richard E. Zapolin
National Bureau of Standards	Melvin E. Conway, Edward A. Gardner, Martin E. Johnson, Joseph T. O'Neill
Northeastern University	Wendy D. Mela
Regional Health Resource Center	Thomas T. Chen, Henry A. Warner
Stanford University Medical Center	Russell Briggs, Robert A. Greenes
University of California, Davis	Richard F. Walters, Jerome C. Wilcox
University of California, San Francisco	David D. Sherertz, Anthony I. Wasserman
University of Massachusetts	Jeffrey Rothmeier
University of Missouri	James L. Lehr
University of Pennsylvania	Martin Pring
University of Tennessee	Larry J. Peck
University of Washington (Seattle)	Arden W. Forrey
University of Wisconsin	Ellis A. Bauman, Gary S. Holmes
Washington University (St. Louis)	W. Edward Long, Joan Zimmerman

TABLE OF CONTENTS*

*Note: Each of the three parts is preceded by a more detailed table of contents.

MUMPS LANGUAGE STANDARD

Part I: MUMPS Language Specification

Table of Contents

1. Overview of MUMPS Language Specification

1.1 Organization of This Document

This document describes the MUMPS language at two levels of detail. Subsection 1.2 gives an overview of the prominent features of the language, intended for the reader who is already familiar with at least one existing dialect. Section 3 describes the static syntax of the language. The distinction between "static" and "dynamic" syntax is as follows. The static syntax describes the sequence of characters in a program as it appears on a tape in program interchange or on a listing. The dynamic syntax describes the sequence of characters actually encountered by an interpreter during execution of the program. The dynamic syntax takes into account transfers of control and values produced by indirection. Section 2 describes the metalanguage used for the static syntax.

1.2 Summary of the Language

1.2.1 Character Set

The character set which is used for the interchange of MUMPS programs and data is the seven-bit USA Standard Code for Information Interchange (ASCII) defined by ANSI X3.4-1968. Programs may be written entirely with the common 64-character subset of ASCII. The character collating sequence is the same as the numeric sequence of the ASCII character codes.

1.2.2 Routine Structure

A MUMPS routine consists of a sequence of lines. For purposes of transfer of control, lines may be optionally labeled. A label is either a conventional MUMPS name (an initial letter or % followed by alphanumerics), or it is an integer literal.

1.2.3 Program Punctuation

The following special characters may occur in programs.

Unary Arithmetic Operators

+	plus
-	negate

Unary Logical Operator

'	not

Binary Arithmetic Operators

+	addition
-	subtraction
*	multiplication
/	division
\	division with integer quotient
#	modulo

Binary Relational Operators

<	numeric less than
'<	numeric greater than or equal
>	numeric greater than
'>	numeric less than or equal
=	string identity
'=	string nonidentity
[	string contains
'[	string not-contains
]	string follows
']	string not-follows
?	string pattern match
'?	string pattern nonmatch

Binary Logical Operators

&	and
'&	nand
!	or
'!	nor

Binary String Operator

_	concatenation

Delimiters

,	argument separation, subscript separation
=	value assignment
:	post-conditional expression, subargument separation
( )	grouping
@	indirection
"	string literals
.	decimal point in numeric literals
^	preceding routine name in DO, GOTO
E	preceding exponent in numeric literals
;	comment
space	separating command words

Prefixes

^	global variable names
$	functions, special variable names
%	available in names of the programmer's choice

1.2.4 Data Types

Arithmetic operations are performed on strings and produce numeric values, which are special cases of strings. This approach to the standard specification does not preclude the use of multiple data representations within an implementation of the standard.

Any string value may enter into an arithmetic operation; there is a uniform rule for interpreting a string as a number. Certain operations deal with integer values, which are special cases of numeric values; the latter may contain decimal fractions. There is a uniform rule for interpreting any number (and, by inference, any string) as an integer.

Certain other operations deal with truth values, which are special cases of numeric values. There are two truth values: 0 and 1. The integer value 0 is the truth value 0. The integer value 1 is the truth value 1. All other numeric values are interpreted as the truth value 1. The truth value 0 denotes False; the truth value 1 denotes True.

1.2.5 Precedence of Operators

All binary operators are at the same level of precedence. Application of unary operators precedes application of binary operators.

1.2.6 Commands

At present, the standard contains only program-mode ("indirect") commands, of which the following are defined.

BREAK provides an access point within the standard for non-standard programming and debugging aids.

CLOSE releases one or more devices from ownership.

DO provides a generalized subroutine call.

ELSE permits conditional execution.

FOR controls repetitive execution over a set of values of a variable.

GOTO provides a generalized transfer of control.

HALT terminates execution.

HANG suspends execution for a specified period of time.

IF permits conditional execution.

KILL controls the elimination of specified variables and their values.

LOCK provides a generalized interlock facility for coordinating concurrent processes.

OPEN obtains ownership of one or more devices.

QUIT defines an exit point of FOR or DO.

READ specifies data input.

SET assigns values to variables.

USE designates a specific device for input and output.

VIEW provides an access point within the standard for the examination of machine-dependent information.

WRITE specifies data output.

XECUTE permits execution of strings arising from the expression evaluation process.

Z reserved for implementation-specific extensions.

All other command words, except those beginning with Z, are reserved.

1.2.7 Functions

The following functions are currently specified.

$ASCII selects a character of a string and returns its code as an integer.

$CHAR translates a set of integers into a string of characters whose codes are those integers.

$DATA returns an integer specifying whether a defined value and/or pointer of a named variable exists.

$EXTRACT returns a character or substring of a string expression, selected by position number.

$FIND returns an integer specifying the end position of a specified substring within a string.

$JUSTIFY returns the value of an expression, right-justified within a field of specified size.

$LENGTH returns the length of a string.

$NEXT returns the lowest numeric subscript value on the same level, but numerically higher than the last subscript of the named global or local variable.

$PIECE returns a string between two specified occurrences of a specified substring within a specified string.

$RANDOM returns a pseudo-random number in a specified interval.

$SELECT returns the value of one of several expressions in a list, selected by the truth values in a second list of expressions.

$TEXT returns the text content of a specified line of the routine in which the function appears.

$VIEW reserved for implementation-specific methods of obtaining machine-dependent data.

$Z reserved for definition of implementation-specific functions.

All other initial letters of function names are reserved.

1.2.8 Special Variables

The following special variables are specified.

$HOROLOG provides the date and time in a single, two-part value.

$IO identifies the currently assigned I/O device.

$JOB has an integer value which uniquely identifies the process which evaluates it.

$STORAGE provides the number of unused characters which remain in a routine's partition.

$TEST makes available the truth value determined by the IF command and by the OPEN, LOCK, and READ with timeouts.

$X gives the horizontal cursor position on the current device.

$Y gives the line number on the current device.

$Z reserved for implementation-specific definitions.

All other initial letters of special variable names are reserved.

2. Static Syntax Metalanguage

The primitives of the metalanguage are the ASCII characters and the metalanguage operators ::= (definition), [] (option), || (grouping), ... (optional indefinite repetition), L (list), and V (value).

In general, defined syntactic objects will have designations which are underlined names spelled with lower-case letters, e.g., name, expr, etc. Concatenation of syntactic objects is expressed in the static syntax by horizontal juxtaposition. Choice is expressed by vertical juxtaposition. The ::= symbol denotes a syntactic definition. An optional element is enclosed in square brackets [], and three dots ... denote that the previous element is optionally repeated any number of times. The definition of name, for example, is written:

```
name  ::=  |  %    |  [ digit ]  ...
           | alpha |  [ alpha ]
```

The vertical bars are used only to group elements for repetition or to make a group of elements more readable. When there is any danger of confusing the square brackets in the metalanguage with the ASCII graphics [and], special care is taken to avoid this. Normally, the square brackets will stand for the metalanguage symbols.

The unary metalanguage operator L denotes a list of one or more occurrences of the syntactic object immediately to its right, with one comma between each pair of occurrences. Thus,

```
L name  is equivalent to  name [ , name ] ...   .
```

The binary metalanguage operator V, used in the specification of indirection, places the constraint on the expratom to its left that it must have a value which satisfies the syntax of the syntactic object to its right. For example, one might define the syntax of a hypothetical EXAMPLE command with its argument list by

```
examplecommand  ::=  EXAMPLE ␣ L exampleargument
```

where

```
                     | expr                         |
exampleargument  ::= |                              |
                     | @ expratom V L exampleargument |
```

This says that, after evaluation of indirection, the command argument list consists of any number of exprs separated by commas. In the static syntax (i.e., prior to evaluation of indirection), occurrences of @ expratom may stand in place of nonoverlapping sublists of command arguments. Usually, the text accompanying a syntax description incorporating indirection will describe the syntax after all occurrences of indirection have been evaluated.

3. Static Syntax

3.1 Basic Alphabet

The *routine*, which is the object whose static syntax is being described in Section 3, is a string made up of the following 98 symbols.

The 95 ASCII graphics, including SP (space)
The line-start symbol *ls*
The end-of-line symbol *eol*
The end-of-routine symbol *eor*

In program interchange, the following ASCII characters are used in place of *ls*, *eol*, and *eor*.

ls:	SP
eol:	CR LF
eor:	CR FF

When a program is stored internally, the standard does not specify what forms *ls*, *eol*, and *eor* take. They may, in fact, be expressed by means other than characters in the program. When a program is entered from a keyboard, the standard does not specify what operator procedures correspond to *ls*, *eol*, or *eor*.

The syntactic types *graphic*, *alpha*, and *digit* are defined here informally in order tc save space.

graphic ::= any of the class of 95 ASCII graphics, including SP (space), represented by ␣ or SP.

alpha ::= any of the class of 52 upper and lower case letters: A-Z, a-z.

digit ::= any of the class of 10 digits: 0-9.

3.2 Expression Atom expratom

The expression, expr, is the syntactic element which denotes the execution of a value-producing calculation; it is defined in 3.3. The expression atom, expratom, is the basic value-denoting object of which expressions are built; it is defined here.

```
                        |       lvn        |
                        |       gvn        |
                        |       svn        |
     expratom   ::=     |     function     |
                        |      numlit      |
                        |      strlit      |
                        |     ( expr )     |
                        | unaryop expratom |

                        | ' |       (Note:  apostrophe)
     unaryop    ::=     | + |
                        | - |       (Note:  hyphen)
```

3.2.1 Name name

```
     name   ::=   |   %   |  [ digit ]  ...
                  | alpha |  [ alpha ]
```

3.2.2 Local Variable Name lvn

```
                  | name [( L expr )] |
     lvn    ::=   |                   |
                  | @ expratom V lvn  |
```

A local variable name is either unsubscripted or subscripted; if it is subscripted, any number of subscripts separated by commas is permitted. An unsubscripted occurrence of lvn may carry a different value from any subscripted occurrence of lvn.

3.2.3 Global Variable Name gvn

```
                  | ^( L expr )          |
gvn  ::=          | ^name [( L expr )]   |
                  | @ expratom V gvn     |
```

The prefix ^ uniquely denotes a global variable name. A global variable name is either unsubscripted or subscripted; if it is subscripted, any number of subscripts separated by commas is permitted. There is permitted an abbreviated form of subscripted gvn, called the "naked reference", in which the name and an initial (possibly empty) sequence of subscripts is absent but implied by the value of the "naked indicator". An unsubscripted occurrence of gvn may carry a different value from any subscripted occurrence of gvn.

Every executed occurrence of gvn affects the naked indicator as follows. If, for any positive integer m, the gvn has the nonnaked form N(v1, v2, ..., vm), then the m-tuple N, v1, v2, ..., vm-1, is placed into the naked indicator when the gvn reference is made. A subsequent naked reference of the form

^(s1, s2, ..., si)   (i positive)

results in a global reference of the form

N(v1, v2, ..., vm-1, s1, s2, ..., si)

after which the m+i-1-tuple N, v1, v2, ..., si-1 is placed into the naked indicator. Prior to the first executed occurrence of a nonnaked form of gvn, the value of the naked indicator is undefined. It is erroneous for the first executed occurrence of gvn to be a naked reference.

Two types of global references leave the naked indicator undefined.

a. A nonnaked reference without subscripts.
b. A nonnaked reference of the form, or a naked reference resulting in the form

   N(v1, v2, ..., vn)   (n positive)

   for which $D(N(v1, v2, ..., vn-1)) < 10.
   (If n=1, read: $D(N) < 10.)

The effect on the naked indicator described above occurs regardless of the context in which gvn is found; in particular, an assignment of a value to a global variable with the command SET gvn = expr does not affect the value of the naked indicator until after the right-side expr has been evaluated. The effect on the naked indicator of any gvn within the right-side expr will precede the effect on the naked indicator of the left-side gvn.

For convenience, glvn is defined so as to be satisfied by the syntax of either gvn or lvn.

```
glvn  ::=   gvn
            lvn
```

3.2.4 Numeric Literal numlit

The integer literal syntax, intlit, which is a nonempty string of digits, is defined here.

```
intlit  ::=  digit [ digit ] ...
```

The numeric literal numlit is defined as follows.

```
numlit  ::=  mant [ exp ]

  mant  ::=  | intlit [ . intlit ] |
             | . intlit            |
             |                     |
                 ┌   ┐
   exp  ::=  E   | + |  intlit
                 | - |
                 └   ┘
```

The value of the string denoted by an occurrence of numlit is defined in the following two subsections.

3.2.4.1 Numeric Data Values

All variables, local, global, and special, have values which are either defined or undefined. If defined, the values may always be thought of and operated upon as strings. The set of numeric values is a subset of the set of all data values.

Only numbers which may be represented with a finite number of decimal digits are representable as numeric values. A data value has the form of a number if it satisfies the following restrictions.

a. It may contain only digits and the characters "-" and ".".
b. At least one digit must be present.
c. "." occurs at most once.
d. The number zero is represented by the one-character string "0".
e. The representation of each positive number contains no "-".
f. The representation of each negative number contains the character "-" followed by the representation of the positive number which is the absolute value of the negative number. (Thus, the following restrictions describe positive numbers only.)
g. The representation of each positive integer contains only digits and no leading zero.
h. The representation of each positive number less than 1 consists of a "." followed by a nonempty digit string with no trailing zero. (This is called a "fraction".)
i. The representation of each positive noninteger greater than 1 consists of the representation of a positive integer (called the "integer part" of the number) followed by a fraction (called the "fraction part" of the number).

Note that the mapping between representable numbers and representations is one-to-one. An important result of this is that string equality of numeric values is a necessary and sufficient condition of numeric equality.

3.2.4.2 Meaning of numlit

Note that numlit denotes only nonnegative values. The process of converting the spelling of an occurrence of numlit into its numeric data value consists of the following steps.

a. If the mant has no ".", place one at its right end.
b. If the exp is absent, skip step c.
c. If the exp has a plus or has no sign, move the "." a number of decimal digit positions to the right in the mant equal to the value of the intlit of exp, appending zeros to the right of the mant as necessary. If the exp has a minus sign, move the "." a number of decimal digit positions to the left in the mant equal to the value of the intlit of exp, appending zeros to the left of the mant as necessary.
d. Delete the exp and any leading or trailing zeros of the mant.
e. If the rightmost character is ".", remove it.
f. If the result is empty, make it "0".

3.2.5 Numeric Interpretation of Data

Certain operations, such as arithmetic, deal with the numeric interpretations of their operands. The numeric interpretation is a mapping from the set of all data values onto the set of all numeric values, described by the following algorithm. Note that the numeric interpretation maps numeric values onto themselves.

> (Note: The "head" of a string is defined to be a substring which contains all of the characters of the string to the left of a given point and none of the characters of the string to the right of that point. A head may be empty or it may be the entire string.)

Consider the argument to be the string S.

First, apply the following sign reduction rules to S as many times as possible, in any order.

a. If S is of the form + T, then remove the +.
(Shorthand: + T → T)
b. - + T → - T
c. - - T → T

Second, apply one of the following, as appropriate.

a. If the leftmost character of S is not "-", form the longest head of S which satisfies the syntax description of numlit. Then apply the algorithm of 3.2.4.2 to the result.
b. If S is of the form - T, apply step a. above to T and append a "-" to the left of the result. If the result is "-0", change it to "0".

The "numeric expression" numexpr is defined to have the same syntax as expr. Its presence in a syntax description serves to indicate that the numeric interpretation of its value is to be taken when it is executed.

numexpr ::= expr

3.2.5.1 Integer Interpretation

Certain functions deal with the integer interpretations of their arguments. The integer interpretation is a mapping from the set of all data values onto the set of all integer values, described by the following algorithm.

First, take the numeric interpretation of the argument. Then remove the fraction, if present. If the result is empty or "-", change it to "0".

The "integer expression" *intexpr* is defined to have the same syntax as *expr*. Its presence in a syntax definition serves to indicate that the integer interpretation of its value is to be taken when it is executed.

intexpr ::= *expr*

3.2.5.2 Truth-Value Interpretation

The truth-value interpretation is a mapping from the set of all data values onto the two integer values 0 and 1, described by the following algorithm. Take the numeric interpretation. If the result is not "0", make it "1".

The "truth-value expression" *tvexpr* is defined to have the same syntax as *expr*. Its presence in a syntax definition serves to indicate that the truth-value interpretation of its value is to be taken when it is executed.

tvexpr ::= *expr*

3.2.6 String Literal *strlit*

Let *nonquote* temporarily be defined as any of the class of 94 graphics, excluding the quote symbol.

$$\underline{\text{strlit}} \;::=\; \text{"} \left[\begin{array}{c}\text{""}\\ \underline{\text{nonquote}}\end{array}\right] \;\ldots\; \text{"}$$

In words, a string literal is bounded by quotes and contains any string of graphics, except that when quotes occur inside, they occur in adjacent pairs. Each such adjacent quote pair denotes a single quote in the value denoted by *strlit*, whereas any other graphic between the bounding quotes denotes itself. An empty string is denoted by exactly two quotes.

3.2.7 Special Variable Name *svn*

Special variables are denoted by the prefix $ followed by one of a designated list of *name*s. Any of the following defined special variables satisfies the definition of *svn*.

Syntax Definition

$H[OROLOG] $H gives date and time with one access. Its value is D,S where D is an integer value counting days since an origin specified below, and S is an integer value modulo 86,400 counting seconds. The value of $H for the first second of December 31, 1840 after midnight is defined to be 0,0. S increases by 1 each second and S clears to 0 with a carry into D on the tick of midnight.

$I[O] $I identifies the current I/O device. See 3.6.2 and 3.6.16.

$J[OB] Each executing MUMPS process has its own job number, a positive integer which is the value of $J. The job number of each process is unique to that process within a domain of concurrent processes defined by the implementor. $J is constant throughout the active life of a process.

$S[TORAGE] Each implementation must return for the value of $S an integer which is the number of characters of free space available for use. The method of arriving at the value of $S is not part of the standard.

$T[EST] $T contains the truth value computed from the execution of the most recent IF command containing an argument, or an OPEN, LOCK, or READ with a timeout.

$X $X has a nonnegative integer value which approximates the value of a carriage or horizontal cursor position on the current line as if the current I/O device were an ASCII terminal. It is initialized to zero by input or output of control functions corresponding to CR or FF; input or output of each graphic adds 1 to $X. See 3.5.5 and 3.6.16.

$Y $Y has a nonnegative integer value which approximates the line number on the current I/O device as if it were an ASCII terminal. It is initialized to zero by input or output of control functions corresponding to FF; input or output of control functions corresponding to LF adds 1 to $Y. See 3.5.5 and 3.6.16.

$Z[unspecified] Z is the initial letter reserved for defining nonstandard special variables. The requirement that $Z be used permits the unused initial letters to be reserved for future enhancement of the standard without altering the execution of existing programs which observe the rules of the standard.

3.2.8 Functions function

Functions are denoted by the prefix $ followed by one of a designated list of names, followed by a parenthesized argument list. Any of the following specifications satisfies the definition of function.

$A[SCII](expr)

produces an integer value as follows:

a. -1 if the value of expr is the empty string.
b. Otherwise, the decimal equivalent of the ASCII code of the leftmost character of the value of expr.

$A[SCII](expr1 , intexpr2)

is similar to $A(expr1) except that it works with the intexpr2th character of expr1 instead of the first. Formally, $A(expr1,intexpr2) is defined to be $A($E(expr1,intexpr2)).

$C[HAR](L intexpr)

returns a string whose length is the number of argument expressions which have integer values in the closed interval [0,127]. Each intexpr in that interval maps into the ASCII character whose code is the value of intexpr; this mapping is order-preserving. Each negative-valued intexpr maps into no character in the value of $C. Any intexpr whose value is greater than 127 is erroneous.

$D[ATA](glvn)

returns a nonnegative integer which is a characterization of the variable named. The value of the integer is p+d, where:

d = 1 if the named variable has a defined value;
d = 0 otherwise;
p = 10 if either:

a. The named variable exists and contains no subscripts, and there exists (or did exist and was killed) a subscripted variable with the same name, or
b. The named variable exists and contains n subscripts, and there exists (or did exist and was killed) a subscripted variable with m > n subscripts whose first n subscript values are the same as the values of those in the named variable;

p = 0 otherwise.

$E[XTRACT](expr1 , intexpr2)

returns the intexpr2th character of the value of expr1. That is, let m be the value of intexpr2. If m is less than 1 or greater than $L(expr1), the value of $E is the empty string. Otherwise, the value of $E is the mth character of the value of expr1. (1 corresponds to the leftmost character; $L(expr1) corresponds to the rightmost character.)

$E[XTRACT](expr1 , intexpr2 , intexpr3)

returns the string between positions intexpr2 and intexpr3 of the value of expr1. Let m be the value of intexpr2 and let n be the value of intexpr3. The following cases are defined:

a. m > n. Then the value of $E is the empty string.
b. m = n. $E(expr1,m,n) = $E(expr1,m).
c. m < n ≤ $L(expr1). $E(expr1,m,n) = $E(expr1,m) concatenated with $E(expr1,m+1,n).
d. m < n and $L(expr1) < n. $E(expr1,m,n) = $E(expr1,m,$L(expr1)).

$F[IND](expr1 , expr2)

searches for the leftmost occurrence of the value of expr2 in the value of expr1. If none is found, $F returns zero. If one is found, the value returned is the integer representing the number of the character position immediately to the right of the rightmost character of the found occurrence of expr2 in expr1. In particular, if the value of expr2 is empty, $F returns 1.

$F[IND](expr1 , expr2 , intexpr3)

Let m be the value of intexpr3. $F begins the search at the mth position of expr1. If no instance of expr2 is found, $F returns zero; otherwise, $F(A,B,m) = $F($E(A,m,$L(A)),B) + m - 1.

$J[USTIFY] (expr1 , intexpr2)

returns the value of expr1 right-justified in a field of intexpr2 spaces. Let m be $L(expr1) and n be the value of intexpr2. The following cases are defined:

a. m ≥ n. Then the value returned is expr1.
b. Otherwise, the value returned is S(n-m) concatenated with expr1, where S(x) is a string of x spaces.

$J[USTIFY] (numexpr1 , intexpr2 , intexpr3)

returns an edited form of the number numexpr1. Let R be the value of numexpr1 after rounding to intexpr3 fraction digits, including possible trailing zeros. (If intexpr3 is zero, R contains no decimal point.) The value returned is $J(R,intexpr2). Negative values of intexpr3 are reserved for future extensions of the $JUSTIFY function.

