Language Extensions to Improve the Readability of M

by Lloyd Botway

Introduction

M gets more powerful with each new version of the standard. Many vendors' Z-extensions gradually evolve into new language features. MDC has incorporated extensions to the language, bringing it more into the mainstream as it adopts ideas and constructs from other programming languages. Still, M retains its own flavor and style that wins fanatic adherents and makes just as many fanatic enemies.

As a veteran of twenty years of development with M and other languages, I applaud the evolution of the language. But comfortable as I am with M coding, I am bewildered and frustrated by the *lack* of certain constructs that other languages take for granted.

Current works on software engineering stress that *maintenance* consumes two-thirds of the cost and life cycle of a software product.[1] For code to be maintainable (assuming it did not come from a code generator), it has to be readable. For code to be readable it must, to the greatest degree possible, reflect the meaning intended by the programmer.

Note that readable does not necessarily imply "few grammatical constructs." Adding new features to M certainly complicates the language for interpreter writers, but may make it easier to express higher-level ideas. Some languages are grammatically extremely simple but achieve that simplicity at the cost of readability, such as LISP, or length, such as an assembly language.

Consider this example from the current M language. Ask yourself, What is good or bad (readable or unreadable) about writing a loop in the following way?

```
     SET count=1
loop DO process(array(count))
     IF count<10 SET count=count+1 GOTO loop
```

This is legal M code, but we normally do not write this way because there already is an automatic looping mechanism in M. The FOR command gives us a higher-level construct by which to express the idea that a loop is taking place. The lower-level, more atomic SET and GOTO commands do not instantly convey this meaning to the reader. By high-level I mean referring to a service or function by simply naming it (and possibly giving parameters). By low-level I mean referring to the details of implementation of the service or function. Here we can consider FOR to be the name of a service ("give me a loop"), and the beginning, incrementing, and ending values to be parameters passed to that service.

Suppose a programmer, upon being told to "use FOR instead" had written the following:

```
SET count=1
FOR DO QUIT:count=10 SET count=count+1
.DO process(array(count))
```

Here the GOTO was eliminated, but the manual counting was not. This is definitely a higher-level expression of the loop. Nevertheless, SET, IF (or post-conditions), and GOTO statements do not contain as much meaning as the single FOR statement equivalent:

```
FOR count=1:1:10 DO process(array(count))
```

A by-product of using a higher-level service should be that less code needs to be written. This may not always be the case in terms of actual byte count, but will usually happen. What is more important, using a higher-level service should make the code easier to read, because the meaning of the code should jump out at the reader.

One way to express meaning in a program is by means of data types or classes. Arguments have been presented elsewhere concerning whether M is a type-free or single type language.[2] I am a proponent of type declarations and (in an interpretive environment) run-time checking of type violations. The work in progress includes a description of how such features might be added to M. Here, however, I will focus on readability and semantics.

The main point of this article is to plead for programmers to write (and managers to insist upon) M code that is legible, meaningful, clear, self-documenting, and easily maintainable. Whether using today's standard M, particular vendors' extensions, or tomorrow's enhancements (possibly including suggestions from this article), we can do our part to alleviate

the software industry's maintenance crisis by *demanding readable programs*. Programs should reflect the *meaning*, rather than the *mechanics*, of the task.

This article outlines some straightforward language extensions that have the simple goal of enhancing readability. With only a few exceptions, these extensions can be implemented right now by means of a precompiler. Such a compiler is under construction. The remainder of this article is written mainly as if the proposed changes are already part of the language.

This is part of a larger work-in-progress that defines a portable object-oriented system based on and translatable to Standard M. Readers interested in object-oriented techniques for the time being may consult the works of Dymond, Nelson, Wiechmann, and Goodnough, which appear as references for this article.[3,4,5,6]

ENDDO

The command ENDDO provides an alternative to dot structure. We use it to terminate an argumentless DO. Indentation may be used for readability; it is optional, but strongly encouraged. ENDDO has no arguments. For example:

```
FOR i=1:1:10 DO
     SET a(i)=b(i)+c(i)
     DO ^routine
     FOR j=10:10:100 DO
          SET d(i,j)=d(j,i)
     ENDDO
ENDDO
```

