Converting LaTeX `.sty` Style Files to LaTeX2HTML `.perl` Style Files

Nicola Talbot

Friday 10th June, 2005

Contents

1 Introduction

LaTeX2HTML is a Perl script that translates LaTeX files into HTML. If the LaTeX source code loads packages via `\usepackage`, LaTeX2HTML will search for a file of the same name, but with a `.perl` extension, instead of a `.sty` extension. This means that if you write a LaTeX package, and you want LaTeX2HTML to understand it, you need to convert your LaTeX code into a Perl script. This requires an understanding of Perl, and an understanding of the inner workings of LaTeX2HTML (and, of course, an understanding of LaTeX and HTML).

If you don't know any Perl, you'll need to learn before proceeding further. If you use Unix or Linux, try `man perl`, better still, try reading "Programming Perl"[2]. If you don't know any HTML, try "HTML: The Definitive Guide"[1].

Note: there are many different programming styles, particularly for Perl. Some people like to optimize the code at the expense of legibility, whilst others may prefer to use slower code that's easier to understand. Since this is a tutorial

I will tend to opt for legibility, you can then translate this into obfuscated Perl if that is your desire.

2 The LaTeX2HTML Process

LaTeX2HTML parses the document in a different manner to LaTeX, and this is something you need to be wary about. The `&translate` subroutine splits up the input into corresponding segments (if `-split` is given a non zero value), and these segments are then translated. Therefore, if you want to define a command that starts a new section type (e.g. `\makeglossary`), LaTeX2HTML won't translate the command, until after the document has been split, at which point it's too late to split it into a new segment.

The translator replaces all braces with marks containing unique identifiers, to make it easier to match opening braces with their corresponding closing braces. For example, consider the following code:

```
Some \textbf{bold \textit{italic}} text.
```

This will be translated into:

```
Some \textbf<#6#>bold <I>italic</I><#6#> text.
```

The number inside the `<#`*n*`#>` sequence, `6` in the above example, is the unique identifier for that set of braces. Note also that commands are translated inside out, so `\textit` has been translated before `\textbf`. The maximum number of identifiers is given by `$global{'max_id'}`, so if you want to add a new set of braces, you will need to do something like:

```
$id = ++$global{'max_id'};
$_ = "${OP}${id}${CP}Some text${OP}${id}${CP}";
```

where `$OP` is a predefined variable with the value `<#` and `$CP` is a predefined variable with the value `#>`.

3 Writing Subroutines to Translate LaTeX Commands

For every command `\`*cmdname* that is encountered, LaTeX2HTML will use the subroutine `&do_cmd_`*cmdname*, if it exists[1]. The remainder of the entire segment will be passed to this subroutine as a single string, and the subroutine must return the remainder of this string when it's done.

3.1 Commands without an argument

The easiest way to explain something is via example, so let's try creating a very simple package. Suppose you want a package called, say, `mydate.sty` which redefines the command `\today` so that the date is formatted in the form *year-month-day*, e.g. 2005-6-30. The LaTeX code will look something like:

[1] and of course, if the command hasn't been specified as one to pass to the image file, such as the maths commands

```
\NeedsTeXFormat{LaTeX2e}
\ProvidesPackage{mydate}

\renewcommand{\today}{\number\year-\number\month-\number\day}

\endinput
```

Since the package is called `mydate.sty`, the Perl file needs to be called `mydate.perl`, and should look something like:

```
#!/usr/bin/perl

sub do_cmd_today{
   local($_) = @_;
   local($sec,$min,$hr,$day,$month,$year) = localtime(time);
   $year += 1900;
   $month++;

   "$year-$month-$day" . $_;
}

1;
```

Things to note: the last line in the file must always be `1;` and the first line may vary depending on your system. Since the command `\today` does not take any arguments, the subroutine `do_cmd_today` does not need to read any information in from its input string, however, it must append this string to the text generated by the command, and return it. In the above example, this is done by string concatenation:

```
"$year-$month-$day" . $_;
```

but can also be done using the `join` function:

```
join('', "$year-$month-$day", $_);
```

or even:

```
join('-', $year,$month,$day) . $_;
```

If you do not do this, you will lose the rest of the text in that segment.

