Formatting a Web Page, Redux

by Jeffreys Copeland and Haemer

We've been writing for *RS/Magazine* since 1993. Some of our columns generate more mail than others. We enjoy getting mail but, even after three years of columns, we haven't figured out a way to guess which subjects will generate a lot of mail and which will not.

For example, three months ago, we wrote a column on formatting a Web page. Now, we'll admit that the title had the key phrase "Web Page." Even so, if someone began reading the column because it sounded trendy, they quickly found themselves facing a bizarre idea–using a text formatter as a macro processor, implemented in an arcane, antique language, `troff`, to make it possible to do things that HTML's designers think you shouldn't do in the first place. That is, control the layout of Web pages. Apparently, our readers thought this was so interesting that they decided to send us lots of mail.

Go figure.

Now, before we go on, a style caveat: Strictly speaking, we should be treating HTML as a structural markup language, not as a layout markup language. That is, in the best of all possible worlds, we'd treat HTML as we would SGML–as a way to identify elements in the document like chapter titles, and footnotes, not to say "display this text in 14-point bold type." Note: SGML is the Standard Generalized Markup Language, a standard for developing markup languages, of which HTML, Hypertext Markup Language, is one instance.

Jeffrey Copeland (`copeland@alumni.caltech.edu`) *is a member of the technical staff at QMS's languages group, in Boulder, CO. His recent adventures include internationalizing a large sales and manufacturing system and providing software services to the administrators of the 1993 and 1994 Hugo awards. His research interests include internationalization, typesetting, cats and children.*

Jeffrey S. Haemer (`jsh@canary.com`) *is an independent consultant based in Boulder, CO. He works, writes and speaks on the interrelated topics of open systems, standards, software portability and porting and internationalization. Dr. Haemer has been a featured speaker at Usenix, UniForum and Expo Kuwait.*

Unfortunately, to get our Web pages to display in a reasonable way, we must work around the inadequacies of HTML. As an interesting side point, it turns out that a common use of SGML is to develop documentation that is filtered into an on-line Web version in HTML and a print version in PostScript or something like that.

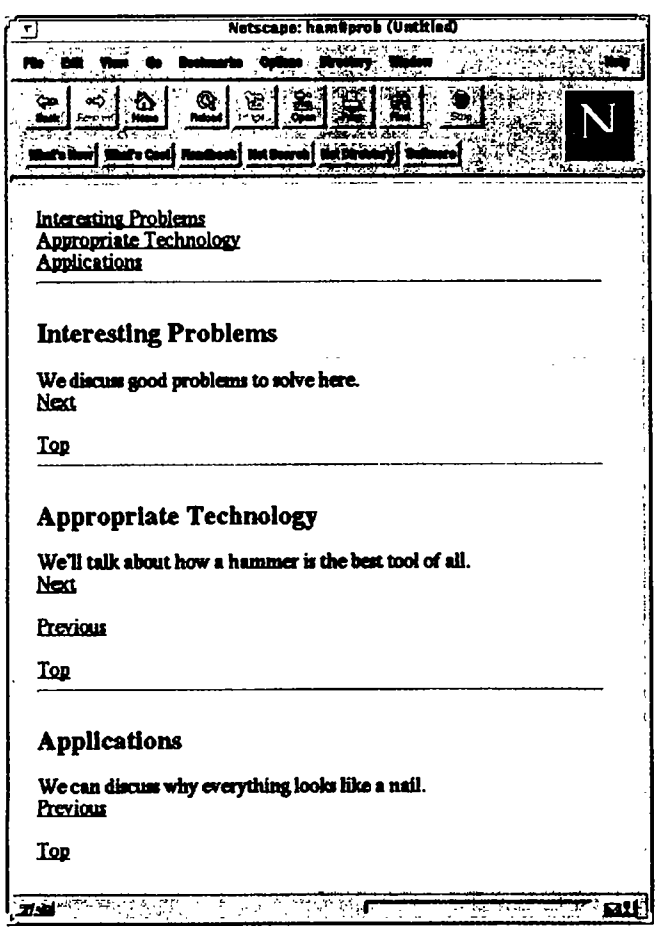
In this column, we'll tie up a loose end by doing a dramatic reading of the tool we wrote to generate links within documents. If readers write in about this one, we'll be mystified but overjoyed.

