M1: A Micro Macro Processor

Jon Bentley

AT&T Bell Laboratories
Murray Hill, NJ 07974

ABSTRACT

This paper describes `m1`, a tiny macro processor. The simple macro language supports the essential operations of defining strings and replacing strings in text by their definitions. It also provides facilities for file inclusion and for conditional expansion of text. It is not designed for any particular application, so it is mildly useful across several applications, including document preparation and programming. This paper describes the evolution of the program; the final version is implemented in about 110 lines of Awk.

1. Introduction

The UNIX® System provides several macro processors. The Shell contains powerful mechanisms for text manipulation, the C language has a macro preprocessor, document preparation tools like Troff, Pic and Eqn all have macros, and the `m4` macro language is a general-purpose tool useful in many contexts. This paper describes a basic macro language named `m1`, which is at least three notches below `m4` (it may well be six below, but `m-2` was too hard to type).

This paper describes the history of `m1`, showing how its implementation grew from a dozen-line Awk program that provides rudimentary services to a limited but useful two-page program. But why should a programmer of the 1990's study a kind of program that was first built in the 1950's? Here are a few reasons I find convincing.

- Macro processors provide a fine playground for learning about programming techniques; Kernighan and Plauger [1981] devote the final chapter of their text on software tools to the topic.
- The particular implementation we'll study illustrates a number of devices useful in building Awk programs. (This paper assumes that the reader is familiar with the Awk language; readers who aren't should study Aho, Kernighan and Weinberger's [1988] definitive text on Awk.)
- Investigating the design considerations of this simple macro processor can help you appreciate the macro languages you use. (Studies indicate that programmers spend 1.7% of their time cursing unexpected side effects of macros.)

And if those reasons aren't good enough, here's the most convincing argument: Since the beginning of time, programmers have studied macro processors, so you have to, too. It's an essential rite of passage.

Section 2 introduces the macro language that we will process. Section 3 starts with a small program to process the language, and grows it to its final version. Applications are surveyed in Section 4, and conclusions are offered in Section 5.

2. The Problem

Given that the UNIX System on which I live already has so many macro processors, why did I even consider building one more? Most macro languages assume that the input is divided into tokens by the rules of an underlying language. I recently faced a problem that didn't come in such neatly wrapped packages; I needed to make substitutions in the middle of strings, as in:

```
@define Condition under
  ...
You are clearly @Condition@worked.
```

The first line defines the string `Condition`, and in the second line that string (surrounded by the special at-sign characters) is replaced by the text `under`. Definitions must start on a new line with the string `@define`; the name is the next field separated by white space (blanks and tabs), and the replacement text is the rest of the line. Replacements are insensitive to context: the string `@Condition@` is always replaced, even if it is inside quotes or not set apart by white space.

A macro language of about this simplicity was sufficient to solve my immediate problem. But once I had it available, more applications of the macro processor started to wander across my terminal screen. The next section starts by implementing this simple language, and then adds operations that were required by other problems. The resulting tool has been useful in several applications; we'll see some in Section 4. (But a quick hint for greedy readers: I typed the last two words in the previous sentence as `Section @APPSEC@`.)

3. The Program

When attending computer conferences and the like I have listended to (and probably delivered) my full share of boring lectures, but there is one class of bore who easily outshines all the others: this is the man who talks in full detail about the way his system has been implemented. — P. J. Brown, *Macro Processors* , page 71.

I'm going to try to outshine even the bores of Brown's nightmares by not only describing the implementation but also giving the complete code. We'll start with the simple version that supports definition and replacement. Here is the complete Awk program:

```
awk '
/^@define[ \t]/ { name = $2
                  $1 = $2 = ""; sub(/^[ \t]+/, "")
                  symtab[name] = $0
                  next
                }
                { for (i in symtab)
                        gsub("@" i "@", symtab[i])
                  print
                }
' $*
```

This program contains two pattern-action pairs. The first pattern recognizes `@define` lines. Its action stores the name, erases the `@define` and name fields and the white space around them, then stores the remainder of the input line in the symbol table (implemented as an Awk associative array); execution then proceeds with the `next` input line. The null second pattern ensures that the action will be executed on all other input lines. The `for` loop iterates over all entries in the symbol table, and the `gsub` globally substitutes replacement values for their names; the `print` statement writes the transformed input line.

