PharmaSUG 2014 - Paper BB08

Express Yourself! Regular Expressions vs SAS Text String Functions

Spencer Childress, Rho®, Inc., Chapel Hill, NC

ABSTRACT

SAS® and Perl regular expression functions offer a powerful alternative and complement to typical SAS text string functions. By harnessing the power of regular expressions, SAS functions such as PRXMATCH and PRXCHANGE not only overlap functionality with functions such as INDEX and TRANWRD, they also eclipse them. With the addition of the modifier argument to such functions as COMPRESS, SCAN, and FINDC, some of the regular expression syntax already exists for programmers familiar with SAS 9.2 and later versions. We look at different methods that solve the same problem, with detailed explanations of how each method works. Problems range from simple searches to complex search and replaces. Programmers should expect an improved grasp of the regular expression and how it can complement their portfolio of code. The techniques presented herein offer a good overview of basic data step text string manipulation appropriate for all levels of SAS capability. While this article targets a clinical computing audience, the techniques apply to a broad range of computing scenarios.

INTRODUCTION

This article focuses on the added capability of Perl regular expressions to a SAS programmer's skillset. A regular expression (regex) forms a search pattern, which SAS uses to scan through a text string to detect matches. An extensive library of metacharacters, characters with special meanings within the regex, allows extremely robust searches.

Before jumping in, the reader would do well to read over 'An Introduction to Perl Regular Expressions in SAS 9', referencing page 3 in particular (Cody, 2004). Cody provides an excellent overview of the regex and a convenient table of the more common metacharacters, with explanations. Specifically, knowledge of the basic metacharacters, [\^$.|?*+(), goes a long way. Additionally, he covers the basics of the PRX suite of functions.

SAS character functions and regexes have many parallels. They both perform searches, search and replaces, and modifications. A clear breakdown and understanding of their similarities and differences allow a programmer to choose the most powerful method for dealing with text fields.

SAS MODIFIERS AND REGEX EQUIVALENTS

The SAS modifier, introduced in SAS 9, significantly enhances such functions as COMPRESS, SCAN, and FINDC. SAS modifiers are to regex character classes what Vitamin C is to L-ascorbic acid: an easily remembered simplification. A programmer with an understanding of these modifiers can jump right into regex programming.

Table 1 illustrates the relationship between SAS modifiers and regex character class equivalents:

SAS Modifier	SAS Definition	POSIX Character Class	Regex Option	Regex Explanation
a or A	adds alphabetic characters to the list of characters.	/[[:alpha:]]/		
c or C	adds control characters to the list of characters.	/[[:cntrl:]]/		
d or D	adds digits to the list of characters.	/[[:digit:]]/	/\d/	\d is the metacharacter for digits.
f or F	adds an underscore and English letters (that is, valid first characters in a SAS variable name using VALIDVARNAME=V7) to the list of characters.		/[a-zA-Z_]/	A character class defined within square brackets has a different set of metacharacters. For example, a '-' represents a range within square brackets and a literal dash outside. As such, 'a-z' captures all lowercase letters.