$L[ENGTH](expr)

returns an integer which is the number of characters in the value of expr. $L of the empty string is zero.

$N[EXT](glvn)

returns an integer which is a subscript value. Only subscripted forms of lvn and gvn are permitted. Let lvn or gvn be of the form Name(s1, s2, ..., sn) where sn has an integer value ≥ -1. If it exists, the integer value t is returned, where:

a. t is greater than sn, and
b. t is the lowest number such that
$D(Name(s1, s2, ..., sn-1, t) is not zero.

If such a t does not exist, -1 is returned. It is an error if sn has a noninteger value or an integer value less than -1.

$P[IECE](expr1 , expr2 , intexpr3 [, intexpr4])

is defined here with the aid of a function, NF, which is used only for definitional purposes called "find the position number following the mth occurrence".

NF(S, D, m) is defined, for strings S, D, and integer m, as follows:

When m ≤ 0, the result is zero.

When D is not a substring of S, i.e., when $F(S, D) = 0, then the result is $L(S) + $L(D) + 1.

Otherwise, NF(S, D, 1) = $F(S, D).

For m > 1,

NF(S, D, m) = NF($E(S, $F(S, D), $L(S)), D, m-1) + $F(S, D) - 1.

That is, NF generalizes $F to give the position number of the character to the right of the mth occurrence of the string D in S.

$P[IECE](expr1 , expr2 , intexpr3)

Let expr1, expr2 be the strings S, D. Let intexpr3 be the integer m. $P(S, D, m) returns the substring of S bounded by but not including the m - 1th and the mth occurrences of D.

$P(S, D, m) = $E(S, NF(S, D, m-1), NF(S, D, m) - $L(D) - 1).

$P[IECE](expr1 , expr2 , intexpr3 , intexpr4)

Let intexpr4 be the integer n. $P(S, D, m, n) returns the substring of S bounded on the left but not including the m - 1th occurrence of D in S, and bounded on the right but not including the nth occurrence of D in S.

$P(S, D, m, n) = $E(S, NF(S, D, m-1), NF(S, D, n) - $L(D) - 1).

Note that $P(S, D, m, m) = $P(S, D, m).

$R[ANDOM](intexpr)

returns a random or pseudo-random integer uniformly distributed in the closed interval [0, intexpr - 1]. If the value of intexpr is less than 1, an error will occur.

$S[ELECT](L |tvexpr:expr|)

returns the value of the leftmost expr whose corresponding tvexpr is true. The process of evaluation consists of evaluating the tvexprs, one at a time in left-to-right order, until the first one is found whose value is true. The expr corresponding to this tvexpr (and no other) is evaluated and this value is made the value of $S. An error will occur if all tvexprs are false. Since only one expr is evaluated at any invocation of $S, that is the only expr which must have a defined value.

```
         |+ intexpr|
$T[EXT]( |         | )
         | lineref |
```

returns a string whose value is the content of the line of this routine specified by the argument. Specifically, the entire line, with ls replaced by one SP and eol deleted, is returned.

If the argument of $T is a lineref, the line denoted by the lineref is specified. If the argument is + intexpr, the intexprth line of the routine is specified. The first line is numbered 1; an error will occur if the value of intexpr is less than 1.

If no such line as that specified by the argument exists, an empty string is returned. If the line specification is ambiguous, the results are not defined.

$V[IEW](unspecified)

makes available to the implementor a call for examining machine-dependent information. It is to be understood that programs containing occurrences of $V may not be portable.

$Z[unspecified](unspecified)

is the name reserved for defining escapes to nonstandard functions. This requirement permits the unused function names to be reserved for future use.

3.2.9 Unary Operator unaryop

There are three unary operators: ' (not), + (plus), and - (minus).

Not inverts the truth value of the expratom immediately to its right. The value of 'expratom is 1 if the truth-value interpretation of expratom is 0; otherwise its value is 0. Note that '' performs the truth-value interpretation.

Plus is merely an explicit means of taking a numeric interpretation. The value of + expratom is the numeric interpretation of the value of expratom.

Minus negates the numeric interpretation of expratom. The value of - expratom is the numeric interpretation of -N, where N is the value of expratom.

Note that the order of application of unary operators is right-to-left.

3.3 Expressions expr

Expressions are made up of expression atoms separated by binary string, arithmetic, or truth-valued operators.

```
         expr   ::=  expratom [ exprtail ] ...

                    || binaryop   |   expratom  |
     exprtail   ::= ||['] truthop |             |
                    |                           |
                    |      ['] ? pattern        |

                    | _ |       (Note:  underscore)
                    | + |
                    | - |       (Note:  hyphen)
     binaryop   ::= | * |
                    | / |
                    | \ |
                    | # |

      truthop   ::= | relation  |
                    | logicalop |

                    | = |
                    | < |
     relation   ::= | > |
                    | [ |
                    | ] |

    logicalop   ::= | & |
                    | ! |
```

The order of evaluation is as follows:

a. Evaluate the left-hand expratom.
b. If an exprtail is present immediately to the right, evaluate its expratom or pattern and apply its operator.
c. Repeat step b. as necessary, moving to the right.

In the language of operator precedence, this sequence implies that all binary string, arithmetic, and truth-valued operators are at the same precedence level and are applied in left-to-right order.

Any attempt to evaluate an expratom containing an lvn, gvn, or svn with an undefined value is erroneous.

3.3.1 Arithmetic Binary Operators

The binary operators + - * / \ # are called the arithmetic binary operators. They operate on the numeric interpretations of their operands, and they produce numeric (in one case, integer) results.

+ produces the algebraic sum.

- produces the algebraic difference.

* produces the algebraic product.

/ produces the algebraic quotient. Note that the sign of the quotient is negative if and only if one argument is positive and one argument is negative. Division by zero is erroneous.

\ produces the integer interpretation of the result of the above division.

# produces the value of the left argument modulo the right argument. It is defined only for nonzero values of its right argument, as follows.
A # B = A - (B * floor(A/B))
where floor(x) = the largest integer ≤ x.

3.3.2 Relational Operators

The operators = < >] [produce the truth value 1 if the relation between their arguments which they express is true, and 0 otherwise. The dual operators 'relation are defined by:

A 'relation B has the same value as '(A relation B)

3.3.2.1 Numeric Relations

The inequalities > and < operate on the numeric interpretations of their operands; they denote the conventional algebraic "greater than" and "less than".

3.3.2.2 String Relations

The relations =] [do not imply any numeric interpretation of either of their operands.

The relation = tests string identity. If the operands are not known to be numeric and numeric equality is to be tested, the programmer may apply an appropriate unary operator to the nonnumeric operands. If both arguments are known to be in numeric form (as would be the case, for example, if they resulted from the application of any operator except _), application of a unary operator is not necessary. The uniqueness of the numeric representation guarantees the equivalence of string and numeric equality when both operands are numeric. Note, however, that the division operator / may produce inexact results, with the usual problems attendant to inexact arithmetic.

The relation [is called "contains". A [B is true if and only if B is a substring of A; that is, A [B has the same value as ''$F(A,B). Note that the empty string is a substring of every string.

The relation] is called "follows". A] B is true if and only if A follows B in the conventional ASCII collating sequence, defined here. A follows B if and only if any of the following is true.

a. B is empty and A is not.
b. Neither A nor B is empty, and the leftmost character of A follows (i.e., has a numerically greater ASCII code than) the leftmost character of B.
c. There exists a positive integer n such that A and B have identical heads of length n, (i.e., $E(A,1,n) = $E(B,1,n)) and the remainder of A follows the remainder of B (i.e., $E(A,n+1,$L(A)) follows $E(B,n+1,$L(B))).

3.3.3 Pattern Match

The pattern match operator ? tests the form of the string which is its left-hand operand. S ? P is true if and only if S is a member of the class of strings specified by the pattern P.

A pattern is a concatenated list of pattern atoms.

```
               | patatom [ patatom ] ... |
pattern  ::=   |                         |
               | @ expratom V pattern    |
```

Assume that pattern has n patatoms. S ? pattern is true if and only if there exists a partition of S into n substrings

S = S1 S2 ... Sn

such that there is a one-to-one order-preserving correspondence between the Si and the pattern atoms, and each Si "satisfies" its respective pattern atom. Note that some of the Si may be empty.

Each pattern atom consists of a pattern code patcode or a string literal strlit, preceded either by an integer literal intlit multiplier or by the indefinite multiplier ".".

```
patatom  ::=   intlit   strlit
                 .      patcode
```

```
                 C
                 N
                 P
patcode  ::=     A    ...
                 L
                 U
                 E
```

Each patcode is satisfied by any single character in the union of the classes of characters represented, each class denoted by its own patcode letter, as follows.

C	33 Control characters, including DEL
N	10 Numeric characters
P	33 Punctuation characters, including SP
A	52 Alphabetic characters
L	26 Lower-case alphabetic characters
U	26 Upper-case alphabetic characters
E	Everything (the Entire set of characters)

The strlit is satisfied by, and only by, the value of strlit.

If the indefinite multiplier "." is present, patatom is satisfied by a concatenation of any number of strings (including none), each of which satisfies the patcode or strlit following the multiplier.

If the intlit multiplier is present, patatom is satisfied by a concatenation of exactly intlit strings, each of which satisfies the patcode or strlit following the multiplier. In particular, if the value of intlit is zero, the corresponding Si is empty.

The dual operator '? is defined by:

A '? B = '(A ? B)

3.3.4 Logical Operators

The operators ! and & are called logical operators. (They are given the names "or" and "and", respectively.) They operate on the truth-value interpretations of their arguments, and they produce truth-value results.

$$A \; ! \; B = \begin{cases} 0 & \text{if both A and B have the value 0} \\ 1 & \text{otherwise} \end{cases}$$

$$A \; \& \; B = \begin{cases} 1 & \text{if both A and B have the value 1} \\ 0 & \text{otherwise} \end{cases}$$

The dual operators '& and '! are defined by:

```
A '& B = '(A & B)
A '! B = '(A ! B)
```

3.3.5 Concatenation Operator

The underscore symbol _ is the concatenation operator. It does not imply any numeric interpretation. The value of A _ B is the string obtained by concatenating the values of A and B, with A on the left.

3.4 Routines

The routine is the unit of program interchange. In program interchange, each routine begins with its routinehead, which contains the identifying routinename, and the routinehead is followed by the routinebody, which contains the executed code. The routinehead is not part of the executed code.

```
        routine  ::=  routinehead  routinebody

    routinehead  ::=  routinename  eol

    routinename  ::=  name
```

The routinebody is a sequence of lines terminated by the eor. Each line ends with eol, starts with ls optionally preceded by a label, and may contain zero or more commands (separated by single spaces) between ls and eol. Any line may end with a comment immediately preceding the eol.

```
    routinebody  ::=  line  [ line ] ... eor

                    ┌ command [ ␣ command ] ... [ ␣ comment ] ┐
 line  ::=  [ label ] ls  │                                         │  eol
                    └               comment                   ┘

          label  ::=  | name   |
                      | intlit |

             ls  ::=  SP       (one space)

            eol  ::=  CR LF    (two control characters)

            eor  ::=  CR FF    (two control characters)
```

Each occurrence of a label to the left of ls in a line is called a "defining occurrence" of label. No two defining occurrences of label may have the same spelling in one routinebody.

3.5 General command Rules

Every command starts with a "command word" which dictates the syntax and interpretation of that command instance. The standard contains the following command words.

```
B[REAK]
C[LOSE]
D[O]
E[LSE]
F[OR]
G[OTO]
H[ALT]
H[ANG]
I[F]
K[ILL]
L[OCK]
O[PEN]
Q[UIT]
R[EAD]
S[ET]
U[SE]
V[IEW]
W[RITE]
X[ECUTE]
Z[unspecified]
```

Unused initial letters of command words are reserved for future enhancement of the standard.

The formal definition of the syntax of command is a choice from among all of the individual command syntax definitions of 3.6.

```
                 | syntax of BREAK command   |
                 | syntax of CLOSE command   |
                 |               .           |
command  ::=     |               .           |
                 |               .           |
                 | syntax of XECUTE command  |
```

Any implementation of the language must be able to recognize both the initial letter abbreviation and the full spelling of each command word. When two command words have a common initial letter, their argument syntaxes uniquely distinguish them.

For all commands allowing multiple arguments, the form

```
command word  arg1, arg2 ...
```

is equivalent in execution to

```
command word  arg1  command word  arg2  ...    .
```

3.5.1 Post Conditionals

All commands except ELSE, FOR, and IF may be made conditional as a whole by following the command word immediately by the post-conditional postcond.

```
postcond  ::=  [ : tvexpr ]
```

If the tvexpr is either absent or present and true, the command is executed. If the tvexpr is present and false, the command word and its arguments are passed over without execution.

The postcond may also be used to conditionalize the arguments of DO, GOTO, and XECUTE.

3.5.2 Spaces in Commands

Spaces are significant characters. The following rules apply to their use in lines.

a. There may be a SP immediately preceding eol only if the line ends with a comment. (Since ls may immediately precede eol, this rule does not apply to the SP which may stand for ls.)

b. If a command instance contains at least one argument, the command word or postcond is followed by exactly one space; if the command is not the last of the line, or if a comment follows, the command is followed by exactly one space.

c. If a command instance contains no argument and it is not the last command of the line, or if a comment follows, the command word or postcond is followed by exactly two spaces; if it is the last command of the line and no comment follows, the command word or postcond is immediately followed by eol.

3.5.3 Comments

If a semicolon appears in the command word initial-letter position, it is the start of a comment. The remainder of the line to eol must consist of graphics only, but is otherwise ignored and nonfunctional.

```
comment  ::=  ; [ graphic ] ...
```

3.5.4 format in READ and WRITE

The format, which can appear in READ and WRITE commands, specifies output format control. The parameters of format are processed one at a time, in left-to-right order.

```
              | | ! |  [ ! ]   ...  [ ? intexpr ] |
format  ::=   | | # |  [ # ]                      |
              |            ? intexpr              |
```

The parameters, which need not be separated by commas when occurring in a single instance of format, may take the following forms.

! causes a "new line" operation on the current device. Its effect is the equivalent of writing CR LF on a pure ASCII device. In addition, $X is set to 0 and 1 is added to $Y.

# causes a "top of form" operation on the current device. Its effect is the equivalent of writing CR FF on a pure ASCII device. In addition, $X and $Y are set to 0. When the current device is a display, the screen is blanked and the cursor is positioned at the upper left-hand corner.

? intexpr produces an effect similar to "tab to column intexpr". If $X is greater than or equal to intexpr, there is no effect. Otherwise, the effect is the same as writing (intexpr - $X) spaces. (Note that the leftmost column of a line is column 0.)

3.5.5 Side Effects on $X and $Y

As READ and WRITE transfer characters one at a time, certain characters or character combinations represent device control functions, depending on the identity of the current device. To the extent that the supervisory function can detect these control characters or character sequences, they will alter $X and $Y as follows.

graphic:	add 1 to $X
backspace:	set $X = max($X-1,0)
line feed:	add 1 to $Y
carriage return:	set $X = 0
form feed:	set $Y = 0, $X = 0

3.5.6 Timeout

The OPEN, LOCK, and READ commands employ an optional timeout specification, associated with the testing of an external condition.

```
timeout  ::=   : numexpr
```

If the optional timeout is absent, the command will proceed if the condition, associated with the definition of the command, is satisfied; otherwise, it will wait until the condition is satisfied and then proceed. $T will not be altered if the timeout is absent.

If the optional timeout is present, the value of numexpr must be nonnegative. If it is negative, the value 0 is used. numexpr denotes a t-second timeout, where t is the value of numexpr.

> If t = 0, the condition is tested. If it is true, $T is set to 1; otherwise, $T is set to 0. Execution proceeds without delay.
>
> If t is positive, execution is suspended until the condition is true, but in any case no longer than t seconds. If at the time of resumption of execution the condition is true, $T is set to 1; otherwise, $T is set to 0.

3.5.7 Line References

The DO and GOTO commands, as well as the $TEXT function, contain in their arguments means for referring to particular lines within any routine (in the case of DO and GOTO) or within the routine executing the line reference (in the case of $TEXT). This section describes the means for making line references.

Any line in a given routine may be denoted by mention of a label which occurs in a defining occurrence on or prior to the line in question.

```
lineref  ::=  dlabel [ + intexpr ]

             |  label                |
dlabel   ::= |                       |
             | @ expratom V dlabel   |
```

If + intexpr is absent, the line denoted by lineref is the one containing label in a defining occurrence. If + intexpr is present and has the value $n \geq 0$, the line denoted is the nth line after the one containing label in a defining occurrence. A negative value of intexpr is erroneous. When label is an instance of intlit, leading zeros are significant to its spelling.

In the context of DO or GOTO, either of the following conditions is erroneous.

a. A value of intexpr so large as not to denote a line within the bounds of the given routine.
b. A spelling of label which does not occur in a defining occurrence in the given routine.

In any context, reference to a particular spelling of label which occurs more than once in a defining occurrence in the given routine will have undefined results.

DO and GOTO can refer to a line in a routine other than that in which they occur; this requires a means of specifying a routine name.

```
                         routinename
    routineref  ::=
                        @ expratom V routineref
```

The total line specification in DO and GOTO is in the form of an entryref.

```
                        lineref [ ^ routineref ]
    entryref  ::=
                                ^ routineref
```

If the delimiter ^ is absent, the routine being executed is implied. If the lineref is absent, the first line is implied.

3.5.8 Command Argument Indirection

Indirection is available for evaluation of either individual command arguments or contiguous sublists of command arguments. The opportunities for indirection are shown in the syntax definitions accompanying the command descriptions.

Typically, where a command word carries an argument list, as in

```
        COMMANDWORD ␣ L argument    ,
```

the argument syntax will be expressed as

```
                     |  individual argument syntax  |
    argument  ::=    |                              |
                     | @ expratom V L argument      |
```

This formulation expresses the following properties of argument indirection.

a. Argument indirection may be used recursively.
b. A single instance of argument indirection may evaluate to one complete argument or to a sublist of complete arguments.

Unless the opposite is.explicitly stated, the text of each command specification describes the arguments after all indirection has been evaluated.

3.6 Command Definitions

The specifications of all commands follow.

3.6.1 BREAK

```
B[REAK] postcond  | [ ␣ ]                            |
                  | argument syntax unspecified      |
```

BREAK provides an access point within the standard for nonstandard programming aids. BREAK without arguments suspends execution until receipt of a signal, not specified here, from a device.

3.6.2 CLOSE

```
C[LOSE] postcond ␣ L closeargument

                       | expr [ : deviceparameters ]       |
  closeargument   ::=  |                                   |
                       | @ expratom V L closeargument      |

                       | expr                              |
deviceparameters  ::=  |                                   |
                       | ( expr [ : [ expr ] ] ... )       |
```

The value of the first expr of each closeargument identifies a device (or "file" or "data set"). The interpretation of the value of this expr is left to the implementor. The deviceparameters may be used to specify termination procedures or other information associated with relinquishing ownership, in accordance with implementor interpretation.

Each designated device is released from ownership. If a device is not owned at the time that it is named in an argument of an executed CLOSE, the command has no effect upon the ownership and the values of the associated parameters of that device. Device parameters in effect at the time of the execution of CLOSE are retained for possible future use in connection with the device to which they apply. If the current device is named in an argument of an executed CLOSE, the implementor may choose to execute implicitly OPEN P USE P, where P designates a predetermined default device. If the implementor chooses otherwise, $IO is given the empty value.

3.6.3 DO

D[O] postcond ␣ L doargument

```
                 | entryref postcond           |
doargument  ::=  |                             |
                 | @ expratom V L doargument   |
```

DO is a generalized subroutine call. Each doargument is executed, one at a time in left-to-right order. Execution of a doargument is described below.

a. If postcond is present and false, execution of the doargument is complete at this point. If postcond is absent, or present and true, proceed to the following step.

b. Before proceeding to the next argument of this DO or to the command following this DO, execution continues at the left end of the line specified by the entryref. Execution returns to the argument or command following this argument upon encountering an executed QUIT or eor not within the scope of a subsequently executed doargument or FOR. The scope of this doargument extends to the execution of that QUIT or eor.

3.6.4 ELSE

E[LSE] [␣]

If the value of $T is 1, the remainder of the line to the right of the ELSE is not executed. If the value of $T is 0, execution continues normally at the next command.

3.6.5 FOR

F[OR] ␣ lvn = L forparameter

```
                              | exprl                                 |
    forparameter   ::=        | numexprl  :  numexpr2  :  numexpr3    |
                              | numexprl  :  numexpr2                 |
```

The "scope" of this FOR command begins at the next command following this FOR on the same line and ends just prior to the eol on this line.

FOR specifies repeated execution of its scope for different values of the local variable lvn, under successive control of the forparameters, from left to right. Any expressions occurring in lvn, such as might occur in subscripts or indirection, are evaluated once per execution of the FOR, prior to the first execution of any forparameter.

For each forparameter, control of the execution of the scope is specified as follows. (Note that A, B, and C are hidden temporaries.)

a. If the forparameter is of the form exprl.

   1. Set lvn = exprl.
   2. Execute the scope once.
   3. Processing of this forparameter is complete.

b. If the forparameter is of the form
   numexprl : numexpr2 : numexpr3
   and numexpr2 is nonnegative.

   1. Set A = numexprl.
   2. Set B = numexpr2.
   3. Set C = numexpr3.
   4. Set lvn = A.
   5. If lvn > C, processing of this forparameter is complete.
   6. Execute the scope once.
   7. If lvn > C-B, processing of this forparameter is complete.
   8. Otherwise, set lvn = lvn + B.
   9. Go to 6.

c. If the forparameter is of the form
   numexprl : numexpr2 : numexpr3
   and numexpr2 is negative.

1. Set A = numexpr1.
2. Set B = numexpr2.
3. Set C = numexpr3.
4. Set lvn = A.
5. If lvn < C, processing of this forparameter is complete.
6. Execute the scope once.
7. If lvn < C-B, processing of this forparameter is complete.
8. Otherwise, set lvn = lvn + B.
9. Go to 6.

d. If the forparameter is of the form numexpr1 : numexpr2.

1. Set A = numexpr1.
2. Set B = numexpr2.
3. Set lvn = A.
4. Execute the scope once.
5. Set lvn = lvn + B.
6. Go to 4.

Note that form d. specifies an endless loop. Termination of this loop must occur by execution of a QUIT or GOTO within the scope of the FOR. These two termination methods are available within the scope of a FOR independent of the form of forparameter currently in control of the execution of the scope; they are described below. Note also that no forparameter to the right of one of form d. can be executed.

Note that if the scope of a FOR (the "outer" FOR) contains an "inner" FOR, one execution of the scope of the outer FOR encompasses all executions of the scope of the inner FOR corresponding to one complete pass through the inner FOR's forparameter list.

Execution of a QUIT within the scope of a FOR has two effects.

a. It terminates that particular execution of the scope at the QUIT; commands to the right of the QUIT are not executed.

b. It causes any remaining values of the forparameter in control at the time of execution of the QUIT, and the remainder of the forparameters in the same forparameter list, not to be calculated and the scope not to be executed under their control.

In other words, execution of QUIT effects the immediate termination of the innermost FOR whose scope contains the QUIT.

