INTERNATIONAL STANDARD

ISO/IEC 11756

First edition
1992-12-15

Information technology – Programming languages – MUMPS

Technologies de l'information – Langages de programmation – MUMPS

Reference number
ISO/IEC 11756:1992 (E)

Table of Contents

Part 1: MUMPS Language Specification

Part 2: MUMPS Portability requirements

Foreword

ISO (the International Organization for Standardization) and IEC (the International Electrotechnical Commission) form the specialized system for worldwide standardization. National bodies that are members of ISO or IEC participate in the development of International Standards through technical committees established by the respective organization to deal with particular fields of technical activity. ISO and IEC technical committees collaborate in fields of mutual interest. Other international organizations, governmental and non-governmental, in liaison with ISO and IEC, also take part in the work.

In the field of information technology, ISO and IEC have established a joint technical committee, ISO/IEC JTC 1. Draft International Standards adopted by the joint technical committee are circulated to national bodies for voting. Publication as an International Standard requires approval by at least 75 % of the national bodies casting a vote.

International Standard ISO/IEC 11756 was prepared by American National Standards Institute (ANSI) (as ANSI/MDC X11.1-1990) and was adopted, under a special "fast-track procedure", by Joint Technical Committee ISO/IEC JTC 1, *Information technology*, in parallel with its approval by national bodies of ISO and IEC.

Appendices A and B of this International Standard are for information only.

Terminology and conventions

The text of American National Standard Institute ANSI/MDC X11.1-1990 has been approved for publication, without deviation, as an International Standard. Some terminology and certain conventions are not in accordance with the ISO/IEC Directives Part 3: "Drafting and presentation of International Standards"; attention is especially drawn to the following:

Wherever the word "standard" appears, referring to this International Standard, it should be read as "International Standard".

Cross reference

American National Standard	Corresponding International Standard
ANSI X3.4-1986	ISO/IEC 646:1991, *Information technology – ISO 7-bit coded character set for information interchange.*

Information technology – Programming languages – MUMPS

Part 1: MUMPS Language Specification

Introduction

Part 1 consists of two sections that describe the MUMPS language. Section 1 describes the metalanguage used in the remainder of Part 1 for the static syntax. Section 2 describes the static syntax and overall semantics of the language. The distinction between "static" and "dynamic" syntax is as follows. The static syntax describes the sequence of characters in a routine as it appears on a tape in routine interchange or on a listing. The dynamic syntax describes the sequence of characters that would be encountered by an interpreter during execution of the routine. (There is no requirement that MUMPS actually be interpreted). The dynamic syntax takes into account transfers of control and values produced by indirection.

1 Static Syntax Metalanguage

The primitives of the metalanguage are the ASCII characters. The metalanguage operators are defined as follows:

Operator	Meaning
::=	definition
[ ]	option
| |	grouping
...	optional indefinite repetition
L	list
V	value
SP	space

The following visible representations of ASCII characters required in the defined syntactic objects are used: SP (space), CR (carriage-return), LF (line-feed), and FF (form-feed).

In general, defined syntactic objects will have designators which are underlined names spelled with lower case letters, e.g., name, expr, etc. Concatenation of syntactic objects is expressed 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 to group elements or to make a choice of elements more readable.

Special care is taken to avoid any danger of confusing the square brackets in the metalanguage with the ASCII graphics] and [. 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 places the constraint on the syntactic object 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 SP L exampleargument
```

where

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

This example states: 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.

2 Static Syntax and Semantics

2.1 Basic Alphabet

The routine, which is the object whose static syntax is being described in Section 2, is a string made up of the following 98 ASCII symbols.

> The 95 printable characters, including the space character represented as SP, and also,
> the carriage-return character represented as CR,
> the line-feed character represented as LF,
> the form-feed character represented as FF.

See 2.4 for the definition of routine.

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

```
graphic  ::= any of the class of 95 ASCII printable characters, including
             SP.

nonquote ::= any of the characters in graphic except the quote character.

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.
```


2.2 Expression Atom expratom

The expression, expr, is the syntactic element which denotes the execution of a value-producing calculation; it is defined in 2.3. The expression atom, expratom, is the basic value-denoting object of which expressions are built; it is defined here.


```
                | lvn      |
expratom  ::=   | gvn      |
                | expritem |
```

See 2.2.2.1 for the definition of lvn. See 2.2.2.2 for the definition of gvn.

