Regular Expressions in SAS® Version 8 and Beyond
Francis Harvey, Westat

Abstract

Many of the problems encountered when working with data in SAS can be broken down into two simple groups of problems: matching specific patterns in a string and then changing the string based upon those matches. This presentation illustrates how regular expressions can be used for these purposes by examining some typical validation problems as well as some sample text searches and replacements. We begin by looking at how the SAS regular expression (rx) functions can be used to solve these problems. For an interesting contrast, we then solve these same problems within SAS using methods from publicly available regular expression libraries. The methods used for linking the regular expression libraries to SAS sessions do not require a cross-platform compiler.

Introduction

As we develop programs to extract useful information from various sources of data, it can sometimes seem as if every situation is unique and requires special coding tailored for that particular problem. Yet, as programmers, it is our ability to generalize problems into more abstract concepts, which can then be handled using similar techniques, that is most useful in saving us from tedious effort as we progress from task to task. This paper will focus on describing one of these abstract concepts, pattern matching, and the use of regular expressions as a general-purpose method for solving problems that fall into this classification. Although this paper focuses on working with SAS, regular expressions are supported in a number of products and may have additional capabilities beyond those demonstrated in this paper.

Pattern matching

Pattern matching is defined for this paper as the process of searching a string for a sequence of characters that satisfy a pattern. A pattern is built by identifying the basic pattern elements and then gradually adding further elements to extend these patterns to create a new, larger pattern. At the culmination of this process, you will have a single pattern that describes all of the characteristics that a sequence of characters must have to be considered a match. These elements are illustrated with regular expression syntax to provide some necessary grounding to the generalized concepts.

The most trivial type of pattern element is the exact match of a character in the pattern to a character in the search string. As you explore the regular expression syntax, which must use pre-defined characters to stand for the various matching elements, you will learn which characters are used to match exactly and which represent other pattern elements. Reversing this process, you will also come to learn how to represent characters that can’t be typed or alternative methods of representing those that can, using elements from the regular expression syntax.

From exact matching, you can proceed to the grouping of sequences into larger subpatterns with various logical relationships between them. Using the concatenation element, you can form a larger pattern that matches only when two subpatterns are both matched in sequence. For instance, if characters a and b are the two subpatterns, the concatenations of these, ab, will only match when both characters are found together in that order in the search string. If instead either of two subpatterns indicates a match, you would indicate this using an alternation element (commonly represented by the | character) between the two subpatterns. In regular expression syntax, this element has the lowest precedence of any pattern element, so the alternation will always be between the largest surrounding subpatterns. For instance, the pattern ab|c will match the characters ab or c as the concatenation must be matched before the alternation. To control this behavior, and the default behavior of various other pattern elements, you will need to create subpattern groups using the grouping element (commonly represented by matched parenthesis). Returning to our example of ab|c, by grouping the alternation between b and c to create the new pattern a(b|c), you can limit the size of the surrounding subpatterns found by the alternation so that characters ab or ac are the only matches for this new pattern.

To simplify the common process of specifying a large set of alternative characters using alternation patterns, you can use list elements (commonly represented by matched square brackets []) to create matching lists where a match on any item from the list will cause the list pattern to match. List items are usually limited to single characters; ranges of characters, where any character from the range can serve as the match; or named pre-defined matching lists known as character classes. A non-matching list element (commonly represented by placing a ^ immediately after the leading square bracket of a list element) further adds to this convenience by allowing you to match any character other than one from the list rather than having to specify all of the other possible characters.

Abstract

Many of the problems encountered when working with data in SAS can be broken down into two simple groups of problems: matching specific patterns in a string and then changing the string based upon those matches. This presentation illustrates how regular expressions can be used for these purposes by examining some typical validation problems as well as some sample text searches and replacements. We begin by looking at how the SAS regular expression (rx) functions can be used to solve these problems. For an interesting contrast, we then solve these same problems within SAS using methods from publicly available regular expression libraries. The methods used for linking the regular expression libraries to SAS sessions do not require a cross-platform compiler.

Introduction

As we develop programs to extract useful information from various sources of data, it can sometimes seem as if every situation is unique and requires special coding tailored for that particular problem. Yet, as programmers, it is our ability to generalize problems into more abstract concepts, which can then be handled using similar techniques, that is most useful in saving us from tedious effort as we progress from task to task. This paper will focus on describing one of these abstract concepts, pattern matching, and the use of regular expressions as a general-purpose method for solving problems that fall into this classification. Although this paper focuses on working with SAS, regular expressions are supported in a number of products and may have additional capabilities beyond those demonstrated in this paper.

