The MAGIC/L Programming Language

Arnold Epstein and Claire H. Gilliatt
Prepared for presentation at the 1985 Rochester Forth Applications Conference

When Forth was created in the early 1970's, it represented a major change in the way programmers could think about their machines and the programming process. It allowed fully interactive coding and provided for almost unlimited extension of the environment. This has undoubtedly been an important breakthrough. However, there are a number of characteristics of Forth which can be problematic, especially where group productivity is concerned. Among these are its reverse polish notation and the tendency towards highly dialectical programming. MAGIC/L, developed by Loki Engineering, Inc., combines the fundamental advantages of Forth with a stable syntactical structure and a full set of programming constructs, increasing efficiency in programming and project management.

The main advantage of Forth is its interactivity. This enables immediate testing of subroutines and cuts development time for an individual programmer over languages which require separate compile, link and run phases. Forth is also fully extensible. This has both positive and negative implications. The programmer is not only allowed but required to create a dictionary of routines and definitions for each application. Therefore, most words in a Forth program are user-defined. There is no formal syntax structure and there are limitless ways to solve any particular problem. This makes programming in Forth intellectually stimulating and can be considered an advantage in the short run. It allows the environment to be completely tailored to the "natural" characteristics of the application.

However, this also implies that each programmer writes in a unique "dialect" which can be difficult for others to understand. Even the author can have trouble recreating his own environment a day later. If the author leaves in the middle of a project, it can be disastrous. There is a tradeoff here between custom-fit and consistency; a large programming team can easily become a Babel of programming styles. While it can be argued that Forth is efficient for an individual program, this does not apply to long run project management.

The issue of efficiency also important when considering the Forth compiler. Forth is coded using post-fix notation (RPN). This makes the compiler extremely compact. It was designed at a time when memory was very expensive and limited. Thus it was important for the compiler to be restricted to about 1 kilobyte. The compiler does no syntax analysis; code is predigested by the programmer. However, a result of this space limitation is that Forth code is difficult to read and does not at all resemble English grammar. Since the development of Forth, memory has become less of a restriction in many cases. With the exception of special situations such as cross-compilation to single chip processors, at least 64Kb of RAM is generally available for compiler, code and data. For this reason, one can usually afford to have a somewhat larger compiler in order to improve code legibility.

The advantages of Forth, its interactivity and extensibility, can be overshadowed by its shortcomings as a "write-only" language. It encourages task-specific coding styles and uses reverse polish notation at the expense of consistency, legibility and stability. These are particularly critical factors when the programming is done at a team project level. The idea behind MAGIC/L was to create an environment which supports interactive and extensible programming while providing stable and maintainable syntax.

MAGIC/L was first conceived in 1979 by Arnold Epstein and Jeffrey Morris at the Smithsonian Astrophysical Observatory in Cambridge, MA as a result of working with STOIC, a Forth derivative. At the time, we had six major software packages written in STOIC including an image processing system and a database query system which were highly interactive. In particular, two interactive programs were to be used by visiting scientists, ten of whom per week would be making a two or three day visit to use this software. One program was a database system which contained the observing schedule and analysis status for the Einstein X-ray Observatory Satellite. The other was an image processing system used to analyze the imaged x-ray data. In order to provide a familiar environment to these visitors, we felt it was necessary to create a forward notation user interface for these programs.

At the same time, we were becoming overburdened with supporting the existing packages and wanted to offload some of that support to other non-Forth programmers. The combination of the user interface requirement and the software support issue conspired to make us look at integrating the more familiar forward notation into the language environment. By the end of 1979, we had our applications recoded in MAGIC, a prototype which fully supported forward notation, but lacked in the areas of formatted I/O, argument declarations, and compiler error checking. During the next two years MAGIC was used successfully in the areas of process control, hardware debugging and image processing. The demand for a commercial version led to the founding of Loki Engineering. The first task at Loki was to specify a language which would eliminate the shortcomings of MAGIC, as well as to create a well-documented product.

