MUMPS

The MUMPS language originated in the late 1960s, and has evolved since that time into an important programming language that is used in medical, financial, and other application areas such as managing most containerized shipping activities worldwide. MUMPS is an acronym; the first letter stands for Massachusetts General Hospital, where the language got its start. The remaining letters stand for Utility Multi-Programming System. In recent years the term M has been used instead of MUMPS. We will follow that convention in this article.

M was designed as an interpreted language. Currently there are both interpretive and precompiled versions. Some of the features that distinguish M from other programming languages include (1) provision for persistent, shared data, defined using a hierarchical sparse-array structure that allows for noninteger subscripts; (2) a powerful set of complex string manipulation operators and functions; and (3) late binding constructs that enable run-time definition of storage locations and treatment of data as code. M was first accepted as an American National Standards Institute (*ANSI*) standard language in 1977; it is now both an ANSI and an International Standard Organization (*ISO*) standard available on a wide range of computers.

The Evolution of M

Figure 1 depicts the evolution of M (1). The need for interactive computing languages became apparent during the 1960s. Research at the Rand Corporation led to some early interactive languages and operating systems, including JOSS and later JOVIAL (an acronym for Jules' Own Version of Interactive ALGOL, created by Jules Schwartz at the Rand Corporation). These early efforts were taken over by Bolt, Beranek, and Newman (*BBN*) to create a commercial interactive system called TELCOMP. In an effort to improve the text manipulation features of this system, an experimental language called STRINGCOMP was created by BBN. in the mid-1960s.

Researchers at the Laboratory of Computer Science, a computing arm of Massachusetts General Hospital, the principal teaching hospital of Harvard University's medical school, initiated a research effort to design a language that would serve the needs of a hospital information system. They based much of their early work on the concepts embodied in STRINGCOMP, but added to it a number of other features that enabled the language to meet the perceived needs of a hospital system. These needs included timely response in an interactive system, sharing of data, and a file structure that was suitable for database storage of complex hierarchical files, largely textual in nature. The language design and implementation led to the creation of the first version of MUMPS by 1968 (2). This version was implemented under its own operating system on an early minicomputer (PDP-9), had in memory resident partitions of approximately 4 kbit for four users who shared buffers for persistent data. This design made it highly responsive to interactive users. It supported an early hospital information system at Massachusetts General Hospital.

Researchers at Massachusetts General Hospital shared tapes of this early version of MUMPS, and many recipients took the language and added new features to it. Digital Equipment Corporation, on whose computers

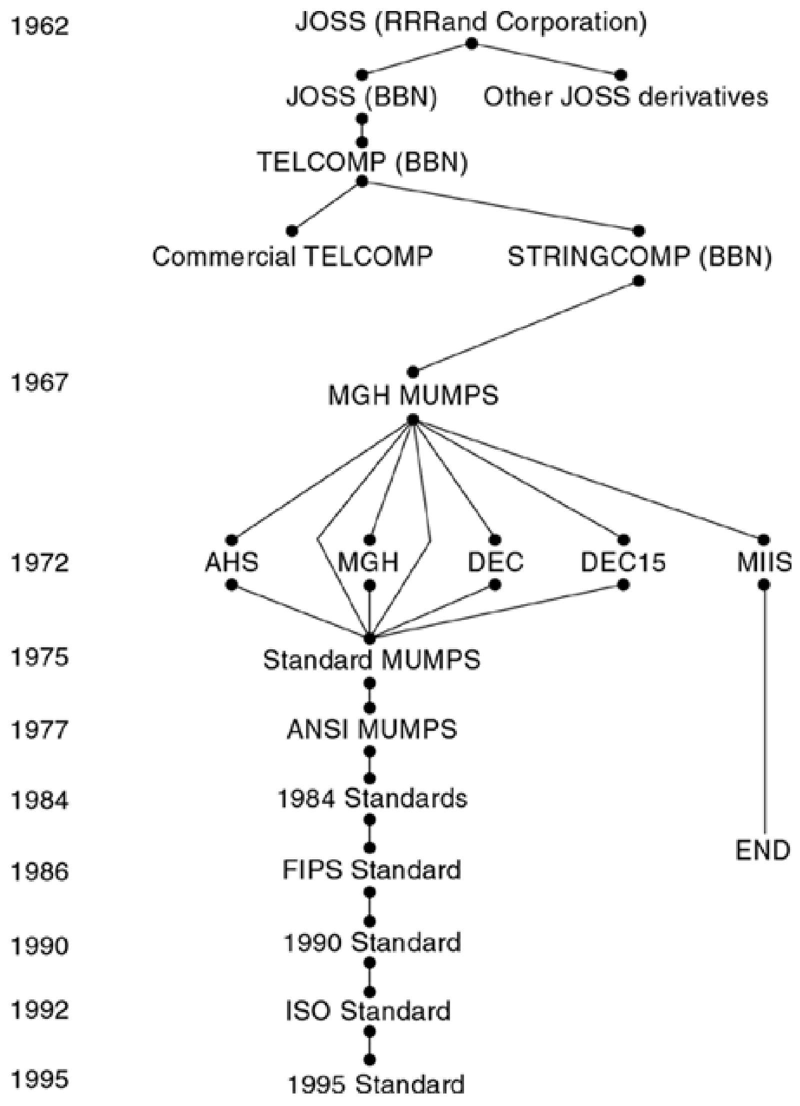

Fig. 1. Evolution of MUMPS. [Courtesy of Digital Equipment Corporation (1).]

the language ran, developed versions specific to several different PDP models. By 1972, there were eight major dialects with as many as 14 subdialects (3), all essentially incompatible with one another.

Standardization of M. During the period 1967–1972, the National Institutes of Health supported the Laboratory of Computer Science at Massachusetts General in the development of several clinical applications. Because of the lack of any standards for the language, these applications, once completed, ran only at the originating site. In an effort to provide wider use of these applications and of the MUMPS language, the National Institutes of Health, together with the National Bureau of Standards, sponsored an unusual experiment, inviting users and developers of the various dialects of MUMPS to come together and define a single standard, which would then be used by all participants. This process began late in 1972, and by September 1975, this group, which became the MUMPS Development Committee, succeeded in defining a new standard. With assistance from the National Bureau of Standards, this standard was submitted to ANSI as a new programming-language standard. Approval by ANSI was given in September 1977, and MUMPS became ANS X11.1 Programming Language: MUMPS. With one exception, all implementors of previous MUMPS dialects adopted the new standard. With federal support, programs were made available to translate earlier dialects into the new standard, facilitating its smooth adoption.

