PharmaSUG 2025 – Paper SD-400

Unleash Your Coding Potential: SAS PRX Functions for Next-Level String Manipulations

Edwin Xie, John M. LaBore, SAS Institute Inc.

ABSTRACT

Explore the power of SAS PRX functions to elevate your string manipulation skills and enhance coding efficiency. These essential tools, available in both SAS 9 and SAS Viya, allow advanced SAS programmers to achieve complex data analysis with fewer lines of code and tackle challenges beyond traditional functions. Syntax examples and valuable resources are provided to help kickstart your usage.

INTRODUCTION

The SAS PRX functions are invaluable tools for SAS users who possess a solid understanding of their application. This paper elucidates fundamental concepts associated with these functions and provides illustrative examples, as well as a discussion of readily available resources for programmers eager to enhance their knowledge of SAS PRX functions.

WHAT IS A REGULAR EXPRESSION?

A regular expression is defined as a sequence of characters that establishes a search pattern within a given text. Each character within a regular expression can either be classified as a metacharacter, which has a specific function, or as a regular character, which conveys a literal meaning. The syntax for patterns utilized in the PRX functions closely resembles that of Perl.

To illustrate, consider the following example of a simple regular expression:

```
data _null_;
    text = "A tidy tiger tied a tie tighter.";
    pattern = "/ti(dy|ger)/i";
    pos = prxmatch(pattern, text);
    put pos=;
run;
```

The output generated in the log will be:

```
pos=3
```

In this example, the `PRXMATCH` function, which is one of the PRX functions, conducts a pattern match and returns the position at which the specified pattern is detected, or returns 0 if the pattern is not found. The first argument represents the regular expression pattern, while the second argument indicates the source string to be searched. The character “/” serves as a delimiter. Moreover, both the “|” and “()” characters function as metacharacters; the former denotes an OR condition, while the latter signifies a grouping mechanism. The trailing “i” character functions as a modifier, ensuring that pattern matching is case-insensitive.

METACHARACTERS

A metacharacter is a character that holds a specific significance within a pattern. This section will introduce several fundamental metacharacters.

For instance, if the objective is to match the word "tie," using the pattern “/tie/” would not be sufficient, as it would also match other words such as "tied." A more effective strategy is to employ the metacharacter “\b,” which denotes a word boundary:

```
data _null_;
    text = "A tidy tiger tied a tie tighter.";
    pattern = "/\btie\b/";
    pos = prxmatch(pattern, text);
    put pos=;
run;
```

The output generated in the log will be:

```
pos=21
```

To match any word that begins with "ti," metacharacters can also be utilized effectively:

```
data _null_;
    text = "A tidy tiger tied a tie tighter.";
    pattern = "/\bti[a-z]*/";
    pos = prxmatch(pattern, text);
    put pos=;
run;
```

The output produced in the log will be:

```
pos=3
```

In the above example, the use of “[]” (square brackets) signifies a metacharacter that defines a character set, matching any single character contained within the brackets. Thus, “[a-z]” matches any lowercase alphabetical character from 'a' to 'z.' The character “*” also serves as a metacharacter, acting as a repetition operator that matches the preceding subexpression zero or more times.

In addition to custom character sets, predefined character sets are available. For example, “[[:alpha:]]” matches any alphabetic character, “\w” matches any "word" character (including alphanumeric characters and underscores), “\d” matches a digit character equivalent to [0–9], and the period “.” matches any single character except for a newline.

When defining a character set using “[]”, a preceding “^” can be employed to denote its complement. For instance, [^a-z] matches any character that is not a lowercase letter from 'a' to 'z,' while [[:^alpha:]] matches a non-alphabetic character.

For character sets denoted by \w, \d, \b, and \s, their complements can be specified using corresponding uppercase letters. For example, “\W” matches any character that is not classified as a "word" character.

