Paper TU02

An Introduction to Regular Expressions with Examples from Clinical Data

Richard F. Pless, Ovation Research Group, Highland Park, IL

ABSTRACT

Programmers frequently encounter problems when data are collected in “free text” fields. This situation may result from ill-conceived data collection methods or possibly the realization that comment fields contain valuable information. In either case, this loosely structured data may contain valuable insights if it can be converted into a usable format.

This paper discusses how to use the new Perl regular expression functions available in SAS® Version 9 for cleaning text fields and extracting information. Elementary regular expression syntax will be covered as will the functions PRXPARSE, PRXMATCH, PRXCHANGE, and PRXPOSN. Examples from clinical data will be used.

INTRODUCTION

REGULAR EXPRESSIONS AND THEIR USES

Regular expressions are simply ways of describing pieces of text. These descriptions can be used to find and change pieces of text, much in the same way that the arguments to SUBSTR, INDEX and the concatenation operator could. The advantage of regular expressions is the tremendous flexibility that they offer.

In working with clinical data, SAS programmers often encounter data that have not been systematically collected. For example, untrained data entry personnel or under-performing optical character recognition systems can mangle the spelling of medications. The goals of a study could change slightly when investigators realize that useful information has been collected in text fields. Maybe information is expected in a certain format such as “medication dose unit-of-measure” but the data collection mechanism did not enforce this rule. These situations share a common characteristic, namely there are patterns in the text but they are so complex that solutions using other text functions may be impractical if not impossible. To address situations like these, SAS has implemented regular expression functions.

SAS IMPLEMENTATION

SAS has two different sets of functions that implement regular expressions. With SAS Version 8.1 the RX functions were available. They include RXPARSE, RXMATCH, RXCHANGE and others. The RX functions offer “standard” regular expression functionality and they have some special behaviors that are specific to the SAS implementation. In addition to the RX functions, Version 9 includes the PRX functions (i.e. PRXPARSE, PRXMATCH, PRXCHANGE, PRXPOSN, PRXDEBUG, etc.).

The PRX functions offer almost all of the functionality of Perl regular expressions and are generally faster than RX functions. Most importantly, the PRX functions are syntactically very similar to Perl. Beyond allowing programmers familiar with Perl regular expressions to use a familiar syntax, the PRX function novice will have a wealth of documentation and an established user community to further their understanding.

BASICS OF REGULAR EXPRESSIONS

SAS SYNTAX

Unlike other text functions where all of the necessary information is in the function call, regular expressions need to be defined before they can actually be used. Creating regular expressions in SAS is basically a two-step process. First a regular expression is defined using the PRXPARSE function. The variable created by the PRXPARSE function is then used as an argument to other PRX functions.

To define a regular expression the PRXPARSE function is called with the regular expression as its argument:

```
*** create the regular expression only once ***;
retain myREGEX;
if _N_=1 then do;
    myREGEX = prxparse("/Aspirin/");
end;
```

Regular expressions in SAS are enclosed with forward slashes to denote the beginning and end of the expression. For instance, a regular expression that searches for the word “Aspirin” would look like this

```
/Aspirin/
```

This text expression is the only argument to PRXPARSE and is enclosed in quotes.

The PRXPARSE function creates a variable, myREGEX, containing a number that identifies the regular expression. The first regular expression that you create using PRXPARSE is given the number 1, the second is given the number 2 and so on. Once you have created these regular expression id’s, you use them as arguments to the other PRX functions including PRXCHANGE, PRXMATCH and PRXPOSN.

Note, the myREGEX variable is created only once in the DATA step. Without the RETAIN statement and DO loop, the regular expression would be parsed anew with every record. Using the RETAIN statement and DO loop can save processing time and memory.

No single paper could adequately cover the details of regular expressions. However, enough of the basics can be covered in a few pages to enable a novice to unleash some of their power.

POSITION WITH ^

One of first noticeable differences between regular expressions and other text matching functions is the ability to match text in specific positions in a string. In particular, there are specific meta-characters that indicate whether you want your pattern to match the beginning or end of a string. Meta-characters are any characters in your pattern that have special meaning such as positional indicators, wildcards, optional indicators and so forth.

To match patterns at the start of a character field, precede the search pattern with a caret. In this context the caret indicates that the pattern is at the start of the character field being searched. The expression

```
/^Aspirin/
```

would match “Aspirin” or “Aspirin twice daily” but not “Children’s Aspirin”.

WILDCARDS WITH . AND [ ]

Like most text functions, regular expressions support wildcards. Similar to other applications and languages, wildcards match any character. In regular expressions the wildcard to match any single character is the dot, “.”. The regular expression

```
/Celon.in/
```

would match “Celontin” or “Celonsin” or “Celonrin”.

