A Regular Expression Is A Healthy Expression

Paul Shannon, RTI Health Solutions, Manchester, UK

ABSTRACT
Data collection often takes the form of open-ended text fields. The pharmaceutical industry is no exception; from clinical trials to patient record data, open ended text is used to record information considered relevant. Dosages, event dates or even patient identifying data could be written as free text. The principle of regular expressions as a tool for mining this data for various types of characters and/or combinations of text exists in many programming languages. SAS has supported regular expressions since v6.12, with v9 seeing considerable enhancements. Languages such as Java and Visual Basic, each support their own take on regular expression technology to provide programmers with a powerful tool for mining text or string data based on patterns of text, rather than fixed characters. As a result, regular expressions have a very wide scope to solve many different problems in a range of programming environments.

INTRODUCTION
The principle of Regular Expressions (RegEx, RegExp) as a technology is nothing new. The term “Regular Expression” was originally penned by in the 1950’s as algebraic terminology. The first RegEx library which could be used by other programs was released in the 1980’s. Since then, most programming languages have contained some form of Regular Expressions. The principle is often more simple than the application, Regular Expressions are used to determine patterns of text within a string. This is an enhancement to a substring (SUBSTR function in SAS) which only looks for a specific string within another. Whilst RegEx can look for specific strings, it also has the power to search for any string which follows a specified pattern. The real-world benefit of this is that the programmer can analyze text strings for required information and consider possible variations of how crucial information can be entered, or look for different pieces of related information, all in one go.

INTRODUCTION TO REGEX SYNTAX
RegEx consists of it’s own syntax which is used to specify a pattern of text to search for. This is principally based on Perl syntax. In essence, the syntax works like any other mark-up language, where special characters or tags are used to specify keywords or meanings in order to pass instructions to the relevant engine. Just to make things more complicated, each take on RegEx, be it in SAS, .NET, Java etc., tend to use slightly different syntax. However these differences are few and far-between. There is a cheat-sheet provided later in this paper.

If you need to search for the text string "run;", you can use the simple text "run;" as your match string. This will work either in RegEx, or in normal string functions. However, if you want to look for the text string “run;” and also ensure that this text appears at the end of a line, you can use the search for the RegEx pattern of "run;$". The dollar sign is a special character to represent the end of a string (or end of a line).

REGULAR EXPRESSIONS IN SAS
SAS has supported Regular Expressions since v6.12, consisting of five functions all starting with RX. However, it was somewhat limited (and in my opinion, quite slow). Version 9 saw a host of new RegEx functions, dubbed Perl Regular Expressions (with all the function names starting PRX), this paper will focus on the version 9 (PRX) functions. RegEx in SAS (or any other language) is not intended to be a complete replacement to the standard string functions, such as INDEX, SUBSTR, SCAN, etc.; these are still perfectly valid functions which provide their own solution and can also be used alongside RegEx functions where necessary.

The Enhanced Editor in SAS has the option to use RegEx in its Find and Replace dialog. Figure 1 shows the standard find and replace dialog box used in SAS. There is a tickbox option to perform a “Regular expression search”. The arrow button immediately to the right of the find combobox allows a shortcut for specifying various patterns to search for. Whilst I rarely use this within SAS itself, it can also be used as a tool to help familiarise yourself with RegEx syntax.
In the examples below we will look at a fictional dataset containing patient prescription data. An open text variable exists, which in some cases contains drug dose information, the examples will work towards extracting the dose information into separate variables for strength and frequency where specified.

PRXPARSE AND PRXMATCH FUNCTIONS

Imagine you wish to find all drugs which belong to a specific class of drug called “zolamides”. These drugs can be identified by name as they all contain the text string “zol” in their name.

The first example looks for a simple substring within a specified variable (this could also be done using the INDEX function in SAS). Nevertheless, there are two PRX functions used here. The first is the PRXPARSE function. This creates a regular-expression-id for a specific RegEx match string. The PRXMATCH function then refers to this ID number in order to perform the search on a specified variable and return the index where the match was successful. If the match is unsuccessful, the PRXMATCH function returns 0. The PRXPARSE function is the only place in this example where the match string is specified.

Regular expression match strings are written in the form of: "/pattern/options". The forward slashes are required. The pattern you wish to look for is placed inside two forward slashes (/), this is required in SAS. Any characters specified after the closing forward-slash specify RegEx options. In the example below, “I” is used to specify that the search is not case sensitive. See the cheat-sheet in the references for a more comprehensive list of options.