Execution of GOTO effects the immediate termination of all FORs in the line containing the GOTO, and it transfers execution control to the point specified.

3.6.6 GOTO

```
G[OTO] postcond ␣ L gotoargument

   gotoargument  ::=  doargument
```

GOTO is a generalized transfer of control. If provision for a return of control is desired, DO may be used.

Each gotoargument is examined, one at a time in left-to-right order, until the first one is found whose postcond is either absent, or present and true. If no such gotoargument is found, control is not transferred and execution continues normally. If such a gotoargument is found, execution continues at the start of the line it specifies.

See 3.6.5 for a discussion of additional effects of GOTO when executed within the scope of FOR.

3.6.7 HALT

```
H[ALT] postcond [ ␣ ]
```

First, LOCK with no arguments is executed. Then, execution of this process is terminated.

3.6.8 HANG

```
H[ANG] postcond ␣ L hangargument

                        | intexpr                         |
   hangargument  ::=    |                                 |
                        | @ expratom V L hangargument     |
```

Let t be the value of intexpr. If t ≤ 0, HANG has no effect. Otherwise, execution is suspended for t seconds.

3.6.9 IF

```
I[F]  | [ ␣ ]            |
      | ␣ L ifargument   |

                       | tvexpr                       |
     ifargument  ::=   |                              |
                       | @ expratom V L ifargument    |
```

In its argumentless form, IF is the inverse of ELSE. That is, if the value of $T is 0, the remainder of the line to the right of the IF is not executed. If the value of $T is 1, execution continues normally at the next command.

If exactly one argument is present, the value of tvexpr is placed into $T; then the function described above is performed.

IF with n arguments is equivalent in execution to n IFs, each with one argument, with the respective arguments in the same order. This may be thought of as an implied "and" of the conditions expressed by the arguments.

3.6.10 KILL

```
                          [ _ ]
K[ILL] postcond
                         _ L killargument

                            glvn

  killargument  ::=         ( L lvn )

                            @ expratom V L killargument
```

The three argument forms of KILL are given the following names.

a. Empty argument list: Kill All.
b. glvn: Selective Kill.
c. (L lvn): Exclusive Kill.

Killing the variable M sets $D(M) = 0 and causes the value of M to be undefined. Any attempt to obtain the value of M while it is undefined is erroneous. The value of M remains undefined until M appears to the left of the delimiter = in an executed SET command. $D(M) remains 0 until M or a descendant of M appears to the left of the delimiter = in an executed SET command. Killing a variable whose $D = 0 has no effect.

To kill a variable with the unsubscripted name N also kills all subscripted variables with the same name N.

To kill an m-tuply subscripted variable N(v1, v2, ..., vm) with name N and subscript values v1, v2, ..., vm also kills all n-tuply subscripted variables N(v1, v2, ..., vm, vm+1, ..., vn), for all n > m, with the same N and identical values for the first m subscripts. (These derived n-tuply subscripted variables are called the "descendants" of the m-tuply subscripted variable.)

The Kill All form kills all local variables.

The Selective Kill form kills the variables named.

In the Exclusive Kill form lvn must not contain subscripts, although lvn may have descendants. Exclusive Kill kills all local variables except those named and their descendants.

If M is not killed but N, a descendant of M, is killed, the killing does not effect a change to the value of $D(M).

3.6.11 LOCK

```
                    | [ ␣ ]              |
LOCK postcond       |                    |
                    | ␣ L lockargument   |

                    | |    nref     |                  |
                    | |             |   [ timeout ]    |
    lockargument ::=| ( L nref )    |                  |
                    |                                  |
                    | @ expratom V L lockargument      |

                    | [^] name [( L expr )] |
            nref ::=|                       |
                    | @ expratom V nref     |
```

LOCK provides a generalized interlock facility available to concurrently executing MUMPS processes to be used as appropriate to the applications being programmed. Execution of LOCK is not affected by, nor does it directly affect, the state or value of any global or local variable, or the value of the naked indicator.

Each lockargument specifies a subspace of the total MUMPS name space for which the executing process seeks to make an exclusive claim; the details of this subspace specification are given below. Prior to evaluating and executing each lockargument, LOCK first unconditionally removes any prior claim on any portion of the name space made by the process as the result of a prior execution of LOCK. Then, if a lockargument is present, an attempt is made to claim the entire subspace defined by the lockargument. If this subspace does not intersect the union of all other subspaces claimed at this instant by all other processes defined by the implementor as sharing the interlock facility, the claim is successfully established and execution proceeds to the next lockargument or command. If the subspace defined by the lockargument intersects any other claimed subspace, execution of this process is suspended until all interfering claims are removed by one or more other processes, or, when a timeout is present, until the timeout expires, if that occurs first.

The subspace defined by one lockargument is claimed effectively all at once or not at all; thus, the observance of appropriate conventions on the use of the name space by all concurrently executing processes can eliminate the possibility of races and deadlocks.

If a timeout is present, the condition reported by $T upon resumption of execution is the successful establishment of the claim. If no timeout is present, execution of the lockargument does not change $T.

The subspace of the total name space defined by each lockargument is the union of the subspaces defined by each of the name references nref in the lockargument. Each nref specifies its subspace as follows.

a. If the occurrence of nref is unsubscripted, then the subspace is the set of the following points: one point for the unsubscripted variable name nref and one point for each subscripted variable name N(s1, ..., si) for which N has the same spelling as nref.

b. If the occurrence of nref is subscripted, then the subspace is the set of the following points: one point for the spelling of nref after all subscripts have been evaluated and one point for each descendant of nref. (See KILL for a definition of descendant.)

3.6.12 OPEN

O[PEN] postcond ␣ L openargument

```
                         | expr [ : openparameters ]        |
   openargument   ::=    |                                  |
                         | @ expratom V L openargument      |

                         | deviceparameters  [ timeout ]    |
  openparameters  ::=    |                                  |
                         | timeout                          |
```

The value of the first expr of each openargument identifies a device (or "file" or "data set"). The interpretation of the value of this expr or of any exprs in deviceparameters is left to the implementor. (See 3.6.2 for the syntax specification of deviceparameters.)

The OPEN command is used to obtain ownership of a device, and does not affect which device is the current device or the value of $IO. (See the discussion of USE in 3.6.16.)

For each openargument, the OPEN command attempts to seize exclusive ownership of the specified device. OPEN performs this function effectively instantaneously as far as other processes are concerned; otherwise, it has no effect regarding the ownership of devices and the values of the device parameters. If a timeout is present, the condition reported by $T is the success of obtaining ownership. If no timeout is present, the value of $T is not changed and process execution is suspended until seizure of ownership has been successfully accomplished.

Ownership is relinquished by execution of the CLOSE command. When ownership is relinquished, all device parameters are retained. Upon establishing ownership of a device, any parameter for which no specification is present in the openparameters is given the value most recently used for that device; if none exists, an implementor-defined default value is used.

3.6.13 QUIT

Q[UIT] postcond [␣]

The end-of-routine mark eor is equivalent to an unconditional QUIT. If the last command of the routine is executed in such a manner as not to transfer control, or if the last command of the routine is an executed DO and control is returned, then the effect of executing off the end of the routine is to execute the QUIT which is implied by the eor.

The effect of executing QUIT in the scope of FOR is fully discussed in 3.6.5. Note that eor never occurs in the scope of FOR.

If an executed QUIT is not in the scope of FOR, then it is in the scope of some doargument or xargument, if not explicitly then implicitly, because the initial activation of a process may be thought of as arising from execution of a DO naming the first executed routine of that process. The effect of executing a QUIT in the scope of a doargument or xargument is to return control to the most recently executed doargument or xargument to which control has not yet been returned by a QUIT. What is executed immediately following the QUIT is the command, doargument, or xargument immediately following the doargument or xargument which most recently transferred control and to which control has not yet been returned. Thus, executed doarguments and xarguments are added to a list of pending returns from which execution of a QUIT (not in the scope of FOR) removes entries in last-in, first-out order.

3.6.14 READ

```
R[EAD] postcond ␣ L readargument

                                strlit
                                format
   readargument  ::=            lvn [timeout]
                              * lvn [timeout]
                              @ expratom V L readargument
```

The readarguments are executed, one at a time, in left-to-right order.

The top two argument forms cause output operations to the current device; the next two cause input from the current device to the named local variable. If no timeout is present, execution will be suspended until the input message is explicitly terminated. (See 3.6.16 for a definition of "current device".)

If a timeout is present, it is interpreted as a t-second timeout, and execution will be suspended until the input message is explicitly terminated, but in any case no longer than t seconds. If $t \leq 0$, $t = 0$ is used.

When a timeout is present, $T is affected as follows. If the input message has been explicitly terminated at or before the time at which execution resumes, $T is set to 1; otherwise, $T is set to 0.

When the form of the argument is *lvn [timeout], the input message is by definition one character long, and it is explicitly terminated by the entry of one character, which is not necessarily from the ASCII set. The value given to lvn is an integer; the mapping between the set of input characters and the set of integer values given to lvn may be defined by the implementor in a device-dependent manner. If timeout is present and the timeout expires, lvn is given the value -1.

When the form of the argument is lvn [timeout], the input message is a string of arbitrary length which is terminated by an implementor-defined procedure, which may be device-dependent. If timeout is present and the timeout expires, the value given to lvn is the string entered prior to expiration of the timeout; otherwise, the value given to lvn is the entire string.

When the form of the argument is strlit, that literal is output to the current device, provided that it accepts output.

When the form of the argument is format, the output actions defined in 3.5.4 are executed.

$X and $Y are affected by READ the same as if the command were WRITE with the same argument list (except for timeouts) and with each expr value in each writeargument equal, in turn, to the final value of the respective lvn resulting from the READ.

3.6.15 SET

S[ET] postcond _ L setargument

```
                           glvn
                                            = expr
    setargument   ::=    ( L glvn )

                          @ expratom V L setargument
```

SET is the general means for explicitly assigning values to variables. Each setargument assigns one value, defined by its expr, to each of one or more variables, each named by one glvn.

Each setargument is executed one at a time in left-to-right order. The execution of one setargument occurs in the following order.

a. The glvns to the left of the = are scanned in left-to-right order and all subscripts are evaluated, in left-to-right order within each glvn.

b. The expr to the right of the = is evaluated.

c. The value of expr is given to each glvn, in left-to-right order. For each subscripted glvn of the form N(v1, v2, ..., vn), each variable M whose name is of the form N(v1, v2, ..., vm) for all m < n, as well as the unsubscripted variable N, will be affected as follows.

   1. If M already has a "pointer", that is, if $D(M) has a value of 10 or 11, no change is made to the value of $D(M).

   2. If M has no pointer, that is, if $D(M) has a value of 0 or 1, then it is given a pointer. If $D(M) was 0 it becomes 10, and if $D(M) was 1 it becomes 11.

   The $D value of the glvn itself is changed as follows:

0	becomes	1
1	remains	1
10	becomes	11
11	remains	11.

   That is, the pointer status is not altered, but the variable's value becomes defined. If the glvn is a global variable, the naked indicator is set at the time that the glvn is given its value. If the glvn is a naked reference, the reference to the naked indicator to determine the name and initial subscript sequence occurs just prior to the time that the glvn is given its value.

3.6.16 USE

U[SE] postcond _ L useargument

```
                          | expr [ : deviceparameters ] |
    useargument  ::=      |                             |
                          | @ expratom V L useargument  |
```

The value of the first expr of each useargument identifies a device (or "file" or "data set"). The interpretation of the value of this expr or of any exprs in deviceparameters is left to the implementor. (See 3.6.2 for the syntax specification of deviceparameters.)

Before a device can be employed in conjunction with an input or output data transfer it must be designated, through execution of a USE command, as the "current device". Before a device can be named in an executed useargument, its ownership must have been established through execution of an OPEN command.

The specified device remains current until such time as a new USE command is executed. As a side effect of employing expr to designate a current device, $IO is given the value of expr.

Specification of device parameters, by means of the exprs in deviceparameters, is normally associated with the process of obtaining ownership; however, it is possible, by execution of a USE command, to change the parameters of a device previously obtained.

Distinct values for $X and $Y are retained for each device. The special variables $X and $Y reflect those values for the current device. When the identity of the current device is changed as a result of the execution of a USE command, the values of $X and $Y are saved, and the values associated with the new current device are then the values of $X and $Y.

3.6.17 VIEW

V[IEW] arguments unspecified

VIEW makes available to the implementor a mechanism for examining machine-dependent information. It is to be understood that routines containing the VIEW command may not be portable.

3.6.18 WRITE

W[RITE] postcond ␣ L writeargument

```
                         |   format                      |
  writeargument  ::=     |   expr                        |
                         | * intexpr                     |
                         | @ expratom V L writeargument  |
```

The writearguments are executed, one at a time, in left-to-right order. Each form of argument defines an output operation to the current device.

When the form of argument is format, the output actions defined in 3.5.4 are executed. Each character of output, in turn, affects $X and $Y as described in 3.5.4 and 3.5.5.

When the form of argument is expr, the value of expr is sent to the device. The effect of this string at the device is defined by the ASCII standard and conventions. Each character of output, in turn, affects $X and $Y as described in 3.5.5.

When the form of the argument is *intexpr, one character, not necessarily from the ASCII set and whose code is the number represented in decimal by the value of intexpr, is sent to the device. The effect of this character at the device may be defined by the implementor in a device-dependent manner.

3.6.19 XECUTE

```
X[ECUTE] postcond ␣ L xargument

                          | expr postcond                |
     xargument   ::=      |                              |
                          | @ expratom V L xargument     |
```

XECUTE provides a means of executing MUMPS code which arises from the process of expression evaluation.

Each xargument is executed one at a time in left-to-right order. If the postcond in the xargument is false, the xargument has no effect. Otherwise, if the value of expr is x, execution of the xargument is equivalent to execution of DO y, where y is the spelling of an otherwise unused label attached to the following two-line subroutine considered to be a part of the currently executing routine.

```
y   ls   x    eol

    ls  QUIT  eol
```

3.6.20 Z

Z[unspecified] arguments unspecified

All command words in a given implementation which are not defined in the standard are to begin with the letter Z. This convention protects the standard for future enhancement.

MUMPS LANGUAGE STANDARD

Part II: MUMPS Transition Diagrams

Table of Contents

Section Diagram Page

1. Introduction to MUMPS Transition Diagrams

This document presents the MUMPS dynamic syntax in transition diagram form. This type of representation was introduced[1] for computer programming language definition by Melvin E. Conway and was applied to MUMPS in an early specification.[2]

The diagrams serve as a comprehensive implementation outline for the MUMPS language. It is possible to implement the MUMPS language directly from these diagrams,[3] although specific implementation techniques are not stipulated in semantic actions of the diagrams. The necessary operations are given here, but their detailed implementation is left to individual implementors.

One departure from the MUMPS Language Specification is made in the transition diagrams. Syntax checking is performed after false postconditionals and after the IF and ELSE commands. This was done in order to simplify the presentation of the diagrams.

1.1 Scanning Algorithm

A transition diagram is a network of nodes and directed paths with at least one entrance node (indicated by the symbol Ⓔ or the symbol ▽n , where n is an integer such as 1, 2, 3, etc.), and at least one exit node (indicated by one of the following four symbols Ⓦ , Ⓧ , Ⓨ , Ⓩ ; Ⓧ is the "normal" exit).

Each transition diagram defines a syntactic type, whose name is written in underscored lower-case letters, e.g., expr. A string being scanned through a "window" which looks at one character at a time is declared to be an instance of the syntactic type defined by a diagram if and only if the algorithm given below exactly scans the string while traversing the transition diagram from an entrance node to an exit node.

Each directed path from one node to another node is either associated with a symbol "on" that path, or the path is "blank". If a path has a symbol on it, the symbol can be either the name of a syntactic type defined by a transition diagram, implying a "call" (possibly recursive) to that diagram, or a symbol from the primitive alphabet. The primitive alphabet consists of the 95 ASCII graphics, including SP (space), plus ls, eol, eor, and eoi (the eoi character is used to indicate the end of argument or sub-argument level indirection; its actual form is not defined).

[1] Conway, M. E., "Design of a Separable Transition-Diagram Compiler," Communications of the Association for Computing Machinery, 6:7 (July 1963), pp. 396-408.

[2] Conway, M. E., "Preliminary MUMPS Language Specification," MDC 1/3, Draft Proposal, Revised 7/12/74, MUMPS Development Committee.

[3] Wasserman, A. I. and Sherertz, D. D., "Implementation of the MUMPS Language Standard," MDC 2/3, 6/15/75, MUMPS Development Committee.

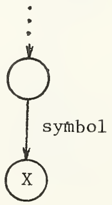
The scanning algorithm works as follows. In the next paragraph, the rule for following a path from one node to the next node is given. Once that can be done, the rule is applied iteratively starting at an entrance node until an exit node is encountered. At this point, a call to the diagram just traversed has been completed, and the path which made the call may be traversed. If a dead end is reached, and the window has not moved since entering the diagram, it only means that the call to this diagram has not succeeded and another path must be attempted. If a dead end is reached after moving the window over at least one input symbol since the entrance node, this is the indication of a syntax error.

The rule for leaving a node is as follows. All nonblank paths are tried, with primitive symbols tried first, then calls to other diagrams. The blank path, if it is present, provides the default case and is taken only after all other paths fail. A path with a primitive symbol on it may be traversed if and only if the symbol in the window equals the symbol on the path. In this case, the path is traversed and the window is moved one position to the right. A path with a call to a transition diagram may be traversed if and only if a call to that diagram results in successfully reaching an exit node of that diagram. Any window movement arising from a call to a diagram will be done within the called diagram.

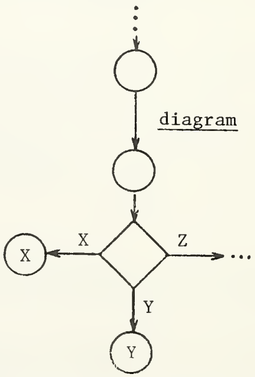
Any path can direct the performance of an action, indicated by a number in a square box on the path. The action may be performed after the path is traversed. Certain actions, called "privileged actions", are always executed after traversing the path on which they appear; all other actions are executed only if both the semantic action flags Linact and Comact are True (see the scanning algorithm). Note that the actions within a called diagram precede the action specified on the path of the call.

If a diagram contains only one exit type, the X type is used. If it contains W, X, Y and/or Z, the exit actually used as a result of a given call may be tested by the caller. The notation used is as follows.

single-exit diagram

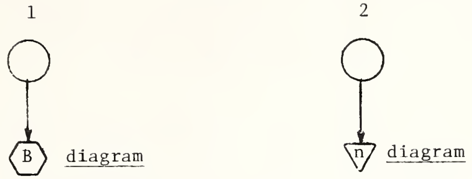


multiple-exit diagram



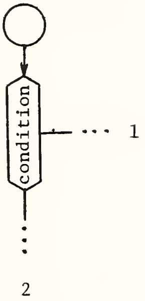
Actions generally make reference to temporary variables, such as A, B, C, D (see the diagram for the $PIECE function). These variables are strictly local to each invocation of a diagram. A communication variable "Result" is used to pass values among diagrams.

Two branching notations are used in the diagrams. These are illustrated below.



Notation 1 indicates that this syntactic construct continues at the Ⓔ entrance of the named diagram. In other words, whenever the ⟨B⟩ (branch) symbol is encountered, the logical flow is transferred to the beginning of the diagram whose name appears to the right of the ⟨B⟩ . This construct is merely a convenient notation for presentation of the diagrams. Notation 2 indicates a transfer of control to a specific entry point (other than the Ⓔ entrance) in the named diagram. The n in the symbol ▽n is an integer which indicates the number of the entry point in the diagram whose name appears to the right of the ▽n . This construct is used primarily to show clearly the control flow in the MUMPS language from DO, FOR, and XECUTE commands.

One other construct used in the diagrams is shown below.



The logical value of "condition" is tested. If it is True, path 1 is taken. If it is False, path 2 is taken.

The scanning algorithm on the next page is used to move from one "syntax node" to the next (a syntax node is denoted by a circle). Any of the intervening branching or testing nodes discussed above are taken automatically. The variable "case" in the algorithm is used for determining the actual exit from a diagram call with multiple exit. Underlined parts of the algorithm represent operations which have not undergone detailed stepwise refinement.

To execute a MUMPS routine, the window is initially positioned to the first character of the text of the routine, and the <u>routine</u> diagram in Section 2 is invoked. The algorithm below then scans through the routine text using this diagram and all subsequently called diagrams.

TRANSITION DIAGRAM SCANNING ALGORITHM (PASCAL-like)

```
TYPE alpha = ARRAY[1..16] OF char;
VAR linact, comact : Boolean; case : char;
FUNCTION traversediagram(diagramname : alpha) : integer;
TYPE exitnode = (X,Y,Z,W);
VAR oneprimitivetraversed, continuescanning, pathtaken, pathexists : Boolean;
    exittype : integer;
BEGIN oneprimitivetraversed := FALSE; continuescanning := TRUE;
   WHILE continuescanning DO
      BEGIN pathtaken := FALSE; pathexists := TRUE;
         WHILE pathexists DO
            BEGIN
               find next path from this syntax node with a primitive symbol on it;
               IF such a path found THEN
                  BEGIN
                     IF character on the path = character in the window THEN
                        BEGIN exittype := 1; oneprimitivetraversed := TRUE;
                           mark this path as the chosen path to be taken;
                           move window one position to the right;
                           IF input string is exhausted THEN syntax error
                           ELSE BEGIN pathtaken := TRUE; pathexists := FALSE; END
                        END
                  END
               ELSE pathexists := FALSE;
            END;
         IF ¬pathtaken THEN
            BEGIN pathexists := TRUE;
               WHILE pathexists DO
                  BEGIN
                     find next path from this syntax node with a call to "diagramname";
                     IF such a path found THEN
                        BEGIN
                           exittype := traversediagram("diagramname");
                           IF exittype≠∅ THEN
                              BEGIN pathtaken := TRUE; pathexists := FALSE;
                                 mark this path as the chosen path to be taken;
                              END
                        END
                     ELSE pathexists := FALSE;
                  END
            END;
         IF ¬(pathtaken) ∧ there is a blank path from this syntax node THEN
            BEGIN exittype := 1; pathtaken := TRUE;
               mark this path as the chosen path to be taken;
            END;
         IF pathtaken THEN
            BEGIN
               CASE exittype OF
                  1:  case := 'X';
                  2:  case :='Y';
                  3:  case := 'Z';
                  4:  case := 'W';
               END;
               find marked path from this syntax node and move to next syntax node along it;
               IF path traversed specified an action THEN
                  BEGIN
                     IF (linact ∧ comact) ∨ action is a privileged action THEN DO action;
                  END;
               IF next syntax node is an exitnode THEN
                  BEGIN
                     continuescanning := FALSE;
                     CASE exitnode OF
                        X:  traversediagram := 1;
                        Y:  traversediagram := 2;
                        Z:  traversediagram := 3;
                        W:  traversediagram := 4;
                     END
                  END
            END
         ELSE
            BEGIN
               IF oneprimitivetraversed THEN syntax errcr
               ELSE
                  BEGIN traversediagram := ∅; continuescanning := FALSE; END
            END
      END
END;
```

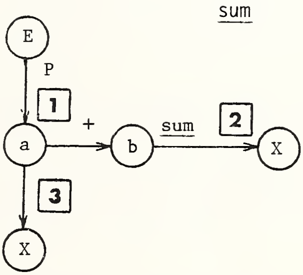
1.2 Example

A simple arithmetic expression, sum, can be defined as follows.

sum ::= P[+ sum]

The primitive symbols are P and +.

