PharmaSUG 2014 - Paper TT08

Investigating the Irregular: Using Perl Regular Expressions

Peter Eberhardt, Fernwood Consulting Group Inc., Toronto, ON, Canada

ABSTRACT

A true detective needs the help of a small army of assistants to track down and apprehend the bad guys. Likewise, a good SAS® programmer will use a small army of functions to find and fix bad data. In this paper we will show how the small army of regular expressions in SAS can help you.

The paper will first explain how regular expressions work, then show how they can be used with CDSIC.

INTRODUCTION

The Baker Street irregulars were a group of street urchins whom Sherlock Holmes paid to search for clues in the noisy streets of London. Holmes knew that amid all the noise there was information. He also knew that to access the information he sometimes had to employ what appeared to be random agents – a group of street urchins; however, when his irregulars were given the appropriate directions, they provided valuable information. As a SAS programmer you will find Perl Regular Expressions (PRX) can play a similar role for you; the apparent random collection of symbols can improve the signal to noise ratio in the unstructured text fields you encounter.

This paper will start with a brief introduction to Perl Regular Expressions and use this to build some simple but useful regular expressions for CDISC ISO date processing.

PERL REGULAR EXPRESSIONS

Unstructured text fields in our data may contain valuable information that, when extracted, validated, and quantified, can help to improve our understanding of the process we are studying. The problem with unstructured text data is that they were often “free hand” which leads to inconsistencies and errors; for example a dosage may be recorded as mg, MG, m.g. etc. As the human reader we understand these are all the same however to SAS these are all different.

If there are only a few records we could manually change the ‘incorrect’ spellings to the ‘correct’ spelling. Although this is possible it is not advisable. First, you may introduce yet another incorrect spelling, and second you may risk invalidating your results because you tampered with the source data.

Since you know it is unwise to tamper with your data, you can opt to let SAS help you through its many character functions such as **scan()**, **find()**, **input()**, **substr()** etc.. When you have small unstructured text fields and the variety of searches you need to perform is small, this is an effective and proven approach. However, even looking at the ‘mg’ example we can see that amount and complexity of our code will grow quickly. To help control the size and complexity of out SAS code we can turn to function dedicated to character matching – the Perl Regular Expression functions, commonly called the **PRX** functions. Although the PRX functions greatly improve our ability to extract information from the noise in the string, they do come at a cost – the complexity of several lines of SAS code with traditional character functions becomes compressed into one character string, the regular expression. This apparent complexity has kept many SAS programmers from using PRX functions. Mastering regular expressions is not an easy task, nor is it a Herculean task. It is a task that will require some patience and a large dose of attention to detail. Although this paper cannot provide you with patience or with attention to detail, it will provide you with valuable tips on getting your regular expressions in order.

The paper will first introduce the concept of a regular expression. This will be followed by a listing of the PRX functions along a brief explanation of how they will help you. The goal here is to give you an idea of the broad capabilities of the PRX functions; before you get lost in the details of building regular expressions we want you to have an understanding of what you can do with them. Using a twist on an old expression, we want you to see the forest before you get lost amongst the trees.

REGULAR EXPRESSIONS

What is a regular expression? A regular expression is string of 'normal' (letters, numbers) characters coupled with some special meta-characters that, when applied to another text string, provides a concise and flexible means to "match" (specify and recognize) strings within the text, such as particular characters, words, or patterns of characters. There are several flavours of regular expressions; in this paper we are talking about Perl Regular expressions as implemented by SAS.

"**m/hello/**" is a regular expression. The components are:

m/: tells the regular expression engine we are building a match string. In SAS, the m is optional.

hello: tells the regular expression engine we want to match the literal string ***hello***.

/: closes the opening match command.

Although this is a valid regular expression, it is not a particularly useful one, unless the characters ***hello*** have particular value to your study.

"**m/ ^(((19|20)(([0][48])|([2468][048])|([13579][26]))|2000)[\-](([0][13578]|[1][02])[\-]([012][0-9]|[3][01])|([0][469]|11)[\-]([012][0-9]|30)|02[\-]([012][0-9]))|((19|20)(([02468][1235679])|([13579][01345789]))|1900)[\-](([0][13578]|[1][02])[\-]([012][0-9]|[3][01])|([0][469]|11)[\-]([012][0-9]|30)|02[\-]([012][0-8])))$/**"

is another regular expression that uses a mix of normal and meta-characters; we will revisit this expression later.

The mix of regular expressions you build will probably fall somewhere between these two in terms of complexity. Before we examine regular expressions and their building blocks, meta-characters, we will introduce the functions and call routines that apply regular expressions to the data. As you look at these functions you will see some of the meta-characters that are used to build regular expressions.

PRX FUNCTIONS

SAS has implemented five functions and six call routines to facilitate string matching and updating. The following tables list the functions and call routines along with a short description and a code snippet showing a call to the function.