g or G	adds graphic characters to the list of characters. Graphic characters are characters that, when printed, produce an image on paper.	/[[:graph:]]/		
h or H	adds a horizontal tab to the list of characters.		/\t/	\t is the metacharacter for tab.
i or I	ignores the case of the characters.		/expression/i	The 'i' after the second delimiter of the regex tells the regex to ignore case in ‘expression’.
k or K	causes all characters that are not in the list of characters to be treated as delimiters. That is, if K is specified, then characters that are in the list of characters are kept in the returned value rather than being omitted because they are delimiters. If K is not specified, then all characters that are in the list of characters are treated as delimiters.		/[^expression]/	The '^', as the first character of a character class enclosed in square brackets, negates 'expression'. That is, this character class matches everything not included in ‘expression’.
l or L	adds lowercase letters to the list of characters.	/[[:lower:]]/		
n or N	adds digits, an underscore, and English letters (that is, the characters that can appear in a SAS variable name using VALIDVARNAME=V7) to the list of characters.		/[a-zA-Z_0-9]/	Similar to SAS modifier 'f', 'n' adds digits. To match, a character class needs only the range 0-9 added to the character class equivalent of 'f'.
o or O	processes the charlist and modifier arguments only once, rather than every time the function is called.			Equivalent to initializing and retaining the regex ID with PRXPARSE at the top of the data step, rather than initializing it at each data step iteration.
p or P	adds punctuation marks to the list of characters.	/[[:punct:]]/		
s or S	adds space characters to the list of characters (blank, horizontal tab, vertical tab, carriage return, line feed, and form feed).	/[[:space:]]/	/\s/	\s is the metacharacter for invisible space, including blank, tab, and line feed.
t or T	trims trailing blanks from the string and charlist arguments.		/ \b/	The word boundary metacharacter \b, positioned after a space, prevents a regex from matching trailing blanks.
u or U	adds uppercase letters to the list of characters.	/[[:upper:]]/		
w or W	adds printable (writable) characters to the list of characters.	/[[:print:]]/		
x or X	adds hexadecimal characters to the list of characters.	/[[:xdigit:]]/		

Table 1. SAS Modifiers and Equivalent POSIX Character Classes and/or Regexes

POSIX character classes are collections of common characters and map not only to a subset of SAS modifiers, but to the ANY and NOT collection of functions such as ANYALPHA or NOTPUNCT. However other modifiers do not map directly, such as 'n', which can be used to identify appropriately named SAS variables. Note that character classes within square brackets can be customized extensively to identify any set of characters.

BYTE offers a simple method to check which characters a character class identifies. The code snippet below makes for an excellent SAS abbreviation to test regexes:

```
data test;
    file print;

    do i = 0 to 255;
        char  = byte(i);
        regex = prxmatch('/expression/', char);
        put i char regex;
        output;
    end;
run;
```

This data step creates a dataset called 'test' and prints to the Output window. By feeding the function BYTE values ranging from 0 to 255, SAS illustrates the ASCII or EBCDIC collating sequence. In Windows, Unix, and OpenVMS operating system environments, 0 through 127 comprise the standard set of ASCII characters while 128-255 vary between OS environments.

Within the regex, 'expression' represents the character class of interest. PRXMATCH matches the regex in its first argument against each character captured in variable 'char'. If the character matches PRXMATCH returns a 1.

SEARCHING TEXT

One of a SAS programmer's most common tasks involves text searches. Below, examples range from simple to complex.

SEARCHING TEXT – INDEX

INDEX might very well be the first function a programmer uses. It returns the position of the first occurrence of a substring within a string:

```
data Search_INDEX;
        indexed = INDEX('asdf', 'sd');
    put indexed;
run;
```

INDEX searches source string 'asdf' for substring 'sd'. As one would expect the variable INDEXED returns a 2, corresponding to the second character in 'asdf'.

The same outcome can be accomplished with PRXMATCH:

```
data Search_PRXMATCH;
        prxmatched = PRXMATCH('/sd/', 'asdf');
    put prxmatched;
run;
```

PRXMATCH takes as its first argument either the regex itself or a regex ID and the source string as its second. The forward slashes in the first argument are called delimiters, which open and close the regex. Everything in between them is the search pattern.

SAS generates a regular expression ID which defines the regex at each invocation of a PRX function. Thus, to reduce processing time, a regex could be defined at the top of a dataset and retained like so:

```
data Retain_PRXPARSE;
            retain regexid;
    if _n_ = 1 then regexid = prxparse('/sd/');

        prxmatched = prxmatch(regexid, 'asdf');
    put prxmatched;
run;
```

PRXPARSE only processes a regex; it does not match it against anything. For simplicity's sake, PRXPARSE will not appear in code examples.