This example can be used to illustrate recursion and the technique for scanning text with a transition diagram. The definition above forces a right-to-left order of evaluation. Because this definition is recursive, a "Stack" is needed to save intermediate results whenever sum reinvokes itself. Items are saved onto this Stack by a PUT operation, and are retrieved by a GET operation, in a last-in-first-out order. The transition diagram for the definition of sum above is shown below.



1. Place value of P into A, that is, P → A

2. A + Result → Result

3. A → Result

In order to illustrate the effect of a diagram's structure on the order of evaluation, this example can be tested on the input string P+Q+R eol , where Q and R are separate occurrences of the symbol P, and eol is the string termination character. The steps below interpret this string using the above diagram, in the same fashion as the algorithm on the previous page uses the MUMPS diagrams in Section 2.

1. The window is initially positioned at the first character of the string, P.

2. Path Ea contains the same symbol as in the window (P); consequently, the path to node a is traversed and the window is moved to the next character of the input string.

3. Action 1 is executed. The value of P is placed in A_0. (A_0 is the zero-level occurrence of temporary variable A.)

4. Path ab contains the same symbol as in the window (+), causing transversal of the path to node b, and moving the window to the next character of the input string.

5. Path bX is a call to sum. PUT A_0 on the Stack and start at node E, now at level 1.

6. Path Ea contains the same symbol as in the window (Q≡P), so traverse the path to node a, moving the window to the next input character.

7. Action 1 is executed, placing the value of Q in A_1.

8. Path ab contains the same symbol as in the window (+), so traverse the path to node b, and move the window to the next character.

9. Path bX is again a call to sum. PUT A_1 on the Stack and start at node E, now at level 2.

10. As before, path Ea is traversed and the value of R is placed in A_2.

11. The default blank path aX is now traversed because there is an end-of-line symbol in the window, which is not the same as a +.

12. Action 3 is executed, which places the value in A_2 (i.e., R) into Result.

13. The second call to sum is now complete, so effectively path bX can now be traversed at level 1.

14. Action 2 is executed, by first performing a GET A_1 from the Stack, then forming A_1 + Result (which is Q + R), and finally placing the sum into Result.

15. The first call to sum is now successful, so path bX can now be traversed at level 0.

16. As above, action 2 is executed, doing a GET A_0 from the Stack, forming the sum A_0 + Result (which is P + Q + R), and placing this value into Result.

17. The variable Result now contains the value obtained from scanning the input string P+Q+R eol. This value can in turn be used in another diagram which invoked the sum diagram, much as Result was used in the recursive calls of sum above.

1.3 Common Data Used by the MUMPS Diagrams

The following variables are common to the actions used by all of the MUMPS diagrams of Section 2.

1. Naked indicator. This is the n-tuple described in Section 3.2.3 of MDC/28. It is initially undefined and becomes undefined under certain circumstances discussed in Section 3.2.3.

2. Devicelist. This is an n-tuple of device names with associated device parameters. It is added to whenever a new device is successfully secured with the OPEN command. It is updated when the device parameters of a previously owned device are changed with the OPEN, CLOSE, or USE commands.

3. Openlist. This is an n-tuple of device names which are created by the OPEN command and can be removed with the CLOSE command.

4. Devnam. This is the device name from an argument of the OPEN, CLOSE, or USE command. It is used by deviceparameters to get default values for omitted parameters.

5. Argind. This is the argument-level indirection flag. It is initially False, and is set True if argument-level indirection is encountered. It is saved each time argument-level indirection is detected.

6. Nameind. This is the name-atom (sub-argument) level indirection flag. It is initially False, and is set True if name-atom indirection is encountered. It is saved each time name-atom indirection is detected.

7. Indsw. This is the command-level indirection counter. It is initally zero, is incremented each time command-level indirection is detected, and is decremented when an eor is encountered or a QUIT is executed under command-level indirection.

8. Forsw. This is the FOR command counter. It is initially zero, is incremented each time a FOR command is encountered, and decremented when a FOR has exhausted its FOR list. It is also decremented appropriately by the QUIT and GOTO commands.

9. Dosw. This is the DO command counter. It is initially zero, is incremented each time a DO command is encountered, and decremented when a DO is completed, either from a QUIT or an eor.

10. Ifswitch. This is the name used in the diagrams for the special variable $TEST. It is initially 0 (False).

11. Linact. This has the value True or False. If False, all semantic actions are inhibited for all commands until an eol is detected (see the diagrams for the IF and ELSE commands). Linact is set True prior to executing a new line of a routine.

12. Comact. This has the value True or False. If False, all semantic actions are inhibited for the duration of the command (see the diagram for postcond). Comact is set True prior to executing a new command.

13. Present routine name. This defines the scope of label values. It is saved when a DO or XECUTE command is executed for the return.

14. Window position. This is a pointer which gives the current character position being scanned. It is saved whenever a DO or FOR command is executed, and when any indirection is encountered.

15. Result. This is used to pass values among various diagrams.

16. Timeout. This is used to set up a time limit for the following: (1) timed-length reads in the READ command, (2) securing device ownership in the OPEN command, and (3) interlocking software resources in the LOCK command. It is initially 0.

17. Setsw. This is a flag which is always False unless a SET command is being executed. It is used in the glvn diagram to determine whether or not the Naked indicator is immediately affected by a global reference.

18. Indcom. This is a communications flag used only by indarg and indnam. It is always False unless name level indirection is detected in indarg.

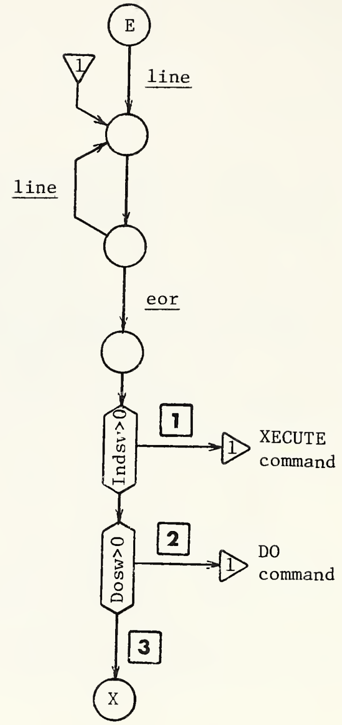
Additionally, a Stack is used in the diagrams to explicitly show the mechanism by which indirection is handled and the transfer of control commands are executed. It is a simple push-down stack operating on a last-pushed, first-pulled basis. Items pushed onto the Stack are listed in the semantic actions following a PUT directive. Items pulled off the Stack are listed in the semantic actions following a GET directive. A RESET directive is used in a few semantic actions to indicate items whose values are recovered from the Stack without actually removing them from the Stack.

2. MUMPS Transition Diagrams

The transition diagrams on the following pages are organized in a "top-down" manner. The first diagram invoked is always the routine diagram; it is therefore the highest level diagram. The routine diagram then invokes the line diagram, which may in turn invoke the command diagram, and so on.

Thus, the logical flow is reflected in the order of presentation of the diagrams. For ease of reference, the individual command and function diagrams are organized alphabetically. A number of command primitive diagrams appear after the command diagrams. These are also organized in a top-down logical manner, as much as possible. The expression diagrams and expression atom diagrams are similarly arranged.

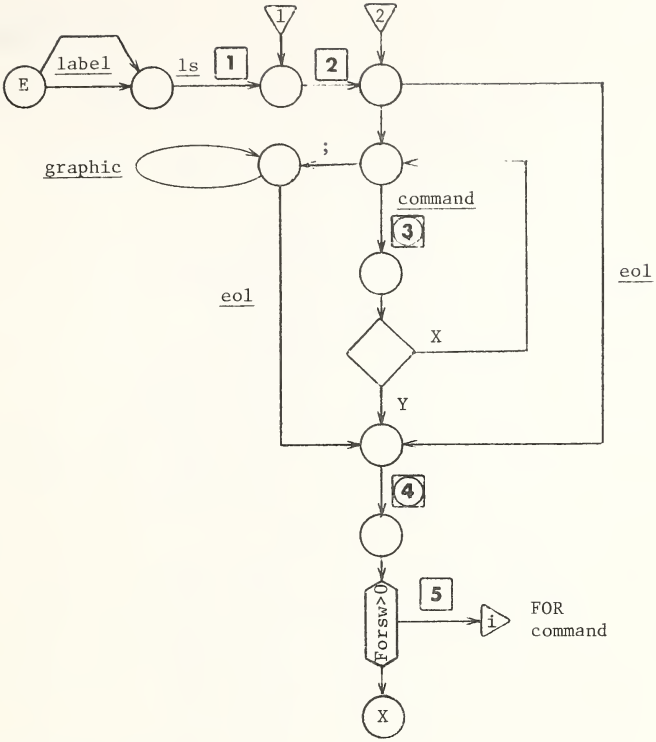
routine



1. GET Argind, Forsw, Window position, and Present routine name from Stack. GET the indirect string off the Stack. Load routine if necessary. Link window to retrieved Window position

2. GET Argind, Indsw, Forsw, Window position, and Present routine name from Stack. Load routine if necessary. Link window to retrieved Window position

3. Terminate execution



1. Set Linact = True

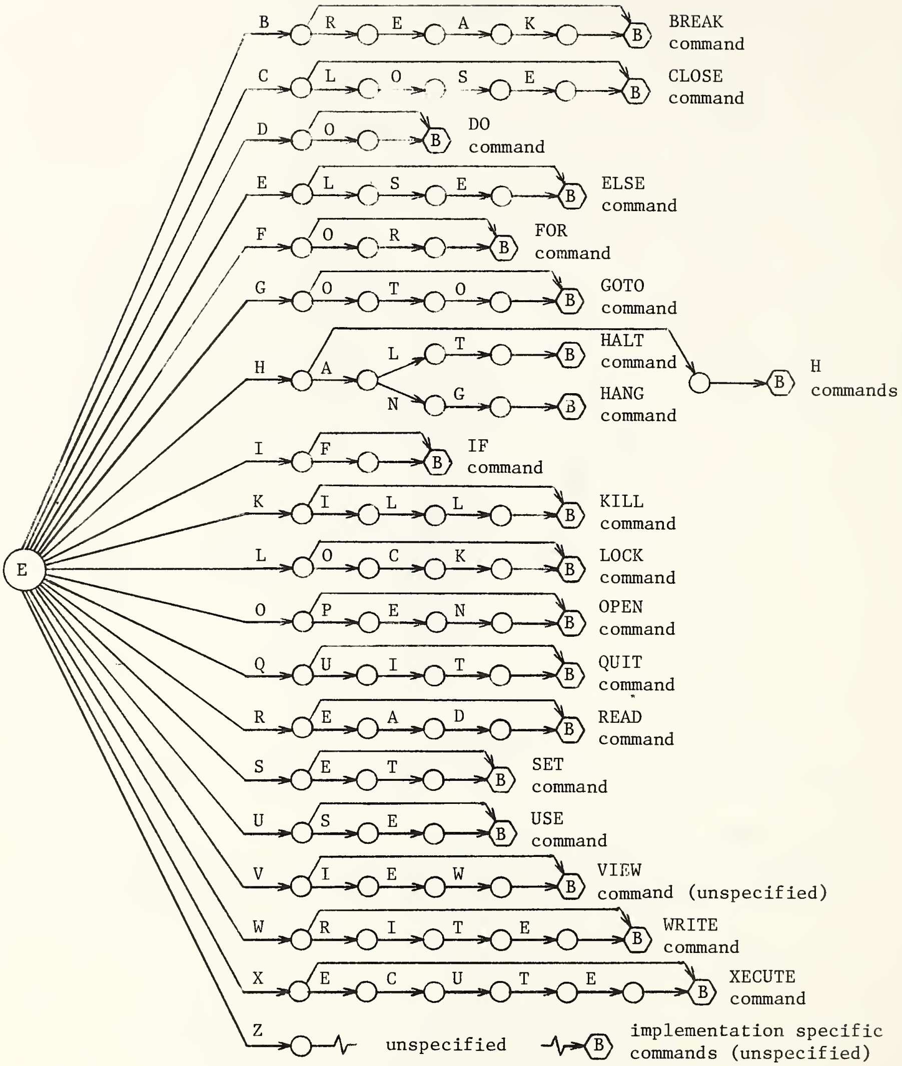
2. Set Forsw = 0, Comact = True, Argind = False

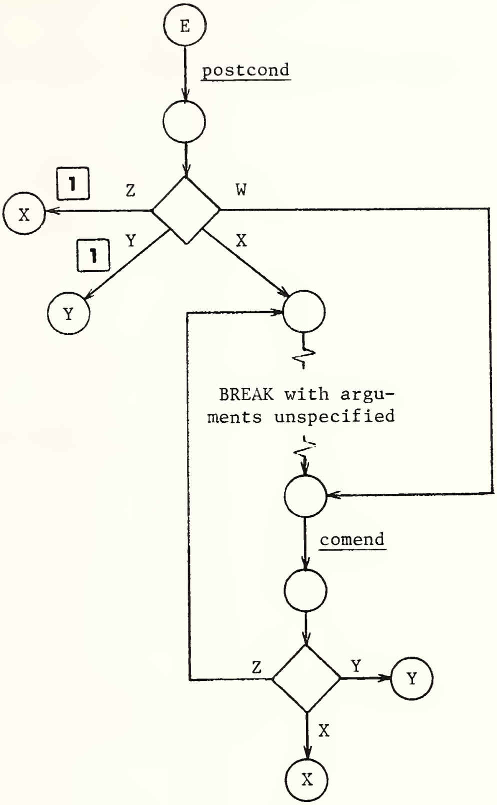
3.[*] Set Comact = True

4.[*] Set Linact = True

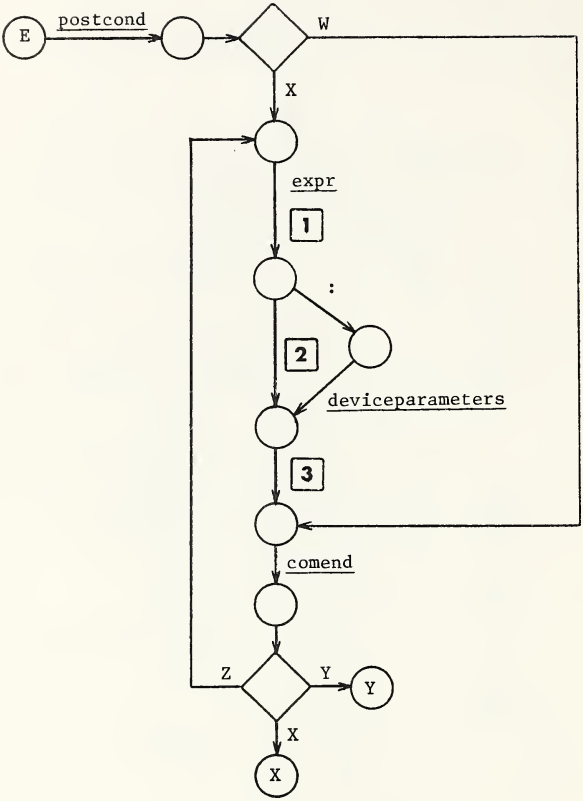
5. RESET FOR lvn name, FOR body position, loop counters (if any), and FOR window position from Stack. The value of entry i going to the FOR command is the number of loop counters + 1. Link window to FOR window position

[*] Privileged action, always executed


E
B
R
E
A
K
B
BREAK command
C
L
O
S
E
B
CLOSE command
D
O
B
DO command
E
L
S
E
B
ELSE command
F
O
R
B
FOR command
G
O
T
O
B
GOTO command
H
A
L
T
B
HALT command
N
G
B
HANG command
B
H commands
I
F
B
IF command
K
I
L
L
B
KILL command
L
O
C
K
B
LOCK command
O
P
E
N
B
OPEN command
Q
U
I
T
B
QUIT command
R
E
A
D
B
READ command
S
E
T
B
SET command
U
S
E
B
USE command
V
I
E
W
B
VIEW command (unspecified)
W
R
I
T
E
B
WRITE command
X
E
C
U
T
E
B
XECUTE command
Z
unspecified
B
implementation specific commands (unspecified)



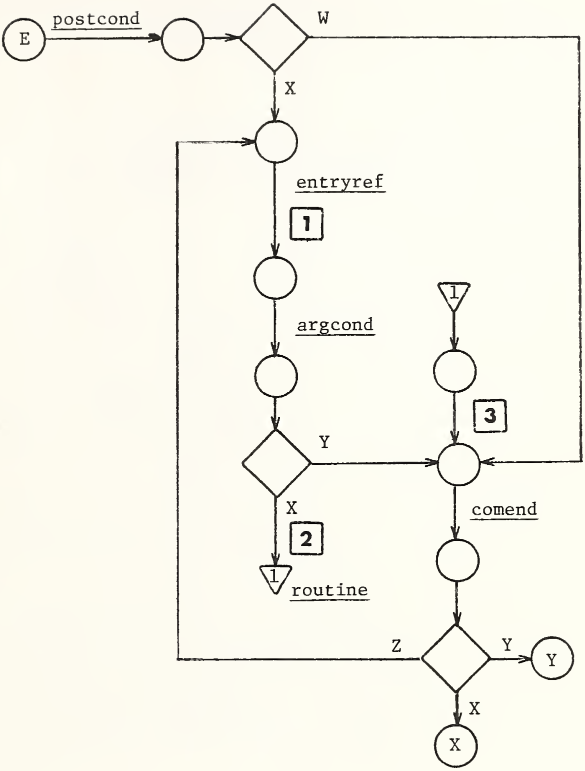
1. Suspend operation until receipt of the proceed-from-break signal



1. Result → Devnam

2. Null string ("") → Result

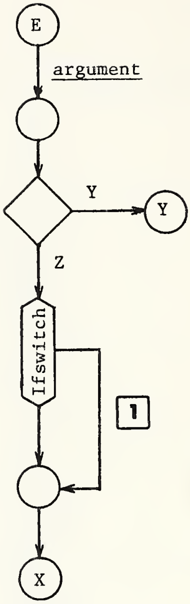
3. Search the Openlist for the device named in Devnam. If not found, take no further action. Otherwise, perform the following operations:
   a. Remove the device specified in Devnam from the Openlist
   b. If Result contains any device parameters, find the device named in Devnam in the Devicelist, and change those parameters which appear in Result to their new values from Result
   c. Perform any termination procedures for the device named in Devnam according to its device parameters in Devicelist
   d. If the named device is the current device ($IO), execute an OPEN P USE P where P designates a predetermined default device



1. Result → A

2. Dosw + 1 → Dosw. PUT Present routine name, Window position, Forsw, Indsw, and Argind on Stack. Load routine named in A if necessary. Set Indsw = 0. Position window to the first character of the entry reference named in A

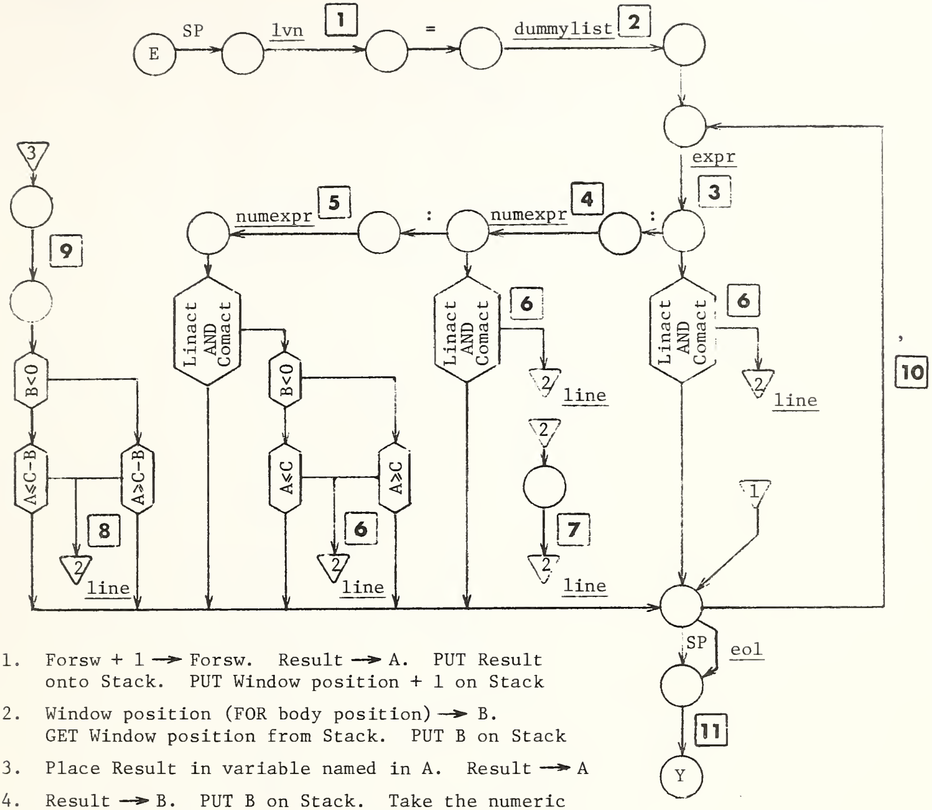
3. Dosw - 1 → Dosw



1. Set semantic action flag Linact False

   Note: If Ifswitch ($TEST) is True (1), then Linact will be set False, which will inhibit execution of all semantic actions for all <u>commands</u> until a new <u>line</u> is encountered. However, all syntactic paths for all commands are traversed (that is, syntax checking is performed).