The initial standard underwent periodic review and revision, resulting in new versions of the standard, which were approved in 1984, 1990, and 1995 by ANSI. The 1990 ANSI standard was accepted by the ISO as ISO/IEC 11756:1992, and the current version of the standard is ISO/IEC 11756:1999.

With the adoption of a standard version of M, interest in the language expanded. Most new implementations no longer required that M run under a dedicated operating system, making it more widely available. However, the PDP-11 version remained embedded in a dedicated operating system, limiting its use in that environment. This fact may be an important reason for the lack of acceptance of M by computer science departments.

With the appearance of the standard, M versions appeared on a large number of different vendors' systems, and a number of commercial versions supported were available. By the mid-1980s, versions running on personal computers also became widespread, and the availability of the language greatly expanded. M became widely used in Europe and Japan, and was one of the main languages used in banking, television, and other industries in Brazil. Its use has since expanded to other areas, including China.

Principal Characteristics of M

M behaves like an interpreted language, with a number of features that are unlike other programming languages. It was designed to support hospital information systems, and many of the unique features are derived from these design constraints. In this section, we review some of the important features of the language that are different from most other programming environments.

Basic Characteristics. M runs in a multiuser environment. Modifications to shared data become immediately available to group members. Buffers are also shared by users logged on to an M environment. As a result, M has built-in persistent (shared) variables, which, in this language, are called *global variables*, or *globals*. This is unlike the normal use of that term, and signifies that a global variable (whose name is identified by a leading caret ^) is placed temporarily in the shared buffer space and then stored on disk as buffers are cleared. *Local variables* are those variables created by a user, stored within the user's partition, and deleted when that user logs off. These variables are global in scope for that partition, meaning they can be accessed by any routine running in the partition.

Another unusual feature of the language relates to its data type and structure. All data are of type *variable length string,* (The next revision of M will probably contain a second data type to refer to objects) dynamically created, and require no declarations. Data are stored in sparse arrays that may be defined in hierarchical format. Subscripts may include noninteger values: floating-point numbers and even strings may be used as subscripts, giving the subscript itself meaning. Arrays are stored using either a default collating sequence based on canonical numbers followed by code values of characters in the subscript or according to optional collation algorithms suitable for alphabetizing English or non-English character sets in accordance with local cultural usage in different countries. (In normal code-value collation of ASCII codes, upper- and lowercase letters do not collate according to alphabetization when they are mixed in strings.)

Syntax and Parsing. (In this section, M commands will be written in uppercase, variables in lowercase. Neither is a requirement of the language. Commands can be abbreviated to their first unique letter or letters, and they are not case-sensitive. Variable names *are* case-sensitive.)

M was originally designed as an interpreted programming language, and it contains some elements that require dynamic execution based on content of variables and pointers. Declarations are not required to create variables or arrays (although a declaration is allowed to limit the scope of a variable). The syntax of M is simple, but it requires correct use of spaces (*sp*) to preserve that syntax. In the most general form, M syntax consists of the following:

$$[label]\ linestart\ [command]\ [sp\ command\ldots]\ [;comments]$$

where the elementary syntax of a command is *command sp argument* [,*argument*] . . . . Labels are optional, as are commands or comments. A *linestart* may be one or more spaces. Spaces are required to separate commands and to separate commands from their arguments. If a command has no argument, *two spaces* are required before the next command. Examples of M command lines are:

```
        SET x = 3.1414, y = "hello, world"    [no spaces between arguments]
label   ;comment(s)    [a command line need not include commands]
        IF x > 3 WRITE "x is greater than 3"
        ELSE  WRITE  "x is not greater than 3"]
        [two spaces after ELSE before next command]
```

A command line in M may be up to 255 characters long.

Parsing M commands is done strictly from left to right, with no precedence of operators. For instance,

$$\text{WRITE}\ \ 5 + 3^*2 - 8/4$$

would yield 2, not 9 as might be the case in most other programming languages. To get the answer 9, one would issue the command WRITE 5+(3∗2)-(8/4). By the same reasoning, an IF statement that has two arguments requires that the second be parenthesized:

$$\text{IF}\ \ x > 3\,\&\,(y < 27)\ldots$$

If the parentheses were not present, M would take the truth value of y (zero or nonzero) before evaluating the remainder of that expression.

The execution control of conditionals (IF, ELSE) and the execution flow control command FOR apply to all commands on the remainder of the line.

Since there is currently only one data type in M, the language permits implicit treatment of variables in a left-to-right interpretation of numerics versus nonnumerics. For instance, if we set the value of x to "123abc" then WRITE x∗2 will yield 246, since M will ignore all characters in the string beyond the last digit.

Review of Basic Language Elements

In this section, we review the basic elements of the language as a reference for discussion in other sections.

Character set. M was originally implemented using the 7-bit ASCII standard character set. Although 7-bit ASCII can still support all language capabilities, the language now supports a number of formalized character sets, and even the use of multiple character sets simultaneously.

Commands. The set of commands available in M is small, but adequate for general-purpose programming. The set is small enough that a single letter suffices to identify uniquely most of the commands. This feature makes it possible to refer to commands by their first letter, so that S x="hello, world" W x is equivalent to SET x="hello, world" WRITE x.

Commands available in M may be grouped as follows:

I/O	READ, WRITE, OPEN, USE, CLOSE
Assignment	SET, READ, KILL, MERGE
Conditional	IF, ELSE
Execution flow control	DO, FOR, GOTO, HALT, HANG, QUIT, JOB, XECUTE
Variable Scoping	NEW [limits scope of variable(s) to a subroutine]
Synchronization	LOCK, TSTART, TCOMMIT, TROLLBACK, TRESTART
Other	BREAK, VIEW, Z commands (implementation-specific extensions, not discussed here)

A few of these commands require additional comment. READ includes an implicit (WRITE) in order to prompt the user for the desired information:

```
READ "Enter your Social Security No:",ssno
```

will prompt the user with the request in quotes, and assign the user's input to the variable ssno. The READ and WRITE commands can be modified with format controls: ! takes output to a new line prior to issuing the prompt; ?*nn* moves the output of the next character to the column number specified by *nn*, and # moves to a new page.

Both READ and WRITE can be modified to input or output a single numeric code. WRITE ∗51 (defined as implementation-specific) will usually output the numeral 3 (the character represented by the ASCII code value 51). READ ∗z will store the value 51 in the variable z if the user types the numeral 3. One useful application of the WRITE ∗*nn* option is to output codes specific to certain terminal operations, such as cursor movement or clearing the screen.

In another variation, READ can be modified to limit the number of characters entered. READ x#5 will accept the first five characters typed in by the user, then terminate the READ command.