SEARCHING TEXT – HANDLING CHARACTER CASE

INDEX cannot inherently account for letter case. Suppose the letter case of the source string is unknown. In this situation INDEX would require the services of UPCASE or LOWCASE:

```
data Search_INDEX;
        indexed = INDEX('ASDF', UPCASE('sd'));
    put indexed;
run;
```

As one might expect, the substring needs to be hardcoded to 'SD' or nested within UPCASE; otherwise, INDEX might come back empty-handed. A regex handles letter case a little more easily:

```
data Search_PRXMATCH;
        prxmatched = PRXMATCH('/sd/i', 'ASDF');
    put prxmatched;
run;
```

Notice the regex now contains an 'i' after the closing forward slash. This modifier simply ignores case in the source string 'ASDF'.

SEARCHING TEXT – DIGITS

FINDC trumps INDEX when dealing with character classes because of its modifier argument. Suppose one is interested in identifying any digit:

```
data Search_FINDC;
        found = FINDC('2357', , 'd');
    put found;
run;
```

The modifier 'd' in the third argument identifies any digit in a string. Similarly, with a regex, the character class '\d' applies:

```
data Search_PRXMATCH;
        prxmatched = PRXMATCH('/\d/', '2357');
    put prxmatched;
run;
```

Notably, PRXMATCH handles the functionality of both INDEX and FINDC.

SEARCHING TEXT – DATES

Dates are wonderful bits of data. They come in all shapes and sizes, at times in the same variable. This variability can raise significant programming hurdles which the regex mitigates. With a few regexes, any date can be identified. 'DATEw.' and 'YYMMDDw.' provide excellent examples:

```
data Search_DATEw;
  *Four 'DATEw.' examples, the last of which is not a valid date.;
  dates = '05jan1986 5jan1986 05jan86 05jau1986';

  do i = 1 to countw(dates);
    *Matching simply two digits followed by a valid three character
     month followed by four digits (DDMMMYYYY).;
    datew1 = prxmatch('/\d\d(jan|feb|mar|apr|may|jun|'
                        || 'jul|aug|sep|oct|nov|dec)\d{4}/i',
                    scan(dates, i));

    *Matching as above except with optional leading '0' to month and day.;
    datew2 = prxmatch('/[0123]?\d(jan|feb|mar|apr|may|jun|'
                            || 'jul|aug|sep|oct|nov|dec)\d{4}/i',
                    scan(dates, i));
```

```
      *Matching as above except with optional first two digits of year.;
      datew3 = prxmatch('/[0123]?\d(jan|feb|mar|apr|may|jun|'
                                || 'jul|aug|sep|oct|nov|dec)(19|20)?\d\d/i',
                        scan(dates, i));
      output;
    end;
  run;
```

In the second and third examples, the metacharacter '?' tells the regex to look for the preceding character or group of characters 0 or 1 time, essentially classifying it as optional. In the third example the parentheses surround two two-digit numbers, '19' and '20'. The '|' is the alternation operator, equivalent to 'or', and tells the regex to match the pattern to the left or the pattern to the right.

```
  data Search_YYMMDDw;
    *Five 'YYMMDDw.' examples.;
    dates = '1986-01-05 1986-1-5 86-05-01 1986/01/05 19860105';

    do i = 1 to (count(dates, ' ') + 1);
      *Matching 'clean' YYMMDD10 date (YYYY-MM-DD).;
      yymmddw1 = prxmatch('/\d{4}-\d\d-\d\d/',
                          scan(dates, i, ' '));

      *Matching as above except with optional leading '0' to month and day.;
      yymmddw2 = prxmatch('/\d{4}-[01]?\d-[0123]?\d/',
                          scan(dates, i, ' '));

      *Matching as above except with optional first two digits of year.;
      yymmddw3 = prxmatch('/(19|20)?\d\d-[01]?\d-[0123]?\d/',
                          scan(dates, i, ' '));

      *Matching as above except regex accepts any punctuation delimiter.;
      yymmddw4 = prxmatch('/(19|20)?\d\d[[:punct:]][01]?\d[[:punct:]][0123]?\d/',
                          scan(dates, i, ' '));

      *Matching as above except with optional delimiter.;
      yymmddw5 = prxmatch('/(19|20)?\d\d[[:punct:]]?[01]?\d[[:punct:]]?[0123]?\d/',
                          scan(dates, i, ' '));
      output;
    end;
  run;
```

In the fourth example, use of the POSIX character class PUNCT allows the regex to accept any punctuation mark as a delimiter, including '-' and '/'. In the final example, applying '?' to the punctuation character class makes the delimiter optional.

