PhUSE 2012

Paper CS02

Parsing ISO 8601 Standard Partial Dates using Perl Regular Expressions

Mark Crangle, ICON Clinical Research, Dublin, Ireland

ABSTRACT

Regular expressions have been included in SAS since version 6 and were supplemented with the addition of Perl regular expressions in version 9 but there are still relatively few examples of where regular expressions can help out in the day to day tasks of a SAS programmer in the pharmaceutical industry. Although a comprehensive set of 'traditional' string functions exist, regular expressions can provide a more compact and robust method of performing complex tasks.

This paper explains the basics of Perl regular expressions in SAS and goes on to show how they can be applied to the problem of parsing partial date strings that conform to the ISO 8601 standard into a DDMMMYYYY format string which is commonly used for data listings. This example is extended into a macro for performing basic imputations on partial dates by using the functions that can extract substrings that match specific patterns.

INTRODUCTION

Regular expressions have been available in SAS for a number of years but are still relatively under utilized compared to other programming techniques. Whether this is due to a lack of awareness, understanding or examples of cases where regular expressions can applied to clinical data, this paper hopes to demonstrate how a little knowledge of regular expressions can be used to solve an everyday problem – handling ISO 8601 dates.

With the advent of the CDISC standards for clinical data it is becoming increasingly common for dates to be represented in ISO 8601 format as either a complete date/time or a partial/incomplete date/time. The SDTM Implementation Guide (v3.1.3) states that "*Using a character-based data type to implement the ISO 8601 date/time standard will ensure that the date/time information will be machine and human readable without the need for further manipulation*" [2] and while it is certainly the case that the information is instantly readable it doesn't really look very "pretty" so often further manipulation is required to get the value into a format that is both nice and easy to read.

ISO 8601 PARTIAL DATES

A further advantage of the use of the ISO 8601 standard for dates is that the precision or completeness of the date can be inferred by the presence or absence of components in the date. When following the standard, missing components are represented in one of two ways, either by right truncation (if the missing element is at the right of the date string) or by a hyphen (for intermediate components that are missing). Having intermediate components missing (for example, month and day are known but year is not) is much more unlikely that having components missing at the right hand end of the date string (for example, year and month are known but day is not) but these scenarios cannot be discounted if we are attempting to find a "complete" solution to the problem of partial dates.

A full list of possible combinations of missing and present date components is as follows:

#	Year Known	Month Known	Day Known	ISO 8601 Pattern	Example Date
1	Yes	Yes	Yes	YYYY-MM-DD	2005-09-12
2a	Yes	Yes	No	YYYY-MM	2005-09
2b				YYYY-MM--	2005-09--
3a	Yes	No	No	YYYY	2005
3b				YYYY----	2005----
4	Yes	No	Yes	YYYY---DD	2005---12
5	No	Yes	Yes	--MM-DD	--09-12
6	No	No	Yes	----DD	----12
7	No	Yes	No	--MM--	--09--
8a	No	No	No	(missing)	(missing)
8b				-----	-----

As well unlikely cases such as item 6 where Year and Month are unknown but Day is known, the full list of combinations also includes items 2b, 3b and 8b which would be used to represent the date part of a date/time value when the time is known (at least partially) but parts of the date are unknown. In these cases, right truncation to represent the missing parts would not work because the time part would need to be concatenated onto the right of the date string. Even the SDTM IG recognizes that such cases would be "very unusual" [2] but while provision exists for them in the ISO 8601 standards it is important to ensure that any solution we develop can handle them.

A method for processing these dates using "more traditional" functions could use the length function to determine which pattern is present. For example, if the only date types present in the data were the more common 1, 2a and 3a (from Table 1) then the following code can be used to differentiate between the different types:

```
data _null_;
  if length (in_date) = 10 then do;
    /*Process as YYYY-MM-DD*/
  end;
  else if length(in_date) = 7 then do;
    /*Process as YYYY-MM*/
  end;
  else if length(in_date) = 4 then do;
    /*Process as YYYY*/
  end;
  else if missing(in_date) then do;
    /*Process as null*/
  end;
  run;
```

If however, the data contains some of the more unusual date types or we are trying to build a solution that can handle all date types then using the length of the date string will not allow us to differentiate between the different types. For example, types 6 and 7 both have length 6 but need to be handled differently for any imputations or conversions. We might consider using substrings to examine where the "-" character is present in the date string but then we realize that we will have to use multiple calls to the substring function for each character pattern. So we eventually come to realize that matching each pattern separately can be quite complicated so what we really need is a method ideally suited for matching patterns which is where regular expressions come in!