The first implementation of MAGIC/L was on a Data General Nova in 1981. Since then, it has been sucessfully ported to DEC minicomputers under RT-11 and RSX-11M operating systems, to microcomputers using CP/M-80 and MS-DOS, to various UNIX systems, and to the Apple Macintosh. VAX-VMS and VERSADOS implementations have also been completed.

MAGIC/L is a programming environment which combines the interactivity and extensibility of Forth with the familiarity and maintainability of forward notation. It provides a syntactical structure not found in Forth. For some people, it may be faster to write code with RPN. However, we have decided that it is more important to encourage consistency and readability than to allow an infinite number of ways to tailor the language to a particular application. This optimizes long-range productivity and improves efficiency on a team programming level.

MAGIC/L defines a generic set of system services to perform terminal and file I/O (see the I/O Package below), providing source code portability even at the I/O call level. While MAGIC/L can operate in stand-alone and minimal O/S applications, our standard products run under widely used systems such as MS-DOS and UNIX. This allows MAGIC/L to be used in conjunction with other software (e.g. 1-2-3 , WordStar) increasing its general applicability. Further, most of our customers wish to use other programming languages in addition to MAGIC/L. One of the most popular aspects of MAGIC/L is the ability use it as the system shell.

For years, many others have experimented with enhancements to Forth. MAGIC/L is not just another such experiment, it is the synthesis of hundreds of experiments. Although all of its solutions may not be ideal for a given problem, it is a complete, consistent and portable programming environment. Following is a description of MAGIC/L syntax and functionality.

Data Typing

MAGIC/L variables are similar to variables in C or Pascal. Data-types supported include: CHAR (8-bit unsigned integer), INTEGER (16-bit signed integer), LONG (32-bit signed integer), ADDRESS (variable containing an address, same as INTEGER for 16-bit implementations and same as LONG for 32-bit implementations), and REAL (32-bit single precision floating point). The ADDRESS data-type is used to provide source-level compatibility between 16-bit and 32-bit versions of MAGIC/L.

Both scalar variables and one-dimensional arrays may be declared by following the data-type name with a list of variable names:

```
INTEGER  X, Y, Z(10)
REAL     ANGLE, SIDE, DATA (5*4)
```

The array size declaration may be any constant expression which can be evaluated at the time of declaration. Array subscripts range from 0 to SIZE - 1. Variables can be used in conjunction with operators to form expressions.

```
X := Y * 5
Y := Y + X
```

Note that the variable "Y" was used on both sides of the assignment. The MAGIC/L compiler determines whether the "address" or "value" of Y is implied and takes the appropriate action. This example compiles to 5 words of threaded code. Internally this is accomplished by compiling two code pointers preceding the parameter field. One of the action routines pushes the address of the variable's data onto the stack, the other pushes the value of the data. The compiler determines from context which code pointer to use.

Operators

As shown in Figure 1 (below), MAGIC/L has a full set of assignment, arithmetic, bitwise and relational operators for each data-type. The same operator names are used in expressions of all data-types. The proper type-specific operator is compiled by the MAGIC/L compiler. MAGIC/L will give an error message when a mixed-mode operation is specified, or when the operator is not defined for a particular data-type.

Assignment Operators

:=	store	INCREMENT	add 1 to contents
+=	add to contents	DECREMENT	subtract 1 from contents
-=	subtract from contents	SET	set all bits to 1
		CLEAR	set contents to 0

Arithmetic Operators

+	add	-	subtract (binary), negate (unary)
*	multiply	/	divide

Relational Operators

==	equal to	<>	not equal to
>	greater than	>=	greater than or equal to
<	less than	<=	less than or equal to

Bitwise Operators

NOT	one's complement	OR	bitwise inclusive OR
AND	bitwise AND	XOR	bitwise exclusive OR

Figure 1 - MAGIC/L Operators

Record Structures

One of the most dialectic aspects of Forth is the description of complex data structures. MAGIC/L provides great consistency in this area by allowing the programmer to create complex data structures, called RECORDs, out of implicit data-types and other RECORD types. A data structure containing a date might be defined as:

```
RECORD DATE_REC
   INTEGER MONTH
   INTEGER DAY
   INTEGER YEAR
ENDRECORD
```