Pattern matching

Pattern matching is defined for this paper as the process of searching a string for a sequence of characters that satisfy a pattern. A pattern is built by identifying the basic pattern elements and then gradually adding further elements to extend these patterns to create a new, larger pattern. At the culmination of this process, you will have a single pattern that describes all of the characteristics that a sequence of characters must have to be considered a match. These elements are illustrated with regular expression syntax to provide some necessary grounding to the generalized concepts. The most trivial type of pattern element is the exact match of a character in the pattern to a character in the search string. As you explore the regular expression syntax, which must use pre-defined characters to stand for the various matching elements, you will learn which characters are used to match exactly and which represent other pattern elements. Reversing this process, you will also come to learn how to represent characters that can’t be typed or alternative methods of representing those that can, using elements from the regular expression syntax.

From exact matching, you can proceed to the grouping of sequences into larger subpatterns with various logical relationships between them. Using the concatenation element, you can form a larger pattern that matches only when two subpatterns are both matched in sequence. For instance, if characters a and b are the two subpatterns, the concatenations of these, ab, will only match when both characters are found together in that order in the search string. If instead either of two subpatterns indicates a match, you would indicate this using an alternation element (commonly represented by the | character) between the two subpatterns. In regular expression syntax, this element has the lowest precedence of any pattern element, so the alternation will always be between the largest surrounding subpatterns. For instance, the pattern ab|c will match the characters ab or c as the concatenation must be matched before the alternation. To control this behavior, and the default behavior of various other pattern elements, you will need to create subpattern groups using the grouping element (commonly represented by matched parenthesis). Returning to our example of ab|c, by grouping the alternation between b and c to create the new pattern a(b|c), you can limit the size of the surrounding subpatterns found by the alternation so that characters ab or ac are the only matches for this new pattern.

To simplify the common process of specifying a large set of alternative characters using alternation patterns, you can use list elements (commonly represented by matched square brackets []) to create matching lists where a match on any item from the list will cause the list pattern to match. List items are usually limited to single characters; ranges of characters, where any character from the range can serve as the match; or named pre-defined matching lists known as character classes. A non-matching list element (commonly represented by placing a ^ immediately after the leading square bracket of a list element) further adds to this convenience by allowing you to match any character other than one from the list rather than having to specify all of the other possible characters.
that should be accepted. Continuing in this vein, you have the match-any-character element (commonly represented by .) for matching any single character (although it is common to exclude the undesirable newline character from this list). After introducing ambiguity in the characters that can serve as a match, you can now look to introduce ambiguity in the number of times a character sequence can match a pattern.

Repetition pattern elements allow you to specify that a subpattern should repeat a fixed number of times, a range of possible times, or an open-ended range of possible times. For instance, you may wish the subpattern to be optional in which case you would use a match-zero-or-one element (commonly represented by ?) to match a single instance of the subpattern when present or allow it to be missing from a match in the search string. Similarly, the match-zero-or-many element (commonly represented by *) or match-one-or-many element (commonly represented by +) allow for matching as their name suggest. For less specific uses, you can use an interval element (commonly represented by matched braces {}) to represent a closed-range of repetitions with a minimum and maximum number of repetitions or an open-ended range where no limit is set on the number of matches that can be made. As repetition elements operate on the smallest preceding subpattern, you will once again need to use grouping to vary the type of matches to be made. Another point to consider is that regular expressions follow a “greedy” approach, where the most number of repetitions of the subpattern that the element allows for is chosen, rather than a “non-greedy” approach, where the fewest is chosen. Finally, you would attempt to identify the rest of the most common pattern elements.

Regular expressions will commonly support at least two other basic pattern elements: the back-reference (commonly represented by \digit) and positional elements. With a back-reference, you can match a preceding group that has been denoted with a grouping element. This is accomplished by assigning each group a sequential number as it occurs in the pattern and then using that number with the back-reference element. This creates a subpattern that matches the exact same characters that matched the group later on in the pattern. For positional elements, you can use the match-beginning-of-line element (commonly represented by ^) or the match-end-of-line element (commonly represented by $). The ^ allows you to match the beginning of a string or to specify that a match should be sought immediately after a newline character in a multi-line search string. The $ allows you to match the end of a string or to match immediately before a newline character. With this basic list of common elements, you can compactly represent most of the patterns you might need to match in a search string. However, it is always a good idea to expand your abilities and look into the extended elements that are supported by the particular regular expression syntax that you are using.