3.2 Commands with arguments

The previous example did not have any arguments, so let's try one with an argument. The `mydate` package described above is now going to be modified so that it defines the command `\monthname` which takes one argument—the number of the current month (from 1 to 12.) The command `\today` will be modified so that it uses the month name instead of a number. The LaTeX code in `mydate.sty` now looks like:

```
\NeedsTeXFormat{LaTeX2e}
\ProvidesPackage{mydate}
```

```
\newcommand{\monthname}[1]{%
\ifcase#1
\or January%
\or February%
\or March%
\or April%
\or May%
\or June%
\or July%
\or August%
\or September%
\or October%
\or November%
\or December%
\fi}

\renewcommand{\today}{%
\number\year-\monthname{\month}-\number\day}

\endinput
```

LaTeX2HTML defines the array `@Month` which contains the month names in the currently defined language, so the `&do_cmd_monthname` subroutine can use this, but first it needs to determine the month number which is passed as the argument. Recall from section 2 that braces are replaced by markups with unique identifiers in the form `$OP`n`$CP`. This means that you need to search for these identifiers at the start of the string passed to the `&do_cmd_monthname` subroutine. Rather than having to remember the form of these identifiers, you can use the variable `$next_pair_pr_rx` which provides the correct regular expression, where `$1` will contain the unique identifier, and `$2` will contain the text found inside the set of braces corresponding to that identifier.

For example, suppose your LaTeX code looked something like:

```
\documentclass[a4paper]{article}

\usepackage{mydate}

\begin{document}
\monthname{1} is a very chilly month
in Britain.
\end{document}
```

then the argument passed to `&do_cmd_monthname` will be the string

```
<#4#>1<#4#> is a very chilly month
in Britain.
```

The Perl code

```
s/$next_pair_pr_rx/$month=$2;''/eo;
```

will set `$month` equal to `$2`, which in this case is simply `1`. The value of the identifier in this instance is not necessary, but if you wanted to know it for some

reason, you can add `$id=$1`. Note that if a match is found, the empty string `''` will be substituted, which means the substring `<#4#>1<#4#>` will be removed from `$_`.

It is possible that the user may have omitted the braces around the argument to the command (e.g. `\monthname 1`), in which case you need to get the first character, and warn about the missing braces. This can be done using the `&missing_braces` subroutine:

```
unless (s/$next_pair_pr_rx/$month=$2;''/eo)
{
   $month = &missing_braces;
}
```

or more succinctly:

```
$month = &missing_braces unless s/$next_pair_pr_rx/$month=$2;''/eo;
```

So the whole subroutine should look like:

```
sub do_cmd_monthname{
   local($_) = @_;
   local($month);

   $month = &missing_braces unless
               s/$next_pair_pr_rx/$month=$2;''/eo;

   $Month[$month] . $_;
}
```

The subroutine `&do_cmd_today` can now be modified so that it uses the `\monthname` command:

```
sub do_cmd_today{
   local($_) = @_;
   local($sec,$min,$hr,$day,$month,$year) = localtime(time);
   $year += 1900;
   $month++;

   local($id) = ++$global{'max_id'};

   join('-',
        $year,
        "\\monthname${OP}${id}${CP}$month${OP}${id}${CP}",
        $day) . $_;
}
```

Of course, it would be even easier to use `$Month[$month]` instead of

```
\\monthname${OP}${id}${CP}$month${OP}${id}${CP}
```

but this way illustrates the use of `$OP`, `$CP` and `$global{'max_id'}`.