FOR
command



1. Forsw + 1 → Forsw. Result → A. PUT Result onto Stack. PUT Window position + 1 on Stack

2. Window position (FOR body position) → B. GET Window position from Stack. PUT B on Stack

3. Place Result in variable named in A. Result → A

4. Result → B. PUT B on Stack. Take the numeric interpretation of the value in A. That is, apply the rules given in Section 3.2.5 of MDC/28 to A

5. Result → C. PUT C on Stack

6. Current Window position → D. RESET FOR body position from Stack and link window to it. PUT D on Stack

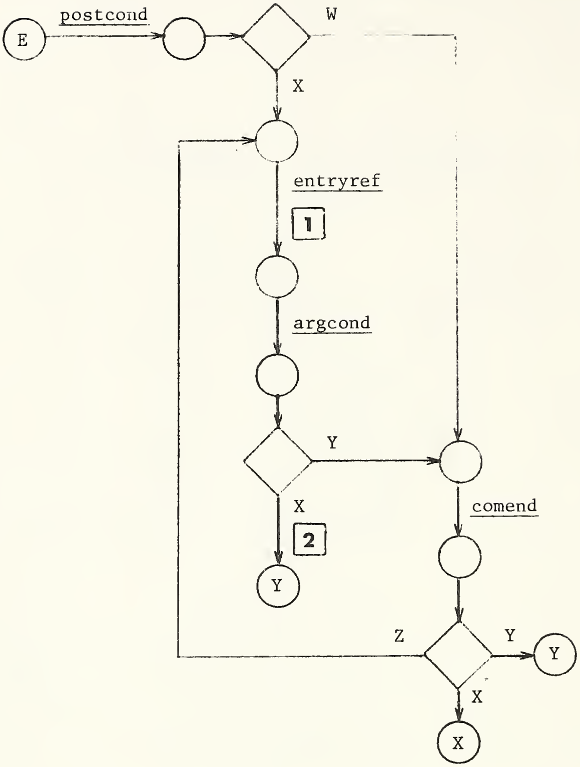
7. Place value of retrieved FOR <u>lvn</u> name into A, value of 1st retrieved loop counter into B. A + B → A. Place the result in A in the FOR variable. Link window to FOR body position

8. A + B → A. Place the result in A in the FOR variable. Link window to FOR body position

9. Place value of retrieved FOR <u>lvn</u> name into A, value of 1st retrieved loop counter into B, value of 2nd retrieved loop counter into C

10. GET FOR window position and loop counters (if any) from Stack

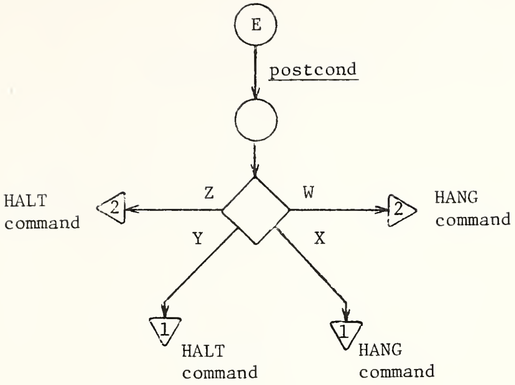
11. Do action 10. GET FOR body position and FOR <u>lvn</u> name from Stack. Forsw - 1 → Forsw

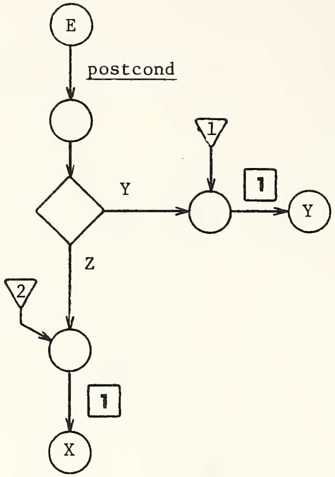


1. Result ⟶ A

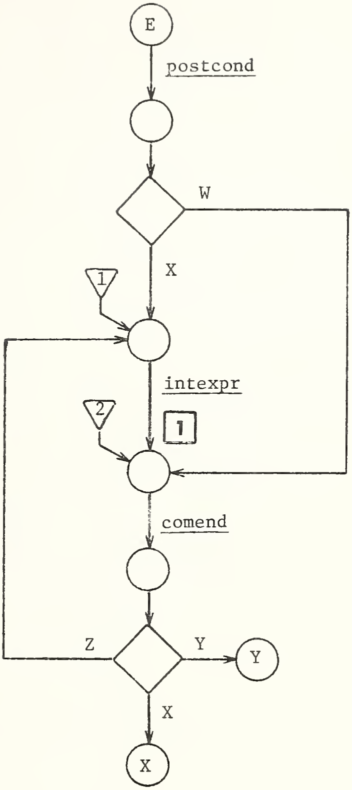
2. Perform the following operations in order:
   a. Examine Argind. If Argind = True, GET previous level's Argind and Window position from the Stack. GET the indirect argument off the Stack. Repeat until Argind = False
   b. Examine Forsw. If Forsw > 0, GET that FOR information off the Stack. Forsw - 1 ⟶ Forsw. Repeat until Forsw = 0

   Load routine named in A if necessary. Then position window to the first character of the entry reference named in A

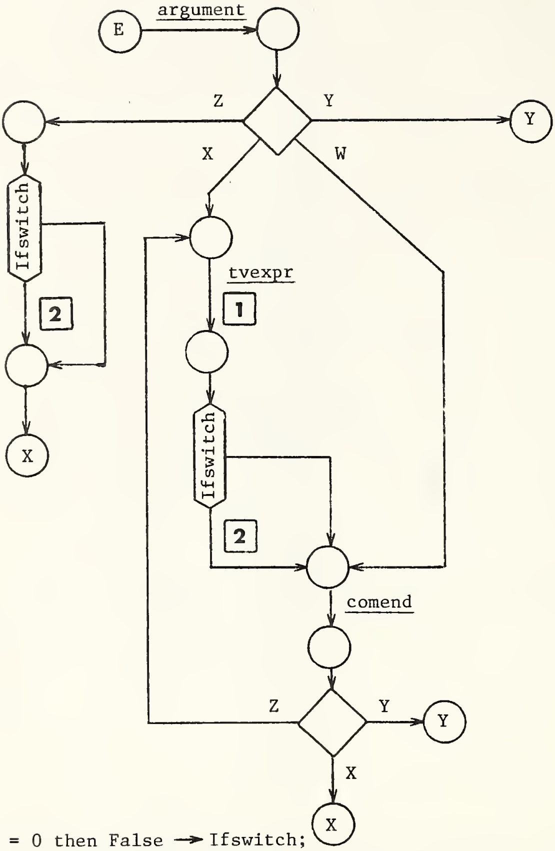

E
postcond
Z
W
Y
X
2
HALT command
2
HANG command
1
HALT command
1
HANG command



1. Perform a LOCK with no arguments (action 1 of the LOCK command). Then terminate execution



1. If Result > 0, suspend execution for the number of seconds specified by the value in Result

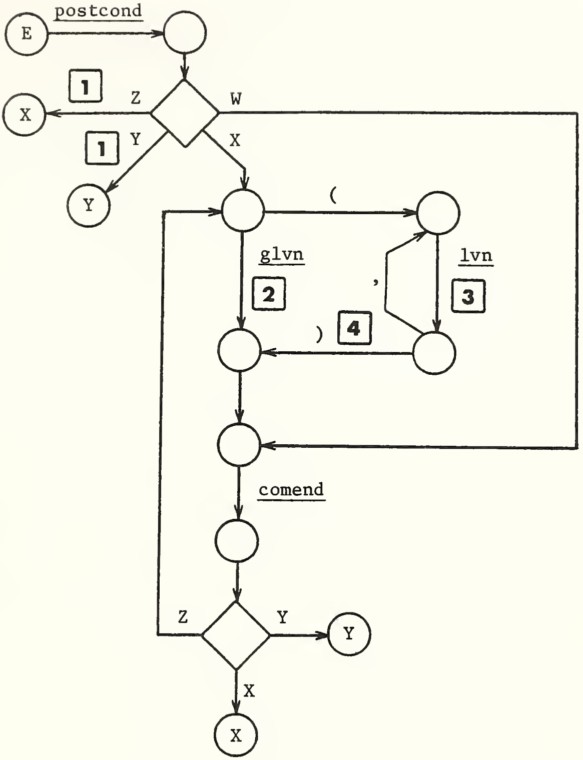


1. If Result = 0 then False → Ifswitch; otherwise True → Ifswitch ($TEST)

2. Set semantic action flag Linact False

Note: If Ifswitch ($TEST) is False (0), then Linact will be set False, which will inhibit execution of all semantic actions for all commands until a new line is encountered. However, all syntactic paths for all commands are traversed (that is, syntax checking is performed).

KILL
command



1. Kill all local variables

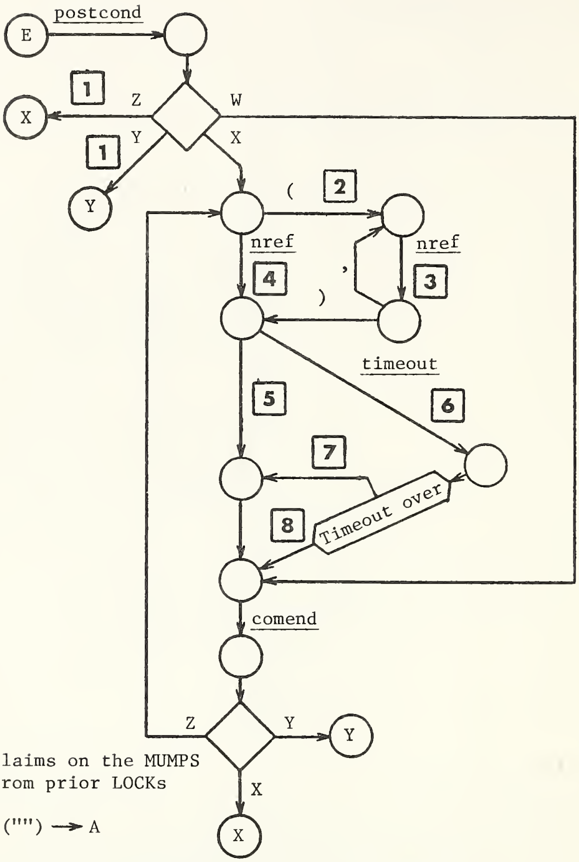
2. Kill the variables whose names contain the name in Result

3. Check to see that Result is at most a 1-tuple (that is, that the local variable is not subscripted). If it is not, trap execution. Otherwise, mark the local variables whose names contain the name in Result

4. Kill all local variables except those marked by action 3. Remove the marks

Note: The n-tuple name (a_1, a_2, ..., a_n) contains the m-tuple name (b_1, b_2, ..., b_m) if and only if $m < n$, and for each i where $i = 1, 2, \ldots, m$, $a_i = b_i$.

LOCK
command



1. Remove all claims on the MUMPS name space from prior LOCKs

2. Null string ("") ⟶ A

3. Replace the n-tuple in A by the n + 1-tuple (A:Result)

4. Result ⟶ A

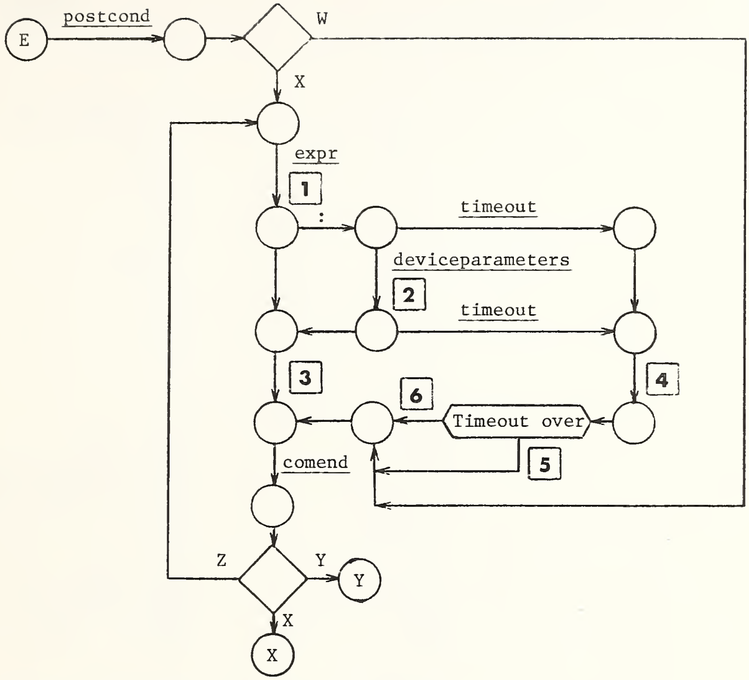
5. Do action 1. Then attempt to claim the subspace of all names in A. This action suspends execution until it succeeds

6. Set up a timer of Timeout seconds. Do action 1. Then attempt to claim the subspace of all names in A at least once, and then repeatedly until the claim succeeds or the timer expires, whichever occurs first

7. False ⟶ Ifswitch ($TEST)

8. True ⟶ Ifswitch ($TEST)

OPEN
command



1. Result ⟶ Devnam, Null string ("") ⟶ B

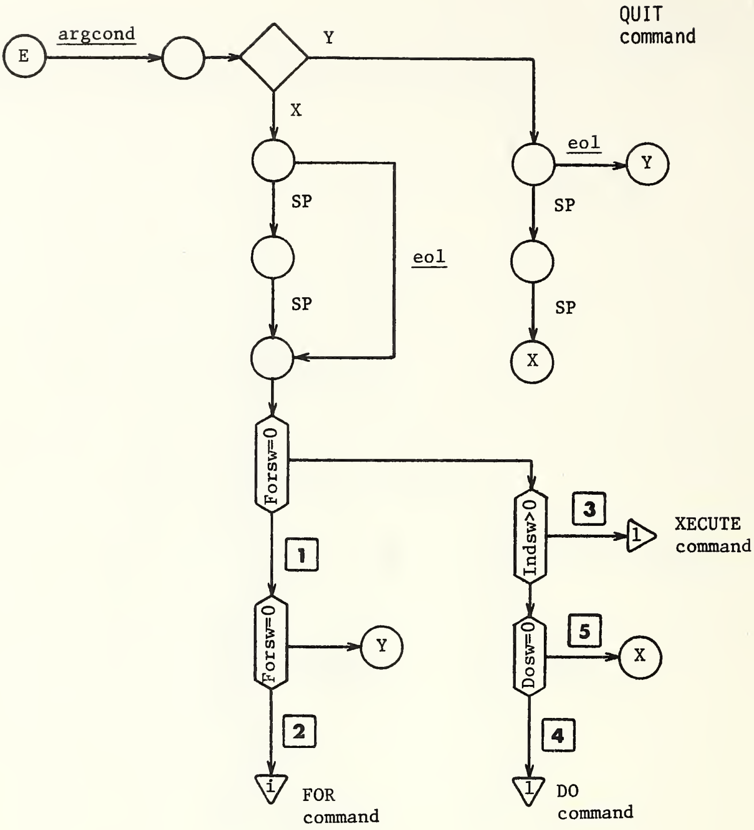
2. Result ⟶ B

3. Search the Openlist for the device named in Devnam. If found, perform only operation c. below and take no further action. Otherwise, perform all the following operations:
   a. Attempt to seize exclusive ownership of the device named in Devnam. This operation suspends execution until it succeeds
   b. Add the specified device to the Openlist
   c. If B contains any device parameters, find the device named in Devnam in the Devicelist, and change those parameters which appear in Result
   d. Perform any initiation procedures for the device named in Devnam according to its device parameters in Devicelist

4. Set up a timer of Timeout seconds. Search the Openlist for the device named in A. If found, perform operation 3c and indicate that the timer has not expired. Otherwise, attempt to seize exclusive ownership of the device named in Devnam at least once, and then repeatedly until it succeeds, or the timer expires, whichever occurs first. If ownership is established prior to expiration of the timer, perform actions 3b, 3c, and 3d

5. False ⟶ Ifswitch $TEST)

6. True ⟶ Ifswitch ($TEST)



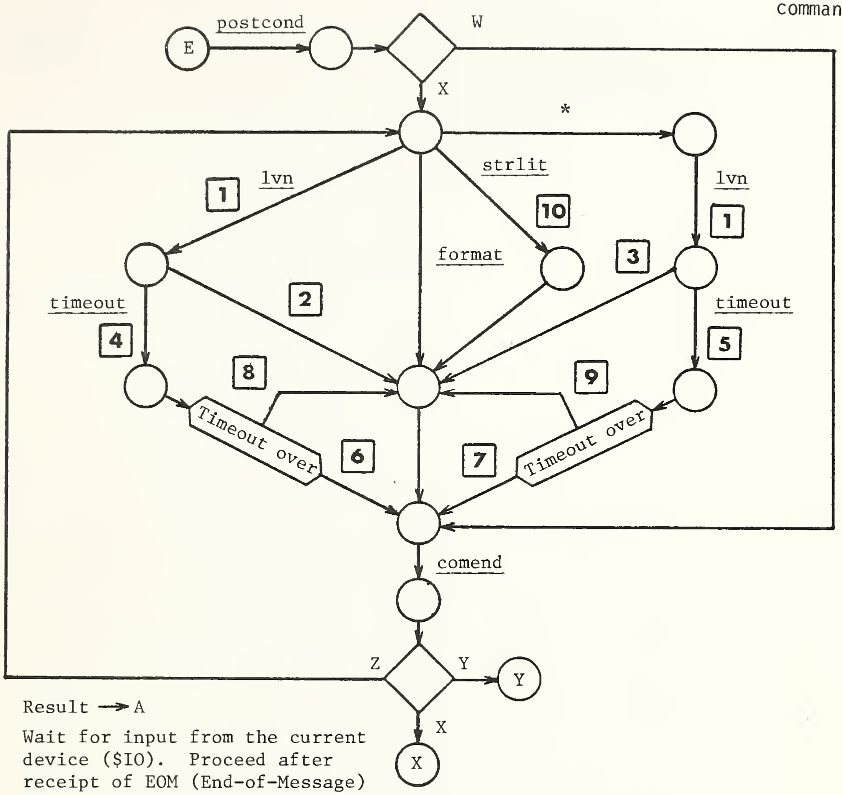
1. GET current FOR information off the Stack. Forsw - 1 ⟶ Forsw

2. RESET FOR lvn name, FOR body position, loop counters (if any), and FOR window position from Stack. The value of entry i going to the FOR command is the number of loop counters + 1. Link window to FOR window position

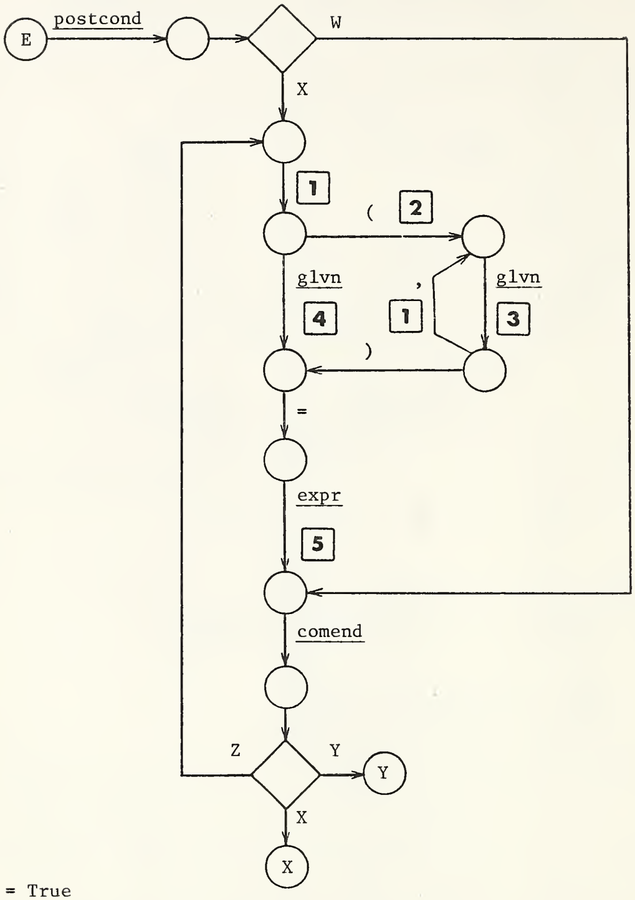
3. GET Argind, Forsw, Window position, and Present routine name from Stack. GET the indirect string off the Stack. Load routine if necessary. Link window to retrieved Window position

4. GET Argind, Indsw, Forsw, Window position, and Present routine name from Stack. Load routine if necessary. Link window to retrieved Window position

5. Terminate execution



1. Result → A
2. Wait for input from the current device ($IO). Proceed after receipt of EOM (End-of-Message) and place input string into variable named in A. $X + $L(A) → $X
3. Wait for input from the current device. Proceed after receipt of one character and place the integer character code into variable in A. $X + 1 → $X
4. Set up a timer of Timeout seconds. Check for receipt of EOM and proceed if received. Otherwise, proceed after receipt of EOM or expiration of timer, whichever occurs first
5. Set up a timer of Timeout seconds. Check for receipt of one character and proceed if received. Otherwise, proceed after receipt of one character or expiration of timer, whichever occurs first
6. Place input string into variable named in A. True → Ifswitch ($TEST). $X + $L(A) → $X
7. Place the integer character code of input into variable named in A. True → Ifswitch ($TEST). $X + 1 → $X
8. Place the input string so far received into variable named in A. False → Ifswitch ($TEST). $X + $L(A) → $X
9. Place -1 into variable named in A. False → Ifswitch ($TEST)
10. Output Result to the current device. $X + $L(Result) → $X



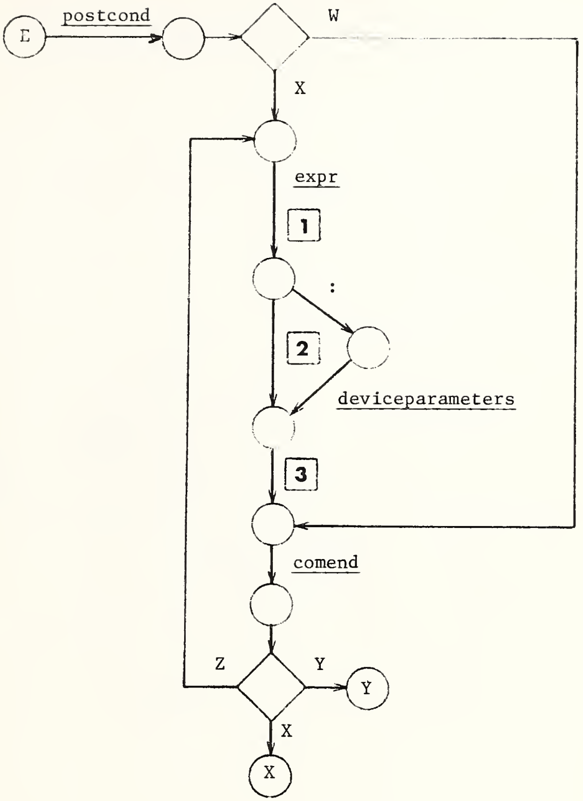
1. Set Setsw = True

2. Null string ("") ⟶ B

3. Replace the n-tuple in B with the n + 1-tuple (B:Result)

4. Result ⟶ B

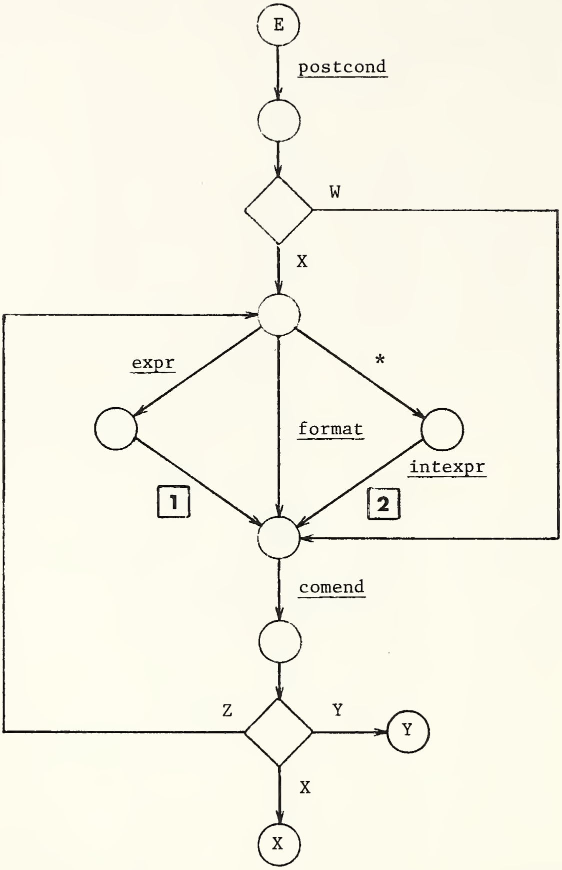
5. For each of the variables named in B, proceeding from left-to-right, perform the following operations:
   a. Place the next variable in B into A
   b. If the variable in A is a global variable, do action 3 of glvn
   c. Place the value in Result in the variable named in A