Both characters and character sets can be utilized alongside repetition operators. The repetition operator “*” matches the preceding subexpression zero or more times, “+” matches one or more times, and “?” matches zero or one time. For instance, the pattern '/go+d/' can match any word that begins with 'g,' contains one or more 'o's, and concludes with 'd.' To specify an exact number of repetitions, one can use {m} or {m,n}. For example, the pattern “/go{1,2}d/” matches “god” and “good,” but not “gd” or “goood.”

Additionally, there exists a category of metacharacters that correspond to specific positions. The “^” metacharacter matches the beginning of a string, while the “$” metacharacter matches the end of a string. The “\b” metacharacter matches a word boundary. For example, the pattern “/^a$/” matches a string containing solely the single word “a.” If any extraneous characters are present in the string, the match will fail:

```
data _null_;
    pos = prxmatch("/^a$/", "ab");
    put pos=;
run;
```

The output in the log will be:

```
pos=0
```

To match a metacharacter itself, it must be escaped to remove its special meaning. This can be accomplished by placing a backslash before the metacharacter. For

example, "()" is a metacharacter used for grouping; to match parentheses, one would escape it:

```
data _null_;
    text = "My phone number is (555)123-4567";
    pattern = "/\(\d{3}\)\s?\d{3}-\d{4}/";
    pos = prxmatch(pattern, text);
    put pos=;
run;
```

The output generated in the log will be:

```
pos=20
```

MODIFIERS

Modifiers are specific letters appended to a pattern that alter how PRX functions and call routines interpret that pattern. SAS PRX functions support five distinct modifiers, as outlined in Table 1 below:

Modifier	Description
i	Enables case-insensitive pattern matching.
o	Compiles the pattern once for optimization.
x	Utilizes extended regular expressions.
m	Treats the string as containing multiple lines.
s	Treats the string as a single line.

Table 1: Supported Modifiers

The modifier “i” specifies case insensitivity, altering the default behavior from case-sensitive to case-insensitive pattern matching. For example, the results of matching the character “a” vary when using the "i" modifier:

```
data _null_;
    text = "A tidy tiger tied a tie tighter.";
    without_i = prxmatch("/a/", text);
    with_i = prxmatch("/a/i", text);
    put without_i= with_i=;
run;
```

The output generated in the log will be:

```
without_i=19 with_i=1
```

The modifier “o” denotes optimization. When a pattern is a string literal, it is compiled only once by default:

```
prxparse("/a/");
```

However, if the pattern is defined as a variable, it will be compiled for every observation:

```
pattern = "/a/";
re = prxparse(pattern);
```

If the pattern remains unchanged during a loop, it is more efficient to compile it only once upon its initial use. This can be achieved by optimizing the code as follows:

```
pattern = "/a/";
if _N_ = 1 then do;
    retain re;
    re = prxparse(pattern);
end;
```

Alternatively, one can utilize the modifier "o" directly after the pattern:

```
pattern = "/a/o";
re = prxparse(pattern);
```

It is essential to note that if the function is invoked from a macro, such as through %SYSFUNC, the pattern will be recompiled with each call, regardless of whether it is a string literal or if '/o' is employed.

The modifier “x” indicates that the pattern is in an extended format. In this context, whitespace that is not backslashed or enclosed within a bracketed character class is ignored. Furthermore, the “#” character serves as a metacharacter introducing a comment that extends up to the closing delimiter of the pattern. This modifier is particularly useful for organizing complex regular expressions across multiple lines:

```
data _null_;
    prx = prxparse("/foo
                    bar #to match foobar, not foo bar
                    /x");
    str = "foo bar foobar";
    pos = prxmatch(prx, str);
    put pos=;
run;
```

The output generated in the log will be:

```
pos=9
```

The modifier “m” modifies the default behavior of the metacharacters “^” and “$.” Without this modifier, “^” matches the beginning of a string, while “$” matches the end of a string. When the “m” modifier is applied, both “^” and “$” match the start and end of each line within a string:

```
%let newline = '0a'x;
data _null_;
    x = "First line" || &newline ||
        "Second line" || &newline;
    without_m = prxmatch("/^Second/", x);
    with_m = prxmatch("/^Second/m", x);
    put without_m= / with_m=;
run;
```