Writing the Document

Basically, we want to begin with a document that has sections, like this:

- Interesting Problems
  We will discuss good problems to solve here.
- Appropriate Technology
  We'll talk about how a hammer is the best tool of all.
- Applications
  We can discuss why everything looks like a nail.

and we will put a clickable table of contents at the front, and links at the end of each section, like this:



Naturally, we want these links to be encoded in HTML, like this:

```
<HTML> <BODY>

<A NAME="top">
<A HREF="#prob">Interesting Problems</A><BR>
<A HREF="#tech">Appropriate Technology</A><BR>
<A HREF="#appl">Applications</A><BR>

<HR>

<A NAME="prob"><H2>Interesting
Problems</H2></A>
We discuss good problems to solve here.<BR>
<A HREF="#tech">Next</A><P>
<A HREF="#top">Top</A>

<HR>

<A NAME="tech"><H2>Appropriate
Technology</H2></A>
We'll talk about how a hammer is the best
tool of all.<BR>
<A HREF="#appl">Next</A><P>
<A HREF="#prob">Previous</A><P>
<A HREF="#top">Top</A>

<HR>

<A NAME="appl"><H2>Applications</H2></A>
We can discuss why everything
looks like a nail.<BR>
<A HREF="#tech">Previous</A><P>
<A HREF="#top">Top</A>

</BODY> </HTML>
```

Of course, we don't want to handcraft this. After all, what are computers for? We feel strongly about this because we get paid for using computers to do things *and* because we spent an hour crafting the examples above in order to get them to print properly in the magazine.

Nor do we want any special magic in the source because we also want a `troff`-able document in which links would look silly.

Think for a second about how the formatter could solve this problem. The most straightforward way would be to recognize the beginning of each list, gather the entire contents of the list, build the table of contents, put it out and then put out the entire list, pausing at the end of each section to put in links. The bad news is that `troff` lets you do this with traps and diversions which might tempt you down this rat hole.

A less straightforward, but more fun, approach is to turn the problem on its head. Suppose, for example, we use a standard list notation, like this:

```
.BL
.LI
Here's the first list item.
.LI
Here's the second item.
.LI
Here's the third list item.
.LI
Here's the last list item.
.LE
```

If we read the list from back to front, then we know we have a list to process as soon as we see the .LE macro. We also know that as soon as we hit the corresponding .BL macro, we can write out an index that points to each of the .LI items that we've encountered, backing up through the file.

Moreover, while we're looking for the .BL macro, we can record the location of each .LI item we come to in order to construct the table of contents. We can even output an HTML command that links us to the "Next" item, as we've already seen.

This simple algorithm doesn't require keeping track of much, but–like our first algorithm–it's more work than we really want to do in troff. Fortunately, troff and HTML don't overlap so we can do this job with a simple troff front end, which we'll write in perl.

The Right Tool for the Job

Mr. Natural says, "Get the right tool for the job."
–R. Crumb

We begin with the simple step of reading in the entire file and flipping it upside down.

```
@in = <>;
@in = reverse(@in);
```

This leaves the file as an array of lines, with the last line of the file in $in[0] and the first line in $in[$#in]. We could try to edit this array in place, using a variety of tricks to inject lines of HTML into the right place in the array, but it's easier to create a new array, called @out, containing all the lines we produce. This means that the rest of our program looks like this:

```
foreach (@in) {
    # process each line,
    # adding HTML whenever appropriate
    push(@out, $_);
}
@out = reverse(@out);
print @out;
```

The foreach loop iterates through the input array putting each line, in turn, into the scratch variable $_, processing it and then pushing it, together with any extra HTML that we need, onto @out. At the end, we flip the file back over and spit it out in one piece.

To process the lines, we need to search for troff list macros. When we did this the first time, we used our own macros; then we switched to using .VL/.LI/.LE; and then we switched back again. Once we got tired of searching through the filter for places to fix, we put in this associative array:

```
%macro = (
    "begin_list" => "^\.\s*.LL",
    "list_item" => "^\.\s*.LI",
    "end_list" => "^\.\s*.LE",
);
```

This lets us handle list items like this:

```
if (/$macro{"begin_list"}/) {
    toc(keys %paragraph);
}
if (/$macro{"list_item"}/) {
    # register last list item
    toc("Next", "Previous", "Top");
    # for preceding list item
}
if (/$macro{"end_list"}/) {
        toc("Previous", "Top");
}
```

Here, toc() is a subroutine that pushes HTML for links onto @out. Intermediate list items have links for "Next," "Previous" and "Top." The last list item only has links for "Previous" and "Top" because there is no next.

The first list item is preceded by the table of contents for the entire list, which consists of keys for the hash %paragraph, an array that pairs the list item names with integers. "Hash" is perl for "associative array." Given this list:

```
.LL
.LI hello
some test code
.LI world
some more test code
.LE
```

Our filter ends up doing assignment statements like these:

```
$paragraph{'world'} = 0;
$paragraph{'hello'} = 1;
```

and the table of contents for the entire paragraph is generated with `toc(keys %paragraph);`, which translates to `toc('hello', 'world')`.

The rest of the code requires a little care to get the details right but is fairly straightforward. The resulting filter is shown in Listing 1.

