From .XLS to .HTML – A Quick Demonstration

by Andrew Kohlsmith <akohlsmith-kwpm@benshaw.com>
20030115

=Head1 History

Historically, Benshaw has used Microsoft Excel to generate quotation and acknowledgment forms for its business. This was very easy to set up as it was a grid-based system which allowed you to alter the text attributes (size, font, bold, italics, etc.), draw lines, add graphics and perform calculations on cells.

Each quotation exists as a single .xls file and is based on a number (80 or so) product templates which gave the verbiage that was simply cut and pasted into the quotation. Options to the products were stored in other templates and copied and pasted into the quotation.

Once the quotation was accepted and a purchase order given, the quotation .xls file would be edited to change the "Quotation" line to "Acknowledgment", assigned an order number and saved into a different directory and the job would be added to the production queue.

Of course this is sub-optimal. Not only are you constantly cutting and pasting and storing a lot of redundant data over and over again (lines, text from the cut and paste, internal Excel data) but you are also more or less locking yourself into a particular version of Excel. Microsoft, to their credit, has kept the .xls file format more or less backward-compatible and we have not had many issues at all with these simple .xls files. However every year we have subtle changes to our product templates and option templates such as new or removed features or additional functionality. This means that for every horsepower range and every voltage range there is a lot of hand-editing that has to be done to the template files so that when doing quotes you can just cut and paste the data.

I set out to create a web-based application which let them use dropdowns to select the product, voltage, horsepower, enclosure type, etc. and then add any options. I was going to store all of the base text in a database and create tables to replace the matrices used to select what frame size is what horsepower at which voltage.

use Spreadsheet::ParseExcel;

Perl has an excellent module for reading and parsing Microsoft Excel spreadsheets, written by Kawai Takanori. Without this module I would not have gotten this accomplished so easily, or even at all. His module relies on `OLE::Storage::Lite` to actually access the OLE bits of the file (meaning pretty much everything).

Opening up the .xls file and iterating through all the columns in every row in every sheet of the workbook is quite simple:

```
use strict;
use Spreadsheet::ParseExcel;
my $oBook = Spreadsheet::ParseExcel::Workbook-
>Parse('Excel/Test97.xls');
my($iR, $iC, $oWkS, $oWkC);

foreach my $oWkS (@{$oBook->{Worksheet}}) {
    print "--------- SHEET:", $oWkS->{Name}, "\n";
    for(my $iR = $oWkS->{MinRow} ; defined $oWkS->{MaxRow} && $iR <=
$oWkS->{MaxRow} ; $iR++) {
        for(my $iC = $oWkS->{MinCol} ; defined $oWkS->{MaxCol} && $iC <=
$oWkS->{MaxCol} ; $iC++) {
            $oWkC = $oWkS->{Cells}[$iR][$iC];
            print "( $iR , $iC ) =>", $oWkC->Value, "\n" if($oWkC);
        }
    }
}
```

For my needs, I did not need to go through every cell, as the data I needed to pluck out of the .xls file was always in certain locations, with a few exceptions. However the iterative process is very much the same.

The raw text in each cell is contained in `$cell->Value`. Unfortunately (for me) the powers that be wanted the font attributes as well as the raw text. These attributes are stored in the `$cell->Rich` array reference. `perldoc Spreadsheet::ParseExcel` gives us some information:

```
Rich
    Array ref of font informations about each characters.
    Each entry has : [ Start Position, Font Object]

Font
    Format class has these properties:
       Name
            Name of that font.
       Bold
            Bold (or not).
       Italic
            Italic (or not).
       Height
            Size (height) of that font.
       Underline
            Underline (or not).
       UnderlineStyle
            0: None, 1: Single, 2: Double, 0x21: Single(Account), 0x22:
Double(Account)
       Color
            Color index for that font.
       Strikeout
            Strikeout (or not).
       Super
            0: None, 1: Upper, 2: Lower
```

With this knowledge I was able to create a basic Benshaw Quotation parser.

=Head From RTF to HTML

The first procedure I used to read in the .xls file, parse the `Rich` array list and insert HTML elements into a copy of the actual cell text was pretty straightforward:

```
$line = '';
$bold = $italics = $underline = $charsadded = 0;
```

```
        # we're only interested in column 2
        $cell = $worksheet->{Cells}[$row][2];
        $celltext = $cell->Value;

        if($cell->{Rich})
            {
            foreach my $rtfElem (@{$cell->{Rich}})
               {
               $insertpoint = $rtfElem->[0] + $charsadded;
               $fonttext = parseFont($rtfElem->[1]);

               substr($celltext, $insertpoint, 0) = $fonttext;
               $charsadded += length($fonttext);
               }
            }
```

Basically just go through each row, looking at column two and if the cell has any rich text elements, use `parseFont()` to convert the RTF text attribute change into HTML and insert that HTML text into the appropriate spot. `$charsadded` is necessary because as you add HTML tags to the original text value, subsequent RTF element insert points will move over by the number of characters you've added.