The basic principle can be extended to commands with more than one argument. Each argument is dealt with in the same way. For example, suppose you have a command called, say `\fmtdate` that formats a specific date in a certain way, then this command would need to take three arguments representing the day, month and year. The LaTeX code might look something like:

```
\newcommand{\fmtdate}[3]{#3-#2-#1}
```

The Perl subroutine `&do_cmd_fmtdate` would then look something like:

```
sub do_cmd_fmtdate{
   local($_) = @_;
   local($day,$month,$year);

   $day = &missing_braces unless
             s/$next_pair_pr_rx/$day=$2;''/eo;

   $month = &missing_braces unless
             s/$next_pair_pr_rx/$month=$2;''/eo;

   $year = &missing_braces unless
             s/$next_pair_pr_rx/$year=$2;''/eo;

   join('-', $year, $month, $day) . $_;
}
```

3.3 Optional arguments

Suppose you want the `\monthname` command defined in the previous section to have an optional argument instead of a mandatory argument. If the argument is omitted, the current month will be used. The LaTeX code will now look like:

```
\newcommand{\monthname}[1][\month]{%
\ifcase#1
\or January%
\or February%
\or March%
\or April%
\or May%
\or June%
\or July%
\or August%
\or September%
\or October%
\or November%
\or December%
\fi}
```

The Perl subroutine `&do_cmd_monthname` will now need to use the subroutine `&get_next_optional_argument`. This returns two parameters: the contents of the optional argument and the pattern. For example, if `$_` contains the string

```
[1] is a very chilly month
in Britain.
```

then

```
($month,$pat) = &get_next_optional_argument;
```

will result in `$month="1"` and `$pat="[1]"`, and `$_` will now contain the string:

```
is a very chilly month
in Britain.
```

The subroutine `&do_cmd_monthname` can now be modified as follows:

```
sub do_cmd_monthname{
   local($_) = @_;
   local($sec,$min,$hr,$day,$month,$pat);

   ($month,$pat) = &get_next_optional_argument;

   if ($month eq '')
   {
      ($sec,$min,$hr,$day,$month) = localtime(time);
      $month++;
   }

   $Month[$month] . $_;
}
```

Note: be careful not to do:

```
$month = &get_next_optional_argument;
```

4 Writing Subroutines to Translate LaTeX Environments

For every environment *env-name* that is encountered, LaTeX2HTML will use the subroutine `&do_env_`*env-name*, if it exists. The body of the environment will be passed to this subroutine as a single string.

Consider a trivial example: suppose you want to define an environment called, say, `bfit` that typesets its contents in a bold italic font, the LaTeX code might look something like:

```
\newenvironment{bfit}
{\begin{bfseries}\itshape}
{\end{bfseries}}
```

The corresponding Perl file will need to define a subroutine called `&do_env_bfit`, that will look something like:

```
sub do_env_bfit{
   local($_) = @_;

   "<B><I>" . $_ . "</I></B>";
}
```

This puts the contents of the environment into the HTML bold and italic markups.

4.1 Environments with an argument

Obtaining the argument information for an environment is much the same as that for a command, as described in section 3.2, except that the argument is delimited by `$O`*n*`$C` instead of `$OP`*n*`$CP`. Suppose you have an environment called, say, `exercise` which will typeset an exercise for the reader. Each exercise has an argument—the exercise title—which will be typeset in bold at the start of the environment. The LaTeX code might look something like:

```
\newenvironment{exercise}[1]%
{\begin{quote}\textbf{Exercise : #1}\par}
{\end{quote}}
```

The Perl code will then look something like:

```
sub do_env_exercise{
   local($_) = @_;
   local($title);

   $title = &missing_braces
             unless (s/$next_pair_rx/$title=$2,''/eo);

   "<BLOCKQUOTE><B>Exercise : $title</B><P>"
   . $_ . "</BLOCKQUOTE>";
}
```

Note the use of `$next_pair_rx` instead of `$next_pair_pr_rx`.

Optional arguments are dealt with in the same way as described in section 3.3 For example, suppose the `exercise` environment above should have an optional argument instead. If the argument is present it is used as a title, e.g. **Exercise : The Title**, but if it is not present, the title is simply **Exercise**. The LaTeX code will look something like[2]:

```
\newenvironment{exercise}[1][]%
{\begin{quote}
\textbf{Exercise\ifthenelse{\equal{#1}{}}{}{ : #1}}\par}
{\end{quote}}
```

and the Perl code should look something like:

```
sub do_env_exercise{
   local($_) = @_;
   local($title,$pat);

   ($title, $pat) = &get_next_optional_argument;

   if ($title ne '')
   {
      $title = " : $title";
   }

   "<BLOCKQUOTE><B>Exercise$title</B><P>" . $_ . "</BLOCKQUOTE>";
}
```

[2] Remember to use the `ifthen` package.

