

Introducing Regular Expressions

by Craig Murphy

I'm writing this whilst travelling from a small town near Edinburgh to Aberdeen. I've just awoken after falling asleep behind the wheel. It's hard enough driving from Edinburgh to Aberdeen but add the complexity of typing and a cup of coffee, and you have a recipe for disaster. So, to avoid disaster, and so that I can complete this article, I'm travelling by train.

The interesting thing about this [great?] train journey is the fact that the comparable car journey takes just as long. The road north is littered with speed cameras thus ensuring that all but the Hakkinen-wannabees trundle along at a sensible pace. Although, the fact the train journey takes just as long, it makes me wonder if there are speed cameras at the side of the railway too.

Introduction

Over the past few months I've been delivering a number of XML presentations – under the catchall title of Using the XML Recommendations in Delphi. My last session, in November 2002, was all about using XML Schema in a Delphi application.

As part of that presentation I had a slide that explained how we could use XML Schema facets to create and extend data types. For example, using XML Schema, it is possible to create/define a user-defined string data type that is restricted in length. We could take the restriction further, and ensure that the string data type is a specific format. XML Schema allows us to perform such validation using regular expressions. Later in this article, I'll discuss the particular regular expression that I used in the said presentation.

We've all heard of regular expressions; if you've used Unix in the past, `grep` is probably a familiar tool. Indeed, if you've used the scripting language Perl, it's very likely that you will have used some form of regular expression(s).

The regular expression syntax has been the subject of many books and countless articles. Indeed, whilst I was busy ranting on about how cool XML Schema and regular expressions actually are, it wasn't until a fellow UK-BUG member in the audience asked about the regular expression syntax. Then I realised just how little I actually know about regular expressions. After all, every time I need to create a regular expression, with the exception of all but the most trivial, I find myself consulting a crib sheet.

What is a Regular Expression?

Regular expressions provide a concise and flexible notation for matching and replacing patterns of text within a body of text. Some might say that regular expressions are concise and cryptic, perhaps because most regular expressions are built from a combination of metacharacters such as `^$*+?`. The actual regular expression itself is known as a pattern – over the course of this article, I'll use the phrase pattern and regular expression interchangeably.

Here's an analogy. The `pos` function in the System unit is useful, however it is the Mini whereas regular expressions might be likened to the Rolls Royce of string manipulation. Similarly, the `StringReplace` function in SysUtils is useful, but regular expressions take string replacement to another level.

A Trivial Example

To demonstrate regular expressions, I have put together a small example application. I will be using the Microsoft VBScript_RegExp_55 type library – it provides a regular expression object that is useable in Delphi. Figure 1 presents a screenshot of the type library being imported into Delphi.

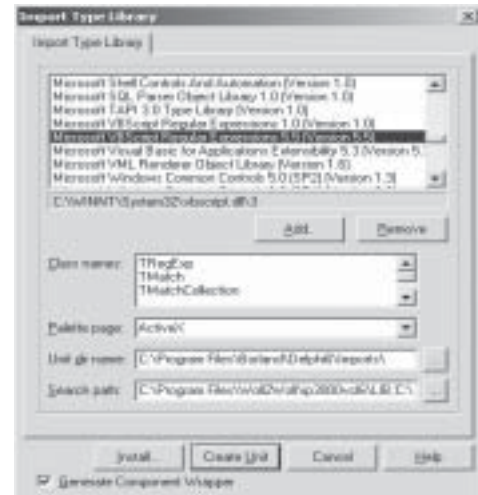


Figure 1 – Type Library Import

This trivial example is based around the controls on the Simple Matching tab of the example application. Listing 1 presents the code behind the Apply button. Given some text that contains integers, the regular expression `[0-9]+` will allow us to extract the numeric integer content. Figure 2 presents the Simple Matching tab after the Apply button has been pressed.



Figure 2 – Simple Matching

Not So Trivial

The `^` metacharacter is used to represent the beginning of input. Consider the regular expression `^“BUG.*”`. The `.` Metacharacter matches any single character, except a new

line character. The ‘*’, as you might expect, is the wildcard, and matches any number (zero or more) of the previous characters – in this case, because the ‘.’ is the previous character, ‘.*’ will match everything that comes after ‘BUG’. Thus “^BUG.*” would match “BUG members”, but not “members of the BUG”.

However, the regular expression “.*BUG\$” would not match “BUG members”, but would match “members of BUG”. The metacharacter ‘\$’ is used to represent the end of input. You can try these expressions via the Scratch tab of the example application.

Character Sets

Consider the following strings: A123, 234, C456. I’ve deliberately missed the ‘B’ from the second string.