The output produced in the log will be:

```
without_m=0
with_m=12
```

When using the “m” modifier, if the goal is to match the beginning of the string, “\A” may be utilized in place of “^.” To match the end of the entire string, “\Z” or “\z” can be used instead of “$.” The distinction between "\Z" and "\z" is that "\Z" ignores an optional newline at the end:

```
%let newline = '0a'x;
data _null_;
    x = "First line" || &newline ||
        "Second line" || &newline;
    /* Matched at the beginning of the string */
    first = prxmatch("/\AFirst/m", x);
    /* Cannot match the second line */
    second = prxmatch("/\ASecond/m", x);
    /* Matched; the trailing newline is ignored */
    big_z = prxmatch("/line\Z/m", x);
    /* Cannot match because of the last newline */
    small_z = prxmatch("/line\z/m", x);
    put first= / second = / big_z = / small_z =;
run;
```

The output generated in the log will be:

```
first=1
second=0
big_z=19
small_z=0
```

Delimiters

In addition to the commonly used slash, delimiters can be any non-alphanumeric characters. If the delimiter is '<', '{', '(', or '[', it must be paired with its corresponding closing delimiter. For example:

```
data _null_;
    pos = prxmatch("[N/A]", "The data is N/A for this period.");
    put pos=;
run;
```

Because the pattern in this example contains a slash, '[' and ']' are used as the delimiters.

FUNCTIONS AND CALL ROUTINES

Upon acquiring a foundational understanding of how PRX functions operate, programmers are encouraged to explore the extensive resources available for a more comprehensive grasp of both the functions and call routines. The functionalities of these functions and call routines are summarized in Table 2 below:

Functionality	Functions	CALL Routines
Pre-compile a pattern	`PRXPARSE`	
Release the compiled pattern		`CALL PRXFREE`
Pattern match	`PRXMATCH`	`CALL PRXSUBSTR`
		`CALL PRXNEXT`
Match and substitution	`PRXCHANGE`	`CALL PRXCHANGE`
Capture buffer related	`PRXPOSN`	`CALL PRXPOSN`
Debug output toggle		`CALL PRXDEBUG`

Table 2: Functionality of SAS PRX Functions and CALL Routines

To illustrate the application of these PRX functions and call routines within SAS, it is important to note that PRX functions yield either numeric or character results. In contrast, PRX call routines do not return a value; instead, they modify variable values directly. While functions can be utilized in assignment statements or expressions, CALL routines must be invoked using CALL statements.

This distinction allows for flexible programming, enabling users to leverage the capabilities of PRX functions and routines effectively in their SAS applications.

PATTERN COMPILATION AND RELEASE

PRXPARSE(perl-regular-expression)
CALL PRXFREE(regular-expression-id)

The `PRXPARSE` function generates a pattern identifier number that is utilized by other Perl functions and CALL routines to match patterns. If an error occurs during the parsing of the regular expression, the function returns a missing value.

It is important to note that when using `PRXPARSE`, if the input pattern is a constant string, the function compiles the pattern only upon its first invocation and returns the same identifier for subsequent calls within a loop. Consider the following example:

```
%let loop=3;
data _null_;
    do i = 1 to &loop;
        id = prxparse("/hello/");
        put id=;
    end;
run;
```

The output generated in the log will be:

```
id=1
id=1
id=1
```

Conversely, if the pattern is stored in a variable, `PRXPARSE` will compile the pattern each time it is invoked, resulting in unique identifiers for each call:

```
data _null_;
    var = "/hello/";
    do i = 1 to &loop;
        id = prxparse(var);
        put id=;
    end;
run;
```

The output produced in the log will be:

```
id=1
```

```
id=2
id=3
```

To prevent recompilation of the pattern, one can utilize the "o" modifier discussed previously.

