PharmaSUG2010 - Paper TT10

Add the Power of Regular Expressions to SAS Macro

Lei Zhang, Celgene Corporation, Summit, New Jersey

ABSTRACT

Regular expressions, as a powerful, efficient, and flexible text manipulation facility in many applications, have been gaining popularity in the SAS community since Perl compatible regular expressions were introduced in SAS 9. However, many SAS users are unaware that the power of regular expressions can be made available in the SAS macro language. The goal of this paper is to get you started with using this facility in SAS macros. The paper first explores key concepts and syntaxes of Perl regular expressions (PRX) and then describes the ways to use them in SAS macros by taking advantage of the PRX functions and CALL routines built in the DATA step via %SYSFUNC function and %SYSCALL statement. Numerous real-life examples are provided to expose the strengths and subtleties of Perl regular expressions when used in the SAS macros. The paper is intended for those who have some knowledge of using regular expressions in DATA step and want to put them to use within the context of SAS macro programming.

1. INTRODUCTION

SAS Macro, built on the top of the entire SAS System, provides a key capability of packing and making dynamic SAS code through macro *variables* and *macros*. As an advanced preprocessor to the SAS compiler, SAS Macro provides built-in functions for basic text manipulation, such as %SCAN, %SUBSTR, and %INDEX. However, if you are a macro developer, you will find from time to time that trying to do simple input validation or parse some trivial strings can be quite cumbersome with these functions, and sometimes that things may even get worse when you have a macro to develop and discover just how frequently some type of text manipulations is required in order to make everything work the way you really want.

A regular expression is a concise notation for a collection of strings. What makes regular expressions so powerful in many applications is that a single regular expression can represent a varying number of strings, without having to list all of them, and you can easily use it to match against strings. Once the matching substrings are discovered, you can handle them in any way you want. Regular expressions (which will be often referred to *regexes* or *patterns* from now on) are defined using a small domain-specific language that is completely different from SAS Base language, but, like many other programming languages and systems, the SAS System incorporates the facility smoothly so that you can seamlessly create and use regexes in DATA steps. Since SAS version 6, two different types of regular expressions have been built in the SAS DATA step: SAS Regular Expressions and Perl Compatible Regular Expressions (PCRE). The former is deprecated, and replaced with the latter, because PCRE is much more popular and powerful. This paper will focus on PCRE features modified from Perl 5.6.1 and used in SAS version 9.

Many users are unaware that the PCRE functionality in DATA step can be made available in the SAS macro language. This is done thought the legendary %SYSFUNC function, and %SYSCALL statement in the macro language. The goal of this paper is to show you how to put regexes to use within the context of SAS macro programming.

2. PERL COMPATIBLE REGULAR EXPRESSIONS IN SAS

There are three main uses for PCRE: matching, which can also be used to validate an input string, or extract matching substrings from the input string; substituting a new string for matching substrings; and splitting a string at each place the regex matches. Data steps in SAS 9 have built-in functions and CALL routines to support these behaviors. Their names start with PRX, such as PRXMatch (), which does a Perl match, and CALL PRXChange, which replaces matching substrings with a new string. For the comprehensive information about these PRX functions and CALL routines, please see the latest SAS documentation.

A regular expression, when used in PRX functions and CALL routines, follows the Perl format, which means that the regular expression must be embedded in one of two Perl operators: match operator (m//) and substitution operator (s///).

The match operator (m//) is used to locate substrings in an input that match a pattern. It has following format:

```
[m]<delimiter><regex><delimiter> [<modifier>]
```

The delimiters within the match operator can be changed by specifying another character after the initial `m`. Most of printable character except whitespaces can be used as delimiters. If you use single quote (`) as the delimiter, then, within SAS macro context, the quoted regular expressions, when containing macro metacharacters, ampersand (&) or percent sign (%), will not be interpolated by SAS macro compiler before they are parsed by function PRXParse (). Balanced cheaters, such as < >, ( ), [ ], { } can be used as delimiters to contain the regular expression. For example,

```
m/Hello/
m`%Heloo`
m<Hello>
```

If you are content with using / / as delimiters, then the initial m can be omitted from the match operator.

The substitution operator (s///) is used to change strings. It has following format:

```
s<delimiter><regex><delimiter><replacement><delimiter> [<modifier>]
```

Any printable character except whitespaces, and balanced characters, can be chosen as the delimiter for a substitution operator. Please note the initial `s` can not be omitted in the substitution operator.

The modifiers at the end of two Perl operators are optional; they can be used to globally modify the interpretation, or behavior of the Perl operators. For instance, the Perl operator `m/cat/` only matches the word `cat`, but `m/cat/i` matches multiple words such as `cat, Cat, or CAT,` because the modifier *i* means ignorance of the case during the matching process.

The table below summarizes the common modifiers supported in PCRE.

Modifier	Meaning
i	Turn off case sensitivity.
s	Treat string as a single line. Allow `.` to match a newline character.
m	Treat a string as multiple lines. Allow '^' and '$' to match next to newline characters.
x	Permit comments in a regular expression and ignore whitespaces.