The READ command can also be modified with a *timeout* specification. READ x:15 instructs the system to wait up to 15 s for the user to respond before aborting the read. (Note: while these options are powerful, they also severely compromise the implementation of M with these features under operating systems that include front-end processors activated either by block transmit or the [Enter] key.)

OPEN, USE, and CLOSE are used in conjunction with files and physical devices, directing I/O to the specified file or device.

LOCK is used to indicate to other users attempting to Lock a given variable or array node (and its descendants) that they are "owned" by another user until released. Locks can be added to incrementally. A *timeout* option can be added to the lock command, allowing a process to recover from an attempt to lock a variable that is owned by another user.

MERGE is described in a later section dealing with arrays.

FOR keeps track of iterations of a command line. FOR i=1:1:10 SET sum=sum+x(I) will go through an array and add the values of x(1), x(2), . . ., x(10) to sum.

DO transfers control to a subroutine, returning control of execution to the position after the DO when that operation is complete.

HANG 10 tells the computer to pause 10 s before proceeding with execution. HALT exits the user's session. It accepts no arguments.

JOB allows M to spawn a separate job, independent of the one currently executing.

XECUTE allows the contents of a variable to be treated as executable M code. For instance, consider these two command lines:

```
SET x="WRITE 3*2"
XECUTE x
```

The second line will result in displaying the value 6 on the output device.

Operators. M offers several different kinds of operators, some of which are not found in other languages. They may be grouped into categories of mathematical, logical, relational, string conditional, and indirection as well as unary and binary.

Mathematical. M uses the standard operators for arithmetic (+, -, ∗, /). In addition, M provides three other operators not found in all programming languages:

\ Integer divide; returns only the integer portion of a divide operation.

# Modulo divide: for positive numbers, returns the remainder after an integer division based on cyclical subdivision of the number (its effect when nonpositive numbers are involved is complicated; see Ref. 4.

∗∗ Exponentiation: uses the same conventions for exponentiation found in several other languages (e.g., FORTRAN).

+ and - The unary operators + and - are used to force numeric interpretation of mixed strings that may contain nonnumeric or noncanonic numbers. For instance, +01483 will cause the value to be treated as 1483. The unary negative operator forces a negative interpretation of a given value. If x=6, then +x yields 6 and −x yields −6, but if x = −6, −x yields 6, whereas +x retains the value of −6.

Logical. M uses the following logical operators:

&	AND
!	OR
’	NOT (apostrophe)

Relational. The standard symbols =, <, and > are used as relational operators. They may be combined with the relational NOT operator to provide a complete set of relational possibilities.

String. The ability of M to manipulate strings effectively is aided by string operators. They include:

=	String equals operator
_(underline)	Concatenate
[	Contains
]	Follows (based on code values of characters in the string)
]]	Sorts after (used in conjunction with special collation algorithms)
?	Pattern match

Of these operators, the last two require explanation. M allows implementors to define alternative collation sequences to the default use of ASCII code values. This option is useful in sorting upper- and lowercase strings, strings with characters having diacritical accents such as é, ñ, etc., or strings using other character sets such as Greek, Russian, or east Asian. The *sorts after* operator]] uses the algorithm currently in use for collation and returns a truth value relating two strings based on that algorithm.

The pattern match operator is a powerful language element (often taking up a significant portion of the interpreter code to process) that allows users to specify specific patterns in strings. The operator can use any of the following definitions (based on ASCII codes):

A	All letters of the alphabet
U	Uppercase alphabetic characters
L	Lowercase
N	Numeric
P	Graphic symbols other than alphabetic or numeric
C	Control characters (of the ASCII code set)
E	Any character
“ . . . ”	A string delimited by quotation marks

To assign counts to these patterns, this operator allows numeric prefixes using numerics augmented by a period:

?3N	Three numerics
?1.3U	From one to three uppercase letters
?.4A	From zero to four alphabetic characters
?1C.E	One control character followed by any number of other characters

The syntax also permits alternate patterns to be specified. For instance, to check if a date has been entered using either two or four digits for the year (e.g., 3/12/1957), the pattern match to verify this format is ?1.2N1“/”1.2N1“/”(2N,4N) which will allow the user one or two digits for day and month, respectively, separated by slashes, followed by either a two- or a four-digit year. The pattern match is extremely helpful in data input verification.

Conditional. The M postconditional operator : can be used at the command level or the argument level to use the truth value of an expression to determine a specific action. A common command level example is QUIT:x=“”, which tells M to quit the current execution flow if the value of x is the *empty string*. An example of argument-level use of the postconditional operator is DO label1:ans=1,label2:ans=2,label3:ans=3, which would evaluate the current value of the variable ans and call the subroutine starting with the label matching that value.

Indirection. The *indirection* operator @ can be thought of as a *pointer* that can be used at both the name and the argument level. An example of name-level indirection is the following continuation of the discussion of the XECUTE command:

```
SET x = “WRITE 3*2”
SET y = “x”
WRITE y (yield, “x”)
WRITE @y (yields “WRITE 3*2”)
XECUTE @y(yields “6”)
```

The use of indirection is further illustrated in a code example in a later section of this article.

Variables. All variables are of the single data type *variable length string*, although they can be operated on numerically. Although the standard sets a portability limit for the length of strings that can be exchanged between implementations, some implementations allow for variable strings to be much longer, facilitating incorporation of nontextual elements such as graphics and sound to be stored as M variables.

M uses the term *local variables* to signify variables that are declared in a user's workspace for use by that session. Local variables are available to all routines operating in that workspace (unless explicitly masked by a NEW command), and they disappear when the session terminates. *Global variables*, once created by a user, are available to other users within the same administrative group as soon as created, and are stored on disk, remaining available after a user session terminates. Like local variables, they are available to all routines in authorized users' workspaces, but they are not affected by the NEW command. Global-variable names are identified by a preceding caret (ˆ) in command lines. Both local- and global-variable names are case-sensitive and can use up to eight characters for unique identification.

Arrays in M are hierarchical, sparse arrays that can be identified by integer, floating-point, or string subscripts. The depth of subscripting is limited by the size of the string required to define the complete set of subscripts; M's portability requirements currently permit up to 255 characters to be used in describing the name, its subscripts, and associated punctuation. Arrays are typically stored in *B trees* using key compression techniques that facilitate use of meaningful information to be included in the subscript. Manipulation of subscripts is enabled through a set of special functions described in a later section.

