Paper CT17

Exploring Underutilized Capabilities in SAS – Approaches Seldom Used

Diganta Bose, Cytel Inc., India

ABSTRACT

While SAS® programmers often rely on familiar techniques to accomplish routine tasks, this presentation explores two underutilized yet powerful alternative programming methodologies that significantly outperform more conventional approaches to data analysis in the pharmaceutical industry: regular expressions and advanced macro implementations.

Certain short demonstrations will be aimed at showcasing how regular expressions excel beyond standard character functions to perform complex pattern matching in biological sequence data. It will also showcase the power of Regular Expressions combined with the power of SAS® Macro Language can be used in unconventional ways to automate the derivations of group of related variables. Often these methods prove more robust.

Through targeted real-world examples, we will see how these alternative methods improve computational accuracy and efficiency when compared to more conventional programming processes. It will also provide practical insights into implementing these advanced techniques so that SAS® programmers can enhance their arsenal and expand their capabilities within the pharmaceutical research environment.

BASICS OF SAS® PERL REGULAR EXPRESSIONS OR PRX

Perl Regular Expressions (PRX) in SAS® provide a powerful and flexible way to search, extract and manipulate text patterns within character data. PRX was introduced to SAS® to overcome the limitations of traditional SAS® string functions and brings Perl-like pattern matching directly into SAS® programming. Using functions such as PRXPARSE, PRXMATCH, PRXSUBSTR and PRXCHANGE, programmers can efficiently validate data, identify complex patterns and perform advanced text transformations.

PRX expressions are defined using standard Perl syntax, making them expressive and concise for tasks like validating emails, parsing free-text fields, or standardizing inconsistent data. Typically, a regular expression is compiled once using PRXPARSE and then reused for performance efficiency, especially within DATA step loops.

Overall, PRX in SAS® significantly enhances text-processing capabilities, reduces code complexity, and improves maintainability, making them essential tool for statistical programmers dealing with real world or unstructured data.

Below table (Table 1) summarizes some of the Key aspects of Regular expressions, delimiters and associated SAS® functions that are commonly used.

Table 1: SAS® Perl Regular Expressions (PRX) – One-Page Cheat Sheet *

Common Regular Expression Delimiters in SAS®

Delimiter	Description	Example
/ /	Default and most commonly used	/\d{4}/
# #	Useful when pattern contains /	#^\w+@\w+\.\w+$#
! !	Improves readability for complex patterns	!\s+!
~ ~	Often used for whitespace-heavy patterns	~\bword\b~
{ }	Supported but less common	{^\d+$}
( )	Allowed but may reduce clarity	(^[A-Z]+$)

Tip: Opening and closing delimiters must match. Regex modifiers (e.g. i) follow the closing delimiter.

Common Regex Symbols (SAS® PRX)

Symbol	Meaning	Example	Matches
.	Any single character	a.b	a1b, acb
^	Start of string	^ABC	ABC123
$	End of string	XYZ$	123XYZ
*	0 or more	ab*	a, abbb

<table>
<tr><th>Symbol</th><th>Meaning</th><th>Example</th><th>Matches</th></tr>
<tr><td>+</td><td>1 or more</td><td>ab+</td><td>ab, abbb</td></tr>
<tr><td>?</td><td>0 or 1</td><td>colou?r</td><td>color, colour</td></tr>
<tr><td>{n}</td><td>Exactly n times</td><td>\d{4}</td><td>2025</td></tr>
<tr><td>{n,}</td><td>At least n times</td><td>\d{2,}</td><td>12, 1234</td></tr>
<tr><td>{n,m}</td><td>Between n and m</td><td>\d{2,4}</td><td>12–1234</td></tr>
<tr><td>[]</td><td>Character class</td><td>[A-Z]</td><td>Any capital letter</td></tr>
<tr><td>[^]</td><td>Negated class</td><td>[^0-9]</td><td>Non-digit</td></tr>
<tr><td>()</td><td>Capture group</td><td>(\w+)</td><td>Stored match</td></tr>
<tr><td>|</td><td>OR (alternation)</td><td>cat|dog</td><td>cat or dog</td></tr>
<tr><td>\</td><td>Escape character</td><td>\.</td><td>Literal .</td></tr>
</table>

Common Character Classes

Class	Meaning	Example Match
\d	Digit (0–9)	5
\D	Non-digit	A
\w	Word character (A–Z, a–z, 0–9, _)	abc_1
\W	Non-word character	@
\s	Whitespace	Space, tab
\S	Non-whitespace	A
\b	Word boundary	\bcat\b