EXAMPLE 1.1: USING PRXPARSE WITH PRXMATCH TO FIND TEXT PATTERNS

```sas
data ParseAndMatch;
    set Prescriptions end=lastObs;
    retain pID;
    if _N_ = 1 then pID = PRXPARSE("/zol/I");
    position = PRXMATCH(pID,drug);
    if lastObs then call PRXFREE(pID);
run;
```

OUTPUT:

<table>
<thead>
<tr>
<th>Obs</th>
<th>patient</th>
<th>Drug</th>
<th>position</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>10006783</td>
<td>Xylometazoline</td>
<td>9</td>
</tr>
<tr>
<td>2</td>
<td>10006783</td>
<td>Xylometazoline</td>
<td>9</td>
</tr>
<tr>
<td>3</td>
<td>10031662</td>
<td>Xylometazoline</td>
<td>9</td>
</tr>
<tr>
<td>4</td>
<td>10031662</td>
<td>Xylometazoline</td>
<td>9</td>
</tr>
<tr>
<td>5</td>
<td>10056700</td>
<td>Timoptol 0.5% b.d.</td>
<td>0</td>
</tr>
</tbody>
</table>

In SAS 9.1 or later, the PRXPARSE function is not required when performing a simple match as in Example 1.1. It is possible to use the match pattern directly in the PRXMATCH function. Example 1.2 below will produce exactly the same results as in example 1.1. I would advise NOT relying this method as many other RegEx functions in SAS do require you to create a regular-expression-id. The shortcuts this method offers doesn’t really hold much value in terms of ease of programming or system resources.
EXAMPLE 1.2: USING THE PRXMATCH FUNCTION WITHOUT PRXPARSE IN SAS 9.1

```sas
data PRXMATCH;
  set RawData;
  position = PRXMATCH("/Meta/I",drug);
run;
```

PRXSUBSTR FUNCTION

Despite the familiar function name, the PRXSUBSTR function does not actually return a substring. Instead it populates specified variables to be used as arguments for the substr function (i.e.: position and length). Unlike the PRXMATCH function, the PRXSUBSTR function can only handle a regular-expression-id, rather than a string directly. Therefore you must create a regular-expression-id using the PRXPARSE function as shown in Example 1.1.

EXAMPLE 1.3: USING PRXSUBSTR FUNCTION

```sas
data PRXSUBSTR;
  set Prescriptions end=lastObs;
  retain pID;
  if _N_=1 then pID = PRXPARSE("/zol/I");
  call PRXSUBSTR(pID,drug,nStart,nLen);
  if nStart>0 then substr = substr(drug,nStart,nLen);
  if lastObs then call prxFree(pID);
run;
```

OUTPUT:

<table>
<thead>
<tr>
<th>Obs</th>
<th>patient</th>
<th>Drug</th>
<th>nStart</th>
<th>nLen</th>
<th>substr</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>10006783A</td>
<td>Xylometazoline</td>
<td>9</td>
<td>3</td>
<td>zol</td>
</tr>
<tr>
<td>2</td>
<td>10006783A</td>
<td>Xylometazoline</td>
<td>9</td>
<td>3</td>
<td>zol</td>
</tr>
<tr>
<td>3</td>
<td>10031662A</td>
<td>Xylometazoline</td>
<td>9</td>
<td>3</td>
<td>zol</td>
</tr>
<tr>
<td>4</td>
<td>10031662A</td>
<td>Xylometazoline</td>
<td>9</td>
<td>3</td>
<td>zol</td>
</tr>
<tr>
<td>5</td>
<td>10056700R</td>
<td>Timoptol 0.5% b.d.</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

SUBEXPRESSIONS

As mentioned earlier, RegEx offers much more than find/replace. One such enhancement is the ability to search for one or more of a collection of strings. Creating a RegEx match pattern which consists of a number of components is referred to as creating subexpressions, where a subexpression is written in parentheses (). The nth subexpression is identified as starting at the nᵗʰ open-bracket as you read the match pattern from left to right. These subexpressions can be in parallel or nested. The example below shows a match pattern searching for at least one of two fixed strings. The vertical bar | is RegEx syntax for an ‘or’ statement. Note: the PRXPAREN function requires a PRXPARSE and a PRXMATCH function to be used before-hand in the data step.

Imagine the drug class zolamides has a subclass called metazolamides. We can search for any drug that contains the text string “zol” or “meta” (or both). The PRXPAREN function can be used to obtain the last found subexpression from a given match pattern. In the example below, “zol” is subexpression 1 and “meta” is the second subexpression.