Some regular expressions provide positional elements beyond simple beginning and end of line recognition; it may also be possible to recognize specific ordinal positions in a string or between newline characters. Case-insensitive matching, skipping of white space, hex and octal character notation for ascii character representation, etc. are several additional capabilities that you may be able to take advantage of. However, it is important to note that not every possible pattern can be easily represented in a regular expression syntax.

As you work with your chosen syntax, you will come to recognize the limitations it imposes on you. For instance, representing a complete scheme for determining whether two lines of a poem rhyme would be practically impossible with regular expressions (although attempts have been made). You must also come to learn how your syntax chooses between patterns that can be matched in more than one way such as a ** pattern which is typically supported by causing the first element to match as much as possible, thus ensuring the second * matches nothing. Similarly, ambiguities exist in how a syntax should backtrack when an initial choice of character sequences for a subpattern does not allow the whole pattern to be matched. Thus, in the search string consisting only of the character a, the pattern a*(ab) would not match if the a? subpattern matched the character a, but if it matches zero occurrences then the whole pattern can be matched. Working with regular expression syntax is obviously not without its challenges, but it is one of the best means of representing a wide variety of pattern matching problems in a simple syntax for easier comprehension and for generalization so a single solution method can be used on a wide variety of problems.

Problem categories
Pattern matching problems can typically be categorized into one of three tasks. A first category is using a pattern match to validate a string. Returning the match found in the search string for the pattern or a subpattern from the pattern is another common task. A third type of problem involves replacing characters from the search string that matched the pattern or a subpattern. These replacements can be constant or can be based on the matches made in the pattern. There are more specialized tasks that fall outside of these classes, but these give you a general idea of how pattern matching is used.

Solutions
After generalizing what you hope to match and deciding what is to be done once you have a match, you can then choose a method for solving your problem. Typically, your first thought would be to use the functions offered by your programming environment (the SAS DATA step in this paper) that best solve the problem at hand. Nevertheless, the advantages that come from using regular expression solutions should not be disregarded lightly.
If you choose to create SAS code that solves your problem as optimally as possible, you will typically end up with a solution that is focused only on your current pattern matching problem. Instead of only needing to fully understand one solution method for all of your problems, you will need to know the full details of every language element you used in your solution. Similar problems that arise may require extensive code modifications and attempts to generalize the code to offer a more flexible approach may fail due to the difficulty of anticipating what changes in the pattern will be required. Creating problem specific code also usually involves sacrificing easy comprehension of the methods involved for performance gains. This difficulty of comprehension is another factor in making the code difficult to modify for reuse when new problems are encountered.

To illustrate these problems, let's solve two matching problems using the best SAS methods possible, where speed is our primary concern. The first example is to verify that a string known to be a number in decimal notation is an integer in that notation as illustrated by this dataset:

```sas
data str;
  length str $ 32;
  str = "5.5";      /* false */
  output;
  str = "  55";     /* true */
  output;
  str = "-55  ";    /* true */
  output;
  str = "0";        /* true */
  output;
run;
```

A first approach might be to convert the string to a number and then verify whether it is an integer as in:

```sas
data _null_;  
  set str;  
  test = index(str,".") = 0;  
  put str= test=;"n  
run;
```

This approach results in correct answers at optimal execution speeds, so we have a satisfactory solution for this problem. It is easily comprehensible but doesn't really offer much flexibility. To allow for later developments, we could instead verify that only the acceptable integer characters are found: numbers, spaces, and minus signs. This proves to be a wise decision as in our second example we drop the assumption that the string must already be a number in decimal notation. Clearly, we will need to do more than just check for the presence of a decimal point against this much larger range of possible strings as illustrated by appending some new invalid strings to our dataset:

```sas
data str;
  set str END=last;
  output;
  if last;
    str = "5e6";      /* false */
    output;
    str = "+55";      /* false */
    output;
    str = "055";      /* false */
    output;
    str = "00";       /* false */
    output;
    str = "-0";       /* false */
    output;
    str = "5  6";     /* false */
    output;
    str = "-";       /* false */
    output;
    str = "-5-6";    /* false */
    output;
    str = " ";       /* false */
    output;
run;
```