This declares a new data-type called DATE_REC which consists of three integers. These elements can be referenced by name or by position in the record. Scalar and array variables of user-defined data-types (called record variables) are declared with the same syntax as intrinsic variables.

```
DATE_REC  BIRTHDAY, HOLIDAYS (10)
```

An element of a record variable may be accessed explicitly using the colon operator to specify which record variable the element belongs to. For example:

```
BIRTHDAY:MONTH := 6
HOLIDAYS(2):DAY := 4
```

In addition, the WITH command directs all record structure element references to a specific record variable For example:

```
WITH HOLIDAYS(8)
MONTH := 12
DAY := 25
YEAR := 85
```

Library Routines

Library routines fall into a number of categories including: Arithmetic functions (eg. MIN, ABS, SQRT, etc.), Trigonometric functions, String functions, Mixed mode operations, Mode conversion routines, Bit manipulation routines, Block data movement, and Memory access functions.

By having a large number (about 100) of often-used routines built into MAGIC/L, programmers do not have to write them for each occasion and can use a vocabulary which is familiar to other programmers. Most of the MAGIC/L routines are named so that their functionality is clear. In this way, even people who do not use MAGIC/L can read MAGIC/L code without much difficulty. Many of the library routines can be redefined if the user so chooses, but this can diminish readability and consistency if carried too far. The user can also delete a number of library routines from MAGIC/L to clear more space.

Formatted ASCII Output

MAGIC/L supports a complete formatted output facility. Using the PRINT command, it is easy to format any desired output line. Integers can be converted as signed or unsigned values, and formatted into any width field either right or left justified. Reals may be formatted in either fixed point or scientific notation. Strings may be either right or left justified within a field. A columnar tabbing facility allows absolute or relative alignment of fields within the line. Lines can be output with or without carriage returns. PRINT statements take a variable number of arguments of independent data-types. Expression arguments are calculated and the resulting values are printed. The contents of variables can also be printed. Output can be directed to the console (default) or to a file, or into memory. This combination of capabilities makes the PRINT command a powerful feature of MAGIC/L. The following example prints a string, an INTEGER and a LONG. The #R directive specifies that all subsequent integral values be printed in the specified radix (HEX in this case):

```
PRINT "Two Hex Values:", #r 16.,  5*3, 123456L
```

Formatted ASCII Input

Another important MAGIC/L feature is the INPUT facility. The INPUT statement is used to convert ASCII text into one or more items of binary or string data. Numeric input is converted into the data-type required, and string input is converted into an unsubscripted CHAR array. The basic form of the INPUT statement is a list of variables to receive the converted data. This data can come either from the keyboard interactively or from a file. INPUT has advanced error handling capabilities which can be used in a default mode or in a user-defined mode. INPUT can take several items at a time, even of different data-types. The INPUT statement consists of a list of previously declared variables or input directives, separated by commas. In following example, a PRINT statement prompts for input of a string and an INTEGER. The #Z directive in the print statement suppresses the output of a final carriage-return. The INPUT statement then reads the two data items into the specified variables:

```
CHAR ISTR(40)
INTEGER X
PRINT "Enter Name and Number: ", #Z
INPUT ISTR, X
```

Program Control Structures

MAGIC/L supports a variety of conditional and looping control structures. The user has the freedom to choose the syntax which best suits the algorithm used. All of the conditionals evaluate a "test expression" which must have the data-type INTEGER. It is "True" if it has a non-zero value. Like Forth, MAGIC/L's looping constructs have internal counters which can be referenced without being declared. It is possible to execute any control structure interactively from the keyboard as well as from a routine. This is a particularly useful MAGIC/L feature for both writing and debugging. MAGIC/L control structures are listed in figure 2.

```
DO low, high ... LOOP                    IF (condition) ... ENDIF
ITER count ... LOOP                      IF (condition) ... ELSE ... ENDIF
   I  J  K  increasing counters          BEGIN ... UNTIL (condition)
   I' J' K' decreasing counters          BEGIN ... IF (condition) ... REPEAT
WHILE (condition) ... REPEAT             BEGIN ... FOREVER
```

Figure 2 - MAGIC/L Program Control Structures

Note the two kinds of looping constructs. In DO - LOOP, the counter "I" goes from "low" to "high" inclusive. In an ITER - LOOP, the counter takes on the values 0 to count - 1. Thus the ITER - LOOP counter is matched to the subscript range of an array.