EXAMPLE 1.4: SUBEXPRESSIONS AND THE PRXPAREN FUNCTION

```sas
data PRXPAREN;
  set rawdata;
  pID = PRXPARSE("/(zol)|(meta)/I");
  position = PRXMATCH(pID,drug);
  subexpression = PRXPAREN(pID);
  call PRXFREE(pID);
run;
```

OUTPUT:

<table>
<thead>
<tr>
<th>Obs</th>
<th>patient</th>
<th>drug</th>
<th>pID</th>
<th>position</th>
<th>subexpression</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>10006783A</td>
<td>Xylometazoline</td>
<td>1</td>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>10031662A</td>
<td>Xylometazoline</td>
<td>1</td>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>10056700R</td>
<td>Azolamide</td>
<td>1</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>10056700R</td>
<td>Amoxicillin 0.2% b.d.</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>10056700R</td>
<td>Clotrimazole B.D.</td>
<td>1</td>
<td>9</td>
<td>1</td>
</tr>
</tbody>
</table>
PRXNEXT FUNCTION
In the example output above, the drug "Xylometazoline" actually contains a match for both the sub expressions. The position and subexpression variables contain values referring to the first match found in the variable drug. It is possible to search for multiple matches within one variable in one record, particularly when using subexpressions. The PRXNEXT function can be used to find subsequent matches within one search text string.

Example 1.5 finds all matches of all subexpressions for each record. The data step creates two datasets; one to represent each subexpression in the match pattern. You will see in the output below that any patient receiving Xylometazoline appears in both datasets as this drug name is metazolamide, which by definition is also a zolamide.

EXAMPLE 1.5: MULTIPLE MATCHES
```
data zol meta;
set Prescriptions end=lastObs;
call PRXDEBUG(0);
if _N_=1 then pID = PRXPARSE("/(zol)|(meta)/I");
retain pID;
start = 1;
ostop = length(drug);
/*Loop through each successful match within string*/
call PRXNEXT(pID, start, stop, drug, pos, length);
do while (pos > 0);
length sub $4.;
sub = substr(drug, pos, length);
   if sub = "zol" then output zol;
   else output meta;
call PRXNEXT(pID, start, stop, drug, pos, length);
end;
drop start stop;
if lastObs then call PRXFREE(pID);
runc;
```

OUTPUT:
META:
```
Obs patient Drug pos length sub
 1 10006783 Xylometazoline 5 4 meta
 2 10031662 Xylometazoline 5 4 meta
 3 10056700 Xylometazoline 5 4 meta
 4 10062843 Xylometazoline 5 4 meta
 5 10408046 Xylometazoline 5 4 meta
```
ZOL:
```
Obs patient Drug pos length sub
 1 10006783 Xylometazoline 9 3 zol
 2 10031662 Xylometazoline 9 3 zol
 3 10056700 AEzolomide 3 3 zol
 4 10056700 Clotrimazole B.D. 9 3 zol
 5 10056700 Xylometazoline 9 3 zol
```

OTHER FUNCTIONS
There are some other RegEx functions not shown in an example but are worthy of a short mention. The PRXDEBUG, which writes debugging information in to the SAS log for each match performed for each record in a data step. I haven’t put any examples on the PRXCHANGE function (the syntax is mentioned below). In essence this performs a RegEx find and replace function. I’ll leave this one for you to work out for yourself, call it a self-study exercise.

EXAMPLE 1.6: USING THE PRXDEBUG FUNCTION
```
data PRXMATCH;
set RawData;
call PRXDEBUG(1);
pID = PRXPARSE("/Meta/I");
position = PRXMATCH(pID,drug);
call PRXFREE(pID);
```
Finally, I’ve provided a couple of real-world examples to show some advanced use of regular expressions. How to search an open text string for dosage information. I recommend you read the cheat-sheet which is shown in the references in order to determine what the match patterns are referring to.

First of all, let’s think about how many possible ways there are of writing dosage information into a free format text box. Let’s take one example; A physician wants to specify that a patient received Xylometazoline, 300mg, twice daily. Here are some legitimate examples of how this could be specified.

- Xylometazoline 300mg bd.
- Xylometazoline 300mg b.i.d.
- Xylometazoline 0.3g b.d.