At the conclusion of the DATA step processing, the identifiers and associated memory are automatically released. Alternatively, you can manually release the identifier using `CALL PRXFREE(identifier)`. This CALL routine also sets the identifier to a missing value, as demonstrated below:

```
data _null_;
    id = prxparse("/hello/");
    put id=;
    call prxfree(id);
    put id=;
run;
```

The log output for this example will be:

```
id=1
id=.
```

This indicates that the identifier was successfully freed and is now set to a missing value.

MATCHING

The following functions are employed for pattern matching in SAS:

PRXMATCH(regular-expression-id | perl-regular-expression, source)
CALL PRXSUBSTR(regular-expression-id, source, position)
CALL PRXNEXT(regular-expression-id, start, stop, source, position, length)

The `PRXMATCH` function searches for a pattern match within a specified source string and returns the starting position at which the pattern is found. The regular expression pattern can either be a direct pattern or an identifier returned by the `PRXPARSE` function.

The `CALL PRXSUBSTR` routine requires the input pattern to be an identifier. This routine returns the starting position of the match along with the length of the substring that corresponds to the pattern. For example, to extract a phone number from a sentence, the following code can be used:

```
data _null_;
```

```
    text = "Our customer service phone number is 800-727-0025.";
    id = prxparse("/\d{3}-\d{3}-\d{4}/");
    call prxsubstr(id, text, pos, len);
    number = ksubstr(text, pos, len);
    put number=;
run;
```

The output in the log will be:

```
number=800-727-0025
```

Additionally, the same result can be achieved using `PRXMATCH` and `PRXPOSN`. Notably, when the second argument of `PRXPOSN` is set to 0, it returns the entire match:

```
data _null_;
    text = "Our customer service phone number is 800-727-0025.";
    id = prxparse("/\d{3}-\d{3}-\d{4}/");
    pos = prxmatch(id, text);
    number = prxposn(id, 0, text);
    put number=;
run;
```

The output in the log will again be:

```
number=800-727-0025
```

The `CALL PRXNEXT` routine allows for specification of the starting and ending positions of the source string during pattern matching, making it particularly useful for consecutive matches. The starting position is updated by the routine after each successful match. For instance, to match all words that begin with "ti," the following example demonstrates its use:

```
data _null_;
    text = "A tidy tiger tied a tie tighter.";
    id = prxparse("/\bti\w*\b/");
    start = 1;
    stop = -1;
    call prxnext(id, start, stop, text, pos, length);
    do while (pos > 0);
        found = ksubstr(text, pos, length);
        put found= pos= length=;
        call prxnext(id, start, stop, text, pos, length);
    end;
run;
```

The output in the log will be:

```
found=tidy pos=3 length=4
found=tiger pos=8 length=5
found=tied pos=14 length=4
found=tie pos=21 length=3
found=tighter pos=25 length=7
```

This example highlights the ability of `CALL PRXNEXT` to identify multiple matches sequentially, along with their respective positions and lengths within the source string.

SUBSTITUTION

PRXCHANGE(perl-regular-expression | regular-expression-id, times, source)
CALL PRXCHANGE(regular-expression-id, times, old-string <, new-string <, result-length<, truncation-value <, number-of-changes> > > >)

The `PRXCHANGE` function performs both pattern matching and substitution, returning the modified string after the replacement has been applied. The following example demonstrates how to rearrange the first and last names in the input:

```
data _null_;
    input name & $32.;
    name = prxchange('s/(\w+), (\w+)/$2 $1/', -1, name);
    put name=;
    datalines;
    Jones, Fred
    Kavich, Kate
    Turley, Ron
    Dulix, Yolanda
    ;
run;
```

The resulting output in the log will be:

```
name=Fred Jones
name=Kate Kavich
name=Ron Turley
name=Yolanda Dulix
```

In this `PRXCHANGE` example, the first argument “s/(\w+), (\w+)/$2 $1/” defines the substitution pattern. This pattern string begins with 's', indicating a substitution operation. The regular expression part, (\w+), (\w+), and the replacement string part, $2 $1, are separated by the forward slash character, which serves as the delimiter. Within

the replacement string, $1 and $2 are backreferences. They refer to the text captured by the first (\w+) and the second (\w+) capturing groups, respectively, within the regular expression.