THE PRX FUNCTIONS

Function	Description / Example call
PRXPARSE	Compiles a Perl regular expression (PRX) that can be used for pattern matching of a character value. Should be called only once per expression. Input: a string with a regular expression Output: a pattern identifier to be used as input to other PRX functions/call routines
	`regEx = PRXPARSE("/(\d{4})\-(\d{2})\-(\d{2})/");` `/* look for a date in YYYY-MM-DD format */`
PRXMATCH	Searches for a pattern match and returns the position at which the pattern is found. ***input***: 1. A pattern identifier from PRXPARSE ***or*** a string with a regular expression. 2.The string to search ***output***: the position at which the first occurrence of the pattern starts. If pattern is not found, 0 (zero) is returned.

	`charVar = "Visit Date: 2013-05-13 @ 2:30";`
	`regMatch = PRXMATCH(regEx, charVar);`
	or
	`regMatch = PRXMATCH("/\d{4}\-\d{2}\-\d{2}/", charVar);`

Function	Description / Example call
PRXPOSN	Returns the value for a capture buffer. The PRXPOSN function uses the results of PRXMATCH, PRXSUBSTR, PRXCHANGE, or PRXNEXT to return a capture buffer. A match must be found by one of these functions for PRXPOSN to return meaningful information. ***input***: 1. A pattern identifier from PRXPARSE 2. The ***capture buffer*** - a number between 1 and the number of open parentheses in the regular expression. 3. The string to search ***output***: the position at which the first occurrence of the pattern starts. If pattern is not found, 0 (zero) is returned,

	`charVar = "Visit Date: 2013-05-13 @ 2:30";`
	`regEx = PRXPARSE("/(\d{4})(\-)(\d{2})(\-)(\d{2})/");`
	`regMatch = PRXMATCH(regEx, charVar);`
	`YY = PRXPOSN(regEx, 1, charVar);/* 1st open paren is year (\d{4}) */`
	`MM = PRXPOSN(regEx, 3, charVar);/* 3rd open paren is month(\d{2}) */`
	`DD = PRXPOSN(regEx, 5, charVar);/* 5th open paren is day (\d{2}) */`

PRXCHANGE	Performs a pattern match change. ***input***: 1. A pattern identifier from PRXPARSE or a string with a regular expression. 2. A number tell PRXCHANGE how many search/replace in the string; a value of -1 will cause the search/replace to cover the entire sting. 3. The string to search ***output***: the string with the replacements.

	`regEx = PRXPARSE("s/(\d{4})(\-)(\d{2})(\-)(\d{2})/$1\/$3\/$5/");`
	`/* replace '-' with "/" only in the date part */`
	`charVar = "Visit Date: 2013-05-13 @ 2:30";`
	`newVar = PRXCHANGE(regEx, -1, charVar);`

PRXPAREN	Returns the last bracket match for which there is a match in a pattern.

	`regEx = PRXPARSE("/(study)|(exam)|(results)/i");`
	`charVar = "If you study you will get better exam results";`
	`regMatch = PRXMATCH(regEx, charVar);`
	`paren =PRXPAREN(regex);`

THE PRX CALL ROUTINES

Call Routine	Description / Example call
CALL PRXCHANGE	Performs a pattern match change. ***input***: 1. A pattern identifier from PRXPARSE. 2. A number tell PRXCHANGE how many search/replace in the string; a value of -1 will cause the search/replace to cover the entire sting. 3. The string to search ***The remaining arguments are optional. These are returned by the call*** 4. Specifies a character variable in which to place the results of the change to input string. If specified, the input string is not modified. 5. The length of the output string. 6. A truncation flag. 1 – the newstring was truncated, 0 – the new string was not truncated. 7. The number of changes made.

	`length text $46 newText $ 66;`
	`regEx = PRXPARSE('s/([135])( times)/a few$2/');`
	`/* replace 1, 3, or 5 with 'a few' */`
	`text = '1 times 2 times 3 times 4 times ';`
	`call PRXCHANGE(regEx, -1, text, newText, size, trunc, changes);`

CALL PRXNEXT	Returns the position and length of a substring that matches a pattern, and iterates over multiple matches within one string. ***input***: 1. A pattern identifier from PRXPARSE. 2. A numeric variable that specifies the position at which to start the pattern matching in source. If the match is successful, CALL PRXNEXT returns a value of position + MAX(1, length). If the match is not successful, the value of start is not changed. 3. is a numeric that specifies the last character to use in source. If stop is -1, then the last character is the last non-blank character in source 4. The string to search ***The remaining arguments are returned by the call*** 5. A numeric variable with the position in source at which the pattern begins. If no match is found, CALL PRXNEXT returns zero. 6. is a numeric variable with a returned value that is the length of the string that is matched by the pattern. If no match is found, CALL PRXNEXT returns zero.