5 Counters

LaTeX counters are stored in the hash table `%global`. For example, consider the `exercise` environment defined in the previous section. Suppose each exercise should have a corresponding counter, also called `exercise`. The LaTeX code will now look something like:

```
\newcounter{exercise}

\newenvironment{exercise}[1][]%
{\begin{quote}
\refstepcounter{exercise}%
\textbf{Exercise \theexercise\ifthenelse{\equal{#1}{}}{}{ : #1}}\par}
{\end{quote}}
```

The Perl code will now look something like:

```
$global{'exercise'} = 0;

sub do_env_exercise{
   local($_) = @_;
   local($title,$pat);

   ($title, $pat) = &get_next_optional_argument;

   if ($title ne '')
   {
      $title = " : $title";
   }

   $global{'exercise'}++;

   "<BLOCKQUOTE><B>Exercise ".
   $global{'exercise'} .
   "$title</B><P>" . $_ . "</BLOCKQUOTE>";
}
```

You can also obtain the value of a counter using the subroutine `&get_counter_value`:

```
$val = &get_counter_value($ctr);
```

where `$ctr` contains the name of the counter, and `$val` is the value of that counter.

If you have a LaTeX command that has the name of a counter passed as an argument, you can read it in using `&read_counter_value`. This reads in a string, extracts the name of the counter at the start of the string and returns the counter name, its value, the unique identifier delimiting it and the remainder of the input string. For example, suppose you want a LaTeX command called, say, `\bfroman` which takes the name of a counter as the argument, and typesets it in bold roman numerals:

```
\newcommand{\bfroman}[1]{\textbf{\roman{#1}}}
```

then the Perl subroutine would look something like:

```
sub do_cmd_bfroman{
   local($ctr,$val,$id,$_) = &read_counter_value($_[0]);

   if ($val < 0)
   {
      $val = join('', "-", &froman(-$val));
   }
   else
   {
      $val = &froman($val);
   }

   "<B>$val</B>" . $_;
}
```

6 Defining Package Options

Given a package called *name*, each package option *opt* is dealt with by the subroutine `&do_`*name*`_`*opt*. Returning to the `mydate` package example, described in section 3.1, suppose this package now has two options: `dash` (e.g. 2005-6-30) and `dot` (e.g. 2005.6.30). The LaTeX code will now look something like:

```
\NeedsTeXFormat{LaTeX2e}
\ProvidesPackage{mydate}
\newcommand{\datesep}{-}
\DeclareOption{dash}{\renewcommand{\datesep}{-}}
\DeclareOption{dot}{\renewcommand{\datesep}{.}}
\ProcessOptions

\renewcommand{\today}{\number\year\datesep
\number\month\datesep\number\day}

\endinput
```

The Perl code will now need the subroutines `&do_mydate_dash` and `&do_mydate_dot` in order to implement the package options. The file `mydate.perl` will now look something like:

```
#!/usr/bin/perl

$datesep = '-';

sub do_mydate_dash{
   $datesep = '-';
}

sub do_mydate_dot{
   $datesep = '.';
}
```

```
sub do_cmd_datesep{
   local($_) = @_;
   $datesep . $_;
}

sub do_cmd_today{
   local($_) = @_;
   local($sec,$min,$hr,$day,$month,$year) = localtime(time);
   $year += 1900;
   $month++;

   "$year\\datesep $month\\datesep $day" . $_;
}

1;
```

Suppose you now want to use the `keyval` package to specify your package options. For example, you might want to do:

```
\usepackage[style=dash]{mydate}
```

or

```
\usepackage[style=dot]{mydate}
```

For each package option in the form *key=value*, you need to supply the subroutine `&do_`*packagename_key_value*. So for the above example, you will need:

```
sub do_mydate_style_dash{
   $datesep = '-';
}

sub do_mydate_style_dot{
   $datesep = '.';
}
```

7 Reading Other Files

There are several subroutines for reading other files:

`&slurp_input` takes one argument, the name of the file to be input. The contents of the file are put in `$_` without any conversion applied.

