Parsing HTML Tags in SAS Using Perl Regular Expressions

Don Boudreaux, PhD, SAS Institute Inc., Austin, TX

ABSTRACT

One of the many new features offered by SAS®9 is the ability to use Perl Regular Expressions to manipulate text. This investigation considers the use of Perl Regular Expressions to parse attribute data out of HTML tags. The expression used is sequentially defined and presented to the reader in order to introduce some of the functionality of Perl Regular Expressions. The new SAS®9 functions needed to process the resulting expression are also discussed. In addition, the example compares the Perl Regular Expression approach with an alternative approach using traditional character functions.

INTRODUCTION

SAS has always provided a number of functions for handling character data. Traditional functions like: **index()**, **substr()**, and **scan()** are good examples. And, as new versions of SAS have become available, enhancements to existing functions, new functions, and additional character handling capabilities have always been forthcoming. An example of this, new in SAS®9, is the ability to process character data with Perl Regular Expressions.

This paper will investigate a subset of this new capability to demonstrate using a Perl Regular Expression pattern (and associated functions) to parse information out of HTML tags. Specifically, consider the need to extract the value of the type attribute from the following HTML tag:

```
<input type="checkbox" name="tag1" />
```

Visually, this would involve looking for the word '`type`', an equals sign, and the attribute value written within the double quotes following the equals sign. Using traditional character functions, one strategy could involve finding the position of the word '`type`' and then from that position on, look for the second word. Care would have to be taken to consider both the case of the word to be searched and the delimiter used to indicate word boundaries. Note that both of these considerations and the representation of the initial search word involve exact character matches. The use of SAS Perl functions to find and parse the same information out of a tag would involve initially specifying a Perl Regular Expression pattern – which typically would not involve just exact matches. But for SAS coders, it does require learning a new and rather different code syntax.

PERL REGULAR EXPRESSIONS

Perl Regular Expressions typically involve a combination of characters, metacharacters, and quantifiers. In the context of our example, the expression will use characters and quantifiers (see references for information on common metacharacters and other regular expression elements). The syntax of the expression will involve placing a search specification pattern within slashes. If the intend is to just find the word '`type`', the expression would look like this:

```
/type/
```

Unfortunately, this word might appear in lower case, upper case, or mixed case. Without having to change the case of the text being searched, Perl Regular Expressions can specify a collection of values called a character class to look for by grouping single characters together within square brackets. On a character by character basis, looking for any case variation of the word 'type' could be specified as:

```
/[Tt][Yy][Pp][Ee]/
```

This would find '`type`', '`TYPE`', '`Type`', '`tyPe`', or any other mixed-case variation of the word. But the word could be part of another tag's value, so it would be advantageous to try and match it as an attribute name – a word with a trailing equals sign. Therefore consider:

```
/[Tt][Yy][Pp][Ee] = /
```

Which will work, but only if the '`type`' name is followed by a single blank, the equals sign, and then another single blank. In fact, on either side of the equals sign there could be one space, two spaces, more than two spaces, or no spaces. Fortunately, this is not a significant problem to handle without changing the original data value. In Perl Regular Expressions a character (or character set) can be described as being present with a quantifier. The syntax to specify a quantifier is a set of curly braces inclosing a number range. However, there are also several number ranges that are so commonly used that they have a simplified syntax specification:

{n,m}	match at least n but not more than m times
?	match 0 or 1 times, shorthand for {0,1}
{ n,}	at least n times
*	match 0 or more times, or use: {0,}
+	match 1 or more times, or use: {1,}
{n}	exactly n times

Continuing on with the example, add in the following:

```
/[Tt][Yy][Pp][Ee] *= */
```

This is now specifying any case variation of '`type`', followed by 0 or more spaces, followed by an equals sign, and finally followed by another set of 0 or more spaces. Next add-on the quotes, which will be shown as two single-character character classes:

```
/[Tt][Yy][Pp][Ee] *= *["]["]/
```

