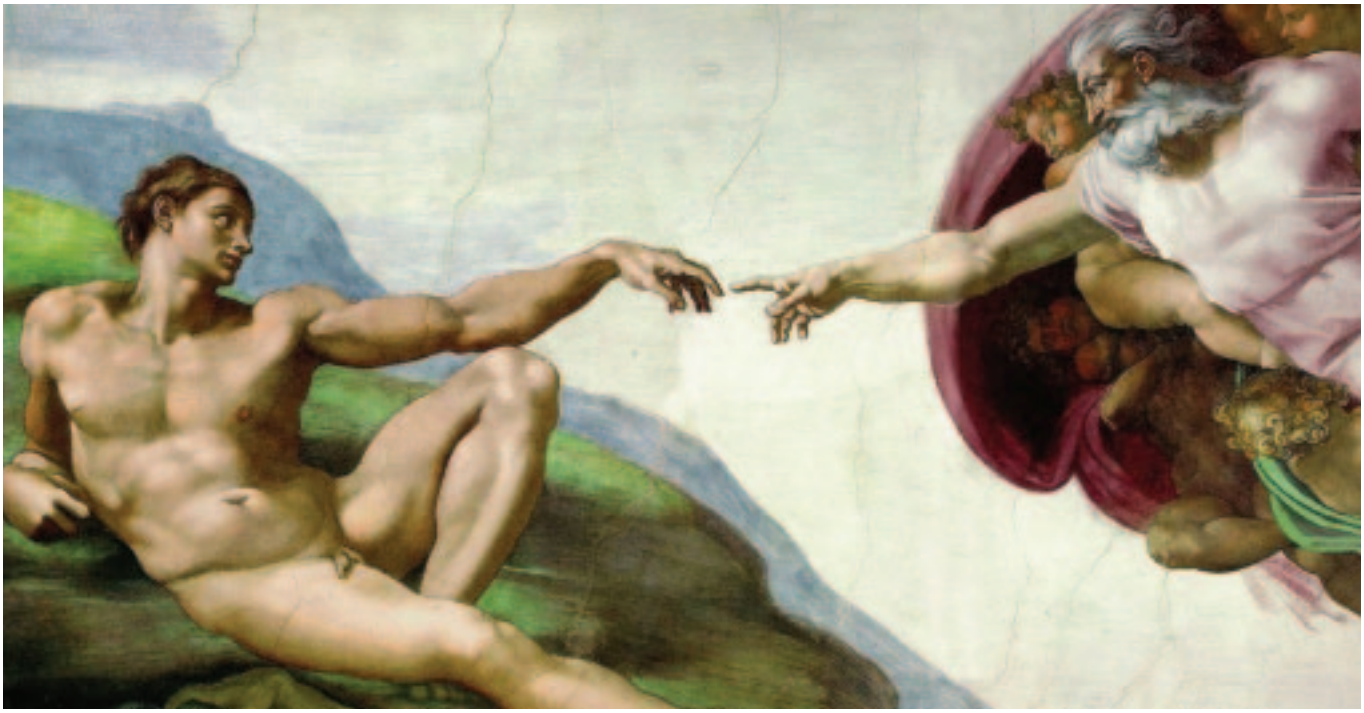
GNU m4

Creating HTML pages

m4 is a macro language used for text processing. It simply copies its input to its output, while expanding built-in or user-defined macros. It can also capture output from standard Linux commands. BY STIG BRAUTASET



We will cover the basics of m4 by separating out content and layout of HTML web pages. The techniques shown are not by any means restricted to HTML, but are applicable elsewhere as well.

Before we begin...

Very basic knowledge of HTML would be beneficial, but should not be necessary to read and comprehend the material covered in this article. It is m4's capabilities as a macro processor language the author wishes to communicate; HTML was chosen because it is something most people should be at least vaguely familiar with.

What is m4?

m4 is a macro processor used by Sendmail and GNU Autoconf, etc. Sendmail relies on m4 for creating its (in)famous sendmail.cf configuration file while Autoconf uses m4 to create the configure script familiar to anyone installing software directly from source. As mentioned earlier, m4 allows us to easily define our own macros, and this is the feature we will focus on. We will learn how to hide layout specifics, long-winded code or arcane syntax behind our own simple macros.