This almost works perfectly.

The only problem I ran across was if the cell text started with bold or italicized text, there would be no rich text element stating that – the rich text element would be in the cell's format block, not in the rich text block. Other than the brain-bending regular expressions, this was probably the single biggest stumbling point I had since I wasn't aware that cell formatting included default font attributes. So tack on these lines just after the `if($cell->{Rich})` line in the previous code block and we're all set:

```
        #determine the cell's default font attributes
        $cellfs = %{$cell->{Format}}->{Font};
        $cellftext = parseFont($cellfs);
```

After we've gone through every cell we're interested in and created HTML tags to represent the RTF attribute changes we are left with a messy but (for my needs) clean enough implementation of the text and all of its formatting. The above code block (and the `parseFont()` function) can be trivially expanded to handle font and colour changes in addition to the font attributes I was originally interested in.

=Head Cleaning Up, Stage One

I found it was far easier to clean up the resultant HTML in stages. The first stage consisted of cleaning up the spacing to (what I consider) normalized, clean HTML and inserting unordered list items. This is done at the same time as the actual "pull the data out of the .xls file" since it's almost entirely line-based, and for me each row was a line.

For the sake of simplicity, I considered every row in the original .xls file its own island – all HTML font elements had to be closed on every line. It's perhaps a little more "wordy" than continuing a font attribute to the next line, but it's a lot easier on the eyes and on the brain.

Cleaning up spaces is done with a few straightforward regular expressions:

```
# opening tags with a space following should have that space preceeding
# closing tags with a space preceeding should have that space following
        $celltext =~ s.<([BIU])> +. <\1>.g;
        $celltext =~ s. +</([BIU])>.</\1> .g;

# remove any leading and trailing spaces
        $celltext =~ s/^[ \t]+//;
        $celltext =~ s/[ \t]+$//;

# remove any multiple spaces
        $celltext =~ s/[ \t]{2,}/ /g;
```

The unordered list creation was a little trickier. Basically if a line started with ' -' , I change that raw character to an `<LI>` tag. I also had to keep track of whether there was a `<UL>` tag already in effect and add it if not. If I was in an unordered list and the line did not start with ' -' , I closed the unordered list. (I was lucky, there were no nested lists, but that could be handled by looking for spaces or tabs before the ' -' character and nesting appropriately.)

```
$litext = '';
if($celltext =~ s/^-[ \t]*//)
    {
    push(@stage1, "  <UL>"), $ul = 1 if(! $ul);
    $litext = "  <LI>";
    }
else { push(@stage1, "  </UL>"), $ul = 0 if($ul); }
```

The last cleanups in the first stage were to close the font tags, remove strings I wasn' t interested in, eliminate empty tags, create paragraph markers and finally push the line out on to the `@stage1` array without duplicating lines.

Closing font tags wasn' t done very intelligently. I could have kept better track of how the tags were created (i.e. the order in which they were created) and close them in reverse order, but for this particular task it wasn' t necessary since they were always opened in the same order and never nested:

```
$line .= "</U>" x $underline;
$line .= "</I>" x $italics;
$line .= "</B>" x $bold;
```

where `$underline`, `$italics` and `$bold` are the count of times the font attribute was opened. Removing strings I didn' t want was just a simple set of `s///i` regexps (i = case insensitive). Removing empty tags was done recursively since eliminating one set of empty tags could create another set of empty tags (e.g. `<B><I></I></B>`):

```
do { $match = $line =~ s.<([BIU])></\1>..g; } while($match);
```

Creating paragraphs (and eliminating double paragraphs) was tricker than I first thought – It was easy to look for an empty line and substitute `<P>`, but that could create strings of empty paragraphs (e.g. with three blank lines, there would be three lines of `<P>`. That is where the "push the line to `@stage1` unless it' s the same as the last line" came in. Problem solved!

```
# create paragraphs wherever there are blank lines
    $line = '<P>' unless $line =~ /[^ ]+/;
# push the line out, but don't push duplicate lines
    push @stage1, $line unless $line eq $lastline;
    $lastline = $line;
```

=Head Cleaning Up, Stage Two

If you were paying attention you would have noticed that I wasn't closing my `<P>` tags in the previous stage. The stages were split up into different types of work; stage one involved itself mainly in getting the individual lines into good form, stage two deals with looking at multi-line cleanups, and stage three with converting the HTML representation of the quotation into something which can be used for more than a single horsepower, voltage and enclosure selection.