The next example uses RegEx to extract all the relevant dosage information and cater for various ways of inputting such information in free text.

```plaintext
data GetDoseInfo;
  set RawData end=last;
  retain pid_freq pid_meas pid_unit;
  /*Create RegEx ID numbers*/
  if _N_ = 1 then do;
    pid_freq = PRXPARSE("/\[qtb\]{1}.i?d/I"); /* dose frequency*/
    pid_meas = PRXPARSE("/[0-9]\.?\[0-9]/I"); /* unit dose number*/
    pid_unit = PRXPARSE("/%|pct|mg|mcg|ml|mcl|g/I"); /* dose measurement*/
  end;
  /*USE PRXSUBSTR to get substring location for each piece of information*/
  call PRXSUBSTR(pid_freq, drug, pos_freq, len_freq);
  call PRXSUBSTR(pid_meas, drug, pos_meas, len_meas);
  call PRXSUBSTR(pid_unit, drug, pos_unit, len_unit);
  /*Add items to output variables*/
  length freq $6. measure 8. unit $5.;
  if pos_freq > 0 then freq = substr(drug, pos_freq, len_freq);
  if pos_meas > 0 then measure = input(substr(drug, pos_meas, len_meas), best.);
  if pos_unit > 0 then unit = substr(drug, pos_unit, len_unit);
  /*Free up system resources*/
  if last then do;
    call PRXFREE(pid_freq);
    call PRXFREE(pid_meas);
    call PRXFREE(pid_unit);
  end;
  keep patient drug freq -- unit;
run;
```

Finally, this example searches for personal contact information, such as email addresses or phone numbers. This can prove extremely valuable to ensure patient confidentiality is not breached by any physician providing such information in open ended text fields.

```plaintext
data FindContactInfo;
  set Prescriptions end=last;
  retain pid_email pid_tel;
  if _N_ = 1 then do;
```
pid_email = PRXPARSE("/\w+\[a-zA-Z_]++\./\?[\[a-zA-Z_]++\]?/I");
pid_tel = PRXPARSE("/\((\+\d)\?\((0)\)?\((s|\-)\d{3,5}(s|\-)\d+/I");
end;

/*USE PRXSUBSTR to get substring location for each piece of information*/
call PRXSUBSTR(pID_email, comments, pos_email, len_email);
call PRXSUBSTR(pID_tel, comments, pos_tel, len_tel);

eemail=(pos_email>0);
tel = (pos_tel >0);

if last then do;
call PRXFREE(pid_email);
call PRXFREE(pid_tel);
end;
run;

SAS REGULAR EXPRESSION SYNTAX GUIDE:

regular-expression-id = PRXPARSE(perl-regular-expression)
position = PRXMATCH(regular-expression-id or perl-regular-expression(9.1), source)
CALL PRXSUBSTR(regular-expression-id, source, position, <length>)
CALL PRXPAREN(regular-expression-id)
CALL PRXNEXT(regular-expression-id, start, stop, source, position, length);
CALL PRXDEBUG(n)
CALL PRXFREE(regular-expression-id)
var=PRXCHANGE(perl-regular-expression or regular-expression-id, times, source)

perl-regular-expression: specifies a character value that is a Perl regular expression.
regular-expression-id: specifies a numeric pattern identifier returned by the PRXPARSE function.
position: specifies the numeric position in source at which the pattern begins.
Source: specifies a character expression to search.
position: returns the position in source where the pattern begins.
length: returns the length of the found pattern.
start: the position at which to start the pattern matching in source.
stop: the last character to use in source.
n: toggles debug mode. 1=on, 0=off.
times: the number of times to search for a match and replace a matching pattern.
REGULAR EXPRESSIONS IN .NET

The string object type in .NET allows contains many built-in methods for manipulating strings, such as SUBSTRING, INDEXOF and REPLACE (which correspond to the SUBSTR, INDEX and TRANWRD functions in SAS). These are based only on basic string comparison. These are not powered by a RegEx engine. So we find ourselves in a similar situation as earlier. The string object in .NET has the power to manipulate string, but only within a limited scope. A RegEx engine will allow for more powerful string manipulation.

Rather like SAS, Visual Studio’s find and replace dialog contains the option to use Regular Expressions in it’s find and replace dialog box. You may also find this a useful 'quick reference' for looking up RegEx syntax.

FIGURE 2: VISUAL STUDIO .NET 2008 FIND AND REPLACE DIALOG BOX

The RegEx library in .NET (System.Text.RegularExpressions.RegEx) can handle each of the above tasks and a lot more. The examples below provide an example of using regular expressions to achieve the same endpoints as using the SUBSTRING, INDEXOF and REPLACE string functions.