How m4 can help you write HTML

There are no (to the author's knowledge) ready-made macros for writing HTML, so we we will have to write them ourself. At this point you may ask "What's the point then? I may as well be writing the HTML code in the usual way, instead of using this m4 mumbo-jumbo!" The astute reader would, of course, be 100% correct in this observation. However, read on as the benefits will be explained in the next couple of paragraphs. Consider a snip of HTML you write a lot; a simple example is the code for creating links. Chances are it will be like this:

```
<a href="http://www.w3.org">➥
www.w3.org</a>
```

Naming conventions

When splitting the content from the layout of web pages, the author prefers to call the common macro file for html.m4. The content of each page goes in a file with the ending ".mac", e.g. index.mac and so on. When explaining this, however, we develop our macro and .mac files bit by bit, so we need to refer to several different files. Thus it is convenient to name them first.m4, first.mac and so on.

See the sidebar "Joining the content and layout" how to create the resulting HTML file from the macros and the content.

Now, what if we instead define a simple macro that will allow us to write

```
__elink(www.w3.org)
```

and let m4 do the tedious job of filling in the necessary bits? That's less than half the number of characters already. Additionally, observe that we only had to write the link name once, so there is less chance of us spelling it wrong.

Another example is if we have a note on our page telling visitors the date of the last update. It is tedious work searching through all the files you have changed, searching for and updating date stamps. Instead we can simply define a macro named, say, __today that will expand into today's date. The search-and-replace business will then be taken care of for us automatically. How to do this will be shown later; we need to take care of the basics first.

Getting your hands dirty

As mentioned earlier, m4 allow us to define our own macros with ease. The command to let us do this is cunningly named define. Here's how to define a macro to let us use the link shorthand above:

```
define(__link, ➥
<a href="http://$*">$*</a>)
```

The part before the comma (but inside the parentheses) is the macro name, and the part after the comma is the macro body. We will refer to the whole line as the macro definition. The definition line above in English: "Define a new macro named __link. Everywhere where this macro occurs, substitute the macro name for the body of the macro, but substitute the macro's arguments (whatever is inside the parentheses following the macro name) for "$*" wherever "$*" occurs in the macro body."

We will store the macros we write ourselves, such as the above, in a file called html.m4. Se sidebar "Comments: documenting our macros" for details about how we can mix comments and macros in this file.

It's worth noting that macro names do not have to start with two underscores. It is just a convention, because we need to make sure that we do not pick a string that naturally occurs in the text. Otherwise we may get spurious replacements of the macro.

Multiple arguments and quoting

The define command we used above expects two arguments; the new macro name, and what to replace the macro name with. Considering again the example with the __link macro above, what should we do if we don't want to use the URL as the visible, "clickable" link? We could simply create a new macro that takes several arguments, and invoke it thus (in context this time, just to show that it is possible): "Here is __link2(www.w3.org, a link to w3.org). It is an informative website." Here's how to define such a beast:

```
define(__link2, ➥
<a href="http://$1">$2</a>)
```

The only change is that we now have "$1" and "$2" instead of "$*". "$1" and "$2" refer to the first and the second argument of our macro. The arguments are separated by the comma character.

So, Sherlock, what now if we want to create a macro that can take an argument with a comma in it? That, my good Watson, is simple. We just have to put quote the argument. When you quote something, everything between the quote-characters will be treated as a single argument, even if it consist in entirety of a string of, say, 90 commas. The default opening quote is a "`" (back-tick) and the default closing quote is a "'" (single quote). We can now invoke __link2 with a comma in the second argument thus:

```
__link2(www.gnu.org, ➥
'www.GNU.org, the home of much ➥
software')
```

The second comma is now quoted, so the macro is indeed invoked with only two arguments. Note that only one layer

Comments: documenting our macros

If a macro is not self-explanatory, we would like to put an explanatory comment along side the macro definition. m4 naturally allows us to do this; it provides the built-in dnl which reads and discards all characters, up to and including the first newline.

```
dnl I am an example comment.
dnl I am highly unhelpful,
dnl but 100% correct.
```

Using dnl as part of a string does not exhibit this behaviour.

Joining the content and layout

Here's how you create the resulting index.html from the macro definitions in html.m4 and the content in index.mac:

```
$ m4 html.m4 index.mac >
index.html
```

This invokes the m4 processor with two arguments. The m4 command will take the macro definitions it finds, do the necessary substitutions and output the result on its standard output. However, we make use of the shell's redirection facilities to make the output go to a file instead of the screen (if this makes no sense to you, just tag along and follow the directions, but you should consider reading up on shell basics). Now open first.html in a browser, and voil! We have a web page!