Call Routine	Description / Example call

	`data _null_;`
	`  regEx = prxparse('/[dlh]og/');`
	`  text = 'The farm has a hog, log, and a dog!';`
	`  start = 1;`
	`  stop = length(text);`
	`  call PRXNEXT(regEx, start, stop, text, position, length);`
	`  do while (position > 0);`
	`  found = substr(text, position, length);`
	`  put found= position= length=;`
	`  call PRXNEXT(regEx, start, stop, text, position, length);`
	`  end;`
	`run;`

CALL PRXPOSN	Returns the start and length of a capture buffer. ***input***: 1. A pattern identifier from PRXPARSE 2. The ***capture buffer*** - a number between 0 and the number of open parentheses in the regular expression. ***The remaining arguments are returned by the call*** 3. The position within the string the capture buffer was found. 4. (***optional***) A numeric variable with the length of the capture buffer text

	`regEx = PRXPARSE("/(\d{4})(\-)(\d{2})(\-)(\d{2})/");`
	`charVar = "Visit Date: 2013-05-13 @ 2:30";`
	`regMatch = PRXMATCH(regEx, charVar);`
	`if regMatch then do;`
	`  CALL PRXPOSN(regEx, 1, position, length);`
	`  yy = substr(charVar, position, length);`
	`  put regMatch= charVar=;`
	`  put YY= ;`
	`end;`

CALL PRXSUBSTR	Returns the position and length of a substring that matches a pattern. ***input***: 1. A pattern identifier from PRXPARSE. 2. The string to search ***The remaining arguments are returned by the call*** 3. A numeric variable with the position of the first match within the search string. 4. (***optional***) A numeric variable with the length of the matching sub-string.
	`  regEx = prxparse('/Dr|Doctor/');` `  call prxsubstr(regEx, 'find Doctor Who', position, length);`
CALL PRXDEBUG	Enables Perl regular expressions in a DATA step to send debug output to the SAS log. ***input***: 1. Numeric. 1 – turn debugging on, 0 – turn debugging off.
	`call PRXDEBUG(1);`

Call Routine	Description / Example call
CALL PRXFREE	Frees unneeded memory that was allocated for a Perl regular expression ***input***: A regular expression identifier
	`call PRXFREE(regEx);`

As can be seen above, all of the functions/call routines can take a regular expression identifier compiled using PXPARSE() function. In addition some of the functions (PRXMATCH, PRXCHANGE) can take the regular expression string instead; this allows these functions usable in PROC SQL. Also note that PRXPARSE should only be called once for each regular expression to be compiled; you will commonly see DATA steps with the following structure:

```
DATA getRegEx;
    retain regEX 0; /* retain the value across data step iterations */
    if _N_ = 1
    then
      do;
         regEx = PRXPARSE(“/regular expression/”);
         if missing(regEx) /* verify it compiled */
         then
           do; /* since it did not compile, print a message and stop */
              put “ERROR: The regular expression could not compile.”;
              stop;
           end;
      end;
/*  more program statements follow */
```

Notice the error check (if missing(regEx)); if the regular expression does not compile then its subsequent use in the data step will lead to errors when the expression is used in other PRX expressions.

SAS has added an option to simplify this structure - ***/o***; if this option is placed at the end of the regular expression SAS will compile the expression once and subsequent calls to PRXPARSE() will not cause a recompile but simply return the regular expression identifier. Needless to say this simplifies the programming and should improve performance by removing a logic test on each iteration of the DATA step:

```
DATA getRegEx;
    regEx = PRXPARSE(“/regular expression/o”);
    if missing(regEx) /* verify it compiled */
    then
      do; /* since it did not compile, print a message and stop */
        put “ERROR: The regular expression could not compile.”;
        stop;
      end;
/*  more program statements follow */
```

If you have a very large datasets to process you could consider wrapping the code in a macro having two data steps, one to test the regular expression compiles, and the second to process the data using the assumption the regular expression will compile; when processing millions of rows removing some IF statements can improve performance.

```
/* note this is NOT production ready. Use at your own risk */
%MACRO runRegEx(regex=);
    DATA _null_;
        regEx = PRXPARSE(“/&regex./”);
        if missing(regEx) /* verify it compiled */
        then
          do; /* since it did not compile, print a message and stop */
            put “ERROR: &regex”;
            put “ERROR: The regular expression could not compile.”;
            call symput('regExOK', put(0, 1.));
```

```
          end;
        else
          do; /* since it did compile, set the OK flag */
            call symput('regExOK', put(1, 1.));
          end;
    run;
%if &regExOK = 1
%then
  %do
     data getRegEx;
          retain regEX 0; /* retain the value across data step iterations */
          regEx = PRXPARSE("/&regex./o"); /* note the /o option */
         /* we do not test since it compiled OK in DATA _null_ */
/*  more program statements follow */
     run;
  %end;
%mend;
```

Obviously great care must be taken to ensure the regular expression being passed into the macro does not cause unwanted side effects; an approach like this should be taken with caution.

In this paper we will look primarily at searching (PRXMATCH()) and extraction (PRXPAREN(), PRXPOSN). However, before we can put the functions to work we will have to learn how to build regular expressions using meta-characters.

META-CHARACTERS

As we saw above "**m/hello/**" is a regular expression. The components are:

m/: tells the regular expression engine we are building a match string. In SAS, the m is optional.

hello: tells the regular expression engine we want to match the literal string ***hello***.

/: closes the opening match command.

This simple expression will search a character variable for the string ***hello***. It will find hello in:

- hello world
- Peter, say hello to the room
- The following does not sound like food: hello-pudding

However, it would not find hello in:

- Hello world
- Peter, say Hello to the room

In these latter two cases hello does not match because of differences in the capitalization (or case) of the words; the regular expression was explicitly instructed to match ***hello*** (all lower case). We could enumerate all of the variants of casing for hello (e.g. hello, HELLO, Hello, HEllo etc) which, even for a five letter word would be a long list, or we can apply the meta-character ***/i*** to tell the regular expression (regex) engine to ignore case when comparing. From this you can see some simple meta-characters can shorten our expression; on the flip side, they also make the expressions cryptic and potentially hard to build, read, and debug. In this section we will review a few of the meta-characters. See the SAS online help for a complete list

The following table has some of the common meta-characters.