ENDDO may occur anywhere a legal command may occur. Commands between DO and ENDDO do not have to be indented. Thus, the following rewrite of the code shown above is legal (but not as readable):

```
FOR i=1:1:10 DO
SET a(i)=b(i)+c(i)
DO ^routine
FOR j=10:10:100 DO
SET d(i,j)=d(j,i) ENDDO ENDDO
```

The precompiler would assume that lines following a line containing an argumentless DO, which do not start with a dot, are under the control of that DO. The range of the DO will be delimited by a matching ENDDO. If one of the ENDDO commands were accidentally omitted from the loop above, M would have to interpret the remaining ENDDO as terminating the inner FOR loop. Instructions beginning at line NEXT would still be in the outer loop (probably not the effect the programmer intended). Here we are definitely relying on some precompiling. Some readers may feel this is counter to the spirit of a true interpreter, but I think that the DO–ENDDO construct is no different from a long expression in parentheses.

Justification:

- ENDDO more clearly delineates the end of control of the argumentless-DO.
- White space may be used instead of dots. This is the first of several changes which move away from M's line-oriented nature and move toward a syntax in which white space may be used more freely. I favor this idea not because it is popular in other languages, but because it is *more readable*.

FOREACH

FOREACH expresses the most frequently used kind of loop in M with a terse syntax. It is closely related to FOR with or without an argument: it performs a loop while hiding details of its implementation.

FOREACH implements a $ORDER loop through a subscripted variable (local or global). Most M applications take advantage of the nature of sparse subscripts and, as a result, are required to use this mechanism for traversing data structures. FOREACH captures this idea in a single easy-to-read command, as follows:

```
FOREACH ^g(id) DO
   FOREACH ^g(id,sub) DO
      WRITE !,id,?10,sub
   ENDDO
ENDDO
```

This is equivalent to:

```
SET id=""
FOR SET id=$ORDER(^g(id)) QUIT:id=""  DO
SET code=""
.FOR SET code=$ORDER(^g(id,sub)) QUIT:sub=""  DO
..WRITE !,id,?10,sub
```

The argument of FOREACH must be a subscripted variable. The command loops on the last (right-most) subscript of that variable in (left-to-right) $ORDER sequence, beginning and ending with the empty string. (Often the programmer might not care about the order in which the variable is traversed, as long as all subscript values were fetched.)

We might also want a way to initialize a beginning or ending subscript other than the empty string. A syntax such as:

```
FOREACH ^g(id) FROM expr1 TO expr2
```

might work, but if we are going to allow variable starting or ending points, we might as well use the FOR syntax:

```
FOREACH ^g(id)=expr1:expr2
```

which at least is more familiar to M programmers. Note that this is *not* the same as a FOR command, in which the $ORDER function would have to be used explicitly.

QUIT exits a FOREACH loop just as with FOR.

Justification:

- Initializing a variable to the empty string and $ORDER-looping through an array is a common phenomenon in M.
- FOREACH expresses *what* the programmer is doing, not *how* it is being done. (In the spirit of object-oriented programming, FOREACH is an abstract method rather than a specific implementation of the method.)

IN

An IN argument may be used in a FOR argument list. It automates looping through each character of a string.

Here is the format:

```
variable IN expression
```

For example:

```
FOR x IN "ABCDE" DO
```

is equivalent to:

```
FOR x="A","B","C","D","E" DO
```

The intent would be that if expression evaluates to the empty string, the FOR loop does not execute.

Just as other FOR arguments, IN may be used in a series of arguments, as:

```
FOR x=1:1:10,x IN strng,x="Sam" DO
```

The interpreter will recognize that IN is not a command following the FOR statement but rather one of the arguments.

Control characters could be part of an IN-argument, just as they can be part of any string. One of *M Computing*'s editors made the interesting suggestion (with which I concur) that, to permit arguments of arbitrary length, a Pascal enumerated type might be more appropriate. This would allow, for example, the construct FOR x IN Northwest, where Northwest is an enumerated data type containing a list of two-letter state abbreviations.

Justification:

- IN implies set membership. Its intent is easier to grasp than if the set (string) members were listed separately.