Although this code works on our test case, we quickly come across a problem as the numbers grow larger. For instance, 1000000000000000.1 will fail to be caught by this test. Only by being sure we know the limitations of every language element in our problem specific code can we be sure we have an actual solution to our problem. In this case, rather than rely on SAS conversions, we should ask ourselves what separates the characters that make up an integer from those that make up a real number (the only two types that can be represented in decimal notation). We would then see that it is the presence of the decimal point that separates the two types. So, we will use the decimal points absence to indicate validity as in:

```sas
Advanced Tutorials
```

```sas
data _null_;  
  set str;  
  test = index(str,"." ) = 0;  
  put str= test=;"n  
run;
```

However, establishing the desired characters in any acceptable string is a good place to start, so we should verify that there are only numbers, spaces, and minus signs in the string. The spaces, if present, can only occur at the beginning or end of the string so that they do not fall between any other character, although we also need to check that there actually are other characters for them to fall between. Next, we should verify that if there is a minus sign, that it is the only one in the string, that it precedes all of the other non-space characters in the string, and that a non-zero digit comes immediately after it. If there is no minus sign, any digit sequence, with or without surrounding spaces, is fine as long as there are no leading zeros. So an accurate, optimal solution would probably go something like this:
data _null_;  
set str; 
/* Valid characters only */  
test =  
not verify(str,"-0123456789"); 
if test then do;  
strT = left(str);  
strS = index(strT," ");  
strL = length(strT);  
/* Spaces can't occur between other characters and can't be the only characters in the string */  
test = strS = 0 or strS > strL;  
if test then do;  
Minus = index(strT,"-");  
/* If there is a minus, */  
if Minus then  
/* it must be the first non-space character, */  
test = Minus = 1 and  
strL =~= 1  
/* occur only once, */  
and  
not index(substr(strT,2),"-"))  
/* & be followed by non-zero digit. */  
and  
substr(strT,2,1) =~= "0";  
/* If there isn't a minus, */  
else  
/* can have spaces around a single digit number */  
test = strL = 1  
/* or a number without a leading 0 */  
or  
substr(strT,1,1) =~= "0";  
end;  
end;  
put str= test=;  
run;  

Clearly, we have lost somewhat in terms of comprehension in order to accommodate the new requirements of our second problem, but have we lost something in flexibility as well? What if we do want to allow an empty string to be acceptable as an integer? In this case, we would have to understand exactly which line is performing this verification (fortuitously commented in this case) and understand how the verification is being made. In this case, the code takes advantage of the fact that the length of the empty string (or a string of spaces) is returned as one, and the location of a space in such a string is also one. So, we would have to change our equality check to allow these two numbers to be equal if the empty string is to be allowed, hardly an obvious change to those unfamiliar with this exact code. While there is no guarantee that using a regular expression solution will overcome all of the difficulties we have noted with this style of solution, its more general approach is often exactly what is called for.

By providing a general approach to solving pattern matching problems, regular expressions seldom excel at solving any particular problem you come across. Your options for optimizing the code are far more limited although a full understanding of what you are capable of doing is much easier to achieve. This approach also has the benefit of typically being more flexible and easier to understand than coding which is based on just one problem. To illustrate, we will go through our earlier examples again using the SAS 8.2 RX functions.

Recalling the earlier discussion of pattern matching elements, there clearly is no method for converting the string to a number. However, you can check for a period in the string as in our solution to the first problem which has the same limitation as before and can be generalized in the same manner:

data _null_;  
set str END=last;  
retain pattern;  
if _n_ = 1 then pattern =  
rxparse(".");  
test = not rxmatch(pattern,str);  
if last then call rxfree(pattern);  
put str= test=;  
run;

This code may seem a little strange and difficult to understand at first, but you should really view it as more of a template for using SAS regular expressions that you can use over and over again. After you have created a pattern that has the characteristics you want, in this case an exact match of the decimal point, you plug it into rxparse for SAS to compile and provide a reference number. You only need do this once per use, so the template limits it to the first time through the DATA step. However, you do need to retain the reference number for the rxmatch function to use the compiled pattern. Rxmatch actually verifies the pattern against the string, and so we get our result. When you are done, it is nice to free the memory for the compiled pattern that you have created although it isn't technically necessary. The key to understanding this solution is to not worry overly much about the template, which varies little from problem to problem, but rather to understand the pattern, as you will see in the solution to the second problem:

data _null_;  
set str END=last;  
retain pattern;  
if _n_ = 1 then pattern =  
rxpathre(  
"$@' '*((['-'][0-9][+d]*)|(0))' '*@0"  
);  
test = rxmatch(pattern,str);  
if last then call rxfree(pattern);  
put str= test=;  
run;
Once again, the code has grown more complicated although this time the complication is found in our pattern string while the template remains the same. We’ll build our pattern for this example following the general path described in the introduction to pattern matching, which will end up being very similar to how we arrived at our earlier solution. Starting with the characters that we want to match exactly, we note that a match must always contain one or more digits but cannot start with a 0, so our subpattern becomes $'1-9\text{s}d*$. SAS RX function syntax differs from the general regular expression syntax originally outlined, but the common elements are still provided. In this case, we have a list element $'1-9'$ which specifies a character from the range 1-9 is to be matched. The $\text{s}$ is equivalent to a named list element and could have been written instead as $'0-9'$ which matches any digit. Finally, the * has exactly the same meaning as before and operate only on the $\text{s}$ as this is the smallest preceding subpattern. So, we end up matching any number beginning with a character from 1-9 that may or may not be followed by any number of other digits. A clear problem with this is that we cannot match 0 with this pattern. So, we add 0 as an alternate, and the pattern becomes $((\text{'}1-9\text{s}d*\text{')}(0))$. Now, only non-zero numbers can have a minus sign if it is present, so we add an optional minus [\text{'-'] to our non-zero alternate and get $((\text{'-']}\text{'}1-9\text{s}d*)(0))$. Finally, we need to allow for beginning and trailing spaces of any number * so our pattern becomes $\text{'*'}((\text{'-']}\text{'}1-9\text{s}d*)(0))\text{'*'}$. While this would appear to be complete, an important detail of regular expressions is that they will match any substring from a larger string that matches the pattern unless you add positional elements to specify where a substring match can occur. Thus, we add a $\text{@0}$ to ensure that the pattern matches from the beginning of our search string to the end of the search string, which is the end of line in this case $\text{@0}$. So, our whole pattern is: $\text{@0}((\text{'-']}\text{'}1-9\text{s}d*)(0))\text{'*'}\text{@0}$. Initially, using regular expressions may seem more complicated than your conventional coding techniques even when the logic behind either solution is very similar. However, with practice, you can come to understand these patterns as easily as you understand any other function you would use in a DATA step. In fact, they can even serve as direct replacements for functions as you can see by comparing our search for a decimal point using the index function versus using the rxparse function. The advantage provided by regular expressions isn’t in an ability to provide functionality that other SAS functions can’t, as there is no such functionality. Instead, it rests in an ability to provide a large amount of general functionality using a single, structured syntax. Thus, solutions based on regular expressions will tend to be less dominated by those features peculiar to a specific problem as the single syntax makes overly specialized coding less likely. When you encounter a regular expression solution, to fully understand what is being done and how it can be modified, you need only fully understand the regular expression syntax. In our second solution, we know everything about the types of strings that can be matched by understanding the syntax from our solution pattern. This syntax also lends itself easily to a variety of modifications as we can see in our second solution by once again trying to accommodate the empty string as an acceptable match. In this case, we would understand that the number is optional in a match, so we would make it optional in our pattern, which becomes: $\text{@0}((\text{'-']}\text{'}1-9\text{s}d*)(0))\text{'*'}\text{@0}$. Making this simple modification only required us to understand the syntax and how it was being used rather than one of many functions that might have been involved in the task.

There is no real guarantee that using regular expressions will ensure that your pattern matching solution is easy to develop, comprehensible, and adaptable to new requirements. In fact, the only certainty is that it will probably not run as quickly as a solution that takes full advantage of the SAS DATA step language. Yet, by adding regular expressions as one of, if not the preeminent, tool you use when performing pattern matching, you end up with a far greater likelihood of arriving at a solution that possesses all of these attributes.

## Beyond validation

String validation is certainly one of the more important uses for regular expressions. There are hundreds of examples of regular expressions readily available from a simple web search that can validate everything from international phone numbers to zip codes to valid IP addresses. However, there are still two other common usages: returning the character sequences that matched a pattern or subpattern and performing replacements in a string based on the characters that matched a pattern or subpattern. To illustrate these two additional types of problems, the text of this paragraph has been placed in a text file for use in our coding examples.

```sas
data paragraph;
  length text $ 32767;
  infile "c:\nesug\paragrph.txt"
    LRECL=32767 TRUNCOVER;
  input text $CHARR32767.;
run;
```