Besides, you can use the combination of the modifiers within a Perl operator. For example, `m/regex/is`.

2.1 TWO MACROS AS TEST HARNESSES FOR PCRE

Before we explore how to use Perl compatible regular expressions in SAS macro language, let us create two inline macro functions as test harnesses, so that you can test a Perl regular expression, either for matching or replacement, simply with a single-line macro call.

The first inline macro is %PRXMATCH defined as follows:

```
%Macro PRXMATCH(Regex, Source, Action);
    %local RegexID RtnVal;
    %let RegexID=%sysfunc(PRXPARSE(&Regex));
    %let RtnVal=0;
    %if RegexID > 0 %then %do;
        %if %length(&Action)=0 %then %let Action=P;
        %let Action=%upcase(&Action);
        %if &Action=P %then %do;
            %let rtnval=%sysfunc(PRXMatch(&RegexID, &Source));
        %end; %else %if &Action=E %then %do;
            %local Pos Len;
            %let Pos=0;
            %let Len=0;
            %syscall PRXSUBSTR (RegexID, Source, Pos, Len);
            %let RtnVal=;
            %if &Pos > 0 %then %do;
                %let RtnVal=%substr(&Source, &Pos, &Len);
```

```
            %end;
            %syscall PRXFREE(RegexID);
        %end; %else %do;
            %let RtnVal=;
            %put Unknown action: &Action;
        %end;
    %end; %else %do;
        %put Errors found in the pattern: &Regex;
    %end;
&RtnVal
%Mend;
```

The %PRXMatch () macro function takes a Perl matching operator (m//), an input string, and an optional action. If action is 'P', it returns the position (>=1) where the first match for the regex pattern was found in the string; or returns 0 if no match is found. If action is `E', it extracts the first matched substring; if no match, it return blanks. For example:

```
%PRXMatch(/He/, Hello); /*Returns 1, meaning matched     */
%PRXMatch(/He/, world); /*Returns 0, meaning no matched  */
%PRXMatch(/He/, Hello, E); /*Returns He.                 */
```

The second inline macro is %PRXChange defined as follows:

```
%Macro PRXCHANGE(Regex, Source);
    %local RegexID RtnVal;
    %let RegexID=%sysfunc(PRXPARSE(&Regex));
    %if RegexID > 0 %then %do;
        %let RtnVal=%sysfunc(PRXCHANGE(&RegexID,-1, &Source));
        %syscall PRXFREE(RegexID);
    %end; %else %do;
        %put Errors found in the pattern: &Regex;
    %end;
&RtnVal
%Mend;
```

The %PRXChange () macro function takes a Perl substitution operator (s///), and an input string as parameters. It returns the fully replaced string if any matches for the regex pattern were found in the string. If not, it returns the original string. For example:

```
%PRXChange(s/or/ou/, world); /*Returns would               */
%PRXChange(s/ar/ou/, world); /*No match, Returns world     */
```

Since PCRE's regular expression syntax can become very complex, a full discussion of all its details would exceed the scope of this paper. In next sections, I will cover just the basics, which should be enough to be very useful in routine SAS macro programming.

2.2 REGULAR EXPRESSION BASICS

A regular expression, or *regex*, is a particular sequence of characters that acts as syntactic shorthand for the expression over a group of text strings. A given regex is said to match a particular string if the string is within that group. A regular expression thus defines the criteria, or pattern that can be applied to any string – it can match, or not match the string. In fact, you can think of regular expressions as a sort of domain-specific language for text manipulation. They are so useful that they are often embedded within the other programming languages as an advanced text manipulation tool for such operations as, validating the input string, finding strings that match a pattern, or replacing matched strings with other strings. In this paper, I only describe the key constructs (metacharacters and their combinations) that are commonly used in the Perl regular expressions for SAS macro programming.

Literals, Metacharacters, and Metasequences

There are two types of characters that you may use in regular expressions: *Literals* and *Metacharacters*. Most of ordinary characters are literal characters, like '1', 'a', or 'X'. They are the simplest regular expressions that just match themselves. For example, an "a" matches an "a", "dog" matches "dog", "12345" matches "12345" and so on. Some characters like '.', '\', or '*', are metacharacters, which either stand for sets of ordinary characters, or affect how the regular expressions around them are interpreted. For example, `.` is a wildcard character meaning match any single character; `\` is an escape character that can either escape special characters, or signal a special regex sequence. A *metasequence*, like a metacharacters, has special meaning in a regular expression, but it is made up of more than one character. The following two tables provide details about common metacharacters and metasequences.