`&slurp_input_and_partition_and_pre_process` takes one argument, the name of the file to be input. This subroutine reads the entire input file and preprocesses it. It is then returned as a single string.

`&process_ext_file` This reads in a LaTeX generated file with the same root name as the main file being processed, but a different extension (e.g. `aux` or `bbl`). It takes one argument, the file extension, and returns success or failure, and `$_` is set.

Be careful when specifying the filename with the first two, as LaTeX2HTML may not be in the directory you started in, so it is best to give the full pathname using $texfilepath and $dd.

Suppose you have a LaTeX command called, say, `\inputcsv{`*filename*`}`, which reads in comma-separated data from the file given by *filename*, and sets it in a table. For example, if the contents of the file `sample.csv` looked like:

```
1,5,10,12,3
4,7,9,5,2
7,3,2,1,0
```

then `\inputcsv{sample.csv}` will produce the following table:

1	5	10	12	3
4	7	9	5	2
7	3	2	1	0

then the Perl code, might look something like:

```
sub do_cmd_inputcsv{
   local($after) = @_;
   local($file,$table);

   $file = &missing_braces
     unless ($after=~s/$next_pair_pr_rx/$file=$2;''/eo);

   $file = "$texfilepath$dd$file";

   &slurp_input($file);

   s/,/<\/TD><TD ALIGN=RIGHT>/sg;
   s/\n/<\/TD><\/TR>\n<TR><TD ALIGN=RIGHT>/sg;
   s/^/<TR><TD ALIGN=RIGHT>/g;

   $table = "</TD></TR><TABLE ALIGN=CENTER>";
   $table .= $_;
   $table .= "</TABLE>";

   $table . $after;
}
```

Since `sample.csv` does not contain any LaTeX code—and so no translation is required—`&slurp_input` is used. Note that the full pathname is given by prepending `$texfilepath$dd` to the given file name. Note also the use of `$after` instead of `$_`. This was done because `&slurp_input` modifies `$_`. If the file is likely to contain LaTeX commands, e.g.:

```
A,B,C,D,E\&F
1,5,10,12,3
4,7,9,5,2
7,3,2,1,0
```

then you should use `&slurp_input_and_partition_and_pre_process` instead.

Useful LaTeX2HTML Subroutines and Variables

`$dd`

The directory divider. This is platform specific, and is determined at the start of the LaTeX2HTML run. Use this variable to ensure that your code is platform independent. 12

`&get_next_optional_argument`

Extracts optional argument at the start of `$_` and returns (`$argument`, `$pattern`). 6

`$global{'max_id'}`

This is the maximum number of unique identifiers. 2, 5

`&missing_braces`

Generate a warning message and extract first character from `$_`. 5

`@Month`

The month names in the currect language are stored in this array. Note that `$Month[0]` is empty, so subscripts effectively start from 1. 4, 5

`$next_pair_pr_rx`

Regular expression used to extract the group at the start of `$_` delimited by `$OP`*n*`$CP`. The contents of the group is given by `$2`, the unique identifier belonging to that group is given by `$1`. 4, 8

`$next_pair_rx`

Regular expression used to extract the group at the start of `$_` delimited by `$O`*n*`$C`. The contents of the group is given by `$2`, the unique identifier belonging to that group is given by `$1`. 8

`&slurp_input`

This takes one argument, the name of the file to be input. The contents of the file are placed in `$_` without any conversion applied. 11, 12

`&slurp_input_and_partition_and_pre_process`

This takes one argument, the name of the file to be read. The contents of the file are translated and placed in `$_`. 11, 12

`$texfilepath`

The directory containing the LaTeX file to be translated. 12

References

[1] Chuck Musciano and Bill Kennedy. *HTML: The definitive guide*. O'Reilly & Associates, Inc, 1996.

[2] Larry Wall, Tom Christiansen, and Randal L. Schwartz. *Programming Perl*. O'Reilly, 2nd edition, 1996.