PharmaSUG 2026 - Paper AP-271

Stop Guessing. Start Matching. High-Impact SAS PRX Patterns in 20 minutes

Charu Shankar, SAS Institute Inc.

ABSTRACT

Messy text is everywhere in clinical programming—IDs embedded in free text, inconsistent visit labels, unpredictable separators, and key-value strings that require precise parsing. Traditional approaches using INDEX, SUBSTR, and FIND often lead to brittle logic, false positives, and code that is difficult to maintain.

This paper presents a pattern-driven approach to text processing in SAS using PRX (Perl Regular Expressions). Through three core patterns—validation, extraction, and precise targeting—it demonstrates how PRX enables reliable, scalable, and maintainable solutions.

Each pattern is illustrated with concise, production-ready code and practical clinical scenarios. The paper emphasizes how PRX functions return numeric results that SAS interprets in a Boolean context, reinforcing a deeper understanding of SAS logic.

Takeaway: cleaner parsing logic, fewer false positives, and reusable patterns that elevate text handling from string manipulation to structured matching.

INTRODUCTION: FROM STRINGS TO PATTERNS

Clinical data rarely arrives clean. Identifiers, visit labels, and key-value strings are often embedded in inconsistent formats, making reliable parsing difficult.

Traditional approaches rely on layered combinations of INDEX, SUBSTR, and FIND. While functional, these methods are fragile, difficult to scale, and prone to subtle errors.

PRX introduces a different way of thinking. Instead of asking *where text occurs*, we define *what valid text looks like*. This shifts text processing from string manipulation to pattern matching—enabling precise validation, controlled extraction, and reduced false positives.

PATTERN 1 — VALIDATION: ENSURING STRUCTURAL INTEGRITY

Validation confirms whether a value matches an exact expected format.

Example: Validate USUBJID Format

```
data valid_ids;
  length usubjid $20 valid_flag 8;

  input usubjid $;

  /* Compile pattern once */
  if _n_=1 then
    re_id = prxparse('/^[A-Z]{3}-\d{3}-\d{4}$/');

  valid_flag = prxmatch(re_id, strip(usubjid));

  datalines;
```

```
ABC-123-4567
abc-123-4567
ABC1234567
;
run;
```

Output

usubjid	valid_flag
ABC-123-4567	1
abc-123-4567	0
ABC1234567	0

Explanation
^ and $ anchor the pattern
[A-Z]{3} enforces uppercase prefix
\d{3} and \d{4} enforce numeric structure

👉 **Key insight:** Anchors ensure full-string validation—not partial matches.

PATTERN 2 — EXTRACTION: CAPTURING STRUCTURE FROM TEXT

Extraction pulls structured values from unstructured strings.
Example: Extract Week Number from Visit Label

```
data visit_extract;
  length visit $40 weeknum 8;

  input visit $char40.;

  if _n_=1 then
    re_week = prxparse('/\bWeek\s+(\d+)\b/i');

  if prxmatch(re_week, visit) then
    weeknum = input(prxposn(re_week, 1, visit), 8.);

  datalines;
Week 12 Visit
Week 3 Follow-up
Screening
;
run;
```

Output

visit	weeknum
Week 12 Visit	12
Week 3 Follow-up	3
Screening	.

Explanation

- (\d+) captures numeric portion
- prxposn retrieves captured group
- case-insensitive match (i)

👉 **Key insight:** Capture groups convert text into structured variables.

PATTERN 3 — TARGETING: PRECISION WITH LOOKAROUNDS

Look arounds isolate values without including surrounding text.

Example: Extract AESER Value from Key-Value String

```
data kv_extract;
  length kvline $200 aeser $20;

  input kvline $char200.;

  if _n_=1 then
    re_aeser = prxparse('/(?<=\bAESER=)[^;]+(?=;)/i');

  if prxmatch(re_aeser, kvline) then do;
    call prxposn(re_aeser, 0, start, len);
    aeser = substr(kvline, start, len);
  end;

  datalines;
AESER=Y;AESEV=MODERATE;
AESER=N;AESEV=MILD;
;
run;
```

Output

kvline	aeser
AESER=Y;AESEV=MODERATE;	Y
AESER=N;AESEV=MILD;	N

Explanation

- (?<=...) lookbehind
- (?=...) lookahead
- extracts value without delimiters

👉 **Key insight:** Lookarounds isolate meaning without extra cleanup.

STANDARDIZATION: PREPARING TEXT FOR MATCHING

Text often requires normalization before matching.

```
data cleaned;
  length raw_id $30 clean_id $30;

  input raw_id $;

  clean_id = prxchange('s/[_\s]+/-/', -1, raw_id);
```

```
  datalines;
ABC_123 4567
XYZ 789_1234
;
run;
```

👉 **Key insight:** Clean inputs improve match reliability.

COMPILE ONCE, APPLY MANY

Efficient PRX usage compiles patterns once.

```
data source;
  length visit $40;
  input visit $char40.;
  datalines;
Week 12 Visit
Screening
Week 3 Follow-up
Unscheduled Visit
Week 20
;
run;

data example;
  set source;
  retain re_week;

  if _n_=1 then
    re_week = prxparse('/\bWeek\s+\d+\b/i');

  match_flag = prxmatch(re_week, strip(visit));
run;
```

In repeated text-processing workflows, the same regular expression is often applied to many observations. In these cases, the pattern should be compiled once with PRXPARSE and then reused throughout the DATA step. This avoids unnecessary recompilation, improves efficiency, and makes the intent of the code clearer. The goal of this pattern is not to produce a complex output, but to demonstrate a scalable way to apply the same matching logic across many rows.