Table 2.2.1 Common Metacharacters

Metacharacter	Meaning
. (dot)	Matches any single character. A dot matches a newline character (\n) only if the s (dot all) modifier is set.
\ (backslash)	Defines metasequences or undoes the special meaning of metacharacters
* (star)	Matches the previous item repeated zero or more times. For more information, see Quantifiers.
+ (plus)	Matches the previous item repeated one or more times. For more information, see Quantifiers.
? (question mark)	Matches the previous item repeated zero or one time. For more information, see Quantifiers.
(and)	Defines groups within a regular expression. For more information, see Grouping and Subpattern.
[and]	Defines a character class, which defines possible matches for a single character. For more information, see Character Classes.
| (pipe)	Used for alternation, to match the part either on the left side or on the right side.
^ (caret)	Matches at the start of the string. With the `m` (multiline) modifier set, the caret matches the start of a line as well. Note that when used at the start of a character class, the caret indicates negation, not the start of a string. For more information, see Character classes.
$ (dollar sign)	Matches at the end of the string. With the `m` (multiline) modifier set, $ matches the position before a newline character (\n) as well.

Table 2.2.2 Common Metasequences

Metasequence	Description
\. \\ \(\)	Period character, backslash character, and left and right parenthesis character.
\n \r \t \f \v	Newline character, returns character, tab character, form feed character, and vertical feed character.
\b \B	Word anchors; however, if \b is inside a character class, it represents the backspace character. For more information, see Anchors.
nnn	Matches any octal byte, where *nnn* represents the octal number that specifies a character.
\x*nn*	Matches any hexadecimal byte, where *nn* represents the hexadecimal number that specifies a character.
\c*n*	Any control character, where *n* is the character, for example, \cY for Ctrl+Y.
\d \D \w \W \s \S	Predefined character classes. For more information, see Character Classes.
[:*class*:]	Any character class defined in IEEE POSIX 1003.2, such as [:ascii:], or [:lower:]. For

	more information, see Character Classes.

Character Classes

A character class provides you a way to specify a set of acceptable characters you are searching for at a certain character position. It likes a "*variable*" in a regular expression. You can build a character class by enclosing the acceptable characters in a pair of square bracket ([ ]), meaning "match any one of these characters". Characters can be listed individually, or a range of characters can be indicated by giving two characters and separating them by a '-'. Within the square bracket, most of special characters and their special forms lose their meanings and only the rules described here are valid. For example, +, *, (,), and so on, are treated as literals inside [], and backreferences (discussed later) are not active inside []; [xyz$] will match any of the characters 'x', 'y', 'z', or '$'; [a-z] will match any lowercase letter, and [a-zA-Z0-9] matches any letter or digit. If you want to include a ']' or a '-' inside a set, precede it with a backslash, or place it as the first character. For example, the pattern []] will match ']'.

You can match the characters not within a range by complementing the set. This is indicated by inserting a '^' as the first character of the set; '^' elsewhere will simply match the '^' character. For example, [^3] will match any character except '3', and [3^] will match characters `3`, or `^'.

Here are a couple of examples using character classes:

```
%PRXMatch(/c[aeiou]t/, I cut my hand)     /*  returns 3 (match)      */
%PRXMatch(/c[aeiou]t/, What cart?)        /*  returns 0 (not match)  */

%PRXMatch(/c[^aeiou]t/, I cut my hand)    /*  returns 0 (not match)  */
%PRXMatch(/c[^aeiou]t/, chthonic)         /*  returns 1 (match)      */

%PRXMatch(/[0-9]%/, 35% completion)       /*  returns 2 (match)      */
%PRXMatch(/[^0-9]%/, 35% completion)      /*  returns 0 (not match)  */
```

PCRE offers a number of *predefined character classes*, which are convenient shorthands for commonly used regular expressions:

Table 2.2.3 Predefined Character Classes

Construct	Matching
. (dot)	Any character (may or may not match line terminators)
\d	A digit. Same as [0-9]
\D	A non-digit. Same as [^0-9]
\s	A whitespace character. Same as [\t\n\f\r] or [\x20\t\n\f\r]
\S	A non-whitespace character. Same as [^\s]
\w	A word character. Same as [a-zA-Z_0-9]
\W	A non-word character. Same as [^\w]

In the above table, each construct in the left-hand column is shorthand for the character class in the right-hand column. For example, `\d` means a range of digits (0-9), and `\w` means a word character (any lowercase or uppercase letter, the underscore character (_), or any digit). The predefined character classes are also acceptable inside a user-defined character set. For example, [^\w] defines a non-word character.

PCRE also supports a set of *POSIX character classes*, which act like predefined character classes, but they are taken from the IEEE POSIX 1003.2 specification for regular expressions. Following table shows the common POSIX character classes.

Table 2.2.4 POSIX Character Classes

Construct	Matching
[:ascii:]	7-bit ASCII. Same as [\x01-\x7F]
[:alpha:]	Alpha characters (letters). Same as [a-zA-Z]

[:lower:]	Lowercase letters. Same as [a-z]
[:upper:]	Uppercase letters. Same as [A-Z]
[:digit:]	Decimal digits. Same as [0-9]
[:alnum:]	Alphanumeric characters. Same as [0-9a-zA-Z]
[:blank:]	SPACE and TAB characters only. Same as [\x20\t]
[:cntrl:]	Control characters. Same as [\x01-\x1F]
[:space:]	White spaces (newline, carriage return, tab, space, vertical tab). Same as \s
[:punct:]	Punctuation symbols, such as period (.) and the semicolon (;). Same as [-. , " ' ? ! ; : # $ % & ( ) * + / < > = @ [ ] \ ^ _ { } | ~]
[:graph:]	Characters that use ink to print (non-space, non-control). Same as [^\x01-\x20]
[:print:]	Printable characters (graph class plus space and tab). Same as [\t\x20-\xFF]
[:word:]	Word characters. Same as \w
[:xdigit:]	Hexadecimal digits. Same as [0-9a-fA-F]

To negate the POSIX class, write it as follows: [:^class:], for instance [:^punct:]. To use the POSIX character classes, you must enclose them within another character class:

```
%PRXMatch(/[[:punct:]]/,  xyz@company.com, E) /* returns '@' */
```

Please use the predefined classes whenever possible, because they make the regex easier to read and eliminate errors introduced by malformed character classes.