BRIEF EXPLANATION OF THE SYNTAX FOR PERL REGULAR EXPRESSIONS IN SAS

SAS provides a number of functions that allow create regular expressions and use them to match patterns and create substrings and it is these functions that we will eventually use to process ISO 8601 dates. The functions used to create (compile) regular expressions is PRXPARSE which takes as its only argument the regular expressions itself. As this function compiles the regular expression on each iteration of the data step which can be computationally expensive it is a good idea to use an `If _N_ = 1 then …` to only execute the PRXPARSE function once and then use a RETAIN statement to retain this value onto each subsequent record.

The other basic function that is needed for regular expressions is the PRXMATCH function which locates the position in a string where a regular expression is found. It takes the input string and the regular expression that has been created by the PRXPARSE function as arguments and returns the first position in a string where the regular expression pattern is found. If the pattern is not found then the function returns 0.

In this first example, the simplest type of regular expression, an exact text match, is shown:

```
data prog1;
  if _n_ = 1 THEN pattern = prxparse("/andy/");
  retain pattern;
  input text $30.;
  match = prxmatch(pattern,text);
  put "RESULT: " text= match=;
datalines;
wes
joe alan aden jay
matt simon tommy gary
james andy
;
run;
```

The text inside the speech marks of the PRXPARSE function are the regular expression used in this simple program (the speech marks are part of SAS syntax, everything else is standard perl regular expression). The "/" character is the default delimiters specifying the start and end of the regular expression definition. This program only matches the exact text "andy" and gives the following output:

```
RESULT: text=wes match=0
RESULT: text=joe alan aden jay match=0
RESULT: text=matt simon tommy gary match=0
RESULT: text=james andy match=7
```

Only the final observation contains the text "andy" so all of the other observations return match = 0. The final observation returns match = 7 which indicates that the first character matched by the regular expression holds position 7 in the input string.

Regular expressions such as the above aren't really very useful but fortunately, Perl regular expressions define a set of metacharacters which allow us to specify types of character to be matched and allow us to add quantity to our regular expressions. This creates many more options for things that we can do with regular expressions. The SAS online documentation [3] gives a full list of the metacharacters, quantifiers and groupings that are available but the ones that will be useful for the partial date patterns are explained below.

Metacharacter	**Description**	**Example**
^	matches the position at the beginning of the input string	^red matches "red" but not "it's red"
$	matches the position at the end of the input string	red$ matches "red" but not "reds"
?	matches the preceding sub-expression zero or one time	123? matches "123" and "12" but not "124"
\d	matches a digit character	\d matches "0"
[123]	matches any one of the numbers inside the square brackets	1[123] matches "11" but not "14"
[0-2]	matches the numbers 0 to 2 inclusive	1[1-3] matches "11" but not "14"
{n}	matches the preceding sub-expression exactly n times	\d{2} matches "11" but not "1"
|	specifies the or condition	1|2 matches "1" and "2" but not "3"

A grouping construct that can take on special relevance when used with SAS is the grouping construct (*pattern*). This can be used to split the regular expression into groups and the parts of the string that match each group are stored in a capture buffer. For example the pattern (\d) (\d{2}) contains 2 capturing groups, the first matching one digit and the second matching 2 digits. These parts of the string matching each capturing group can be accessed directly using the

PRXPOSN SAS function. The syntax for this function is:

```
   PRXPOSN(regular-expression-id, capture-buffer, source)
```

where regular-expression-id is usually the variable created by the PRXPARSE function, capture-buffer is the id of the capturing group and source is the input string. With the above example regular expression (\d) (\d{2}) the code

```
   part1=PRXPOSN('/(\d) (\d{2})',1,input)
```

would extract the part of the variable input that matches 1 digit and the code

```
   part2=PRXPOSN('/(\d) (\d{2})',2,input)
```

would extract the part of the variable input that matches 2 digits.

If the capturing group begins with the metacharacter ?: then this creates a non-capturing group whereby the result is not assigned to the capturing buffer.