PIECEOF

A PIECEOF argument may be used in a FOR argument list. It automates looping through each piece of a string. It assumes that there is a single delimiter which, by design agreement, is used application-wide, and that this delimiter has been specified in the new ^$PIECE(systemexpr,"DEFAULT") structured system variable (discussed more below).

The format is:

```
variable PIECEOF expression
```

For example:

```
FOR x PIECEOF "AB,CD,EF" DO
  ...
ENDDO
```

is equivalent to:

```
SET string="AB,CD,EF"
FOR i=1:1:$LENGTH(string,",") DO
   SET x=$PIECE(string,",",i)
   ...
ENDDO
```

This assumes that the system default delimiter has been set up as:

```
^$PIECE($SYSTEM,"DEFAULT")=","
```

The default delimiter may be temporarily overridden as follows:

```
FOR x PIECEOF "ABCDEF":"|" DO
```

Here, "|" replaces the default comma. This is not a pretty notation. It would be more readable by making it more English-like, at the expense of terseness of coding. For example, we might have:

```
FOR x PIECEOF "ABCDEF" WITH "|" DO
```

but this is wordy. Suggestions are welcome here!

The structured system variable ^$PIECE is intended to contain default $PIECE delimiters for use by other language constructs. System-wide default piece delimiters may be overridden by application-specific default delimiters. I say delimiters in the plural, because many applications provide for pieces within pieces, and so on. This, and the fact that an application may need to override the system, means that ^$PIECE can benefit from a hierarchical substructure.

The precise syntax of the subscripts needs to be decided, but should follow the general plan now used for such structured system variables as ^$CHARACTER and ^$SYSTEM. I would expect the structure to look something like:

```
^$PIECE(systemexpr,"DEFAULT") =
```

system default delimiter

```
^$PIECE(systemexpr,"APPL",application,1) =
```

application major delimiter

```
^$PIECE(systemexpr,"APPL",application,2) =
```

application subdelimiter

and so forth. Here, systemexpr is the current system identified by $SYSTEM, and application is a string determined by the application system designer. (Do we also need a $APPLICATION system variable?)

As an example of the use of ^$PIECE, the American Society for Testing and Materials (ASTM) standard 1238, a data format governing clinical records, recognizes three levels of delimiter. At my workplace we use | for the major delimiter, & for the next, and ^ for the lowest level. We would write:

```
SET ^$PIECE(system,"APPL","ASTM",1)="|"
SET ^$PIECE(system,"APPL","ASTM",2)="&"
SET ^$PIECE(system,"APPL","ASTM",3)="^"
```

Justification:

- Looping through pieces of a string, on a standard delimiter for the application, is a frequent operation.
- Certain standard data formats (for example, ASTM 1238 and HL7) rely on piece, subpiece, and subsubpiece delimiters.

UNTIL

This is another FOR argument. It is popular in other languages and reflects the actual logic of a loop.

The format is:

```
UNTIL condition
```

It tests condition, continuing the loop if *false* and stopping (or using the next argument of the FOR command) if *true*. Here is an example:

