m1 : A Micro Macro Processor

June 13, 2022

Synopsis

```
awk -f m1.awk [file...]
```

Description

M1 is a simple macro language that 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.

M1 copies its input file(s) to its output unchanged except as modified by certain "macro expressions." The following lines define macros for subsequent processing:

```
@comment Any text
@@                              same as @comment
@define name value
@default name value             set if name undefined
@include filename
@if varname                     include subsequent text if varname != 0
@unless varname                 include subsequent text if varname == 0
@fi                             terminate @if or @unless
@ignore DELIM                   ignore input until line that begins with DELIM
@stderr stuff                   send diagnostics to standard error
```

A definition may extend across many lines by ending each line with a backslash, thus quoting the following newline. Any occurrence of `@name@` in the input is replaced in the output by the corresponding value. `@name` at beginning of line is treated the same as `@name@`.

Applications

Form Letters

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.

Troff Pre-Processing

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`.

Awk Library Management

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.

Controlling Experiments

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 `@default`s. 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.

The Substitution Function

M1 uses as fast substitution function. 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`:

```
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
```

Possible Extensions

There are many ways in which the m1 program could be extended. Here are some of the biggest temptations to "creeping 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.

Code

The following code is short (around 100 lines), 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.

error

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

dofile

```
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
}
```

readline

Put next input line into global string `buffer`. Return `EOF` or `""` (null string).

```
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
}
```

gobble

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

dosubs

```
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
}
```

dodef

```
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

```
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...]")
}
```

Bugs

M1 is three steps lower than m4. You'll probably miss something you have learned to expect.

History

M1 was documented in the 1997 *sed & awk* book by Dale Dougherty & Arnold Robbins (ISBN 1-56592-225-5) but may have been written earlier.

This page was adapted from `131.191.66.141:8181/UNIX_BS/sedawk/examples/ch13/m1.pdf`, also available at `http://lawker.googlecode.com/svn/fridge/share/pdf/m1.pdf`. *(Note: Both URLs from the original text are now dead.)*

Author

Jon L. Bentley