EXAMPLE 2.1: BASIC STRING FUNCTIONS USING REGEX

```vbnet
Public Sub BasicRegExCheck(ByVal sender As Object, ByVal e As EventArgs) Handles btnMatch.Click
    'If textbox is blank, skip RegEx checks
    Dim sText As String = Request.Form("txtInput")
    If sText = "" Then
        Response.Write("No text specified")
        Exit Sub
    End If

    Response.Write("Performing a RegEx on the string <b>" & sText & "</b><br />")

    'Find the letter 't' in sText
    Dim rm As Match = Regex.Match(sText, "t", RegexOptions.IgnoreCase)
    Dim nIndexOf As Integer = rm.Index

    'Write result
    Response.Write("Character t " & _
        If(rm.Success, "found at position " & nIndexOf, "was not found.")) _
        & "<br />
    rm = Nothing

    'Now find any of l,o, [whitespace]
    rm = Regex.Match(sText, "(l|o|\s)")
```

---

PhUSE 2009
Dim nIndexOf3 As Integer = rm.Index

'Write Result
Response.Write("Character ", 
 & IIf(rm.Value = " ", "[whitespace]", rm.Value) & _ 
" was found at position " & nIndexOf3 & "<chr /><br />")

End Sub

Based on the above example, RegEx may not appear to offer much of an advantage over using ordinary string functions, but as with PRX functions in SAS, RegEx has the power to create much more advanced searches, as this paper will demonstrate. The RegEx.MATCH function returns a 'match object' which contains information about the match as well as the actual string found. One example is the Success property. This is a Boolean variable specifying whether a match was successful.

Furthermore, the RegEx.MATCHES function returns a collection of match objects. This way, more information about the match(es) can returned and many instances of a match can be returned with one function call.

The next example counts the number of words in a given text string. Note: the use of RegEx syntax here. The \b refers to a word boundary (i.e.: the start or end of a word) and the \w refers to a word character.

EXAMPLE 2.2: COUNT NUMBER OF WORDS IN A TEXT STRING
Public Sub CountWords(ByVal sender As Object, ByVal e As EventArgs) Handles _
btnWordCount.Click

'Search the letter 'w' in stext
Dim sText As String = Request.Form("txtInput")
If sText = "" Then
   Response.Write("No text specified")
   Exit Sub
End If

Dim rmc As MatchCollection = Regex.Matches(sText, "\b\w+\b", _
RegexOptions.IgnoreCase)
Response.Write("Performing a RegEx word count on the string <b>" & _
sText & "</b><br /><br />")
Response.Write("There are " & rmc.Count & " words.<hr /><br />")
End Sub

TEXT INPUT
We saw in the previous section that we can use RegEx in SAS to collect information from open-ended text boxes. In ASP.NET, we could control the data entry as to ensure that any information typed in to a textbox conformed to a desired format. The example below shows code to control for a page of an e-CRF for two textboxes, one containing a drug name, and one containing dose information in a specific format.

EXAMPLE 2.3: VALIDATING TEXT INPUT
Protected Sub btnSubmit_Click(ByVal sender As Object, ByVal e As _
System.EventArgs) Handles btnSubmit.Click

'Check D1 has at least 1 word in it
If Regex.Match(D1.Text, "\w").Success = False Then
   MsgBox("You must provide a response do question D1", , _
   "Error in Question D1")
End If

'Check D2 matches form of dose unit frequency
Dim sMatch as String = _
"(\d+?)(s*?(m*?\.*?c*?\.*?d*?\.*?)+\s*{(b|t|q|m)*?i*?d*?\.*?
\s*\s*\d*?\.*?)+"
If Regex.Match(D2.Text,sMatch).Success = False Then
   MsgBox("Your response do question D2 must be in the format of " & _
   "'dose' 'unit' 'frequency'", , "Error in Question D2")
End If
End Sub

SUBEXPRESSIONS
The use of parentheses to create subexpressions works in much the same way as subexpressions in SAS (You
PhUSE 2009

will also notice the use of subexpressions in the previous example. You can refer to the result of
subexpressions by using the format "$n", where n is the n’th subexpression. The entire match pattern can be
referred to as $0. This notation can be utilised in the RegEx.REPLACE function; the RegEx equivalent to the
string replace function in .NET.

Imagine you wish to display the contents of a file in a webpage, such as a SAS log file. You may wish to
highlight keywords such as NOTE, ERROR and WARNING, perhaps by using a larger font, or by colouring
the words as per the log window within SAS. Regular Expressions can handle such a request easily and return the
log as formatted HTML to a webpage.

