Paper 245-2011

Grouping, Atomic Groups, and Conditions: Creating If-Then statements in Perl RegEx

Toby Dunn, AMEDDC&S, San Antonio, TX

Starting in SAS ® Version 9, Perl regular expressions were added and confusion soon followed. Since regular expressions are known for writing and not reading data, this is not surprising. Many SAS papers have been written describing what they are, their basic syntax, and giving a few simple examples, but none so far have specifically focused on three primary but puzzling concepts of regular expressions: grouping, atomic groups, and conditionals. This paper will elucidate those topics, starting with their basic definitions and working up to creating an If-Then statement using one line of regular expression code.

Keywords: Perl, RegEx, grouping, atomic grouping, If-Then, conditional RegEx

Introduction

To quote Larry Wall, "Perl is often said to be a language in which it is easy to write something that is difficult to read". One of, if not, the leading cause of this is the syntax for Regular Expressions (RegEx). Like any popular programming language Perl has evolved over the years and so has Perl's RegEx syntax. Part of the problem for RegEx's confusing syntax is the fact that Larry was trying to extend the functionality of the RegEx language but still keep it compatible with existing code. This meant that any new features had to be implemented with the already existing RegEx syntax and not break existing syntax constructs. The chief problem was and still is (until Perl 6 becomes fully operational) that Perl's RegEx syntax is reliant on too few metacharacters trying to do too much.

Almost all the previous SAS SUG papers have dealt almost exclusively with the Alphanumeric RegEx Metacharacters and only tangentially with the Capturing Parentheses and Extended RegEx Sequences. This paper will focus its attention on the significance the later has on the RegEx language. Specifically, it will cover the grouping and capturing features and functionality that parentheses play in the RegEx language. I will further delve into the extended sequences to perform look-arounds. Finally, I will bring this all together to show how to create a RegEx If-Then style pattern.

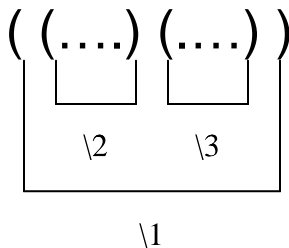
Parentheses in the RegEx language allow you to group parts of the RegEx pattern together into subpatterns, allowing you to either *Capture* or *Cluster* parts these subpatterns. Capturing subpatterns is useful when you need to reuse what is captured later on in the pattern or have the captured text returned to a SAS variable. Clustering allows you to effectively group or bind the subpattern inside of the parentheses in order to limit backtracking due to alteration, specify what is to be quantified, or to specify an extended RegEx sequence (ie. Lookarounds, atomic groups, inline modifiers, and If-Then constructs).

Capturing is performed by surrounding the subpattern with a set of 'bare naked' parentheses (<pattern>). Clustering is denoted by a set of parentheses wherein the opening parenthesis is immediately followed by a question mark, (?:<pattern>) or (?><pattern>). While Capturing parentheses can be used to group subpatterns for readability it does have the side effect of always holding the captured subpattern in memory. The alternative to grouping for readability is to use non-capturing parentheses (?:<pattern>). Non-Capturing parentheses are somewhat unsightly in that they have a ?: after the opening parenthesis. The general rule of thumb is if you are certain that what is matched will be a reasonably small amount of text then it is often just as easy to use Capturing parentheses.

Capturing Parentheses

As mentioned earlier, Capturing parentheses are normally used to denote a subpattern that you wish to capture and hold in memory for use in the pattern, replacement text or to place the captured text in a SAS variable. Once the RegEx engine matches the subpattern it places the captured text in what is called a *BackReference*. Each BackReference is denoted by \1 (backslash and the number corresponding to the captured set of parentheses), if it is to be used within the pattern; or $1 (dollar sign and the number corresponding to the captured set of parentheses) if it is to be used in replacement text.

BackReferences are numbered by the set of Capturing parentheses they correspond too. The RegEx engine numbers Captured parentheses outside in and left to right.