Grouping and Subpattern

You can use parentheses ( ) to group pieces of a regular expression together and treat them as a single unit called a *subpattern*. If X is a regular expression, then (X) is also a regular expression, matching whatever regular expression is inside the parentheses. The left parenthesis indicates the start of a group while the right parenthesis indicates the end of a group. The contents of a group can be retrieved by PRX functions and CALL routines after a match has been performed, and can be matched later in the string with the *\number* special sequence, described later. To match the literals ` (` or ')', use \ (or \), or enclose them inside a character class: [( ] [ )]. If a single literal '(' or ')' appears in a regex, you may have to mask it with %NRSTR compile-time macro function in order for it to be treated correctly.

Line and Word Anchors

Anchors, or zero-width assertions, specify a position in the string where a match must occur. When you use an anchor in a regex, the PCRE engine does not advance through the string or consume any characters; it looks for a match in the specified position only. For example, ^ specifies that the match must start at the beginning of a line or string. Therefore, the regular expression ^Hello matches "Hello" only when it occurs at the beginning of a line. The following table lists the common anchors supported by PCRE.

Table 2.2.5 Line and Word anchors

Construct	Meaning
^ (caret)	It causes the match to happen at the beginning of the string or line.
$ (dollar)	It causes the match to happen at the end of the string or line.
\b	It causes the match to happen at the beginning and/or end of a word, thus `\bside\b` will find `side` but not `sides`, but `\bside` will find `sides`,
\B	It causes the match not to happen at the beginning and/or end of a word, thus `\Bside\B` will find `besides` but not `sides`, but `side\B` will find both `besides` and `sides`.

Here are some examples.

```
%PRXMatch(/\bFOOL/,  xyz@FOOL.com) /* returns 5 (match)       */
%PRXMatch(/\BFOOL/,  xyz@FOOL.com) /* returns 0, (not match)  */
```

Concatenation and Alternation

Regular expressions can be concatenated to form new regular expressions: usually, if A and B is both regular expressions, then AB can also be a regular expression. Thus, complex expressions can easily be constructed from simpler expressions. Besides, you can use the vertical bar (|) character to match any one of a series of patterns, where the | character separates each pattern. This is called *alternation*. For example, A|B is a regular expression; to match, a string must match either of the two regexes A, and B. Here are two examples that specify alternatives in a regular expression:

```
%PRXMatch(/cat|dog/, The cat rubbed my boots) /* returns 4 (match) */
%PRXMatch(/cat|dog/, The rabbit rubbed my boots) /* returns 0 (no match) */
```

The precedence of alternation operation can be a little bit of confusing: '^cat|dog$' selects from '^cat' and 'dog$', meaning that it matches a line that either starts with "cat" or ends with "dog". If you want a line that contains just "cat" or "dog", you need to use the regex '^(cat|dog)$'. You can also combine character classes with alternation, for example, check for strings that don't start with a capital letter:

```
%PRXMatch(/^([a-z]|[0-9])/, The quick brown fox) /* return 0 (not match) */
%PRXMatch(/^([a-z]|[0-9])/, jumped over)         /* return 1 (match)     */
```

Repeating Sequences and Quantifiers

To specify a repeating pattern, you use something called a *quantifier*. Quantifiers specify how many instances of a character, character class, or subpattern must be present in the input string for a match to be found. The following table lists the quantifiers supported by PCRE.

Table 2.2.6 Regular Expression Quantifiers

Greedy Quantifier	Non-greedy Quantifier	Meaning
?	??	Match 0 or 1 time
*	*?	Match 0 or more times
+	+?	Match 1 or more times
{n}	{n}?	Match Exactly n times
{n,}	{n,}?	Match at least n times. Note: when you use this quantifier in the regex, you must mask the comma (,) with %NRSTR if the comma is not enclosed in the subpatterns.
{n,m}	{n,m}?	Match at least n, no more than m times. Note: same as above. When you use this quantifier in the regex, you must mask the comma (,) with %NRSTR if the comma is not enclosed in the subpatterns.

The quantities *n* and *m* are integer constants. Ordinarily, quantifiers are greedy; they cause the PCRE engine to match as many occurrences of a particular pattern as possible. Appending the? Character to a quantifier makes it non-greedy, or lazy; because the character causes the PCRE engine to match as few occurrences as possible.

