Translating M into English

by W. Lloyd Milligan

M code has an opaque, even cryptic, appearance to the uninitiated. Part of the reason is the near universal practice of abbreviating M names (commands, functions, and special variables). To make matters worse, prior to the NEW command, variable names typically lacked mnemonic significance. Older code also had to deal with an awkwardly limited scope of IF, ELSE, and FOR. This limitation, since overcome by the addition of block structure, led to long and complex code lines consisting of many commands. These and other factors too often combined with an undisciplined programming style to produce M code that is difficult to read and maintain. This article presents a system for reading and interpreting M code that is based partly on ideas proposed by Donald Knuth a decade ago.

Knuth introduced the term "literate programming" in a 1984 paper aimed at promoting better program documentation. In this influential paper, Knuth suggested that computer programs should be written in such a way as to explain to human beings what the programs are doing. Programs, in this view, become "works of literature." Correspondingly, the programmer's main concern is with "exposition and excellence of style." [1,2,3]

The literate programming viewpoint represents an important shift in emphasis. Instead of focusing on algorithmic qualities, elegance of data structure, and so forth, the programmer must think about how readily the program can be understood by others, and how conveniently it can be maintained and adapted to changing requirements.

Knuth applied to literate programming his usual meticulous and exhaustive approach, creating a completely new language environment called WEB.[4] At the time, Pascal was a popular programming language, and a second language to many programmers. The WEB system combined Pascal with TEX, Knuth's beautiful document-formatting language.[5] WEB has since been adapted to a number of other languages.

This article describes a system for reading and interpreting M code. This system is not a programming language. (The programmer creates M source code in the usual way.) Perhaps an example will clarify what the application does. Given the M source code

```
Q X_$S(T]""&$D(VERBOSE):" {"_T_"}",1:"")
```

as input, the system produces TEX source code as output (actually LATEX):[6]

```
Quit returning X concatenate \char 36 SELECT(if T
follows{\em null\/} and \char 36 DATA({\em verbose
mode flag\/}) : then "\verb+{+" concatenate T
concatenate "\verb+}+",
if 1 : then {\em null\/})
```

TEX then compiles this source to produce

> Quit returning X concatenate $SELECT (if T follows *null* and $DATA (*verbose mode flag*) : then "{" concatenate T concatenate "}", if 1 : then *null*)

This example shows only a fragment of the complete documentation for an M routine. In practice, appropriate header and sectioning information is generated, consistent with LATEX's article style (similar to the style of this article). Even footnotes are included.

Consider another potential benefit of a WEB-like M routine-documentation system: A serious shortage of experienced M programmers prevails in several geographic areas.[7] In consequence, many organizations train their own M programmers. Whereas only a few months are needed to learn the rudiments of M, much longer time is required to acquire a *safe* working knowledge of complex M applications, such as one of the VA DHCP (U.S. Department of Veterans Affairs Decentralized Hospital Computer Program) packages. The documentation system described here can be useful to the beginning or intermediate programmer attempting to read and understand complex code.

Old code is never adequately documented, or so it seems. Can the present application facilitate the task of understanding old code? Again, the answer depends on the programmer's experience level. In one respect this documentation system speeds the task of reading and understanding code by folding remote references into the code location being analyzed, thus sparing the human reader a memory-taxing effort. The way in which this is accomplished will be explained in more detail later.

The M Code Reader

Call the application the "M Code Reader." At present the code reader consists of nine M routines and four VA FileMan files exported as ^DIFROM initialization routines. FileMan version 18 or higher and the device selector ^%ZIS are needed

to initialize and run the application. The VA Kernel is not required. The Kernel PACKAGE file (included with FileMan) should exist, however. The code reader accepts M code as input and generates either plain text or TEX language source code as output. Most definitions used by the program are stored in FileMan files. File entries and program parameters determine the verbosity and exact wording of the output. For example, the M construct `$P(Y,U)` might be read as $PIECE(Y, delimited by *up-arrow*). The words "delimited by" are defined as the second argument of `$PIECE` in the INTRINSIC FUNCTION file; the variable `U` is defined as "up-arrow" in the KERNEL VARIABLES file.[8] These files are an integral part of the code reader application.