Example 1: Finding Consecutive Words

In this example the task is to find duplicate words that are next to each other in the target string and remove one.

```
211   Data _Null_ ;
212   Text = "One Two Two Three One" ;
213   Text = PrxChange( 's/\b([A-Z]+)\b\s+\1/$1/i' , -1 , Text ) ;
214   Put Text= ;
215   Run ;

Text=One Two Three One

's/\b([A-Z]+)\b\s+\1/$1/i'
```

- Match a word boundary
- Match the following Subpattern and capture its match into backreference number 1
- Match a Single Character A-Z or a-z at least once and as many as possible
- Match a Word Boundary
- Match a single White Space character at least once and as many times as possible
- Match the same text as most recently matched in capture group number 1
- Replace matched pattern with most recently matched capture group number 1

In the above example the *\b* metasequences bound the capture parentheses to ensure that the RegEx matches a word. It begins at the very start of the line right before the ‘O’ in the first instance of One in our target string. It then grabs the ‘n’ and ‘e’ and finds the next word boundary right after the ‘e’ and before the white space. It then matches the white space and attempts to then match the captured text ‘One’ against ‘Two’ in the target string. Since the ‘O’ does not match the ‘T’ the attempted match fails.

Then it backtracks and tries to match the leading \b again. It finds a match between the ‘e’ in the first ‘One’ and the white space. However, there is no text for the captured character class [A-Z]+ to match so it fails again. It then tries once again to match the \b and does so right before the ‘T’ in ‘Two’. It then matches the ‘Two’ and stops when it gets to the \b assertion right between the ‘o’ and white space. It then matches the white space and matches the captured ‘Two’ in the backreference against ‘Two’ in the target string. Since it can find a match it replaces ‘Two Two’ in the target string with a single occurrence of ‘Two’.

Again it is important to notice that if the backreference is used inside of the RegEx pattern to be matched it uses the form \1 and if it is used in the replacement text part of a search and replace argument it then uses the form $1.

Sometimes it is more useful to specify what it is you want to match and then have that matched text placed into a variable’s value. To do this one simply needs to capture the subpattern and then use *PrxPosn* to retrieve the matched text.

Example 2: Retrieving Matched Text

```
216   Data _Null_ ;
217   Text            = "One Two Two Three One" ;
218   Pattern        = PrxParse( '/\b([A-Z]+)\b\s+\1/i' ) ;
219   Match         = PrxMatch( Pattern , Text ) ;
220   DupWords = PrxPosn( Pattern , 1 , Text ) ;
221   Put DupWords= ;
222   Run ;

DupWords=Two
```

In this example we see basically the same RegEx pattern. The *PrxParse* function was used to compile the RegEx pattern because the PrxPosn function cannot compile its own pattern so therefore this requires an already compiled Pattern. As well as not being able to compile its own pattern it also requires that a match was already peformed, so this example uses the *PrxMatch* to do this. Finally, the PrxPosn function is used to retrieve the captured text. The first argument is the compiled pattern, the second is the number of the capture parentheses you want to retrieve the value from and the last argument is the Target string itself.

Limiting Alteration

While not specific to Capturing parentheses, parentheses can be used to limit the scope of alternation. One of the uses of *Alteration* is to allow for possible misspelling of words. In these cases many people have a tendency to list each word with each possible spelling in their entirety. This can greatly increase the amount of backtracking that the RegEx engine may have to do in the event that the alternative it is trying to match cannot be matched. By limiting the scope you can effectively minimize the amount of backtracking that the RegEx Engine may have to do to find a match.

Pattern	Target String	Matched Text	Matching Info
m/Wendesday|Wednesday/	Wednesday	W	
m/Wendesday|Wednesday/	Wednesday	We	
m/Wendesday|Wednesday/	Wednesday	We	Back Track
m/Wendesday|Wednesday/	Wednesday	W	
m/Wendesday|Wednesday/	Wednesday	We	
m/Wendesday|Wednesday/	Wednesday	Wed	
m/Wendesday|Wednesday/	Wednesday	Wedn	
m/Wendesday|Wednesday/	Wednesday	Wedne	
m/Wendesday|Wednesday/	Wednesday	Wednes	
m/Wendesday|Wednesday/	Wednesday	Wednesd	
m/Wendesday|Wednesday/	Wednesday	Wednesda	
m/Wendesday|Wednesday/	Wednesday	Wednesday	Match Found

In the above table we see that when the pattern 'm/Wendesday|Wednesday/' is used to look for a possible misspelling of 'Wednesday' it requires 12 attempts and backtrack to find a match. As the RegEx engine bumps along it can match until it comes to the misspelling "nd" in Wednesday. At this point it must *backtrack* to the beginning of the target string and start the whole matching process over again with the second alternative. This happens even though the only difference between the two alternatives is that the 'n' and 'd' are reversed. So this is somewhat inefficient, however, if one was to limit the alternation to just the two letters and their transposition it would effectively limit the scope that the RegEx engine has to backtrack too.

