by Jeffreys Copeland and Haemer

SCOTT ROBERTS

One satisfactory answer … is to use a table to show the numbers. Tables usually outperform graphics in reporting on small data sets of 20 numbers or less.
– Edward Tufte, "The Visual Display of Quantitative Information"

Talent imitates, but genius steals.
– Thomas Stearns Eliot

Weighting Tables

More than a year ago, we got a note from our friend John McMullen, Canada's best technical writer, which we still haven't gotten around to answering properly. John had just written an *awk* program that converts input for the *troff* table preprocessor *tbl* into HTML. This is a useful tool when you want to generate Web pages from your *troff* input. One of the things John told us at the time was, "I'm kind of a brute-force guy, so when you rewrite it in Perl (which I'm sure you will) you can make it elegant. However, considering that programming is not my first language (innuendo is), I'm pleased to have it work."

We won't claim that this is elegant, and indeed, it doesn't handle some of the odd cases of the *tbl* input language, but it does generate reasonable HTML for the mainstream cases.

Since we're some of the few people left in the Western Hemisphere who still use *troff* as our principal text processing language, you may wonder why you'd be interested. Because we're exploring a general trick here of processing a whole chunk of text from one language into another, and you may be able to adapt it for something a little closer to home. Since you probably don't use *tbl*, let's quickly review what its input looks like.

To begin with, *tbl* only looks at input lines between the `TS` and `TE` macros; all other lines are passed through unmolested. Within that `TS/TE` pair, the table specification begins with a set of global options, ending with a semicolon. The three most common options center the table, enclose each entry in a box, and change the character to separate columns (the default is horizontal tab).

```
.TS
center allbox tab(#);
  ...
.TE
```

The table itself begins with rules for aligning each column, called the "format section," and then the data. A very simple example, showing each child's earnings from household chores this week, would be:

```
l n .
Allie#15.75
James#8.50
```

The possible column's alignments are left, right, center and numeric, all specified by their initials. In numeric columns, the data is aligned by its decimal points. You can provide more than one column specification if you wish, separating them with commas or newlines, and you can provide a rule across the table. The last specification is reused as necessary.

```
c c , l n .
child#earnings
_
Allie#15.75
James#7.50
```

All this together would produce:

```
child     earnings
-------------------
Allie        15.75
James         8.50
```

(The horizontal rule is redundant if you've specified `allbox`, so we took the latter out of the example above.)

There are additional wrinkles, but we'll discuss them as we handle them in our code.

The Basic Processor

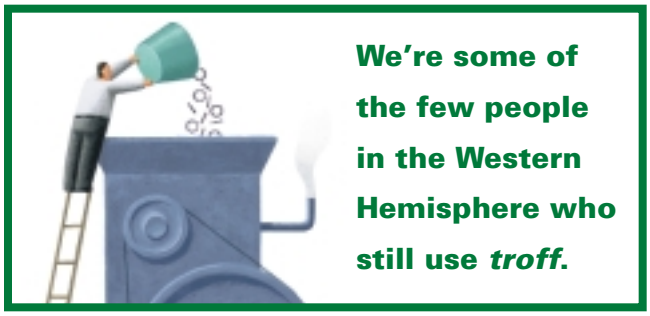
We begin our Perl script in the usual way, with a prolog and some global variables:

```
#! /usr/bin/perl -w
#  tbl-to-html converter: a strict filter
#  $ID: htbl,v 1.9 2000/09/02 00:38:41 ...

use strict;

my @table_options;
my @fmt_lines;
my $tabchar;
```

(Okay, they aren't really global variables. Because they are introduced with the keyword `my`, they have scope only in this module, but that's a nit.)



Where to from there? If you get stuck in the minutiae of column specifiers and spanned columns and rows and text blocks and partial rules, and lions and tigers and bears (oh, my), you quickly get bogged down in details and the problem looks intractable. However, on the surface, handling a *troff* file with embedded *tbl* is really quite simple:

First, read the file, processing only the lines between TS and TE; pass everything else unchanged. Next, within a TS/TE pair: process the global options ending with a semicolon, collect the format section, and process each line of the table based on the format section.