To repeat a single character, simply put the quantifier after the character:

```
%PRXMatch(/ca+t/, caaaaaaat)   /* returns 1  (true)   */
%PRXMatch(/ca+t/, ct)          /* returns 0 (false)   */
%PRXMatch(/ca?t/, caaaaaaat);  /* returns 0 (false)   */
%PRXMatch(/ca*t/, ct)          /* returns 1 (true)    */
```

With quantifiers and character classes, we can actually do something useful, like checking valid U.S. telephone numbers:

```
%PRXMatch(/[0-9]{3}-[0-9]{3}-[0-9]{4}/, 303-555-1212)   /* returns 1 */
%PRXMatch(/[0-9]{3}-[0-9]{3}-[0-9]{4}/, 64-9-555-1234); /* returns 0 */
```

The quantifiers without ? are always greedy. That is, when faced with a greedy quantifier, the PCRE engine matches as much as it can while still satisfying the rest of the pattern. For instance:

```
%PRXMatch(/<.*>/, Do <b>not</b> click, E) /*  returns <b>not</b> */
```

The regular expression matches from the first less-than sign to the last greater-than sign. In effect, the.* matches everything after the first less-than sign, and the engine backtracks to make it match less and less until finally there's a greater-than sign to be matched. This greediness can be a problem. Sometimes you need minimal (non-greedy) matching— that is, quantifiers that match as few times as possible to satisfy the rest of the pattern. Here's how to match a tag using a non-greedy quantifier:

```
%PRXMatch(/<.*?>/, Do <b>not</b> click, E) /*  returns <b> */
```

Another, faster way is to use a character class to match every non-greater-than character up to the next greater-than sign:

```
%PRXMatch(/<[^>]*>/, Do <b>not</b> click, E) /* returns <b> */
```

Capture Buffer and Backreferences

The parenthesis in regular expressions, in addition to grouping and subpattern mentioned before, also have a good side effect, that is, patterns matched within parenthesis are stored, and can be used later either in the regular expression, or the replacement part of the substitution operator (s///). This storage of matched patterns is called *capturing*, and referring to the capture buffers are *backreferences*.

Each set of capturing parenthesis encountered takes the portion of the target string matched by the pattern and stores it in a *capture buffer*. The capture buffers are numbered 1, 2, and 3, and so on up to the number of parenthesis in the entire pattern match. If you nest subpatterns, the first begins with the first opening parenthesis; the second begins with the second opening parenthesis, and so on. During the match, any captured values are available by referring to the proper capture buffer with *n*. This allows you to refer to something previously matched later in the pattern. For example,

```
%PRXMatch(/([[:alpha:]]+)\s+\1/, City in the the spring)  /*  returns 9  */
```

In the preceding example, ([[:alpha:]]+) captures word `the` into the first capture buffer, and \1 looks for whatever word was stored there after the whitespace character.

If you use the substitution operation (s///), the captured value will appear in the variables named $1, $2, $3, and so on up to the number of parenthesis captured in the substitution part of the operator. For example:

```
%PRXChange(s/([[:alpha:]]+)\s+\1/$1/, City in the the spring)
```

In this example, the backreference \1 in the Perl substitution operator is used to find the repeated word `the` as shown previously. During the substitution, $1 is used to put back just one instance of the repeated word 'the'. Once the match is found, the phase `City in the spring` will be returned by %PRXChange.

Lookahead and Lookbehind

Lookahead and lookbehind or lookaround constructs let you look either ahead or behind the current location in the string for a specified expression. If the expression is found, PCRE engine attempts to match a given pattern. Table 2.2.7 shows the four lookaround expressions: lookahead, negative lookahead, lookbehind, and negative lookbehind.

Table 2.2.7 Lookahead and Lookbehind

Construct	Meaning
(?=regex)	Look ahead from current position and test if regex can be found
(?!regex)	Look ahead from current position and test if regex can not be found
(?<=regex)	Look behind from current position and test if regex can be found.
(?<!regex)	Look behind from current position and test if regex can not be found.

Lookaround constructs do not change the current parsing location in the input string. They are more of a condition that must be satisfied for a match to occur. For example, the following command uses an expression that matches numeric characters (\d+) that meet the condition that they look ahead to (i.e., are immediately followed by) the letters `visit`. The resulting match includes only that part of the string that matches the \d+; it does not include those characters that match the lookahead expression `(?=visit)`:

```
%PRXMatch(/\d+(?=visit)/i, There are 3visits, E)
/* returns '3', instead of '3visits' */
```

If you repeat this command and match one character beyond the lookahead expression, you can see that parsing of the input string resumes at the letter v, thus demonstrating that matching the lookahead construct has not consumed any characters in the string:

```
%PRXMatch(/\d+(?=visit)\w/i, There are 3visits, E)   /* returns '3v'  */
```

PCRE engine limits lookbehind to constant-width expressions. That is, the expressions cannot contain quantifiers, and if you use alternation, all the choices must be the same length. The Perl-compatible regular expression engine also forbids quantifiers in lookbehind, but does permit alternatives of different lengths. For more examples of using lookaround constructs, please see the Section 4.