Examining the code, you should note SAS is limited to working with 32,767 characters in a single string. This limitation and several others will be covered later, but for right now we will assume that this is a practical approach to working with strings. To find the actual character sequences that matched, you will need to use the special implementation provided by the functions you are using. For SAS RX functions to get the first sentence in our paragraph, we could use RXSUBSTR:
The pattern matches any character up to the first period and RXSUBSTR gives you its position and length so that you can find the string that actually matched. This grossly oversimplifies the problem as sentences do not have to end in periods and periods may be used for other purposes than ending sentences, but this simplification will be sufficient for this paragraph. To find the third word in that first sentence (which we will assume has at least three words in our pattern), we would have to alter the pattern to recognize words, and specifically, the third word.

As you were warned, this needs to be very specific to how SAS implements regular expressions. Ignoring the <>’s and the second line of the pattern for now, we can see that the repeated pattern of the first two groups will recognize any non-space character followed by a space. While there are some simplifications that can be made to this pattern, unfortunately, specifying a specific number of allowable repetitions for a subpattern is not one of them. This is yet another limitation that will be discussed later. Finally, our pattern recognizes the last group of non-space characters followed by either a space or a period. This gives us the first three words from our first sentence. To pull the third word, we tag the subexpression that will match those characters using the <>’s which makes those characters available for later use as =1. To pull only that variable, the characters following TO in the second line of our pattern become the value of the variable word3 from RXCHANGE where the =1 specifies that it should return the characters from our tagged value. The reason we couldn’t just use RXCHANGE on the original string is that the changed value returned actually consists of all of the characters before the match in the search string, the new characters from the change expression, and all of the characters after the match, which would have been the rest of the paragraph. Thus, we first make sure that we are dealing only with the match when pulling values out. For the final RX example, we have decided that the paragraph is too long and needs to lose the second line.

Our pattern in this case, recognizes two instances of any character followed by a period. For our change, we tag only the first group and change the matched text to be only that sentence. So, we have effectively dropped the second line from the paragraph.

This brief look at the regular expression capabilities beyond validation is not meant to be exhaustive. There are many more possibilities that you can explore and mixing these features with other SAS functions allows for even greater creativity. Although the methods used are specific to the SAS RX functions, the capabilities are not, and you can probably count on them being available in some form in any sufficiently complex regular expression package you choose to use.

Beyond SAS

Originally this paper was motivated less by a desire to explore regular expressions in general than dissatisfaction with their implementation through the RX functions in SAS version 8. The limitation to SAS character variables which can be at most 32,767 characters, the lack of a means to specify a specific number of repeats for a subpattern, the inability to refer to specific subpatterns later in the pattern, and the choices for syntax elements in general were all issues that negatively affected every solution that was built using these functions. However,
enhancements planned for release with version 9 have rendered this less of an issue, as there will now be a new set of functionality based on Perl regular expressions which addresses almost all of these issues. To see how to make use of these new functions, you can refer to Jason Secosky’s paper, “The DATA step in Version 9: What’s New?” from SUGI 27

As not everyone has version 9 and there are still some benefits to be gained from using regular expression functions external to SAS, we will continue with a look at alternative methods for using regular expressions in SAS. By using these methods you can no longer rely on the documentation and support from SAS in solving your problems. However, the advantages to be gained may outweigh these drawbacks.

Regular expression libraries
The first step is to find a library of functions that can be used from the DATA step. Fortunately, there are a number of free-to-use regular expression libraries readily available. One site with a large amount of public code including regular expression libraries is that offered by the GNU Project at http://www.gnu.org. Even if you choose not to use their regular expression packages, there are a number of other coding libraries that may prove useful to you. Alternatively, you can visit the Perl homepage at http://www.perl.com for the source code for the Perl regular expression library.

These free-to-use packages cannot always be used completely without reservation. For instance, using code from the GNU Project or using the Perl code has associated licensing issues that need to be read and considered when you are redistributing software based on these solutions. If the terms are not acceptable, then you will have to find libraries that carry no such restrictions or obtain licensed software that matches your needs. Fortunately, for personal use, as in this paper where I make use of the software but do not attempt to redistribute it, these licensing issues are not applicable.

Another drawback is the lack of support and testing from a corporate vendor like SAS. Although this may seem to be a significant negative factor, the fervent and large group of people who develop and use these packages can serve as a reliable source for help and verification of these coding packages. In some aspects, such as the rate of improvements to the package, this support can even be regarded as superior to vendor support.

