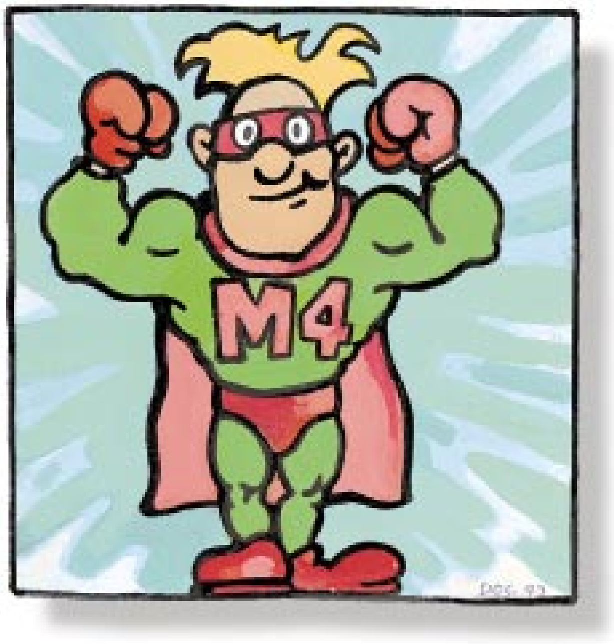
Q&AIX

by Jim Fox



M4 – A More Powerful Macro

Jim Fox works as a systems programmer for the University of Washington. He writes and maintains distributed applications that run on a variety of UNIX systems–and some non-UNIX ones. He is also the deputy manager for the Interoperability Project for SHARE's Open Systems Group. Email: `fox@cac.washington.edu`.

wizard's apprentice

super user

wizard

Q: Your October 1997 article ("Virtual Windows," Page 75) mentioned two ways to customize the `rc` file used by the `fvwm2` window manager. One was `cpp` and the other was M4. Would I be better off going straight to M4?

Beverly Rhodes
City University

A: Last month, we looked at the `cpp` preprocessor, which is simple and very easy to work with. This time, we'll take a look at a much more powerful macro language, M4. Will M4 be the right tool? That depends on how much processing you would like to do.

A macro processor can be defined quite easily–it copies input to output, expanding macros as it encounters them. *Expanding* a macro means to replace it with its definition. How it goes about that macro expansion defines the character of the macro language. Some languages, like TeX, are very complicated; others, like `cpp`, are simple; M4 is somewhere in between.

M4 was written quite a while ago–in computer years. It was originally the macro preprocessor for RATFOR, the rational FORTRAN compiler, which, I think, no longer exists. Its preprocessor, however, has enjoyed something of a revival. It is used by `fvwm` and by the program that generates the GNU Configure files.

The Free Software Foundation (`http://www.gnu.ai.mit.edu`) has made some enhancements to M4; some of which are very useful. The M4 you get with AIX is the original version. If you find you like the program and want to do more with it, you should consider getting hold of the FSF's version.

M4 Macros

M4 normally uses the pound sign "#" to indicate comments. Oddly enough, M4 passes comments into the output stream. I have no idea why it does this, but it will certainly cause trouble if your target program doesn't recognize comments the same way. Fortunately, it is possible to define comments that are not output. We'll see in a moment how to do that. M4 macros can appear anywhere in the input line, not just at the front like `cpp`. They don't have any introductory symbol. A macro name appearing anywhere triggers macro expansion.

Although we always use the term *macro* to indicate a special word that is not simply

passed to the output stream, some of these macros could rightfully be considered commands. We use the term *expand* to mean the action of a macro, even those that are internal and appear to have spurious functions. In general, an M4 macro invocation has the following syntax:

```
macroname(arguments)
```

where *arguments* is a comma-separated list of arguments to the macro. The opening parenthesis must follow immediately after the *macroname*. M4 is very sensitive about spaces, which are almost always significant. Text strings are delimited by quote marks: a single left-quote to start a string and a single right-quote to end it. Macros found in the argument list will be expanded unless they are quoted.

