by PETER COLLINSON, Hillside Systems

Automatic Web Page Creation

My first articles on the topic of HTML and the Web were published in *SunExpert* in 1994 ("The World Wide Web," September, Page 22; "HTML," October, Page 28). At that time, the only way to create Web pages was to use a standard text editor, typing your page markup into the machine. Since then, we've seen the emergence of a great number of differently flavored visual HTML editors, including add-ons to the mainstream browsers.

Undoubtedly, the new editors have opened the Web publishing door to many people for whom creating page markup using raw text seems an alien and difficult task. Like many children, my son Glyn has grown up in an age of visual computing, with the Windows environment forming part of his basic toolkit for life. He finds the notion of programming the look of the page using text gets in the way of the ideas he wants to put across. However, he's been happily using a mixture of Quarterdeck's add-on for Word for Windows and Netscape Gold to create visually exciting pages about his favorite computer game.

Sadly, his ability to generate pages using completely visual means often has to be augmented by my HTML knowledge. We jointly make minor changes to the HTML text that are needed to generate some effect that we both know can be achieved. I still haven't come across a visual HTML page editor that works 100% of the time. The editors either cannot generate "modern" HTML because the goal posts keep moving at a phenomenal rate, or they simply don't render the HTML accurately. Even Netscape Gold doesn't seem to be able to accurately position aligned graphics when you are using its editor. It's worse to have an editor that is supposedly WYSIWYG but displays the output incorrectly.

Of course, the effect of visual page editors is to remove the need for their users to have any knowledge of the underlying HTML. It becomes easy to create complex HTML markup without comprehending the meaning of the text that is produced. I don't object to this. After all, I drive my car with no great understanding about how it works or how to fix it when it breaks down. When the car breaks, which is rare, I call in the experts and get them to repair it. However, when I was my son's age, I think car drivers were expected to know much more about the internal workings of their vehicles than we need to know today. User pressure for reliability, for cars that don't break down, has forced the manufacturers to build vehicles that are "more user friendly." The same user-driven process of making it easier for the user to know less is happening with HTML editors.

However, these visual editors suffer from the same problem that I described last month. They don't scale. When I set out with my text editor to create my Web-based tour of the City of Canterbury in 1995, I was commencing the generation of a huge number of pages. At the end of the summer of 1995, the Tour was 100 pages. By the end of the summer of 1996, it had grown to 350 pages. The pages all

MICHELLE FRIESENHAHN WILBY

have a consistent set of elements, by which I mean that the pages display the same set of objects in roughly the same positions. They each have a title, a photograph, text, some navigation buttons and some standard links to other pages.

Some of these objects are the same on every page. For example, each page has the same set of links to other facilities on my server, like a clickable map or the search engine. Although most of the objects vary in content from page to page, most of the HTML that is used to place the object on the page remains the same. So, for example, the HTML that inserts the photograph onto the screen is nearly the same on every page. Only the file name and the size information changes, specifying the image that pertains to that page.

So back in 1995 when I started the project, I had typed three pages before realizing that what I was doing was a waste of time. There was a consistent pattern to the pages, and I could use UNIX tools to help me to minimize my input. I could easily create a system where I supplied only the content and generated the HTML automatically. The system would remove the possibility of making random human errors when entering the HTML for any one page. The HTML would always be correct, because it would be generated automatically.

Another payoff came at the start of 1996, when I decided to restyle all the pages in the light of experience. I worked on the new style to ensure it was correct, and then simply regenerated all the pages using the new page layout. I've made global changes like these several times now. You simply cannot contemplate making sweeping changes if your main input tool is a visual editor and you have to make edits to each page individually.

m4 Is Key

The key to the page generation system is the `m4` macro processor. A *macro processor* is a text replacement engine that looks through a text file for known strings. Once the processor has found a string, it will replace that string with other text. The replacement operation is often known as an *expansion* because early macro processors were used by assembly language programmers. Macro processing allowed the programmer to grow some simple syntax consisting of a few words into a complex set of machine instructions.

Macro processors evolved from their roots in assembly language programming to become general-purpose programs. For example, ML/1 was a general-purpose macro processor that could replace arbitrary strings in the source with other arbitrary strings from a set of definitions. ML/1 was written by Professor Peter Brown from the University of Kent in the UK.