It would be useful to be able to scan these strings to pick out strings similar to A123, i.e. an alphabetic character, followed by some numeric content. Alphabetic characters are represented using character sets enclosed in square brackets. Assuming the alphabetic character was allowed a range of A through to Z, we could represent this set like this: [A-Z]. Numeric sets work in the same way; the range 0 to 9 is the pattern [0-9]. Thus given the pattern [A-Z][0-9]+, we can match the two strings A123 and C456. Figure 3 demonstrates this pattern in action.



Figure 3 – Character Sets

If we augmented the strings to be A123, B234, C345, we could use the pattern [A,C][0-9]+ to match A123 and C345, to give us the same result. Once again, the Scratch tab of the example application will allow you to try these expressions for yourself.

Negating Character Sets

The use of character sets is a very powerful means of performing complex character-level matching. Given the strings 10t, 20t, 30g, 400g, a useful task might be to match all the strings without a ‘t’ at the end. The pattern [0-9]{1,}[^t]\$ achieves this. There are a couple of new features in this pattern. Here’s a breakdown of the pattern:

[0-9]	matches the numeric content
{1,}	indicates that the previous [0-9] must contain at least 1 character in the range [0-9]. The ‘,’ indicates that it can contain more than one [0-9] character
[^t]\$	indicates that we do not want the end of the input to be the letter ‘t’.



Figure 4 – Selective Matching

The metacharacter ‘^’ is used to indicate negation. Figure 4 presents a screenshot demonstrating this particular pattern in action. We could force the pattern to return all the strings with a ‘g’ at the end – in which case the pattern would be: [0-9]{1,}[g]. In this case, both patterns return the same matches, however be aware that it is possible to be blinkered into thinking that a given pattern is perfect. Sometimes you will find additional matches appearing, often because the pattern was built up using known test data – as I’m sure you know, live data can be very different from test data.

Advanced String Extraction

Consider the following sentence: “The first 12-345 caused 23-456 to start looking at the third 34-567 and involved 45-678”.

I’m sure you can guess where this is going – we’re going to extract the “12-345” codes. The regular expression to achieve this is \d{2}-\d{3}. However, in addition to extracting these codes, wouldn’t it be nice if we could extract the component parts “12” and “345”?

By enclosing parts of a regular expression in round brackets, we can extract the component parts – these are known as parenthesised matches. Typically these parenthesised matches are stored in variables/properties “\$1” through to “\$9”. These work well in scripting languages and in Unix/Linux shell environments, however they do not work very well in Delphi. Thankfully the Microsoft Regular Expression object exposes these properties through a SubMatches collection.

Using the pattern (\d{2})-(\d{3}) will not only match 12-345, but it will extract the “12” and the “345” in to the “\$1” to “\$9” properties. However unlike the scripting languages where regular expressions are widely used, accessing the \$1 to \$9 properties in Delphi requires a little more work. Listing 2 presents the interface definition for the IMatch2 – notice the SubMatches property. The SubMatches property allows us to extract the components inside each set of parenthesised matches.

Listing 3 demonstrates the use of the SubMatches property. Figure 5 depicts the example application after the Advanced Matching tab’s Apply button has been clicked. Now that, in my opinion, is pretty cool.



Figure 5 - Advanced Matching

Extracting the component parts could prove fairly useful, particularly if the component parts are key fields in your database application. Extracting the component parts is also useful in situations where additional validation is required. For example, consider the match 12-345, with component parts “12” and “345”. We could validate that one or both of the component parts are in a chosen range – perhaps in the set [12,23,34,45]?

Replacing Text

The Regular Expression Object doesn't just perform matching; it can perform text replacement too.

Listing 4 presents the code required to perform some basic string replacement – in this case we are replacing all occurrences of the word ‘Blue’ with ‘Red’ (note the capitalisation). Fairly trivial, and something that Delphi provides support for via the `StringReplace` function in `SysUtils`. Figure 6 presents the screenshot after some basic string replacement has taken place.



Figure 6 - Simple Replacement

`StringReplace`, however, is limited to matching and replacing strings, not the patterns that regular expressions are famous for. The regular expression object allows us to extend the string replacement functionality via the use of the standard patterns and metacharacters that I have been using elsewhere in this article. Given the string “How now, brown cow”, the pattern `[n|c|ow]` can be used to change the

string into “How now pat, brown cow pat”. In other words, any word beginning with ‘n’ or ‘c’, that ends in the ‘ow’ is considered a match.

The replace pattern “`+$ pat`” indicates that whenever a match is found, output the match “`+$`”, then add the letters ‘pat’. Listing 5 presents the code that performs more advanced string replacement; Figure 7 presents the associated screenshot.



Figure 7 - Augmented Replacement

Reordering Matches Using Parenthesised Matches

Earlier in this article I mentioned parenthesised matches, and how their contents are stored in the \$1 to \$9 properties/variables. I also mentioned that access the \$1 to \$9 properties required a little bit of work. The replace operation allows us to make use of the \$1 to \$9 properties without any extra work.