3. USING PERL REGULAR EXPRESSIONS IN MACRO LANGAUGE

Although almost all PRX functions and call routines in DATA step can be invoked by macro programs via %SYSFUNC and %SYSCALL, you have to pay attention to following differences.

1. In DATA steps, you can directly use a Perl operator with the function PRXMATCH, and PRXCHANGE; but, in macro, you are required to use PRXPARSE function to compile the Perl operator and create the regex ID first and then use the regex ID with other PRX functions, or CALL routines. In order to simplify the macro programming, you can use two macros %PRXMATCH and %PRXCHANGE defined in Section 2 to perform common text manipulation with Perl regular expressions in macro statements.
2. You do not have to quote the Perl operator in PRXPARSE() with a pair of single quotes or double quotes like you do in DATA steps. However, if the Perl operator contains special characters such as comma (,), ampersand (&), and ampersign (%), you may have to either use macro function %NRSTR to make them, or use special delimiters such as single quote (`) for the Perl operator.
3. When %SYSCALL invokes a PRX call routine, the value of each macro variable as argument is retrieved and passed to the call routine. Upon completion of the call routine, the value for each argument is written back to the respective macro variable. Since the arguments in the call routine invoked by the %SYSCALL macro are resolved before execution. Therefore, all the arguments must be either the existing macro variable names (without the ampersand &), or the macro expressions that can be resolved to existing macro variable names.
4. If you want the results from PRX functions and CALL routines to be masked, you have to use %QSYSFUNC or %QSYSCALL instead.
5. Although a macro variable can hold as many as 64K characters, a PRX function or call routine used in a macro statement can only handle first 32K characters in the macro variables, or expressions because of the limitation of DATA step functions.
6. It is strongly suggested that you invoke the %SYSCALL PRXFREE(*RegexID*)) statement to free the unneeded memory that was allocated for a pattern once a regex operation is finished.

Table 3.1 below gives the list of PRX functions that can be used in the macro statements via the %SYSFUNC, or %QSYSFUNC function.

Table 3.1 PRX Functions that can be Invoked from the Macro Language

Function	Description and Syntax
PRXPARSE	Compiles a Perl operator at runtime, either matching operator or substitution operator, or returns a numeric identifier. `%let RegexID=%SYSFUNC (PRXPARSE(PerlOperator));`
PRXMATCH	Returns the numeric position in source at which the pattern begins; or a zero if no match is found. `%let Pos=%SYSFUNC(PRXMATCH(RegexID, Source));`
PRXCHANGE	Performs a pattern-matching replacement operation. It returns the value in source with the changes that were specified by the pattern. If there is no match, returns the unchanged value in source. The parameter Times can be used to control the number of replacements. If the value of *Times* is -1, the replacement is performed as many times as possible. `%letNewVal=%SYSFUNC(PRXCHANGE(RegexID,Times, Source));`
PRXPAREN	Returns the last bracket match for which there is a match in a pattern. `%let NewVal=%SYSFUNC(PRXPAREN (RegexID));`

PRXPOSN	Returns the value for a capture buffer. `%let NewVal=%SYSFUNC(PRXPOSN(RegexID,CaptureBuffer,Source));`

Table 3.2 below gives the list of PRX call routines that can be used in the %SYSCALL, or %QSYSCALL statement.

Table 3.2 PRX CALL Routines that can be Used in the Macro Language

CALL Routine	Description and Syntax
PRXSUBSTR	Returns the position and length of a substring that matches a pattern. `%SYSCALL PRXSUBSTR (RegexID, Source,Position <,Length>);`
PRXCHANGE	Matches and replaces a pattern. If the value of *Times* is -1, the replacement is performed as many times as possible. `%SYSCALL PRXCHANGE (RegexID, Times, Source <, NewVal <, NewValLength <, Truncation <, NumberOfChanges >>>>);`
PRXPOSN	Returns the start position and length for a capture buffer. `%SYSCALL PRXPOSN (RegexID, CaptureBuffer, Start <, Length>);`
PRXNEXT	Returns the position and length of a substring that matches a pattern and iterates over multiple matches within one string. `%SYSCALL PRXNEXT (RegexID, Start, Stop, Source, position, Length);`
PRXFREE	Frees unneeded memory that was allocated for a pattern. `%SYSCALL PRXFREE (RegexID);`

For detailed information about these functions, please see the latest SAS 9 documentation.

4. A FEW PRACTICAL EXAMPLES

Here are a few SAS inline macro functions that use regular expressions to simplify common string manipulations in macros, which you could directly use or modify for your own macro programming tasks.

Example 1: Validate a SAS name. This macro function validates whether a string confirms to the following SAS naming convention for a dataset, variable, and library.

- SAS dataset or variable names can be 32 characters long; but library names only can have a maximum length of 8.
- The first character of SAS library, dataset or variable names must be an alpha letter, or underscore (_). Subsequent characters can be any word letters.
- The SAS names are case-insensitive.