However, the general-purpose programs are often difficult to use. The problem is one of syntax. It's necessary for the program to recognize some of its input text as instructions for the processor itself. Minimally, you need to be able to define macro names and the string that will replace those names in the source text. To introduce the definition, you'll need to put something in the text that the processor can understand as the start of a macro definition. This "something" is conventionally a keyword. Herein lies the problem: How is that keyword recognized? What happens if the keyword occurs naturally in the text? Various solutions of varying complexity emerged to solve these problems.

The `m4` macro processor is not a general-purpose macro processor. It grew from the C language preprocessor, which itself was a macro processor designed originally to permit programmers to insert constants in their code using a name rather than just baldly writing a number. The intention of the C preprocessor was to make the C code more readable.

The `m4` processor "knows" that it is running on some text that looks like a C program. It can pick out individual words or numbers from the text as long as those words look like identifiers in the C language. The names must start with a letter and consist only of alphanumeric characters (and underscore). Commands to `m4` look like a function or routine call in C. To define a macro that replaces "Jack" with "Jill," you will say:

```
define(Jack, Jill)
```

Then when the processor finds the word "Jack" in its input text, it will replace it with "Jill." The `define` here is a reserved word, so the source text cannot contain the word `define` unless it introduces a macro definition. There are several predefined macro names in `m4` that cannot appear in the input text. Inspect your manual page to find a list. However, if these names do appear "naturally" in the source text and you are not using the identically named `m4` function, then you can remove the special meaning of the keyword by using the `undefine` macro.

Like all `m4` keywords, the `define` keyword is a macro and can be found anywhere in the input stream being processed. However, any new line that follows the `define` statement will be added to the output, so it's usual to add the magic token `dnl` to the end of the definition. This means "delete from here up to and including the new line." The `dnl` keyword is often used as a comment statement, causing `m4` to delete the whole line.

A fundamental feature of macro processing is the idea of *rescanning*. When the processor replaces some text, it looks again at the new text to attempt to apply its definitions to change the text again. So an `m4` input file of

```
define(Jack, Jill)dnl
define(Jill, Hill)dnl
Jack and Jill
```

would result in the single line being output:

```
Hill and Hill
```

The `Jack` in the output is replaced by `Jill` and then by `Hill`, while `Jill` is simply replaced by `Hill`.

There's one further complication. When `m4` reads the `define` statement, it will scan it looking for possible replacement text. It will replace any tokens that it finds in arguments to the statement. Sometimes, the replacement is convenient. Mostly, it is not. It's possible to avoid any text replacement by

quoting the arguments. I tend to always quote both the first and second arguments to define just so that I know exactly what is going on.

By default, the quote characters are different, and m4 uses open (`) and close (') single quotes. However, open and close single quote characters often appear in HTML. So I always start my m4 definition sequences by using

```
changequote({,})dnl
```

This sets the quoting characters to open and close braces, which rarely appear in HTML.

HTML Using m4

Well, by now I hope that you are beginning to get the idea. The plan is to create a file that is a template of the HTML pages I want to generate. The template file will contain m4 tokens at any point where there is variable text. So for a simple set of pages, I might have a template file like this:

```
<html><head>
<title>TITLE</title>
</head>
<body>
<h1>PAGETITLE</h1>
<p>
<img src=IMAGE SIZE align=left
     hspace=8 vspace=4>
TEXT
<br clear="left">
<hr>
Last changed on: DATE
</body></html>
```

Here, I've capitalized all the keywords that I expect m4 to replace. Also, using lowercase in the HTML tags allows me to use m4 macro definitions like TITLE, while making sure that the HTML <title> is not touched.

To expand this template, I'll create an m4 file like this:

```
changequote({,})dnl
undefine({index})dnl
define({TITLE},{Sample})dnl
define({PAGETITLE},{Sample page})dnl
define({IMAGE},{"sample.gif"})dnl
define({SIZE},{width=50 height=100})dnl
define({DATE},{7 December 1996})dnl
define({TEXT},{<p>
This is a sample automatically generated
page.})dnl
```

Incidentally, I'll always remove the m4 built-in definition for index. I have many links on my pages that point to index.html, and these links are changed to -1.html unless I explicitly remove the internal meaning of index. Also, the real template file for my Canterbury Tour is considerably more complex.