PRXMATCH harnesses the power of the regex to match a wide range of text patterns, all within the bounds of a single function.

SEARCH AND REPLACE

Modifying a text string logically follows searching for a text string. A number of SAS functions modify text strings: COMPRESS eliminates specified characters; COMPBL reduces multiple blanks to a single blank; LEFT, TRIM, and STRIP remove leading, trailing, and both leading and trailing blanks; UPCASE and LOWCASE modify letter case; TRANWRD replaces one substring with another substring. That is a long list of functions, a list which PRXCHANGE duplicates almost entirely.

A regex which searches and replaces has two parts: the search pattern between the first and second delimiters, just like in PRXMATCH, and the replacement pattern between the second and third delimiters. A regex search and replace has the basic form 's/<search pattern>/<replacement pattern>/'. Note the leading 's'; the function will cause an error and stop the data step without this signifier.

SEARCH AND REPLACE – COMPRESS

COMPRESS provides a good example of one of PRXCHANGE's parallels:

```
data Replace_COMPRESS;
        compressed = compress('abacadabra', 'a');
    put compressed;
run;
```

The second argument of COMPRESS tells it which character, in this case ‘a’, to dispense with in the first argument, ‘abacadabra’. Variable ‘compressed’ returns ‘bcdbr’.

PRXCHANGE works a bit differently:

```
data Replace_PRXCHANGE;
        prxchanged = prxchange('s/a//', -1, 'abacadabra');
    put prxchanged;
run;
```

This regex looks a little more complicated than one which simply searches. The leading ‘s’ dictates that this regex is a search and replace. The ‘a’ between the first two delimiters specifies what to search for. Notice the second and third delimiters have no intermediate characters. Effectively, this regex replaces ‘a’s in ‘abacadabra’ with nothing. However, to replace all ‘a’s, PRXCHANGE’s second argument must be ‘-1’; were it ‘1’, only the first ‘a’ would be removed.

Keep in mind that the power of the regex is that it can identify and modify a number of different characters with a character class enclosed in square brackets. The inverse subset of characters, i.e. everything except ‘a’, can be identified with a slight modification to the search:

```
data Replace_COMPRESS;
        compressed = compress('abacadabra', 'a', 'k');
    put compressed;
run;

data Replace_PRXCHANGE;
        prxchanged = prxchange('s/[^a]//', -1, 'abacadabra');
    put prxchanged;
run;
```

Both methods return ‘aaaaa’. Setting the third parameter of COMPRESS to ‘k’ compresses everything except ‘a’. Similarly, the character class ‘[^a]’ specifies everything but ‘a’.

SEARCH AND REPLACE – COMPBL

COMPBL reduces multiple blanks to a single blank:

```
data Replace_COMPBL;
        compbled = compbl('  abc def   ghi');
    put compbled;
run;
```

PRXCHANGE, in conjunction with a quantifier of the form ‘{n,m}’ where ‘n’ is the lower bound of the quantifier and ‘m’ the upper, behaves similarly:

```
data Replace_PRXCHANGE;
        prxchanged = prxchange('s/ {2,}/ /', -1, '  abc def   ghi');
    put prxchanged;
run;
```

Between the first two delimiters is the expression ‘ {2,}’, which identifies any sequence of two or more spaces. Notice the upper bound of the quantifier is missing; no upper bound instructs the search pattern to find any number of the previous character class or group of characters.