Meta-character	Description/Example
[…]	(square brackets) specifies a character set that matches any one of the enclosed characters. The characters can be enumerated, or a range can be specified.
	[a-z] – match any lower case letter [A-Z] – match any upper case letters [a-zA-Z] – match any letter, lower or upper case [aeiou] - match a vowel
[^…]	specifies a character set that is NOT to be matched. The characters can be enumerated, or a range can be specified.
	[^a-z] – match any character EXCEPT a lower case letter. This would match upper case letters, digits, punctuation etc. [^aeiou] would match any character not a vowel,
\d	any single digit number. This is equivalent to [0-9].
\D	matches a non-digit character that is equivalent to [^0−9].
\w	matches a word characters (letters, alpha-numerics, underscore)
\W	matches a non-word characters (non-letters, non-alpha-numerics), anything not matched by \w
\s	matches a white space character (space, tab, formfeed etc.).
\S	matches a character that is not white space.
\b	matches a word boundary. A word boundary is the location between a word character (\w) and a non-word character (\W)
	/or\b/ • would match the or in “motor” • would match the word or in “this or that” • would not or in “motorcar”
\B	matches a non-word boundary
	/or\B/ - • would not match the or in “motor” • would not match the or in “this or that” • would match the or in “motorcar”

^	matches the position at the beginning of the string
	/^A/ • would match the first A in “A long time ago” • would not match the first a in “a long time ago” • would not match the A in “I got an A in English”
$	matches the position at the end of the string
	/er$/ • would match the final er in “This is super” • would not match er in “superman” /^error$/i • would match any line that just had the word error. The /I makes the match case insensitive
.	the period. Matches any single character except the new line
	/1.2/ • would match digit 1 and digit 2 separated by any character EXCEPT a new line (\n) /[.\n]/ • would match any character, including the new line
(…)	parentheses surrounding a pattern. This specifies a grouping and creates a capture buffer.
	/(\d4)-(\d2)-(\d2)/ • matches a sting that looks like an ISO date, 4 digits, 2 digits, 2 digits separated by “-“ • the first four digits are capture buffer 1, the second two are capture buffer 2 and the last two are capture buffer 3.
(?:…)	creates a grouping but does not create a capture buffer
\n	where n is a number, refers to capture buffer n
|	creates an OR condition.
	/(b|c)at/ • matches either “bat” or “cat”. Capture buffer 1 would have the “b” or “c”
****	used to “escape” other meta-characters
	/\\/ • matches “\” /\(/ • matches “(”
*****	matches the preceding sub-expression zero (0) or more times
	/fo*/ • this means match an “f” followed by zero or more letter “o” • matches stings “f” “fo” “foo” “food” “fooooooood”

+	matches the preceding sub-expression one or more times
	/fo+/ • this means match an “f” followed by at least oneletter “o” • matches stings “fo” “foo” “food” “fooooooood” • does not match string “f”
?	matches the preceding sub-expression zero or one time
{n}	matches the preceding sub-expression at least n times; n is a positive integer
	/fo{2}/ **1.** matches “f” followed by at least two “o”s
{n,}	matches the preceding sub-expression at least n times; n is a positive integer. Spaces are NOT allowed between the comma and the number.
	/fo{2,}/ matches “f” followed by at least two “o”s
{n,m}	matches the preceding sub-expression at least n times and at most m times; n and m are positive integers and m >= n. Spaces are NOT allowed between the comma and the numbers.
	/fo{1,3}/ matches the first three “o”s in “foooood”.
(?=…)	A positive look ahead.
	/\w*\s\w*\s(?=MD)/ 2. matches a two sets of word characters separated by a space, followed by the string “MD”. The pattern in the (?=) grouping is not included in the final match.
(?!...)	A negative look ahead
	/\w*\s\w*\s(?!MD)/ 3. matches a two sets of word characters separated by a space but not followed by the string “MD”. The pattern in the (?!) grouping is not included in the final match.
(?<=…)	A positive look behind
	/(?<=Mr.)\s\w*\s\w*/ 4. matches a two sets of word characters separated by a space, these sets of characters must be preceded by the string “Mr.”. The pattern in the (?<=) grouping is not included in the final match.
(?<!...)	A negative look behind
	/(?<!Mr.)\s\w*\s\w*/ matches a two sets of word characters separated by a space, these sets of characters must not be preceded by the string “Mr.”. The pattern in the (?<!) grouping is not included in the final match.

There are more meta-characters than these, but with this set we can start building expressions.

How do you build a regular expression? First, you have to know your data and understand you have one or more free form text columns that contain valuable information. Because the columns are free form text you cannot use SAS formats to pull out the nuggets you need. Moreover, one of the side effects of free form text is free form inconsistencies; that is, in study notes the word “patient” may have any number of creative spellings and/or abbreviations. Your first task is to look for patterns in the text; once you have found patterns then you can start to build you regular expressions. Work at gaining small successes. Write, test and verify short patterns; do not try to capture all the variety of one pattern in one expression initially. Even in the end you may find it more efficient of your time to have several expressions search through the free form text to find the permutations of “patient” than to have one catch-all expression.