Most Used SAS® PRX Functions

Function	Purpose
PRXPARSE	Compile regex pattern
PRXMATCH	Check if pattern exists
PRXSUBSTR	Locate matched substring
PRXCHANGE	Search and replace
PRXPOSN	Extract capture groups
PRXNEXT	Iterate over multiple matches
PRXFREE	Free regex memory

The power of regular expressions can be combined with other SAS® programming aspects like normal data-steps as well as SAS® Macro language. Unfortunately, regular expressions cannot be used in proc SQL directly in the same way as it can be used in DATA steps. However, with a combination of DATA-Steps and proc SQL, most of the intended tasks can be achieved.

SAS® regular expressions are especially powerful when used in combination with the SAS® Macro language, as they enable **dynamic pattern-driven code generation and validation**. PRX functions can be embedded within macro logic to validate macro parameters, parse complex user inputs, and enforce naming conventions before executing DATA steps or procedures. For example, macros can use PRXMATCH to ensure that a dataset name, variable list, or date parameter conforms to an expected pattern, thereby preventing downstream execution errors. Regular expressions can also be used to **tokenize and transform macro variables** using PRXCHANGE, making macros more flexible and reusable across studies or projects. By combining PRX with macro loops and conditional logic, programmers can build robust, defensive macros that adapt to varying inputs, reduce manual checks, and improve maintainability. This synergy is particularly valuable in large-scale, automated SAS® workflows where consistency, validation, and scalability are critical.

EXAMPLES OF VERY BASIC APPLICATION

In order to understand how PRX is used in SAS® programming language let us start with a simple example.

```
data a;
  text = "test123@example.com is my email id";
  re = prxparse('/(\w+)@(\w+\.\w+)/');

  if prxmatch(re, text) then do;
    call prxsubstr(re, text, s, l);
    email = substr(text, s, l);
    user  = prxposn(re, 1, text);
    dom   = prxposn(re, 2, text);
  end;

  clean = prxchange('s/\d+/XXX/', -1, text);

  call prxfree(re);
run;
```

From the above code, let us first understand the regular expression statement.
/(\w+)@(\w+\.\w+)/

Let us break it down to understand what pattern this regular expression is trying to extract.

Table 2: Breaking down a Regular Expression (/(\w+)@(\w+\.\w+)/)

```
( \w+ )  @  ( \w+  \.  \w+ )
|  |     |   |     |   |
|  |     |   |     |   └─ Top-level domain (com)
|  |     |   |     └──── Literal dot (.)
|  |     |   └────────── Domain name (example)
|  |     └────────────── @ symbol
|  └──────────────────── Username (test123)
└─────────────────────── Capture Group 1
         └────────── Capture Group 2
```

SAS Function	What It Extracts	Result
PRXMATCH	Confirms pattern exists	Match found
PRXSUBSTR	Full matched email	test123@example.com
PRXPOSN(re,1,...)	Capture Group 1	test123
PRXPOSN(re,2,...)	Capture Group 2	example.com

USAGE IN BLINDING DATA/ HIDING IDENTIFYABLE TRACES FROM DATA

When we run the above dataset we get the below result.

Table 3: Example of Data extraction and blinding data

text	s	l	email	user	dom	clean
test123@example.com is my emailid	1	19	test123@example.com	test123	example.com	testXXX@example.com is my emailid

Notice how the variable "clean" in the above table presents blinded version of the variable "text", wherein the userid numerals has been blinded. This was accomplished using the statement `clean = prxchange('s/\d+/XXX/', -1, text)`. "\d+" indicates one or more digits, and the numerals "123" which was the user id, was blinded and replaced by a text "XXX".

USAGE IN PROC SQL

Unfortunately, PRX cannot be used directly inside a PROC SQL WHERE clause in the same way as LIKE or CONTAINS. SAS® Regex is function based and not operator based in PROC SQL. For complex parsing or repeated use, prefer a DATA step, then filter in SQL. PROC SQL does not natively support regex syntax. However, you *can* call PRX functions such as PRXMATCH, PRXCHANGE, etc., inside a SQL query.

