Using regular expressions

Michaël Van Canneyt

February 23, 2021

Abstract

Regular expressions should be a part of every programmer's toolbox. Once grasped, they are easy to use and can accomplish otherwise difficult or time-consuming tasks for you. A short introduction in Regular Expressions.

1 Introduction

Regular expressions have been around for almost 70 years, and are commonly used to perform search and replace operations. They can be found in most IDEs or text editors. Still, surprisingly many people do not know how to use them. Therefor we give a simple introduction on how to use them. I

n what follows, the POSIX conventions for regular expressions are described. Some editors and IDEs use slightly different conventions.

A good overview of all possibilities (and in particular the Perl Compatible Regular Expressions (PCRE)) can be found on the wikipedia page:

`https://en.wikipedia.org/wiki/Regular_expression`

Many programmers will be familiar with pattern matching for filenames:
A mechanism used in the `FindFirst`/`Findnext` calls: They allow for simple wildcard matching, where `*` stands for an arbitrary sequence of characters, and `?` stands for a single character.

Thus ?.pas will match all filenames of 1 character long with extension .pas, and System.*.pas will match all filenames that start with the System namespace.

Similarly, many programmers fluent in SQL will know the `LIKE` operator that allows the use of % and _ wildcards.

So, how to match these filenames with regular expressions ? Like in filename wildcard matching, letters and digits (and many symbols) in a regular expressions match only that letter or digit. But a regular expression can contain some special characters. Here are some of them:

^ The ^ matches the beginning of the input string (usually a line of text).

. The dot matches any single character. It is equivalent to the ? wildcard in filename matching.

***** The asterisk matches the previous element any number of times (also zero).

$ The $ matches the end of a search string (usually a line of text).

If you want to include a literal dot or asterisk in your regular expression, you must escape them with a backslash.

With these 2 characters, we can already emulate the filename pattern matching. The following regular expression:

```
^.\.pas$
```

This will match all filenames with a name of 1 character long, and extension .pas. Likewise, the following:

```
^System\..*\.pas$
```

will match all filenames that start with the System namespace.

However, you can do more with Regular expressions than just emulate file name wildcard matching.

For instance, you might want to search for files that are either a .pas file or a .dcu file. To be able to do this, we need 2 more concepts in Regular expressions:

( ) The brackets mark a subexpression.

| The vertical bar matches the element before or the element after.

To find either a .pas file or a .dcu file, we can combine these two with

```
^.*\.pas|dcu$
```

If you want to find files that end on -1, -2 or -3, then you can use the following operator:

[ ] This will match any letter between the brackets.

[^] This will match any letter not between the brackets

You can specify a range of characters by separating them with a dash, for example a-z or 0-9, just as you specify a range in Pascal.

Thus, the following expression will match all .txt filenames that end on -1, -2 or -3:

```
^.*-[123]\.txt$
```

The following will also do:

```
^.*-[1-3]\.txt$
```

The POSIX standard specifies also several named ranges of character sets. For example [:alpha:] denotes all alphanumerical characters, whereas [:digit:] denotes all digits and [:space:] denotes whitespace. Some languages (notably, Perl) or editors use escaped characters for this: \w means words, \s means spaces etc..

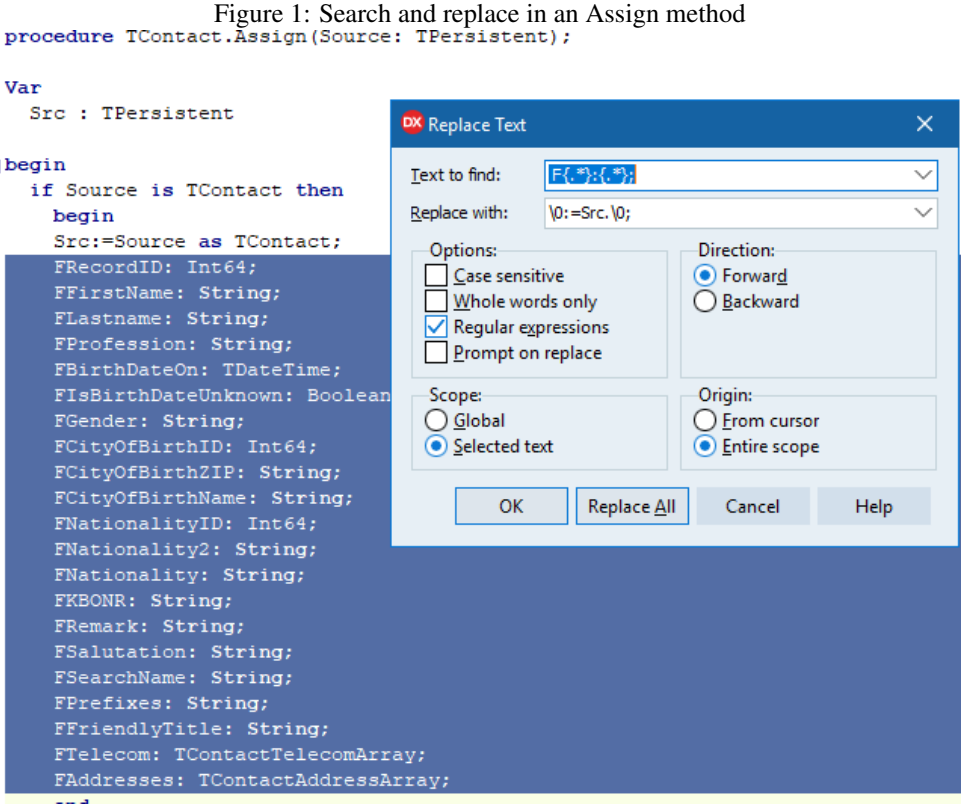
Note that there are no operators to specify case-insensitive or case sensitive match: this kind of property of a search must be specified separately.