In the next version of the program we will add a simple include facility. The input line

`@include` *filename*

is replaced by the contents of *filename* . We will restructure the program around a recursive routine to read files, and add functions to make it easier to extend. Here is the resulting code:

```
awk '
function dofile(fname) {
        while (getline <fname > 0) {
                if (/^@define[ \t]/)
                        dodef()
                else if (/^@include[ \t]/)
                        dofile(dosubs($2))
                else
                        print dosubs($0)
        }
        close(fname)
}

function dodef(  name) {
        name = $2
        sub(/^[ \t]*[^ \t]+[ \t]+[^ \t]+[ \t]+/, "")
        symtab[name] = $0
}

function dosubs(s,  i) {
        for (i in symtab)
                gsub("@" i "@", symtab[i], s)
        return s
}

BEGIN { if (ARGC == 2) dofile(ARGV[1])
        else dofile("/dev/stdin")
      }
' $*
```

If the program is invoked with a single argument, the `BEGIN` block takes that as the name of the input file; otherwise it processes the standard input. The function `dofile` processes a file, `dodef` processes a definition, and `dosubs` applies the substitutions in the symbol table to its input string. The `dodef` function uses a complex regular expression in a `sub` command to remove the first two fields (because setting them to blanks — as in the first version — causes Awk to replace all field separators with a single blank).

So far we have assumed that macro definitions expand into unadorned text. But what happens when the replacement text itself contains further macro calls, as in:

```
@define DIR /usr/jlb/macro.paper
@define PROBSECFILE @DIR@/sec2.in
```

After these definitions, the string `@PROBSECFILE@` should be expanded into `/usr/jlb/macro.paper/sec2.in`. The previous implementation may or may not handle this correctly (details are left as an exercise for Awkophiles). This implementation of `dosubs` handles nested macros by repeatedly expanding the string until no more expansions are made:

```
function dosubs(s,  changes, i) {
        do {
                changes = 0
                for (i in symtab)
                        changes += gsub("@" i "@", symtab[i], s)
        } while (changes)
        return s
}
```

This version is correct but slow; we can speed it up with a guard to check for the common case of no remaining at-signs:

```
                        ...
                changes = 0
                if (s ~ /@.*@/)
                        for (i in symtab)
                             ...
```

Without the guard the program took 5.4 seconds to process one large file; with the guard, the time dropped to 2.3 seconds. The faster version of `dosubs` described in Appendix 1 takes just 0.8 seconds on the same file.

Sometimes you want to make a file that you can conditionally change. As an example of conditional text, we'll consider the arduous task of writing a Ph.D. thesis, which can strain even the best professor-student relationship. A friend of mine organized his thesis so that by setting a given flag he could remove all references to his thesis advisor. The version that he showed his advisor (whom we'll call Professor Newton, to protect the innocent) was compiled from a file like this:

```
@define WANTNEWT 1
   ...
@if WANTNEWT
This area was profoundly influenced by
the groundbreaking work of Professor Newton.
@fi
```

For his private amusement, the poor student could recompile the document after setting `WANTNEWT` to zero. The semantics of the `@if` statement are that the text up to the next `@fi` statement is included if the variable is defined and not equal to zero. To implement the statement we need this function to discard text:

```
function gobble(fname) {
        while (getline <fname > 0)
                if (/^@fi/)
                        break
}
```

We then add these lines to the chain of `if` statements in `dofile`:

```
} else if (/^@if[ \t]/) {
        if (!($2 in symtab) || symtab[$2] == 0)
                gobble(fname)
} ...
```

The complete `m1` program has a couple of additions to this simple conditional. Text may contain nested `if` statements; `gobble` is modified to keep a counter of the current `if/fi` nesting. The `@unless` statement is the complement of `@if`; it includes the subsequent text (up to the same `@fi` delimeter) if the variable is undefined, or defined to be zero.

The final version of `m1` also supports multi-line `@defines`. If a `@define` line ends with the backslash character `\`, the text is continued on the next line (discarding white space before the first text character). To implement long defines we make the minor change to `dodef` to continue reading text as long as lines end with a backslash. We must also make a major change to the input/output structure of the entire program, because macro expansion can generate lines that need to be read by the `dofile` function. The new `readline` function reads a line from the text `buffer` if it is not empty and otherwise reads from the current file. The string `s` can be ‘‘pushed back’’ onto the input stream by concatenating it on the front (left) of `buffer` by the idiom `buffer = s buffer`.