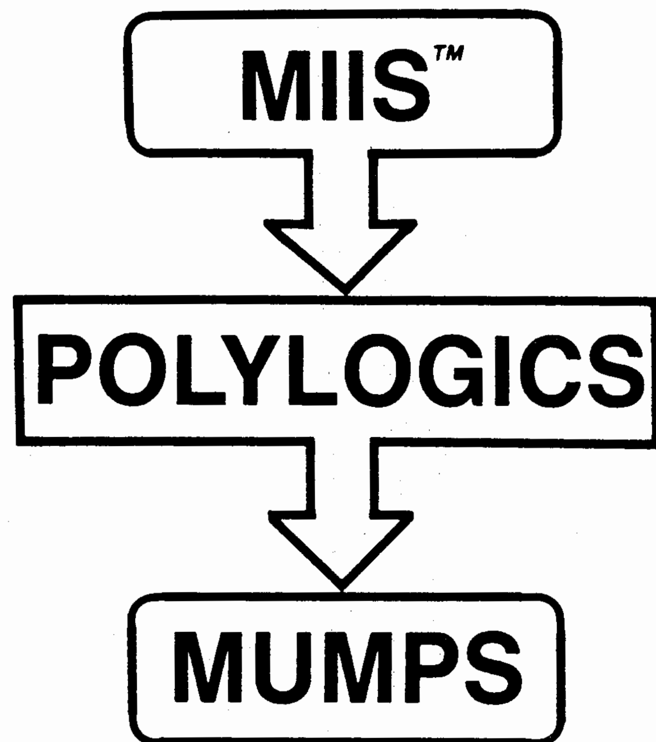
```
FOR UNTIL a>b DO
```

If a >b initially, the loop is not performed at all. Otherwise, when a >b, the loop stops. As mentioned above for IN, the interpreter (or precompiler) will recognize that UNTIL is not a command following the FOR statement but rather one of the arguments.

By varying the position of UNTIL in the FOR statement, the condition specified can be tested either before or after the body of the loop is executed. Thus,

```
FOR DO UNTIL a>b
```

would execute the body first. Note also that a case could be made for defining UNTIL condition to means precisely QUIT:condition, and allowing it to be used anywhere (not just after a FOR command).



We turn running MIIS programs into running MUMPS programs. Efficiently, with maximum accuracy and minimum down-time.

MIIS in, MUMPS out. That's all there is to it.

We specialize in MUMPS language conversions. We also convert MAXI MUMPS, old MIIS, BASIC and almost anything else into standard MUMPS. Polylogics will be there with experienced project management, training and documentation.

So, give us a call today. Ask for a free demonstration on a few of your programs. That's all there is to it.

POLYLOGICS CONSULTING

136 Essex Street
Hackensack, New Jersey 07601

Phone (201) 489-4200
Fax (201) 489-4340

MIIS is a trademark of Medical Information Technology, Inc.

Justification:

- Looping until a certain condition obtains is a natural logical construct. It has the same effect as `QUIT:condition`, but is more expressive of the programmer's intent.

WHILE

This is another `FOR` argument. It is the inverse of `UNTIL`. Format:

```
WHILE condition
```

It tests condition, continuing the loop if *true* and stopping if *false*. For example:

```
FOR WHILE c<d DO
```

If `c<d` is false initially, the loop is not performed at all. When `c<d`, the loop is executed. Otherwise, it stops. Again, the interpreter (or precompiler) will recognize that `WHILE` is not a command following the `FOR` statement but rather one of the arguments.

As above, `WHILE c<d` could be defined to mean `QUIT:c'<d`, freeing it from the confines of the `FOR` statement. Also, the `WHILE` could be put after the `DO`, having the effect of executing the body of the loop first, regardless of the value of the stated condition.

Justification:

- As for `UNTIL`, `WHILE` is a natural logical construct. It has the same effect as `QUIT:'condition`, but is more readable.

FORLEVEL

The system variable `$FORLEVEL` (abbreviation `$FORL`) is a nesting count describing how many `FOR` or `FOREACH` loops deep the code is. Initially, `$FORLEVEL` is zero. It is incremented by one (by the interpreter, not the programmer) for each `FOR` or `FOREACH` executed. It is decremented upon exiting the corresponding loop. It is intended to be used together with the system structured `^$FORCOUNT`, described next.

For example (assume no other `FOR`-loops are in control):

```
FOR i=1,2 DO
   WRITE $FORLEVEL," "
   FOR j=3,4 DO
      WRITE $FORLEVEL," "
   ENDDO
ENDDO
```

This generates:

```
1 2 2 1 2 2
```

Justification:

- This variable is needed as an index to the structured system variable `^$FORCOUNT`, described next.

$FORCOUNT

This is both a system variable and a structured system variable. The abbreviation is `$FORC`. As a system variable, `$FORCOUNT` tells which numbered execution of the innermost `FOR` or `FOREACH` loop is underway.

As a structured system variable, `^$FORCOUNT` tells which particular numbered execution of any nested `FOR` or `FOREACH` loop the program is in. It assigns to each execution of a loop an integer, beginning with one and incrementing by one for each execution. Each `FOR` or `FOREACH` loop has its own counter in `^$FORCOUNT($FORLEVEL)`.