Because M4 is interactive, you can type into it and see the results of any macro expansions immediately. This is a good way to learn the language. Here are some of M4's most useful macros:

- `define(`name',value)` – Defines a new macro, *name*, giving it the expansion of *value*. The value string can contain these symbols to make use of arguments:
  - `$n` – Will expand to the *n*th argument of the invocation.
  - `$0` – Is the macro's name.
  - `$#` – Will expand to the number of arguments.
  - `$*` – Will expand to a comma-separated list of all arguments.
  - `$@` – Will expand to a quoted, comma-separated list of all arguments.

  Note: The quoting prevents *name* from being expanded before it is defined.
- `dnl` – Deletes all characters until the new line. This is often a convenient way to avoid new lines in the output stream. Here's a way to use `dnl` to define a comment that does not get passed on to the output:

```
define(`C',`dnl')
```

Now any text after a `` `C' `` will be ignored. Actually, the `` `C' `` must appear as a word. Text such as `` `Chapter' ``, for example, will not invoke the macro. The `dnl` is quoted to avoid having it expanded, which would have the unfortunate consequence of deleting the rest of the definition.

- `pushdef(`name',value)` – Also defines a new macro, but saves the old definition on a stack. This is useful for defining temporary variables in complex macros.
- `popdef(`name')` – Recovers a pushed macro definition.
- `ifdef(`name',true_text,false_text)` – If the macro, *name*, is defined, this expands to *true_text*; otherwise, it expands to *false_text*.
- `ifelse(string1,string2,true_text,false_text)` – If *string1* is equal to *string2*, this expands to *true_text*; otherwise, it expands to *false_text*.

  Note: The *false_text* may be omitted.
- `ifelse(string1,string2,true_text,more args)` – `ifelse` can be invoked with more than four arguments. If the strings are equal, it expands to *true_text*; otherwise, it discards the first three arguments and repeats the `ifelse` with what's left:

```
ifelse(arg4,arg5,...)
```

- `include(filename)` – Expands to the contents of the named file. This allows you to conveniently include macro libraries.

M4 also has predefined macros designed to work with numbers and strings:

- `incr(number)` – Expands to the argument plus one.
- `decr(number)` – Expands to the argument minus one.
- `eval(expression)` – Expands to the integer value of the expression. The expression can contain numbers, macros and the usual set of operators. It's very similar to C programming. For example, `eval(45*3)` expands to `135`.
- `len(string)` – Expands to the length of *string*.
- `index(string,substring)` – Expands to the index of the first occurrence of *substring* in *string*. It returns `-1`, if there are no occurrences. Note: The first character of a string is at index zero.
- `substr(string,index,length)` – Expands to the substring of *string*, which starts at *index* and is *length* characters long. If the length is missing, the substring contains characters to the end of *string*.
- `translit(string,chars,replacement)` – Expands to *string* with characters in *chars* replaced by the corresponding characters in *replacement*.

There are a few more commands in AIX's M4, and quite a few more in GNU's M4, but these will give us something to work with to explore the power of the language. For more information, consult the M4 man page. Also, check out one of the M4 Web documentation sites. One is `http://www.stat.ucla.edu/develop/gnu/m4_toc.html`. These sites describe the GNU M4, but most commands that also exist on the AIX M4 work the same way. I don't know of any books dedicated to M4.

Using M4

Here's an example of how we can use M4's language to write a macro to do loops. We'll define the macro

```
for(var,
```

start,*end*,
procedure)

such that the *procedure* is expanded for each value of *var* from *start* to *end*. For example,

```
for(`x',1,5,`
   x squared = eval(x**2)')
```

would expand to

```
   1 squared = 1
   2 squared = 4
   3 squared = 9
   4 squared = 16
   5 squared = 25
```

Notice a couple of fine points: The first `x` is quoted, which prevents it from being expanded too soon; and there is a new line in the procedure part, which gives us a new line at the start of each iteration.

Figure 1 shows the definition of the `for` macro. In Figure 1 and in Figure 2, I have included line breaks and leading spaces in the definitions. This is only to help show the structure of the macros. Actual M4 macro definitions almost never have spaces or line breaks. Also, note that I have made use of the `` `C' `` comment macro.

In Figure 1, we have added a `break` macro to provide an escape from the loop. See if you can figure out how this macro works. It uses a couple of techniques common to macro programming. The public `for` macro just sets up some parameters and then calls the private `_for` macro to do all the work. The private `_for` macro performs the loop function by conditionally re-expanding to itself.

