kleiner kommunikation
max@kleiner.com

//Exklusiv für Entwickler 2013

1 Wer versucht der findet (Express yourself)

Für reguläre Ausdrücke gibt es jede Menge Einsatzgebiete, bei der alltäglichen Programmierarbeit sind sie quasi unverzichtbar. Man kann die Eigenschaften, z. B. das Vorkommen bestimmter (Arten von) Zeichen, Reihenfolge, Häufigkeit etc. eines Textes oder einer Datenmenge allgemein beschreiben und danach suchen oder ersetzen lassen. Besonders hilfreich sind RegEx als Regelwerk zum Prüfen einer Syntax oder eines Formats. Dieser Bericht zeigt Einsatz und Beispiele aus den Gebieten Web, DB und Objektstrukturen, denn wer sucht will gefunden werden.

1.1 Ein Regelwerk

Reguläre Ausdrücke (regular expressions) sind sehr leistungsfähig und ermöglichen das komplizierte und herausfordernde Zerlegen (Parsing) von Texten viel leichter. Durch einen Quasi-Standard sind auch die Muster hochgradig wieder verwendbar. In dem Sinne gibt es keine irregulären Ausdrücke.

Was sind RegEx? Sie dienen dazu, Muster als Suchmasken oder Filter zu definieren; Muster oder neudeutsch Patterns, die Klassen von Zeichenketten beschreiben und festlegen.

RegEx Methoden haben in den meisten Umgebungen das vordefinierte Objekt namens Regex. Sie können Regex schreiben gefolgt von einem Punkt um die Verfügbaren Methoden zu sehen.

```
Bsp.: RegEx.Options:= [preCaseLess, preMultiline];
```

Alle Methoden akzeptieren einen Muster-String (pattern string). Dies ist das Muster des regulären Ausdrucks. Beachten Sie, daß intern die kompilierten Muster zwischengespeichert werden. Es gibt also keinen Performance-Verlust bei mehrfacher Verwendung des gleichen Musters.

Betrachten wir mal einen ersten kleinen RegEx, der seinen Nutzen auch in Suchmasken oder Editoren hat (siehe Abb. 1). In der Regel kennt man Ausdrücke wie `'*.txt'`, z.B. beim Dateimanagement, somit ist das Prinzip der regulären Ausdrücke bereits bekannt. Einzelne Sonderzeichen (wie hier der asterisk[1] `'*'`) haben eine spezielle Bedeutung und stehen z.B. für beliebigen Text (der `'*'` steht als Platzhalter für einen beliebigen Dateinamen). Es geht nun um das Suchen oder Ersetzen von leeren Zeilen, sozusagen das „Hello World“ bei RegEx:

→`'^$'`

Bereits bei diesen ersten zwei Zeichen (auch Anchors genannt) kommt der Kontext ins Spiel, der mit einer Tabelle einhergeht:

^ , \A	Beginn des Strings
$, \Z	String-Ende
[]	Auswahlmöglichkeiten einzelner Zeichen / ODER
[^]	Negative Auswahlmöglichkeiten einzelner Zeichen / NICHT ODER
.	Beliebiges Zeichen
?	Kein- oder einmal, {0,1}
*	Keinmal bis beliebig oft {0,} (gierig, greedy; später mehr)
+	Ein- oder mehrmals {1,} (gierig;)
*?	Keinmal bis beliebig oft {0,}? (nicht gierig; später mehr)
+?	Ein- oder mehrmals {1,}? (nicht gierig;)

[1] Bedeutet: kleiner Stern, nicht zu verwechseln mit Asterix ;-)

{7}	siebenmal
{3,5}	Drei- bis fünfmal
{4,}	Viermal oder mehr (mindestens)

Tabelle 1

Das ^ steht für Zeilenumbruch oder Vergleich am Anfang einer Zeile aber auch als Negation. Bsp: '^b' bedeutet alle Zeichen außer 'b', also eine Negativauswahl.

Das $ wiederum meint Zeilenumbruch oder Vergleich am Ende einer Zeile. Falls die Multiline-Option aktiv ist, wird zusätzlich jeder Zeilenanfang und jedes Zeilenende gefunden.