For instance you cannot use PRX like below directly in PROC SQL.

```
/* Below Code won't work*/
proc sql;
select *
  from emails where email regex '/\w+@\w+\.\w+/';
quit;
```

Instead, what you can do is, define the regular expression in a DATA step and then process as needed withing PROC SQL. Below is a recommended usage.

```
data emails2;
  set emails;
  re = prxparse('/\w+@\w+\.\w+/');
  is_valid = prxmatch(re, email);
run;

proc sql;
  select *
  from emails2 where is_valid > 0;
quit;
```

APPLICATION TO STANDARDIZE INCONSISTENT DATA

When it comes to data from Clinical Trials and sometimes from Real World Data, often we receive data in the most unstructured form and often is not clean. Using PRX, programmers can identify variable text formats, remove unwanted characters, enforce naming conventions and normalize values into standardized structures. PRX also support data quality checks by flagging invalid patterns and ensuring compliance with predefined rules. Overall, PRX provide a powerful, scalable and reusable approach for cleaning and harmonizing unstructured data across large datasets.

EXAMPLE OF REPLACING KEY TREATMENT TEXTS

Let us evaluate with an example, wherein you have a study which follows parallel design and subjects are randomized into certain treatment groups. However the treatment element texts in SDTM SE, the treatment names in SDTM EX and the ARM codes in SDTM DM are inconsistent. Let us imagine, our job is to correct this and harmonize the data in order to have a final, submission ready SDTM.

In the given data, the treatment names are inconsistently named as Cyt0001 0.3 MG at some places whereas the actual treatment names are different at other places printed as DRUG X 0.3 MG though they are functionally the same product administered. We want to harmonize such texts to be represented as DRUG X 0.3 MG. Similarly, Cyt001 1 MG should be represented as DRUGX 1 MG and so on. In order to have this corrected the below simple statements can be used.

```
elementk = prxchange("s/Cyt001/DRUGX/",-1,element);
element_B = prxchange("s/Comparative/DRUGY/",-1,elementc);
```

The outcomes of the applying the above statements is represented in the below table (Table 4) and notice how the original variable ELEMENT was transformed to variable ELEMENT_B. The first statement replaces all texts like *Cyt001* with *DRUGX* whereas the second statement replaces the text *Comparative* with *DRUGY*.

Table 4: Example of text replacements

ELEMENT	ELEMENT_B
Comparative 20 MG	DRUGY 20 MG
Cyt001 0.3 MG	DRUGX 0.3 MG
Cyt001 1 MG	DRUGX 1 MG
Cyt001 10 MG	DRUGX 10 MG
Cyt001 3 MG	DRUGX 3 MG

Suppose you wanted to extract specific information from the treatment element names like DRUGX 1 MG and DRUGX 3 MG, so that you can do a quick pattern matching check between SDTM SE ELEMENT texts and SDTM EX EXTRT texts and potentially develop a macro to derive treatment related variables in ADSL (TRT01A, TR01SDT and TRT01EDT, e.t.c..). Below simple statement can easily accomplish this.

```
dosnum = prxchange("s/[^0-9]//",-1,element_b);
dname = prxchange("s/[0-9]|MG//",-1,element_b);
```

Table 5: Example of extracting specific information from texts

ELEMENT_B	DOSNUM	DNAME
DRUGX 1 MG	1	DRUGX
DRUGX 3 MG	3	DRUGX

Using these techniques one can easily do a pattern matching task to cross check between SDTM SE/ TE and SDTM EX and DM in order to extract specific information to derive key treatment variables in ADSL like TRT01A, TR01SDT and TRT01EDT, e.t.c..

EXAMPLE OF NORMALIZING UNSTRUCTURED LABORATORY TEST NAMES

When we encounter clinical trial lab data, especially from unstructured sources like Real World, it is very common to imagine Laboratory test names entered as free texts at sites or clinics resulting in multiple representations of the same tests. These inconsistencies complicate analysis and CDISC Mapping and hence, as a programmer, we try to write robust codes that handles these kinds of situations. In this given example, the same test is represented in 5 different ways as follows.