One strength of regular expressions is the ability to match restricted subsets of characters rather than any character like a typical wildcard. Suppose that you have an optical character recognition system that reads physician case report forms and that this system typically reads one physician’s entry for the drug “Celontin” as “Cetontin”. Using other SAS text functions you could rectify the problem by searching for “Cetontin” and replacing the text with “Celontin”. Now suppose that this system also confuses the physician’s “o”s with “e”s and “n”s with “s”s. You have now increased the number of potential misspellings of Celontin from two to 16. One possible solution would be to replace the “l”, “o”, and “n”s with single character wild cards. Such as

```
/Ce...ti./
```

But that is just too broad. It could easily confuse the drug “Celontin” with the drug “Cenestin”. This highlights a very important consideration in creating regular expressions, namely you should always consider what you want to exclude as well as what you want to capture. It is very easy to make expressions too broad.

Regular expressions circumvent this problem with the use of character classes. Character classes restrict the possible values that the pattern will match in a particular position. They are defined by “[“ and “]”. The regular expression

```
/Ce[lt]ontin/
```

will match “Celontin” or “Cetontin” but not “Cesontin”. When you see “[“ and “]” in a regular expression, no matter how many characters these brackets contain, they are only matching a single character (unless they are followed by one of the repetition meta-characters described below.)

Character classes can also support ranges of characters with brackets. Suppose that you wanted to identify any text that starts with a capital letter. The regular expression

```
/^[A-Z]/
```

will match “Morphine” or “Aspirin” but not “morphine” or “aspirin”. The hyphen between the “A” and “Z” in the brackets indicates that you want to capture any character between “A” and “Z”.

Ranges are not limited to characters. Suppose that you wanted to match any digit in a particular field as in the medications listed below:

```
MORPHINE SULFATE SOLUBLE
MORPHINE SULFATE TABLETS SOLUBLE 1 BID
MORPHINE SULFATE SOLUBLE TABLETS PRN
DAILY MORPHINE SULFATE TABLETS SOLUBLE 500 mg
HUMULIN L INJECTION
HUMULIN INJECTION
```

The expression

```
/[0-9]/
```

would identify any listing containing at least one number, namely

```
MORPHINE SULFATE TABLETS SOLUBLE 1 BID
DAILY MORPHINE SULFATE TABLETS SOLUBLE 500 mg
```

One very handy shortcut for matching numbers is using \d in place of [0-9]. \d will match any digit. The expression

```
/\d/
```

would match the same two records:

```
MORPHINE SULFATE TABLETS SOLUBLE 1 BID
DAILY MORPHINE SULFATE TABLETS SOLUBLE 500 mg
```

You can also specify characters that you do not want to have matched in the code. Suppose that you wanted to match any text in a field that started with text and not a number:

```
/^[^0-9]/
```

will match “Aspirin” but not “2 Aspirin per day”. This regular expression will identify any text that starts with any character other than 1,2,3,4,5,6,7,8,9 or 0. Notice that the caret has a different interpretation within the brackets than it does outside of the brackets. Inside of the brackets it tells the parser to match any character except those listed. Outside of the brackets it indicates the beginning of a character string. The range [^0-9] also has a shortcut, \D as you will see in Example #2 below.

REPITITION WITH *, { }, AND THE OPTIONAL INDICATOR ?

Another strength of regular expressions is the ability to specify the number of times that a particular pattern will repeat itself. You can combine the wildcard and the repetition indicator to find words separated by any number of any characters. Remember that the dot is a wildcard and will match any single character. The asterisk indicates that any number of the preceding character are allowed but none are required. Using the medication data from our above example, the expression

```
/MORPHINE SULFATE.*TABLETS/
```

will match any text that contains “MORPHINE SULFATE“ followed by any number of any kind of characters followed by “TABLETS”. Therefore this expression would return

```
MORPHINE SULFATE TABLETS SOLUBLE 1 BID
MORPHINE SULFATE SOLUBLE TABLETS PRN
DAILY MORPHINE SULFATE TABLETS SOLUBLE 500 mg
```

You can also specify the number of times that the pattern may occur in order to be matched by using curly brackets. The regular expression

```
/[0-9]{2,5}/
```

would identify any listing that contained between two and five consecutive numbers. This expression would match

```
DAILY MORPHINE SULFATE TABLETS SOLUBLE 500 mg
```

but not

```
MORPHINE SULFATE TABLETS SOLUBLE 1 BID
```

as it has only one digit.