Use `$FORCOUNT` to avoid having to initialize and increment your own loop counter in situations in which you otherwise would not use a counter. Prime examples are `FOREACH`, `WHILE`, and `UNTIL`.

An example, using the (unstructured) system variable `$FORCOUNT` (no nesting), is:

```
FOREACH ^g(s1) DO
   WRITE !,$FORCOUNT," ",s1
ENDDO
```

This writes a list of all first-level subscripts of `^g`, numbering each subscript from one to the number of subscripts found.

This example uses `$FORCOUNT`, but this time it is used with nesting:

```
FOREACH ^g(s1) DO
   WRITE $FORCOUNT," ",s1
   FOREACH ^g(s1,s2) DO
      WRITE ?10,$FORCOUNT," ",s2,!
   ENDDO
ENDDO
```

Suppose `^g` has the following nodes defined: `^g("A","A")`, `^g("A","B")`, `^g("A","C")`, `^g("Z","X")`, and `^g("Z","Y")`. The above loop writes:

```
1 A    1 A
       2 B
       3 C
2 Z    1 X
       2 Y
```

The structured system variable `^$FORCOUNT` would be used when, in an innermost loop, a programmer wants to access the count for higher-level loops. The previous example could be:

```
FOREACH ^g(s1) DO
   FOREACH ^g(s1,s2) DO
      WRITE !,^$FORC($FORLEVEL-1)
      WRITE " ",s1,?10,$FORC," ",s2
   ENDDO
ENDDO
```

(The output of this list is formatted slightly differently from the one above, but the idea is the same.)

Justification:

- Using FOREACH, the programmer describes a loop through subscripts without specifying the number, order, or search mechanism. If the programmer needs to know the number of times a loop is performed, such as to count the subscripts of an array, that value is available with ^$FORCOUNT.

ELSE, ELSEIF, and ENDIF

I have always been perturbed by the lack of a true logical ELSE in M. In all other languages, ELSE depends only on the condition stated in the corresponding IF, and not on possible intervening instructions. M couples ELSE to $TEST and decouples ELSE from IF. Since other commands can affect the value of $TEST, the decoupling from IF can potentially cause problems with program logic.

This new syntax for IF, backwards-compatible with the current definition, has the following features:

- The new command THEN (no arguments) signals this new version of the IF command. THEN must be paired with the new command ENDIF. THEN must occur on the same line as its corresponding IF, immediately after the IF-condition. A lack of a THEN command means that the original standard IF is being used, and that any following ELSE is to use $TEST as usual.
- The new command ELSEIF can test an alternative condition. It has the obvious meaning. It must be used in conjunction with THEN.
- ELSE, if used in conjunction with THEN, is a true logical "else." Commands following it are executed if the IF condition, and any following ELSEIF conditions, were false. Nothing executed as a result of a true IF or ELSEIF condition can make the ELSE execute.
- Probably the most heretical feature is that THEN signals that *line boundaries may be ignored*. The range of the IF command is delimited by the corresponding ENDIF.

General layout of the new IF command:

```
IF condition THEN ...
     ...
   ELSEIF condition THEN ...
     ...
   ELSEIF condition THEN ...
     ...
   ELSE ...
ENDIF
```

IF-THENS may be nested to any depth, and retain their logical cohesion. An IF without a THEN terminates its effect at the end of its own line as usual.

Can we have an ambiguous case? Consider:

```
IF cond1 THEN ...
   ...
   IF cond2 SET a=3 ; a "normal" IF-command:
   ELSE SET a=4 ; to which IF does this apply?
ENDIF
```

I suggest that the interpreter simply assume that ELSE applies to the most recently-issued IF command. What this should mean in practice is that normal IF commands (without THEN) *won't be used*, because of the danger associated with them of other commands changing the value of $TEST. ELSE is now overloaded, in the sense that it has two different meanings depending on whether the most recently executed IF had a THEN or not. There is, however, no ambiguity.