Pretty simple, right? Let's begin with the first item:

```
while ( grab_non_tbl()) {
 parse_tbl( grab_tbl() );
}
```

In other words, grab the stuff up to a TS macro, then pass the text up to the next TE macro to `parse_tbl()`.

It turns out that `grab_non_tbl()` and `grab_tbl()` are pretty simple given that we can set Perl's record separator.

```
sub grab_non_tbl {
 $/ = ".TS\n";
 $_ = <>;
 print  if( $_ );
 return ! eof STDIN;
}

sub grab_tbl {
 $/ = ".TE\n";
 $_ = <>;
}
```

In the first routine, we set the record separator, `$/`, to the table introducer, read a single record–that is, the text up to a TS line–and emit it. If we read to the end of the file, return false.

We get to the second routine when we know we've seen a table beginning, so we change the record separator to TE, and then read and return a single record containing the whole table. We're assuming that the input actually contains a TE macro. What if it doesn't? We leave a fix for graceful recovery as an exercise for the reader. You'll need to fix the code that puts a TE into the output, too.

What happens next? That's the job for `parse_tbl()`, which is where we start to really worry about the details.

Inside a Table

Starting the routine is (as always) simple.

```
sub parse_tbl {
 my $tbl = shift;
 my $options = "";
 my $htmlopt = "";
 my $fmt = "";
```

We get the whole table body as the argument, and we set up variables for the options (both in *tbl* and HTML forms) and for the format lines.

We need to begin the code by stripping the options off the table, and storing them in an array.

```
 # get the options for the table, if any
 $options = $1
  if( ($tbl =~ s/(^.*);\s*\n//) );
 @table_options = split /[ ,]+/, $options;
```

We immediately check that array to see if the table's global options have changed the tab character.

```
# and now parse the options
 if( (my @tabchar =
  grep(/tab/, @table_options)) ) {
   # only the last tab spec counts
  ($tabchar = $tabchar[-1]) =~ s/tab\((.)\)/$1/;
```

```
} else {
 $tabchar = "\t";
}
```

Notice that we've carefully allowed for the possibility of multiple `tab(x)` entries in the options; we only look at the last one. (Remember that Perl's `$foo[-1]` gives us the last entry in the array, and that we can have an array and a scalar with the same name.)

Continuing the same thought, we check for the `allbox` and `center` options, converting these to the appropriate HTML clauses, and then outputting our first HTML directives, including the opening `<TABLE>` tag.

```
$htmlopt .= " BORDER=\"2\""
  if( grep(/allbox/, @table_options) );
 print "<CENTER>"
  if( grep(/center/, @table_options) );

  # table setup
 print "<TABLE$htmlopt>\n";
```

One of the interesting features of *tbl* is that you can provide multiple segments in the table, each with a new set of format specifications, but separated by `.T&` lines. In other words, our example above could have been rendered as the following:

```
.TS
center tab(#);
c c .
child#earnings
_
.T&
l n .
Allie#15.75
James#8.50
.TE
```

This means that within our table processing code, we want to only grab the current segment for processing. That is, we peel off the part of the table up to the next `TE` or `T&` macro.

```
# grab each segment separated by .T&
 while( $tbl =~ s/^(.+?)\n\.T[\&E]\n//s ) {
  tbl_segment($1);
 }
```

Our first regular expression to perform this task was `s/^(.+)T[E]//s`. The `s` qualifier is correct: it spans lines, allowing the dot in a regular expression to match newlines. The fatal error was the `(.+)` clause, which is, in regular expression parlance, "greedy." This means that it matches everything until the part of the regular expression *following* it is matched, including intervening instances of the closing clause. To give a concrete example, in our last rendition of the table a few paragraphs back, `(.+)` would happily have matched the `T&` in the middle of the table, leaving the `\n\.T[\&E]\n` to be matched only against the `TE` that closes the table. To prevent this, we use the question mark modifier, saying `(.+?)`. This only matches until the following part of the expression is matched the first time.

(OO guru Dave Taenzer claims that the difference between developing in C++ and Perl is that C++ programmers say, "I wonder why that didn't work," while Perl programmers say, "I wonder why that worked.")

What do we do with the part of the table we've stripped off this way? We send it on, without the trailing macro, to our routine `table_segment()`, which we'll discuss shortly.