The complete program is adorned with several other bells and whistles. Here are the most interesting and important:

Comments. It is immoral to design a language without comments. Lines that begin with `@comment` are therefore ignored.

Error Checking. The final Awk program has a number of `if` statements that check for weird conditions, which are reported by the `error` function.

Defaults. The `@default` statement is a `@define` that takes effect only if the variable was not previously defined; we'll see its use shortly. We could achieve the same effect with an `@unless` around a `@define`, but the `@default` is used frequently enough to merit its own command.

Performance. When `dofile` reads a line of text unadorned with at-signs, it performs several tests and function calls; the final version adds a new `if` statement to print the line immediately.

The complete `m1` language is summarized in this table:

```
@comment Any text
@define name value
@default name value     Set if name undefined
@include filename
@if varname             Include subsequent text if varname != 0
@fi                     Terminate @if or unless
@unless varname         Include subsequent text if varname == 0
Anywhere in line @name@
```

There are many ways in which the `m1` program could be extended. Here are some of the biggest temptations to ‘‘feeping creaturism’’:

- A long definition with a trail of backslashes might be more graciously expressed by a `@longdefine` statement terminated by a `@longend`.
- An `@undefine` statement would remove a definition from the symbol table.
- I've been tempted to add parameters to macros, but so far I have gotten around the problem by using an idiom described in the next section.
- It would be easy to add stack-based arithmetic and strings to the language by adding `@push` and `@pop` commands that read and write variables.
- As soon as you try to write interesting macros, you need to have mechanisms for quoting strings (to postpone evaluation) and for forcing immediate evaluation.

Appendix 2 contains the complete implementation of `m1` in about 100 lines of Awk, which is significantly shorter than other macro processors; see, for instance, Chapter 8 of Kernighan and Plauger [1981]. The program uses several techniques that can be applied in many Awk programs.

- Symbol tables are easy to implement with Awk's associative arrays.
- The program makes extensive use of Awk's string-handling facilities: regular expressions, string concatenation, `gsub`, `index`, and `substr`.
- Awk's file handling makes the `dofile` procedure straightforward.
- The `readline` function and pushback mechanism associated with `buffer` are of general utility.

4. Applications

Macroprocessors are appealing. They're simple to implement, and you can get all kinds of wondrous effects. Unfortunately, there's no way of telling what those effects are going to be. — B. W. Kernighan.

We'll start with a toy example that illustrates some simple uses of `m1`. Here's a form letter that I've often been tempted to use:

```
@default MYNAME Jon Bentley
@default TASK respond to your special offer
@default EXCUSE the dog ate my homework
Dear @NAME@:
    Although I would dearly love to @TASK@,
I am afraid that I am unable to do so because @EXCUSE@.
I am sure that you have been in this situation
many times yourself.
                Sincerely,
                @MYNAME@
```

If that file is named `sayno.mac`, it might be invoked with this text:

```
@define NAME Mr. Smith
@define TASK subscribe to your magazine
@define EXCUSE I suddenly forgot how to read
@include sayno.mac
```

Recall that a `@default` takes effect only if its variable was not previously `@define`d.

I've found `m1` to be a handy Troff preprocessor. Many of my text files (including this one) start with `m1` definitions like:

```
@define ArrayFig @StructureSec@.2
@define HashTabFig @StructureSec@.3
@define TreeFig @StructureSec@.4
@define ProblemSize 100
```

Even a simple form of arithmetic would be useful in numeric sequences of definitions. The longer `m1` variables get around Troff's dreadful two-character limit on string names; these variables are also available to Troff preprocessors like Pic and Eqn. Various forms of the `@define`, `@if`, and `@include` facilities are present in some of the Troff-family languages (Pic and Troff) but not others (Tbl); `m1` provides a consistent mechanism.

I include figures in documents with lines like this:

```
@define FIGNUM @FIGMFMOVIE@
@define FIGTITLE The Multiple Fragment heuristic.
@FIGSTART@
.PS <@THISDIR@/mfmovie.pic
@FIGEND@
```

The two `@define`s are a hack to supply the two parameters of number and title to the figure. The figure might be set off by horizontal lines or enclosed in a box, the number and title might be printed at the top or the bottom, and the figures might be graphs, pictures, or animations of algorithms. All figures, though, are presented in the consistent format defined by `FIGSTART` and `FIGEND`.