In addition to returning the newly formatted string, the `CALL PRXCHANGE` routine can also provide additional information, such as the length of the new string (excluding trailing blanks), a numeric value indicating whether the new string is truncated, and the total number of substitutions that were performed. This functionality enhances the utility of the `CALL PRXCHANGE` routine in various programming scenarios.

CAPTURE BUFFER

PRXPOSN(regular-expression-id, capture-buffer, source)
PRXPAREN(regular-expression-id)
CALL PRXPOSN(regular-expression-id, capture-buffer, start)

The `PRXPOSN` function retrieves the matched string from the specified capture buffer, while the `PRXPAREN` function returns the highest capture buffer number that can be utilized with the `PRXPOSN` function.

Here is an example illustrating the use of these functions:

```
data _null_;
    fullname = "Fred Jones";
    id = prxparse("/(\w+)\s(\w+)/");
    pos = prxmatch(id, fullname);
    paren = prxparen(id);

    if paren = 2 then do;
        first = prxposn(id, 1, fullname);
        last = prxposn(id, 2, fullname);
        put first= last=;
    end;
run;
```

The output generated in the log will be:

```
first=Fred last=Jones
```

If grouping is desired without capturing, non-capturing groups can be employed using the syntax `(?:)`. For instance, the expression `/ti(?:dy|ger)/` matches the same words as `/ti(dy|ger)/`, but it does not create capture groups as denoted by the parentheses.

DEBUGGING

CALL PRXDEBUG(on-off)

The `CALL PRXDEBUG` routine enables or disables the debugging log for regular expressions. This feature allows users to toggle debugging on and off as needed, facilitating a clearer understanding of how patterns are matched and processed.

USING PRX FUNCTIONS IN VIYA

CAS DATA STEP

SAS PRX functions and call routines can be used within a CAS DATA step, but support for native execution on the CAS server has evolved across SAS Viya versions. In SAS Viya 3.5 and Viya 4 releases prior to 2024.05, only a subset of PRX functions were supported for native execution by the CAS server.

These included **PRXPARSE**, **PRXMATCH**, **PRXCHANGE**, and **PRXPOSN**. Notably, in these older versions, **PRXPAREN** and all PRX call routines are not supported by the CAS server. If a DATA step in these releases employed an unsupported PRX function or call routine, that part of the DATA step would execute on the Compute server. Fortunately, in SAS Viya 4 releases from 2024.05 onwards, all PRX functions and call routines are now supported for native execution on the CAS server.

For example, the following program runs on the CAS server and produces the log note: “NOTE: Running DATA step in Cloud Analytic Services”:

```
data _null_/ sessref="MySession";
    /* Note: MySession is an active CAS session */
    id = prxparse('/(magazine)|(book)|(newspaper)/');
    pos = prxmatch(id, 'find book here');
    put pos=;
run;
```

The output in the log will be:

```
NOTE: Running DATA step in Cloud Analytic Services.
pos=6
```

In contrast, in SAS Viya 3.5 and older Viya 4, the program below will not execute because it utilizes **PRXPAREN**, which is unsupported by the CAS server in those versions:

```
data _null_/ sessref="MySession";
    /* Note: MySession is an active CAS session */
    id = prxparse('/(magazine)|(book)|(newspaper)/');
```

```
    pos = prxmatch(id, 'find book here');
    paren = prxparen(id);
    put paren=;
run;
```

If the CAS server does not support the function, the log output will indicate:

```
NOTE: Running DATA step in Cloud Analytic Services.
ERROR: The function PRXPAREN is unknown, or cannot be accessed.
```

If the same program is executed in Viya outside of a CAS server session, it will run successfully on the Compute server. Note that there is no indication in the log that the DATA step is not running in CAS:

```
data _null_;
    id = prxparse('/(magazine)|(book)|(newspaper)/');
    pos = prxmatch(id, 'find book here');
    paren = prxparen(id);
    put paren=;
run;
```