Falls dieser Mehrzeilenmodus gewählt ist, findet ^ jeden Beginn einer neuen Zeile und $ jedes Zeilenende sowie die Position vor dem Ende der Zeichenfolge bzw. vor dem letzten \n (neue Zeile). Falls die Option fehlt, finden die beiden Ausdrücke nur den absoluten Beginn bzw. das Ende der Eingabezeichenfolge. Beispiel gefällig mit Optionen:

```
RegEx.Options:= [preCaseLess, preMultiline];
  regex:= '^No.*0$';
Firstmatch: northwest   NW  Charles Main    30000.00
NextMatch: Northeast    NE  AM Main Jr.     57800.10
```

Es wird also jede ganze Zeile (.*) gefunden, die mit ’no’ beginnen und mit ’0’ enden, zusätzlich wird gross und klein nicht unterschieden. Trainieren wir gleich weiter.

Ausgehend von unserem Anfangszeichen wollen wir uns als nächstes ansehen, wie wir alle Zeilen ausgeben lassen, die mit einem Großbuchstaben beginnen:

```
Regex:= '^[A-Z]';
Firstmatch: Western     WE   Sharon Gray    53000.89
NextMatch: Southern     SO   Suan Chin      54500.10
NextMatch: Northeast    NE   AM Main Jr.    57800.10
```

Neu hinzugekommen ist die Auswahlmöglichkeit als [] mit ODER Logik (siehe oben Tabelle). Aber aufgepasst, dies war eine Falle, denn solange die Option `[preCaseLess]` eingestellt ist, kommt jede Zeile mit einem simplen Buchstabenanfang, doch wir wollen ja explizit nur nach Grossbuchstaben suchen! Bisher haben wir noch trocken geübt, wie sieht das im Tool Grep oder maXbox aus:

Beide sind im RegEx-Dialekt "PCRE" ("Perl-compatible regular expressions") daheim, und sie haben richtig gelesen, Dialekte gibt es nicht nur in SQL. (PHP unterstützt übrigens ebenfalls RegEx, jedoch mit einer leicht anderen Syntax).

```
$ grep "^[A-Z]" demo1.txt
```

Das Programm, mit dessen Hilfe wir diesen RegEx formulieren werden, ist ’grep’. Der Name "grep" ist, wie so oft in der UNIX Welt, ein Akronym - in diesem Fall für "global regular expression print". Grep werkelt, wie die meisten RegEx fähigen Programme, zeilenorientiert. Wird das geforderte Muster in einer Zeile der angegebenen Datei gefunden, wo wird diese auf stdout alias writeln (meist also dem Monitor) ausgegeben, andernfalls ignoriert grep ihre Existenz ganz einfach. RegEx - und damit auch grep - sind im Übrigen "case-sensitive", sie unterscheiden Groß- und Kleinschreibung.

Trainieren lässt sich das obige Konstrukt auch in der Box, anbei ein Auszug aus Script 309_regex_powertester3.txt.

```
  rx:=TPerlRegEx.Create;
    try
      rx.RegEx:= '^[A-Z].*';
      rx.Subject:= fs; //fstr;
      rx.Options:= PR1.Options + [preMultiline, preCaseLess];
      if rx.Match then begin
        WriteLn('Firstmatch: '+rx.MatchedText);
```

```
      while PR1.MatchAgain do
        WriteLn('Nextmatch: '+rx.MatchedText); // Extract subsequent )
```

Es gibt auch die Möglichkeit einer Funktion (als Funktionsobjekt):

```
if ExecRegExpr('^[A-Z].*',fs)
      then writeln('regex found') else writeln('regex not found');
```

Grundsätzlich hat der Zeichensatz, vor allem bei Multiline, im jeweiligen Betriebssystem einen Einfluss, denn jeder Zeichensatz ist von 0 - 127 ASCII gültig und genau dort spielt sich ja auch die CRLF-Tragödie ab (CarriageReturn#13, LineFeed#10). Schließlich geht es ja um korrekte Datenverarbeitung und nicht um wilde Datenkorruption.

```
$zeile =~ /^\n$/ ;
```