Variable Scoping: the NEW Command. The NEW command in M allows local variables to be considered *private* to a given subroutine. NEW a,b,c stores the current values of variables a, b, and c until that subroutine is terminated. NEW(a,b,c) stores values of all variables *except* those listed within parentheses. This command facilitates use of subroutines in general-purpose code where some variables might have been defined elsewhere.

Special Variables. The M language defines several variables maintained by the system, identified by their first character $. Some are not standardized and are not described herein. A few of the others merit comment. They include the following:

$IO identifies the active device accepting I/O from M.

The value of $HOROLOG consists of two parts. The first value represents the number of complete days that have elapsed since December 31, 1840, a date selected to permit identification of patients aged up to 120 years old when the language was originally defined. The second portion of $HOROLOG gives the number of seconds elapsed since midnight of the current day. For instance, a $HOROLOG value of 58088,57054 represents January 15, 2000 at approximately 3:50 PM.

$X and $Y are used to indicate the current cursor position on a display device. Unlike other system special variables, these values can be set by the user so as to reposition a cursor at a specified location. $TEST stores the truth value of the most recent IF command or timeouts. It is used with subsequent ELSE or argumentless IF commands to avoid repeating the expression evaluated.

Functions. The principal functions provided in M are designed to facilitate the manipulation of strings in M's sparse array structure. These functions are described below.

$RANDOM is used to generate a pseudorandom number within a range specified by the argument: $RANDOM(100) will return an integer value between 0 and 99 inclusive. Different implementations use differing techniques to generate this value.

$SELECT is a function that parses options in a left-to-right sequence and selects the first value based on truth of the expression encountered. The following command illustrates its use:

$$\text{SET } y = \$\text{SELECT}(x < 1 : 0, x = 1 : 1, 1 : \text{“error”})$$

If, for example, x = 2, then x<1 will result in assigning a 0 value to y; if x=1, y is set to 1; otherwise (i.e., no matter what other values may be assigned to x), the variable y is set to the string “error” as a default when no other true expressions are encountered.

String Manipulation in M

M was designed to facilitate manipulation of text data. The string operators described above (*concatenate, contains, follows, sorts after*, and *pattern match*) provide one way in which these strings can be manipulated. There are in addition several functions that provide even more flexibility in string manipulation. These functions are briefly summarized below.

The first pair of functions, $CHAR and $ASCII, allow users to work with code values of characters, using the default ASCII string. $CHAR(*code-value*) returns the character specified by *code-value*, using the ASCII character string as the default. WRITE $CHAR(65) displays A on the output device, whereas WRITE $CHAR(7) will cause a bell to sound. $ASCII performs the reverse function, returning the code value of a given character: $ASCII(“A”) returns 65.

Two functions not described in detail are $JUSTIFY and $FNUMBER, used to format output in accordance with several different cultural specifications used in numeric tables and financial statements. $JUSTIFY (*variable-name*, integer1,[integer2]) will right-justify a number specified by *variable-name* within integer1 spaces, optionally including a decimal and integer2 decimal place values. $FNUMBER provides for special formats such as negative signs or parentheses around negative values, etc.

Other string functions allow more sophisticated operations on string variables. Let us assume that x has the current value “August 6, 1955” and see how this value might be manipulated.

$LENGTH(variable-name) returns the total length of the contents of variable-name. $LENGTH(x) in this case returns a value of 14. The second form of $LENGTH returns a *count* of the number of occurrences specified by the second argument, plus 1. $LENGTH(x, “ ”) returns a value of 3 (there are two spaces in the contents of x as defined in the previous paragraph). This function is helpful in such tasks as developing wraparound algorithms required in word processing.

$FIND(*variable-name,string*,[*start-at*]) returns the position immediately following the location of *string* in *variable-name* optionally starting after position *start-at*. For example, $FIND(x,“ ”,7) will search for the first space after the seventh character in the string x, returning a value of 11, since the next space is the tenth character in the string.

$EXTRACT(*variable-name,start*,[*end*]) retrieves the characters in *variable-name* beginning with *start* position and (optionally) ending with *end* position or the end of the string, whichever occurs first. Thus, $EXTRACT(x,1,3) returns Aug, and $EXTRACT(x,11,99) returns 1955 (the end value is set arbitrarily large to make sure all characters after *start* are included).

$PIECE(*variable-name,delimiter,starting-position*,[*ending-position*]) returns all characters in *variable-name* lying between the specified occurrences of delimiter, beginning with *starting-position* and optionally ending with *ending-position*. Thus, $PIECE(x,“ ”,3) returns 1955, and $PIECE(x,“ ”,1,2) returns August 6, (comma included), with delimiters included when two pieces are called for.

These functions can be nested. For instance, WRITE $EXTRACT($PIECE(x,“ ”,2),1) returns the value 6 (the first character in the second piece of x, using a space as a delimiter).

A different kind of string manipulation function is provided by $TRANSLATE(*variable-name,char1,char2*), which replaces in *variable-name* every occurrence of characters defined in *char1* with the corresponding character in *char2* based on their positions. For instance, if x = “08/06/55”, then $TRANSLATE(x,“/”,“-”) results in x having the new value 08-06-55. To convert a string to uppercase, one could write

$$\text{SET } \textit{string} = \text{\$TRANSLATE}(\textit{string}, \text{“abcdefg} \ldots \text{yz”}, \text{“ABCDEFG} \ldots \text{YZ”})$$

$REVERSE(*string*) reverses the order of characters in a string. (This can be done by using $TRANSLATE, but it is much more easily done with this function.)

Taken together, the string manipulation operators and functions facilitate powerful data manipulation of textual data.

The last string function, $TEXT, is used in manipulating string of text in M routines (programs). $TEXT(*label*+*offset*) returns the entire line specified by *label* and *offset* by a specific integer number of lines. This function is illustrated in an example of an M routine produced near the end of this article.

Manipulation of M Sparse Arrays

Arrays in M are specified by subscripts, which can be integer, decimal, or string. This flexibility allows much more meaningful use of subscripts to convey real meaning. If the variable date has the value “2000/1/5” and id has the value “978372-W”, then one can write SET ˆpt(id,date,“medication”) =“Tylenol PRN”. This flexibility carries with it, however, the need for functions that will enable users to manipulate the value stored in this manner.

The M command MERGE allows users to copy subtrees of an array into another array. For instance, to transfer or edit the record of a single patient, one can say MERGE ˆtemp(id)=ˆpt(id), which will transfer all data pertaining to the patient with that id number to a temporary global array.