Justification:

- IF, THEN, ELSE, and ELSEIF bring M in line with true logical execution of conditions, as exemplified by other languages.
- ELSE and ELSEIF allow expression of program logic in a more natural way.
- Lifting the restriction that an IF applies only to the line it is on provides greater freedom of expression and better documentation. A routine is easier to read if it can be made to read more like natural language.

CASE

The action of a CASE command is similar to that in other languages. It requires a new case expression syntax, which I have expressed as:

```
expression:
```

A case expression may occur anywhere on a line, but it must be the first thing on a line. (Perhaps it should be a new kind of line label.) It serves to introduce each new case.

General structure:

```
CASE [optional-argument]
   expr1: commands...
          ...
   expr2: commands...
          ...
ENDCASE
```

Here again, line breaks are not significant. The action is similar in intent to $SELECT. In the argumentless form, each case expression is evaluated in turn as a truth value. The first case

expression that is true has its commands executed. Only the first true case expression will take effect. If no case expressions are true, no commands are executed. Example:

```
CASE
   choice="I": DO ^inquiry
   choice="R": DO ^report
   choice="Q": QUIT
   1: DO ^bad
ENDCASE
```

CASE with an argument uses the value of the argument to determine which case to use. Here, each case expression is evaluated and compared to the value of the argument expression. The first one which matches is used. As before, if no case expressions match, no commands are executed.

Example:

```
CASE choice
   "I": DO ^inquiry
   "R": DO ^report
   "Q": QUIT
ENDCASE
IF $CASE="" DO ^bad
```

Here it is useful to have a system-maintained variable $CASE (see next section) whose value is that of the case expression that was used, if any.

It is also useful to have an otherwise case, in the event none of the case expressions matched the CASE argument. We need either a special key word or a symbol indicating this case. My proposal is based on the fact that we have already reserved the colon to separate the case expression from the rest of the line. Designate the otherwise case by a colon by itself, as in:

```
CASE choice
   "I": DO ^inquiry
   "R": DO ^report
   "Q": QUIT
   : DO ^bad
ENDCASE
```

Using a symbol is less readable than using a key word, but the convention at least has the advantage of being clean (no new key words or symbols are introduced).

Note also that the intent of this structure is to ignore line boundaries. Each case's commands extend to the next case expression or to the ENDCASE command, whichever comes first.

Justification:

- This provides a structure which is much easier to read than a sequence of IF and ELSEIF commands. The conditions controlling each case are explicitly stated and stand out.

$CASE

This system variable retains the value of the CASE expression chosen. If no cases matched the CASE condition, $CASE is the empty string. If the argumentless CASE was used (each case expression has a Boolean value), $CASE will be true if any case was chosen and false otherwise.

Justification:

- Without automatic retention of this information by the system, the programmer would have to initialize and manage a variable to record it.

Conclusions

These enhancements have been offered purely as suggestions to improve code readability. Industry experience shows that readable code leads to a reduction in logical errors and an increase in ease of program modification.

Many of these extensions could be provided right now by means of a precompiler. CASE, ELSEIF, ENDDO, ENDIF, FOREACH, the new ELSE command, and the new FOR arguments IN, PIECEOF, UNTIL, and WHILE are easily translatable into standard M. Of course, the resulting (translated) code may not look very nice.

Less easily implemented by precompiling are the new system variables. Lacking a system-embedded way to deal with these, they would have to be implemented in agreed-upon globals and local variables. But a precompiler could insert code which produced the same effect as $CASE, $FORCOUNT, and $FORLEVEL.

Obviously, a great many other suggestions could be made to enhance M syntax and to incorporate abbreviation mechanisms (such as FOREACH). The set presented here is not definitive; each could be expressed in alternative ways, and some may be more important than others. I have many other extensions in the works, only a few of which have been described here. (Others are all for the management and manipulation of classes, objects, and messages. They do not affect the "base syntax" of M as do the above examples.) I am more interested in improving our programming style than in adopting any particular conventions.

Finally, why am I not simply writing this article as a submission to the MUMPS Development Committee? It is because I would like us all, users and vendors alike, to consider what we can do to alleviate the maintenance bottleneck. One way to do this is to make M the most legible and expressive language it can be. I hope the reader takes these suggestions in that spirit. **M**

Continued on page 18

PROGRAMMER ANALYSTS (MUMPS)

SciCor is an international leader in providing the pharmaceutical industry with laboratory data management for clinical trials. As a subsidiary of Corning Life Sciences, Inc., we're part of a worldwide network of quality-oriented laboratory testing and research companies that serve the clinical, environmental, pharmaceutical and life science industries.