oder dasselbe für Win (manchmal)

```
$zeile =~ /^\r\n$/ ;
```

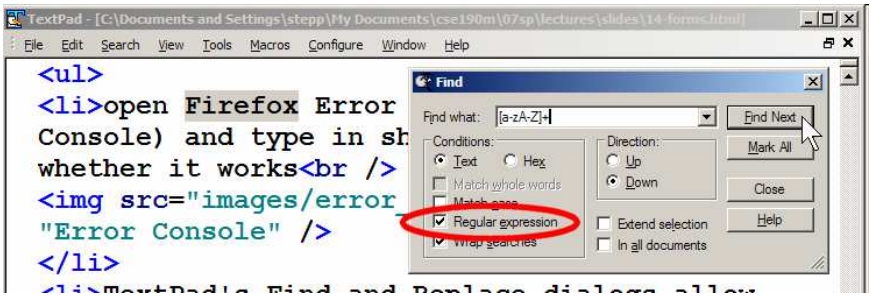

Abb1: RegEx im Editor

Decodieren wir nun weiter die Abb1 und beenden Anfang^ und Ende$[2]. Jedes Zeichen aus dem Zeichenbereich 'a' bis 'z' oder 'A' bis 'Z' wird gefunden, d.h. das + steht für ein oder mehr Zeichen und bei zwei Mengen die einander folgen gilt die Oder Regel.

Für jede Methode gibt es ja zwei Varianten. Der Unterschied zwischen den Varianten ist, daß die zweite einen "Option" Parameter enthält, der auf das Verhalten der RegEx Engine wirkt. In der Regel hat jeder RegEx mindestens zwei Optionen, CASE_INSENSITIVE und MULTILINE. Das erstere macht den Mustervergleich unempfindlich gegen Groß- und Kleinschreibung und MULTILINE setzt die String Anker ^ und $ auf Beginn und Ende jeder Zeile statt Beginn und Ende des ganzen Strings, je nachdem wie gierig die Engine ist. Nun was heißt gierig (greedy)?

Grep, Perl, PCRE ist in der Standard-Einstellung sog. gierig, d.h. ein * oder + versucht immer so viele Zeichen wie möglich zu sammeln, so dass die RegEx noch maximal möglich ist. Hängt man an das + oder * noch ein Fragezeichen an (+? bzw. *?), hat man das umgekehrte Ergebnis: Perl nimmt nun nur so viele Zeichen wie absolut notwendig, um die RegEx noch gültig zu machen.

Nun das müssen wir mal klar als Beispiel sehen, da es hier immer wieder Differenzen gibt:

```
RegEx.Options:= [prepreCaseLess, preMultiline];
  regex:= '^No.*';
Firstmatch: northwest  NW Charles Main  300000.00
NextMatch: Northeast  NE AM Main Jr.  57800.10
NextMatch: north NO  Ann Stephens  455000.52
```

Dies also der Default als Gierig, nun explizit ungierig:

```
RegEx.Options:= RegEx.Options +[ preUnGreedy];
  regex:= '^No.*';
```

[2] RegEx kann auch philosophisch sein

```
Firstmatch: no
NextMatch: No
NextMatch: no
```

Er gibt nur das Minimum als ‚no' raus; eigentlich einfach, wenn man es mal gesehen hat. Doch nun der Hammerausdruck, bei eingestelltem Endanker $ also `'^No.*0$'` ist dann wieder die ganze Zeile gefragt, auch wenn nicht gierig eingestellt ist (eigentlich auch logisch)!

```
Firstmatch: northwest   NW  Charles Main    30000.00
Nextmatch: Northeast    NE  AM Main Jr.     57800.10
```

Das Wissen um greedy kann auch im Zusammenhand mit der Zwischenablage (Klammern(), auch Group Operator) notwendig sein, um zu definieren, wie viel Text man in die entsprechende Zwischenablage kopieren soll.

Als nächstes kommt die Geschichte vom Ersetzen und da kann ich auch gleich die Back Reference einführen. Nachdem wir das Suchen nun in seiner Tiefe behandelt haben, bereitet uns das Ersetzen keine grossen Probleme mehr, hoffe ich doch.

```
procedure PerlRegexReplace;
begin
  with TPerlRegex.create do try
    Options:= Options + [preUnGreedy];
    Subject:= 'I like to sing out at Foo bar';
    RegEx:= '([A-Za-z]+) bar';
    Replacement:= '\1 is the name of the bar I like';
    if Match then ShowMessageBig(ComputeReplacement);
  finally
    Free;
  end;
end;
```

Mit diesem Konstrukt lässt sich ungierig (minimal) in einem Gruppen Operator (Klammer) folgende Ersetzung bilden:

```
Foo is the name of the bar I like'
```

Die RegEx nimmt das Wort vor ‚bar' nach seiner Gültigkeit lässt sich in `Replacement` einfügen. Das Gebilde \1 nennt man eine Back Reference (Rückverweis), der einen Buffer oder Zwischenspeicher ermöglicht. Ein Rückverweis sucht die durch die Nummer in \1 nach Position oder die durch den Namen in \ definierte aufgezeichnete Teilzeichenfolge die eben in die runde Klammer gesetzt ist.