Creating New Definitions

MAGIC/L programs are constructed from variables and routines. The types of routines possible are subroutines which act on input arguments, functions returning values, and commands. MAGIC/L also allows for routines which neither accept nor return values. The basic structure of a routine definition consists of up to 5 sections: 1) The DEFINE statement, 2) The input argument declarations, 3) The local variable declarations 4) The code section, and 5) The END statement. Only the DEFINE and END statements are required. A schematic of routine definition syntax and a code example are shown in figure 3.

```
DEFINE routine-name [routine-type]
    [input argument declaration]
  [LOCAL
         local variable declarations]
  [routine code]
END
```

Figure 3A - MAGIC/L Definition Structure

```
; sum := VSUM (vector, size)
;   Return the sum of all elements of
;   an INTEGER array
;
DEFINE VSUM INTEGER
         INTEGER VEC (0)
         INTEGER NVAL
  CLEAR VSUM
  ITER NVAL
    VSUM += VEC (I)
  LOOP
END
```

Figure 3B - Example of a MAGIC/L Definition

The DEFINE statement specifies the name of the routine as well as the type of routine. It is here that the compiler is told, for example, that the returned value from a routine is to be treated as a particular data-type. The input arguments declaration specifies each input argument in the order of the calling sequence. The declaration is analogous to the declaration of a RECORD structure. The LOCAL variable declaration section specifies a local variable frame which is allocated at the time the routine is entered, allowing for locals within recursive routines. Local variables and argument names are accessible only within the scope of the definition.

As the code is entered, it is incrementally compiled. The END statement causes the compiled code to be permanently saved under the name specified in the DEFINE statement. After the END statement, the routine may be executed immediately from the keyboard or called from subsequently compiled routines. The ability to name input arguments and to declare local variables is a major improvement over Forth in the area of program readability and maintainability.

New Classes of Definitions

A unique and powerful feature of Forth is its ability to create defining words for new classes of routines using the <BUILDS...DOES> construct. MAGIC/L maintains this important extensibility feature. In MAGIC/L, the "DOES" code is defined as a routine of the type ACTION. The function BUILD may be used in any MAGIC/L routine to enter a new word of this class. Since most instances of this feature involve a short list of integer parameters, the command MAKE is provided. Figure 4 demonstrates the creation of a class of integer functions which scale their input arguments.

```
DEFINE SCALEFUNC INTEGER ACTION
         INTEGER   VAL
         INTEGER   PARAM(0)
  SCALEFUNC := VAL * PARAM(0)
END

MAKE    "TIMES3"    SCALEFUNC     3
MAKE    "TIMES5"    SCALEFUNC     5
MAKE    "TIMES9"    SCALEFUNC     9

mgl>    PRINT    TIMES3(5), TIMES9(4)
             15                36
```

Figure 4 - Definition, declaration and usage of an Action Routine

Creating an End-User Command Interface

When creating a program which provides an end-user command interface, most programming languages fall short. In general, it is necessary for the programmer to write an input parser, and if really being elaborate, to write an expression evaluator. This is an exercise that many programmers have repeated in varied ways throughout their careers. The MAGIC/L routine types COMMAND and PARSED significantly inprove productivity in this area. These routines are designed to do most of the work in creating an end-user command interface.

COMMANDs are defined identically to normal subroutines, but the specification of the routine-type COMMAND has special consequences. First, the routine is called without parentheses, providing a "command" syntax. Second, the routine accepts a variable number of arguments, with the internal variable CMDCNT (analogous to "I" in DO-LOOPs), containing the number of input arguments. Third, the routine NXTARG allows users access to the various input arguments.