Some Odds, Mostly Ends

L. Scott Emmons writes: "Tables are a great way to get things to line up right where you want them, and by turning off the border you can make it look like you can lay

Listing 1

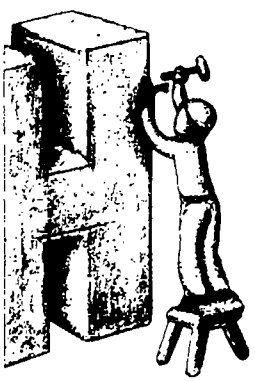
```
#!/bin/perl

sub P;
sub toc;
sub tick_toc;

%macro = (
      "begin_list" => "^\.\s*LL",
      "list_item" => "^\.\s*LI",
      "end_list" => "^\.\s*LE",
);

tick_toc;

sub br {
      push(@out, ".br\n");
}

sub SP {
      push(@out, ".SP @_\n");
}

sub P {
      push(@out, ".P\n");
}

sub tick_toc {
      @in = <>;
      @in = reverse(@in);

      for (@in) {
         if (/$macro{"begin_list"}/) {
            for ($i = 0; $i<10; $i++) {
                  # pop bogus TOC
                  pop(@out);
            }

            push(@out, ".dots green\n");
            toc(keys %paragraph);
            $id++;
            s/$macro{"begin_list"}//;
            chop;
            s/^"//; s/"$//;
            push(@out, "</A></A>\n");
            push(@out, ".H 2 $_\n")
                  if (/\S/);
            push(@out,
                  "<A name=$id><A name=TOC>\n");
                  # make "Previous" work
                  # for first item
                  next;
            }
            if (/$macro{"list_item"}/) {
                  $id++;
                  s/$macro{"list_item"}//;
                  chop;
                  s/^"//; s/"$//;
                  push(@out, "</A>\n");
                  push(@out, ".B \"$_\"\n");
                  push(@out, "<A name=$id>\n");
                  SP;
                  toc("Next", "Previous", "Top");
                  SP;
                  $paragraph{$_} = $id;
                  next;
            }
            if (/$macro{"end_list"}/) {
                  toc("Previous", "Top");
                  SP;
                  $id = 0;
                  %paragraph = ();
                  next;
            }
            push(@out, $_);
      }
      @out = reverse(@out);
      print @out;
}

sub toc {
      my(@entries) = reverse(@_);
      local(%paragraph) = %paragraph;
      $paragraph{"Next"} = $id;
      $paragraph{"Previous"} = $id+2;
      $paragraph{"Top"} = "TOC";

      br;
      foreach $entry (@entries) {
            push(@out,
"<A href=#$paragraph{$entry}>$entry</A>\n");
            push(@out,
                  ".right 8\n");
      }
      br;
}
```

Note: Source is available on the Web at *http://www. qms.com/*, so you don't have to type it in.

things out in space anywhere on the page you want."

Good point! David Siegel also recommends using tables (see "Formatting a Web Page," *RS/Magazine*, February 1996, Page 25), and we incorporated Siegel's suggestion in our macro package, but only to reset the margins to bring the number of words per line in line with approved typesetting practice.

Emmons also suggests using the tables as a substitute for single-pixel-gifs. We like the single-pixel-gif trick better for indenting because it corresponds conceptually to what we're doing–inserting blank space. Because HTML lets us set parameters for the image with `width=`, `height=`, `hspace=` and `vspace=` modifiers, we can give the indent to the macro as a parameter. For interline spacing, the choice between tables and single-pixel-gifs is more difficult. We used single-pixel-gifs only because we understand them better.



A more conventional approach to connecting `troff` and HTML is a stand-alone `troff`-to-HTML translator. T. Kurt Bond wrote us to say:

"I read your article in the February *RS/Magazine* about formatting Web pages and `troff` and thought that you might be interested in Unroff, a [nt]roff-to-HTML translator. Actually, it's a [nt]roff translator to anything, but the existing distribution just includes output to HTML. Unlike most [nt]roff translators, Unroff has a full `troff` parser and closely mimics `troff`'s processing. It is extensible (using a dialect of the Scheme programming language). The distribution includes customizations for the 'man' and 'ms' macros, but not for the 'me' or 'mm' macros.

"The HTML that it generates is not particularly flashy (it sticks strictly to HTML 2, for instance), but it's serviceable. More information about Unroff is available at `http://www.informatik.uni-bremen.de/~net/unroff/`."

We particularly like the phrase "Actually, it's a [nt]roff translator to anything," which really makes us want to think about other peculiar and warped things that we could do with it. Unfortunately,

```
(setq unroff-implementation-language-attribute
(dialect (dialect LISP)))
```

but that may not deter people from the MIT Artificial Intelligence labs, or even you. ▲