Given the string “North South East West”, the pattern `(\S+)\s+(\S+)\s+(\S+)\s+(\S+)` would match each collection of non-whitespace (`\S+`) characters separated by spaces (`\s`). “North” would appear in \$1, “South” in \$2, etc. Assuming the replace string was “`$4 $3 $2 $1`”, we could reverse the word order in just a few lines of code. “West East South North” would be our expected output. Figure 8 demonstrates word swapping in action. Listing 6 presents the code fragment required to perform the word swapping.



Figure 8 - Swapping Words

Postcode Matching

The regular expression that I used as part of my XML Schema (mentioned earlier in this article) looked like this: `[A-Z]{2}\d\s\d[A-Z]{2}`

This expression essentially allows us to match the some UK postcodes. However, had I examined this expression in more detail I would have noticed a fundamental flaw. Here is how this expression works:

<code>[A-Z]{2}</code>	matches two letters
<code>\d</code>	matches one number
<code>\s</code>	matches a space
<code>[A-Z]{2}</code>	matches two letters

So this allows such postcodes as AB1 1XR to be matched, but not AB10 1XR, or even G1 1XR. Perhaps a better postcode matching expression would be: `([A-Z][A-Z]{2})([0-9][0-9]{2})\s\d[A-Z]{2}`. Here's how this expression works:

<code>([A-Z][A-Z]{2})</code>	matches one letter OR () two letters
<code>([0-9][0-9]{2})</code>	matches one number OR () two numbers
<code>\s</code>	matches a space
<code>\d</code>	matches one number
<code>[A-Z]{2}</code>	matches two letters

Figure 9 depicts the Postcode matching tab in action. Whilst the regular expression matches the postcode syntax, the verification of the matches is the responsibility of a secondary process – I'm sure a database of valid prefixes and postfixes, such as AB, XR, etc. exists. Once your regular expression has confirmed that the string in question might be a postcode, it's up to you to perform some secondary validation. Nonetheless, this kind of postcode matching is useful inside web forms, for example, where we may wish to perform some field-level validation.



Figure 9 - Postcode Matching

In retrospect using a postcode regular expression in a presentation is a bad idea – you would be surprised at the

debate that followed! It would appear that there are a handful of peculiar [notably governmental and defence estates] postcodes that do not match the regular expression.

Alternatively, we could use `[A-Z]{1,2}[0-9]{1,2}\s\d[A-Z]{2}`...you can see why this is a regular expression that can be expressed in more ways than one. In fact, whether validating postcodes using regular expressions or not, the whole postcode validation topic always seems to spark off debate!

Summary

Earlier in this article, I mentioned that many books have been written on the subject on regular expressions. Apart from the definitive Perl books (camel and llama), I have read Friedl's Mastering Regular Expressions book. I say read, I really mean dipped into: it's nearly 800 pages long! Therefore, it's fair to say that an article this size, indeed even a magazine this size, couldn't cover every aspect of regular expressions. I hope this article has provided you with a flavour of what can be achieved with regular expressions. Regular expressions are a fundamental part of the Perl language; their practicality and use cannot be understated

I noticed a pitfall whilst preparing this article. Using the Scratch tab of the example application, I entered a pattern that looked correct. After clicking on the Apply button, the results were not what I was expecting. Thankfully I spotted the problem fairly quickly – an extra space in the pattern caused the match to fail. Check your patterns carefully – the smallest mistake can cause a regular expression to fail or return odd results. Also, as the postcode example demonstrated, regular expressions offer more than one way to skin a cat.

Finally, be aware that the regular expression syntax varies from between implementations, so the regular expressions you have seen in this article may not necessarily work with your particular toolset.

My return to Inverkeithing (that was the small town near Edinburgh), some thirteen hours later, causes me to collapse. Thankfully, my television is used to this, and unlike my new fridge, it isn't connected to the Internet, so it doesn't call out the emergency services. Nor does my wife. And, I nearly forgot to mention: the wheel that I fell asleep at was, in fact, the wheel of a mouse.