The output in the log will be:

```
paren=2
```

THE CAS PROCEDURE

In the latest SAS Viya 4, all PRX functions and call routines can be invoked directly within a CAS procedure as follows:

```
proc cas;
    id = prxparse('/world/');
    position = prxmatch(id, 'Hello world!');
    print "pos=" position;
    prxfree(id);
run;
```

The output in the log will be:

```
pos=7
```

In the example above, the functions are executed on the Compute server. To execute PRX functions on the CAS server, the program must be submitted to the CAS server as an action (using the action set “sscasl.runcasl” in this instance):

```
proc cas;
    sessref = "MySession";
    source pgm;
    position = prxmatch('/world/', 'Hello world!');
    print 'pos=' position;
    run;
    endsource;
    sccasl.runcasl / code=pgm;
run;
quit;
```

The output in the log will be:

```
pos=7
```

OTHER PROCEDURES

When data is stored on the CAS server, executing PRX functions directly on the server can enhance efficiency. Typically, if a procedure is executed on the server, any PRX functions invoked by that procedure will also run on the server.

For instance, the following example illustrates the use of the DS2 procedure to select all car models with two doors using the **PRXMATCH** function. By specifying **SESSREF=**, this program operates on the CAS server and utilizes threading to run in parallel:

```
/* Load the sashelp.cars table to the CAS server */
proc casutil outcaslib="casuser";
    load data=sashelp.cars casout="cars" replace;
run;

/* Define a method to select 2-door models and execute it in parallel
*/
proc ds2 sessref=mysess;
    thread cars_thd / overwrite=yes;
    method run();
        set casuser.cars;
        if (prxmatch('/2dr/', model) > 0) then do;
            put make= model= msrp=;
            output;
        end;
    end;
endthread;

data cars_2dr / overwrite=yes;
```

```
    dcl thread cars_thd t;
    method run();
        set from t threads=4;
    end;
enddata;
run;
quit;
```

In addition to the above example, a similar task can be accomplished using the SQL procedure. Since the table is stored on the CAS server, both the SQL procedure and **PRXMATCH** will execute on the CAS server as follows:

```
libname caslib cas;
data caslib.cars;
    set sashelp.cars;
run;

proc sql;
    create table cars_2dr as
    select make, model, msrp
    from caslib.cars
    where prxmatch("/2dr/", model);
quit;
```

This approach demonstrates how executing PRX functions on the CAS server can improve performance and efficiency in data processing.

MORE METACHARACTERS

LOOK AROUND

In certain scenarios, you may need to examine the context surrounding a potential match without including that context in the actual match. This can be achieved using look-around assertions that check for the presence or absence of characters ahead of or behind the current position without consuming those characters. There are four types of look-around assertions: positive look-ahead `(?=...)`, negative look-ahead `(?!...)`, positive look-behind `(?<=...)`, and negative look-behind `(?<!...)`. For instance, the pattern `(?<=red)` is a positive look-behind assertion. It asserts that the current position in the string must be immediately preceded by the text “red”. A pattern using this assertion, such as “(?<=red)\w+”, would then effectively match a word that follows “red”.

Here is an example:

```
data _null_;
    text = "Yellow bananas, red apples, and green grapes";
    id = prxparse("/(?<=red )\w+/i");
    call prxsubstr(id, text, pos, len);
    word = ksubstr(text, pos, len);
    put word=;
run;
```

The output in the log will be:

```
word=apples
```

GREEDY AND LAZY REPETITION FACTORS

Repetition factors such as "*", "+", and "?" are classified as greedy repetition factors, meaning they attempt to match a subexpression as many times as possible. In contrast, lazy repetition factors match a subexpression the minimum number of times necessary to satisfy the match condition. To denote lazy repetition, simply append a question mark to the greedy repetition factors. For example, "*?" represents the lazy version of "*".