1. Result ⟶ Devnam

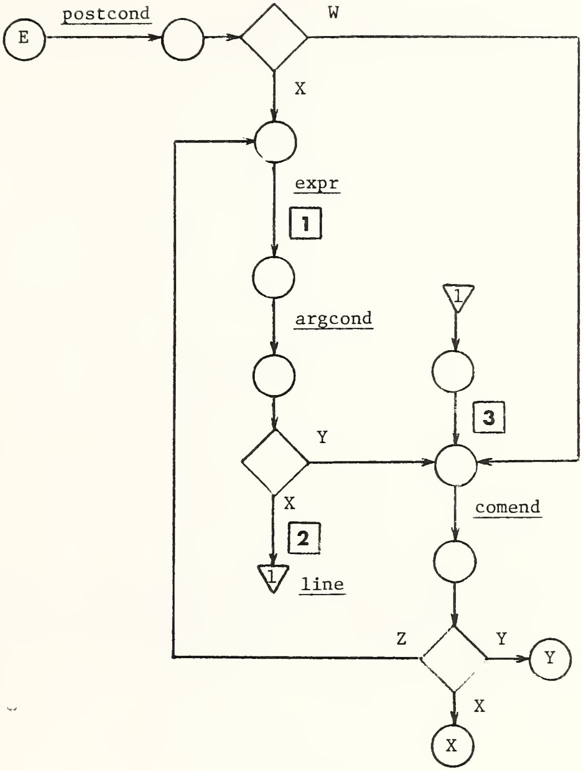
2. Null string ("") ⟶ Result

3. Search the Openlist for the device named in Devnam. If not found, trap execution. Otherwise, perform the following operations:
   a. Set $IO to the device named in Devnam, and make this device the current device for all input and output
   b. If Result contains any device parameters, find the device named in Devnam in the Devicelist, and change those parameters which appear in Result to their new value from Result
   c. Perform any initialization procedures for the device named in Devnam according to its device parameters in Devicelist



1. Output to the current device the value of Result as a string. The effect of this string at the device is defined by the ASCII Standard and conventions. Update $X and $Y as described in Section 3.5.5 of MDC/28

2. Output the character whose character code is in Result. This output may be performed in a device-dependent manner

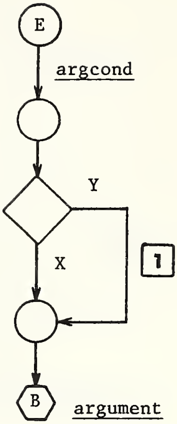


1. Result → A

2. Indsw + 1 → Indsw. Append the string eol ls Q eol to the string in A, and PUT A on Stack. PUT Present routine name, Window position, Forsw, and Argind on Stack. Position window to the first character of the indirect string

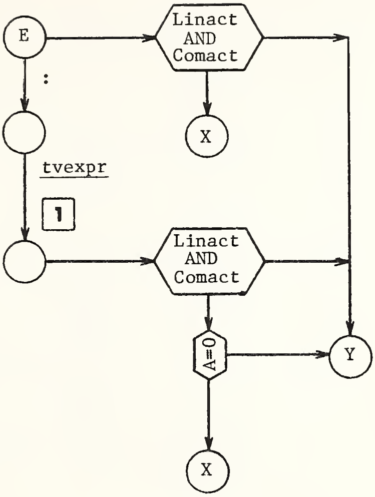
3. Indsw - 1 → Indsw

postcond



1. Set semantic action flag Comact False

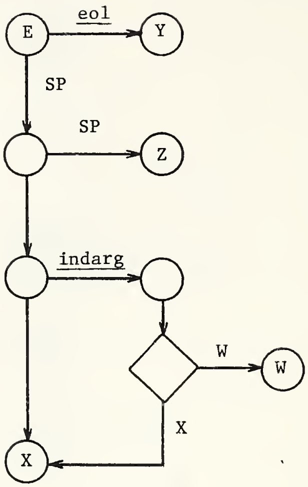
Note: If argcond returns a Y condition (tvexpr is False), postcond inhibits execution of all semantic actions for the command until the arguments terminate. However, all syntactic paths are traversed as normal (that is, syntax checking is performed).



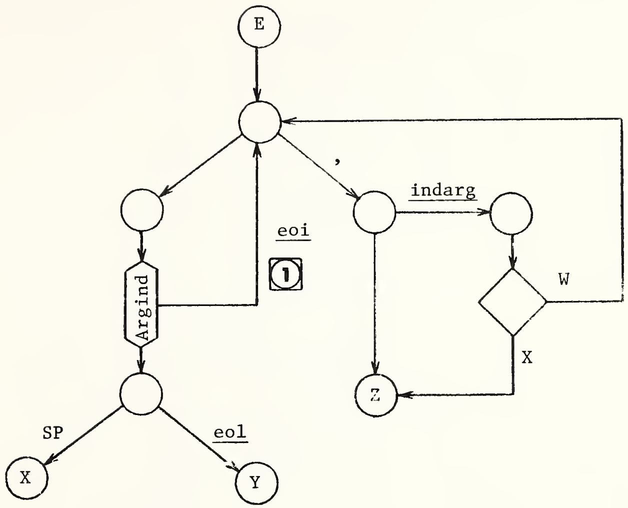
1. Result → A

Note: The argcond diagram scans off the optional post-conditional wherever it may appear, both after command names and within command arguments. The meanings of the exits from argcond are:

X condition is true; execute argument
Y condition is false; skip argument.



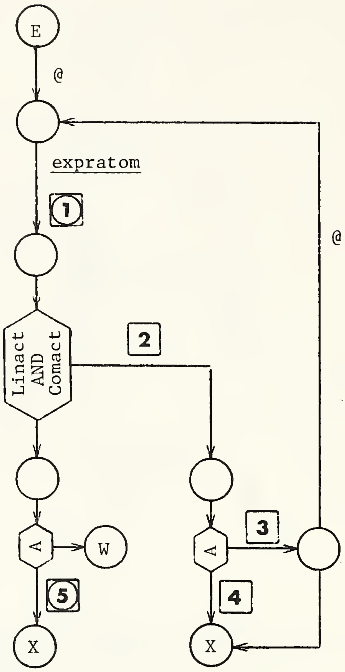
Note: The argument diagram is used to scan off the delimiter between the command word and its arguments (if any). It also handles indirection on the first argument of a command.



(1.)* GET previous level's Argind and Window position from the Stack. GET the indirect argument from the Stack. Link window to retrieved Window position

Note: The comend diagram is used to scan off the delimiter after each argument of a command. It also handles termination of argument level indirection, and tests for argument-level indirection in the next argument (if one is present).

*Privileged action, always executed



(1.)* False → A. If character in window is , (comma), SP (space), eol, or eoi, then True → A

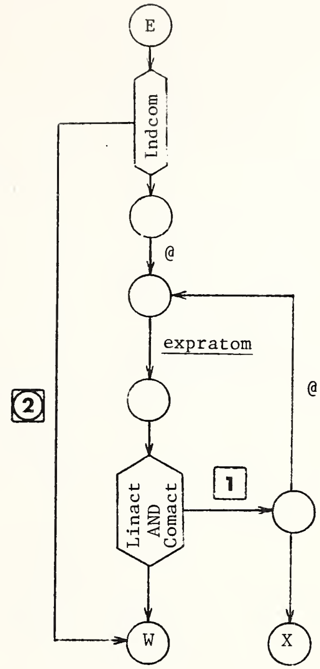
2. If Result contains an SP (space) not within quotes, or an eol or eoi, trap execution. Otherwise, append an eoi to Result, then PUT Result and current Window position on the Stack. Position window to the first character of the indirect argument

3. PUT Argind on Stack. True → Argind

4. If character in window is not an alpha, digit, % (percent), @ (commercial at), or ∧ (circumflex), trap execution. Otherwise, PUT Nameind on Stack. True → Nameind

(5.)* True → Indcom (used to indicate that name level indirection has been detected and scanned while syntax checking; see the indnam diagram)

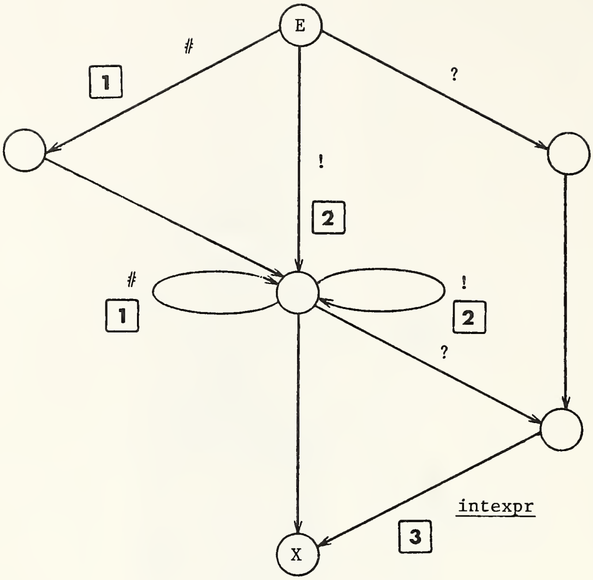
*Privileged action, always executed



1. Append an eoi to Result, then PUT Result and current Window position on the Stack. Position window to the first character of the indirect name. PUT Nameind on Stack. True ⟶ Nameind

(2.)* False ⟶ Indcom

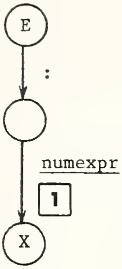
*Privileged action, always executed



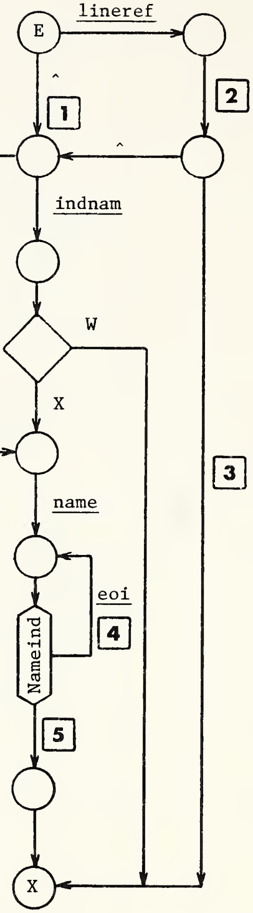
1. Output top-of-form operation on current device. Place 0 → $X, 0 → $Y

2. Output new line operation on current device. Place 0 → $X, $Y + 1 → $Y

3. Result → A. Tab to column A on current device; that is, output max(0, A - $X) spaces. Place max($X, A) → $X



1. If Result > 0, Result → Timeout. Otherwise, 0 → Timeout



1. Null string ("") → A

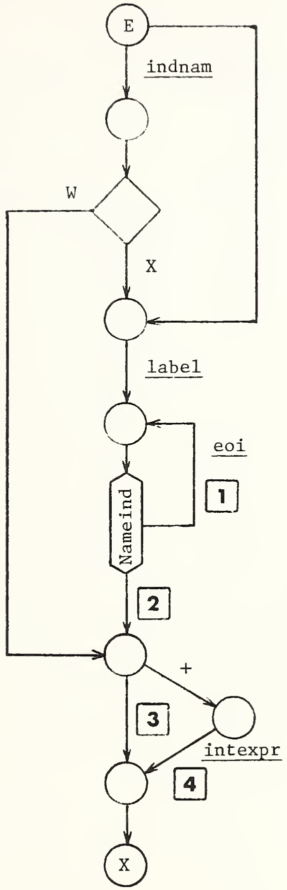
2. Result → A

3. Place the 2-tuple (A,"") → Result

4. GET previous level's Nameind and Window position from the Stack. GET the indirect name off the Stack. Link window to retrieved Window position

5. Place the 2-tuple (A,Result) → Result

Note: An entryref is an ordered pair of the form (a,b), where a is a label + offset and b is a routine name. If a is null, the interpretation of (a,b) is the first line of routine b. If b is null, the interpretation is label a in the present routine. If neither is null, the interpretation is label a of routine b.



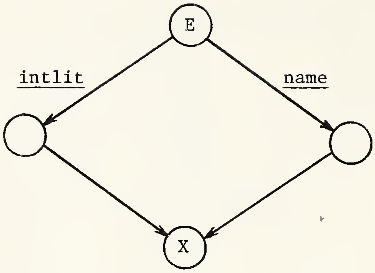
1. GET previous level's Nameind and Window position from the Stack. GET the indirect name off the Stack. Link window to retrieved Window position

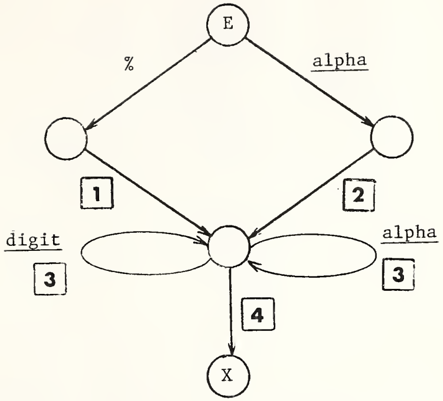
2. Result → A

3. Place the 2-tuple (A + 0) → Result

4. If Result < 0, trap execution. Otherwise, place the 2-tuple (A + Result) → Result

Note: A lineref is of the form label + offset, where offset is a positive integer n denoting the nth line after the one containing label.


E
intlit
name
X

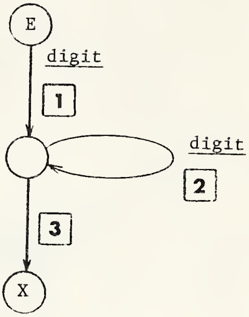


1. Percent ("%") → A

2. Char → A

3. Concatenate (A, Char) → A

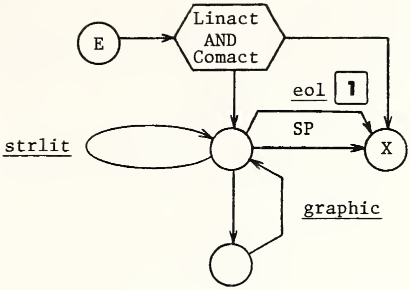
4. A → Result



1. Char → A

2. Concatenate (A, Char) → A

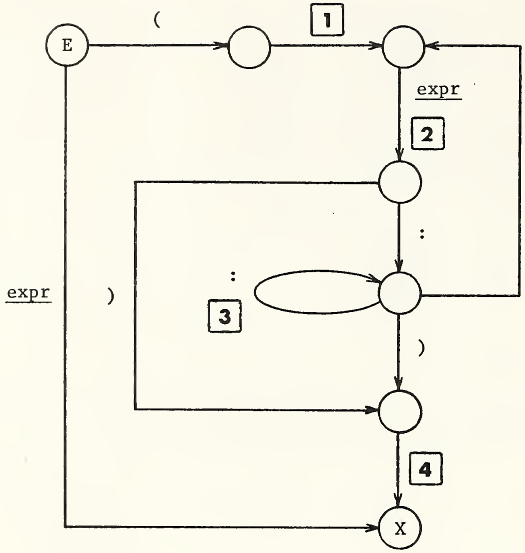
3. A → Result



1. Move Window position one to the left

Note: The dummylist diagram is used by the FOR command to scan through the argument list of the FOR to the body of the FOR.

deviceparameters

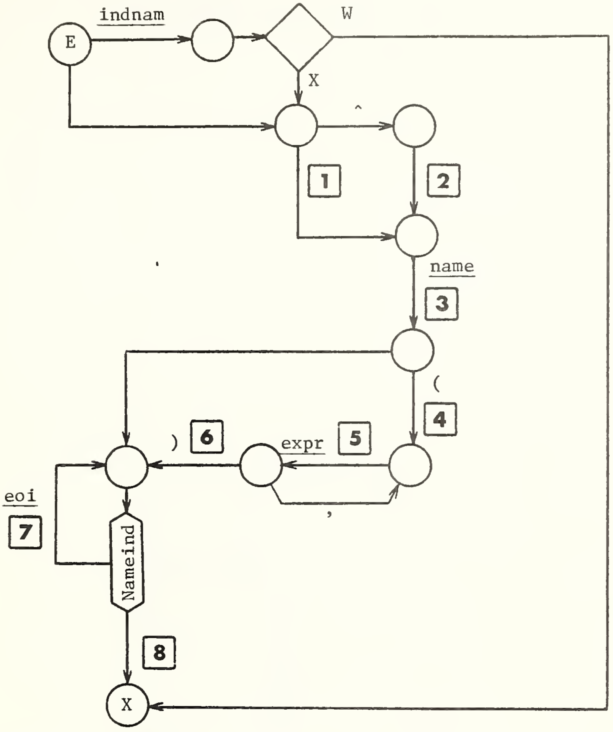


1. Null string ("") ⟶ A

2. Replace the n-tuple in A by the n + 1-tuple (A:Result)

3. If there exists a default value for this parameter of the device named in Devnam, replace the n-tuple in A by the n + 1-tuple (A:d), where d is the default value. Otherwise append a colon (":") to A to hold this parameter's position

4. A ⟶ Result



1. Null string ("") → A

2. Circumflex ("^") → A

3. Concatenate (A, Result) → A

4. PUT Nameind on the Stack. False → Nameind

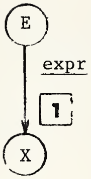
5. Replace the n-tuple in A with the n + 1-tuple (A, Result)

6. GET Nameind from the Stack

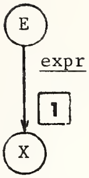
7. GET previous level's Nameind and Window position from the Stack. GET the indirect name off the Stack. Link window to retrieved Window position

8. A → Result

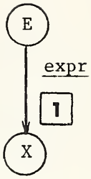
Note: The result of calling nref is an n-tuple of values, where the first value is the name, possibly preceded by a "^", and subsequent values (if any) are subscripts.



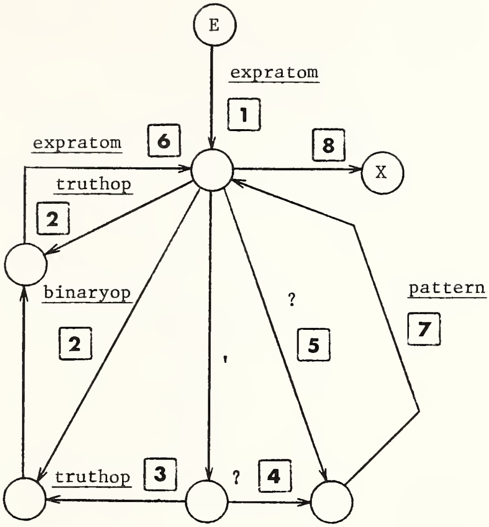
1. Result → A. Take the numeric interpretation of the value in A. That is, apply the rules given in Section 3.2.5 of MDC/28 to A. A → Result



1. Result → A. Take the numeric interpretation of the value in A. That is, apply the rules given in Section 3.2.5 of MDC/28 to A. Remove any fraction. If the result is the null string (""), or "-", 0 → A. A → Result



1. Result → A. Take the numeric interpretation of the value in A. That is, apply the rules given in Section 3.2.5 of MDC/28 to A. If A ≠ 0, then 1 → A. A → Result



1. Result → A

2. Operator (Result) → B

3. Operator (not Result) → B

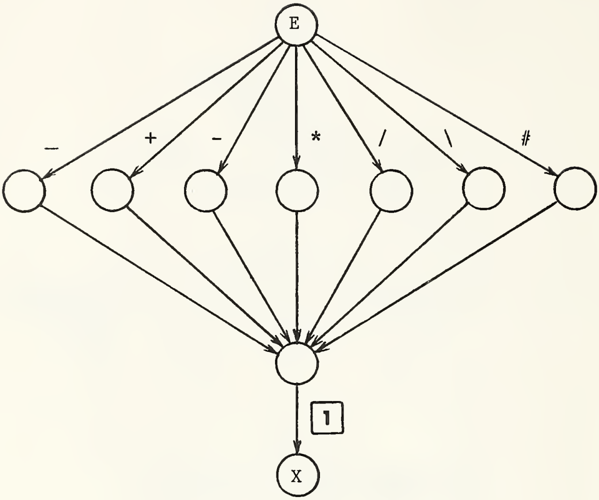
4. Operator (not ?) → B

5. Operator (?) → B

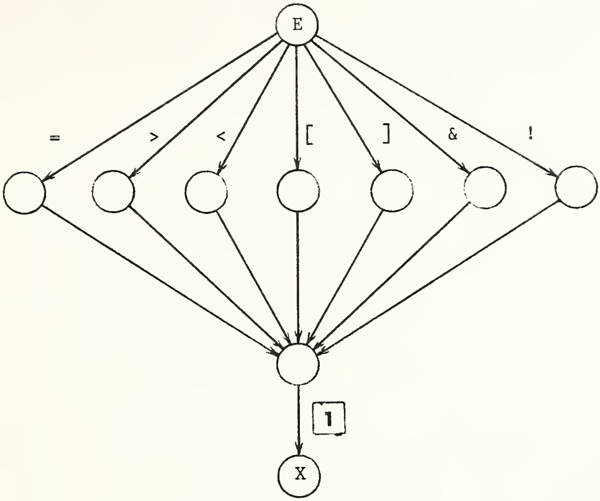
6. Apply the binary operator in B to A as left operand and Result as right operand, placing the resulting value in A

7. Apply the pattern in Result to A, placing the resulting truth value in A. If the operator in B is not ?, logically complement A

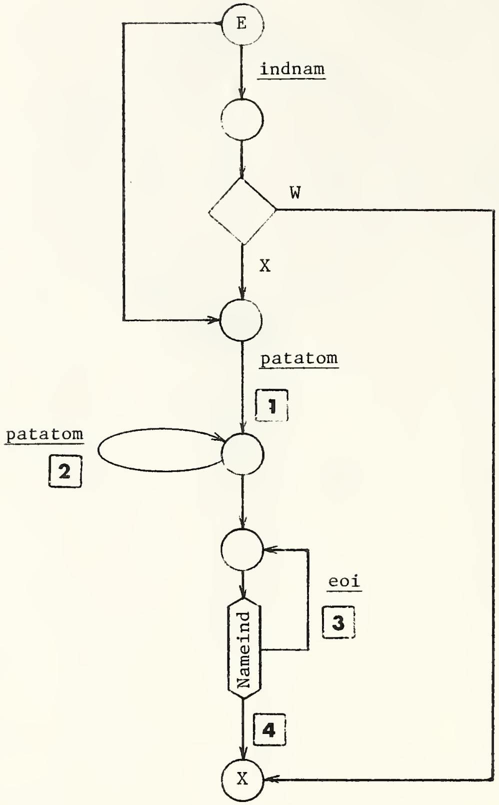
8. A → Result



1. Char → Result



1. Char → Result



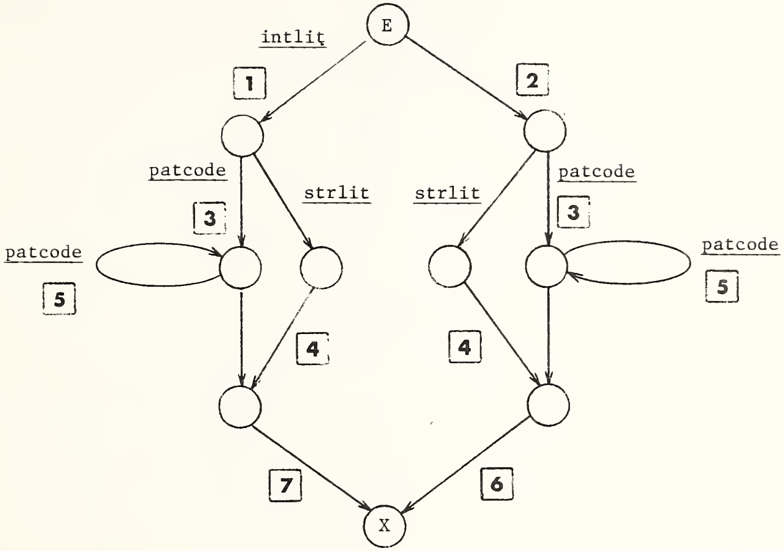
1. Result ⟶ A

2. If Result is not the null string (""), replace the n-tuple A by the n + 1-tuple (A, Result)

3. GET previous level's Nameind and Window position from the Stack. GET the indirect name off the Stack. Link window to retrieved Window position

4. A ⟶ Result

   Note: The value of pattern is an n-tuple of patatoms. It may be an 0-tuple.