The attentive reader may have wondered where the code reader found the first example's definition (in code on page 22) of the variable VERBOSE. Obviously, variable definitions are context-specific. In resolving a local variable name, the M code reader first consults the ROUTINE DOCUMENTATION file, then the KERNEL VARIABLES file. In other words the code reader selects a routine specific or package-wide definition, if one is available, before a system wide one. For example, if `U` stands for the Mann-Whitney U statistic in some routine, that routine's definition will take precedence over package-wide or VA Kernel usage. It is possible to avoid substituting any definition (say `U` is a scratch variable) by defining a variable equal to itself (`U=U`). Variables that are not defined in either the ROUTINE DOCUMENTATION or KERNEL VARIABLES files are rendered verbatim.

The ROUTINE DOCUMENTATION file points to the VA Kernel PACKAGE file (9.4) and includes fields for routine-specific variables, package-wide variables, and their definitions. Again, the high-to-low precedence order for variable definitions is 1) routine-specific, 2) package-wide, 3) system-wide.

Definitions may be entered for subscripted variables. For example, FileMan variables `DIC(0)`, `DIC("A")`, `DIC("B")`, and so on, are defined in the KERNEL VARIABLES file. And, of course, definitions are substituted for defined variables in subscripts. Thus, although the code reader parses expressions to the atomic level, in *very verbose* mode, it checks to see if substitutions are possible at higher levels. For example, the truth-valued expression `$D(X)#2` would be read as "X is defined." More precisely, in *very verbose* mode, the literal reading "$DATA(X) modulo 2" is followed by "Note: `$D(X)#2`" may be read as "X is defined." This interpretive note is provided as an example. The user may insert code at `VV^APFCXP1`, in the form of the example provided, to produce additional customized notes.

Routine Comment Lines

A traditional form of M documentation uses the semicolon comment within routines to describe what the routine or code segment does. Some programmers also document variables in this way. It is a fairly common convention to comment entry points. In verbose mode, the M Code Reader resolves routine line references and includes the comment in analysis of code from the calling point. For example, suppose the code line being analyzed includes `S Y=$$TR^RTN(X)`, and further suppose that the entry point `TR^RTN` includes `;;Translate TeX control characters`. The code reader appends this comment (with semicolons and leading spaces stripped) in its analysis of the set command. It will say, "Set Y equal to $$TR^+RTN Translate TeX control characters."

Here is another example:

```
Q:B=2 $$RndBin(L)
```

The reader interprets this command as,

> Quit, if B equals 2, returning $$RndBin(L) Random binary string of length L.

An interesting and WEB-like extension of this facility is the code reader's ability to consider M comments (and variable definitions, as well) as TEX source code. Had the comment at tag `^RTN` read,

```
;;\begin{mweb} Translate \TeX\ control characters
\end{mweb}
```

the output would have converted \TeX to TEX.

TEX source code does not have to begin and end on the same line. A multiline comment can be written, for example, to include a math formula that is being coded in an extrinsic function.[9] In such unusual cases it may be desirable to suppress other output, such as the line tag and verbatim comment. This is accomplished through setting a program parameter (described below).

Another source of explanatory text is the FileMan dictionary of files, `^DIC`. Under some circumstances, the code reader recognizes a file root and is able to work back to the file name. Everything must be in perfect agreement for the code reader to hypothesize a name. Thus, false file names for non-FileMan global references are highly improbable. The code reader interprets the following line from its own code,

```
S:Y Y(0)=$G(^DIZ(16021892.2,Y,0)),F=+$P(Y(0),U,3)
```

as,

> Set, if Y, Y(0) equal to $GET(^DIZ(16021892.2, Y, 0) INTRINSIC FUNCTION file) and F equal to canonic $PIECE(Y(0), delimited by *up-arrow*, piece 3).

The program does not always succeed, however, in identifying a file reference that would be obvious to an experienced programmer.

The code reader produces voluminous output—a one-page routine may expand to twenty pages of liberally spaced analysis. Each code line corresponds to a LaTeX "\subsection," and is identified using the natural TAG+OFFSET format. Due to the quantity of output generated, it is not possible to include an informatively long example in this summary description. One final example will be given to illustrate several ideas in combination. This example includes a LaTeX display formula as an M comment, and an expression using several single-character variables that have routine-specific definitions. The example is an extrinsic function for the well-known loan amortization problem. Error-checking code has been omitted to shorten the example:

```
AMT(P,I,N) ;;Equal Payment Amount
    ;;\begin{mweb} \[ A=iP\frac{(1+i)^n}
    {(1+i)^n-1} \] \end{mweb}
    Q I*P*((1+I)**N/((1+I)**N-1))
```