```
                | svn                  |
                | function             |
                | exfunc               |
                | exvar                |
expritem  ::=   | numlit               |
                | strlit               |
                | ( expr )             |
                | unaryop expratom     |
```

See 2.2.6 for the definition of svn. See 2.2.7 for the definition of function. See 2.2.10 for the definition of exfunc. See 2.2.9 for the definition of exvar. See 2.2.3 for the definition of numlit. See 2.2.5 for the definition of strlit. See 2.3 for the definition of expr.

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

2.2.1 Name name

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

See 2.1 for the definition of alpha and digit.

2.2.2 Variables

The MUMPS standard uses the terms *local variables* and *global variables* somewhat differently from their connotation in certain other computer languages. This section provides a definition of these terms as used in the MUMPS environment.

A MUMPS routine, or set of routines, runs in the context of an operating system process. During its execution, the routine will create and modify variables that are restricted to its process. It can also access (or create) variables that can be shared with other processes. These shared variables will normally be stored on secondary peripheral devices such as disks. At the termination of the process, the process-specific variables cease to exist. The variables created for long term (shared) use remain on auxiliary storage devices where they may be accessed by subsequent processes.

MUMPS uses the term *local variable* to denote variables that are created for use during a single process activation. These variables are not available to other processes. However, they are generally available to all routines executed within the process' lifetime. MUMPS does include certain constructs, the NEW command and parameter passing, which limit the availability of certain variables to specific routines or parts of routines. See 2.2.2.3 for a further discussion of variables and variable environments.

A *global variable* is one that is created by a MUMPS process, but is permanent and shared. As soon as it has been created, it is accessible to other MUMPS processes on the system. Global variables do not disappear when a process terminates. Like local variables, global variables are available to all routines executed within a process.

2.2.2.1 Local Variable Name lvn

```
lvn     ::=  | rlvn
             | @ expratom V lvn |
```

See 2.2 for the definition of expratom. See section 1 for the definition of V.

```
rlvn    ::=  | name [ ( L expr ) ]
             | @ lnamind @ ( L expr ) |
```

See 2.2.1 for the definition of name. See 2.3 for the definition of expr. See section 1 for the definition of L.

```
lnamind ::=  rexpratom V lvn
```

See section 1 for the definition of V.

```
rexpratom ::=  | rlvn
               | rgvn
               | expritem |
```

See 2.2.2.2 for the definition of rgvn. See 2.2 for the definition of expritem.

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.

When lnamind is present it is always a component of an rlvn. If the value of the rlvn is a subscripted form of lvn, then some of its subscripts may have originated in the lnamind. In this case, the subscripts contributed by the lnamind appear as the first subscripts in the value of the resulting rlvn, separated by a comma from the (non-empty) list of subscripts appearing in the rest of the rlvn.

2.2.2.2 Global Variable Name gvn

```
gvn      ::=  | rgvn                          |
              | @ expratom V gvn              |
```

See 2.2 for the definition of expratom. See section 1 for the definition of V.

```
rgvn     ::=  | ^( L expr )                   |
              | ^name [ ( L expr ) ]          |
              | @ gnamind @ ( L expr )        |
```

See 2.3 for the definition of expr. See 2.2.1 for the definition of name. See section 1 for the definition of L.

```
gnamind  ::=  | rexpratom V gvn               |
```


See section 1 for the definition of V.

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. An abbreviated form of subscripted gvn is permitted, called the *naked reference*, in which the prefix is present but 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.

When gnamind is present it is always a component of an rgvn. If the value of the rgvn is a subscripted form of gvn, then some of its subscripts may have originated in the gnamind. In this case, the subscripts contributed by the gnamind appear as the first subscripts in the value of the resulting rgvn, separated by a comma from the (non-empty) list of subscripts appearing in the rest of the rgvn.

Every executed occurrence of gvn affects the naked indicator as follows. If, for any positive integer m, the gvn has the nonnaked form

$$N(v_1 , v_2 , \ldots , v_m)$$

then the m-tuple $N, v_1 , v_2 , \ldots , v_{m-1}$, is placed into the naked indicator when the gvn reference is made. A subsequent naked reference of the form

$$\hat{}(s_1 , s_2 , \ldots , s_i) \qquad (i \text{ positive})$$

results in a global reference of the form

$$N(v_1 , v_2 , \ldots , v_{m-1} , s_1 , s_2 , \ldots , s_i)$$

after which the $m+i$-1-tuple N , v_1 , v_2 , ... , s_{i-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. A nonnaked reference without subscripts leaves the naked indicator undefined.

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 |
```

See 2.2.2.1 for the definition of lvn.