1. Result → A

2. Zero ("0") → A

3. Result → B

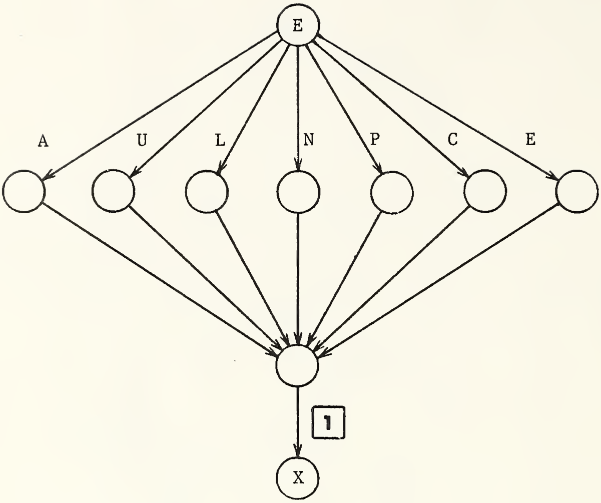
4. Result as "literal" → B

5. Concatenate (B, Result) → B

6. Place the 2-tuple (A,B) → Result

7. If A > 0, do action 6. Otherwise, null string ("") → Result

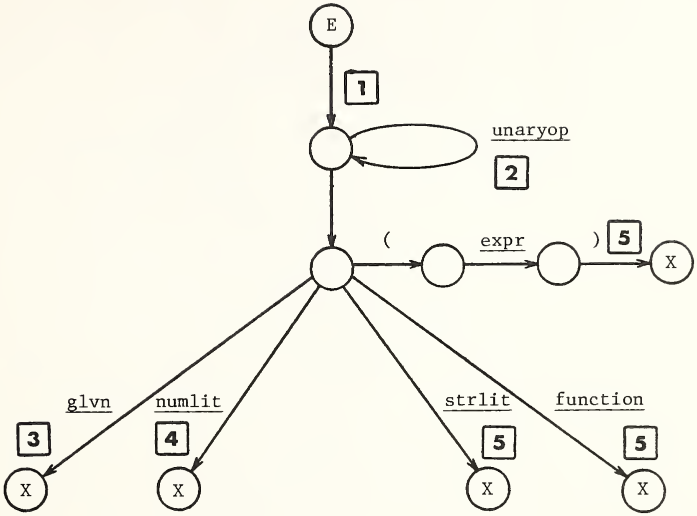
Note: The Result returned by patatom is a 2-tuple, where the value of the first element is either zero or the value of intlit, and the value of the second element is either the Result from strlit as "literal", or a string of patcodes. If intlit returns zero, patatom returns the null string.



1. Char → Result

2.5 Expression Atom Diagrams

expratom



1. Null string ("") ⟶ A

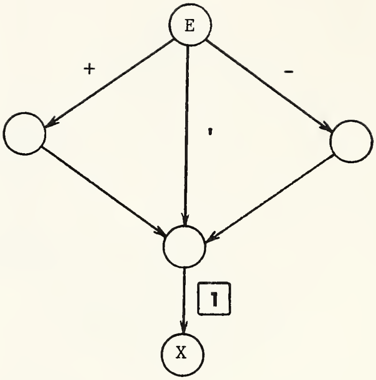
2. Concatenate (A,Result) ⟶ A

3. Search the set of local or global variables for the one with the name contained in Result. If the search is successful, place the value of the variable into Result. Then do action 5. If the search is not successful, trap execution

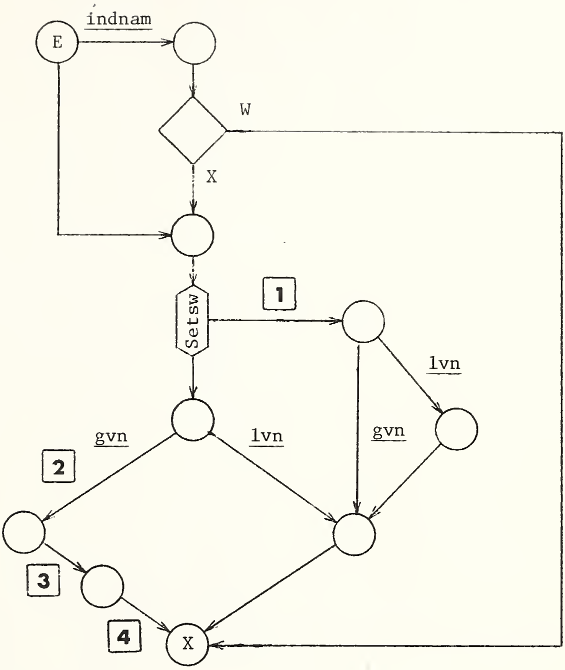
4. If Nameind is True, trap execution. Otherwise, do action 5

5. Apply the unary operators in A (if any) to the value in Result, in a right-to-left order. Place the result of this application into Result

unaryop



1. Char → Result

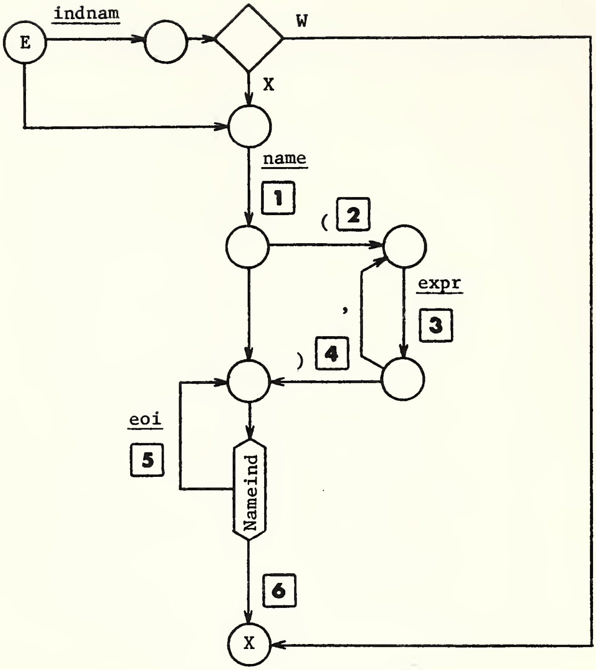


1. Set Setsw = False

2. Result ⟶ A

3. Perform the following operations in order:
   a. If the first value of the n-tuple in A is the naked symbol alone ("^"), substitute the value of the Naked indicator for it. For example, if A is the n-tuple (^, s_1, s_2, ..., s_{n-1}) and the Naked indicator is the m-tuple (^name, v_1, v_2, ..., v_{m-1}), then A becomes the m - 1 +n-tuple (^name, v_1, v_2, ..., v_{m-1}, s_1, s_2, ..., s_{n-1})
   b. The value in A is now an n-tuple (^name, x_1, x_2, ..., x_{n-1}). If A is a 1-tuple (no subscripts), or $D(A) < 10, make the Naked indicator undefined. Otherwise, replace the Naked indicator with the n - 1-tuple (^name, x_1, x_2, ..., x_{n-2})

4. A ⟶ Result



1. Result → A

2. PUT Nameind on the Stack. False → Nameind

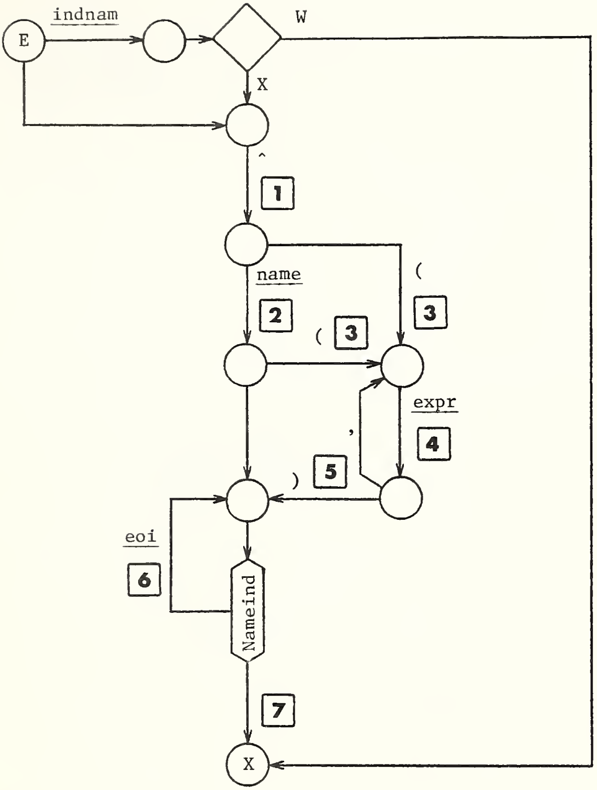
3. Replace the n-tuple in A with the n + 1-tuple (A,Result)

4. GET Nameind from the Stack

5. GET previous level's Nameind and Window position from the Stack. GET the indirect name off the Stack. Link window to retrieved Window position

6. A → Result

Note: The result of calling lvn is an n-tuple of values, where the first value is the name and subsequent values (if any) are subscripts.



1. Circumflex ("^") → A

2. Concatenate (A, Result) → A

3. PUT Nameind on the Stack. False → Nameind

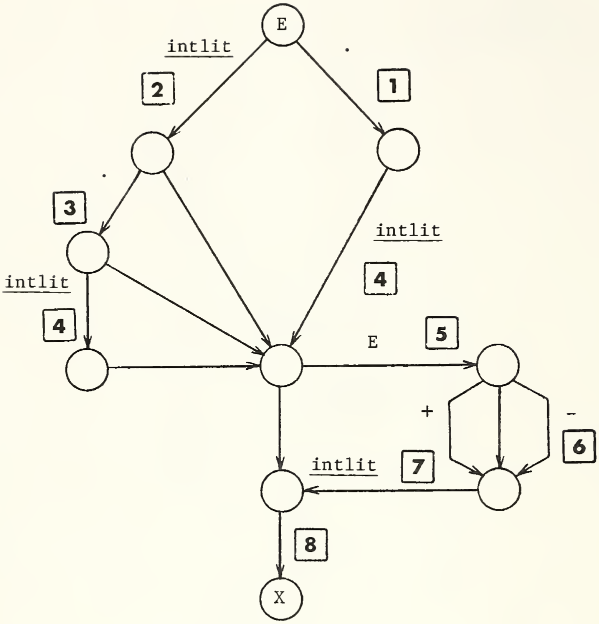
4. Replace the n-tuple in A with the n + 1-tuple (A, Result)

5. GET Nameind from the Stack

6. GET previous level's Nameind and Window position from the Stack. GET the indirect name off the Stack. Link window to retrieved Window position

7. A → Result

Note: The result of calling gvn is an n-tuple of values, where the first value is either the global name preceded by a "^", or the "^" alone, and subsequent values (if any) are subscripts.



1. Period (".") ⟶ A

2. Remove leading zeros from Result. If Result > 0, Result ⟶ A.
Otherwise, zero ("0") ⟶ A

3. Concatenate (A, ".") ⟶ A

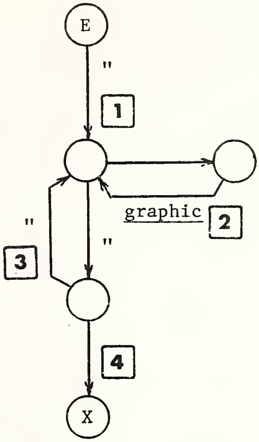
4. Remove trailing zeros from Result. Concatenate (A, Result) ⟶ A

5. Concatenate (A, "E") ⟶ A

6. Concatenate (A, "-") ⟶ A

7. Remove leading zeros from Result. Concatenate (A, Result) ⟶ A

8. Convert A to a numeric data value, using the algorithm defined in Section 3.2.4.2 of MDC/28. Place A ⟶ Result



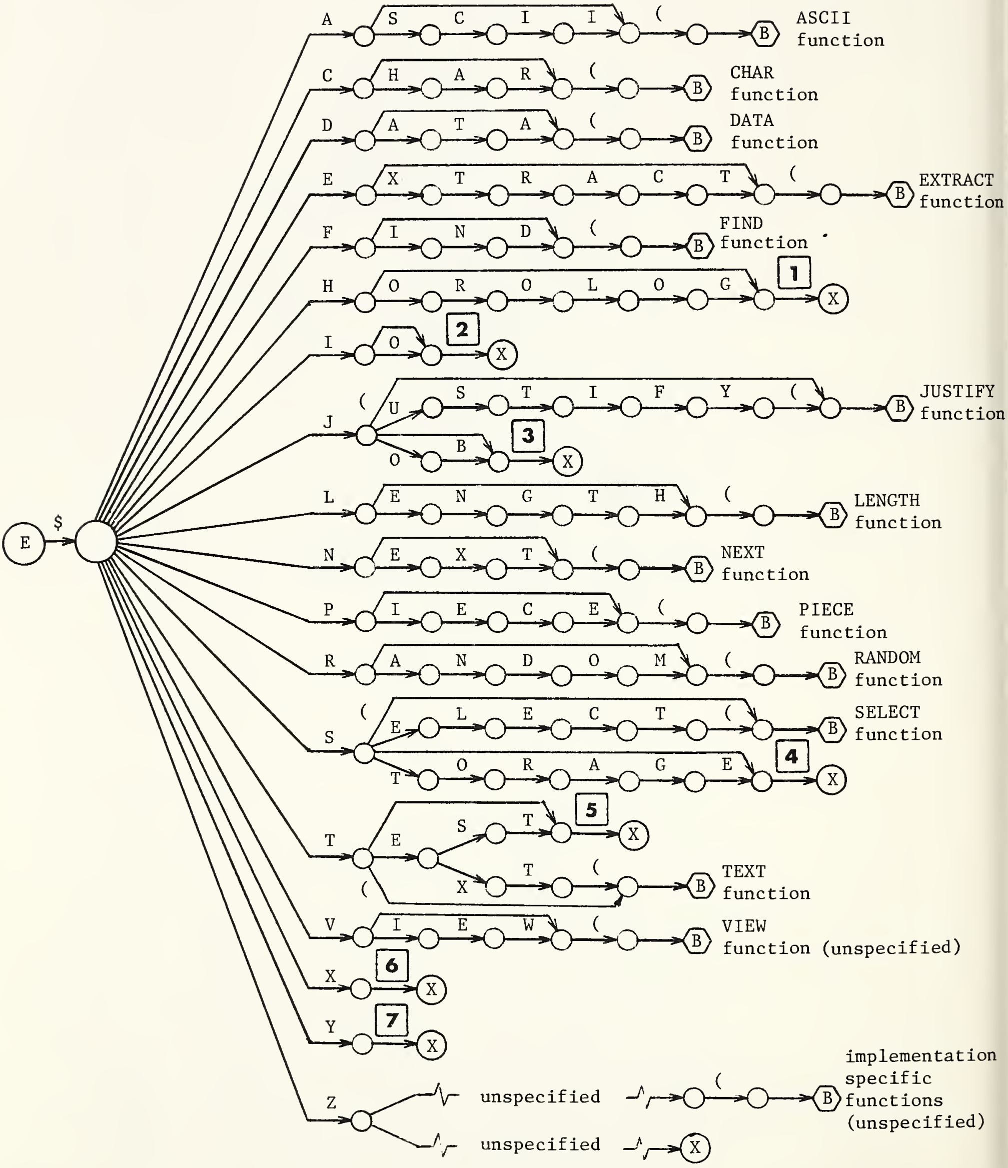
1. Null string ("") ⟶ A

2. Concatenate (A,Char) ⟶ A

3. Concatenate (A,"""") ⟶ A

4. A ⟶ Result

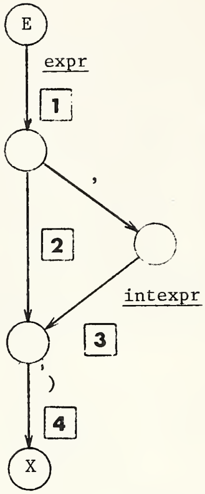
function


E
$
A S C I I (B ASCII function
C H A R (B CHAR function
D A T A (B DATA function
E X T R A C T (B EXTRACT function
F I N D (B FIND function
H O R O L O G 1 X
I O 2 X
J (U S T I F Y (B JUSTIFY function
O B 3 X
L E N G T H (B LENGTH function
N E X T (B NEXT function
P I E C E (B PIECE function
R A N D O M (B RANDOM function
S (E L E C T (B SELECT function
T O R A G E 4 X
T E S T 5 X
X T (B TEXT function
(
V I E W (B VIEW function (unspecified)
X 6 X
Y 7 X
Z unspecified (B implementation specific functions (unspecified)
unspecified X

Actions for function

1. Place present value of $HOROLOG (date and time) into Result
2. Place present value of $IO (current I/O device) into Result
3. Place present value of $JOB (number of this process) into Result
4. Place present value of $STORAGE (remaining available storage) into Result
5. Place present value of $TEST (Ifswitch) into Result
6. Place present value of $X (horizontal cursor position of $IO device) into Result
7. Place present value of $Y (vertical cursor position of $IO device) into Result

$ASCII
function

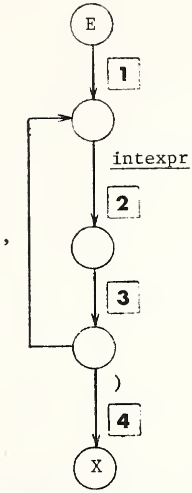


1. Result → A

2. One ("1") → B

3. Result → B

4. If A is the null string, place -1 into Result. Otherwise, place the decimal equivalent of the ASCII code of the Bth character of A into Result



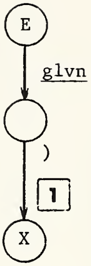
1. Null string ("") → A

2. Result → B

3. If B > 127, trap execution. Otherwise, if B > -1, convert B to its ASCII character, place it into C, and concatenate (A,C) → A

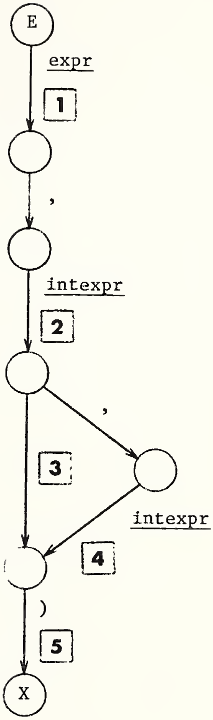
4. A → Result

$DATA
function



1. Result → A. Place characterization of the named variable in A into Result

Note: The meaning of the value returned by the $DATA function is discussed in Section 3.2.8 of MDC/28.



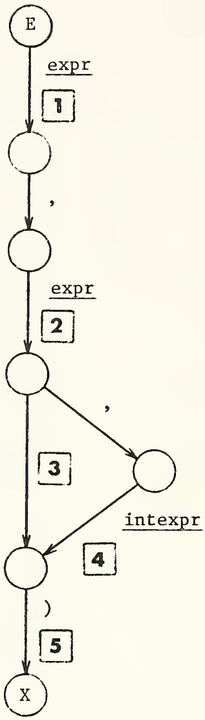
1. Result → A

2. Result → B

3. B → C

4. Result → C

5. Extract from A the Bth through Cth characters and place into Result



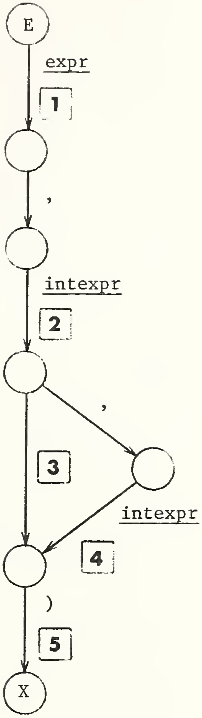
1. Result ⟶ A

2. Result ⟶ B

3. One ("1") ⟶ C

4. Result ⟶ C

5. Find the first occurrence of B in A, beginning at the Cth character of A. If B is contained in A, place n + 1 into Result, where n is the position in A of the last character in B. Otherwise, 0 ⟶ Result



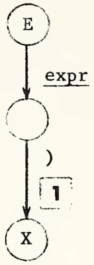
1. Result ⟶ A

2. Result ⟶ B

3. Right justify A in a field of B spaces and place into C

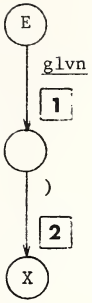
4. If Result < 0, trap execution. Otherwise, perform the following operations in order:
   a. Take the numeric interpretation of A. That is, apply the rules given in Section 3.2.5 of MDC/28 to A
   b. If Result = 0, round the numeric value in A to an integer and remove the decimal point. Otherwise, round A to Result fraction digits (pad with trailing zeros if necessary)
   c. Do action 3

5. C ⟶ Result



1. Result → A. Place length of A in characters into Result

$NEXT
function

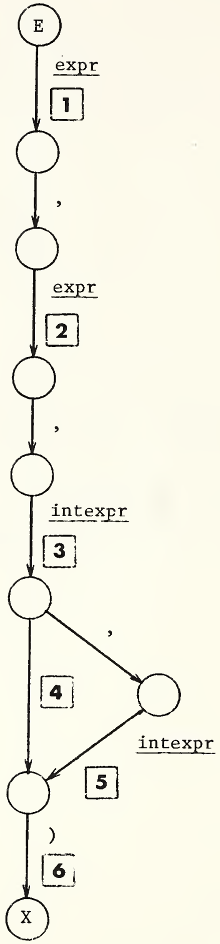


1. Result → A. Check to see that A is at least a 2-tuple (that is, that the variable is subscripted). If not, generate an execution trap

2. Find the next higher subscript of the variable named in A and place into Result. If no higher subscript exists, place -1 into Result

   Note: The details of action 2 are discussed in Section 3.2.8 of MDC/28.