By using the regular expression functionality, the macro called %IsSASName can be simply implemented as follows:

```
%Macro IsSASName(s,length);
    %if %length(&length)=0 %then %let length=32;
    %PRXMatch(%nrbquote(/^[A-Za-z_]\w{0,%eval(&length-1)}$/), &s)
%Mend;
```

You can call this macro like

```
%IsSASName(_ABC12)     /* Returns 1.  __ABC12 is a 32-char SAS  name   */
%IsSASName(#ABC12)     /* Returns 0. #ABC12 is  not a SAS name         */
%IsSASName(ABC12LIB,8) /* Returns 1. ABC12LIB is a 8-char  SAS name    */
%IsSASName(ABC123LIB,8)/* Returns 0.ABC123LIB is not a 8-char SAS name   */
```

Explanation:

1. %IsSASName returns 1 if the input string confirms the SAS naming conversion above, otherwise 0;

2. By default, the macro check whether a string is a variable, or dataset name, which can be 32 characters long. If you set the second argument to 8, it will check whether the string is a library name, with only having a maximum length of 8.
3. The macro invokes the macro %PRXMatch defined early in the Section 2.
4. Since the regular expression used in this macro contains comma (,), %nrbquote has to be used to mask it when calling %PRXMatch.
5. You can use the macro variables or expressions within the regular expression if it won't confuse you too much.

Example 2: Validate a macro parameter to see if it contains one of following mutual exclusive optional values (case-insensitive): Yes, No, and Unknown.

```
%Macro ChkOPT(option);
    %PRXMatch(/^(YES|NO|UNKNOWN)$/i, &option)
%Mend;
```

You can call this macro like

```
%ChkOpt(Yes)     /* Returns 1.    */
%ChkOpt(UNK)     /* Returns 0.    */
```

Example 3: Here is a little complex case of validating a macro parameter to see if it contains one, two, or more of following specified terms: Screen, Visit1, and Visit2. Visit3 and Followup.

```
%Macro ChkOPTS(options);
    %local s found;
    %let found=0;
    %if %length(&options) %then %do;
       %let s=%PRXChange(s/\b(Screen|Visit1|Visit2|Visit3|Followup)\B//i,
              &options);
     %if %length(&s)=0  %then %let found=1;
    %end;
&found
%Mend;
```

You can call this macro like

```
%ChkOpts()                    /* Returns 0.    */
%ChkOpts(Screen)              /* Returns 1.    */
%ChkOpts(Screen Visit1)       /* Returns 1.    */
%ChkOpts(Screen Visit1 Visit5) /* Returns 0.    */
```

Example 4: Parse a composite parameter in a macro. This example decomposes a composite parameter in form LIB.DSN[*var1 var2* …] of a macro into three parts: library name, dataset name, and a list of variables. If LIB part is missing from the expression, WORK will be used as the default library name.

```
%Macro ParseComposite(Param);
    %local RegexID found;
    %local lib dsn vlst valid;
    %let RegexID=%sysfunc(prxparse(/
                                  ((\w+)\.)?
                                  (\w+)
                                  \[
                                  \s*(|\w+(\s+\w+)*)\s*
                                  \]
```

```
                                    /x));
    %if &RegexID > 0 %then %do;
       %let found=%sysfunc(prxmatch(&RegexID, &param));
       %if &found %then %do;

                  %let lib=%sysfunc(prxposn(&RegexID,2, &param));
                  %let dsn=%sysfunc(prxposn(&RegexID,3, &param));
                  %let vlst=%sysfunc(prxposn(&RegexID,4, &param));
                  %if %length(&lib) = 0 %then %let lib=WORK;

                  %let valid=0;
                  %if %IsSASName(&lib,8) AND %IsSASName(&dsn)
                 %then %let valid=1;
            %put lib=&lib dsn=&dsn vlst=&vlst valid=&valid;

       %end;
       %syscall PRXFREE(RegexID);
    %end;
%Mend;
```

You can call this macro like

```
%ParseComposite(DB.AE[x1 x2])   /* lib=DB dsn=AE vlst=x1 x2  valid=1     */
%ParseComposite(2DB.AE[])       /* lib=DB dsn=AE vlst=     valid=0       */
%ParseComposite(DB.AE[   x1 ])  /* lib=DB dsn=AE vlst=  X1 valid=1       */
%ParseComposite(AE[x1 x2])      /* lib=WORK dsn=AE vlst=x1 x2 valid=1    */
```

Explanation:

1. The regular expression used in this macro look a little complex; I explain the meanings of its individual parts in the following table.