2.2.2.3 Variable Handling

MUMPS has no explicit declaration or definition statements. Local and global variables, both non-subscripted and subscripted, are automatically created as data is stored into them, and their data contents can be referred to once information has been stored. Since the language has only one data type - string - there is no need for type declarations or explicit data type conversions. Array structures can be multidimensional with data simultaneously stored at all levels including the variable name level. Subscripts can be positive, negative, and/or noninteger numbers as well as nonnumeric strings (other than empty strings).

In general, the operation of the local variable symbol table can be viewed as follows. Prior to the initial setting of information into a variable, the data value of that variable is said to be undefined. Data is stored into a variable with commands such as SET, READ, or FOR. Subsequent references to that variable return the data value that was most recently stored. When a variable is killed, as with the KILL command, that variable and all of its array descendants (if any) are deleted, and their data values become undefined.

No explicit syntax is needed for a routine or subroutine to have access to the local variables of its caller. Except when the NEW command or parameter passing is being used, a subroutine or called routine (the callee) has the same set of variable values as its caller and, upon completion of the called routine or subroutine, the caller resumes execution with the same set of variable values as the callee had at its completion.

The NEW command provides scoping of local variables. It causes the current values of a specified set of variables to be saved. The variables are then set to undefined data values. Upon returning to the caller of the current routine or subroutine, the saved values, including any undefined states, are restored to those variables. Parameter passing, including the DO command, extrinsic functions, and extrinsic variables, allows parameters to be passed into a subroutine or routine without the callee being concerned with the variable names used by the caller for the data being passed or returned.

The formal association of MUMPS local variables with their values can best be described by a conceptual model. This model is NOT meant to imply an implementation technique for a MUMPS processor.

The value of a MUMPS variable may be described by a relationship between two structures: the NAME-TABLE and the VALUE-TABLE. (In reality, at least two such table sets are required, one pair per executing process for process-specific local variables and one pair for system-wide global variables.) Since the value association process is the same for both types of variables, and since issues of scoping due to parameter

passing or nested environments apply only to local variables, the discussion that follows will address only local variable value association. It should be noted, however, that while the overall structures of the table sets are the same, there are two major differences in the way the sets are used. First, the global variable tables are shared. This means that any operations on the global tables, e.g., SET or KILL, by one process, affect the tables for all processes. Second, since scoping issues of parameter passing and the NEW command are not applicable to global variables, there is always a one-to-one relationship between entries in the global NAME-TABLE (variable names) and entries in the global VALUE-TABLE (values).

The NAME-TABLE consists of a set of entries, each of which contains a name and a *pointer*. This pointer represents a correspondence between that name and exactly one DATA-CELL from the VALUE-TABLE. The VALUE-TABLE consists of a set of DATA-CELLs, each of which contains zero or more tuples of varying degrees. The degree of a tuple is the number (possibly 0) of elements or subscripts in the tuple list. Each tuple present in the DATA-CELL has an associated data value.

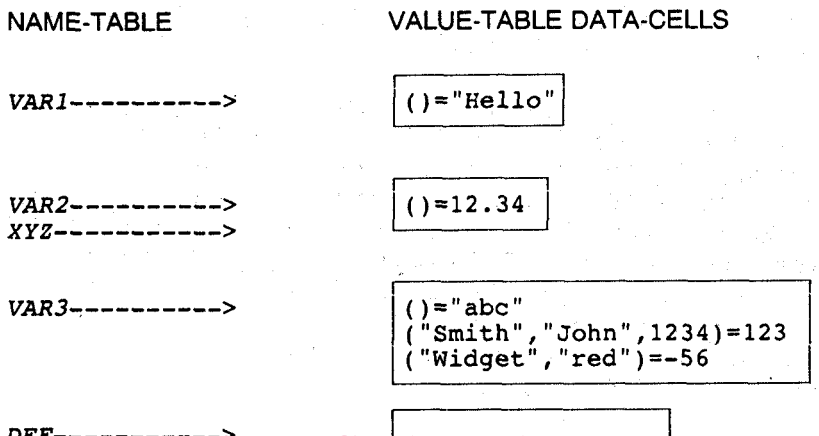
The NAME-TABLE entries contain every non-subscripted variable or array name (name) known, or accessible, by the MUMPS process in the current environment. The VALUE-TABLE DATA-CELLs contain the set of tuples that represent all variables currently having data-values for the process. Every name (entry) in the NAME-TABLE refers (*points*) to exactly one DATA-CELL, and every entry contains a unique name. Several NAME-TABLE entries (names) can refer to the same DATA-CELL, however, and thus there is a many-to-one relationship between (all) NAME-TABLE entries and DATA-CELLs. A name is said to be *bound* to its corresponding DATA-CELL through the *pointer* in the NAME-TABLE entry. Thus the pointer is used to represent the correspondence and the phrase *change the pointer* is the equivalent to saying *change the correspondence so that a name now corresponds to a possible different DATA-CELL (value)*. NAME-TABLE entries are also placed in the PROCESS-STACK (see 2.2.2.4).