EXAMPLE 2.4: READING AND FORMATTING THE CONTENTS OF A SAS LOG

Protected Sub ReadLogFile()
Try
'Upload the file to the server.
fileLog.SaveAs(Server.MapPath("~/sas.log"))

'Now it is on the server, we can read it and send it back,
Dim sR As New IO.StreamReader(Server.MapPath("~/sas.log"))
Dim sFile As String = sR.ReadToEnd()
sR.Close()

'Now do simple string replace to handle < and > signs
sFile = sFile.Replace("\", "\";"

'sFile = sFile.Replace("\", "\";"

'Now do RegEx to replace all keywords in one go.
sFile = Regex.Replace(sFile, "^\(ERROR|NOTE|WARNING\)\$\</span>$1</span>"", RegexOptions.Multiline)

'Handle line breaks
sFile = sFile.Replace(Chr(13), "")
sFile = sFile.Replace(Chr(10), "<br />")

'Write output to webpage
Response.Write("<pre>" & sFile & "</pre><hr />")
Catch EX as Exception
End Try
End Sub

Here we can see the RegEx.REPLACE function above contains a subexpression in parentheses. The result of
this subexpression is used in the replace string by the notation of $1. (In this case, any of "ERROR", "NOTE" or
"WARNING"). As a result, we can find and replace with formatted HTML, all instances of all specified keywords
in one line of code.

It is entirely possible to take this example one step further and colour the entire NOTE, ERROR or WARNING
in the same way that an interactive SAS session does. However, rather than going through this now, I’ll leave you
to work this one out for yourselves.

Finally, I have given an example below on how to check for e-mail addresses to ensure that any specified
address fits the appropriate format. The function also breaks down the email into separate components
(username, domain, suffix).

EXAMPLE 2.4: CHECKING AN EMAIL ADDRESS

Public Sub doEmailCheck()
'Set up match pattern.
Dim sMatch As String = "^[\w]+@[\w.-]+\.[\w.-]+\$"

'Get email address entered by user.
Dim sEmail As String = Request.Form("email")

'Do RegEx match and return result
If Regex.Match(sEmail, sMatch).Success Then
Response.Write("<p>Your email address is: " & Regex.Replace(sEmail, _
       sMatch, "$0") & "<br />")
Response.Write("Your user name is: " & Regex.Replace(sEmail, sMatch, _

"$1" & "<br/>")
Response.Write("Your Domain name is: " & Regex.Replace(sEmail, sMatch, _
"$2") & "<br/>")
Response.Write("Your Suffix name is: " & Regex.Replace(sEmail, sMatch, _
"$3") & "<br/>")
Response.Write("</p>"
Else
Response.Write("<p>Invalid email</p>"
End If
End Sub

.NET REGULAR EXPRESSION SYNTAX GUIDE

RegEx.match(input, pattern, [options]) as match
RegEx.matches(input, pattern, [options]) as matchCollection
RegEx.Replace(input, pattern, Replace, [Options]) as String

input: A string variable containing the text you wish to search
pattern: A string variable containing the match pattern to apply to the input string
Options: Optional: RegEx behaviour options.
Replace: A text string to replace the pattern when found in input.
CONCLUSION

Regular Expressions have the power to search for patterns of text, rather than exact text-strings thus offering a host of possibilities to a programmer, regardless of the language and/or environment they are programming for. Even very large amounts of text, or text-based data can be analyzed very quickly in order to identify important elements. This can be used to mine large datasets or to enhance the look of a report.

Real-world examples include ensuring no data identifies individual patients in a trial or GP records. RegEx can be used in .NET or JavaScript to enhance web reports, by recognising and highlighting important information such as postcodes or telephone numbers or any specified keywords.

Whilst Regular Expressions offer massive power to the programmer, they can also be very complex and care must be taken to keep on top of the complexity and rigorous quality controls should be maintained.

It is worth noting that Regular Expressions are available in almost any language. One notable absentee from this paper is JavaScript regExp functions. ASP.NET covers RegEx in web pages and in my personal experience, most use of RegEx for web languages tends to occur server-side. However, there are valid uses for RegEx to be executed in a client’s browser as well.

REFERENCES


CONTACT INFORMATION

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

Paul Shannon
RTI Health Solutions
Williams House
Manchester Science Park
Lloyd St North
Manchester
M15 6SE
+44 (0) 161 232 4926
+44 (0) 161 232 3409
pshannon@rti.org
www.rtihs.org