I have also used `m1` as a preprocessor for Awk programs. The `@include` statement allows one to build simple libraries of Awk functions (though some — but not all — Awk implementations provide this facility by allowing multiple program files). File inclusion was used in an earlier version of this paper to include individual functions in the text and then wrap them all together into the complete `m1` program. The conditional statements allow one to customize a program with macros rather than run-time `if` statements, which can reduce both run time and compile time.

The most interesting application for which I've used this macro language is unfortunately too complicated to describe in detail. The job for which I wrote the original version of `m1` was to control a set of experiments. The experiments were described in a language with a lexical structure that forced me to make substitutions inside text strings; that was the original reason that substitutions are bracketed by at-signs. The experiments are currently controlled by text files that contain descriptions in the experiment language, data extraction programs written in Awk, and graphical displays of data written in Grap; all the programs are tailored by `m1` commands.

Most experiments are driven by short files that set a few keys parameters and then `@include` a large file with many `@defaults`. Separate files describe the fields of shared databases:

```
@define N       ($1)
@define NODES   ($2)
@define CPU     ($3)
   ...
```

These files are `@included` in both the experiment files and in Troff files that display data from the databases. I had tried to conduct a similar set of experiments before I built `m1`, and got mired in muck. The few hours I spent building the tool were paid back handsomely in the first days I used it.

5. Conclusions

I've found `m1` to be a useful tool in several applications. If it is close to what you want but doesn't meet your exact needs, the code is small enough for you to consider tailoring it to your application.

Building `m1` has been a fun exercise in programming. I started with a tiny program, and grew it as applications demanded, using the techniques described by Bentley [1988]. It uses several useful Awk techniques, and even had some interesting performance problems. All in all, `m1` provided me with an almost painless way to learn more about macros, one of the grand old problems of computing.

Acknowledgments

I am grateful for the helpful comments of Brian Kernighan and Doug McIlroy.

References

Aho, A. V., B. W. Kernighan, and P. J. Weinberger [1988]. *The Awk Programming Language*, Addison-Wesley.

Bentley, J. L. [1988]. ``Working smarter: Three prescriptions for productivity'', *Computer Language 5*, June 1988, pp. 36-41.

Brown, P. J. [1974]. *Macro Processors and Techniques for Portable Software*, John Wiley & Sons.

Kernighan, B. W. and P. J. Plauger [1981]. *Software Tools in Pascal*, Addison-Wesley.

Kernighan, B. W. and D. M. Ritchie [1983]. ``The M4 macro processor'', in Bell Laboratories, *Unix Programmer's Manual*, Holt, Rinehart and Winston, vol. 2 (seventh edition).

McIlroy, M. D. [1960]. ``Macro instruction extensions of compiler languages'', *Communications of the ACM 3*, 4, pp. 214-220. (Brown [1974, p. 40] describes this as a ``classic paper ... the most important one yet published on the subject of macro processors''.)

Appendix 1: A Substitution Function

Section 3 describes several versions of the `dosubs` function that performs macro substitution. The final version of the program in the next Appendix uses an even faster version of the function, which we will now briefly study.

The idea is to process the string from left to right, searching for the first substitution to be made. We then make the substitution, and rescan the string starting at the fresh text. We implement this idea by keeping two strings: the text processed so far is in `L` (for Left), and unprocessed text is in `R` (for Right). Here is the pseudocode for `dosubs`; the final version is in the next appendix.

```
L = Empty
R = Input String
while R contains an "@" sign do
        let R = A @ B; set L = L A and R = B
        if R contains no "@" then
                L = L "@"
                break
        let R = A @ B; set M = A and R = B
        if M is in SymTab then
                R = SymTab[M] R
        else
                L = L "@" M
                R = "@" R
return L R
```