SEARCH AND REPLACE – LEFT, TRIM, STRIP

PRXCHANGE can also emulate LEFT, TRIM, and STRIP. The metacharacters ‘^’ and ‘$’ come in handy here:

```
data Replace_PRXCHANGE;
    *Leading blanks;
    lefted   = count(prxchange('s/^ +//',       1, '   abc xyz   '), ' ');

    *Trailing blanks;
    trimmed  = count(prxchange('s/ +$//',       1, '   abc xyz   '), ' ');
```

```
    *Both leading and trailing blanks;
    stripped = count(prxchange('s/(^ +| +$)//', 2, '   abc xyz   '), ' ');

    put lefted trimmed stripped;
run;
```

In the first example, '^' instructs the search pattern to search only the beginning of the text string. In the second, '$' searches only the end of the text string. And in the third, enclosing the first and second search patterns in parentheses and concatenating them with the alternation operator '|' captures both conditions. Notice that the second argument changed to '2'. With two conditions PRXCHANGE needs to match both patterns to alter both the leading and trailing blanks.

SEARCH AND REPLACE – UPCASE AND LOWCASE

PRXCHANGE can change letter case as well. This task is simple enough with UPCASE and LOWCASE:

```
data Replace_UPCASE;
        upcased = upcase('asdf');
    put upcased;
run;

data Replace_LOWCASE;
        lowcased = lowcase('ASDF');
    put lowcased;
run;
```

The regex requires a capture buffer and the application of a character class between the second and third delimiters:

```
data Replace_PRXCHANGE;
        upcased  = prxchange('s/(.*)/\U$1/', 1, 'asdf');
        lowcased = prxchange('s/(.*)/\L$1/', 1, 'ASDF');
    put upcased;
run;
```

Everything between the parentheses goes into what is known as a capture buffer. A search pattern can have any number of capture buffers, which may then be referred to between the second and third delimiters by '$n', where 'n' refers to the where the capture buffer falls in the entire sequence of capture buffers. In the above example, '$1' refers to the single capture buffer between the first two delimiters, '(.*)'. This search pattern effectively recognizes the entire source string: '.' is the wildcard metacharacter and '*' matches 0 or more instances of the previous character or group of characters.

To alter character case, apply either the character class '\U' or '\L', to up-case or low-case the subsequent capture buffer reference.

SEARCH AND REPLACE – TRANWRD

TRANWRD replaces all occurrences of a substring in a character string:

```
data Replace_TRANWRD;
        tranwrded = tranwrd('As easy as 1-two-three!', '1', 'one');
    put tranwrded;
run;
```

TRANWRD's arguments are three-fold: source string, target string, and replacement string. In this example, TRANWRD replaces all occurrences of '1' with 'one' to return 'As easy as one-two-three!'. PRXCHANGE easily reproduces the same functionality:

```
data Replace_PRXCHANGE;
        prxchanged = prxchange('s/1/one/', -1, 'As easy as 1-two-three!');
    put prxchanged;
run;
```

The search pattern in the above regex equates to the second argument of TRANWRD. The replacement pattern equates to the third argument of TRANWRD. These two functions diverge in that PRXCHANGE can replace one or more occurrences of '1'.

PRXCHANGE can reproduce most SAS text string modification functions. In some cases it adds unnecessary complexity, but in most cases it provides a superior, more versatile approach.

APPLICATION

Regexes have many practical applications, as the two examples below demonstrate.

APPLICATION – SPACE-DELIMITED TO COMMA-DELIMITED

A common macro parameter is a list of variables, space-delimited. Space-delimited values work great for keep lists, but not so well in PROC SQL or CALL MISSING. Thus, conversion to a comma-delimited list allows a list to be used in both cases:

```
%let VarList         = Var1 Var2  Var3   Var4 Var5;

%let TRANWRDCommas   = %sysfunc(tranwrd(&VarList, %str( ), %str(, )));
%let PRXCHANGECommas = %sysfunc(prxchange(s/%str( +)/%str(, )/, -1, &VarList));

%put &TRANWRDCommas;
Var1, Var2, , Var3, , , Var4, Var5
%put &PRXCHANGECommas;
Var1, Var2, Var3, Var4, Var5
```

Input list 'VarList' is space-delimited, but it has a variable number of spaces between each item. TRANWRD by itself behaves predictably, matching each space with a comma, but this new list will cause errors. Of course, %SYSFUNC(COMPBL()) could be nested around 'VarList', but PRXCHANGE handles this issue by itelf. With the quantifier '+', which signifies one or more, following the search pattern, any number of spaces can be replaced with a single ', '.

