Toolman: Generating Web Pages with sh and make, Part 1

by Daniel E. Singer

Dan has been doing a mix of programming and system administration since 1983. He is currently a system administrator in the Duke University Department of Computer Science in Durham, North Carolina, USA.

<des@cs.duke.edu>

Yeah, yeah, I've been hearing plenty lately about using Perl to generate Web pages. But I've been doing the same thing for a while now with the humble ol' shell. Basically, I use the Bourne shell and its intrinsic string-based processing as a macro-processing language to generate HTML. Arguably, Perl or even m4 or various other scripting or macro languages [1] would be better suited for this purpose, but unless your Web pages are going to be fairly complex or extensive, good ol' sh does quite a nice job, thank you. Throw in some appropriate scripting and a makefile, and you can automate processing – and even generate various versions of the same pages, for example, for prototyping or for different network sites.

Dubious as these claims may seem, the techniques I'm going to describe are really quite practical. What it gets down to, though, is that I'm just one of those incorrigible (and unrepentant) sh hackers at heart. So if you've got a hankering to do some shell and make programming to whip out consistent, easily maintained HTML code, please read on. And if you're one who prefers an alternative scripting language, these concepts are likewise applicable.

Due to space considerations, we'll cover this material in two parts; fortunately the content easily accomodates this. In this first installment, we'll talk about the shell, and in the February issue of *;login:*, we'll cover make.

Tutorial Or Not Tutorial?

This article will *not* be a tutorial on shell programming or on writing makefiles (or on writing Web pages, for that matter), but will demonstrate how these tools can be applied in the context of Web page generation. And by generation of Web pages and HTML, I mean the pregeneration of static Web pages only, not dynamic generation or CGI techniques. (The latter are within the realm of possibility, I just haven't messed with it.) Please note that my use of these methods is a work in progress, so you will likely be able to improve upon what is presented here.

Objectives: Consistency, Simplification, Automation, Mutability

We want to satisfy several objectives with our choice of tools for Web page generation. One is to provide consistency to the appearance of, and the data in, the pages. This can be accomplished by the use of variables to hold definitions, such as colors and addresses, and by the use of user-defined functions to generate constant or similar sections of HTML, such as page headers and footers. Another objective is simplification. Again, shell functions can be used to consolidate much of the tedium of writing, say, an HTML table. A third objective is automation – having to go through as few steps as possible to get from one state to the next, in particular, from our source files to our final HTML documents. A fourth objective is mutability: we might want to create different versions of the same Web pages, for instance, for two separate servers at separate sites. make becomes useful for these latter two objectives.

Why Shell?

I never really thought of the shell as a macro processor until I started using it in this context. But when you get down to it, the shell is very heavily built around string processing [2], which is what macro processors are all about, right? A line in a script is interpreted as a command only after variables and various other substitutions are interpolated. A shell variable behaves much like a simple macro. And a shell user-defined function can be tantamount to a macro with parameters. Plus you get "all that good stuff," the usual benefits of the shell: control structures, pipes, easy access to external commands, environment variables, etc. Personally, I find HTML somewhat tedious to write. Even comments are awkward, and are much easier in shell syntax. For me, the shell provides a much more comfortable (familiar?) style.

. . .a shell user-defined function can be tantamount to a macro with parameters.

Here's a simple example of how you might use shell code to generate some HTML. A variable can be defined to hold a value or some HTML code:

```
HAPPY="<EM>Let's get happy!</EM>"
```

and then that value can be written out at any following point in the shell code with:

```
echo "$HAPPY"
```

Unless you're planning on saying that a lot of times, this construct isn't going to be terribly productive. A more useful approach is to use a function, such as:

```
emphasize() {
  echo "<EM>$*</EM>"
}
```

This function can then be invoked with:

```
emphasize "Let's get happy!"
```

The `$*` in the function body is replaced by the parameter(s) to the function, and the function is reusable with other text. And here's an even more general form:

```
tag_data() {
  _TAG="$1"
  shift
  echo "<$_TAG>$*</$_TAG>"
}
emphasize() {
  tag_data EM "$@"
}
```

And if we *were* going to be saying it alot, then adding this might even make sense:

```
get_happy() {
  emphasize "Let's get happy!"
}
...
get_happy
```

I hope you're starting to get the idea.

How the Processor Works

One approach to using the shell as an HTML-generating macro processor is to write a master script consisting of your standard definitions (variables) and macros (functions), and then have the master script *source* (the shell's period "." built-in command, see `man sh`) the files that define individual Web pages, producing a browser-ready HTML output file for each input file.

So, to elucidate, each Web page is derived from a source file of shell code consisting largely of calls to the functions defined in the master script, and possibly redefinition of

some of the variables and/or functions. Unique features for each Web page are specified through parameters to functions, variable redefinitions, choice and order of function calls, and/or specific HTML coding. Optionally, local functions and other special processing can also be added to the source files.