👉 **Key insight:** Compile the pattern once, then reuse it for every observation.

COMMON PITFALLS AND HOW TO AVOID THEM

1. Partial Matches

Without anchors, unintended matches occur.

👉 Always use ^ and $ when validating full structure.

2. Case Sensitivity

Patterns default to case-sensitive.

👉 Use /i for case-insensitive matching.

3. Overuse of INDEX

INDEX finds position—not meaning.
👉 Use PRX for structural logic.

4. Recompiling Patterns
Avoid compiling inside loops.
👉 Use _n_=1 to compile once.

PUTTING IT ALL TOGETHER

A typical workflow:

1. **Standardize input**
2. **Validate structure**
3. **Extract components**
4. **Apply logic downstream**

This layered approach replaces fragile string logic with robust pattern-based processing.

SECTION TAKEAWAY

PRX transforms text handling into:

Function	Purpose
PRXMATCH	Detect patterns
PRXPOSN	Extract values
PRXCHANGE	Standardize text

👉 Each returns numeric results interpreted by SAS as Boolean logic.

CONCLUSION

Text parsing is not about slicing strings—it is about recognizing patterns.
PRX enables:

- precise validation
- structured extraction
- reliable targeting

By shifting from positional logic to pattern logic, programmers can write code that is clearer, more maintainable, and more accurate.
Stop guessing. Start matching.

CONTACT INFORMATION

Charu Shankar
SAS Institute Inc.
Charu.Shankar@sas.com

APPENDIX A — SAS PRX Recipe Card (Quick Reference)

This appendix provides a compact, reusable reference for common PRX patterns used in validation, extraction, and text standardization. It is designed for practical use during development and QC workflows.

GOLDEN RULE: COMPILE ONCE, APPLY MANY

```
retain re;
if _n_=1 then re = prxparse('/pattern/options');
if prxmatch(re, text) then ...;
```

👉 **Key insight:** Compile patterns once for performance and consistency.

1. VALIDATE — ANCHORS AND BOUNDARIES

Use anchors to enforce full-string matches and avoid false positives.

A. Exact ID Format (USUBJID = AAA-999-9999)

```
valid_usubjid = prxmatch('/^[A-Z]{3}-\d{3}-\d{4}$/', strip(usubjid));
```

B. Exact Visit Label (Week 1, Week 12, …)

```
valid_visit = prxmatch('/^Week\s+[1-9]\d*$/i', strip(visit));
```

👉 **Key insight:** ^ and $ enforce full matches; \b prevents partial matches.

2. EXTRACT — CAPTURING GROUPS

Use capturing groups to pull structured values from text.

A. Extract Week Number from "Week 12"

```
retain re_week;
if _n_=1 then re_week = prxparse('/\bWeek\s+(\d+)\b/i');

if prxmatch(re_week, visit) then
  weeknum = input(prxposn(re_week, 1, visit), 8.);
```

B. Extract USUBJID from Text Blob

```
retain re_usub;
if _n_=1 then re_usub = prxparse('/\bUSUBJID=([A-Z]{3}-\d{3}-
\d{4})\b/i');

if prxmatch(re_usub, blob) then
  usubjid_from_blob = prxposn(re_usub, 1, blob);
```

👉 **Key insight:** (…) captures structure; PRXPOSN retrieves it.

3. TARGET — LOOKAROUNDS FOR PRECISION

Use lookarounds to extract values without including surrounding text.

Extract AESER Value from Key-Value String

```
retain re_aeser;
if _n_=1 then re_aeser = prxparse('/(?<=\bAESER=)[^;]+(?=;)/i');

if prxmatch(re_aeser, kvline) then do;
  call prxposn(re_aeser, 0, start, len);
  aeser = strip(substr(kvline, start, len));
end;
```

👉 **Key insight:** Lookarounds isolate values without extra cleanup.

4. STANDARDIZE — PREPARE TEXT BEFORE MATCHING

Standardization improves pattern reliability and consistency.

A. Normalize Separators

```
retain re_sep;
if _n_=1 then re_sep = prxparse('s/[_\s]+/-/');

clean_id = prxchange(re_sep, -1, strip(raw_id));
```

B. Collapse Whitespace

```
retain re_space;
if _n_=1 then re_space = prxparse('s/\s+/ /');
clean_text = prxchange(re_space, -1, strip(text));
```

C. Filter to Allowed Characters

```
retain re_safe;
if _n_=1 then re_safe = prxparse('s/[^A-Z0-9\-\/=;: ]//i');

clean_text = prxchange(re_safe, -1, clean_text);
```

👉 **Key insight:** Clean inputs produce more reliable matches.

MICRO-CHECKLIST

- Compile patterns once (_N_=1 + RETAIN)
- Anchor validation patterns (^...$)
- Use capturing groups for extraction (PRXPOSN)
- Use lookarounds for key-value parsing
- Standardize text before deriving variables

APPENDIX TAKEAWAY

PRX usage becomes significantly more effective when applied through consistent patterns:

- **Validate** structure
- **Extract** meaning
- **Target** precisely
- **Standardize** inputs

This transforms text handling from ad hoc string logic into a repeatable, pattern-driven workflow.