I can create many of these description files and pass them into m4 along with the template file to generate pages that are displayed to the user. For each page, all I do is say

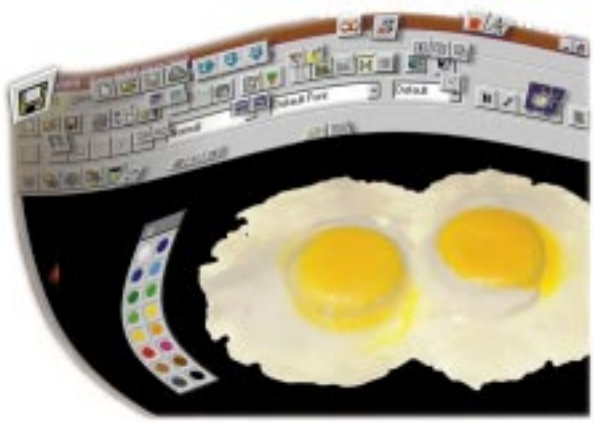
```
m4  samp.desc template.m4  > out.html
```

Preprocessing

Well, I can hear you saying, "that definition file is not a great advance on readability." It's stuffed with curly and round brackets. It's hard to see the wood for the trees. Well, I agree with you. I actually want to be able to input the description file in a much simpler way. I like files that have keywords and values, so I would start by creating a description file like the following:

```
TITLE: Sample
PAGETITLE: Sample page
IMAGE: sample
SIZE: width=50 height=100
TEXT:
<p> This is a sample
automatically generated page.
```

With this type of file, it's easy to see the data and the keyword fields that will supply values on the final HTML page. Also, when you make a new file, it's simple to have an empty stub file lying around that you pick up into your editor and fill in.



However, this keyword file does have to be turned into a set of m4 definitions. I'll use sed to do this. The entire processing sequence will have two stages. First, the keyword description file is passed through sed to create the m4 definition file. Second, the definition file and the template are given to m4, allowing it to create the final HTML file. Of course, all the code is placed into a small(ish) shell script that will do the job.

I am making a couple of assumptions about the description file. First, I assume that all the text in the file from the TEXT keyword to the end of the file will replace the TEXT keyword in the template page. Second, the description file doesn't contain any dates. I plan to generate the "Last changed" date field on the HTML page automatically in the processing script by calling the UNIX date program with appropriate arguments.

With these assumptions in mind, I can start writing the script. Once I've set up the quote characters that I intend to

use, and removed the built-in index macro, I'll call sed:

```
#!/bin/sh
echo 'changequote({,})dnl
undefine({index})dnl' > m4defs
sed -e '
```

The sed program follows the -e option to the command. The sed program is quoted using single quotes meaning that the shell will not look in the string to find shell variables that it can replace. Using single quotes allows me to use an unquoted dollar sign in the sed program.

Here's the next line of my shell script, which is also the first line of the sed program:

```
/^TITLE: /s/^TITLE: \(.*\)$/
   define({TITLE},{\1})dnl/
```

The text should actually be input on a single line, but I have wrapped it for printing. The sed command has two parts, a line selector and a substitute command. The selector is a regular expression that matches lines starting with the word TITLE followed by a colon and a single space character. It will only match the first line of the description file.

Once sed has found the line, it will execute the substitute command to create an m4 definition. It needs to extract the part of the input line that is the active data, and makes its selection using the regular expression on the left-hand side of the substitute command. This expression first matches the start of the line (^), the string TITLE: and a space. Then there is a section of the expression marked by quoted brackets. The source text that matches the expression inside the quoted brackets will be stored. Inside the brackets, the expression dot-star (.*) matches any character (the dot) repeated zero or more times (the star). Matching will stop when the end of the line is reached ($) that is outside the quoted brackets.

The effect of this substitute command is to replace the whole source line with new contents specified by the right-hand text string. However, the \1 on the right-hand side is first replaced by source text that was matched inside the bracketed expression on the left-hand side.

We can construct similar sed lines for the page title, image and size definitions:

```
/^PAGETITLE: /s/^PAGETITLE: \(.*\)$/
   define({PAGETITLE},{\1})dnl/
/^IMAGE: /s/^IMAGE: \(.*\)$/
   define({IMAGE},{\1.gif})dnl/
/^SIZE: /s/^SIZE: \(.*\)$/
   define({SIZE},{\1})/
```

Again, I am wrapping the lines for printing. Notice how I supply the file name suffix .gif for the image. Actually, in the Canterbury Tour, all the images are photos that are a constant size. However, some of the photos are in landscape format and some in portrait. So, rather than having to specify the size of the image repetitively, I have defined two keywords PORTRAIT and LANDSCAPE that are processed by

```
/^PORTRAIT: /s/^PORTRAIT: \(.*\)$/
   define({IMAGE},{\1.gif})dnl\
   define({SIZE},{width=256 height=384})dnl/
/^LANDSCAPE: /s/^LANDSCAPE: \(.*\)$/
   define({IMAGE},{\1.gif})dnl\
   define({SIZE},{width=384 height=256})dnl/
```

There's an assumption that I either use IMAGE and supply a SIZE, or use LANDSCAPE or PORTRAIT to specify the picture to go onto the page.