After as many calls to `table_segment()` as we need, we finish up our table by adding the closing HTML tags to our output stream:

```
# table cleanup
 print "</TABLE>\n";
 print "</CENTER>"
  if( grep(/center/, @table_options) );
 print ".TE\n";
}
```

Note that we've output both the opening `TS` and the closing `TE`. Why? We're presumably going to turn this file into HTML, so why would we want to maintain these macros? Because the macro package we use to do the conversion may want to do some special handling for the `TS/TE` pair. If not, the downstream programs can strip them out without further action.

Processing the table segment in `tbl_segment()` is simply a matter of splitting the formatting section from the lines of the table, and then regurgitating the lines of the table in HTML form.

```
sub tbl_segment {
 my $seg = shift;

 # strip off the column formats
 $seg =~ s/^([^.]+)\.\s*\n//s;
```

The column formats end on a line with a period as its last text character. We strip them off from our scalar variable `$seg` for the moment.

```
 # associate vert rules with columns
 (my $fmt = $1) =~ s/\s+\|/|/g;
 my @fmt = split / *[\n,] */, $fmt;
```

Once we've got them, we simultaneously assign them into a variable `$fmt` and convert lines like

```
c | l | r
```

into

```
c| l| r
```

What does this mean? *tbl* allows us to use `|` to represent a vertical line between columns. We want to associate it with the column it follows. This means that the count of elements in the array `@fmt` is the same as the number of columns we expect to produce. Of course, HTML's table mechanism does not actually support arbitrary vertical rules, so we'll ignore the `|` specifiers as we proceed.

(This omission of rules doesn't bother us: On matter of style, we defer to the opinion of George Bernard Shaw, who once complained, "The only one thing that never looks right is a rule. There is not in existence a page with a rule on it that can't be instantly and obviously improved by taking the rule out.")

Now we've got the format in `@fmt` and the remainder of the table segment in `$seg`. Each line in `$seg` represents a line in the table, and we have one format in `@fmt` per line of the table–repeating the last one if necessary–so the processing is really very simple.

```
# table body
 foreach (split /\n/, $seg) {
  chomp;
  tbl_line($fmt[0], $_);
  shift @fmt  if( $#fmt > 0 );
 }
}
```

Well, maybe it's not so simple. What happens in `tbl_line()`? Pushing our problems away until later has become a nasty habit in this project.

The Hard Part

This brings us down to what, at least at first glance, seems complicated. We've got a format specifier, such as `r n` and a line of text such as `Allie#15.50`, out of which we want to generate a row of an HTML table, such as,

```
<TR><TD ALIGN="right"> Allie 
 <TD ALIGN="char" CHAR="."> 15.50 
 </TR>
```

Why the HTML non-breaking space characters? Because we need a little extra side bearing on the table entries, otherwise they mush together on the screen.

There's another variation we need to take care of. It's possible to have data span columns. We use this, for example, if we want to have a heading bridge several columns. We'll have to deal with this if we see a column format of `s`. Similarly, we may see a data item containing `\^` , which is a column that's spanned vertically, that is, a data item that takes up more than one row. We'll handle these as we come to them in the input data.

From here, we'll take the problem in bite-sized pieces, and make it simple again.

We begin `tbl_line()` by taking the format `$fmt` and the row `$line` off the stack. We also reset the array that will contain the formatting information.

```
sub tbl_line {
 my $fmt = shift;
 my $line = shift;
 my @fmt = { };
```

We make some changes to the format specifier before we split it into its array. First, we completely remove the vertical rules we discussed earlier. (Why not elide them sooner? Because we may later figure out how to handle them with the limited facilities in HTML, or we may decide to process them in some other way. Anticipating either possibility, it's better to remove them at the last moment.) Next, we collapse the spanned columns so they are attached to their predecessors. After that, we have a variable with one "word" for each column, and we can split it into the `@fmt` array.

```
# get the format specifier...
 $fmt =~ s/\|//g;  # elide vertical rules
 $fmt =~ s/ s/s/ig; # collapse spanned cols
 @fmt = split / +/, $fmt;
```