2 Search and replace

The Delphi IDE and the Lazarus IDE both allow you to use regular expressions: the search (and replace) dialog has a checkbox 'Regular expression'. When checked, the IDE will interpret the text in the search box as a regular expression.

For example

Figure 1: Search and replace in an Assign method




```
^ *F.* : .*;$
```

Will match all field or variable definitions starting with an F. (they must have a space before and after the colon)

```
Items\[.*\]:=
```

will find all assignments to an Items array element.

In the IDE you will usually want to use the replace functionality when using regular expressions. When you want to replace the found matches with something else, you will most likely want to refer to part of the matched term in your replacement text. In order to do so, you can use the { } brackets to group particular parts in your expression. You can refer in your replacement text to these groups by \0 to \9.

A good example where this can be useful is the `Assign` method, which must be implemented in descendants of `TPersistent` to copy all properties from one to another instance.

The text and the search and replace dialog to do so are shown in figure figure 1 on page 3.

When implementing such a method you start by copying all private fields that make up your persistent class to the body of your `Assign` method.

Keep the list selected, and in the search and replace dialog, you can enter the following:

```
F{.*} : {.*};
```

This will match all field definitions in the selection. Because we used the grouping operators, we can use the group in the replace text:

Figure 2: The completed Assign method

```
if Source is TContact then
  begin
  Src:=Source as TContact;
  RecordID:=Src.RecordID;
  FirstName:=Src.FirstName;
  Lastname:=Src.Lastname;
  Profession:=Src.Profession;
  BirthDateOn:=Src.BirthDateOn;
  IsBirthDateUnknown:=Src.IsBirthDateUnknown;
  Gender:=Src.Gender;
  CityOfBirthID:=Src.CityOfBirthID;
  CityOfBirthZIP:=Src.CityOfBirthZIP;
  CityOfBirthName:=Src.CityOfBirthName;
  NationalityID:=Src.NationalityID;
  Nationality2:=Src.Nationality2;
  Nationality:=Src.Nationality;
  KBONR:=Src.KBONR;
  Remark:=Src.Remark;
  Salutation:=Src.Salutation;
  SearchName:=Src.SearchName;
  Prefixes:=Src.Prefixes;
  FriendlyTitle:=Src.FriendlyTitle;
  Telecom:=Src.Telecom;
  Addresses:=Src.Addresses;
  end
```

```
\0 := Src.\0;
```

This will replace something like

```
FFirstName : String;
```

With:

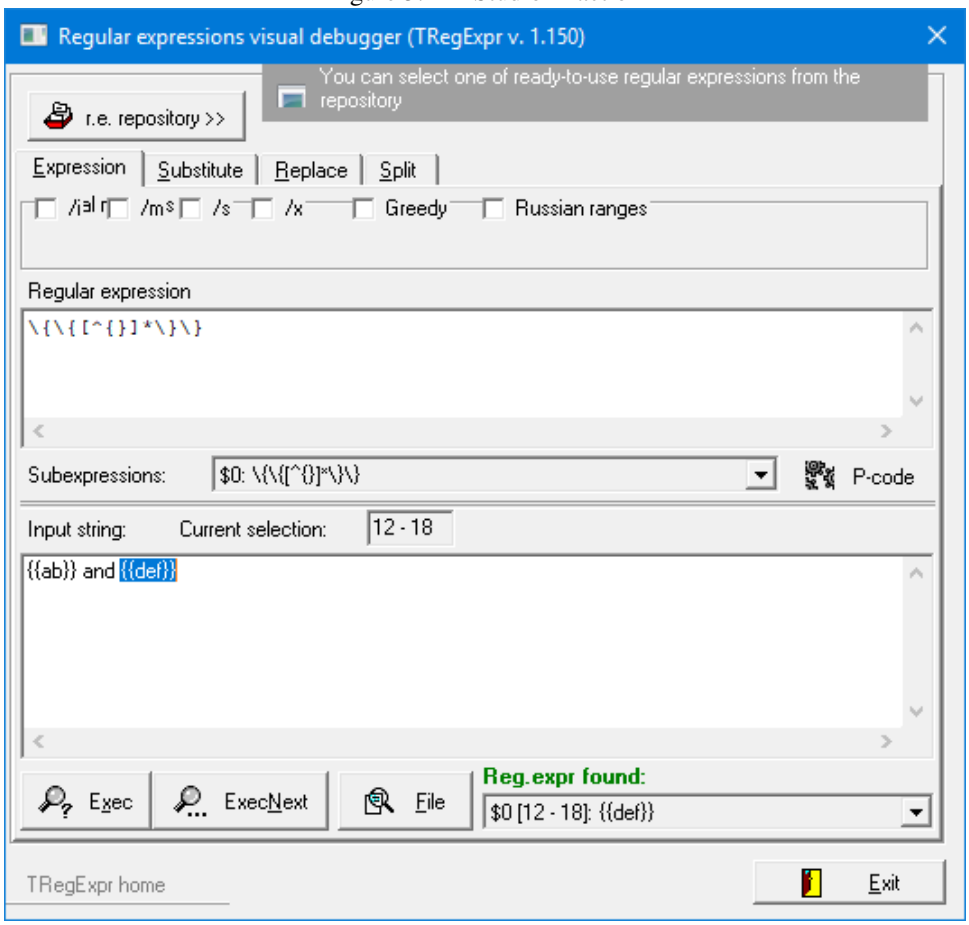
```
FirstName := Src.FirstName;
```

When we hit 'Replace all' the IDE will do all replacements, and the result can be seen in figure figure 2 on page 4.

3 Using regular expressions in Delphi or Lazarus

Newer versions of Delphi come with the `RegularExpressionsCore` unit which contains the `TPerlRegEx` class. It uses an external library which contains the actual implementation of the regular expression engine. The commercial tool `RegexBuddy` is a GUI

Figure 3: RE Studio in action



program that assists you in creating a regular expression and can generate the necessary Delphi code for handling the regular expresssion.

There is also an open source all-native Object Pascal implementation of regular expressions:

`https://github.com/andgineer/TRegExpr`

It is implemented by Andrey V. Sorokin, has been around for a long time, works for all versions of Delphi, and also for Lazarus/Free Pascal.

In fact, Free Pascal ships with a version of `TRegexpr`. Therefor, we'll use this implementation to demonstrate how regular expressions can be used.

The `TRegexpr` source code comes with a demo application `REStudio`, which is similar in purpose to RegexBuddy. It needs some care, but after some small fixes it can be made to work in Delphi. The necessary fixes to let it compile in Lazarus and Delphi Rio (and newer) have been pushed to the following repository:

`https://github.com/mvancanneyt/TRegExpr`

When started, it looks like figure 3 on page 5. The working is quite simple: in the top memo, you can enter a regular expression. The program will give you feedback, so you will immediately see if the expression is valid. The bottom memo allows you to enter a

text, and with the Exec.../ExecNext buttons you can let the regular expression execute it's action.

4 Using search and replace in your program

The `TRegExpr` class is not a component. It must be created in code, and has the following properties:

Expression The regular expression.

ModifierStr (string) set the regular expression modifier options as a string (as one would in Perl or Javascript). The string consists of a concatenation of the uppercase letter of the next properties.

ModifierI (boolean) The I modifier, for case insensitive search.

ModifierR (boolean) The R modifier: enables some extented Russian characters.

ModifierS (boolean) The S modifier: when True, the . character also matches a newline.

ModifierG (boolean) The G modifier: turn on greedy matching.

ModifierM (boolean) The M modifier: enables multiline mode. In that mode, the ^ and $ characters match the beginning and end of each line in the search text, not just the beginning and end of the whole search text.

ModifierX (boolean) The X modifier: Allows you to use comments in your regex using the # character.

SubExprMatchCount (integer) the amount of sub expression matches after an Exec or ExecNext.