A more difficult problem in manipulating sparse arrays relates to uncertainty as to what values might be stored under what subscripts. Since arrays are not declared, their size is unpredictable, as are the number of subscripts used in an array and whether each node has a value or serves as a pointer to the next level. In the previous example of ˆpt(id,date,“medication”), for example, there need be no value stored under the subscript date. M provides solutions to these problems with a series of functions, beginning with $DATA and $ORDER. $DATA(*variable-name*(*subscript*(*s*))) returns one of the following truth values:

0	No data, no descendants in the array (i.e., the variable does not exist).
1	Data value is present, but no descendants.
10	No data value is stored at this node, but there are descendants.
11	Data value is present and there are descendants.

Consider the following sparse array:

```
ˆpt(id) = “Lastname, FirstnameˆStreetˆCityˆStateˆZip”
ˆpt(id,date,“medications”) = “Aspirin”
```

In this example, the value of $DATA(ˆpt(id,date)) would be 10, assuming that no value is stored at that level, but that there are one or more subscripts representing data stored for that patient on that date. The value of ˆpt(id) would be 11, since the node has both data and descendants. The value of the complete reference ˆpt(id,date,“medication”) would be 1, since data are stored, but there are no further descendants.

$ORDER(*variable-name*(*subscript*(*s*))) returns a value representing the next subscript at the level specified, beginning and terminating with an empty string (“”). In the previous example, $ORDER(pd(id,“”)) would return the first date in the array for that patient id number. If one wishes to iterate through all date values for a specific patient using the structure defined above, the following command line would accomplish that task:

```
SET x = ““ FOR I=1:1 SET x = $ORDER(^pt(id, x)) QUIT:x = ””  WRITE !, x
```

This line begins with an empty string for x, obtains the first date for that patient, writes the date on a new line, and continues to execute the line until x returns an empty string, meaning that the list of subscripts at that level is exhausted. (Note the two spaces required before the WRITE statement, since QUIT has no arguments in this use of the command.) $ORDER works at any level of subscripting, returning values that are stored at that depth without changing values of preceding subscripts.

An important feature of the M language that makes these functions even more significant is the fact that subscripts are automatically stored according either to a default *collation sequence*, based on the ASCII character code set, or to another approved character set profile approved by the MUMPS Development Committee. The default collation sequence treats canonical numbers (any number without nonsignificant leading or trailing zeros) as a primary collation set, followed by the ASCII code value of each character in the string under consideration. In this way, subscripts not only can have real-world significance, but they are also automatically sorted (alphabetized if the subscripts are alphabetic characters). Since sorting constitutes a large part of many database operations, this feature of M provides for major saving in coding complexity.

These two functions are further augmented by the $GET function, which returns the actual value stored at a specified location: $GET(^pt(id,date,“medication”)) would return “Tylenol PRN”. If no data value exists at that subscript level, $GET returns a null value, but does not return an error, which would result if one were to attempt to write a nonexisting value.

Other functions assisting in traversing sparse arrays include $QUERY, which performs a depth-first search of an entire array; $NAME, which returns the full global reference (name and subscripts); $QLENGTH, which returns the number of subscripts of a given global name; $QSUBSCRIPT, which returns a subscript at a specific level in the hierarchy; and $ORDER(array,-1), which reverses the direction of searching a global variable. These somewhat more complex forms of array manipulation are not described in detail herein.

Transaction Processing

Many database operations require that several steps be considered as a single process. Transferring funds from one account to another, for example, must go to completion (funds must be taken from one account and deposited in the other) in order for the transaction to be complete. If one step is omitted, the entire transaction must be aborted or restarted. Financial institutions insist on this provision in database packages. The MUMPS Development Committee adopted language to incorporate this functionality in the latest revision of M approved by ANSI in 1995. Four new commands were defined. TSTART signals the beginning of a set of operations that will be considered as a single transaction. If tests at the conclusion of this transaction verify that the entire operation has completed successfully, then the command TCOMMIT will cause the transactions temporarily stored in a provisional status to be committed, affecting the original database. If the verification process indicates that some part of the transaction was not properly completed, a TROLLBACK command will erase all changes made temporarily since the issuance of the TSTART command. TRESTART can reinitiate the entire transaction, returning control to the TSTART command. A special variable, $TRESTART, is incremented each time TRESTART is invoked, so that one can test this variable to determine if the number of attempts to restart the transaction exceeds a predefined level of acceptability.

Interaction with the Underlying Operating System

Although M is no longer implemented under its own operating system, it does provide some tools that enable it to interact with the system. The special variables $HOROLOG, $X, $Y, and $TEST described earlier are examples of such interaction. There is a need for additional information to be exchanged between the system and M. Many of these requirements were implemented by different vendors of the language in nonstandard form. Recent extensions to the language have standardized a number of these features, enabling code written in Standard M to be ported across a wider variety of systems even though the features rely on system interaction.

One way in which standardization was achieved was by addition of some special variables. $PRINCIPAL was accepted as a standard reference to the user's console (keyboard, screen, mouse, etc.). With this special variable in place, users can issue the command line USE $PRINCIPAL and, on all M implementations, force I/O to be directed to the principal device of that user's process (in most cases, screen or keyboard). Another special variable is used to determine the actual characters used by a specific device to terminate a READ command. Since the characters sent at the end of a READ command vary with different devices, this is a useful way of determining the actual codes used.

Another mechanism devised to provide this standardized functionality is the *structured system variable*, a kind of variable identified by the initial characters ^$. For instance, ^$DEVICE provides information to M about what devices are connected to the system in use: their existence, their characteristics, and their availability to the user. Since it is possible to spawn new tasks from M, the structured system variable ^$JOB identifies tasks that have been started in an M environment and provides information about their status. ^$LOCK provides a list of global variables that are currently locked. ^$ROUTINE and ^$GLOBAL contain information about the available routines and global variables, and about the character sets that are valid for the various routines and global names. ^$SYSTEM provides some standard information about which M implementation is in use. The MUMPS Development Committee assigned numbers to several of the most common commercial implementations, making them available through the special variable $SYSTEM. The structured system variable allows users to further interrogate these values so as to obtain additional information about their characteristics. ^$WINDOW is a system structure that can be used to create and manipulate windows. Its specification is found in a separate standard: ANSI X11.6-1995, ISO 15852-1999.

Error Management

Errors occur at all levels of code development and use. Syntactical errors, which are usually detected prior to execution of a routine, can be treated in normal ways by the interpreter/compiler. However, run-time errors are more complex. In order to facilitate communication between user and system when run-time errors occur, M contains a set of special variables that facilitate error processing. $ECODE defines a number of standard codes representing errors that might occur during execution. $ETRAP allows a user to define actions that can be taken (using the XECUTE command) based on the likely cause of such an error in a given application.