1. ALT (SGPT)
2. S.G.P.T
3. Alanine Transaminase
4. alt level
5. SGPT – serum

In order to harmonize the above and for analysis be identified as a single test with lbtest = 'Alanine Aminotransferase and lbtestcd = ALT, a simple SAS code utilizing the power of PRX can handle this situation effectively.

```
data lb_standardized;
    retain lbtest_raw lbtest_clean lbtestcd lbtest;
    set lb_raw;

    /* Convert to uppercase; alternatively can use the /i operator in regEX*/
    lbtest_raw = upcase(lbtest);

    /* Remove special characters */
    lbtest_clean = prxchange('s/[^A-Z ]+/ /', -1, lbtest_raw);

    /* Normalize multiple spaces */
    lbtest_clean = prxchange('s/\s+/ /', -1, lbtest_clean);

    /* Identify ALT / SGPT using regex */
    if prxmatch('/\b(ALT|S\s*G\s*P\s*T|ALANINE\s+TRANSAMINASE)\b/', lbtest_clean) then
do;
        lbtestcd = 'ALT';
        lbtest   = 'Alanine Aminotransferase';
    end;
run;
```

The below table (table 6) depicts the results of the above line of codes. Notice how lbtest_raw initially was, unclean and inconsistent, and how by using PRX the transformed values are displayed in lbtest and lbtestcd. In real time situations in the studies you are handling, one can easily write a robust macro to handle all possible unclean situations when it comes to Lab data.

Table 6: Example of transformed values of lab tests

lbtest_raw	lbtest_clean	lbtestcd	lbtest
ALT (SGPT)	ALT SGPT	ALT	Alanine Aminotransferase
S.G.P.T	S G P T	ALT	Alanine Aminotransferase
ALANINE TRANSAMINASE	ALANINE TRANSAMINASE	ALT	Alanine Aminotransferase
ALT LEVEL	ALT LEVEL	ALT	Alanine Aminotransferase
SGPT - SERUM	SGPT SERUM	ALT	Alanine Aminotransferase

APPLICATION OF PRX IN COMBINATION WITH SAS® MACRO

Imagine you are working in the standards team of your organization, and you have a task in hand to develop a Utility macro that utilizes certain key ADaM ADSL variables, and process them in order to carry out some derivations down the line. In this particular task, the ADSL variables in question are the population flag variables and the treatment related variables. As a programmer, the first step in order to identify specific variables from a given dataset, you would need to either use a PROC SQL dictionary table or simply a PROC COMPARE as below.

```
proc contents data=ads.adsl out=_m_contents_adsl(keep=memname name length type varnum)
noprint;
run;
```

The next step can easily be accomplished by using certain simple regular expressions withing a DATA step by copying the PROC COMPARE outputted dataset _m_contents_adsl as below.

1. `prxmatch(prxparse("/[\D+]fl/i"),name)>0:` Can be used to identify the population flag variables like SAFFL, ITTFL and so on. This single statement can easily capture all possible flag variables, by capturing the naming convention patterns followed in CDISC ADaM – All flag variable names end with “FL” and “/i” simply indicates capturing uppercase or lowercase.
2. `prxmatch(prxparse("/trt([\d][\d])p/i"),name)>0:` Can be used to capture all planned treatment related variables like TRT01P, TRT02P and up to TRT99P, again based on CDISC naming convention. Similar statement can be used to capture TRT01A, TRT02A and so on.
3. `prxmatch(prxparse("/tr([\d][\d])[s|e]dt/i"),name)>0`: Can simply capture all treatment start and end dates like TR01SDT, TR02SDT, up to TR99SDT and TR01EDT, TR02EDT, up to TR99EDT.

As a programmer working on developing SAS® macros, you would agree that tasks like these wherein you have to select or check the presence of certain key required variables for further processing of macros is a common one and PRX can play a key role in achieving this.

PRX can be embedded within SAS® macro language to parse complex user inputs, validate macro parameters, and enforce naming conventions before executing DATA steps or procedures. In this particular example, we will see how this can be done. The situation is that the programmer is required to write a SAS® macro to derive agegroup variables in ADSL using numeric AGE values from SDTM DM. Let us call this macro as %m_groupage and the macro declaration looks like this.

```
%m_groupage(inds=testdata,/*Input Dataset*/
            agegrlst=%str(<30#30-45#46-65#>65), /*age group specification*/
            varname=agegr1,/*Age Group Variable Name*/
            outdata=_1_sample); /*Output Dataset*/
```

While three of the parameters are very straightforward to understand, which are inds, varname and outdata, the user of the macro may face difficulty in understanding the macro parameter ***agegrlst*** in particular. This user is required to enter inputs in this parameter in a certain way. The requirement is to enter a “#” separated list of age group ranges in the below format only.