The value of an unsubscripted lvn corresponds to the tuple of degree 0 found in the DATA-CELL that is bound to the NAME-TABLE entry containing the name of the lvn. The value of a subscripted lvn (array node) of degree n also corresponds to a tuple in the DATA-CELL that is bound to the NAME-TABLE entry containing the name of the lvn. The specific tuple in that DATA-CELL is the tuple of degree n such that each subscript of the lvn has the same value as the corresponding element of the tuple. If the designated tuple doesn't exist in the DATA-CELL then the corresponding lvn is said to be *undefined*.

In the following figure, the variables and array nodes have the designated data values.

VAR1 = "Hello"
VAR2 = 12.34
VAR3 = "abc"
VAR3("Smith","John",1234)=123
VAR3("Widget","red") = -56

Also, the variable *DEF* existed at one time but no longer has any data or array value, and the variable *XYZ* has been *bound* through parameter passing to the same data and array information as the variable *VAR2*.



The initial state of a MUMPS process prior to execution of any MUMPS code consists of an empty NAME-TABLE and VALUE-TABLE. When information is to be stored (*set*, *given*, or *assigned*) into a variable (lvn):

a. If the name of the lvn does not already appear in an entry in the NAME-TABLE, an entry is added to the NAME-TABLE which contains the name and a pointer to a new (empty) DATA-CELL. The corresponding DATA-CELL is added to the VALUE-TABLE without any initial tuples.

b. Otherwise, the pointer in the NAME-TABLE entry which contained the name of the lvn is extracted. The operations in step c. and d. refer to tuples in that DATA-CELL referred to by this pointer.

c. If the lvn is unsubscripted, then the tuple of degree 0 in the DATA-CELL has its data value replaced by the new data value. If that tuple did not already exist, it is created with the new data value.

d. If the lvn is subscripted, then the tuple of subscripts in the DATA-CELL (i.e., the tuple created by dropping the name of the lvn; the degree of the tuple equals the number of subscripts) has its data value replaced by the new data value. If that tuple did not already exist, it is created with the new data value.

When information is to be retrieved, if the name of the lvn is not found in the NAME-TABLE, or if its corresponding DATA-CELL tuple does not exist, then the data value is said to be undefined. Otherwise, the data value exists and is retrieved. A data value of the empty string (a string of zero length) is not the same as an undefined data value.

When a variable is deleted (*killed*):

a. If the name of the lvn is not found in the NAME-TABLE, no further action is taken.

b. If the lvn is unsubscripted, all of the tuples in the corresponding DATA-CELL are deleted.

c. If the lvn is subscripted, let *N* be the degree of the subscript tuple formed by removing the name from the lvn. All tuples that satisfy the following two conditions are deleted from the corresponding DATA-CELL:

1. The degree of the tuple must be greater than or equal to *N*, and

2. The first *N* arguments of the tuple must equal the corresponding subscripts of the lvn.

In this formal language model, even if all of the tuples in a DATA-CELL are deleted, neither the DATA-CELL nor the corresponding names in the NAME-TABLE are ever deleted. Their continued existence is frequently required as a result of parameter passing and the NEW command.

2.2.2.4 Variable Contexts

The organization of multiple variable contexts requires the use of a PROCESS-STACK. This is a simple push-down stack, or last-in-first-out (LIFO) list, used to save and restore items which control the execution flow or variable environment. Five types of items, or frames, will be placed on the PROCESS-STACK, DO frames, exfunc frames, exvar frames, NEW frames, and parameter frames:

a. DO frames contain the execution level and the execution location of the doargument. In the case of the argumentless DO, the execution level, the execution location of the DO command and a saved value of $T are saved. The execution location of a MUMPS process is a descriptor of the location of the command and possible argument currently being executed. This descriptor includes, at minimum, the routinename and the character position following the current command or argument.

b. Exfunc and exvar frames contain saved values of $T, the execution level, and the execution location.

c. NEW frames contain a NEW argument (newargument) and a set of NAME-TABLE entries.

d. Parameter frames contain a formallist and a set of NAME-TABLE entries.

2.2.3 Numeric Literal numlit

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

```
intlit  ::=  digit ...
```

See 2.1 for the definition of digit.

The numeric literal numlit is defined as follows.

```
numlit  ::=  mant [ exp ]
```