For cases where execution is complex, involving called procedures requiring stacking of system values, the variable $STACK was added to the language to indicate the current depth of the stack. $ESTACK provides a similar stack representing the depth of stacked errors. Finally, the function $STACK() returns information from the error information stack. As a result, users can get deeply into the actual interaction between application code and the system, detecting logical or data-driven errors that would be extremely difficult to unravel otherwise.

Internationalization

M is one of the few high-level languages to have explicitly addressed many of the issues associated with writing applications for non-English languages. Since it is an international standard, widely used in Europe, Japan, and Brazil for applications in the financial and commercial sectors, it is important that non-English characters be properly handled in data, and, because of the availability of string subscripts, it is also necessary to permit use of non-English in arrays both as data and as subscripts.

The way in which M handles these matters is to require a given non-English localization to define a *character-set profile*. This profile includes definitions of the actual character set(s) used, pattern codes that are to be applied to these character sets, rules governing the use of the character sets in names of variables and labels in M routines, and the collation algorithm to be implemented for a given local setting. To date, there are three character-set profiles in the M standard: M, ASCII, and the Japanese character set JIS 90. The enabling mechanism for this profile is the structured system variable ^$CHARACTER. The first subscript under this variable defines the character set approved in the M standard. Other subscripts provide ways to define pattern codes and collation algorithms. This mechanism is important not only for east Asian languages; collation in many European languages using diacritical characters is quite complex, sometimes requiring as many as five separate passes through the text to determine the proper collation position of a string.

Auxiliary tools have also been defined to facilitate *multilingual* data processing. For instance, if one wishes to use different collation rules to store sets of global variables, it is necessary to store character set profile information specific to a given global variable name. The structured system variable ^$GLOBAL provides this level of granularity, so that it can be used to identify different globals stored within a given database.

By the same token, routines (programs) written in M can be tailored to different localities. ^$ROUTINE is the structured system variable that allows routines to make use of specific character-set profiles, so that an application can have different routines for different countries to reflect their specific cultural preferences.

Interfacing M to Other Standards

The MUMPS Development Committee has been in existence for over 25 years. During that time, it has maintained a strong dedication to standardization, including establishing close ties with other standards. These ties include use of standards appropriate for incorporation into M code; binding other standards with M using standard interfaces; and embedding code from other standards within M with a standard syntax recognized by M to begin and end those segments. Examples of each follow.

Interface to Standard Character Sets. From its inception, M has used the ANS standard character-set ASCII as the basis for language definition. This union has worked well, but it falls short as soon as one deals with non-English languages. In considering the issues of internationalization, the MUMPS Development Committee took a broad view, defining the ways in which other character sets might be interfaced to M. This approach is reflected in the earlier discussion on internationalization of the language. The first character set profile to be approved dealt with Japanese standard documents X0201-1990 and X0208-1990. Since then, the Committee has approved a binding to the ISO character set used for western european languages: ISO-8859-1 (ISO-LATIN-1), the 8-bit character set that contains diacritical marks for languages in western Europe. This character set profile will be formally adopted in the next revision of the standard; it is already accepted and used by most implementations of M.

Device Control Mnemonics. I/O is an important element in programming languages. Since there are many different types of devices, it is difficult to know exactly how each device is to be used. Prior to adoption of the 1995 M standard, different implementations of the language used different, incompatible ways of dealing with device I/O. With the 1995 standard, however, M relies on another standard: ANSI 3.64-1979 R1990 (ANSI Terminal Device Control Mnemonics). Using these standard references, it is possible to perform a great many I/O operations that are specific to such tasks as cursor control, font selection, device status, and others.

Interface to the Graphical Kernel System. Another binding to device controls deals with windows on display screens. The Graphical Kernel System (ANSI X3.124–1985: Computer Graphics—Graphical Kernel System) was an attempt early in the development of windows-based applications to provide graphic display functions for windows. M became interested in linking to that language during the mid-1980s. It was an important historical departure for M, in that it was one of the earliest efforts to bind M to another formal standard. The M document describing this binding is X11.3 M[UMPS]-GKS Binding. Since GKS is currently undergoing revision, we do not include details in this article.

Relating M to SQL. SQL, thestructured query language used in most implementations of relational model databases, has had an active standardization history. Since M is closely associated with database management systems, it made sense for the M Development Committee to form an association with the evolving SQL standard. This association took the form of having members of the MUMPS Development Committee participate in the deliberations of a subcommittee of the ANSI committee X3H2: Databases. This committee was charged with standardization of SQL as used in different relational systems, and in the linking of SQL to other standard languages such as C and M. Two documents resulting from these activities cross-reference each other. The 1992 SQL standard (ANSI X3.135 and ISO/IEC 9075: SQL2) contains a section explicitly describing how SQL can be embedded in M code. On the M side of the fence, Annex D of M's X11.1(1995) reproduces portions of the X1.135 document that explicitly reference M. Embedding SQL code in an M routine is quite easy to accomplish. The SQL code must be inserted in the form a &SQL(*SQL code*). There is no real problem in referring to M stored data (globals) as relational, providing that the user has created a variable structure that is consistent with the relational model structure. With a single subscript file, it is not difficult to implement the relational model structure and thus to perform operations on such a database that are similar to those used in the relational model. (This has been implemented in a number of commercial versions of M.) There is always the potential of conflicting structural assumptions, but these become dependent on the schema used in the SQL code versus the structural definitions that might exist in the more open-ended M file structure. M has successfully implemented a number of applications in which SQL commands can be used to manipulate M data.

Interface to Other de Facto Standards. Creation and revision of standards is a constantly evolving process. Usually, a standard gets its start by being adopted by several groups, which then combine forces to agree on the specifics of the standard, use it, and ultimately submit it for approval through a national or international body. Along the way, there may be many de facto standards that have been accepted by various groups but not yet approved by ANSI or ISO. Two examples of widely accepted standards that have not yet achieved formal approval by a standards body are Open Data Base Connectivity (*ODBC*) and Open GL, a graphics control language first proposed by Silicon Graphics, Inc. and now widely used and a candidate for formal adoption as a standard. M has examples of implementations that support both of these evolving standards. They serve as models for future negotiations with groups interested in creating and maintaining standards whose applicability extends into the M domain.