{(Below x1) or (Under x1) or (< x1) or x0 - (x1 - 1)} #
x1 - x2 #
(x2 + 1) - x3 #
(x3 + 1) - x4 #
………
(xi + 1) - xj #
(xj + 1) - xn #
{(Over xn) or (Above xn) or (>xn) or (xn + 1) - xm}

This means entries like agegrlst=%str(**>30**#30-45#46-65#>65) is incorrect as >30 and the next groups have unclear boundaries and would overlap. Hence the expected correct age group specification entry would be one of the following alternatives:

1. agegrlst=%str(<30#30-45#46-65#>65) or
2. agegrlst=%str(below 30#30-45#46-65#Above 65) or
3. agegrlst=%str(Under 30#30-45#46-65#Over 65)
4. or any combination of the above 3

It is very apparent, as a developer of a macro, one would need to write checks to ensure the entries made to this parameter are as per specification in the user manual of the macro.

Let us first see an example of correct entry to the parameter *agegrlst* and expected output.

agegrlst=%str(<30#30-45#46-65#>65);

Table 7: Input and expected output of macro %m_groupage

Sample Input

USUBJID	AGE
CYT001201-101-1089	56
CYT001201-102-1105	42
CYT001201-103-1131	75
CYT001201-203-1186	26

Expected Output

USUBJID	AGE	agegr1	agegr1N
CYT001201-101-1089	56	46-65	3
CYT001201-102-1105	42	30-45	2
CYT001201-103-1131	75	>65	4
CYT001201-203-1186	26	<30	1

Let us now examine certain cases of incorrect entries to the parameter and certain SAS macro codes, with help of PRX handles the situation in the below two examples.

MACRO %M_GROUPAGE INCORRECT ENTRY CHECKING EXAMPLE 1

Let us examine the below two situations of incorrect entry
*agegrlst=%str(**>30**#30-45#46-65#>65):- Incorrect entry because >30 overlaps and could be any number in 30-45, 46-65 and so on.*

The first step would be to initiate a loop using the following proc sql to determine the number of age group ranges.

```
proc sql noprint;
 select max(count(agegrlst,"#"))+1 into:nagegrp from _m_t;
quit;
```

Then we determine the correct order of the ranges using the below steps. Basically the below steps determine the order of the ranges based on increasing value of numerical age.

```
data _m_t_expand;
   set _m_t;
   array range{&nagegrp} $100;
   array rsum{&nagegrp};
   array rord{&nagegrp} $200;
  do q = 1 to &nagegrp;
      range{q}=strip(scan(agegrlst,q,"#"));
    if prxmatch(prxparse("/(above|over|>)( *)[\d+]/i"),range{q})>0 then rsum{q} =
9999;
    else if prxmatch(prxparse("/(below|under|<)( *)[\d+]/i"),range{q})>0 then rsum{q}
= 1;
    else if prxmatch(prxparse("/-/i"),range{q})>0 then
                            rsum{q} = input(strip(scan(range{q},1,"-")),best.) +
input(strip(scan(range{q},2,"-")),best.);
```

```
        rord{q} = strip(put(rsum{q},best.))||"|"||strip(range{q});
   end;
run;
```

After accomplishing arranging in order the age group ranges based on increasing values of age by using a simple proc sort, we can write a code like below to catch the invalid entry case of >30 overlapping with other ranges.

```
data _m_invalid_entry_checks;
   set _m_ord;
   iord=_n_;
*------------------------------------*
|Below steps check incorrect entries|
*------------------------------------;
  if prxmatch(prxparse("/to/i"),group_name)>0 then flag="Y";
  if prxmatch(prxparse("/=/i"),group_name)>0 then flag="Y";
  if &lobs > 1 then do;
    if iord = 1 and prxmatch(prxparse("/(>)|(above)|(over)/i"),group_name)>0 then
flag="Y";
    if iord = &lobs and prxmatch(prxparse("/(<)|(below)|(under)/i"),group_name)>0 then
flag="Y";
  end;
 if 1 < ord < &lord and
prxmatch(prxparse("/(>)|(<)|(above)|(below)|(under)|(over)|(>=)|(<=)/i"),group_name)>0
then flag="Y";
if 1 < iord < &lobs and
prxmatch(prxparse("/(>)|(<)|(above)|(below)|(under)|(over)|(>=)|(<=)/i"),group_name)>0
then flag="Y";
run;
```

The line in **bold** in the above SAS® code catches the incorrect entry as the order gets misplaced into thinking the last range in the group between >65 and >30 and the symbol ">" is permitted only to specify the last range in order. Below table shows the outcome of the above code and where the incorrect entry is flagged and printed as a user defined error message in the log.