One last problem is that the code must be compiled to run on your the operating system. Fortunately, there are often pre-compiled versions of the software available from various sites for the most popular packages. For instance, the Perl library used for this paper is a Windows build of ActivePerl® 5.6.1 available from ActiveState at http://www.activestate.com/Products/ActivePerl.

Alternatively, as C is the programming language of choice for these libraries, an obvious answer would be to obtain SAS/TOOLKIT® and compile it from SAS. Another SAS-based approach is to attempt to convert the code to SAS, although this won’t likely meet with a great deal of success. Turning away from SAS based options, you could consider obtaining a C compiler for your system either from a corporate or from a variety of free sources. As covering all of these various options is beyond the scope of the paper, you will need to investigate the possibilities for yourself. For this paper, the second library chosen was the GNU regex 0.12 library compiled with Microsoft Visual C++ 6.0 for Windows. Although this library has since been replaced by the GNU rx functions, it still provides sufficient functionality to serve as a good test package.

Using external programs from SAS
In order to make use of the regex and Perl packages from SAS, it is still necessary to establish a way for your code to be able to use the functions from the DATA step. This means that there must be some form of communication allowing parameters to be passed to the functions and results returned to the DATA step so that they may then be used in the rest of your code. Fortunately, SAS provides a number of methods for communicating with external programs.

One method to consider is to create a DLL and use the SAS CALL MODULE functions to invoke it. While this is a very effective method, its reliance on DLL’s limits its usage only to programs run from Windows. If you wish to investigate this method, you should refer to the SAS language documentation.

A second technique is to create a server application and use sockets to communicate between SAS and the server. While this method is not operating system limited, the need to establish a specific port number for communication and to ensure the regular expression server is executing while the methods are being used is somewhat discomforting for such a simple usage. Sigurd Hermansen’s SUGI 27 paper, “Data Socket Adapters”, http://www2.sas.com/proceedings/sugi27/p166-27.pdf provides an excellent background in the usage of this sockets for communicating with a server.

Using named pipes within SAS provides another method for communicating with the programs. This is one of the better communication methods as the pipe server provided by the operating system takes care of communication between SAS and the external program once a pipe has been established. Unfortunately, we once again run into problems with operating system support as Windows 9x lacks support for named pipes. If you are using an operating system that supports named pipes, however, you may want to investigate this technique in the SAS language documentation.
One method that will be used by this paper is the usage of unnamed pipes to communicate with the external program. The exact implementation of unnamed pipes will vary based on your operating system, so their usefulness will vary depending on the way they are implemented. For Windows, the parameters you can send are limited by the capabilities of the Windows command prompt which has severe formatting and length restrictions.

Another method illustrated in this paper is the usage of temporary files to communicate with the external program. This method is also supported in all operating systems as only the names of the files being used for communication need to be passed to the external program. Another advantage to this method is that any type of character string of any length may be placed in the temporary file that the external program will work on.

This provides a sample of some of the diverse methods of using other programs from SAS where communication between the two programs must be established. There are undoubtedly additional methods beyond those that have been listed. Continued development in integrating SAS with Java may provide yet more techniques for communicating with external programs than are available today. In short, if you use external programs from SAS, you should try to keep up on developments in this area.

Unnamed pipes
In order to use unnamed pipes, there needs to be a method to add the string being searched and the pattern being sought as part of the command sent to run the external program. The external program must then run the appropriate function and return the result to the standard output. In SAS, this can be accomplished by creating a new file reference with the filename function. As the path for the reference, you will need to specify an appropriate command line for the external program. You will also need to specify the keyword pipe to use unnamed pipes. After this, you execute the command using fopen and fread. The value placed in standard output by the external program can then be read using fgets. Finally, you should close the reference and clear it from memory.

```
%macro rx_pipe(cmd);
    result = "";
    rc = filename('ref',"&cmd","pipe");
    fid = fopen('ref','S');
    rc = fread(fid);
    rc = fgets(fid,result,200);
    rc = fclose(fid);
    rc = filename('ref','CLEAR');
%mend rx_pipe;
```

With Windows, each filename invocation will cause a DOS window to be displayed for a brief period of time. Unfortunately, there does not seem to be a way of preventing this from happening.

Examining the runtimes for the SAS specific coding, SAS RX functions, GNU regex functions, and ActivePerl functions for our second pattern matching problem will help us to decide the effectiveness of each routine. However, we cannot simply repeat the tests multiple times for each record from our sample dataset as regular expression libraries can be setup to optimize repeated function calls with the same input string. Instead, we will run through the entire dataset multiple times, which should provide a better comparison.