Unlike most text processing functions, regular expressions also support an optional indicator, “?”. The optional indicator tells the parser that the character immediately preceding the indicator may occur one time or it may not. The expression

```
/HUMULIN L? ?INJECTION/
```

would match

```
HUMULIN L INJECTION
HUMULIN INJECTION
```

Note that two “?”s are used, one for the “L” and another for the trailing space.

ALLOWING A RANGE OF OPTIONS WITH ( ) AND |

Regular expressions also allow you to specify complicated expressions where there is more than one option. The parentheses contain a list of options, each separated by a pipe, "|". Suppose that you are looking a list of medication similar to those above:

```
MORPHINE
MORPHINE SOLUBLE
MORPHINE INJECTION
MORPHINE SULFATE SOLUBLE
MORPHINE SULFATE TABLETS 1 BID
```

Suppose that you want to find morphine that is either in tablet or soluble form. Since you want to match either "TABLET" or "SOLUBLE", you can put them in parentheses separated by a pipe like so

```
/MORPHINE.*(TABLETS|SOLUBLE)/
```

This expression matches

```
MORPHINE SOLUBLE
MORPHINE SULFATE SOLUBLE
MORPHINE SULFATE TABLETS 1 BID
```

but not

```
MORPHINE
MORPHINE INJECTION
```

This brief overview should help you to follow more complicated examples.

EXAMPLE #1: A SIMPLE SEARCH

Suppose that you have a dataset containing patient medications. You are asked to find every record with the medication Kaldera. Unfortunately, your data are rife with misspelled medications. The following example shows how to use regular expressions to find all records that contain some variation of "Kaldera" in the drugname field.

```
*** create the regular expression only once ***;
retain kalderaREGEX;
if _N_=1 then do;
    *** use "i" option to have REGEX ignore case ***;
    kalderaREGEX = prxparse("/k[ea]l[dt][ea]ra/i");
end;
```

This code creates a regular expression object called kalderaREGEX that identifies some common misspellings of the drug Kaldera. Note the use of "i" after the second slash in the regular expression. This tells the parser to ignore case when matching the expression. The variable kalderaREGEX is then used as the first argument to the function PRXMATCH with the field to be searched as the second argument.

```
*** create a flag by changing values greater than 1 to 1;
kalderaFLAG = min((prxmatch(kalderaREGEX,drugname),1);
```

PRXMATCH returns the position where the pattern is found in the text. A flag is created anytime the pattern is found in the field drugname using the MIN function.

One possible solution without regular expressions would require spelling out every possibility.

```
*** one possible solution without using REGEX ***;
if upcase(drugname) in ("KELDERA" "KALDERA" "KELTERA" "KALTERA"
  "KELDARA" "KALDARA" "KELTARA" "KALTARA") then kalderaFLAG=1;
else kalderaFLAG=0;
```

As you can see, allowing two possibilities for each of three characters would require you to test eight different possibilities. Not only is this approach less efficient, but checking your code becomes an onerous task as you broaden the possible spellings.

EXAMPLE #2: VALIDATING THE FORMAT OF A FREE TEXT FIELD

Suppose that you are asked to develop a report that lists the medications of each patient at the start of a study. The medications are captured in a free text field that contains the product name and dose. Prior to reporting the information, you are asked to find any records that are not in this format so that complete information can be requested from the investigator.

You have a sample of the data that looks like this:

```
Hydro-Chlorothiazide 25.5
Ziagen 200mg
Zerit mg
Insulin 20 cc
Dapsone 100 g
Kaletra 3 tabs
```

After a few minutes of testing you develop the following code:

```
*** create the regular expression only once ***;
retain fmtre;
if _N_=1 then do;
    fmtre=prxparse("/^\D*\d{1,4}\.?\d{0,4} ?(tabs?|caps?|cc|m?g)/i");
end;
```

The regular expression in this code looks for records that start with a group of non-digits (i.e. characters, special characters and white spaces). These non-digits are followed by a number. The number must contain at least one digit prior to an optional decimal point which is followed by up to four more digits. "d{1,4}" indicates that between one and four numbers are to be matched. The "\.?" tells the parser to match an optional period. The backslash tells the parser that the dot, ".", should be matched and is not a wildcard. "\d{0,4}" indicates that the optional period may be followed by up to four more digits. Finally the pattern ends with one of six units of measure: "tab", "tabs", "cap", "caps", "cc", "mg", or "g".

To keep those records that do not match this pattern you will look for those records where PRXMATCH returns a zero.

```
*** sub-setting if statement to keep poorly formatted data ***;
if prxmatch(fmtre,medication)=0;
```

This code will keep the improperly formatted records from the above list, namely:

```
Hydro-Chlorothiazide 25.5
Zerit mg
```

The first record, "HYDRO-CHLOROTHIAZIDE 25.5", was included because it did not have any unit of measure. The third record, "Zerit mg", was included because it did not have any digits.