M and the Windows Environment. In the past, the most common user interface for M applications was dialog-driven question-and-answer interaction. While this approach was satisfactory prior to the widespread adoption of the Microsoft Windows paradigm, it is no longer acceptable. More sophisticated user interfaces are needed, and a graphical user interface (*GUI*) is essential in today's marketplace. Realizing this, M developers explored various options and ultimately settled on two different approaches: embedding windows functionality within the language, and interfacing M with other languages suited to GUI operations. The modifications to M that permit windows programming are defined in the ANSI standard document X11.6: M[UMPS] Windowing Application Programmer's Interface, and the acronym *MWAPI* is often used to refer to this approach. This document is longer than the M standard itself, and it is inappropriate to attempt to cover it in detail. It provides system-independent tools to design general screen layout, details of screen display, gadgets commonly used in GUI windows, and the processing of events arising from user interaction. The principal structured system variables used to manage Windows programming are ^$DISPLAY, ^$WINDOW, and ^$EVENT. An example comparing the use of both of these approaches is found in Chapter 23 of Ref. 1.

Example of An Application Written in M

In order to illustrate the characteristics of the M language, we present below an adaptation of a program taken from Ref. 4. It was provided by Ed P. deMoel, former Chair of the MUMPS Development Committee. We gratefully acknowledge his contribution. This example follows the convention of capitalizing the first letter of commands and using lowercase letters for variable names and labels.

```
Admenu    ;Menu for Address Package; EDM; Feb 2000
    ;
    New menuname
    ;
    ;   Identify the menu subscript to be used in this routine
    Set menuname="Address"
    ;  Check whether the menu is defined
    Do : ′$Data (ˆmenu(menuname) init
    ;
    ;  Repeat asking for menu options till done
    Write !, "Get Address Book Choices"
    Kill done
    ;
    For  Do  Quit :$Get (done)
    ·   New choice, option
    ·   Read !,"Enter choice (?for help): ",choice
    ·   If choice["?" Do help (menuname)  Quit
    ·  ;
    ·  ;   Check whether the requested option exists
    ·  ;
    ·  Set choice=+choice
    ·  Set option=$Get (ˆmenu (menuname, choice),"err")
    ·  If option="err"  Do  Quit
    ·   · Write !, "Please enter the number of a menu option." , !
    ·   · Quit
    ·  ;
    ·  ; Acknowledge the request and act on it
    ·  Write "", $Piece (option,";", 2)
    ·  Xecute  $piece (option,";", 1)
    Quit
    ;
help (list) ; Show the menu options
    New  i
    Set i="" For Set i=$ order (ˆmenu) list, i),";", 2) ) Quit :  i="" Do
    · Write !,$Justify (i,3),":"  ,$Piece (ˆmenu (list,i),";",2)
    · Quit
    ;
    ; Initialize the menu
init New menu
    Set menu (1)=" Do ˆaddress;Address Information"
    Set menu (2)=" Do ˆphone; Phone/fax numbers (Work and Home)"
    Set menu (3)=" Do ˆemail; Email Address"
    Set menu (4)=" Do ˆstats; Date of Birth, etc."
    Set menu (5)=" Set done=1; Quit"
    Merge ˆmenu("Address")=menu
    Quit
```

Let us examine this code in more detail. The first line starts with a label and contains only comments. This a convention followed by most programmers in M. The convention is for the first line to begin with the name of the program, then list as comments the function of the program, its author, and the last date the program was modified:

```
Admenu     ;Menu for Address Package; EPM; Feb 20; 2000
```

The body of the program begins with a declaration:

```
New menuname
```

Declarative statements are not required in M, but it is good programming practice to list variables specific to the current program. Using the NEW command has the added benefit of limiting the scope of the declared variables to the program in which they are declared, together with any subroutines called by that program.

In the next lines, the program initializes a (local) variable that will be used throughout the program, then removed after the program terminates execution. Its purpose is to restrict the program's effective domain to data elements belonging to this package. Having defined the variable, the second line reproduced below checks to see if a persistent variable of that name already exists under the global variable name ˆmenu. If it does not exist (i.e., $DATA(ˆmenu(menuname), returns a zero), a subroutine init is called to initialize this global variable:

```
Set menuname = “Address”
Do: ′$ Data (ˆmenuname) ) init
```

The subroutine labeled init begins with a declaration limiting the variable menu to the scope of this subroutine. This local variable will cease to exist when the subroutine returns control to the main program. The next five lines (Set menu(1)=“Do ˆaddress;Address Information” through to Set Menu(5)= . . .) create the menu structure for this application. Each menu item is defined as a node consisting of a string containing two parts separated by a semicolon. The first part is an M command that will be invoked if that choice is selected. The second part is a descriptive text used as a prompt to guide the user to the appropriate selection.

Once these values have been defined as a local-variable array, the next line uses the MERGE command to copy these values into a global persistent array that remains after the program has terminated:

```
Merge ˆmenu (“address”) = menu
```

Note that the merge command transfers the entire array with a single command (even if the array is a tree structure with descendants). Following this transfer, this subroutine terminates with a quit command, returning control to the main program, and restoring values of variables defined by the NEW command—in this case menu—to whatever might have existed before the subroutine was invoked.

The next two lines of executable code are:

```
Write !, “Get Address Book Choices”
Kill done
```

The first line displays a prompt for the user. The second erases any dangling values of the local variable done in order to prepare it for use in subsequent lines, as explained below.

The purpose of this program example is to allow the user to select from a set of options. The program should loop as many times as the user specifies another option and execute each selection in turn, returning to this control loop. M uses the FOR command to control loop executions. The FOR command can be used in a number of ways:

As a "repeat until" structure: For Do code Quit:Condition
As a "while repeat" structure: For Quit:Condition Do code
As a specified number of iterations: For count=1:1:10 Do code

In this case, the program uses the "repeat until" structure:

```
For  Do  Quit:$Get (done)
```

Two spaces are required after the FOR and DO commands, signifying the absence of any specific parameters controlling these commands. In the case of the FOR command, it is terminated by the QUIT command later in the line, which becomes TRUE when the variable done is defined with a nonzero value (i.e., when option 5 is executed). The DO command without parameters signifies the start of a block of code identified by an initial dot (·) at the start of the line, one dot for each level of nesting of the block(s).