Pattern	Target String	Matched Text	Matching Info
m/We(nd|dn)esday/	Wednesday	W	
m/We(nd|dn)esday/	Wednesday	We	
m/We(nd|dn)esday/	Wednesday	We	Back Track
m/We(nd|dn)esday/	Wednesday	Wed	
m/We(nd|dn)esday/	Wednesday	Wedn	
m/We(nd|dn)esday/	Wednesday	Wedn	
m/We(nd|dn)esday/	Wednesday	Wedne	

m/We(nd|dn)esday/	Wednesday	Wednes	
m/We(nd|dn)esday/	Wednesday	Wednesd	
m/We(nd|dn)esday/	Wednesday	Wednesda	
m/We(nd|dn)esday/	Wednesday	Wednesday	Match Found

Here it takes 11 attempts to find a match. Yes, I know this doesn't seem like much, however, this is a cheesy overly simplified example using one word. In reality the savings are more often way more than just one attempt. Consider the following pattern:

m/Today is January 5th, 2010 a Wednseday|Wendesday/

In patterns like these the RegEx engine does not know whether you want limit the alternation to just 'Wednseday|Wendesday' or everything before versus everything after the |. In this case the Engine would interpret as the later. So to limit to just the characters you want you will need to do the following:

m/Today is January 5th, 2010 a (Wednseday|Wendesday)/

Which as we saw previously can be reduced to:

m/Today is January 5th, 2010 a We(dn|nd)seday/

Defining a quantified Subpattern

As many of you may already know by now *Quantifiers* specify how many times the RegEx engine will try to match the preceeding metasymbol or character. While quantifying one character or metasymbol is handy, doing the same thing for a subpattern is equally as handy. To do this simply wrap the subpattern with parentheses either with the capturing or non-capturing variety. Then follow that with the desired quantifier.

Example 3 Matching with a Quantifier

```
88    Data _Null_ ;
89    Text1 = "123456789" ;
90    Pattern1 = PrxParse( '/(\d*)/' ) ;
91    Pattern2 = PrxParse( '/(\d)*/' ) ;
92    Match1   = PrxMatch( Pattern1 , Text1 ) ;
93    Match2   = PrxMatch( Pattern2 , Text1 ) ;
94    Val1     = PrxPosn( Pattern1 , 1 , Text1 ) ;
95    Val2     = PrxPosn( Pattern2 , 1 , Text1 ) ;
96    Put Text1= Match1= Match2= Val1= Val2= ;
97    Run ;

Text1=123456789 Match1=1 Match2=1 Val1=123456789 Val2=9
```

Here we see two RegEx's which at first glance many might think will match the same thing, 0 or more numbers. However, when using quantifiers with subpatterns *it makes a difference where the quantifier is placed*. On line 90 we see Pattern1 has the * or star inside of the parentheses. This argument is stating that what should be captured is 0 or more numbers. While on line 91 the pattern has the * on the outside of the parentheses which states match 1 digit 0 or more times.

What is interesting is what is captured by these two patterns. Pattern1 does pretty much what we would expect in that it captures all the digits in the target string. The captured text from Pattern 2 is somewhat perplexing in that while we expect it to capture 1 digit what it ends up with is the last digit in our target string. This is due to the fact that the RegEx engine first matches the '1' in our target string and stores it in backreference 1. Then it matches the '2' and replaces the backrefernece value with that. It continues on doing this until it cannot match another digit in the target string, which in this example is a '9'. This is usually not what most people want to do, so watch where you place the quantifier on your subpattern.

Example 4 Quantifiers and BackTracking

```
Data _Null_ ;
Text = "aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa" ;
Match = PrxMatch( '/(\D+|<\d+>)*[!?]/' , Text ) ;
Run ;
```

Here the RegEx is looking for one or more non-digits or one or more digits enclosed in <>, that is followed by either a ! or ? mark. If it can match it will do so fairly quickly, the problem is when it doesn't match. When it cannot find a match it will iterate over 1 million times trying to find a match and then backtrack in an attempt to find a match. This is due to the fact that you have two quantified choices within the parentheses which is taken as one subpattern and is quantified. Again you have to watch were you put quantifiers and know exactly what you are asking the Engine to match. We will see in the Atomic Grouping section how to mitigate all the needless backtracking.