This would be reasonable, if the type attribute was only specified as having a double-quoted null value. Assuming this is not always true, it is necessary to look for a value between the quotes that could be a word or set of words with multiple or no spaces included. And although this initially sounds difficult, what it really means is that we are looking for anything between the quotes that is not another quote. Now, to this point our character classes have contained the character or characters that we wanted. It is also possible with Perl Regular Expressions to add a caret at the beginning of a character class specification to designate that the class contains the characters you do not want. In our case any number of non-quote characters could make up the value. Consequently, our expression would now look like the following:

```
/[Tt][Yy][Pp][Ee] *= *["][^"]*["]/
```

Which is a valid Perl Regular Expression and will find the type attribute and its associated value. However, to be able to extract part of a found pattern it is also necessary to mark it. Within the expression this only involves enclosing the desired part within parentheses; making the final version of this expression:

```
/[Tt][Yy][Pp][Ee] *= *["]([^"]*)["]/
```

PERL REGULAR EXPRESSION FUNCTIONS

The SAS®9 functions that are needed to process the Perl Regular Expression developed earlier and extract the value form the specified attribute are: **prxparse()**, **prxmatch()**, and **prxposn()**. The **prxparse()** function takes a Perl Regular Expression pattern as an argument, compiles it, and returns an identification number used by the other Perl Regular Expression functions. If there is a problem with the expression, a missing value is returned. The **prxmatch()** function searches a text source for a match on a given pattern. The two arguments it takes are the identification number for a compiled expression and the text source. A zero is returned if the pattern of the Regular Expression is not found within the text source. Otherwise a number representing the position of the pattern within the text is returned. The **prxposn()** function will extract grouped information from a text source. The arguments for this function are: the identification number for a pattern, the number position for the group within the expression to extract, and the source text. This function requires that the **prxparse()** and **prxmatch()** functions have already compiled the expression (to create the identification number) and matched on the same source text.

HTML PARSING EXAMPLE DATA AND CODE

The first SAS data step shown below will load a set of HTML tags into a SAS data set called tag_data. It is important to emphasize that the tags are assumed to be well-formed following the HTML v4 or XHTML syntax specifications. In particular, for this example code (which does not do any significant value filtering), the tag attribute values are assumed to be quoted and not contain any embedded quotes. The only modification made to the original HTML tag data is to convert all single quotes into double quotes. Do note, for testing purposes, the fifth tag is just a comment, and the sixth tag has the word '`type`' entered as part of the name attribute.

The second SAS data step reads the tag data and attempts to extract out the value of the type attribute using both Perl Regular Expressions and traditional character functions. The variable named `regEx` contains the extracted value using the regular expression pattern described earlier and the variable named `cFunc` contains the extracted value using the traditional character functions. The strategy for using the traditional character functions is to:

- Find the position of the word ‘TYPE’ within the uppercase value of the HTML tag using the **index()** function.
- Create a new variable, using the **substr()** function, that contains the text of the HTML tag starting from the position of the word ‘TYPE’ through the end of original value.
- Find the second word in the new variable using **scan()** and a double quote as the word delimiter.

Notice this strategy does not involve working with the equals sign – assumed to be between the name of the tag attribute and the quoted value. This provides some relief from having to consider the spacing (or lack of spacing) around it. Care was also taken not to change the case of the text in the tag. This helps preserve the case of the extracted attribute value.

```
Data tag_data ;
  Input HTMLtag $ 1-60 ;                * input HTML tag data ;
  HTMLtag = tranwrd(HTMLtag,"'",'"') ;  * convert to double quotes ;
Datalines ;
<input type="checkbox" name="tag1" />
<input name ="tag2"      type ='radio' />
<input type="hidden"     name="tag3"/>
<input name="tag4" type=   'reset'/>
<!-- look to parse type and = and "value" -->
<input name="tag type" type="hidden" value="?" />
;

%Let pattern = '/[Tt][Yy][Pp][Ee] *[=] *["]([^"]*)["]/' ;

Data type_value ;
  Set tag_data ;

  id = prxparse(&pattern) ;             * create pattern id ;
  mx = prxmatch(id,HTMLtag) ;           * match for pattern in tag ;
  regEx = prxposn(id,1,HTMLtag) ;       * extract attribute value ;

  pos = index(upcase(HTMLtag),"TYPE") ; * find position of word TYPE ;
  type_toEnd = substr(HTMLtag,pos) ;    * substr from TYPE to end ;
  cFunc = scan(type_toEnd,2,'"') ;      * 2nd word is attribute value ;
Run ;

Title1 "regEx: using Perl Regular Expressions" ;
Title2 "%superq(pattern)" ;
Title3 " " ;
Title4 "cFunc: using index(),substr(),and scan()" ;
Title5 "traditional datastep character functions" ;
Proc Print data = type_value ;
   Var regEx cFunc HTMLtag ;
Run ;
```