The DO subroutine embedded in the next few lines begins by initializing the two variables choice and option and then prompts the user to select a choice. The available options are defined by the subroutine init supplemented by allowing the user to enter a ? in order to review all the choices available (this program assumes the user already is familiar with those choices and does not repeat them each time this code is invoked unless the user requests it):

```
. Read !, "Enter choice (? For list) : ", choice
```

The user's entry is checked first to see if a question mark was entered, in which case the program transfers control to the help menu (described in a later section):

```
. If choice["?" Do help (menuname) Quit
```

The contains operator [in this line will evaluate true if any character typed by the user is a question mark, even at the end of a sentence. If this code is invoked, the help subroutine is executed and the progam QUITs to the continuation of the menu choice FOR loop

If the user enters a value other than a question mark, the program continues:

```
. Set choice = +choice
. Set option = $Get (^menu(menuname, choice), "err")
```

The contents of choice are converted to a canonic number by the unary + operator. This line has the effect of converting the user's input to a numeric even if a letter has been typed (in which case the value will be 0). The next line assigns a value to option corresponding to the user's selection, if valid, and otherwise, the $Get function selects the next, default value of "err" if no value exists for ^menu(menuname,choice). If the user's choice results in choice having a value of “err”, then the following lines are executed:

```
.  If option = “err” Do  Quit
. . Write !,“Please enter the number of a menu option”,!
. . Quit
```

Note that a new DO block is invoked, nested within the first, so that two initial periods are required to signify the second level of nesting. The QUIT at the end of this block returns control to the outer block, which then QUITs to the FOR loop level.

Execution of the requested option is controlled by the following lines:

```
. Write “ ”, $Piece(option, “;”, 2)
. Xecute  $Piece (option, “;”, 1)
```

Since a valid option has been selected, the program uses the value returned by the $GET function stored at that level of ˆmenu. The $Piece function uses a semicolon as a delimiter, and extracts the piece identified by the third argument from the value stored in option. The first line retrieves the prompt (e.g., if option is 1, the value returned is “Address Information”, which is then displayed to confirm the user’s choice. The next line uses the XECUTE command to treat the first datum stored as a command line; if option is 1, the result will be to Do ˆaddress. Once that routine has been executed, control returns to the line immediately following the Xecute command, QUITs back to the FOR loop, and continues to execute user choices until a choice of 5 is entered, at which point the variable done is set to 1, signaling the end of the FOR loop. The final QUIT line (without a period) is the termination of the entire program, since both the DO and FOR command loops have terminated.

The help subroutine is invoked if the user does not know what choice to select. This section reads:

```
help (list) ; Show the menu options
  New  i
  Set  i = “ ” For  Set  i = $Order (ˆmenu (list, i) ) Quit : i=“ ”  Do
 ·   Write !, $Justify (i, 3), “:”, $Piece (ˆmenu (list, i).“;”,2)
 ·   Quit
Quit
```

This subroutine uses a local variable, i, to loop through the ˆmenu(. . .) list, displaying each number defined in the ˆmenu global variable, followed by the second piece of the value stored at that node (the option displayed to the user upon selecting that number). This subroutine is called by the command Do help(menuname), with the value of menuname having been set to “Address” at the start of the program. The first line of code above stores that value in the local variable list and uses that to access the appropriate option information. The information is displayed with the number right-justified in a 3-column field followed by a colon and space and then the prompt information.

This code is versatile. It can easily be adapted to create a menu driver for any type of application package. The only lines that will need to be replaced are those that explicitly reference the “Address” domain targeted by this program, i.e., the first executable line of code and the subroutine init.

Future Directions of M

M contains many features that are unique among programming languages, including persistent data, sparse arrays with string subscripts, and built-in multiuser functionality. It has been shown to outperform relational model databases, especially in data storage requirements and the processing of queries requiring natural joins. The uses of M have expanded with the increased emphasis on client–server systems, where M has been shown to be a highly effective scripting language for use with browsers for both clients and Web-based servers.

As with any language in today's world, however, M must continue to evolve if it is to stay competitive. That evolution now appears to be headed down two paths. On the one hand, the largest vendor of M has elected to develop versions that embody functions beyond those approved by the standard, changing the name of the language to reflect its departure from adherence to standards established by the MUMPS Development Committee. On the other hand, that committee is continuing to discuss new features that might be included in future standard releases, even though the number of participants in the discussion has decreased.

One effort that appears to bear promise of more widespread use of standard M implementations is a group effort to produce an open–source implementation of the language. Experience with other more recent languages (such as Perl) suggest that a concerted effort to increase the visibility of the language through increased availability without licensing fees may lead to its greater acceptance. It remains to be seen if that effort will succeed in generating increased interest in the standard versions of M. The fact that a great many applications already written in standard M are in active use around the world may lead to an effort to provide further support for the maintenance of a standards review body to support this language.

Regardless of the manner in which the language evolves, work has already begun on the incorporation of some important extensions to it. By far the most significant of these is inclusion of object-oriented concepts. A subcommittee of the MUMPS Development Committee has been at work on this extension for several years, and some pilot implementations of object-oriented features in otherwise standard M implementations suggest that this feature is not very difficult to accommodate within the language, and it remains only to define the precise manner in which it should be implemented. The most impressive pioneer work in this area, involving an actual implementation of object-oriented concepts, is that of Weichmann (5).

In other domains, the interface with other standards and de facto standards will continue to play an important role in its extended use. M has been implemented under a wide range of operating systems. It has been successfully integrated with technologies such as Visual Basic, Delphi, and Java. The fact that M's "globals" (persistent, shared variables) offer an attractive means of providing persistence to other languages has led to efficient interfaces between M and C/C++ and other languages, giving what may be the best of both worlds. Programmers in these languages can take advantage of the efficient sparse-array structure of persistent data with a few calls that in effect replicate some of the more powerful data manipulation functions provided in M, without having to learn a complete new language.

Summary

M has been around for many years, yet it still offers features not found in other high-level languages. The standardization effort that began more than twenty-five years ago helped the language evolve in important ways, further enhancing its functionality. It remains one of the most flexible languages for internationalization and for database management. It has competed successfully against many other commercial products, demonstrating superior performance in a number of head-to-head performance tests. While the future of the standardization effort is at present uncertain, the contributions made by the language will survive in at least one important commercial implementation. The language continues to serve a wide range of application fields, ranging from containerized shipping through banking to hospital information systems. It appears likely that these applications will continue to flourish. Time will tell whether new major application domains will evolve in the future.

BIBLIOGRAPHY

1. R. F. Walters *M Programming: A Comprehensive Guide*, Marlboro, MA: Digital Press, 1987.
2. R. A. Greenes *et al.* Design and implementation of a clinical data management system, *Comput. Biomed. Res.* **2**: 469–485, 1969.
3. M. E. Johnson *MUMPS: A Preliminary Study*, National Bureau of Standards and National Center for Health Services Research and Development, 1972.
4. D. E. Knuth *The Art of Computer Programming*, Vol 1, Reading MA: Addison-Wesley, 1973, pp. 38–39.
5. T. L. Weichmann Moving M technology into the future with object technology, *M Comput.*, **6**(2): 8–9, 1998.

RICHARD F. WALTERS
University of California, Davis