The cleanups involved in stage two consisted of adding `<BR>` where necessary and removing unnecessary unordered list creations. I work on the latter first.

You'll remember that in stage one I convert blank lines into `<P>` tags and then eliminate lines which are the same as the previous one. With the original .xls quotes it was possible to have a list split across two pages. This would create three consecutive lines which read `</UL><P><UL>` -- I search for this and then `undef`ine the lines if the pattern is found. (Later on I don't copy lines if they "don't exist.").

Adding `<BR>` only when necessary needed a little bit of regular expression work, but nothing too crazy yet:

```
$line .= "<BR>" unless $line =~ /<\/?(UL|LI|P)>/ || $stage1[$i + 1] =~
/<\/?(UL|LI|P)>/ || ! length($line);
```

As you can see, I tack on `<BR>` unless the line has an opening or closing UL, LI or P HTML tag or if the next line has such a tag, or the line has been rubbed out from my cleaning up in the previous paragraph.

Any lines which weren't `undef`ined get pushed in to the `@stage2` array. I also clean up my `<P>` tags by inserting a `</P>` before any `<P>`'s found:

```
push @stage2, $line if length($line);
push @stage2, "</P>" if $stage1[$i + 1] =~ /<P>/;
```

=Head ache (Stage Three, regexps)

Stage three passes all lines through a `scantemplate()` function. This function applies a number of search and replace regular expressions to convert numeric and text data into `<TMPL_VAR>` tags suitable for `HTML::Template`.

I'm going to look at a few of these regular expressions in detail here.

```
        # RSD6-100-600-C c/w
        s/(RSD|RSM|RDB|RMB|MVRSM)(4|6|9|11|12|18)-\d+-\d+-([^- ]*)/\1\2-
        <TMPL_VAR NAME=HP>-<TMPL_VAR NAME=VOLTS>-<TMPL_VAR NAME=ENCL>/;
```

This regular expression looks for a model number, horsepower, voltage and enclosure rating and replaces them with `HTML::Template` tags. There isn' t anything too scary to this one.

```
s/                                                              # search and replace
(RSD|RSM|RDB|RMB|MVRSM)                                 # match any of these into \1
(4|6|9|11|12|18)-                  # and then any of these into \2 followed by -
\d+-                                             # 1 or more digits followed by -
\d+-                                           # 1 more more digits followed by -
([^- ]*)                # match 0 or more non-'-' and non-space chars into \3
/                                            # and replace the matched text with
\1                                               # whatever matched into \1 above
\2                                       # followed by whatever matched \2 above
-<TMPL_VAR NAME=HP>-<TMPL_VAR NAME=VOLTS>-<TMPL_VAR NAME=ENCL>   # literal text
/;                                                          # end search and replace
```

So "RSD6-10-600-C c/w" would put "RSD" into \1, "6" into \2 and "C" into \3 – the resultant string would be "RSD6-<TMPL_VAR NAME=HP>-<TMPL_VAR NAME=VOLTS>-<TMPL_VAR NAME=ENCL> c/w"

I don' t need to put a match \3 at all, I could have replaced `([^- ]*)` with `(?:[^- ]*)` -- `(?:)` is the the non-storing form of `()`.

Why does the "c/w" not get zapped? Because I don' t ever match it – the regexp only looks for the model followed by the number of SCRs followed by a -numbers-numbers-(nonspace,non-) and replaces it with model, SCRs and some literal text.

```
        # Starter Fault Withstand Rating : 42 K Symmetrical Amps
        s/(Fault Withstand Rating)[^\d]+\d+/\1: <TMPL_VAR
        NAME=FAULT_WITHSTAND>/;
```

```
s/                                                              # search and replace
(Fault Withstand Rating)      # match the text "Fault Withstand Rating" into \1
[^\d]+                                   # match any non-digit one or more times
\d+                                          # match any digit one or more times
/                                          # replace the text matching above with
\1                                                       # Whatever matched into \1
: <TMPL_VAR NAME=FAULT_WITHSTAND>                                  # literal text
/;                                                                       # done.
```

Looking at this I can see one potential improvement right off the bat – why match "Fault Withstand Rating" into \1? It' s always going to match and always be the same thing, so why not just use `s/Fault Withstand Rating[^\d+]\d+/Fault Withstand Rating: <TMPL_VAR NAME=FAULT_WITHSTAND/;` ? I could have done that. I would have written the text "Fault Withstand Rating" twice and made the line longer. I guess I was lazy.