Wirklich nützlich und kreativ ist folgendes kleine Web-Beispiel (Ich nutze der Einfachheit wegen das Funktionsobjekt `ReplaceRegExpr` direkt:

```
Writeln(ReplaceRegExpr('<.*?>','Dies ist ein <b>Text</b> mit <i>HTML</i>-
Kennzeichen', '',true));
```

```
→ Dies ist ein Text mit HTML-Kennzeichen
```

Dieses Beispiel löscht alle HTML-Kennzeichnungen innerhalb eines Textes. Also von einem "<" eine beliebige Anzahl Zeichen bis zum nächsten ">" durch nichts zu ersetzen. Deshalb ist der dritte Parameter ein Leerstring. Die RegEx selbst ist als erstes Argument zu finden: `<.*?>`. Die spitzen Klammern gehören aber zum Text als HTML und sind kein RegEx Operator oder dergleichen. Wichtig ist hier das *? in der RegEx, also die Definition von nicht gierig, da ansonsten der Text vom ersten "<" bis und mit dem letzten ">" ersetzt werden würde (Was ein Fragezeichen alles bewirkt!):

```
Regex:= <.*> //falsch
→ Dies ist ein –Kennzeichen
Regex:= <.> //auch falsch
→ Dies ist ein Text</b> mit HTML</i>-Kennzeichen
```

Für weitere Tests lässt sich als Inputstring auch mal eine Datei nutzen: Mit dieser Funktion `ReplaceRegExpr()` sind folgende Parameter definiert

```
function ReplaceRegExpr (const ARegExpr, AInputStr, AReplaceStr: string;
AUseSubstitution: boolean = False): string;
```

So, diese beiden Suchen und Ersetzen Beispiele geben genug Hirnschmalz zum Braten und Trainieren, lassen wir am Schluss noch die Musik erklingen und analysieren den nächsten RegEx:

```
with TRegExpr.Create do try
  gstr:= 'Deep Purple';
  modifierS:= false; //non greedy
  Expression:= '#EXTINF:\d{3},'+gstr+' - ([^\n].*)';
     if Exec(fstr) then
      Repeat
         writeln(Format ('Songs of ' +gstr+': %s', [Match[1]]));
        if AnsiCompareText(Match[1], 'Woman') > 0 then begin
          closeMP3;
          PlayMP3('..\EKON16\06_Woman_From_Tokyo.mp3');
        end;
      Until Not ExecNext;
  finally Free;
end;
```

Hier setze ich eine Bibliothek aus dem Regexstudio ein (http://regexpstudio.com/tregexpr/), die sich auch am PCRE-Dialekt orientiert, wobei die Objektnamen Unterschiede zeigen.

Die Regex `'#EXTINF:\d{3},'+gstr+' - ([^\n].*)'` sucht aus einer zuvor mit `Exec(fstr)` eingelesenen m3u- Liste alle Songs einer bestimmten Band und spielt die dann ab. Die runde Klammer dient wieder als Zwischenspeicher, der unter `match[n]` gespeichert ist. Wobei dann der `match[1]` als Resultatliste einen definierten Song vergleicht und abspielt.

Das Erstaunlich ist die Iteration, die denn Programmfluss erst verlässt, wenn kein definierter Song mehr gefunden wurde, zudem lassen sich bei mehr als ein Resultat, die Lieder asynchron abspielen, was aber offensichtlich hörbar keinen Sinn macht, oder hat man das 3D Hören noch nicht erfunden?