BUILDING PATTERNS TO MATCH PARTIAL DATES

Now that we have a basic understanding of regular expressions we can build a regular expression that matches the defined patterns for complete and partial dates. We can start by designing a pattern for a complete YYYY-MM-DD date and then remove the relevant parts to define patterns for all possible combinations of missing information.

From the metacharacters described above we can see that there are a number of different ways to define a pattern that matches a complete date. For example would probably be the simplest representation:

```
   \d\d\d\d-\d\d-\d\d
```

But we can be more specific than that as we know, for example, that the month part must be either a 0 followed by a 1-9 digit or a 1 followed by a 0, 1 or 2. This can be represented by:

```
   (0[1-9])|(1[012])
```

Similarly, we know the day part must be a 0, 1 or 2 followed by a digit (0-9) or a 3 followed by a 0 or 2. A representation of this is:

```
   ([0-2]\d)|(3[01])
```

Of course, the we know that some months have different numbers of days but including this incorporating this into the regular expression probably over-complicates the pattern to the extent that this check is probably better implemented as part of "traditional" code rather than a regular expression.

The final step of building the regular expressions is to realize that if we have each part of the pattern as a capturing group then we can get easy access to each component using the PRXPOSN function as demonstrated above. That means that the 3 parts of the regular expression become:

```
   yyyy="(\d{4})";
   mm="((?:0[1-9])|(?:1[012]))";
   dd="((?:[0-2]\d)|(?:3[01]))";
```

By default every pair of round brackets represent a capturing group so we need to define the 'inner' groups as non-capturing groups using the ?: syntax. In this way the PRXPOSN function can access each part of the regular expression using 1 for yyyy, 2 for mm and 3 for dd as the second argument to the function.

Now that we have the pattern for each component defined we can easily combine these to create one regular expression for each partial date pattern.

```
   pattern1=PRXPARSE("/^"||yyyy||"-"||mm||"-"||dd||"$/");  *Pattern: YYYY-MM-DD;
   pattern2=PRXPARSE("/^"||yyyy||"-"||mm||"(--)?$/");   *Pattern: YYYY-MM or YYYY-MM--;
   etc.
```

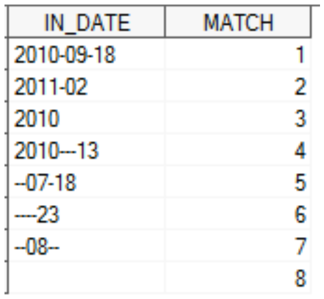
When creating the regular expressions, the `^` and `$` characters have been concatenated on at the start and end to specify that only text strings that match the pattern completely (ie. aren't preceded or followed by any other text) are matched to the regular expression. The other addition to pattern2 specifically is the optional group `(--)` (made optional by the following ?) which allows us to match patterns 2a and 2b (see table 1) using the same regular expression.

MATCHING PARTIAL DATES AND EXTRACTING PARTS

Now that we have defined the regular expression patterns we can put this to use to identify complete or partial date types and parse the components into separate variables for further uses. We can define the pattern variables into an array and loop through the array to create an indicator variable to hold the complete/partial date type. The following code stores the variables pattern1 to pattern8, which represent the 8 different partial date types from Table 1, into an array and then loops through this array to identify the type of the input date, in_date, which is stored in the variable match.

```
array pattern(8) pattern1-pattern8;
do j=1 to 8;
  if prxmatch(pattern(j),strip(in_date)) then match=j;
  if match ne . then leave;
end;
```

This gives the following result:



IN_DATE	MATCH
2010-09-18	1
2011-02	2
2010	3
2010---13	4
--07-18	5
----23	6
--08--	7
	8

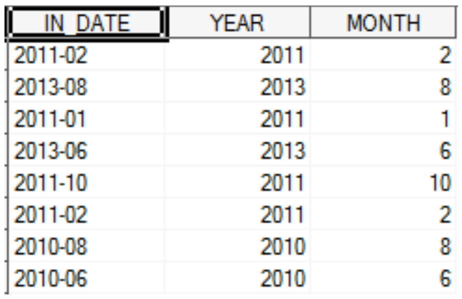
Note the use of the strip function on the variable in_date as part of the call to the pattern matching function PRXMATCH. This is important because the pattern has been defined with the ^ and $ meta-characters so that only a string completely matching the pattern. If the strip function is not used then the value of in_date that is passed to the PRXMATCH function is concatenated with trailing blanks up to the length of the variable. For example, if in_date is of length 10 and strip is not used then the value `"2011"` is actually passed to the PRXMATCH function as `"2011      "` which doesn't match the regular expression "^\d{4}$" because the \d{4} is not followed directly by the end of the string.