For example, consider an image processing package where the current ZOOM factor is stored in the variable CURZOOM. Figure 5A shows the user command ZOOM which sets the current ZOOM factor, or displays it if none was specified. PARSED routines are analogous to COMMANDs except that their arguments are strings. Figure 5C shows a skeleton PARSED routine which simply prints all of its inputs.

```
DEFINE ZOOM COMMAND
          INTEGER NEWZOOM
  IF (CMDCNT ==0)
    PRINT "The Current Zoom is", CURZOOM
  ELSE
    CURZOOM := NEWZOOM
  ENDIF
END
```

Figure 5A - Definition of the "ZOOM" Command

```
mgl> ZOOM 5
mgl> ZOOM
The Current Zoom is          5
mgl> ZOOM CURZOOM * 4 + 2
mgl> ZOOM
The Current Zoom is         22
```

Figure 5B - Usage of the "ZOOM" Command

```
DEFINE ECHO PARSED
          CHAR ISTR (0)
  ITER CMDCNT
    PRINT ISTR
    NXTARG
  LOOP
END
```

Figure 5C - Definition of the "ECHO" Parsed Routine

```
mgl> echo this is a command line
this
is
a
command
line
```

Figure 5D - Usage of the "ECHO" Parsed Routine

One important distinction between MAGIC/L COMMANDs and the equivalent Forth IMMEDIATE construct is that COMMANDs may be called from within other MAGIC/L definitions. This allows for the advanced user to create his own "macro" commands based on existing ones.

I/O Package

A key requirement for having code which is portable across processors and operating systems is a portable I/O package. MAGIC/L's I/O package is complete and has proven to be portable to systems as diverse as UNIX and CP/M, MS-DOS and Macintosh.

MAGIC/L I/O routines provide easy access to file I/O, isolating the programmer from system dependencies. It consists of four pairs of routines: OPEN and CLOSE, RDS and WRS for binary I/O, RDTXT and WRTXT for text I/O, and SPOS and GPOS for file positioning (seeking). The OPEN allows for file creation before opening and for appending to the file. The sequential I/O calls RDS and WRS read and write a specified number of bytes starting at the current file position (byte offset from the beginning of the file). Text I/O calls provide ASCII data handling. In particular, they eliminate system dependency from text processing. To the MAGIC/L programmer, a line of text is terminated by a new-line (0A hex) character. RDTXT and WRTXT perform conversion from and to the operating system's concept of "end-of-line". The file position calls, SPOS and GPOS, set and return the position at which the next I/O will take place.

This simple set of routines provides all of the functionality to create applications which are 100% source-code portable. It is worth noting that in order to implement these routines on primitive block I/O only systems (eg. RT-11, CP/M) it was necessary to implement a complete buffered stream I/O system within MAGIC/L. This is one of the many ways in which MAGIC/L provides consistency, freeing the programmer to work on the problem at hand.

Space Efficiency

In comparing MAGIC/L to its grandparent Forth, one must discuss space efficiency, as well as compile and execution speeds. In terms of memory usage, compiled MAGIC/L code is about as compact as compiled Forth code. MAGIC/L has some overheads associated with LOCAL variable frames which is about offset by eliminating the need for "@" when fetching the value of a variable. MAGIC/L tends to encourage writing of readable code over the use of stack manipulations which may save a few words. The major difference is that the MAGIC/L root code is 8 Kb. While this includes full operator, library and I/O support, there is undoubtedly some subset which in any application could have been omitted. The MAGIC/L symbol table is separate from the code (headerless code). There is a facility to selectively remove symbol names and recover the space for use at runtime.

Execution Speed

Since execution speed in both Forth and MAGIC/L is essentially proportional to the number of next-cycles, the two languages are comparable in speed. There is about a 4 next-cycle overhead in the subroutine linkage to routines with LOCAL variable frames, but since these routines are typically several lines of code,the overhead tends to be under 10%.

The real cost in both space and speed is found in the root of MAGIC/L's benefits, the compiler. The MAGIC/L compiler requires about 8 Kb of memory, much more than Forth. In addition, since the MAGIC/L compiler is significantly more complex than Forth's, compile speeds are slower. Much of the slower compile time is due to the symbol table scan which is still a linear search. We have plan in the next revision to use a more efficient search method.