Sequence	Regex Component	Meanings
1	^	Matches the beginning of the string
2	((\w+)\.)?	Matches data library name
3	(\w+)	Matches dataset name
4	\[	Matches left square bracket '['
5	\s*	Matches whitespaces if any
6	(\w+(\s+\w+)*)*	Matches a list of variable names
7	\s*	Matches whitespaces if any
8	\]	Matches right square bracket ']'
9	$	Matches the end of the string

2. This macro uses %IsSASName to verify whether the decomposed parts, &LIB and &DSN confirm the SAS naming convention.

Example 5: Parse a list parameter in a macro. This example decomposes a component list as a macro parameter into individual components, each of which has the form LIB.DSN [*var1 var2* …], which will be further decomposed into three parts: library name, dataset name, and a variable list.

```
%Macro ParseCompositeLST(ParamLST);
    %local RegexID found;
    %local lib dsn vlst valid seqno;
    %local start stop length position;
    %local PLST;

    %let RegexID=%sysfunc(prxparse(/
                                  ((\w+)\.)?
                                  (\w+)
                                  \[
                                  \s*(|\w+(\s+\w+)*)\s*
                                  \]
                                  /x));
    %if &RegexID > 0 %then %do;

        %let PLST=&ParamLST;
          %let start=1;
        %let stop=%length(&Plst);
        %let length=0;
          %let position=0;
          %let seqno=0;

        %syscall prxnext(RegexID, start, stop,
                         PLST, position, length);
        %do %while(&position >0);
                  %let seqno=%eval(&seqno + 1);
                  %let lib=%sysfunc(prxposn(&RegexID,2, &plst));
                  %let dsn=%sysfunc(prxposn(&RegexID,3, &plst));
                  %let vlst=%sysfunc(prxposn(&RegexID,4, &plst));
                  %if %length(&lib) = 0 %then %let lib=WORK;

                  %let valid=0;
                  %if %IsSASName(&lib,8) AND %IsSASName(&dsn)
                 %then %let valid=1;
            %put seqno=&seqno lib=&lib dsn=&dsn
                 vlst=&vlst valid=&valid;

            %syscall prxnext(RegexID, start, stop,
                             PLST, position, length);
        %end;
        %syscall PRXFREE(RegexID);
    %end;
%Mend;
```

You call the macro as follows

```
% ParseCompositeLST(xxx lib1.dsn11[] lib2.dsn21[x1] lib3.dsn31[x1 x2] uvw)
```

The output in the Log window will be

```
seqno=1 lib=lib1 dsn=dsn11  vlst= valid=1
seqno=2 lib=lib2 dsn=dsn21  vlst=x1 valid=1
seqno=3 lib=lib3 dsn=dsn31  vlst=x1 x2 valid=1
```

Example 6: Split a list of compressed titles separated by tilde (~) into individual titles. If there is a backslash (\) right before a tilde, the tilde will be treated as a normal character instead of a separator.

```
%Macro split(titles);

    %local RegexID found;
    %local start stop length position;

    %local title count;

    %if %length(&titles)= 0 %then %return;

    %let RegexID=%sysfunc(prxparse(/(?<!\\)~/));
    %if &RegexID > 0 %then %do;

        %let s=&Titles;
        %let start=1;
        %let stop=%length(&s);
        %let length=0;
        %let position=0;
        %let count=0;
        %let pre_position=&start;

       %syscall prxnext(RegexID, start, stop,
                        s, position, length);
       %do %while(&position>0);

            %let title=;
            %if &position > &pre_position
                %then %let title=%substr(&s,&pre_position,
                                &position - &pre_position);

            %let count=%eval(&count + 1);
            %let title=%PRXChange(s/\\~/~/, &title);
            %put title=&title count=&count;

            %let pre_position=&start;
            %syscall prxnext(RegexID, start, stop,
                            s, position, length);
```

```
        %end;

        %if &pre_position <%length(&s)
            %then %let title=%substr(&s,&pre_position);
            %else %let title=;

        %let count=%eval(&count + 1);
        %let title=%PRXChange(s/\\~/~/, &title);
        %put title=&title count=&count;

        %syscall PRXFREE(RegexID);
    %end;
%Mend;
```

You call the macro like

```
%split(Table\~title1~Title2~Title3);
```

The output in the Log window will be

```
title=Table~title1 count=1
title=Title2 count=2
title=Title3 count=3
```

CONCLUSION

Regular expressions provide a very powerful way to describe text patterns that can be used to manipulate varying strings. When combined with SAS PRX functions and CALL routines via %SYSFUNC and %SYSCALL, They can be used to simplify and tackle many complex string manipulation tasks in macro language that are hard to do without dozens line of code, or impossible to do previously. This paper describes the way to use them in the SAS macro language with many examples. Although learning and mastering the regular expressions may take a little bit time, the ultimate reward for a SAS macro developer is an increased ability to develop short and elegant solutions to wide varieties of macro programming problems.

REFERENCE

1. Cassell, David L., “PRX Functions and Call Routines”, *SUGI Proceedings,* 2005
2. SAS® 9.2 Help and Documentation by SAS Institute Inc.
3. Friedl, J.E. “Mastering Regular Expressions”, *O’Reilly & Associates*, 1997.

ACKNOWLEDGMENTS

The author wish to thank Dr. Peter Ouyang for his kind comments and constructive suggestions during the preparation of this paper.

CONTACT INFORMATION

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

Lei Zhang

Celgene Corporation.

86 Morris Avenue

Summit, NJ 07901

Phone: (908) 673-9000

SAS and all other SAS Institute Inc. product or service names are registered trademarks or trademarks of SAS Institute Inc. in the USA and other countries. ® Indicates USA registration.

Other brand and product names are trademarks of their respective companies.