Since a common problem we encounter in study data is inconsistent dates, we will look at building regular expressions to find dates and durations in text. In particular we will look for ISO 8601 consistency.

ISO 8601 DATES

ISO 8601 (ISO) specifies a standard for representing dates, datetimes, times, and durations; for details on ISO Dates see Eberhardt et al. [2013]. Unlike SAS dates which are numbers, ISO specifies that dates must be characters; SAS does provide numerous informats and formats to read and write ISO dates.

Here it will suffice to just lay out some of the standard layouts, then build some expressions to match them in free form text. If the ISO dates are in regular columns in the input data then you should be able to use the SAS built-in informats to read them, however, if you have dates in free text columns you will have to first locate the date, then process it with SAS.

The CDISC SSTM Implementation Guide specifies that dates and times be stored according to the ISO 8601 format for calendar dates and times of day; the guide provides the following template:

- YYYY-MM-DDThh:mm:ss
  - [YYYY] four-digit year
  - [MM] two-digit representation of the month (01-12, 01=January, etc.)
  - [DD] two-digit day of the month (01 through 31)
  - [T] (time designator) indicates time information follows
  - [hh] two digits of hour (00 through 23)
  - [mm] two digits of minute (00 through 59)
  - [ss] two digits of second (00 through 59)

This is not a complete ISO date and time. ISO also adds:

- YYYY-MM-DDThh:mm:ss,ffff±hh:mm
  - [ffff] Fractions, size to be determined by the parties exchanging data
  - [±] (time zone indicator) plus or minus to indicate a UTC offset follows
  - [hh] two digits of hour (00 through 23)
  - [mm] two digits of minute

A fully formatted ISO 8601 datetime looks like:

- 2013-05-13T14:30:00,0+06:00

We see the date is in most significant to least significant order, that is year before month before day etc.. The time is separated from the date with a “T” and the time must be in the 24 hour format. Finally, the offset from Universal Coordinated Time (UTC) is added. The hypen (-) is used to separate the date components and the colon (:) is used to separate the time components and if there is a fraction of a second, the comma (,) is the delimiter. In this example we have 13th day of May, 2013 at exactly 2:30 in the afternoon in a time zone 6 hours behind UTC. This certainly removes any ambiguity as to the point in time!

If all the dates came fully formed we could use a simple regular expression to identify them and ultimately extract them from text fields. However, the ISO standard also allows for dates that are not fully formed; in fact it has planned for incomplete dates and rules for formatting incomplete dates. There are two basic forms of incomplete dates, those that are right truncated, that is missing lower order elements, and the those with omitted components.

An ISO that is right truncated is missing one or more least significant elements. Examples of right truncated ISO dates are:

Date	Truncation
2013-05-13T14:30	Missing Seconds
2013-05-13T14	Missing Minutes and Second
2013-05-13	Missing All Time
2013-05	Missing Day
2013	Missing Month

More problematic are dates with missing components. Where a right truncated date will be missing lower order elements, a date with missing components can be missing any one or components (year, month, day, hour, minute, second) of the date; when a component is missing from the date string it must be replaced by a hypen (-). Examples of ISO dates with missing elements are:

Date	Missing compenents
2013-05-13T-:30	May 14, 2013 at 30 minutes after an unknown hour
2013-05--	an unknown day in May 2013
--05-13	May 13 of an unknown year
----13T13:-:-	the 13th day of an unknown year and month at 1pm unknown minute and unknown second

Before we try to create regular expressions to handle incomplete dates, let's look at a regular expression that can handle a compete ISO date:

- "m/(\d{4})(\-|\/)(\d{2})(\-|\/)(\d{2})T(\d{2})(\:)(\d{2})(\:)(\d{2})/i**"**

Decomposing this we have:

- (/d{4}) – look for four numbers (YYYY)
- (\-|\/) – look for the separator (hyphen or slash). Although the hyphen is the standard, by agreement of all parties in an exchange a slash can be used
- (\d{2}) – look for two numbers (MM)
- (\-|\/) – look for the separator (hyphen or slash)
- (\d{2}) – look for two numbers (DD)
- T – look for the literal T, the time separator
- (\d{2}) – look for 2 numbers (HH)
- (\:) – look for the separator (:)
- (\d{2}) – look for 2 numbers (MM)
- (\:) – look for the separator (:)
- (\d{2}) – look for 2 numbers (SS)
- i – make the search case insensitive (allows T or t for time separator)
- the brackets around the date and time elements create capture groups so we can extract the components if needed. The first set of brackets (\d{4}) will let us capture the four digits of the yeat, the second set of brackets (\-|\/) will let us capture the separator between year and month and so on.