In the following example, the pattern `/a.*b/` employs the greedy repetition factor "*", prompting **PRXMATCH** to match as many characters as possible before reaching the final occurrence of "b":

```
data _null_;
    id = prxparse("/a.*b/");
    text = "abcbc";
    pos = prxmatch(id, text);
    matched = prxposn(id, 0, text);
    put matched=;
run;
```

The output in the log will be:

```
matched=abcb
```

Conversely, when using the lazy version, the pattern matches the minimum number of characters necessary before reaching the final "b":

```
data _null_;
    id = prxparse("/a.*?b/");
    text = "abcbc";
    pos = prxmatch(id, text);
    matched = prxposn(id, 0, text);
```

```
    put matched=;
run;
```

The output in the log will be:

```
matched=ab
```

When using the greedy repetition factor "*", **PRXMATCH** first matches as many characters as possible until it reaches the end of the string, subsequently backtracking when it determines that the remaining characters cannot match the final "b". In contrast, when employing the lazy repetition operator "*?", the function evaluates whether the next character is "b" at each step. Enabling the PRX debug log can provide insight into the differences in matching behavior.

In addition to producing distinct matching results, these two types of repetition operators may lead to varying performance due to their different matching processes. By mastering these and other metacharacters, you can leverage the full potential of regular expressions. The PRX functions and routines support a comprehensive set of over one hundred metacharacters, a complete list is available in the SAS documentation online. Please refer to the link for “Tables of Perl Regular Expression (PRX) Metacharacters” in the References section.

TAKING THE NEXT STEP

For those who are new to the field, the book *Unstructured Data Analysis: Entity Resolution and Regular Expressions in SAS* (2018) by K. Matthew Windham offers practical examples and techniques for utilizing regular expressions to handle unstructured data effectively. Additionally, professionals in the health and life sciences may find value in the research papers authored by Amy Alabaster and Mary Anne Armstrong, specifically titled *Cracking Cryptic Doctors’ Notes with SAS PRX Functions* (2020) and *Interpreting Electronic Health Data Using SAS PRX Functions* (2018).

SAS provides various resources for information regarding PRX functions and call routines. A recommended starting point is the SAS documentation website at https://support.sas.com/en/documentation.html. A search for the term "PRX" on this page yields numerous links to a wide array of resources, including tables detailing Perl Regular Expression Metacharacters and much more. A useful tip sheet can also be found at the following link: https://support.sas.com/content/dam/SAS/support/en/products-solutions/base-sas/tip-sheets/regexp-tip-sheet.pdf.

Furthermore, entering "PRX" into the search bar at https://www.lexjansen.com will return over 800 results, including the previously mentioned papers by Alabaster and

Armstrong, as well as several works by David Cassell, who was among the first to advocate for the capabilities of PRX functions and call routines. Another notable paper that may appear in this search is *Functions (and More!) on CALL!* by Richann Watson and Louise Hadden, presented at PharmaSUG 2022, which discusses the use of **PRXCHANGE** and **PRXPARSE**.

CONCLUSION

SAS PRX functions significantly expand the toolkit available to skilled SAS programmers, allowing for more efficient management of complex text string manipulations compared to traditional string manipulation functions. It is essential for programmers to familiarize themselves with these functions and the associated call routines to effectively apply them in relevant programming contexts.

REFERENCES

- Tables of Perl Regular Expression (PRX) Metacharacters. Available at: https://documentation.sas.com/?docsetId=lefunctionsref&docsetVersion=v_001&docsetTarget=p0s9ilagexmjl8n1u7e1t1jfnzlk.htm
- Perl Regular Expressions. Available at: https://perldoc.perl.org/perlre
- Cassell, David L. 2007. "The Basics of the PRX Functions." *Proceedings of the SAS Global 2007 Conference*, Cary, NC: SAS Institute Inc. Available at: https://support.sas.com/resources/papers/proceedings/proceedings/forum2007/223-2007.pdf

CONTACT INFORMATION

We welcome and encourage your comments and questions. Please contact the authors at:

Edwin (You) Xie
SAS Institute Inc.
you.xie@sas.com

John LaBore
SAS Institute Inc.
john.labore@sas.com