Listing 1: sample.html

```
<html>
<head>
<meta http-equiv="Content-Type"➥
content= "text/html; charset=iso-➥
8859-1">
<meta name="description" ➥
content="Sample HTML page">
<meta name="keywords" ➥
content="gnu m4 html">
<meta name="author" content="➥
Stig Brautaset">
<title>sample html page</title>
</head>
<body>
<p>Hello, World</p>
</body>
</html>
```

Listing 2: first.mac

```
__title({Hello World, HTML ➥
version})
<h1>Hello World</h1>
<p>Look at this link: ➥
__link(www.w3.org)</p>
<p>Last updated: __today</p>
</body>
</html>
```

of quotes (the outermost) are stripped by m4, so the apostrophe in the following invokation will not yield an error:

```
__link2(www.gnu.org, ➥
'GNU, RMS's pet hobby-horse')
```

The author usually changes the default quote characters into "{" and "}" for readability and ease of typing. The command for changing the quote characters, with an appropriate comment attached (see the "Comments: documenting our macros" sidebar), is:

```
changequote({,}) dnl ➥
change the quote characters
```

changequote takes two arguments, the new opening and closing quotes respectively. It can be called at any time, and can even be called several times from the same file.

The effect is immediate, but only for this invokation of m4. The author strongly advocates that you stick to one set of quotes, as it quickly becomes rather hairy having to remember which quotes go where.

The quoting "characters", by the way, need not be single characters; you may use "{([whoopee->" as your opening quote if you wish. Neither is there any need for the closing quote to correspond logically to the opening quote. It is just a convention, and makes the macros easier to read 3 months hence.

```
changequote({,}) dnl change ➥
quote character
dnl create a link with the ➥
link name specified specifically
define(__link2, ➥
<a href="http://$1">$2</a>)
```

With a html.m4 containing the definitions shown above we can invoke our __link2 macro thus:

```
__link2(www.w3.org, {w3.org, ➥
a site well worth reading})
```

Enough basics, let's do some real work

With HTML, there's always a lot of stuff that needs to be set up at the top of each page. If you have more than, say, 2-3 pages that have a similar layout (but with optionally different <title> tags etc.) then you will probably want to create a macro for all this stuff. We will consider the sample HTML page shown in listing 1, and see how we can get a similar result using our newfound macro-skills.

After creating the necessary macros, listing 2 shows the content of the file first.mac. This is the mixture of HTML and macro calls that together with our macro definitions enables us to produce the resulting HTML in listing 1. We already know how to create the __title and the __link macros, the only new addition is the macro __today mentioned above. This macro uses m4's

Listing 3: first.m4

```
changequote({,}) ➥
dnl change quote character
dnl two macros for link-creation.
define(__link, ➥
<a href="http://$*">$*</a>)
define(__link2, ➥
<a href="http://$1">$2</a>)
dnl abstract away all the layout ➥
cruft at the beginning.
define(__title, {
<html>
 <head>
  <meta http-equiv=➥
"Content-Type" content=
   "text/html; charset=➥
iso-8859-1">
  <meta name="description" ➥
content="Sample HTML page">
  <meta name="keywords" ➥
content="gnu m4 html">
  <meta name="author" ➥
content="Stig Brautaset">
  <title>$1</title>
 </head>
 <body>
}) dnl the __title macro ➥
ends here
dnl use built-in 'esyscmd' to call the
standard Linux 'date'
dnl utility and have its output
replaced with the '__today'
dnl macro name. The date will be on
the form "Sun 16 Jun 2002"
define(__today, esyscmd(date '+%a %d
%b %Y'))
```

Listing 4: second.m4

```
__title2({Hello World, ➥
HTML version}, {
  <h1>Hello World</h1>
  <p>Look at this link: ➥
__link(www.w3.org)</p>
  <p>Last updated: __today</p>
})
```

Listing 5: second.m4

```
changequote({,}) ➥
dnl change quote character
dnl two macros for link-creation.
define(__link, ➥
<a href="http://$*">$*</a>)
define(__link2, ➥
<a href="http://$1">$2</a>)
dnl abstract away all the ➥
layout cruft at the beginning.
define(__title, {
<html>
 <head>
  <meta http-equiv=➥
"Content-Type" content=
   "text/html; charset=➥
iso-8859-1">
  <meta name="description" ➥
content="Sample HTML page">
  <meta name="keywords" ➥
content="gnu m4 html">
  <meta name="author" ➥
content="Stig Brautaset">
  <title>$1</title>
 </head>
 <body>
  $1
 </body>
</html>
}) dnl the __title macro ➥
ends here
```

dnl use built-in 'esyscmd' to call the standard Linux 'date' dnl utility and have its output replaced with the '__today' dnl macro name. The date will be on the form "Sun 16 Jun 2002" define(__today, esyscmd(date '+%a %d %b %Y'))

Listing 6: third.mac,25

```
define(__index) dnl allows ➥
conditional processing of the ➥
page
__title2({Hello World, HTML ➥
version}, {
  <h1>Hello World</h1>
  __menu
  <p>Look at this link: ➥
__link(www.w3.org)</p>
  <p>Last updated: __today</p>
})
```

capability to call standard Linux tools, and puts the output of the said command ("date" in this case) into the text. Listing 3 shows the full content of first.m4. This contains all the macro definitions required by first.mac which is found in listing 2.