$PIECE
function



1. Result → A

2. Result → B

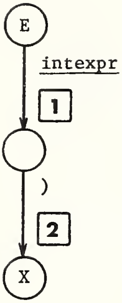
3. Result → C

4. C → D

5. Result → D

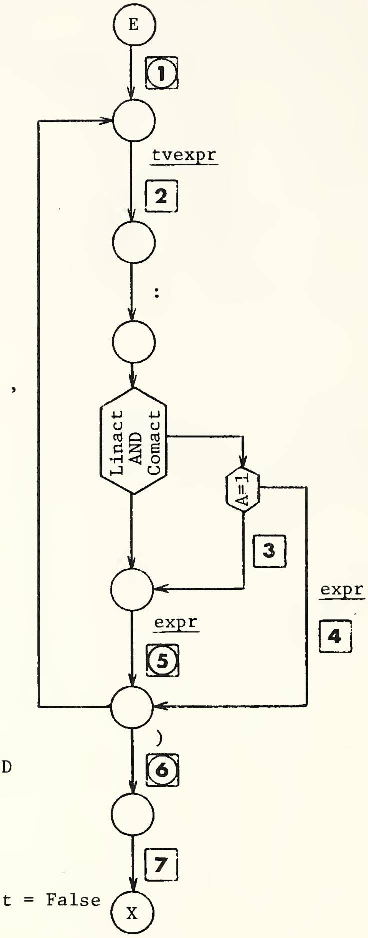
6. Place the C through D fields in A, delimited by B, into Result

Note: The details of action 6 are described in Section 3.2.8 of MDC/28.



1. If Result < 1, trap execution. Otherwise, Result → A

2. Compute a "random" integer between zero and A - 1 inclusive, and place into Result



1.* Linact → C, Comact → D

2. Result → A

3. Set Comact = False

4. Result → B. Set Linact = False

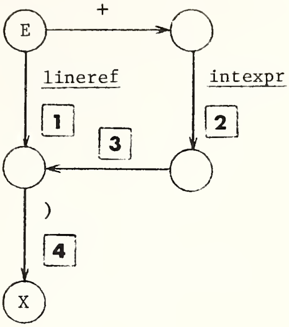
5.* Set Comact = True

6.* C → Linact, D → Comact

7. If B is undefined, trap execution. Otherwise, B → Result

Note: Linact and Comact are used only locally here to inhibit evaluation of the second expression in a False Boolean pair. However, syntax checking is performed.

*Privileged action; always executed



1. Find the line in the present routine referenced by the lineref in Result. If no such line exists, place the null string ("") into C. Otherwise, line ⟶ C

2. If Result < 1, trap execution. Otherwise, Result ⟶ B

3. Find the Bth line of the present routine. If no such line exists, place the null string ("") into C. Otherwise, line ⟶ C

4. If C is not the null string (""), replace the ls in C with one SP (space) and remove the eol. C ⟶ Result

MDC/34
9/17/75

MUMPS LANGUAGE STANDARD

Part III: MUMPS Portability Requirements

Table of Contents

1. Introduction

This document highlights, for the benefit of implementors and application programmers, aspects of the language that must be accorded special attention if MUMPS program transferability (i.e., portability of source code between various MUMPS implementations) is to be achieved. It provides a specification of limits that must be observed by both implementors and programmers if portability is not to be ruled out. To this end, implementors must meet or exceed these limits, treating them as a minimum requirement; application programmers, on the other hand, may meet but must not exceed the limits guaranteed by this document. Any implementor who provides definitions in currently undefined areas must take into account that this action risks jeopardizing the upward compatibility of the implementation, upon subsequent revision of the MUMPS Language Specification. Application programmers striving to develop portable programs must take into account the danger of employing "unilateral extensions" to the language made available by the implementor.

The following definitions apply to the use of the terms "explicit limit" and "implicit limit" within this document. An explicit limit is one which applies directly to a referenced language construct. Implicit limits on language constructs are second-order effects resulting from explicit limits on other language constructs. For example, the explicit command line length restriction places an implicit limit on the length of any construct which must be expressed entirely within a single command line.

2. Expression Elements

2.1 Names

The use of alpha in names is restricted to upper case alphabetic characters. While there is no explicit limit on name length, only the first eight characters are uniquely distinguished. This length restriction places an implicit limit on the number of unique names.

2.2 Local Variables

2.2.1 Number of Local Variables

The number of local variable names in existence at any time is not explicitly limited. However, there are implicit limitations due to the storage space restrictions (Section 6).

2.2.2 Number of Subscripts

There is no explicit restriction on the number of subscripts in any local variable name. The command line length restriction (Section 4.1) places an implicit limit on the number of subscripts in any local variable name.

2.2.3 Values of Subscripts

Local variable subscript values are nonnegative integer values, as defined in Section 3.2.4.1 of the MUMPS Language Specification (also see Section 2.6 below). This restriction is equivalent to the following.

> Any local variable subscript value meets the following criteria:
>
> a. It may contain only digits;
> b. At least one digit must be present;
> c. The number zero is represented by "0";
> d. Except for the respresentation of zero, the string of digits must have no leading zeros;
> e. Its numeric value must be within the integer range stated in Section 2.6.

The use of subscript values which do not meet these criteria is undefined, with the exception of the use of the value "-1" as the last subscript of a reference within the context of the $NEXT function. Note that an implicit integer interpretation of a subscript value is not performed; where desired, this may be accomplished by using the unary plus operator.

2.2.4 Number of Nodes

There is no explicit limit on the number of distinct nodes which are defined within local variable arrays. However, the limit on the number of local variables (Section 2.2.1) and the limit on the number of subscripts (Section 2.2.2) place an implicit limit on the number of distinct nodes which may be defined.

2.2.5 Scope of Local Variables

Local variables are unique to a process. All routines executed by a process share the same name space.

2.3 Global Variables

2.3.1 Number of Global Variables

There is no explicit limit on the number of distinct global variable names in existence at any time.

2.3.2 Number of Subscripts

There in no explicit restriction on the number of subscripts in any global variable name. There is an implicit limit on the number of subscripts within a global name due to the command line length restriction (Section 4.1). This does not restrict the depth of subscripting within a global array, since repeated naked references may be used to access nodes at any depth.

2.3.3 Values of Subscripts

The restrictions imposed on the values of global variable subscripts are identical to those imposed on local variable subscripts (Section 2.2.3).

2.3.4 Number of Nodes

There is no limit on the number of distinct global variable nodes which are defined, since successive naked references may be used to access and/or create nodes at any depth within a global array.

2.4 Data Types

The MUMPS Language Specification defines a single data type, namely, variable length character strings. Contexts which demand a numeric, integer, or truth value interpretation are satisfied by unambiguous rules for mapping a string datum into a number, integer, or truth value.

The implementor is not limited to any particular internal representation. Any internal representation(s) may be employed as long as all necessary mode conversions are performed automatically and all external behavior agrees with the MUMPS Language Specification. For example, integers might be stored as binary integers and converted to decimal character strings whenever an operation requires a string value.

2.5 Number Range

All values used in arithmetic operations or in any context requiring a numeric interpretation are within the inclusive intervals $[-10^{25}, -10^{-25}]$ or $[10^{-25}, 10^{25}]$, or are zero.

The accuracy of any value used in arithmetic operations or in any context requiring a numeric interpretation is nine significant digits.

Programmers should exercise caution in the use of noninteger arithmetic. In general, arithmetic operations on noninteger operands or arithmetic operations which produce noninteger results cannot be expected to be exact. In particular, noninteger arithmetic can yield unexpected results when used in loop control or arithmetic tests.

2.6 Integers

The magnitude of the value resulting from an integer interpretation is limited by the accuracy of numeric values (Section 2.5). The values produced by integer valued operators and functions also fall within this range (see Section 3.2.5.1 of the MUMPS Language Specification for a precise definition of integer interpretation).

2.7 Character Strings

Character string length is limited to 255 characters. The characters permitted within character strings are those defined in the ASCII Standard (ANSI X3.4-1968).

2.8 Special Variables

The special variables $X and $Y are nonnegative integers (Section 2.6). The effect of incrementing $X and/or $Y past the maximum allowable integer value is undefined (for a description of the cases in which $X and $Y are incremented see the MUMPS Language Specification, Section 3.5.5).

3. Expressions

3.1 Nesting of Expressions

The number of levels of nesting in expressions is not explicitly limited. The maximum string length does impose an implicit limit on this number (Section 2.7).

3.2 Results

Any result, whether intermediate or final, which does not satisfy the constraints on character strings (Section 2.7) is erroneous. Furthermore, integer results are erroneous if they do not also satisfy the constraints on integers (Section 2.6).

4. Routines and Command Lines

4.1 Command Lines

A command line (<u>line</u>) must satisfy the constraints on character strings (Section 2.7). The length of a command line is determined as follows. Each character in the label (if present) counts as one character. The <u>ls</u> character counts as one character (note that command lines will therefore always be at least one character long). Each character following the <u>ls</u> up to but not including the following <u>eol</u> counts as one character. The sum of the lengths of these three components (label, <u>ls</u>, and the command line proper) determines the length of the command line.

The characters within a command line are restricted to the 95 ASCII graphics. The character set restriction places a corresponding implicit restriction upon the value of the argument of the indirection delimiter (Section 5).

4.2 Number of Command Lines

There is no explicit limit on the number of command lines in a routine, subject to storage space restrictions (Section 6).

4.3 Number of Commands

The number of commands per line is limited only by the restriction on the maximum command line length (Section 4.1).

4.4 Labels

A label of the form name is subject to the constraints on names; labels of the form intlit are subject to the length constraint on names (Section 2.1).

4.5 Number of Labels

There is no explicit limit on the number of labels in a routine. However, the following restrictions apply.

a. A command line may have only one label.
b. No two lines may be labeled with equivalent (not uniquely distinguishable) labels.

4.6 Number of Routines

There is no explicit limit on the number of routines. The number of routines is implicitly limited by the name length restriction (Section 2.1).

5. Indirection

The value of the argument of indirection and the argument of the XECUTE command are subject to the constraints on character string length (Section 2.7). They are additionally restricted to the character set limitations of command lines (Section 4.1).

6. Storage Space Restrictions

MUMPS has traditionally been implemented on small to medium size computers using a scheme of fixed main memory allocation, one fixed partition per user. It is recognized that more flexible storage allocation techniques can be applied and there is no intent to restrict implementations to use of the traditional techniques. Nevertheless, because partitioned memory implementations will continue to be important for some time, certain storage restrictions are required to permit program portability. These restrictions have been defined in terms of parameters which are implementation-independent and observable to the application programmer.

The storage restrictions on portable programs are expressed in the following rule. At any time during the execution of a process, routine size plus local variable storage size plus temporary result storage size must not exceed 4000 characters. Storage space for control purposes, device buffers, disc buffers, line buffers, etc. is not included in this count.

The size of a routine is the sum of the sizes of all the lines in the routine. The size of each line is its length (as defined in Section 4.1) plus two.

The size of local variable storage is the sum of the sizes of all the simultaneously defined local variables. The size of an unsubscripted local variable is the length of its name in characters plus the length of its value in characters, plus two. The size of a local array is the sum of the following.

a. The length of the name of the array.
b. Two characters plus the length of each value.
c. The size of each subscript in each subscript list.
d. Two additional characters for each node N, whenever $DATA(N)$\geq$10.

All subscripts and values are considered to be character strings for this purpose.

All intermediate results generated during the processes of expression evaluation, indirection, multiple SET command scanning, etc. require the use of temporary storage. At any given time, the amount of temporary storage required is the sum of the lengths of all simultaneously existing temporary results. All temporary results are maintained as strings of contiguous characters.

7. Nesting

Each active DO, FOR, XECUTE, and indirection occurrence is counted as a level of nesting. Control storage provides for fifteen levels of nesting. The actual use of all these levels may be limited by storage restrictions (Section 6).

Nesting within an expression is not counted in this limit. Expression nesting is not explicitly limited; however, it is implicitly limited by the storage restriction (Section 6.).

BIBLIOGRAPHY

MUMPS Development Committee Manuals

MDC Doc. No.	Identification
NBS Handbook 118	November, 1975, MUMPS Language Standard
	Part I: MDC/28, 3/12/75, MUMPS Language Specification M. E. Conway
	Part II: MDC/33, 9/17/75, MUMPS Transition Diagrams D. D. Sherertz and A. I. Wasserman
	Part III: MDC/34, 9/17/75, MUMPS Portability Requirements E. A. Gardner and C. B. Lazarus
29	5/28/75, MUMPS Interpreter Validation Program User Guide J. Rothmeier and P. L. Egerman
30	6/25/75, MUMPS Translation Methodology P. L. Egerman, C. B. Lazarus and P. T. Ragon
35	10/14/75, MUMPS Documentation Manual L. J. Peck and R. A. Greenes
1/11	6/13/75, MUMPS Primer M. E. Johnson and R. E. Dayhoff
2/1	5/15/75, MUMPS Globals and Their Implementation A. I. Wasserman, D. D. Sherertz and C. L. Rogerson
2/2	5/30/75, Design of a Multiprogramming System for the MUMPS Language A. I. Wasserman, D. D. Sherertz and R. W. Zears
2/3	6/15/75, Implementation of the MUMPS Language Standard A. I. Wasserman and D. D. Sherertz
3/5	6/20/75, MUMPS Programmers' Reference Manual M. E. Conway and P. L. Egerman

NBS-114A (REV. 7-73)

U.S. DEPT. OF COMM.
BIBLIOGRAPHIC DATA SHEET

1. PUBLICATION OR REPORT NO.
NBS HB-118

2. Gov't Accession No.

3. Recipient's Accession No.

4. TITLE AND SUBTITLE

MUMPS Language Standard

5. Publication Date
January 1976

6. Performing Organization Code

7. AUTHOR(S) Joseph T. O'Neill, Editor

8. Performing Organ. Report No.

9. PERFORMING ORGANIZATION NAME AND ADDRESS

NATIONAL BUREAU OF STANDARDS
DEPARTMENT OF COMMERCE
WASHINGTON, D.C. 20234

10. Project/Task/Work Unit No.
6401416

11. Contract/Grant No.
5-7750007

12. Sponsoring Organization Name and Complete Address *(Street, City, State, ZIP)*
National Center for Health Services Research
Health Resources Administration
Department of Health, Education, and Welfare
Rockville, Maryland 20852

13. Type of Report & Period Covered
Final
Feb 74 - June 75

14. Sponsoring Agency Code

15. SUPPLEMENTARY NOTES

Library of Congress Catalog Card Number: 75-619261

16. ABSTRACT *(A 200-word or less factual summary of most significant information. If document includes a significant bibliography or literature survey, mention it here.)*

This NBS Handbook contains a three-part description of various aspects of the MUMPS computer programming language. Part I, the MUMPS Language Specification, consists of a stylized English narrative definition of the MUMPS language which was adopted and approved for publication as a Type A release of the MUMPS Development Committee on March 12, 1975. Part II, the MUMPS Transition Diagrams, represents a formal definition of the language described in Part I, employing a form of line drawings to illustrate syntactic and semantic rules governing each of the language elements; it was adopted and approved for publication as a Type A release of the MUMPS Development Committee on September 17, 1975. Part III, the MUMPS Portability Requirements, identifies constraints on the implementation and use of the language for the benefit of parties interested in achieving MUMPS application code portability; it was adopted and approved for publication as a Type A release of the MUMPS Development Committee on September 17, 1975.

A bibliography of other MUMPS Development Committee documents is included.

17. KEY WORDS *(six to twelve entries; alphabetical order; capitalize only the first letter of the first key word unless a proper name; separated by semicolons)*

Data handling language; interactive computing; interpretive computer programming language and operating system; medical automation; minicomputer-based systems; MUMPS; MUMPS Development Committee; MUMPS Language Standard.

18. AVAILABILITY ☒ Unlimited

□ For Official Distribution. Do Not Release to NTIS

☒ Order From Sup. of Doc., U.S. Government Printing Office
Washington, D.C. 20402, SD Cat. No. C13.11:118

□ Order From National Technical Information Service (NTIS)
Springfield, Virginia 22151

19. SECURITY CLASS (THIS REPORT)
UNCLASSIFIED

20. SECURITY CLASS (THIS PAGE)
UNCLASSIFIED

21. NO. OF PAGES
144

22. Price
$2.70

Time Was When Energy Was Cheaper

...but that's all changed now.

Today, increased energy costs—caused by increasing demand—boost the price of consumer goods, fuel inflation, and hurt our balance of payments.

That's why conserving energy is a must today, especially for you in the business and industry community which uses two thirds of our nation's energy.

Now . . . there's a book to tell you how to do it. It's called *EPIC—the Energy Conservation Program Guide for Industry and Commerce*. Prepared for you by the Commerce Department's National Bureau of Standards in cooperation with the Federal Energy Administration.

EPIC is a comprehensive handbook that can help you establish or improve an energy conservation program in your organization.

Not only industry and commerce but also hospitals, universities, research institutes, ANY organization that uses energy—and has to pay the bills—will want a copy of *EPIC*.

***EPIC* outlines in detail the steps in setting up an energy conservation program.**

It contains a checklist of more than 200 Energy Conservation Opportunities—suggestions to save energy in a dozen different areas.

EPIC's actual case histories of energy saving actions help you estimate your potential for saving energy—and reducing costs.

And *EPIC* contains much more—financial analysis procedures to evaluate projects, sources of information on energy conservation, engineering data and factors, an energy conservation bibliography.

Make *EPIC* part of your plan to control energy use and costs. Make *EPIC* part of your contribution to using energy resources wisely . . . so we can keep America working and growing.

To order your copy of *EPIC*, send $2.90 per copy (check, money order or Superintendent of Documents Coupons) to Superintendent of Documents, U.S. Government Printing Office, Washington, D.C. 20402. Ask for NBS Handbook 115, EPIC, C13.11:115. Discount of 25% on orders of 100 copies or more.

U.S. DEPARTMENT OF COMMERCE / National Bureau of Standards
FEDERAL ENERGY ADMINISTRATION / Conservation and Environment

NBS TECHNICAL PUBLICATIONS

PERIODICALS

JOURNAL OF RESEARCH reports National Bureau of Standards research and development in physics, mathematics, and chemistry. It is published in two sections, available separately:

- Physics and Chemistry (Section A)

Papers of interest primarily to scientists working in these fields. This section covers a broad range of physical and chemical research, with major emphasis on standards of physical measurement, fundamental constants, and properties of matter. Issued six times a year. Annual subscription: Domestic, $17.00; Foreign, $21.25.

- Mathematical Sciences (Section B)

Studies and compilations designed mainly for the mathematician and theoretical physicist. Topics in mathematical statistics, theory of experiment design, numerical analysis, theoretical physics and chemistry, logical design and programming of computers and computer systems. Short numerical tables. Issued quarterly. Annual subscription: Domestic, $9.00; Foreign, $11.25.

DIMENSIONS/NBS (formerly Technical News Bulletin)—This monthly magazine is published to inform scientists, engineers, businessmen, industry, teachers, students, and consumers of the latest advances in science and technology, with primary emphasis on the work at NBS. The magazine highlights and reviews such issues as energy research, fire protection, building technology, metric conversion, pollution abatement, health and safety, and consumer product performance. In addition, it reports the results of Bureau programs in measurement standards and techniques, properties of matter and materials, engineering standards and services, instrumentation, and automatic data processing.

Annual subscription: Domestic, $9.45; Foreign, $11.85.

NONPERIODICALS

Monographs—Major contributions to the technical literature on various subjects related to the Bureau's scientific and technical activities.

Handbooks—Recommended codes of engineering and industrial practice (including safety codes) developed in cooperation with interested industries, professional organizations, and regulatory bodies.

Special Publications—Include proceedings of conferences sponsored by NBS, NBS annual reports, and other special publications appropriate to this grouping such as wall charts, pocket cards, and bibliographies.

Applied Mathematics Series—Mathematical tables, manuals, and studies of special interest to physicists, engineers, chemists, biologists, mathematicians, computer programmers, and others engaged in scientific and technical work.

National Standard Reference Data Series—Provides quantitative data on the physical and chemical properties of materials, compiled from the world's literature and critically evaluated. Developed under a world-wide program coordinated by NBS. Program under authority of National Standard Data Act (Public Law 90-396).

NOTE: At present the principal publication outlet for these data is the Journal of Physical and Chemical Reference Data (JPCRD) published quarterly for NBS by the American Chemical Society (ACS) and the American Institute of Physics (AIP). Subscriptions, reprints, and supplements available from ACS, 1155 Sixteenth St. N. W., Wash. D. C. 20056.

Building Science Series—Disseminates technical information developed at the Bureau on building materials, components, systems, and whole structures. The series presents research results, test methods, and performance criteria related to the structural and environmental functions and the durability and safety characteristics of building elements and systems.

Technical Notes—Studies or reports which are complete in themselves but restrictive in their treatment of a subject. Analogous to monographs but not so comprehensive in scope or definitive in treatment of the subject area. Often serve as a vehicle for final reports of work performed at NBS under the sponsorship of other government agencies.

Voluntary Product Standards—Developed under procedures published by the Department of Commerce in Part 10, Title 15, of the Code of Federal Regulations. The purpose of the standards is to establish nationally recognized requirements for products, and to provide all concerned interests with a basis for common understanding of the characteristics of the products. NBS administers this program as a supplement to the activities of the private sector standardizing organizations.

Federal Information Processing Standards Publications (FIPS PUBS)—Publications in this series collectively constitute the Federal Information Processing Standards Register. Register serves as the official source of information in the Federal Government regarding standards issued by NBS pursuant to the Federal Property and Administrative Services Act of 1949 as amended, Public Law 89-306 (79 Stat. 1127), and as implemented by Executive Order 11717 (38 FR 12315, dated May 11, 1973) and Part 6 of Title 15 CFR (Code of Federal Regulations).

Consumer Information Series—Practical information, based on NBS research and experience, covering areas of interest to the consumer. Easily understandable language and illustrations provide useful background knowledge for shopping in today's technological marketplace.

NBS Interagency Reports (NBSIR)—A special series of interim or final reports on work performed by NBS for outside sponsors (both government and non-government). In general, initial distribution is handled by the sponsor; public distribution is by the National Technical Information Service (Springfield, Va. 22161) in paper copy or microfiche form.

Order NBS publications (except NBSIR's and Bibliographic Subscription Services) from: Superintendent of Documents, Government Printing Office, Washington, D.C. 20402.

BIBLIOGRAPHIC SUBSCRIPTION SERVICES

The following current-awareness and literature-survey bibliographies are issued periodically by the Bureau:

Cryogenic Data Center Current Awareness Service

A literature survey issued biweekly. Annual subscription: Domestic, $20.00; foreign, $25.00.

Liquefied Natural Gas. A literature survey issued quarterly. Annual subscription: $20.00.

Superconducting Devices and Materials. A literature survey issued quarterly. Annual subscription: $20.00. Send subscription orders and remittances for the preceding bibliographic services to National Bureau of Standards, Cryogenic Data Center (275.02) Boulder, Colorado 80302.

Electromagnetic Metrology Current Awareness Service

Issued monthly. Annual subscription: $24.00. Send subscription order and remittance to Electromagnetics Division, National Bureau of Standards, Boulder, Colo. 80302.