Applications

MAGIC/L has been in use since 1981 in a broad number of applications including typesetting, image processing, robotics, process control, and authoring. To date, MAGIC/L is being used at over 500 installations worldwide.

Conclusion

MAGIC/L is the first incremental compiler to combine the advantages of Forth, interactivity and extensibility, with the productivity improvements associated with forward notation. It provides a stable, readable syntax, a rich set of library functions and programming constructs, and an inline interactive assembler. It is designed to be completely portable, including system independent I/O, yet runs under an operating system to allow connection with existing software. It encourages a consistent programming style at the project level, while preserving the individual productivity gains which come from having a Forth-like environment. MAGIC/L may not be the perfect solution to every problem, but it tends to be a significant improvement over Forth in most programming situations, especially where project management is concerned.

Assignment Operators

:=	store	INCREMENT	add 1 to contents
+=	add to contents	DECREMENT	subtract 1 from contents
-=	subtract from contents	SET	set all bits to 1
		CLEAR	set contents to 0

Arithmetic Operators

+	add	-	subtract (binary), negate (unary)
*	multiply	/	divide

Relational Operators

==	equal to	<>	not equal to
>	greater than	>=	greater than or equal to
<	less than	<=	less than or equal to

Bitwise Operators

NOT	one's complement	OR	bitwise inclusive OR
AND	bitwise AND	XOR	bitwise exclusive OR

Figure 1 - MAGIC/L Operators

```
DO low, high ... LOOP                IF (condition) ... ENDIF
ITER count ... LOOP                  IF (condition) ... ELSE ... ENDIF
   I  J  K  increasing counters      BEGIN ... UNTIL (condition)
   I' J' K' decreasing counters      BEGIN ... IF (condition) ... REPEAT
WHILE (condition) ... REPEAT         BEGIN ... FOREVER
```

Figure 2 - MAGIC/L Program Control Structures

```
DEFINE routine-name [routine-type]
    [input argument declaration]
  [LOCAL
         local variable declarations]
  [routine code]
END
```

Figure 3A - MAGIC/L Definition Structure

```
; sum := VSUM (vector, size)
;   Return the sum of all elements of
;   an INTEGER array
;
DEFINE VSUM INTEGER
         INTEGER VEC (0)
         INTEGER NVAL
  CLEAR VSUM
  ITER NVAL
    VSUM += VEC (I)
  LOOP
END
```

Figure 3B - Example of a MAGIC/L Definition

```
DEFINE SCALEFUNC INTEGER ACTION
         INTEGER   VAL
         INTEGER   PARAM(0)
  SCALEFUNC := VAL * PARAM(0)
END

MAKE    "TIMES3"    SCALEFUNC     3
MAKE    "TIMES5"    SCALEFUNC     5
MAKE    "TIMES9"    SCALEFUNC     9

mgl>    PRINT    TIMES3(5), TIMES9(4)
            15                36
```

Figure 4 - Definition, declaration and usage of an Action Routine

```
DEFINE ZOOM COMMAND
          INTEGER NEWZOOM
  IF (CMDCNT ==0)
    PRINT "The Current Zoom is", CURZOOM
  ELSE
    CURZOOM := NEWZOOM
  ENDIF
END
```

Figure 5A - Definition of the "ZOOM" Command

```
mgl> ZOOM 5
mgl> ZOOM
The Current Zoom is          5
mgl> ZOOM CURZOOM * 4 + 2
mgl> ZOOM
The Current Zoom is         22
```

Figure 5B - Usage of the "ZOOM" Command

```
DEFINE ECHO PARSED
         CHAR ISTR (0)
  ITER CMDCNT
    PRINT ISTR
    NXTARG
  LOOP
END
```

Figure 5C - Definition of the "ECHO" Parsed Routine

```
mgl> echo this is a command line
this
is
a
command
line
```

Figure 5D - Usage of the "ECHO" Parsed Routine