Due to business growth, the MIS Department has immediate openings. These positions involve programming and information analysis in support of operational needs of the company, interpretation of user requirements, and formulation of technical specifications. Significant interaction with users and a strong commitment to quality and user satisfaction is essential.

Requirements include a BA/BS degree in any discipline and 1-2 years of demonstrated programming proficiency within a business or healthcare environment. Experience with PC packages, SAS and 4GLS, a plus. Demonstrated proficiency in M Technology (MUMPS) is preferred.

As a part of our team, you'll play a critical role in SciCor's success. We offer an excellent salary and benefits package. Please fax or send your resume to: **Lu Ericksen, Human Resources, SciCor Inc., 8211 SciCor Drive, Indianapolis, Indiana 46214-2985; Fax: (317)273-7977.** Equal Opportunity/Affirmative Action Employer H/V.

SCICOR

Right▸From▸The▸Start

8211 SciCor Drive, Indianapolis, IN 46214-2985

This paper was prepared with the cooperation of National Health Laboratories Incorporated (NHL), La Jolla, California. Many thanks to MTA member and MDC participant Darrell Simmerman, of NHL, for his suggestions and critical feedback. My thanks also to the reviewers and editors for their suggestions and comments.

Lloyd Botway is a director of information systems for National Health Laboratories in Nashville, Tennesee. He is also a doctoral student in computer science at Vanderbilt University. His research interests include database theory, artificial intelligence, and the construction of object-oriented languages. His phone number is 615-399-0713. Or, he can be reached on Internet using botwaylf@vuse.vanderbilt.edu.

Endnotes

1. S. Schach, *Software Engineering* (Boston, MA: Richard D. Irwin, Inc., 1993).
2. T. Salander, "Datatypes: Strong, Weak and Imaginary," *MUG Quarterly* 20:2 (1990):43–52.
3. A.M. Dymond, "Object Programming Concepts in Expert System Knowledge Representation and Control Structures," *MUG Quarterly* 19:1 (1989):90–92.
4. M.L. Nelson, "An Object-Oriented Tower of Babel," *MUMPS Computing* 22:4 (1992):41–47.
5. T.L. Wiechmann, "Object Oriented Programming and MUMPS: Tutorial" (MUMPS Users' Group Meeting 1989).
6. T.L. Wiechmann and J. Goodnough, "EsiObjects: An Object-Oriented Application Development Environment," *MUMPS Computing* 22:4 (1992):30–37.

MTA-Europe Prepares for 19th Annual Conference

Finishing touches are now going on the 19th annual conference of the M Technology Association-Europe, which will take place in Luxembourg November 7 through 11, 1994. "The Key to Client/Server Technology" is the theme of the event with papers and tutorials on server technology and solutions, client technology and solutions, client/server architectures, and financial and medical applications. Concurrently, an exhibition of the latest M Technology from manufacturers, vendors, dealers, and developers is scheduled. A gala dinner at the Chateau Bourglinster is the highlight of the social program.

The meeting will take place at the Parc Hotel Luxembourg, which is located on Route d'Echternach, L-1453 Luxembourg. Phone: 352-43-56-43; fax: 352-43-69-03.

For conference information, contact Mr. Pol van de Perre, MTA-Europe, 83 Avenue E. Mounier, B-1200 Brussels, Belgium. Phone: 32-2-772-92-47; fax: 32-2-772-72-37.

Dial into a BBS in the M World

There are three bulletin boards in active operation that you may not have tried. *M Computing* lists them here as a service to people interested in communicating with other M experts and enthusiasts.

Delaware Valley MUG BBS
SYSOP: Donald Benjamin
215-842-9600 (2400 baud) or
215-842-9603 (9600 baud)

M of Georgia
SYSOP: Rob Gura
404-315-0393

New England MUG BBS
SYSOP: Gardner Trask
508-921-6681