Clustering

Probably the most recognized of the *Clustering* parentheses is the non-capturing parentheses (?:<pattern>). But, as I mentioned earlier, it encompasses more than just this. To fully understand clustering and why they always start with a '(?' one has to go back to Perl RegEx language before these types parentheses were implemented. Before they were implemented in the RegEx language all one had available to use was Capturing parentheses. While this was okay, as more people started using RegEx's they wanted more flexibility like non-capturing parentheses, the ability to tell the RegEx engine match a subpattern and not backtrack, inline modifiers, lookarounds, and of course the If-Then construct.

The problem was that the RegEx language depends heavily on metacharacters and metasymbols. So what the developers decided to do was use a hack of sorts by reusing the parentheses and at the same time have these new ones start with '?' question mark right after the opening parenthesis (this would have previously been an illegal construct) to instruct the engine to further interpret the contained subpattern as special. The engine then looks for what comes after the ? to determine specifically what it should do with the subpattern.

What they ended up with was the following:

MetaSequences	
(?:<pattern>)	Non-Capturing
(?# <comments>)	In-Line Comments
(?modifier<pattern>)	In-Line Modifiers
(?><pattern>)	Atomic Group
(?=<pattern>)(?!<pattern>)	Look Ahead
(?<=<pattern>)(?<!<pattern>)	Look Behind
(?:If then\else)	If Then Else Construct

Non-Capturing Parens

If all you want is to cluster some subpattern for readability, limiting alteration, or to specify a subpattern for a quantifier, then you can use *Non-Capturing* parentheses. Non-Capturing parentheses use the sequence (?:<pattern>). So it always has a '?' followed by a ':' after the opening parenthesis. They will perform exactly like the previous Capturing parentheses examples with the exception that they will not capture the subpattern and hold it in a backreference. Honestly that that is pretty much all you need to know about these.

In-Line Comments

In-line comments may at first seem odd, given that one can use the */x* pattern modifier, but trust me they are lifesavers when attempting to comment RegEx patterns in SAS. They are denoted by a open parenthesis, a '?' followed by a '#' or (?# <comment>). This style of commenting is handy when you have a short one liner RegEx and want to add a comment. While one could use the /x pattern modifier you have to remember that the /x pattern modifier makes everything between a # and a newline character a comment. This means, if you wanted a one line

RegEx and you want to add a comment, you will have to specifically add a newline character to your pattern. This not only is unsightly but a real pain adding nothing but more confusion to your RegEx.

Example 5 In-Line Comments

```
45    Data _Null_ ;
46    Text = 'ABCD "XXX" XYZ' ;
47    Match = PrxMatch( '/"[^"]*(?# Match Non Double Quote Chars)"/' , Text ) ;
48    Put Match= ;
49    Run ;

Match=6
```

Here the In-Line comment will help add clarity to the RegEx pattern meaning while not adding to what the engine will attempt to match. When the Engine compiles the pattern it will parse the inline comment and disregard it in the final compiled pattern.

Some of you may be wondering why I wouldn't just use the /x pattern modifier. It's a valid question but one that has little to do with the RegEx engine and language and has more to do with how SAS functions.

Example 6 /x failure

```
Data _Null_ ;
Infile Cards ;
Input Text $20. ;
Match = PrxMatch( '/\d{3}  # Area Numbers
                    [ -]?  # Optional Seperator
                    \d{2}  # Group Numbers
                    [ -]?  # Optional Seperator  |
                    \d{4}  # Serial Numbers/xo' , Strip(Text) ) ;

Put Match= ;
Cards ;
123456789
123-456789
12345 6789
12345678
;
Run ;

Match=1
Match=1
Match=1
Match=1
```

This example attempts to do a simple SSN validation while commenting the RegEx pattern. In reviewing the log we see that it finds a match for every observation even when it shouldn't. The issue here is not with the pattern nor is it with the comments; rather it has to do with SAS. The /x pattern modifier tells the RegEx engine to treat all the text between a # and a newline character in the pattern as a comment and ignore all white space not inside of a parenthesis and character class. So what does this have to do with the false matches and SAS? The problem is that the SAS processor will strip all newline characters from the code before to passes it over to the Data Step parser to process. While I may have hit the return key in my editor to go to a new line and continue my RegEx pattern, these embedded hard returns get stripped out by the SAS processor. This means that they are no longer there to pass over to the SAS Data Step Parser and therefor can't be passed to the RegEx engine. So when the RegEx Engine saw the first # it couln't find a closing newline character and did not know when to end the comments in the pattern. Thus, the compiled pattern instructs it to match 3 digits and the rest was treated as one long comment.