Appendix 2: The Complete Program

```
awk '
# Modifications to published m1:
#  Fixed bugs in dodef (see comment)
#  Interpret ''^@Mname'' as ''@Mname@''
#  Add ''^@@'' as comment
#  Add ''@ignore DELIM'' -- ignore input until line that begins with DELIM
#  Loop over multiple input files (from /usr/bwk/bin/m1)
#  Fixed andrew-found bug in @fi -- @fine used to terminate, too
# Potential modifications
#  Avoid v10 ''/dev/stdin'' -- see comment in line
#  Add ''@shell DELIM shell line here'' -- pipe output until DELIM through pipe
#  Add @longdef/@longend
#  Add @append MacName MoreText, like troff .am

function error(s) {
         print "m1 error: " s | "cat 1>&2"; exit 1
}

function dofile(fname,  savefile, savebuffer, newstring) {
         if (fname in activefiles)
                  error("recursively reading file: " fname)
         activefiles[fname] = 1
         savefile = file; file = fname
         savebuffer = buffer; buffer = ""
         while (readline() != EOF) {
                  if (index($0, "@") == 0) {
                           print $0
                  } else if (/^@define[ \t]/) {
                           dodef()
                  } else if (/^@default[ \t]/) {
                           if (!($2 in symtab))
                                    dodef()
                  } else if (/^@include[ \t]/) {
                           if (NF != 2) error("bad include line")
                           dofile(dosubs($2))
                  } else if (/^@if[ \t]/) {
                           if (NF != 2) error("bad if line")
                           if (!($2 in symtab) || symtab[$2] == 0)
                                    gobble()
                  } else if (/^@unless[ \t]/) {
                           if (NF != 2) error("bad unless line")
                           if (($2 in symtab) && symtab[$2] != 0)
                                    gobble()
                  } else if (/^@fi([ \t]?|$)/) { # Could do error checking here
                  } else if (/^@stderr[ \t]?/) {
                           print substr($0, 9) | "cat 1>&2"
                  } else if (/^@(comment|@)[ \t]?/) {
                  } else if (/^@ignore[ \t]/) { # Dump input until $2
                           delim = $2
                           l = length(delim)
                           while (readline() != EOF)
                                    if (substr($0, 1, l) == delim)
                                             break
                  } else {
                           newstring = dosubs($0)
                           if ($0 == newstring || index(newstring, "@") == 0)
                                    print newstring
                           else
                                    buffer = newstring "\n" buffer
                  }
         }
         close(fname)
         delete activefiles[fname]
         file = savefile
         buffer = savebuffer
}

function readline(  i, status) {
         status = ""
         if (buffer != "") {
                  i = index(buffer, "\n")
                  $0 = substr(buffer, 1, i-1)
```

```
                buffer = substr(buffer, i+1)
        } else {
                # Hume: special case for non v10: if (file == "/dev/stdin")
                if (getline <file <= 0)
                        status = EOF
        }
        # Hack: allow @Mname at start of line w/o closing @
        if ($0 ~ /^@[A-Z][a-zA-Z0-9]*[ \t]*$/)
                sub(/[ \t]*$/, "@")
        return status
}

function gobble(  ifdepth) {
        ifdepth = 1
        while (readline() != EOF) {
                if (/^@(if|unless)[ \t]/)
                        ifdepth++
                if (/^@fi[ \t]?/ && --ifdepth <= 0)
                        break
        }
}

function dosubs(s,  l, r, i, m) {
        if (index(s, "@") == 0)
                return s
        l = ""   # Left of current pos; ready for output
        r = s    # Right of current; unexamined at this time
        while ((i = index(r, "@")) != 0) {
                l = l substr(r, 1, i-1)
                r = substr(r, i+1)         # Currently scanning @
                i = index(r, "@")
                if (i == 0) {
                        l = l "@"
                        break
                }
                m = substr(r, 1, i-1)
                r = substr(r, i+1)
                if (m in symtab) {
                        r = symtab[m] r
                } else {
                        l = l "@" m
                        r = "@" r
                }
        }
        return l r
}

function dodef(fname,  str, x) {
        name = $2
        sub(/^[ \t]*[^ \t]+[ \t]+[^ \t]+[ \t]*/, "")  # OLD BUG: last * was +
        str = $0
        while (str ~ /\\$/) {
                if (readline() == EOF)
                        error("EOF inside definition")
                # OLD BUG: sub(/\\$/, "\n" $0, str)
                x = $0
                sub(/^[ \t]+/, "", x)
                str = substr(str, 1, length(str)-1) "\n" x
        }
        symtab[name] = str
}

BEGIN {  EOF = "EOF"
        if (ARGC == 1)
                dofile("/dev/stdin")
        else if (ARGC >= 2) {
                for (i = 1; i < ARGC; i++)
                        dofile(ARGV[i])
        } else
                error("usage: m1 [fname...]")
     }
' $*
```