You will notice that the above regular expression was somewhat more complicated than those described previously. Complex regular expressions do not have to be built in a single step. The next example shows you how to use SAS to develop a complicated search pattern.

EXAMPLE #3: DEVELOPING A COMPLICATED SEARCH PATTERN

Suppose that you have a dataset that contains information related to serious adverse events associated with a medication. After seeing the medication used in practice, researchers believe that there is some relationship between the drug, the occurrence of headaches and nausea, and the onset of serious adverse events. Since this relationship was not hypothesized until after the data were collected, this information was not specifically requested on the data collection forms. Thankfully, the investigators documented the details surrounding the onset of adverse events copiously in a comment field.

You are asked to identify any records that mention both headaches and nausea in the comment field. After looking at a handful of records you see that there is a tremendous amount of variation in the way this information was collected. A sample of the data is shown below:

```
Headache and nausea
Nausae and headache
Patient reported headache and nausea
Pt. Rptd. Head ache and nausea
Naus. And hdache reported
Pt reported headache at admission; patient later reported nausea.
```

You can see that the words appear in different order and include abbreviations and misspellings. This problem is more complicated than simply searching for a misspelled word.

Rather than developing a single regular expression to handle this difficult match, you can start with simple patterns and combine them to develop more complicated ones. First, create regular expressions to match headache and nausea in their many possible forms.

```
*** create the regular expression only once ***;
retain re1;
if _N_=1 then do;

    ha_regex="he?a?d.?[ -]?(ache)";
    n_regex="na?u?a?se?a?";
```

Note that the PRXPARSE function was not called yet. The code simply defined two text variables each containing a regular expression, one to match “headache” and the second to match “nausea”. The next step is to combine these expressions.

Since either pattern could occur first, you need to develop two more regular expressions to handle both possible word orders. To do this, create another pair of text fields, concatenating our regular expressions with the || operator and separating them with “.*”. The “.*” will allow the patterns to be separated by any kind of text.

```
    n_ha_regex=n_regex || ".*" || ha_regex;
    ha_n_regex=ha_regex || ".*" || n_regex;
```

Finally, you are ready to create the complicated regular expression that accounts for a variety of spellings for each word, allows them to occur in any order, and allows them to be separated by any number of any type of characters.

```
    re1=prxparse("/(" || n_ha_regex || ")|(" || ha_n_regex || ")/i");
end;
```

Using re1 as an argument to PRXMATCH, you can create flags to identify patients reporting both headache and nausea.

```
*** create flag for presense of headache and nausea ***;
headache_n_nausea_flag=min(prxmatch(re1,comments),1);
```

This solution was a lot more complicated than the problem required. The same conclusion could have been reached with a combination of simple regular expressions and “if-then-else” logic.

```
*** An easier alternative ***;
*** create the regular expression only once ***;
retain re1 re2;
if _N_=1 then do;
    re1 = prxparse("/he?a?d.?[ -]?(ache)/i");
    re2 = prxparse("/na?u?a?se?a?/i");
end;

*** a simple if, then, else statement ***;
if prxmatch(re1,comments)>0 and prxmatch(re2,comments)>0 then headache_n_nausea_flag=1;
else headache_n_nausea_flag=0;
```

However this approach to building complex regular expressions from a number of smaller ones can be very useful.

EXAMPLE #4 EXTRACTING TEXT

Again suppose that you are going to comb through a comment field in an adverse events data set similar to the one described in Example #3 above. However, this time you want to see what the patients are reporting to the investigators. A sample of the data looks like this:

```
Patient reported headache and nausea.  MD noticed rash.
Pt. Rptd. Backache.
patient reported seeing spots.
Elevated pulse and labored breathing.
Headache.
```

First, create a regular expression

```
*** create the regular expression only once ***;
retain re1;
if _N_=1 then do;
    re1 = prxparse("/(reported|rptd?\.?)(.*\.)/i");
end;
```

This regular expression should capture the word “reported” or the abbreviations “rpt.” or “rptd.” until a period is reached. Notice that the second set of parentheses does not have a pipe to identify options. This is a clue that you are using the expression to capture text for the PRXPOSN call routine. The PRXPOSN call routine allows you to retrieve the starting position and length of a sub-string in a regular expression if that sub-string is enclosed in parentheses. Note how this differs from the PRXPOSN function which simply extracts the text.