So just how do you add comments in SAS? Well there are two ways one is you can concatenate a newline character into the pattern or you can use the In-line modifier with the /x pattern modifier.

Example 7 Comments the hard way

```
Data _Null_ ;
Infile Cards ;
Input Text $20. ;
Match = PrxMatch( '/\d{3} # Area Numbers        ' || '0A'x ||
                  '[ -]?  # Optional Seperator  ' || '0A'x ||
                  '\d{2}  # Group Numbers       ' || '0A'x ||
                  '[ -]?  # Optional Seperator  ' || '0A'x ||
                  '\d{4}  # Serial Numbers/xo   ' , Strip(Text) ) ;

Put Match= ;
Cards ;
123456789
123-456789
12345 6789
12345678
;
Run ;

Match=1
Match=1
Match=1
Match=0
```

Here we can see that a simply concatenation of newline characters in the pattern. I chose to use the '0A'x hex character or what most people would consider a line feed. However, doing it this way is not only cumbersome but a real pain. This is where the In-Line comments come in handy, especially if we also use them with the /x pattern modifier.

Example 8 Comments the easy way

```
Data _Null_ ;
Infile Cards ;
Input Text $20. ;
Match = PrxMatch( '/\d{3} (?# Area Numbers       )
                    [ -]? (?# Optional Seperator )
                    \d{2} (?# Group Numbers      )
                    [ -]? (?# Optional Seperator )
                    \d{4} (?# Serial Numbers     )/xo    ' , Strip(Text) ) ;

Put Match= ;
Cards ;
123456789
123-456789
12345 6789
12345678
;
Run ;

Match=1
Match=1
Match=1
Match=0
```

Now we see in this example we can use the In-Line comments to add the comments to the pattern without having to concatenate those pesky newline characters in the pattern. Also, you will notice it also uses the /x pattern modifier to be able to have free spacing and place the pattern on multiple lines instead of all on one line.

In-Line Modifiers

The RegEx language has pattern modifiers /i,/x,/s, and /m, these modify how the RegEx engine goes about performing its match. For instance the /i tells the engine to use a case insensitive match while the /x tells it to use free spacing and interpret everything between a # and a newline character as a comment. The great thing about pattern modifiers is it affects the whole pattern, the bad thing is it affects the entire pattern especially if you only want part of your pattern to be affected. So the Perl developers came up with In-Line *mode modifiers* (?ixsm<pattern>) which will affect the subpattern contained within the set of parentheses.

In-Line Mode Modifiers	
i	Case Insensitivity
x	Comments and Free Spacing
s	Dot Metacharacter Matches Everything
m	Enhanced Line Anchor Matching

Example 9 In-Line Modifier

```
Data _Null_ ;
Text  = 'Experience is a gReAt teacher.' ;
Match = PrxMatch( '/Experience is a (?i:great) teacher./' , Text ) ;
Put Match= ;
Run ;
```

In this example of the use of the (?i:…) In-Line modifier, I wanted to match the pattern however, I also wanted the entire string to be case sensitive except for the word 'great' which I wanted to be case insensitive. If I had used the /i pattern modifier the entire pattern would have been case insensitive. The In-line modifier allows for only part of the pattern to be case insensitive while the rest of the patter is case sensistive.

Atomic Grouping

Remember in example 3 and 4 we discussed backtracking saw how it can affect the time it takes to find a match or an even longer time when it can't find a match. Sometimes you can modify the way the code is written to keep this from happening. Other times you have the pattern exactly the way it should be but would really like the RegEx engine to fail as soon as possible and *not* backtrack. This is exactly what the *Atomic sequence* will do for you. It is commonly called Atomic grouping and looks like (?>….). It tells the RegEx engine that once it has found a matching subpattern not to backtrack on any of the quantifiers or alternative that may be inside of it.

Let's take another look at example 4. Remember, if it can find a match it will do so fairly quickly, but if it cannot it will take over 1 million attempts before reporting failure.

Example 10 Atomic Grouping

```
Data _Null_ ;
Text = “aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa” ;
Match = PrxMatch( ‘/((?>\D+)|<\d+>)*[!?]/’ , Text ) ;
Run ;
```

Now in this modified RegEx pattern I have enclosed the first '\D+' inside an Atomic group. If it were checked against our target string now the result only takes 11 attempts instead of taking over *1 million* attempts to report failure. I don't know about you, but that is some serious time savings. Before adding the atomic grouping of the non-digits the engine had two possible choices that had a quantifier which were all enclosed within a quantified subpattern. This made for a large number of choices and backtracking. By wrapping the \D+ in an Atomic group it makes the engine

look for all the non-digits it can find while not giving anything back to backtrack too. Thus failure can happen sooner rather than when Rip Van Winkle wakes up.