I can now deal with the TEXT command. I completely replace the TEXT line in the source file by the start of the m4 macro definition and fix up the end of the definition after sed has run.

```
/^TEXT:/s/^ *$/define({TEXT}, {/' $1 >> m4defs
```

The single quote marks the end of the sed program. The program is used by sed to process a file whose name is taken from the first argument to the shell script. Dollar signs introduce shell variables, and $1 is a *positional variable* that is replaced by the first argument to the script. We expect this argument to be the description file. The sed program is run changing the description file into a bunch of m4 define statements. All the characters that occur after the TEXT marker in the file will be copied to the m4defs file. It is passed through m4 and so may be subject to text replacements. I can now finish the script remembering that the last text in the file needs some ending characters to complete the define statement. I also add the date specification to the file.

```
echo '})dnl' >> m4defs
date=`date '+%e %B %Y'`
echo "define({DATE},{$date})dnl" >> m4defs
m4  m4defs template.m4
```

The TEXT define is closed with the first echo command. The next line in the script sets up the date shell variable to the string that is the current date, using the back-quote operator to run a command, and captures its output.

The last echo line adds the m4 definition for the date to all the previously stored lines. Finally, I call m4 to generate the HTML on the standard output of the script.

Some Conclusions

Well, I hope that gives you a flavor of the automatic approach. I guess that you may still have some questions. First, why am I describing a two-stage system where step one makes sed generate m4 definitions and step two uses m4 to process the template file? Why not use a single-stage process that employs sed to directly edit the template file to generate the HTML?

One part of the rationale for using two stages is demonstrated by the ease with which I used the PORTRAIT and LANDSCAPE keywords to generate alternative settings for some objects in the template file. To be fair, some of the functionality has not been demonstrated by this article. I have used the rescanning replacement ability of m4 to gener-

ate the correct code for the navigation arrows on the Canterbury Tour. The navigational arrows change from visible bitmaps to blank spaces depending on the links that are available on that page. The names of the appropriate image files are generated automatically. Finally, splitting the decoding of description files into two separate stages seems to be good engineering.

Another question that arises is, why do I generate static pages in the first place? Why don't I place the page generation code into a CGI script and send the page image to the user dynamically? Well, yes, this could be done, but I've resisted the idea. I am conscious that each page on the Canterbury Tour is perhaps 30 KB of data. I want to get that data to the user as fast as I can, before they get bored and surf off. If I use CGI scripts then I am not shipping data as fast as I can; I am computing that data and causing some finite extra delay. Also, dynamic pages cannot take advantage of the page caches that store my pages closer to my reader, giving them faster access to the images.

Finally, isn't the Canterbury Tour a special application? It's perhaps unusual to have lots of repeated pages. Can the approach be used for whole sites? The answer is a firm yes, at moment anyway. If you look about the Web, you'll find that Web designers usually generate a complete look that pervades all the pages on that site. The look of the pages is part of the site's identity. Until style sheets become more available (and perhaps even after that) using a template approach can give a site the flexibility to generate the site look independent of the contents of each page. I've successfully used this approach with the Canterbury Web Services site, where every page is created from a description file using a common template.

Getting More Information

You'll find the Canterbury Tour located at `http://www.hillside.co.uk/tour`. Glyn's pages are found at `http://www.hillside.co.uk/glyn`, and the Canterbury Web Services site is at `http://www.cantweb.co.uk`. These sites are on the same machine and share the Apache server.

At the Hillside site, you'll also find a page that lists all the articles I have written for *SunExpert*. I place relevant links on this page for all the articles I write, so it's worth a visit if I mention URLs; it will save you typing them in. On the entry for the article you are reading now are some links to the sample page generated by the script above. There are also links to the full text of the script, with the template and description files. The URL for my *SunExpert* page is `http://www.hillside.co.uk/articles/sunexpert.html`. You don't have to type all this in because there's an obvious route to this page from the welcome page on my server.

Peter Collinson runs his own UNIX consultancy, dedicated to earning enough money to allow him to pursue his own interests: doing whatever, whenever, wherever… He writes, teaches, consults and programs using Solaris running on a SPARCstation 2. Email: `pc@cpg.com`.