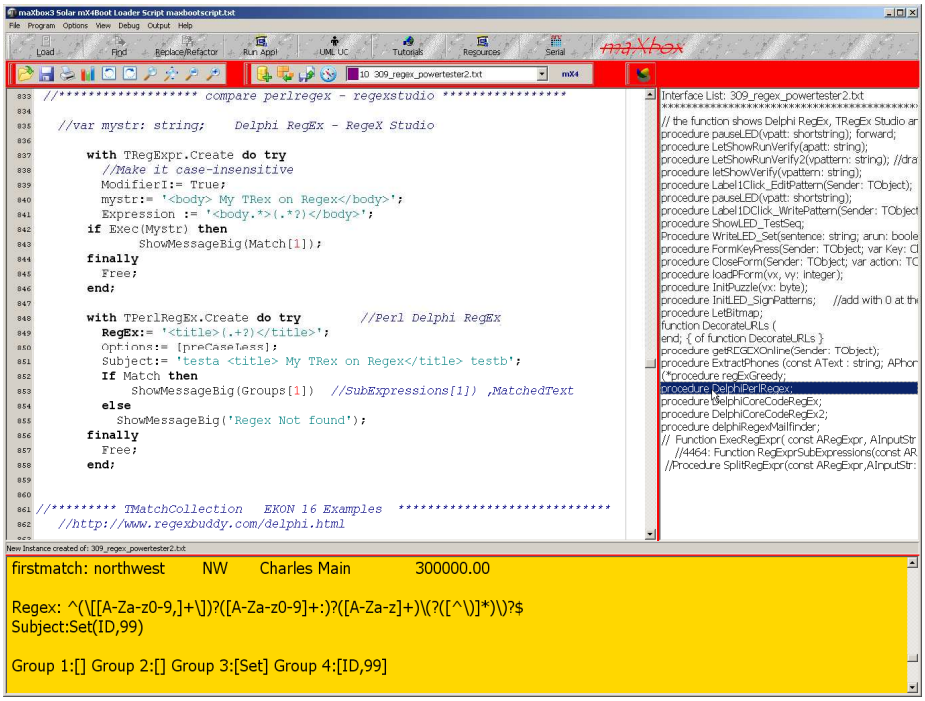

Abb2: RegEx in der Box

Ein Programm, mit dessen Hilfe wir auch RegEx formulieren und testen können, ist der RegexBuddy. Dieser Editor zum Erstellen und Testen regulärer Ausdrücke ist sehr beliebt, hat aber seinen kleinen Preis (ca. 30 Euro). Man kann auch normale englische Bausteine anstelle der komplizierten Syntax zum Erstellen eines RegEx gebrauchen. Besonders hilfreich ist der Codegenerator vom definierten Ausdruck zum Code hin. Das mit dem Tool erstellte Konstrukt steht dann in den verschiedensten Sprachen als Code Snippet bereit zum Kopieren (siehe Abb. 3).

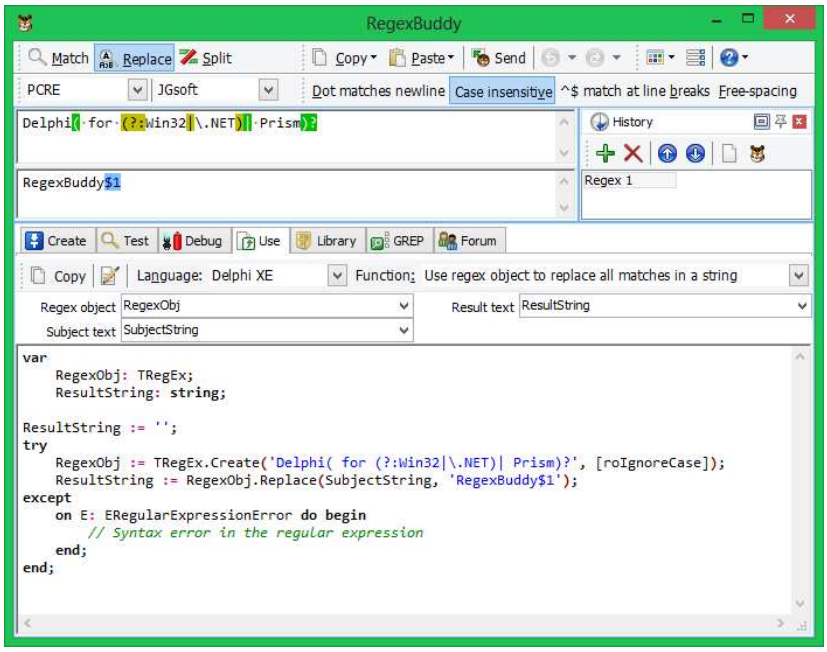

Abb3: RegexBuddy im Codegenerator

1.2 Wer sucht der bindet

Wer versucht der findet, kann dieses Kapitel auch lauten. Besonders aufmerksame Leser werden sich fragen, wie man denn Regex-Zeichen selbst findet. Wie zum Beispiel ist es möglich, die Zeichen "^" oder "-" innerhalb einer eckigen Klammer [] als solche zu matchen?