LookArounds

Sometimes when you are writing a pattern you just pull a ***ninja style*** peek either at the text before or after what you have matched. This gives you the ability to match say 'truck' and look at the text before it in the target string and ensure that it was a 'blue' and not a 'red' truck. To perform this type of task the RegEx language has what are called LookArounds. There are 4 types of these extended RegEx sequences to do this the Positive LookAhead, Negative LookAhead, Positive LookBehind and Negative LookBehind. These extended sequences are just "sneaking a peak", they do not actually consume any of the target strings text. Instead they assert that the enclosed subpattern can or cannot match at some point in the target string. Hence they are often call zero-*width assertions.* Another way of thinking about these is they will anchor the rest of the pattern to a spot in the target string. In other words match some text but only if that text is followed or preceded by some other sub-pattern.

LookAhead's

There is both a positive and negative lookahead. The positive lookahead asserts that the RegEx engine can find the enclosed subpattern and the negative lookahead asserts that it cannot find the enclosed subpattern. Positive Lookaheads are denoted by (?=<pattern>) and Negative Lookaheads by (?!<pattern>).

Example 11 The 'Commafication' Problem

```
Data _Null_ ;
Length Num $ 11 ;
Infile Cards ;
Input Num ;
Num = PrxChange( 's/(\d)(?=(\d{3})+(?!\d))/$1,/io' , -1 , Num ) ;
Put Num= ;
Cards ;
1234
123
12345
123456
1234567
12345678
123456789
;
Run ;


Num=1,234
Num=123
Num=12,345
Num=123,456
Num=1,234,567
Num=12,345,678
Num=123,456,789
```

In this example the RegEx is adding commas in the correct place in the target string such that the results will be as if we had converted the text string into a number and then formatted it with a comma11. Format. It looks for any digit then immediately it looks ahead trying to grab as many sets of three digits as it can (\d){3}. This makes perfect sense if you think about the fact that we only add commas for every three digits. So the first place on the number scale you would add a digit is when you reach 1000.

The commafication problem is a great example for LookAheads in that it has to add the comma in the correct place in the target string. If you think about it needs to consume or match the digit preceding the where the comma needs to be placed while ensuring that there are the correct number of digits to the right of the consumed digit. If we are only wanting to add one comma this would be fine however, we want commas throughout the entire list of digits, so it cannot consume the digits to the right as it will need to match against those to determine if in fact it needs to add more commas. So the (\d) grabs a digit ensuring we have a starting place to then allow the LookAhead to trying to find as many sets of 3 digits as it can '(?=(\d{3})+'.

However, it doesn't stop there it then goes on within the same LookAhead it then attempts a Negative LookAhead to see if it can find another digit. Remember that the (\d{3})+ is quantified as one solid unit of 3 digits. So if the number it is matched against is say 12345 and it didn't do the Negative LookAhead the result would end up being 1,2,345. What happened? Well the RegEx would match the '1', then matched '234' with the (\d{3})+ but couldn't match the '5' given there is only one digit left. It then would inserted the comma and attempted the match again. This time the \d matched the '2' and the (\d{3})+ matched '345' so it inserts another comma. By adding the last Negative lookahead it ensures that it will only add a comma *when the (\d{3})+ can match the rest of the digits in the target string*.

Example 12 Two LookAheads But Two Different Outcomes

Interestingly, many people get confused to how a LookAhead works. When a LookAhead is positioned at the beginning of the RegEx pattern and isn't preceded by anything that will consume text, it will search the entire target string looking for a match. If however, there is at least one non-zero width assertion (commonly called an atom) then it will look ahead in the target string by only the number of spaces specified by the subpattern.

```
1     Data _Null_ ;
2     Text = "She sells seashells by the sea shore" ;
3     Match1 = PrxMatch( '/(?=sea)/' , Text ) ;
4     Match2 = PrxMatch( '/sells(?=sea)/' , Text ) ;
5     Put Match1= Match2= ;
6     Run ;

Match1=11 Match2=0
```

In the above example the pattern in Match1 is looking ahead in the target string searching for the characters 'sea'. Since there is nothing preceeding the LookAhead in the pattern it will search the entire string left to right and sure enough it can match those characters starting at position 11. The pattern in Match2 is looking for the same thing, only it wants the pattern to match if and only if the engine can first match the text 'sells'. As you can see it did not find a match. When a LookAhead is preceded by consumable text (in this case 'sells') it will only look ahead the number of characters that is specified in the subpattern. In this case it was 3 characters ahead since there are there specified characters in the subpattern 'sea'. The engine first matched 'sells' then compared ' se' the next three characters in the target string after 'sells' against the subpattern in the LookAhead 'sea' and reported no match.