MatchPos (array of integer) a 0-based array giving you the position of the subexpressions in the search string. The 0-th entry is the position of the whole regular expression match.

MatchLen (array of integer) a 0-based array giving you the lengths of the subexpressions in the search string. The 0-th entry is the length of the whole regular expression match.

Match (array of string) a 0-based array giving you the actual matches of the subexpressions in the search string. The 0-th entry is the text of the whole regular expression match.

CompilerErrorPos if an error occured during compilation of your regular expresion, this gives the position of the error.

There are many more propertyies, but the above ones are the main properties.

The class also has the following important methods:

```
function Exec(const AInputString: RegExprString): boolean;
function ExecNext: boolean; overload;
function ExecNext(ABackward: boolean): boolean; overload;
function Substitute(const ATemplate: RegExprString): RegExprString;
procedure Split(const AInputStr: RegExprString; APieces: TStrings);
function Replace(const AInputStr: RegExprString;
```

```
      const AReplaceStr: RegExprString;
      AUseSubstitution: boolean = False) // ###0.946
      : RegExprString; overload;
function ReplaceEx(const AInputStr: RegExprString;
      AReplaceFunc: TRegExprReplaceFunction): RegExprString;
```

Exec Starts a search for matches on `aInputString`. Returns `true` if a match was found.

ExecNext Will search for the next occurence. If `aBackWard` is true, the search is done backwards. Returns `true` if a match was found.

Substitute will replace the text in `ATemplate` with the currently found matches: The template can contain placeholders for the subexpressions: `$&` or `$0` is replaced by the whole match, and `$1` till `$n` are replaced by their respective subexpression.

Split will split `aInputStr` into various items split by the regular expression matches, and put all found items in `aPieces`.

Replace will replace `aInputStr` all the matches with the `AReplaceStr` term. If `AUseSubstitution` is true, the replacement term is not treated as a regular text, instead it is used as a pattern for the `Substitute` to replace all items. The function returns the input string with all matches replaced.

ReplaceEx will replace `aInputStr` all the matches with the result of the `AReplaceFunc` callback; the callback is called for each match in the input string, and the result of the allback is then used to replace that particular match in the input string. This is a very powerful mechanism.

So, how can we use this class to implement search and replace ? To demonstrate how to search, we create a small application in Lazarus with a memo (`mdemo`), a button (`bsearch`) and a `TFindDialog` (`FDRE`). The `OnClick` handler of the button shows the find dialog:

```
procedure TMainForm.btnSearchClick(Sender: TObject);
begin
  FDRE.Execute;
end;
```

This is nothing special.

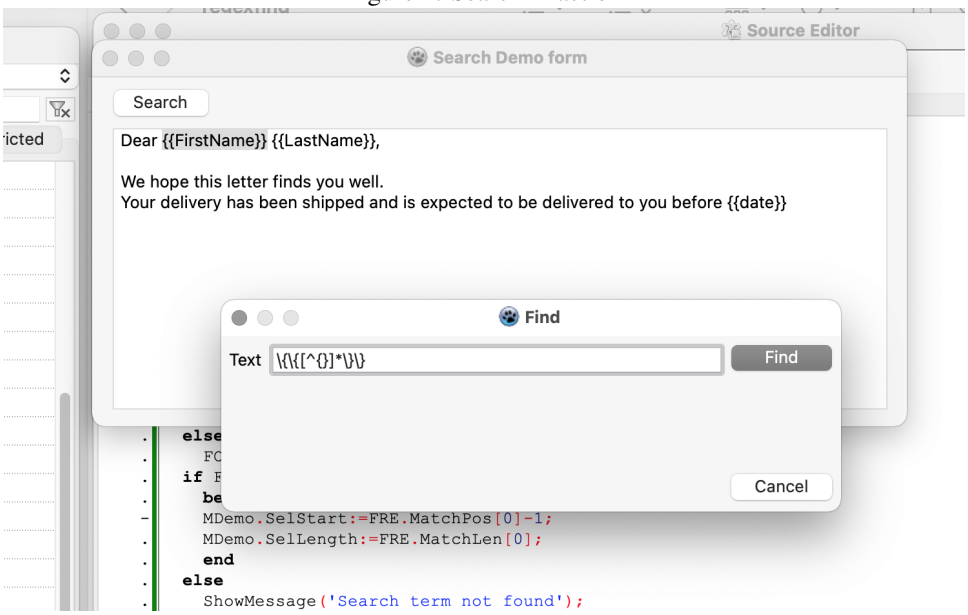
The `OnFind` event handler of the find dialog is called every time the user clicks the `Find` button in the find dialog. Here we implement the logic for searching.