Corresponding to these three lines, the code reader generates the following LaTeX source code (spacing and line breaks are verbatim):

```
\subsection*{AMT}
\begin{verbatim}
 AMT(P,I,N) ;;Equal Payment Amount
\end{verbatim}
\par{\bf Command 1: }\vspace{-.lin}
\begin{verbatim}
;;Equal Payment Amount
\end{verbatim}\vspace{-.lin}
Semicolon denotes comment.
\subsection*{AMT+1}

\begin{verbatim}
;;\begin{mweb} \[ A=iP\frac{(1+i)^n}{(1+i)^n-1} \]
... \end{mweb}
\end{verbatim}
\par{\bf Command 1: }\vspace{-.lin}
\begin{verbatim}
;;\begin{mweb} \[ A=iP\frac{(1+i)^n}{(1+i)^n-1} \]
... \end{mweb}
\end{verbatim}\vspace{-.lin} \[
A=iP\frac{(1+i)^n}{(1+i)^n-1} \]
\subsection*{AMT+2}

\begin{verbatim}
Q I*P*((1+I)**N/((1+I)**N-1))
\end{verbatim}
\par{\bf Command 1: }\vspace{-.lin}
\begin{verbatim}
Q I*P*((1+I)**N/((1+I)**N-1))
\end{verbatim}\vspace{-.lin}Quit returning {\em
interest\/} multiplied by {\em principal\/}
multiplied by ((1 plus {\em interest\/}) raised to
the power {\em number of payments\/} divided by ((1
plus {\em interest\/}) raised to the power  {\em
number of payments\/} minus 1))
```

Compiling this source produces:

AMT

```
AMT(P,I,N) ;;Equal Payment Amount
```

Command 1:

```
;;Equal Payment Amount
```

Semicolon denotes comment.

AMT+1

```
;;\begin{mweb} \[ A=iP\frac{(1+i)^n}{(1+i)^n-1} \]
... \end{mweb}
```

Command 1:

```
;;\begin{mweb} \[ A=iP\frac{(1+i)^n}{(1+i)^n-1} \]
... \end{mweb}
```

$$A = iP\frac{(1+i)^n}{(1+i)^n-1}$$

AMT+2

```
Q I*P*((1+I)**N/((1+I)**N-1))
```

Command 1:

```
Q I*P*((1+I)**N/((1+I)**N-1))
```

means quit returning *interest* multiplied by *principal* multiplied by ((1 plus *interest*) raised to the power *number of payments* divided by ((1 plus *interest*) raised to the power *number of payments* minus 1)).

It might be argued that the example's documentation is excessive—do we really need to be told that a comment is a comment? The example code is described so completely that even a nonprogrammer could figure out what it is doing. Remember, however, that excessively documented code is not the most common cause of maintenance problems.

Robustness

Routine tools such as the M Code Reader conform to language attributes in effect at the time the application is written. The present application is written for the 1990 ANSI standard. The M programming language, of course, continues to evolve. Changes to the language that are highly congruent with existing elements (for example, adding a new intrinsic function) are easily accommodated. Major changes to the programming language would require modifications to the code reader, as they would other M routine tools.[10] The M Code Reader has been tested on old code and new, applications programs and operating system utilities; and it successfully documents itself. The latter feat is a torture test of the code reader's ability to avoid confusing M and TeX.

Continued on page 26

Generated TEX source lines are usually less than 60 characters in length. When translating complex M expressions in verbose mode the reader can construct intermediate strings exceeding the pre-1994 standard maximum length. Recent M implementations do not have a problem with strings longer than 255 characters.

Z commands are read "implementation-specific command." The interpretation does not distinguish different Z commands. $Z functions are returned verbatim. The user may add $Z functions to the INTRINSIC FUNCTION file, along with appropriate definitions, if desired.

Punctuation in the code reader's output is less than satisfactory. There are several reasons for this. Expressions are parsed recursively, and the code reader does not know where it has been or where it is going. Primitive punctuation rules are used. For example, command post-conditional expressions are written as parenthetical phrases, *command, if condition*, and so on, even if no argument follows. Lists are conjoined using *and* in most cases, e.g., `N X,Y,Z` become "new X and Y and Z." Multiple arguments of `GOTO` (post-conditionalized) are rendered, "goto X, if *condition*, else goto Y,. . ."