<table>
<thead>
<tr>
<th># of reps</th>
<th>Non-rx code</th>
<th>SAS RX</th>
<th>GNU regex</th>
<th>Perl</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.11</td>
<td>0.05</td>
<td>8.19</td>
<td>7.57</td>
</tr>
<tr>
<td>10</td>
<td>0.11</td>
<td>0.04</td>
<td>55.8</td>
<td>1:03</td>
</tr>
<tr>
<td>50</td>
<td>0.93</td>
<td>0.22</td>
<td>4:42</td>
<td>4:14</td>
</tr>
</tbody>
</table>

As you can see from the chart, there is a large performance penalty associated with using the regex and Perl routines. As these programs perform efficiently when called directly from a command prompt, the time delay would appear to be caused by the usage of unnamed pipes for program communication. For times where you will only need to call the routines infrequently, you might consider using one of these methods to use the added features they offer.

Temporary files
Using temporary files to communicate with an external program requires that you establish a location and name for the temporary files that will be created. You will also need to know the syntax for creating external files from the regular expression library that you are using. First, you will need to create a new file reference with the filename function. This reference should be set to the path for the temporary file that will hold your commands to the external program. The commands should then be written to this file using fput (if necessary, appending a newline character to each passed string to generate multiple lines) followed by a single fwrite. Next, you will need to close the reference and clear it from memory. In order to run the commands in the temporary file, you will use the call system function with a command line appropriate for your library that will cause it to execute the commands in the temporary file. To ensure the external program runs synchronously with your SAS code, you will need to be sure the xsync option is set. This ensures that your DATA step won’t process any further commands until after the external program has finished running. You will also want to ensure that the x window is minimized and doesn’t require input from you to close, so you will need to ensure that the noxwait and xmin options are also set. Finally, you will need to create a file reference for reading the output from the library using fgets. Once again, you should close and clear the reference after you are finished with it.
Comparing the runtimes for the SAS specific coding, SAS RX functions, GNU regex functions, and ActivePerl functions for our second pattern matching problem using this technique:

<table>
<thead>
<tr>
<th># of reps</th>
<th>Non-rx</th>
<th>SAS</th>
<th>GNU</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.10</td>
<td>0.04</td>
<td>9.00</td>
</tr>
<tr>
<td>10</td>
<td>0.22</td>
<td>0.11</td>
<td>1:43</td>
</tr>
<tr>
<td>50</td>
<td>0.32</td>
<td>0.21</td>
<td>7:47</td>
</tr>
</tbody>
</table>

We once more see a hefty penalty associated with the external programs due to the usage of this technique. The time penalty actually appears to be significantly greater with this method. These results would also suggest that limiting usage of this technique to infrequent calls in your DATA step would be advisable.

**Temporary files additional note**

Unlike this sample problem, you would typically find a way of processing the data so that you could include more than one search string in your file. This allows you to obtain multiple results using a single call to the external library, thus greatly reducing the penalty associated with using an external library. Perl, in particular, has many additional features that make this an important feature to consider.

**Conclusion**

It should now be clear that regular expressions can provide a solid foundation for creating flexible solutions to your pattern matching problems. As with most coding solutions, there is no perfect way of determining whether these methods will be right for you. However, given the variety of benefits they provide, you should certainly consider them when faced with your next data processing task.

**Disclaimer**

The contents of this paper are the work of the author and do not necessarily represent the opinions, recommendations, or practices of Westat.

**References**

ActiveState. *ActivePerl*. ActiveState. http://www.activestate.com/Products/ActivePerl


Pattern Matching. CSE233 Course, Department of Computer Science And Engineering, University of NotreDame. http://www.cse.nd.edu/courses/cse233/www/lectures/match.pdf


**Acknowledgements**

Special thanks to Sigurd Hermansen, Quentin McMullen, Duke Owen, Mike Rhoads, and Kellar Wilson for their help in reviewing this paper. I would also like to thank the contributors to the GNU Project and the Free Software Foundation as well as Larry Wall and the various individuals who have contributed to the development of Perl for their continued efforts to make open source software available to all.

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 registered trademarks or trademarks of their respective companies.

**Contact Information**

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

Francis R. Harvey III
Westat, An Employee-Owned Research Company
1650 Research Blvd
Rockville, MD 20850

Email: harveyfl@westat.com

9