Here's some code from a script named `gen_html.sh` that I used recently to help generate a group of HTML files for a talk. The script begins with about 700 lines of variable and function definitions, option processing, etc. (Many of the variables can be overridden by variables in our HTML source files or by environment variables; the potential here for wrapper scripts is strong.) Finally, the loop below occurs at the end of the script. In it, we just process any arguments left on the command line, which should all be the names or basenames (suffix omitted) of HTML source files. (Actually, by this point, the arguments have already been scanned once to look for errors, and, if there were no arguments, a "-" indicating standard input would have been pushed onto the positional parameters.) Each input file is processed, and a corresponding HTML output file is produced:

```
##
## process (source) each input file
##
for FILE do
  case "$FILE" in
   # for stdin, make a tmp file and just process to stdout
   -)
     : ${UPDATE_DATE=`date '+%e %B %Y'`}
     HTML_FILE="???.$OUT_SUFFIX"
     [ "$QUIET" = 0 ] && echo "$PROG: processing stdin..." >&2
     cat > "$TMP_FILE"
     . "$TMP_FILE"               # source it
     rm -f "$TMP_FILE"
     ;;
   # otherwise, process and output to filename.html
   *)
    case "$FILE" in
    *."$IN_SUFFIX")
     OUT_FILE=`echo "$FILE" |
     sed 's/.'"$IN_SUFFIX"'$/.'"$OUT_SUFFIX"'/'`
     ;;
    *)
     OUT_FILE="$FILE.$OUT_SUFFIX"
     if [ ! -f "$FILE" -a -f "$FILE.$IN_SUFFIX" ]; then
      FILE="$FILE.$IN_SUFFIX"
     fi
    esac
    HTML_FILE="$OUT_FILE"
    [ "$QUIET" = 0 ] && echo "$PROG: processing $FILE" >&2
    {
     UPDATE_DATE=`get_update_date "$FILE"`
     case "$FILE" in
     */*)
       . "$FILE"                 # source it
       ;;
     *)
       . ./"$FILE"               # source it
     esac
    } > "$OUT_FILE"
   esac
  done
exit 0
```

This loop could perhaps be simpler, but, as written, it can work as a filter processing from standard input to standard output, and it can handle filenames with or without a source file suffix. Some date processing also takes place so that a "Last updated:" timestamp based on file modification time can be added to each page. (You know how hard it otherwise is to remember to manually update those dates!) The total length of this script might sound excessive, but it can be reused with minimal modification, and so the initial investment can pay off repeatedly.

This script might be invoked with a command line like:

```
./gen_html.sh index part1 part2 notes
```

Here's a very trivial example of what an HTML source file might look like:

```
## @(#) test.hsrc
## 9/98, D.Singer
begin_doc
begin_head -t "This is a test of \`gen_html.sh'"
end_head
begin_body -bg "skin.jpg"
do_break
heading -C 1 "This Is A Large Title"
heading -C 4 "and this a more subtle title"
do_break
do_hrule -s 6 2
do_break 2
begin_center
begin_font -s +3 -c red
emphasize "Thanks for coming!"
end_font
end_center
do_break 2
do_hrule -s 6 2
do_break
do_last_update
do_break
do_footer
end_body
end_doc
## end of hsrc file
```

And here's the resultant HTML output (slightly altered to conserve space):

```
<HTML>
<!-- html document <www.cs.duke.edu/~des/workdir/test.html> -->
<!-- generated on Mon Sep 21 00:09:31 EDT 1998 -->
<!-- via 'gen_html.sh' -->
<HEAD>
<BASE HREF="http://www.cs.duke.edu/~des/workdir/">
<TITLE>This is a test of `gen_html.sh'</TITLE>
</HEAD>
<BODY
BGCOLOR="#fffff0"
BACKGROUND="skin.jpg"
TEXT="#000000"
LINK="#339999"
ALINK="#BBBB11"
VLINK="#336060">
<BR>
<H1 ALIGN=CENTER>This Is A Large Title</H1>
<H4 ALIGN=CENTER>and this a more subtle title</H4>
<BR>
```

Got a tool that's useful, unique, way cool? Please send a description to <Toolman@usenix.org>.