Now that we have this variable to identify the partial/complete type of the input date we can use this information to extract the components of the date into variables for either re-formatting as a more readable character string or for imputation of the missing parts. In this example the PRXPOSN is used in conjunction with the capturing groups that we defined in the regular expression to extract each component as a string which is then converted to a numeric value.

This example shows the extraction of the component parts of a date matching a YYYY-MM string where the year part is represented by capturing group 1 and the month part is represented by capturing group 2.

```
select (match);
  when (1) do; *Matched pattern YYYY-MM-DD;
    ...
  end;
  when (2) do; *Matched pattern YYYY-MM or YYYY-MM--;
    year  = input(prxposn(pattern2, 1, strip(in_date)) , 4.);
    month = input(prxposn(pattern2, 2, strip(in_date)) , 2.);
  end;
  etc.
```

We can see the second argument to the PRXPOSN function identifies the capturing group required. Again the strip function must be used on in_date in order to ensure that the trailing blanks do not cause us problems. This example gives the following result:



IN_DATE	YEAR	MONTH
2011-02	2011	2
2013-08	2013	8
2011-01	2011	1
2013-06	2013	6
2011-10	2011	10
2011-02	2011	2
2010-08	2010	8
2010-06	2010	6

From the position of having separated out the date components, converting to the date to a more readable string is very straight forward. The example shown here is of the DDMMMYYYY string but adapting this to different required format would be easily possible.

The only further processing required on the date components is to convert the numeric month value into the 3 letter character code commonly used. Perhaps the easiest way of doing this conversion is using a formula similar to the below:

```
proc format;
  value fmt_mm  1 = 'JAN' 2  ='FEB' 3  = 'MAR' 4  = 'APR'
                5 = 'MAY' 6  ='JUN' 7  = 'JUL' 8  = 'AUG'
                9 = 'SEP' 10 ='OCT' 11 = 'NOV' 12 = 'DEC';
  run;
```

The other consideration is how to represent the missing parts and this is often specific to the study being worked on so perhaps this could be specified as part of a macro call or global macro variable. To aid readability it might be necessary to replace the missing parts with another character in order that the parts that are present are aligned within a column. The code below demonstrates how the “missing character” could be specified as a macro variable and shows example output where the “missing character” is specified as a blank space and a hypen.

```
select (match);
  when (1) do; *Matched pattern YYYY-MM-DD;
    ...
  end;
  when (2) do; *Matched pattern YYYY-MM or YYYY-MM--;
    ...
    char_out = "&misschar&misschar"||put(month,fmt_mm3.)||put(year,z4.);
  end;
  when (3) do; *Matched pattern YYYY or YYYY----;
    ...
    char_out = "&misschar&misschar"||"&misschar&misschar&misschar"||put(year,z4.);
  end;
  etc.

if _n_=1 then put "IN_DATE    " "CHAR_OUT   ";
put in_date= char_out=;
```

```
IN_DATE   CHAR_OUT                      IN_DATE   CHAR_OUT
2011-02   FEB2011                       2011-02   --FEB2011
2010        2010                        2010      -----2010
2013-08   AUG2013                       2013-08   --AUG2013
2013        2013                        2013      -----2013
2012        2012                        2012      -----2012
2012        2012                        2012      -----2012
2011-01   JAN2011                       2011-01   --JAN2011
```

So we now have a macro that accomplishes the original task of parsing partial and complete dates from the ISO 8601 standard into the more readable DDMMMYYYY format. But seeing as we have done all the work of designing a the regular expressions to match these partial dates we might consider other tasks we might need to perform on partial dates. One such task is the imputation of the missing date components taking into account many of the more common partial date imputation rules.

IMPUTATION OF PARTIAL DATES

Partial date imputation rules are usually specified in the statistical analysis plan (SAP) and are specific to each study but they can be generalised to the rules that missing parts are imputed to the earliest or latest permissible value. A third category, imputing to a middle or arbitrary value (such as missing day to the 15th or missing month to June) is also occasionally used but as this is less common and more difficult to generalize so is not incorporated into the macro solution.