This does, however, demonstrate an important tenet of Perl: TMTOWTDI – There' s More Than One Way To Do It. I am sure that someone with better regular expression skills than I could have done all of this more elegantly but truth of it is that it doesn' t matter that much – this isn' t in a tight loop and this shaving cycles off isn' t going to help anything.

Now let' s look at something a little crazier:

```
        # DIMENSIONS 27"H X 10"W X 7 7/8"D
        s/(DIMENSIONS)(?![^<]+plus feet)[^<]*/\1: <TMPL_VAR NAME=DIM_CHASSIS>/;
```

See that "`(?! ... )`" block? That' s pretty much what I said when I started using look-aheads and look-behinds. WTF?!

Let' s break down the regexp like the others:

```
s/                                                         # search and replace
(DIMENSIONS)                               # match the text "DIMENSIONS" into \1
[^\d]+                                   # match any non-digit one or more times
(?!           # negative-assertion look ahead (i.e. Match if I don't see)
[^<]+                                             # one or more non-< characters
plus feet)                                             # followed by "plus feet"
[^<]*                                              # followed by no < characters
/                                                # replace the matched text with
\1                                                    # the text matched into \1
: <TMPL_VAR NAME=DIM_CHASSIS>                             # and this literal text
/;                                                                       # done.
```

Basically things started getting a little complex because there are two dimensions lines – one for a unit as a chassis and one for the unit in an enclosure. In this particular case, I am trying to match the ' ##"W x ##"H x ##"D' part of the dimensions lines, but I want to make sure I don' t match the enclosure dimensions line (the one with "plus feet").

The look-ahead I used "`(?! ... )`" is a negative assertion forward-looking look-ahead. That means that Perl "bookmarks" where it is and then scans the rest of the line. If it *does not* find what I specify, it considers the block to have matched, goes back to its bookmark and tries to match the next part of the regular expression. So what I am doing is telling Perl that I want to match a line that has the word "DIMENSIONS" followed by anything BUT "plus feet" and replace it with the template for a chassis dimensions line. The `[^<]` bits are so that I don' t match any HTML tags that may be in the line.

The regexp which matches the dimensions for the unit in an enclosure is simpler:

```
        # DIMENSIONS 30"H X 12"W X 8"D / plus feet (NEMA 1, 4, 12)"
        s/(DIMENSIONS).+plus feet[^<]*/\1: <TMPL_VAR NAME=DIM_ENCL>/;
```

Now I only use negative assertion look-ahead regular expressions in my code, but earlier implementations used more complex regexps which used some of the other varieties. `man perlre` will give you a lot more data more clearly than I will be able to.

package QuoteParse;

I created a package I called QuoteParse which took a filename and spat back an array of lines which represented the quote but in HTML and template-ized. It' s built to be modular and expandable so that if I need to do this all again it shouldn' t be too difficult to fall into the groove again.

I am a big big fan of `use strict` and using lots of comments. I hope that the code is more or less easy to understand and that I haven' t left any confusing bits unexplained.

All of the files described here can be used by anyone if they find they need to do something similar in their ventures.

=Head Why HTML?

Good question. I had originally intended this to be used to rattle through all of the quotation templates and then store these automatically-generated files into a database, write a web-based front-end and have our internal sales people use that to do the majority of the quotations. Unfortunately there were more subtle variations in the quote templates than I had anticipated and it was determined that HTML wasn' t going to do everything for us in the end.

I had also had an epiphany and found XWT, the XML Windowing Toolkit. These HTML templates now had to be converted to XML and then massaged further. The web-based app is now an XWT application and things seem to be going smoothly.

=Head Links

irc.freenode.net #perl #xwt
http://www.xwt.org
http://www.perldoc.com/perl5.6.1/pod/perlre.html
http://www.english.uga.edu/humcomp/perl/regex2a.html
http://www.troubleshooters.com/codecorn/littperl/perlreg.htm

A splitpath that works **properly** (troubleshooters.com is *not* correct):

```
($path, $file) = m#(.*/+)*(.*)$#;

[end]
```