LookBehind's

Just as LookAheads look forward in your target string, LookBehinds looks backwards in your target string. You can recognize them by (?<=<pattern>) as being a Positive LookBehind and (?<!<pattern>) as being a Negative LookBehind. LookBehinds are harder for the RegEx engine to perform than LooAheads. This has to do with the fact that a RegEx engine matches from left to right in the target string. So if it needs to LookAhead it can simply grab more text, take a peek and report back what it finds. However, to perform a LookBehind it must remember where it has been. What all this means to programmer is it cannot accept variable length subpatterns. It has to know beforehand exactly how many characters to keep laying around to compare against. So you cannot use quantifiers, backrefereneces, and in the case of alternation only when all alternatives are the exact same number of characters.

Example 13 Looking Where You Have Been

```
106   Data _Null_ ;
107   Text = "She sells seashells by the sea shore" ;
108   Match1 = PrxMatch( '/(?<=sells\s)sea/' , Text ) ;
109   Match2 = PrxMatch( '/(?<!sells\s)sea/' , Text ) ;
110   Put Match1= Match2= ;
111   Run ;

Match1=11 Match2=28
```

Example 13 illustrates the use of both the Positive and Negative LookBehinds. Match1 is looking for any instance where it can find 'sea' and a 'sells ' subpattern preceding it in the target string while the second Negative LookBehind is looking for an occurrence of 'sea' but cannot find ' sells' before it in the target string.
Looking at the first pattern the RegEx engine first attempts to match 'sea', along the way it keeps track of the last five characters (since the LookBehind is 5 characters in length). Once it is able to match 'sea' in the target string it then takes those saves characters and attempts the LookBehind. Sure enough, both can be matched. The positive LookBehind match starts at position 11 and the Negative LookBehind at position 28. Interesting to note the start position reflects the place in the line that the consumable text starts *not where the beginning of the LookBehind subpattern would match*.

The If Then-Else Construct

The *If-Then-Else* construct (?(<If Condition>)<Then Condition>|<Else Condition>) allows you to test a subpattern. If the RegEx engine can match the *If* condition subpattern it goes on and matches the *Then* condition subpattern, otherwise it matches the *Else* subpattern. The Else part of the construct is completely optional. If the *Then* condition is not used it can be left off the RegEx pattern along with the |. SAS has two types of *If* conditions that can be used; one is a backreference to a set of capture parentheses to test if the subpattern participated in the match and the other is a Lookaround.

Example 14 If-Then Using a BackReference

```
147   Data _Null_ ;
148   Text = "ABC '123' XXX" ;
149   Match1 = PrxMatch( "/(')?\d{3}(?(1)')/" , Text ) ;
150   Put Match1= ;
151   Run ;

Match1=5
```

This example shows how to match a set of digits that may or may not be wrapped in single quotes. It also demonstrates using just using the Else portion to conditionally match certain text only when the previous *If* condition is true. The (')? attempts to match a single quote, the ? after the capturing parentheses makes the whole unit optional, I'll explain in just a minute why this is important. Next it attempts to match 3 digits. The last piece to the RegEx pattern is the *If-Then* construct. Notice the If condition is just a (1), which checking whether or not the first set capture parentheses participated in the match. The important detail is ***participates***, you see the RegEx pattern had the optional ? quantifier on the outside of the parentheses making the parentheses and everything inside of them optional. This means they may or may not participate in the overall match. Had the quantifier been inside of the parentheses it would only make the subpattern optional and the parentheses will always participate in the match meaning the *If* condition would always be true. The last thing to notice is the backreference in the If condition is referenced simply by a number corresponding to the set of capture parentheses and is not preceded by a \ or $.