Additional constraints are often placed on the imputation by specifying that other study dates should be used. For example an adverse event start date with a missing day might be imputed to the first of the month or the day of treatment start date if the month and year of the event match the month and year of treatment start. We can think of this information as a window created by 2 study dates that we attempt to impute the partial date into if possible. Such rules are fairly common so an attempt to generalize these rules and include them into a macro solution would be useful.

A final piece of information that is used in partial date imputation is that most dates that are imputed have a related date that may also be known completely or partially. An example of this is discussed below:

Imputation Rule	For adverse events start dates where day/month are missing and year is known, the date should be imputed to the treatment start date if year of treatment start is equal to year of adverse event. Otherwise, month should be imputed to January and day should be imputed to 1.
Adverse Event Start Date	2012
Adverse Event End Date	2012-06-12
Treatment Start Date	12SEP2012

Using the rule as it is written, we would impute the adverse event start date to 12SEP2012 as the year of the event matches the year of the start date. However, we also know that the event ended on the 12th June in the same year so it would not be possible for the event start in September. Therefore if we apply this information then we would actually impute the date to 01JAN2012. We also need to recognise that the date could be a partial date itself but can still provide information the is required as part of the imputation. For example, if the end date above had day missing we would still know that the event ended in June so would be before the treatment start date.

So we have defined the information that is generally required for the imputation of partial dates; the input date itself, the “related” date which we will term as the primary reference date, a window of study dates which we will term as the reference window and whether the missing information will be imputed to the earliest or latest value that meets the reference constraints. As demonstrated above, the primary reference date is used to ensure that the imputation makes sense but the way it is used depends on whether the input date is a start or end date. Therefore this also needs to be specified.

We will assume that the reference window will only contain complete dates (or missing parts have already been separately imputed) so we will specify that these numeric values or variables need to be input. As a partial primary reference date can also provide information we will allow this to be a character value or variable to provide as input for this parameter. We can pass the primary reference variable through the same regular expressions that we will use on the input date to identify partial/complete date types.

Now that we have defined the parameters required for partial date imputation we can use that partial date parsing process that we have already created to feed into the imputation process. As mentioned above we can use the partial date identification on both the input and primary reference date to identify which parts need to be imputed and whether the primary reference date provides any information that constrains which values can be imputed. The capturing groups and PRXPOSN function can be used to extract the components that are present which can be combined with the imputed parts to create a complete date.

The first step is to check whether it is possible that the imputed date could fall within the reference window and to check whether there is any information that can be taken from the primary reference date that means that it impossible for the date to be imputed within the reference window. If a date within the window is feasible then the date should be imputed to the earliest/latest (depending on which option is specified by the user) date within the window subject to any constraints from the primary reference date. If it is not possible for the imputed date to fall within the window then the earliest/latest value is imputed. Arranging the imputation in this way to always attempt to derive a date within the reference window aims to replicate situations such as treatment emergent adverse event identification whereby partial dates should always be considered on treatment where possible.

CONCLUSION

Regular expressions have been available in SAS for some time but it has been difficult to find examples or instances where they can be applied in everyday tasks. With the advent of CDISC standards, there are clearly defined patterns for complete and partial dates and this is something that lends itself to the pattern matching capabilities of Perl regular expressions and the SAS functions that use them. This paper has shown that with an understanding of how partial dates are represented in the ISO 8601 standard and with a small amount of knowledge about how to define regular expressions a set of patterns to match complete and partial dates can be defined. This can be used to accomplish a couple of the more common manipulations that are performed on partial dates, converting to a more readable format and imputation of the missing information. It is worth investing time in understanding how to use regular expressions and their associated SAS functions as functionality is still being extended in recent versions of SAS. Indeed, SAS v9.3 has added support for including regular expressions in user defined formats and this is something that can be explored further in the future with regards to partial dates.

REFERENCES

1. An Introduction to Perl Regular Expressions in SAS 9, Ron Cody, Paper 265-29, SUGI 29
2. CDISC SDTM Implementation Guide – Version 3.1.3
3. http://support.sas.com/documentation/cdl/en/lrdict/64316/HTML/default/viewer.htm#a003288497.htm

ACKNOWLEDGMENTS

Many thanks to Paul Duckworth for his help in developing the logic for partial date imputation.

CONTACT INFORMATION

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

Mark Crangle
ICON Clinical Research
South County Business Park
Leopardstown
Dublin 18
Ireland

Email: Mark.Crangle@iconplc.com