HTML PARSING EXAMPLE RESULTS

The processing results are shown below in the output from the Proc Print. For the first four observations both the Perl and traditional functions have extracted the same values. The fifth observation was not successfully parsed by the traditional functions. Additional processing would be needed to help them correctly identify that word ‘`type`’ was not part of an HTML tag attribute. Picking up the second word past ‘`type`’ using the double quote as the word boundary was the problem here. The sixth observation was also a problem for the traditional functions. The word ‘`type`’ in the value of the name attribute was the reason the incorrect value was extracted for this observation. And, although these issues could be addressed by augmenting the existing traditional function code segment with additional programming logic, it is remarkable how well the Perl strategy handled these data abnormalities.

```
                 regEx: using Perl Regular Expressions
                '/[Tt][Yy][Pp][Ee] *[=] *["]([^"]*)["]/'

                cFunc: using index(),substr(),and scan()
                traditional datastep character functions

 Obs   regEx       cFunc         HTMLtag

  1    checkbox    checkbox    <input type="checkbox" name="tag1" />
  2    radio       radio       <input name ="tag2"      type ="radio" />
  3    hidden      hidden      <input type="hidden"     name="tag3"/>
  4    reset       reset       <input name="tag4" type=   "reset"/>
  5                value       <!-- look to parse type and = and "value" -->
  6    hidden       type=      <input name="tag type" type="hidden" value="?" />
```

CONCLUSION

The use of Perl Regular Expressions provides the SAS user with a flexible and sometimes more powerful alternative to traditional character function processing. Admittedly the syntax for Perl Regular Expressions is complex and not at all similar to any other part of SAS. Nevertheless, even with the simple example shown here, the use of this new capacity provides a very succinct and elegant way to process text information.

APPENDIX

Modifying the pattern developed earlier for other attributes is simply a matter of switching out the letters of the word '`type`' for other attribute alternatives. Nevertheless, there are other information sources within an HTML document. The following patterns process several of these sources:

Obtaining the contents of comment tags: `'/<!-- *(.*) *-->/'`
Finding the default text within a textarea: `'/<textarea .*> *(.*) *</textarea>/i'`
Finding the title text from the title tag: `'/<title *.*> *(.*) *</title>/i'`
Reading the tag type (input, form, etc): `'/^<(\w*)\b/'`

In these patterns, the period matches any word character, the '`i`' indicates the search is case insensitive, the tilde at the beginning of a pattern forces the match to start at the beginning of the source, the \w matches on word characters, and the \b is looking for a word boundary. Detailed information about these keywords can be found in the resources noted below or in any number of standard Perl or Regular Expression references.

RESOURCES

SAS Institute Inc. 2004. *New Features in SAS System 9*.
Cary, NC: SAS Institute Inc.

Base SAS Community, SAS Institute Inc.
Using Regular Expressions: http://support.sas.com/rnd/base/topics/datastep/perl_regexp/regexp.motivation.html
Regular Expression Tip Sheet: http://support.sas.com/rnd/base/topics/datastep/perl_regexp/regexp-tip-sheet.pdf

Murdock, Kelly. 2000. *Master VISUALLY HTML 4 and XHTML 1*
Forest City, CA: IDG Books Worldwide, Inc.

CONTACT INFORMATION

Please forward comments and questions to: Don Boudreaux, PhD
E-mail: don.boudreaux@sas.com