Example 15 If-Then-Else Using a LookAround

Those programmers that use ICD9 codes often need only the first three digits of these codes to summarize their data based on major ICD9 code categories. The rub comes down to the fact that they can also have what are called E and V codes which are digits prefixed with either an E or V. To make matters worse, the letters aren't always all upcase or lowcase. So when there is only digits they need only the first three characters and when there is a leading E or V they want need first 3 to 4 characters depending on if the E or V code has 2 or 3 digits following the E or V.

```
Data _Null_ ;
Infile Cards ;
Input ICD9 $ ;
MajGrp = PrxChange( 's/^((?(?=[EV]).{3,4}|.{3})).*/$1/io' , 1 , ICD9 ) ;
Put ICD9= MajGrp= ;
Cards ;
E838
7102
30741
v679
918
8399
e9090
V10
;
Run ;

ICD9=E838          MajGrp=E838
ICD9=7102          MajGrp=710
ICD9=30741         MajGrp=307
ICD9=v679          MajGrp=v679
ICD9=918           MajGrp=918
ICD9=8399          MajGrp=839
ICD9=e9090         MajGrp=e909
ICD9=V10           MajGrp=V10
```

In this example the RegEx *If* Condition first looks to see whether or not there is a leading E or V and since the RegEx uses the /I pattern modifier it also looks for e and v characters as well. If it can find those characters the *Then* condition then instructs it to match up to the 3 and up to 4 characters. If, however, the If condition is false and it cannot find a leading E, e, V, or v character it then defaults over to the *Else* condition which matches the first 3 characters.

I know what your think…. 'Hey man but I will still have to capitalize the resulting values if I want to summarize them correctly'. No worries I have you covered, well that is if you have SAS 9.2. Simply use the \U and \E metasequences to upcase the replacement text

```
Data _Null_ ;
Infile Cards ;
Input ICD9 $ ;
MajGrp = PrxChange( 's/^((?(?=[EV]).{3,4}|.{3})).*/\U$1\E/io' , 1 , ICD9 ) ;
Put ICD9= MajGrp= ;
Cards ;
E838
7102
30741
v679
918
8399
e9090
V10
;
Run ;
```

```
ICD9=E838           MajGrp=E838
ICD9=7102           MajGrp=710
ICD9=30741          MajGrp=307
ICD9=v679           MajGrp=V679
ICD9=918            MajGrp=918
ICD9=8399           MajGrp=839
ICD9=e9090          MajGrp=E909
ICD9=V10            MajGrp=V10
```

Presto!!!! Like magic….. The \U metasequence instructs the RegEX engine to captilize everything after it until it reaches the \E metasequence. While this has been a feature in Perl RegEx language for a while it was introduced into SAS's version starting in 9.2.

Conclusion

I hope I have shown that no matter whether you are using parentheses to simply group a subpattern for readability, capture part of your pattern for latter use, limit part of your pattern, do a ninja sneak peak backwards or forwards in your target string or something as complex as an If-Then-Else construct that it all can be done pretty easily. No matter what your need is, the RegEx language has you covered, it may not be simple but it is definitely do-able. I would like to take the time to thank you, the reader, for selecting and reading my paper. I know there are many papers out there competing for your time. The fact that you would take time out of your day to read this humbles me and I am grateful. Remember, keep doing what you do and let not your heart be troubled.

References

SAS OnlineDoc 9.1.3 and 9.2 for the Web
SAS Institute Inc., Cary, NC

Friedl, Jeffrey, 2006, Third Edition, “Mastering Regular Expressions”, O'Reilly Media

Wall, Larry, Christiansen, Tom, & Orwant, Jon, 2000, Third Edition, “Programming Perl”, O'Reilly Media

Goyvaerts, Jan, Levithan, Steven, 2006, “Regular Expressions CookBook”, O'Reilly Media

Special Thanks:

I’d like to thank Paul St.louis, for all his help in editing my paper…. So if there is a misspelling or really bad grammar, blame him…. Just joking he is a awesome person, friend, and wonderful editor. Without whom most of my papers lately wouldn't have happened.

Contact Information

Your comments and questions are valued and encouraged.
Contact Toby at:

Toby Dunn
AMEDDC&S (CASS)
San Antonio, TX
Toby.Dunn@amedd.army.mil

The content of this paper is the work of the author and does not necessarily represent the opinions, recommendations, or practices of AMEDDC&S. SAS® and all other SAS Institute Inc. product or service names are registered trademarks or trademarks of SAS Institute Inc. in the USA and other countries. ® indicates USA registration.

Other brand and product names are trademarks of their respective companies.