Using the shell as your HTML generator provides all of the features and power of the shell, and of course the whole toolbox of UNIX utilities . . . that comes along with it.

```
<HR SIZE=6><HR SIZE=6>
<BR><BR>
<CENTER>
<FONT SIZE=+3 COLOR=red>
<EM>Thanks for coming!</EM>
</FONT>
</CENTER>
<BR><BR>
<HR SIZE=6><HR SIZE=6>
<BR>
<FONT SIZE=-1>
Page last updated: 21 September 1998
</FONT>
<BR>
<FONT SIZE=-1>
URL: http://www.cs.duke.edu/~des/workdir/test.html
</FONT>
<BR>
<FONT SIZE=-1>
Copyright &copy; 1998, Daniel E. Singer. All rights reserved.
</FONT>
</BODY>
</HTML>
<!—- end of generated document -—>
```

As you can see, the master script can also generate HTML comments for each Web page to provide standard identification blocks or other metadata. A more complete example would include tables and other snazzy features. Unfortunately, we don't have the luxury here of the space that would be required. But I can't resist at least showing you what some HTML source might look like for a table:

```
begin_table -cols 2 -w 80% -C
table_data -fs +2 -R "<EM>Row 1"
table_data -nobr "This table has two columns, 80% width, and is
  centered. The first column is right justified and has a bigger
  font. For the second column, we're forgoing putting a break
  between each line."
table_data -fs +2 -R "<EM>Row 2"
end_table_row
table_data -fs +2 -R "<EM>Row 3"
table_data -nobr "The second row only had 1 column."
end_table
```

As the code shows, this table is self-documenting! It's still a long way from WYSIWYG, but I like it better than the HTML that it will generate. See the end of this article for a URL for a sample `gen_html.sh` script; it will include the HTML-table-generating shell code.

Other Benefits of the Shell Approach

Using the shell as your HTML generator provides all of the features and power of the shell, and of course the whole toolbox of UNIX utilities (`grep`, `sed`, `awk`, `ls`, `date`, etc.) that comes along with it. An example of this is a situation where I use a flat database text file (lines with tab-separated fields) along with a supplementary script to produce multiple Web pages, each containing a list derived from the database, each sorted on a different key, and each providing hypertext links based on one of the fields. Believe me, this is much easier than maintaining these separate HTML pages by hand. For the situation I have in mind, a list of alumni email addresses gets sorted by name and by class (that is, year). And a similar database exists for Web addresses. Here's how it works.

First, there's the database (a text file) that looks like this (the names have been changed to protect the innocent):

```
# email-addrs.db
# @(#) /u/des/public_html/sf/email-addrs.db 1.29
Abbott, Gail      gabb@mail.ced.net  1991
Adams, Albert     adman@fisheads.com 1976
Adams, Fred       adams@nunez.org 1982
...
```

A supplementary script named `cvt_addrs.sh` reads the database, and then, depending on selected options, writes out the records in HTML form, sorted, with email addresses converted to `mailto:` links, and with section `NAME` anchors added. In the appropriate spot in the HTML source file, the line:

```
./cvt_addrs.sh -f email-addrs.db
```

invokes the script, and inserts the HTML data derived from the database sorted by last name. In another HTML source file, the same script is called with the addition of the `-c` option to get a sort by class. Additional code in each of these HTML source files generates the jump lists used to go to a `NAME` anchor for a particular year or letter of the alphabet. For instance:

```
echo "<A"
NEXTA="-<A"
for LETTER in A B C D E F G H I J K L M N O P Q R S T U V W X Y Z;
       do
  [ "$LETTER" = "Z" ] && NEXTA=
  echo "HREF=\"email-addrs.html#$LETTER\">$LETTER</A>$NEXTA"
done
```

This is much more concise and maintainable than the equivalent written-out HTML.

Other Shells

Of course, this could all be done with shells other than Bourne shell, such as `ksh`, `bash`, or even `zsh`. In fact some of these would probably make the job easier, as Bourne tends to be a bit archaic in some ways, and they are worth investigating. (I'm just too lazy.) I do tend to avoid the `csh` derivatives for scripting, but if you don't want to take my advice on this one, well, let's just say you're on your own . . . .

Until Next Time

That's it for the shell half of our discussion. To get the whole scoop, you'll just have to bite your fingernails in anticipation until the February issue of *;login:*, when we'll explore how `make` plays an integral role in this process, providing the significant capabilities of automation and mutability. Please tune in next time.

URLs:

<http://www.cs.duke.edu/~des/toolman.html>
<ftp://ftp.cs.duke.edu/pub/des/scripts/gen_html/INDEX.html>

Notes

[1] I've even seen a recent article about using `cpp` (the `C` pre-processor) and `make` to do something very similar (Jim Fox. "Unity Among Web Pages" in *SunExpert Magazine*, August 1998, pp. 42-45.), though there are some substantial differences in style and content between these two approaches. But, shoot, for us shell diehards, well, need I say more?

[2] This is despite Bourne shell's notable lack of built-in string processing functions. For many operations, it is necessary to engage external commands such as `awk`, `sed`, `expr`, etc.

Of course, this could all be done with shells other than Bourne shell . . .