The code reader takes a relaxed view of spacing before M commands (any number of spaces is okay), and keeps track of structure level in block structured segments. Structure level is not presently used in formatting, but might be used in a future revision.

Parentheses are problematic to interpret. It is difficult to construct a natural reading for an expression involving nested parentheses, without introducing ambiguity. The code reader avoids this problem by preserving parentheses in the translation. Therefore, the *human* reader should not ignore parentheses in the output.

User-Selectable Parameters

The M Code Reader was originally conceived to translate a single line of M code into ordinary language. As the project developed, numerous ideas occurred which seemed to demand inclusion. Some of these became user-selectable parameters. There are three degrees of verbosity: normal, VERBOSE, and VVERBOSE. The user may elect to analyze one line of code, or an entire routine, producing either plain text output or TEX (LATEX). At the beginning of the analysis of each code line, the original source is reproduced exactly (except for line wrap). A selectable parameter determines how white spaces are displayed. If SHOSPACE is defined, `S Y=X` becomes S␣Y=X, and so forth. Another parameter, MULTLINE, if defined, suppresses output of the line reference and verbatim source in a `\begin{mweb} ... \end{mweb}` group.[11]

The code reader, unlike `^DIM`, or `^%INDEX`, is not a syntax checker. It is designed to document and assist in understanding working code, not for debugging. Syntactically flawed M code will normally produce an (uninformative) error message from the code reader. Results in this case are unpredictable, however.

Try reading aloud a line of M code. No doubt there are as many styles of sounding M as there are M programmers. I asked three programmers to read aloud `S Y=X` and got three different responses. Purists might object to using keywords in places where they don't appear syntactically, for example, "if" to preface a post-conditional. Explaining to human beings what a program is doing requires greater flexibility in the use of language than the rigorous context of actual programming permits.

The M Code Reader may be viewed as a first approximation to automated program explication, or as a form of routine documentation. As conceived, the tool should be most useful for analyzing small code segments, such as a single line or routine. Clearly, improvements are possible, and environment-specific interpretive constructs could be added to the reader's inventory. It remains to be demonstrated whether this tool will prove useful in maintaining practical M applications.

W. Lloyd Milligan, Ph.D., has been with the Charleston, South Carolina, VA Medical Center, and is now consulting on M. You may reach him at 136 Sparrow Drive, Isle of Palms, SC 29451 or use his e-mail address lmilligan@delphi.com.

Endnotes

1. This paper, originally published in *The Computer Journal*, May 1984, is reprinted as chapter 4 in Knuth's book (see note 2).
2. D.E. Knuth, *Literate Programming*. (Stanford, California: Center for the Study of Language and Information, Leland Stanford Junior University, 1992).
3. The need for improved clarity of exposition in programming had been recognized at least ten years before Knuth's 1984 paper. Kernighan and Plauger write, for example, ". . . it is more important to make the purpose of the code unmistakable than to display virtuosity." See B.W. Kernighan and P.J. Plauger, *The Elements of Programming Style*, Second Edition (New York: McGraw-Hill, 1978).
4. WEB source files combine programming language and descriptive text in a single structure. WEB programs are then separately precompiled by "TANGLE" and "WEAVE" to produce programming language source code (Pascal, in the original implementation) and TEX source code, as output.

5. TEX is a trademark of the American Mathematical Society. A variety of TEX implementations including PC versions are readily available. For more information contact the TEX User's Group, phone 401-751-7760.
6. L. Lamport, LATEX *User's Guide and Reference Manual*, (Reading, Massachusetts: Addison-Wesley Publishing Company, 1986), 242.
7. K.M. Schell, "Grow Your Own M Programmer," *M Computing*, 2:1 (February 1994), 34–39.
8. Not all variables are variable. As an interpreted language, M does not include symbolic constants as a defined type. Constant-valued variables, such as U for *up-arrow*, are a common convention.
9. Multiline TEX source comments must begin on a line that is *not* remotely referenced, i.e., not an entry point.
10. MDC currently is considering an enhanced pattern-match operator that would use *regular expressions*. The change would be backwards compatible; however, the code reader would have to be modified to parse pattern expressions.
11. The strings `\begin{mweb} ... \verb+\end{mweb}` are not TEX commands—they are scanned and removed, by the code reader. Use of the keyword *mweb* is not meant to imply any connection to the Modula-2 version of WEB.