Because the TRegexpr has 2 methods to search, depending on whether it is the first time you search (`Exec`) or it is a next search (`NextExec`), we must keep track of whether it is a new search or the next time a search is done.

To do so, we store the regular expression in a variable, and if the regular expression has changed, we assume it is a new search:

```
procedure TMainForm.FDREFind(Sender: TObject);

var
  Found : Boolean;

begin
  If FDRE.FindText<>FLastRE then
```

Figure 4: Search in action




```
    begin
    FreeAndNil(FRE);
    FRE:=TRegexpr.Create;
    FLastRE:=FDRE.FindText;
    FRE.Expression:=FLastRE;
    Found:=FRE.Exec(MDemo.Lines.text);
    end
  else
    Found:=FRE.ExecNext;
  if Found then
    begin
    MDemo.SelStart:=FRE.MatchPos[0]-1;
    MDemo.SelLength:=FRE.MatchLen[0];
    end
  else
    ShowMessage('Search term not found');
end;
```

When we fiind a match, we use the `MatchPos` and `MatchLen` properties to set the selection of the memo. (note that `Matchpos` is 1-based, and `SelStart` is 0-based) If no match is found, we show a message, but we don't close the dialog. Note that if the user changes the text between searches, the matches will no longer correspond to actual positions, so in a real program, it would be best to make the memo readonly while the search is active.

The result of this can be seen in figure figure 4 on page 8.

To demonstrate the power of the `TRegexpr` class, we'll also implement a small algorithm that transforms a template text with some data placeholders to a ready-to-read text: a kind of mailmerge, or the kind of functionality often found in templating engines such as Mustache.

A template engine such as Mustache will replace a template

```
{{SomeVariable}}
```

with the value of `SomeVariable`. Usually more complex logic is also available, but for this example, we limit ourselves to simple variables, which are kept in a `Key=Value` list.

This time we use a Delphi program. We drop 2 `TMemo`Ò components on it, one for the template (`MTemplate`), one for the result (`MResult`). We also drop a `TValueListEditor` on the form. Here the user can enter the values for the template variables. Last item is a button (`Generate`) to generate the final text based on the template and the user-provided variables and their values.

The `OnClick` event of the button is implemented with the `ReplaceEx` call:

```
procedure TMainForm.GenerateClick(Sender: TObject);

Var
  Regex : TRegexpr;

begin
  Regex:=TRegexpr.Create;
  try
    Regex.Expression:='\{\{[^{}]*\}\}';
    MResult.Lines.Text:=Regex.ReplaceEx(MTemplate.lines.Text,
                                        ReplaceCallback);
  finally
    Regex.Free;
  end;
end;
```

As you can see, the code is almost disappointingly simple. A regexp instance is created, the regular expression to find template names is entered, and the `ReplaceEx` is called with the `MTemplate` text as input. The result of the function is assigned to the text in `MResult`.

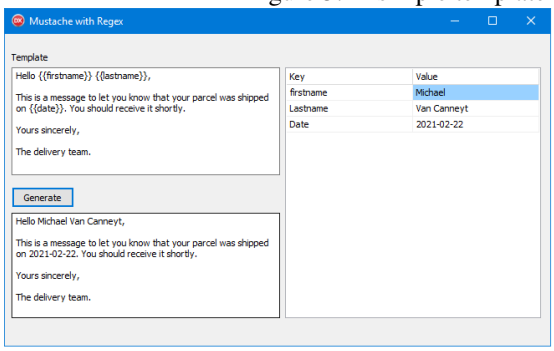
The real magic happens in the `ReplaceCallback` callback:

```
function TMainForm.ReplaceCallBack(Sender: TRegexpr): String;

Var
  aMatch,aName : String;

begin
  aMatch:=Sender.Match[0];
  aName:=Copy(aMatch,3,Length(aMatch)-4);
  Result:=vleKeys.Strings.Values[aName];
end;
```

Again, the function is simple. It could be implemented as a one-liner, but for clarity, it is split out in 3 lines: First the regular expression match is taken, then the curly braces are stripped off to get the variable name, and the last line uses the variable name to look up the replacement value in the value list editor.

That's it. It is so simple to create a simple templating engine. The result can be seen in figure 5 on page 10

Figure 5: A simple template engine in action



5 Conclusion

Being able to use regular expressions can make life a lot easier, both when coding in the IDE as in an actual program: The examples presented here hopefully demonstrated that using simple regular expression is really not difficult and that they can easily be implemented in a Delphi or Lazarus program.