Let's demonstrate use of this new loop function, in a sublime sort of way, by writing a prime number macro. All real programmers write prime number programs in every language. It also demonstrates the use of the `for` loop and complex macro programming, and it could conceivably be useful. Many programs, M4 included, allow you to specify a hash size on the command line. This hash size is supposed to be prime. How do you find a prime? Use the `nextprime` M4 macro. It expands to the next higher prime number from the argument. For example,

```
cmd -H nextprime(50000)
```

will expand to

```
cmd -H 50021
```

Figure 2 shows the definition of `nextprime`. Once more, the indentation is illustrative only. Don't include spaces in a real M4 macro unless you mean them.

The macro in Figure 2 uses the same recursive technique as Figure 1. See if you can figure out how it works. We break out of the `for` loop if we find a factor (the `` `t' `` test) or we

Figure 1. M4 Loop Macro Definition

```
C
C *** for loop ***
C
C      usage:  for(var,start,end,procedure)
C
define(for,`undefine(`_break')
   define(`$1',`$2')
   _for(`$1',`$2',`$3',`$4')')dnl
C
define(_for,`$4`'ifelse($1,`$3',,
   `ifdef(`_break',,
     `define(`$1',incr($1))
      _for(`$1',`$2',`$3',`$4')')')')dnl
define(`break',`define(`_break')')dnl
```

Figure 2. M4 nextprime Macro

```
C *** M4 prime number finder ***
C
C        usage:  nextprime(number)
C
C        expands to:  next prime >= number
C
define(nextprime,`undefine(`_done')
   for(`j',2,$1,
     `define(`t',eval($1%j))
      ifelse(t,0,`break()')
      define(`s',eval($1-j*j))
      ifelse(substr(s,0,1),`-',
     `define(`_done',`d')break')')
   ifdef(`_done',$1,
     `nextprime(incr($1))')')dnl
C
```

Figure 3. Useful Definitions Passed from *fvwm2* to the M4 Preprocessor

`WIDTH`	Width of screen in pixels
`HEIGHT`	Height of screen in pixels
`BITS_PER_RGB`	Number of colors available
`COLOR`	"Yes" or "No"
`USER`	User name
`OSTYPE`	Operating system ("AIX" for all versions of AIX)

go past the square root of the number. Recall from your number theory studies that a composite number must have a factor less than or equal to its square root.

Using M4 with fvwm

Now that you've become accustomed to the M4 macro languages, writing that `.fvwm2rc` file in M4 will be a piece of cake. When `fvwm2` runs the M4 preprocessor, it defines several names, which you can use in your file. The most useful of these are shown in Figure 3. See the FvwmM4 man page for the rest.

Here is how we would use M4 to do the simple task shown last month. Suppose you work at various locations, where there are different size X terminals, maybe a large-screen terminal at your office and a smaller one at home. You might want to use different fonts, depending on the size of your screen. You could make some definitions related to screen size at the start of your `rc` file. We'll do things slightly differently this time.

```
# Define screen sizes
   ifelse(eval(WIDTH/1500),1,
       `define(`BIG_SCREEN')
        define(`FONT',7x13)',
     eval(WIDTH/1200),1,
       `define(`MID_SCREEN')
        define(`FONT',6x12)',
     eval(WIDTH/1000),1,
       `define(`SMALL_SCREEN')
        define(`FONT',6x10)',
       `define(`TINY_SREEN')
        define(`FONT',6x10)')
```

Now we can make direct use of the `FONT` macro

```
WindowFont FONT
```

Be sure to start `fvwm2` with the M4 option:

```
fvwm2 -cmd "FvwmM4 rc_file"
```

Some documentation tells you to use the `-f` option for this command, but that won't work–you have to use `-cmd`, you have to use the quotes and you have to specify the `rc` file. Also, a couple of documented options work only if you have the GNU version of M4, notably the "-m4-prefix" option.

*****

There are probably other good uses of M4. Perhaps you can think of something. Let me know.

If you would like to try these macros, you can find them all at `http://weber.u.washington.edu/~fox/M4/`.