This is an effective expression but it will detect "false positives". For example:

- 2013-02-31T12:30:20 (Feb 31)

- 9999-99-99T99:99:99

To correct this we could build a more complex regular expression. Let's revisit the expression introduced earlier:

- "**m/ ^(((19|20)(([0][48])|([2468][048])|([13579][26]))|2000)[\-](([0][13578]|[1][02])[\-]([012][0-9]|[3][01])|([0][469]|11)[\-]([012][0-9]|30)|02[\-]([012][0-9]))|((19|20)(([02468][1235679])|([13579][01345789]))|1900)[\-](([0][13578]|[1][02])[\-]([012][0-9]|[3][01])|([0][469]|11)[\-]([012][0-9]|30)|02[\-]([012][0-8])))$/**"

This can be used to validate a string matches the ISO standard for date (without the time). This will validate all dates between 1900-01-01 and 2099-12-31, including correct leap year validation. Imagine how much more complex this would be if it also validated times. Do we need such a complex expression? The answer is "NO". SAS provides more than regular expressions to help us validate the dates and times. We can use the regular expression to locate and extract possible date candidates in text fields, then use other SAS functions to validate them. For example:

```
data _null_;
    infile datalines truncover;
    input inpLine $100.;
    regex = prxparse(
"m/(\d{4})(\-|\/)(\d{2})(\-|\/)(\d{2})T(\d{2})(\:)(\d{2})(\:)(\d{2})/io");
    match = prxmatch(regex, inpLine);
    if match
    then
      do;
        strDate = upcase(substr(inpLine, match, 19));
        sasDate = input(strDate, ?? E8601DT.);
        if missing(sasDate)
           then put "WARNING: an invalid date was identified " strDate=;
           else put strDate= sasDate= datetime30. sasDate=  E8601DT.;
      end;
*    put match= strdate=;
    datalines;
This is verbage 2013-05-13T12:30:00 and more
This is verbage that abuts2013-05-13T12:30:00bbbb
not date here
not a full date here 2013-05-13
cccc 2013-05-32t12:30:00 ccccc
at 12:30 on 2013-05-13ddddddd1232013-05-13t12:30:00 dddd
;;
run;
```

and the SAS log:

```
strDate=2013-05-13T12:30:00 sasDate=13MAY2013:12:30:00 sasDate=2013-05-13T12:30:00
strDate=2013-05-13T12:30:00 sasDate=13MAY2013:12:30:00 sasDate=2013-05-13T12:30:00
WARNING: an invalid date was identified strDate=2013-05-32T12:30:00
strDate=2013-05-13T12:30:00 sasDate=13MAY2013:12:30:00 sasDate=2013-05-13T12:30:00
```

In this example we see using a simple regular expression was used to identify candidate dates, then the SAS functions SUBSTR() and INPUT() were used to extract and validate respectively. In the example we see that input line 5 had in invalid date (a 32nd of May). We can do more to identify the possible error:

```
data _null_;
    infile datalines truncover;
    length year $4.
           month day hour minute second $2.
           ;
    array parts (*) $ year month day hour minute second;
    array buffer (6) _TEMPORARY_ (1 3 5 6 8 10);
    input inpLine $100.;
    regex = prxparse(
"m/(\d{4})(\-|\/)(\d{2})(\-|\/)(\d{2})T(\d{2})(\:)(\d{2})(\:)(\d{2})/io");
    match = prxmatch(regex, inpLine);
```

```
    if match
    then
      do;
        strDate = upcase(substr(inpLine, match, 19));
        sasDate = input(strDate, ?? E8601DT.);
        if missing(sasDate)
           then
             do;
               put "WARNING: an invalid date was identified " _n_= strDate=;
               do i = 1 to dim(parts);
                  parts(i) = PRXPOSN(regex, buffer(i), inpLine);
                  put '     ' parts(i)=;
               end;
             end;
           else put strDate= sasDate= datetime30. sasDate=  E8601DT.;
      end;
    datalines;
This is verbage 2013-05-13T12:30:00 and more
This is verbage that abuts2013-05-13T12:30:00bbbb
not date here
not a full date here 2013-05-13
cccc 2013-05-32t12:30:00 ccccc
at 12:30 on 2013-05-13ddddddd1232013-05-13t12:30:00 dddd
;;
run;
```

and the log:

```
strDate=2013-05-13T12:30:00 sasDate=13MAY2013:12:30:00 sasDate=2013-05-13T12:30:00
strDate=2013-05-13T12:30:00 sasDate=13MAY2013:12:30:00 sasDate=2013-05-13T12:30:00
WARNING: an invalid date was identified _N_=5 strDate=2013-05-32T12:30:00
      year=2013
      month=05
      day=32
      hour=12
      minute=30
      second=00
strDate=2013-05-13T12:30:00 sasDate=13MAY2013:12:30:00 sasDate=2013-05-13T12:30:00
```

Using the PRXPOSN() function we extracted all of the date/time components. Although the example simply displayed the components, it would be a simple matter of building a user defined function with PROC FCMP to identify the bad components and ultimately build a strategy to correct the errors. If we had relied solely on a more rigourous regular expression, for example one that would check days in the 0 to 31 range, then we would not be able to identify some of these sorts of errors.

Now that we have seen an attempt to deal with a fully formed ISO date, let's look at some examples of partially formed dates. We will only look at a few examples to get started; given all the components of an ISO date and the combinations of partial dates that are possible, not every combination will be covered. As with all data cleaning exercises, you should first examine your data carefully to determine the most common types of errors and start with them. After a few projects you should have a good library of bad dates.