From there, we need to convert the format keywords into HTML specifiers by looping over each item in `@fmt`.

```
foreach (@fmt) {
  my $l;
  s/[^rlcns]+//i;
  s/r/ALIGN="right"/i  ||
   s/l/ALIGN="left"/i  ||
   s/c/ALIGN="center"/i  ||
   s/n/ALIGN="char" CHAR="."/i;
```

```
  s/(s+)/length($1)+1/e;
  s/\d+$/ COLSPAN=\"$&"/;
 }
```

There are several things to note in this code, which is really the heart of the conversion. First, we remove the format specifiers we don't recognize. This means that when we're done, there is no leftover cruft in the new HTML specifier. Next, the basic substitution lines are joined by logical ORs. This means that our substitution for `n` does not act on `ALIGN=` generating an HTML tag that begins `ALIGALIGN=`... The last thing to note is the way we count up the spanned column specifiers to create the `COLSPAN` keyword. That needs to be handled in two stages because the `COLSPAN=` confuses Perl when we are using it in conjunction with the `e` modifier to expand the right-hand side of the substitution.

What next? We check for the very special case of a horizontal rule in the middle of the table. If the line contains only an underscore, we make a table row containing a rule that spans all the columns.

```
 # special case of a line just containing _
 if( $line eq "_" ) {
  print "<TR><TD COLSPAN=\"",
   $#fmt+1, "\"><HR></TR>\n";
  return;
 }
```

We need to split up the line from the scalar `$line` into an array `@line`, based on the tab character we discovered among the global options.

```
# get the items
 my @line = split /$tabchar/, $line;
```

After that, we're home free: we generate the actual row in the HTML version of the table.

```
  # now generate the table row:
 print "<TR>";
 while( $#fmt >= 0  &&  $#line >= 0 ) {
  my $colfmt = shift @fmt;
  my $colitem = shift @line;
   # ignore elements from spanned rows
  next  if( $colitem eq "\\\^" );
   # a horizontal rule item
  if( $colitem eq "_" ) {
   print "<TD><HR>\n    ";
   next;
  }
  print "<TD $colfmt>",
   " $colitem \n    ";
 }
 print "</TR>\n";
}
```

The row is bracketed by HTML `<TR>` and `</TR>` tags, and each column is introduced by a `<TD>` tag. If the column data represents a vertically spanned row, that is, the data item is `\^`, we ignore it. For our purposes here, rather than being vertically centered, those items will be aligned at the top. (We have a sketchy solution to this problem, which is too large to show in the margins of this column: it takes as much code as the rest of this routine combined. We leave a better solution as an exercise for the reader.) We also do some special handling if there is a horizontal rule in the column. Otherwise, we just use the column format we generated in our ganged substitution above.

While it would be an efficiency improvement to convert the format section to HTML immediately after we read it, we like having the format conversion happening right before the generation of the HTML row. This means that we have format specifiers that are being converted repeatedly, but that's a small price to pay. What's the overall result? The table we showed you at the beginning of the column ends up in HTML as:

```
<CENTER><TABLE>
<TR><TD ALIGN="center"> child 
 <TD ALIGN="center"> earnings 
 </TR>
<TR><TD COLSPAN="2"><HR></TR>
<TR><TD ALIGN="left"> Allie 
 <TD ALIGN="char" CHAR="."> 15.75 
 </TR>
<TR><TD ALIGN="left"> James 
 <TD ALIGN="char" CHAR="."> 8.50 
 </TR>
</TABLE></CENTER>
```

That's all there is to it. One hundred-thirty-odd lines of code to match the functionality of a *troff* preprocessor, available in the lobby on your way out, or from the usual Web page. As we tuck our HTML code into our saddle bags, we'll tell you that next month we'll be talking about reading again. Until then, we wish you happy trails.

Jeffrey Copeland *(copeland@alumni.caltech.edu) is currently living in the Pacific Northwest, where he spends his time writing UNIX software in a large development organization and fighting damp rot.*

Jeffrey S. Haemer *(jsh@usenix.org) works at Minolta-QMS Inc. in Boulder, CO, building laser printer firmware. Before he worked for QMS, he operated his own consulting firm and did a lot of other things, like everyone else in the software industry.*

Note: The software from this and past Work columns is available at `http://alumni.caltech.edu/~copeland/work` *or alternately at* `ftp://ftp.cpg.com/pub/Work`.