More abstraction

Looking at the code in listing 2, you may not want to write the closing </body> and </html> tags either, and indeed you don't have to. m4 allows macros to be nested, thus we can use a macro within another macro. The result is shown in listing 4. Witness that the __title2 macro takes two arguments, the first being the page title and the second being the full page body. Be careful when you go to these lengths of abstraction though, as it is easy to miss out the closing "})" at the end of the file if you do extensive updates.

The change to listing 3 to facilitate this is shown in listing 5.

More advanced macros

Up till now, we have only looked at fairly simple search-and-replace macros. These work fine, but consider if we have a collection of pages, with a common menu. We could put the whole menu in a macro of the type we have used before, but then the pages would include a link to itself as well as all others, and this is not very elegant. A solution, of course, is to just cut-and-paste the menu in to the individual files and change each file to not make a link to itself. This, however, is very tedious. The solution? Use m4's built-in conditionals.

In each source file we define a macro that identifies that file. In index.mac we define, say, __index. The built-in conditional "ifdef" can then use these macro definitions to decide whether to take special actions on this file. The menu could then be something like this:

```
define(__menu, {
  <p>MENU<br>
   ifdef({__index}, index, ➥
  __rlink(index.html, index)) <br>
   ifdef({__pics}, pictures, ➥
  __rlink(pics.html, pictures))
  </p>
})
```

The two ifdef lines are new to us. They first check whether a certain macro name is defined (observe that the first argument of ifdef has to be quoted). If the macro name is undefined, the third argument will be input into the text, and the second argument will be ignored. The use of the __menu macro is shown in listing 6.

The __rlink macro is also new. Its name stands for relative link in that it does not prepend http:// to the link. It is shown in listing 7, which is the final macro listing file.

Summary

We have seen how to use m4 to create macros to help us maintain HTML pages. We went from very simple one-line substitution macros, like __link and __link2, to bigger but still very simple macros of the same type, like __title. From there, we went on to using m4's built-in ability to capture the output of system commands when we created the __today macro. Lastly we used m4's built-in conditionals to create a __menu macro that expands into a different menu on each page. ■

Listing 7: third.m4

```
changequote({,}) ➥
dnl change quote character
define(__link, ➥
<a href="http://$*">$*</a>)
define(__link2, ➥
<a href="http://$1">$2</a>)
define(__rlink, ➥
<a href="$1">$2</a>)
dnl abstract away all the ➥
layout cruft at the beginning.
define(__title2, {
<html>
 <head>
  <meta http-equiv=➥
"Content-Type" content=
   "text/html; charset=➥
iso-8859-1">
  <meta name="description" ➥
content="Sample HTML page">
  <meta name="keywords" ➥
content="gnu m4 html">
  <meta name="author" ➥
content="Stig Brautaset">
  <title>$1</title>
 </head>
 <body>
  $2
 </body>
</html>
})
dnl use built-in 'esyscmd' to call the
standard Linux 'date' dnl utility and have its
output replaced with the '__today' dnl
macro name. The date will be on the form
"Sun 16 Jun 2002"
define(__today, esyscmd➥
(date '+%a %d %b %Y'))
define(__menu, {
  <p>
   ifdef({__index}, index, ➥
__rlink(index.html, index)) <br>
   ifdef({__pics}, pictures, ➥
__rlink(pics.html, pictures))
  </p>
})
```

Using tidy to clean up the mess

If you've opened any of the HTML files you've created from the macro and content files, you've probably found that there's a lot of unnecessary white-space in them. This is OK, since excessive white-space is simply ignored by web browsers. If you're a pedantic zealot like the author, you'll want your source to be beautiful on its own as well.

Enter "tidy", an HTML validating, correcting and pretty-printing program. Simply invoke tidy on your HTML files thus: tidy -im first.html and the file will be audited and printed prettily. See the sidebar "Links and resources" for where to get tidy.

INFO

[1] GNU m4: more information about the m4 macro processor can be found at http://www.gnu.org/software/m4/m4.html

[2] HTML tidy: get your HTML cleaned up and validated http://www.w3.org/People/Raggett/tidy/

THE AUTHOR

Stig Brautaset, born in Norway, is the founder of the Linux Society at the University of Westminster. He is currently in his last year of a BSc Artificial Intelligence degree. His interests largely revolves around computer programming – from e-mail spam filters to games. Regularly spending much time on IRC he can be found there under the nick "Skuggan" or "Skugg".