Our next example will validate incomplete components in the string. Here we are going to build smaller expressions which will represent the individual components of the date; these components will then concatenated to form the compete expression. An example component is:

- yyDExp = "(\d{4}|\-)";
    - \d{4} – four numbers (YYYY)
    - \- - the hyphen to indicate a missing component
    - | - the alternation meta-character.

Here we are defining the expression for a year to be four numbers (\d{4}) or a hypen (-). We have similar expressions for month, day, hour, minute, second as well as for the separators between date components and time components. The example code is:

```
data _null_;
    infile datalines truncover;
    length year $4.
           month day hour minute second $2.
           ;
    array parts  (*) $ year month day hour minute second;
    array buffer (6) _TEMPORARY_ (1 3 5 6 8 10);
    yyDExp = "(\d{4}|\-)";
    mmDExp = "(\d{2}|\-)";
    ddDExp = "(\d{2}|\-)";
    hhTExp = "(\d{2}|\-)";
    mmTExp = "(\d{2}|\-)";
    ssTExp = "(\d{2}|\-)";
    dSep   = "(\-|\/)";
    tSep   = "(\:)";
    regStr = "m/" ||  yyDExp || dSep || mmDExp || dSep || ddDExp || "T" || hhTExp
|| tSep || mmTExp || tSep || ssTExp || "/oi";
    input inpLine $100.;
    regex = prxparse(regStr);
    match = prxmatch(regex, inpLine);
    if match
    then
      do;
        call prxsubstr(regEx, inpLine, dStart, dLen);
        strDate = upcase(substr(inpLine, dStart, dLen));
        *put dStart= dLen=;
        sasDate = input(strDate, ?? $N8601B.);
        if missing(sasDate)
           then
             do;
               put "WARNING: an invalid date was identified " _n_= strDate=;
               do i = 1 to dim(parts);
                  parts(i) = PRXPOSN(regex, buffer(i), inpLine);
                  put i= parts(i)=;
               end;
             end;
           else put "match input " _n_= 3.  strDate= sasDate=$N8601B.;
      end;
     else put "no date in " inpLine;
    datalines;
2013-05-13T12:30:00
2013-05-13T12:30:-
2013-05--T12:-:00
2013-05-13T-:30:00
2013-05--T12:30:00
2013---13T12:30:00
--05-13T12:30:00
2013-05-13T12:30:00
This is verbage 2013-05-13T12:30:- and more
This is verbage that abuts2013---13T12:30:00bbbb
not date here
not a full date here 2013-05-13
cccc 2013-05-32t12:30:00 ccccc
at 12:30 on 2013-05-13ddddddd1232013-05-13t12:30:00 dddd
;;
run;
```

and the log:

```
match input _N_=1 strDate=2013-05-13T12:30:00 sasDate=20130513T123000
match input _N_=2 strDate=2013-05-13T12:30:- sasDate=20130513T1230
match input _N_=3 strDate=2013-05--T12:-:00 sasDate=2013-05--T12:-:00
match input _N_=4 strDate=2013-05-13T-:30:00 sasDate=2013-05-13T-:30:00
match input _N_=5 strDate=2013-05--T12:30:00 sasDate=2013-05--T12:30:00
match input _N_=6 strDate=2013---13T12:30:00 sasDate=2013---13T12:30:00
match input _N_=7 strDate=--05-13T12:30:00 sasDate=--05-13T12:30:00
match input _N_=8 strDate=2013-05-13T12:30:00 sasDate=20130513T123000
match input _N_=9 strDate=2013-05-13T12:30:- sasDate=20130513T1230
match input _N_=10 strDate=2013---13T12:30:00 sasDate=2013---13T12:30:00
no date in not date here
no date in not a full date here 2013-05-13
WARNING: an invalid date was identified _N_=13 strDate=2013-05-32T12:30:00
i=1 year=2013
i=2 month=05
i=3 day=32
i=4 hour=12
i=5 minute=30
i=6 second=00
match input _N_=14 strDate=2013-05-13T12:30:00 sasDate=20130513T123000
```

This example shows the use of CALL PRXSUBSTR() to get the start position and the length of the date candidate followed by the SUBSTR() function to extract the candidate:

```
        call prxsubstr(regEx, inpLine, dStart, dLen);
        strDate = upcase(substr(inpLine, dStart, dLen));
```

The call routine takes in the regular expression ID (regEX) and the source string (inpLine) and returns the start position (dStart) and the length (dLen). We need to use this call routine because we have no way of knowing how many missing components there will be, hence no way of knowing how long the string will be. In addition, we used the SAS informat $N8601B to validate the string; the $N8601B is able to deal with missing components. Once again we see it caught the truly invalid date of May 32.

We can extend our expression to include right truncation as well. Once again we will build strings for each component and concatenate them for the final expression. There are two main differences here

1. We are adding the “?” meta-character. This indicates that the prior sub-expression can be matched zero or one time
2. We are adding the separator (“-“ or “:”) to the string.