Table 7: incorrect entry of *>30#30-45#46-65#>65*

ord	group_name	iord	flag
75	30-45	1	
111	46-65	2	
9999	>30	3	Y
9999	>65	4	

```
ERROR:------ Incorrect Entry in parameter AGEGROUP -----
----Please enter acceptable values as indicated in the user guide----
```

MACRO %M_GROUPAGE INCORRECT ENTRY CHECKING EXAMPLE 2

Let us examine the below two situations of incorrect entry

*agegrlst=%str(<30#**31**-45#46-65#>65): Incorrect entry because between <30 and 31 – 45, values like AGE=30 gets missed.*

We continue from the ordering steps as shown further above in the previous example. Once the ordering is accomplished, we can write a code like this to catch the case of invalid entry in this example, which is 31-45.

```
data _m_check_overlap;
          set _m_ord;
        if prxmatch(prxparse("/-/i"),group_name)>0 then do;
          l1 = input(strip(scan(group_name,1,"-")),best.);
          u1 = input(strip(scan(group_name,2,"-")),best.);
        end;
        else if
      prxmatch(prxparse("/(>)|(<)|(above)|(below)|(under)|(over)/i"),group_name)>0
      then do;
           l1 = input(substr(group_name,anydigit(group_name)),best.) + 1;
           u1 = input(substr(group_name,anydigit(group_name)),best.) - 1;
```

```
          end;
           ux=lag(u1);
          if not missing(ux) then test_val = ux;
          else if missing(ux) then test_val = 0;
          if l1 > test_val then pass = "Y";/*Checking Overlapping Age values in
        AGEgroups defined*/
          else pass = "N";
          if l1 = 0 and ord = &ford then pass = "Y";
          if ord > &ford and (l1 - ux) > 1 then passm = "N";/*Checking Skipped Age
        values in AGEgroups defined*/
          else passm = "Y";
run;
```

The above steps will not only check invalid entry cases like *<30#**31**-45#46-65#>65*, but also overlapping range cases like *<30#30-45#42-65#>65.* The above code first determines the upper and the lower values of the closed ranges (variables l1 and u1) and then tries to check for any instances of overlaps or values not falling in declared ranges. The relevant lines that catches and flags such cases of incorrect ranges specified is in **bold** from the above code.

Table 8: incorrect entry of *<30#31-45#46-65#>65*

ord	group_name	l1	u1	ux	test_val	pass	passm
1	<30	31	29	.	0	Y	Y
76	31-45	31	45	29	29	Y	N
111	46-65	46	65	45	45	Y	Y
9999	>65	66	64	65	65	Y	Y

```
ERROR:  [m_groupage] ----- Ranges specified in parameter AGEGRLST misses out certain values of AGE
```

CONCLUSION

In conclusion, regular expressions integrated with SAS Macro programming provide a powerful and flexible framework for handling complex text processing, dynamic data validation, and large-scale standardization tasks. By leveraging PRX functions within macros, programmers can create reusable, parameter-driven solutions that efficiently manage unstructured and semi-structured data across diverse clinical and operational datasets. This approach enhances automation, consistency, and scalability while reducing manual intervention and programming redundancy. However, regular expressions can be difficult to read, debug, and maintain, particularly within macro code where resolution timing adds complexity. Poorly designed patterns may also impact performance on large datasets. Despite these limitations, careful documentation, modular macro design, and rigorous testing can mitigate risks, making SAS macro-driven regular expressions a valuable and effective tool in advanced data processing workflows.

In recent times, the industry is seeing a gradual shift of focus from SAS to R. Fortunately, R has built-in support for regular expressions, primarily using PCRE (Perl-Compatible Regular Expressions) through base functions like grep, grepl, sub, gsub, and regexpr. They are widely used in R for pattern matching, data cleaning, validation, and text manipulation in both base R and tidyverse workflows.

REFERENCES

1. SAS® Regular expression tip sheet:- https://support.sas.com/content/dam/SAS/support/en/products-solutions/base-sas/tip-sheets/regexp-tip-sheet.pdf
2. How to reduce programming time for ADaM Creation? – Presenting STAG, a Tool Based Approach:- https://www.lexjansen.com/phuse-us/2020/si/SI06.pdf

CONTACT INFORMATION

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

Author Name: Diganta Bose
Company: Cytel Inc.
Address: Remote/ Bengaluru
Email: diganta.bose@cytel.com
LinkedIn: https://www.linkedin.com/in/diganta-bose-386222a/