Resources

- Mastering Regular Expressions, Jeffrey E.F. Friedl, published by O'Reilly
- Search for "Regular Expression Syntax" using the Delphi Information Library
- There is a Delphi-based regular expression unit available here: http://www.preview.org/software/tips/reg_expr.zip



Craig is an author, developer, speaker and Dilbert Evangelist – he specialises in all things XML, particularly SOAP and XSLT. He can be reached via e-mail at: Craig@isleofjura.demon.co.uk, or via his web site: <http://www.craigmurphy.com> (where you can also find the source code and PowerPoint files for all of Craig's UK-BUG articles and presentations).

```

procedure TfrmRegExp.btnMatchClick(Sender:
                                TObject);
var reRegExp      : IRegExp2;
    i              : integer;
    mcMatchCollection : MatchCollection;
    mcMatch        : Match;
begin
    reRegExp := CoRegExp.Create;

    reRegExp.Pattern      := '[0-9]+';
    reRegExp.Global       := true;
    reRegExp.Multiline    := true;

    mcMatchCollection := reRegExp.Execute
                        (mText.Text) as MatchCollection;

    mMatches.Clear;
    for i := 0 to mcMatchCollection.Count - 1 do
        begin
            mcMatch := mcMatchCollection.Item[i] as
                        Match;
            mMatches.Lines.Add(mcMatch.Value);
        end;
end;

```

Listing 1 – Simple Matching

```

IMatch2 = interface(IDispatch)
    ['{3F4DACB1-160D-11D2-A8E9-00104B365C9F}']
    function Get_Value: WideString; safecall;
    function Get_FirstIndex: Integer; safecall;
    function Get_Length: Integer; safecall;
    function Get_SubMatches: IDispatch; safecall;
    property Value: WideString read Get_Value;
    property FirstIndex: Integer read
        Get_FirstIndex;
    property Length: Integer read Get_Length;
    property SubMatches: IDispatch read
        Get_SubMatches;
end;

```

Listing 2 – IMatch2 interface definition

```

procedure
TfrmRegExp.btnAdvancedApplyClick(Sender:
                                TObject);
var
    i, j: integer;
    reRegExp: RegExp;
    mcMatchCollection: MatchCollection;
    mMatch: Match;
    smSubMatches: ISubMatches;
begin
    reRegExp := CoRegExp.Create;

    reRegExp.Pattern      := '(\d{2})-(\d{3})';
    reRegExp.IgnoreCase   := true;
    reRegExp.Global       := true;
    mcMatchCollection := reRegExp.Execute
                        (mSource.Text) as MatchCollection;

    for i := 0 to mcMatchCollection.Count - 1 do
        begin
            mMatch := mcMatchCollection.Item[i] as
                        Match;
            mResults.Lines.Add(mMatch.Value);
            smSubMatches := mMatch.SubMatches as
                        SubMatches;

            for j := 0 to smSubMatches.Count - 1 do
                begin
                    // Secondary validation
                    //if (j = 0) then if StrToInt
                    //(smSubMatches.Item[j]) in [12,23] then
                    // mResults.Lines.Add('in [12,23]');

                    mResults.Lines.Add('    ' + '$' +
                        inttostr(j + 1) + ' = ' + VarToStr
                        (smSubMatches.Item[j]));
                end;
            end;
end;

```

Listing 3 – Advanced Pattern Extraction

```

procedure TfrmRegExp.btnReplaceClick(Sender:
                                TObject);
var reRegExp      : IRegExp2;
    sStr          : string;
    strReplace    : OleVariant;
begin
    reRegExp := CoRegExp.Create;

    // Original string
    sStr := 'Hello big Blue world';

    // Replace 'Blue' with 'Red'
    reRegExp.Pattern := 'Blue';
    strReplace       := 'Red';
    reRegExp.Global  := false;

    edtBefore.Text := sStr;

    sStr := reRegExp.Replace(sStr, strReplace);

    edtAfter.Text := sStr;
end;

```

Listing 4 – Basic string replacement

```

procedure TfrmRegExp.btnAugmentClick(Sender:
                                TObject);
var reRegExp      : IRegExp2;
    sStr          : string;
    strReplace    : OleVariant;
begin
    reRegExp := CoRegExp.Create;

    // Original string
    sStr := 'How now, brown cow';

    // Augment 'How now cow'
    reRegExp.Pattern := '[n|c]ow';
    strReplace       := '$+ pat';
    reRegExp.Global  := true;

    edtBefore.Text := sStr;

    sStr := reRegExp.Replace(sStr, strReplace);

    edtAfter.Text := sStr;
end;

```

Listing 5 – Augmented string replacement

```

procedure TfrmRegExp.btnSwapClick(Sender:
                                TObject);
var reRegExp      : IRegExp2;
    sStr          : string;
    strReplace    : OleVariant;
begin
    reRegExp := CoRegExp.Create;

    // Original string
    sStr := 'North South East West';

    // Reverse word order
    reRegExp.Pattern :=
'(\S+)\s+(\S+)\s+(\S+)\s+(\S+)\s+(\S+)';
    strReplace       := '$4 $3 $2 $1';
    reRegExp.Global  := true;

    edtBefore.Text := sStr;

    sStr := reRegExp.Replace(sStr, strReplace);

    edtAfter.Text := sStr;
end;

```

Listing 6 – Swapping Words

this code is on www.richplum.co.uk/html/downloads.asp