APPLICATION – INTEGERS, NON-INTEGERS, AND OTHER STATISTICS

The clinical programmer produces a LOT of tables. Without the regex, distinguishing a statistic in the form 'Median (Min, Max)' from one in the form 'Count (Percentage)' and their derivations is a tall task. They could be identified, but with no small amount of IF-THEN-ELSE clauses. Regexes can identify almost all statistics with just four (the final ELSE illustrates what would not be captured with the first four clauses):

```
%let width = 20;

data fin;
  set stats;

  array cols (0:3) $25 col5 col2-col4;

  *Pad column variables with spaces, for table alignment, assuming monospaced font.;
  do i = 0 to &ntrt;

    *Strip column variables to remove leading blanks.;
    cols(i) = prxchange('s/^ +//', 1, (cols(i));

    *Match integers first by searching for an optional dash and
     any number of digits, with trailing spaces only.;
    if prxmatch('/^(-?\d+) *$/', cols(i)) gt 0
        then cols(i) = repeat(' ', floor(&width/2) -
                                   length(cols(i)) - 1)
                                || cols(i);

    *Match statistics with commas.;
    else if prxmatch('/,/', cols(i)) gt 0
        then cols(i) = repeat(' ', floor(&width/2) -
                                   prxmatch('/,/', cols(i)))
                                || cols(i);
```

```
      *Match statistics with parentheses which do not conatin commas.;
      else if prxmatch('/ ?\(/', cols(i)) gt 0
          then cols(i) = repeat(' ', floor(&width/2) -
                                     prxmatch('/ ?\(/', cols(i)))
                                  || cols(i);

      *Match statistics with any other punctuation mark.;
      else if prxmatch('/[[:punct:]]/', (cols(i)) gt 0
          then cols(i) = repeat(' ', floor(&width/2) -
                                     prxmatch('/[[:punct:]]/', (cols(i))
                                  || cols(i);

      *Match anything else, but hopefully not.;
      if prxmatch('/[^ ]/', cols(i)) gt 0
          then cols(i) = repeat(' ', floor(&width/2) -
                                     length(cols(i)) - 1)
                                  || cols(i);
    end;
  run;
```

By identifying a search pattern for each type of statistic, the position of that pattern can be used to align all statistics along a common column.

For example, aligned summary statistics might look like so:

Characteristics	Boring N=9	Drugs! N=10
Age at screening (years)		
n	9	10
Mean	36.3	36.7
SD	7.16	6.53
Median	35.0	38.5
Range (Min, Max)	(23, 44)	(27, 47)
Ethnicity - n (%)		
Hispanic or Latino	1 (11.1)	0
Not Hispanic or Latino	8 (88.9)	10 (100.0)
Not Reported	0	0
Unknown	0	0

Table 2. Aligned Statistics

The common column is the punctuation mark, except for the integer and the percentage, which align on the column following the first ones digit. Thus subtracting the position of the common column from the midpoint of the table cell yields the number of leading blanks necessary to align each statistic

CONCLUSION

The regex is a powerful complement to a SAS programmer's repertoire of code. It can reduce multiple IF-THEN-ELSE statements to a single line of code. Not only does the regex allow a SAS programmer to improve his or her efficiency, he or she can perform searches which were previously impossible. And most importantly, crafting a regex is fun, so go out and express yourself!

REFERENCES

Cody, Ronald. 2004. "An Introduction to Regular Expressions in SAS 9." Proceedings of the SAS Global 2004 Conference. Montréal, Canada: Duke Owen, Conference Chair. Available at http://www2.sas.com/proceedings/sugi29/toc.html.

RECOMMENDED READING

http://www.regular-expressions.info/

CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact the author at:

Name: Spencer Childress
Enterprise: Rho, Inc.
Address: 6330 Quadrangle Dr
City, State ZIP: Chapel Hill, NC 27514
Work Phone: 919 408 8000
Fax: 919 408 0999
E-mail: spencer_childress@rhoworld.com