We also simplified by having only one expression (missDExp) to match either month or day, and another (missTExp) to match hour, minute or second:

```
    missTExp = "(\:)?(\d{2}|\-)?";
    missDExp = "(\-)?(\d{2}|\-)?";
```

The example code is:

```
data _null_;
    infile datalines truncover;
    length year $4.
           month day hour minute second $2.
           ;
    array parts  (*) $ year month day hour minute second;
    array buffer (6) _TEMPORARY_ (1 3 5 7 9 11);
    yyDExp = "(\d{4}|\-)";
    missTExp = "(\:)?(\d{2}|\-)?";
    missDExp = "(\-)?(\d{2}|\-)?";

    regStrTrunc = "m/" ||  yyDExp || missDExp || missDExp || "T?" || missTExp ||
missTExp || missTExp || "/oi";
```

```
if _n_ = 1 then put regStrTrunc=;
    input inpLine $100.;
    regex = prxparse(regStrTrunc);
    match = prxmatch(regex, inpLine);
    if match
    then
      do;
        call prxsubstr(regEx, inpLine, dStart, dLen);
        strDate = upcase(substr(inpLine, dStart, dLen));
        *put dStart= dLen=;
        sasDate = input(strDate, ?? $N8601B.);
        if missing(sasDate)
           then
             do;
               put "WARNING: an invalid date was identified " _n_= strDate=;
               do i = 1 to dim(parts);
                  parts(i) = PRXPOSN(regex, buffer(i), inpLine);
                  put i= parts(i)=;
               end;
             end;
           else put "match input " _n_= 3.  strDate= sasDate=$N8601B.;
      end;
     else put "no date in " inpLine;
    datalines;
2013-05-13T12:30:00
2013-05-13T12:30:-
2013-05-13T12
2013-05-13T
2013-05-13
2013-05
2013
2013-05--T12:-:00
2013-05-13T-:30:00
2013-05--T12:30:00
2013---13T12:30:00
-----T12:30:00
----05T12:-:-
-----T12:-:-
--05-13T12:30:00
2013-05-13T12:30:00
This is verbage 2013-05-13T12:30:- and more
This is verbage that abuts2013---13T12:30:00bbbb
not date here
not a full date here 2013-05-13
cccc 2013-05-32t12:30:00 ccccc
at 12:30 on 2013-05-13ddddddd1232013-05-13t12:30:00 dddd
;;
run;
```

and part of the log (cut short for brevity):

```
match input _N_=1 strDate=2013-05-13T12:30:00 sasDate=20130513T123000
match input _N_=2 strDate=2013-05-13T12:30:- sasDate=20130513T1230
match input _N_=3 strDate=2013-05-13T12 sasDate=20130513T12
WARNING: an invalid date was identified _N_=4 strDate=2013-05-13T
i=1 year=2013
i=2 month=05
i=3 day=13
i=4 hour=
i=5 minute=
i=6 second=
match input _N_=5 strDate=2013-05-13 sasDate=20130513
```

```
match input _N_=6 strDate=2013-05 sasDate=201305
match input _N_=7 strDate=2013 sasDate=2013
```

Here we can see we are catching truncated dates except where there is a “T” separator but no time following. This is a valid error; the standard states the time must follow the “T”.

CONCLUSION

The discussion above briefly described Perl Regular Expressions as implemented by SAS. It then showed how regular expressions can be used to help identify possible dates in free text fields in the data. The paper also showed how a combination of relatively simple regular expressions coupled with other SAS components, especially the ISO 8601 informats, can make the identification and validation of dates within free text fields possible with minimal code. Although we did not extend the discussion to ISO duration variables, it should be clear that a similar approach can be used to identify and validate durations. Also note the examples shown here are examples only; although they do work, they have not been thoroughly tested. As with all example code, if you choose to use these examples, test them thoroughly in your own environment and with your own data to ensure they meet your requirements.

REFERENCES

Borowiak, Kenneth [2012] http://analytics.ncsu.edu/sesug/2012/CT-03.pdf

CDISC SDTM Implementation Guide – Version 3.1.3
Available at http://www.cdisc.org/sdtm

Dunn, Toby “Grouping, Atomic Groups, and Conditions: Creating If-Then statements in Perl RegEx
Proceedings of the SAS® Global Forum 2011 Conference

Eberhardt, Peter and Qin Xiao Jin “ISO 101: A SAS® Guide to International Dating”,
Proceedings of the SAS® Global Forum 2013 Conference

ISO8601:2004 “*Data elements and interchange formats — Information interchange — Representation of dates and times*”, ISO 2004, Geneva, Switzerland

SAS Institute Inc. *SAS® 9.3 Functions and CALL Routines: Reference*. (Cary, NC: SAS Institute Inc., 2011)
Available at support.sas.com/documentation/cdl/en/lefunctionsref/63354/PDF/default/lefunctionsref.pdf.

SAS Institute Inc. 2011. *SAS® 9.3 Formats and Informats: Reference* . (Cary, NC: SAS Institute Inc., 2011)
Available at http://support.sas.com/documentation/cdl/en/leforinforref/63324/PDF/default/leforinforref.pdf.

CONTACT INFORMATION

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

Name:	Peter Eberhardt
Enterprise:	Fernwood Consulting Group Inc.
City, State ZIP:	Toronto, ON, Canada
E-mail:	peter@fernwood.ca
Web:	www.fernwood.ca
Twitter:	@rkinRobin
WeChar:	peterOnDroid