In this case, you wanted to capture everything after “reported” until the end of the sentence. This was accomplished by enclosing “.*\.” in parentheses. The “.*” indicates any number of any character can follow. The “\.” indicates that the last character in the pattern is a literal period. Remember that the backslash tells the parser that the dot, “.”, should be matched and is not a wildcard.

```
*** only call the PRXPOSN function if the record contains the text you need ***;
if prxmatch(re1,comments) then do;
    *** call the PRXPOSN routine function ***;
    pt_comments=prxposn(re1,2,comments);
end;
```

The PRXPOSN function above was called using a regular expression that was created previously, re1. The second argument, 2, tells the function to use the second sub-string that was created (i.e. the second set of parentheses). The third argument is the text field that you want to extract data from. The IF PRXMATCH condition ensures that the PRXPOSN function is only called when the pattern is found.

The new field would contain the following values for the first three records:

```
headache and nausea
Backache
seeing spots
```

The last two records would be empty fields.

Note that embedded parentheses are included in the count when you are selecting a field. For example, if you were wanted to extract the third field in the expression

```
/(field1 (field 2)) (field3)/
```

You would use the argument “3” as the second argument to the PRXPOSN function. The easiest way to find which pair of parentheses you want to use for extraction is to count the number of open (or left) parentheses.

This was a very simple example using very straightforward data. Extracting large pieces of text from comment fields can be a very difficult task owing to haphazard grammar, misspellings, and creative abbreviations. Notice that the last record contained an implicit patient reported symptom, “headache”. Anyone can infer that the patient reported this symptom, but our regular expression could not identify that record. This is just one example of the kinds of problems you may encounter when you are trying to pull meaningful data from comments beyond the presence of a few simple words. The problems associated with using regular expressions to extract large pieces of unformatted data often outweigh the benefits.

EXAMPLE #5: STANDARDIZING SPELLING

Suppose that you are not interested in simply identifying records for analysis but you want to correct data for reporting. In particular you are asked to standardize various spellings and abbreviations for “Multi-vitamin”. The function PRXCHANGE can be used to replace a matched pattern with another piece of text. A sample of the data that you are interested in changing looks like this:

```
multivitamin
multi-vitamin
multi-vita
multivit
multi-vit
multi vitamin
```

You are asked to replace each of these occurrences with “Multi-vitamin”.

Creating a regular expression for use in a PRXCHANGE function is slightly different than creating a regular expression for use in a PRXMATCH function.

```
*** create the regular expression only once ***;
retain mvitREGEX;
if _N_=1 then do;
    *** create the regular expression id ***;
    mvitREGEX = prxparse("s/multi[- ]?vit.*/Multi-vitamin/");
end;
```

Notice that the expression starts with an "s" and there are three forward slashes rather than two. The "s" designates that the regular expression will be used in a substitution. The regular expression will search for the pattern between the first two forward slashes and replace it with the text found between the second and third forward slashes. Any time the pattern

```
/multi[- ]?vit.*/
```

is found, it will be replaced with "Multi-vitamin".

The variable mvitREGEX is then used as the first argument in the call routine PRXCHANGE

```
*** Using the mvitREGEX id created above***;
call prxchange(mvitREGEX, -1, drugname)
```

The second argument tells the function how many times to change the pattern once it's found. -1 indicates that the pattern should be changed at every occurrence. `The third argument to the PRXCHANGE is the field name. All of the sample data listed above would be changed to "Multi-vitamin". This looks much better than the varieties of spellings and abbreviations found in the raw data.

CONCLUSION

With a small investment in learning the appropriate syntax, regular expressions can be a powerful tool to extract simple information from text fields, correct misspellings, or verify formats. Knowing the many possible format of the text that you are trying to describe is only half the battle. The challenge of regular expressions is that you need to understand the patterns that you want to match *as well as those patterns that you do not want to match*. With a little practice and a lot of quality assurance, regular expressions can help you gather useful information from free text fields. For more detailed information and additional examples check out www.sas.com, www.perl.com, or the Perl usenet.

REFERENCES

Friedl, J.E. (1997) *Mastering Regular Expressions*, O'Reilly & Associates

Secosky, Jason "The DATA Step in Version 9: What's New?" Proceedings of the Twenty Seventh Annual SAS Users Group International Conference. 2002 http://support.sas.com/rnd/base/topics/datastep/dsv9-sugi-v2.pdf

"PRX Function Reference" http://support.sas.com/rnd/base/topics/datastep/perl_regexp/regexp2.html (4Jan2004)

CONTACT INFORMATION

Richard Pless
Ovation Research Group
600 Central Avenue
Highland Park, Illinois 60035
rpless@ovation.org