Ganz einfach: will man "^" innerhalb eines solchen Ausdrucks finden, stellt man sie einfach NICHT an den Anfang der Liste. Möchte man "-" matchen, stellt man es schlicht an den Anfang, also direkt nach der eckigen Klammer, oder das Ende. Ein analoger Fall ist das Aufspüren von "[" und "]", das wir auch an den Anfang stellen.

Ein Muster, das alle unsere eben kennen gelernten "Problemzeichen" erledigt, ist das folgende: []^[-].

```
'Extra2 ^[A-Z]****[0-9]..$5.00'+CR+LF;
→ Firstmatch:^, Nextmatch: [-][-]
```

Also anderen RegEx-Zeichen, bis auf den escape Char selbst, verlieren innerhalb von [und] ihre Sonderbedeutung, und bereiten uns keine weitere Schwierigkeit mehr.

Ähnlich gelagert ist der Asterisk *, in diesem Fall kommt das so genannte "Escapen", das aus vielen Anwendungen der Programmierung vertraut ist, ins Spiel: einem gewissen Zeichen, meist \ (als "Backslash"), wird eine besondere Bedeutung verliehen: Wird "\" vor ein Zeichen aus dem RegEx-Zeichensatz gestellt (Zeichensatz: `$ ^ { [ ( | ) ] } * + ? \`), verliert dieses Zeichen seine Sonderbedeutung - und wird zu einem ordinären Literal, sozusagen einem Zeichen mit "wörtlicher Bedeutung". Das gilt übrigens auch für "\" selbst; will man einen oder mehr Backslash finden, muss man also \\ notieren.

In vielen Situationen, wo es um Wiederholungen geht, helfen uns Quantoren. Quantoren sind Zeichen aus dem RegEx-Zeichsatz (siehe oben), die vorangegangene Muster mit dem Kriterium "Anzahl von Wiederholungen" erweitern. Manchmal will man z.B. bei einer IP-Adresse wissen, dass eine Zahl zwischen 12 oder 24-mal vorkommen muss. Es gibt drei vordefinierte (*/+/?) sowie selbstdefinierte

Quantoren, letztere werden eben durch Zahlen in geschweiften Klammern ausgedrückt. Hier kommt der curly bracket {} ins Spiel.

Mittels "{" ist es möglich, einen festgelegten Bereich gültiger Wiederholungen zu setzen, und zwar mit {n,m} - wobei n die Unter-, und m die Obergrenze dieser Anzahl definieren. Als gültiges Jahr könnte man akzeptieren:

```
\b(\d{2,4}) //Jahrzahl
'\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3}'//IP-Adresse
```

Mit solchen Quantoren hat man übrigens einen grossen Teil der menschlichen DNA beim Human Genom Project entziffert, Buchtipp: „Perl for Exploring DNA“.

Hier kommt ein weiter Trumpf. Wenn ich bspw. nur nach Zahlen suche, lässt sich mit \d (auch schreibbar als [0-9]) der RegEx abkürzen, oder ich will keine Zahlen: \D (auch als [^0-9]).

Alle alphanumerischen Zeichen inkl. _ und Zahlen aber ohne Umlaute, auch schreibbar als [a-zA-Z0-9_] lässt sich einfach mit \w abkürzen.

Abschließend noch ein Funktionsmuster, das extensiv vom Zwischenspeicher Gebrauch macht. Der Regex sucht alle Telefonnummern mit dem Citycode 812. Die ersten beiden Match sind mögliche Ländercodes/Vorwahl, dann folgt mit `Match[3]` der Citycode und abschließend die Telefonnummer.

```
procedure ExtractPhones(const AText: string; APhones: TStrings);
  begin
    with TRegExpr.Create do try
      Expression := '(\+\d*)?(\((\d+)\)*)?(\d+(-\d*)*)';
      if Exec (AText) then
      Repeat
        if Match[3] = '812'
        then APhones.Add(Match [4]);
      Until not ExecNext;
    finally Free;
  end;
end;
```

Für DB Entwickler interessant sind die Diskussionen ob mit einer RegEx Subroutine ein where-Statement bei einer SQL-Abfrage nützlich ist.

```
Select * from customer where company like "%SCUBA%"''';
Select * from customer where company like ExecRegExpr();
```

Wer erst einmal etwas tiefer ins Thema Regex eingestiegen ist, wird sehr schnell erkennen, wesentlich schwerer als Rechnen mit Brüchen und Prozenten sind Reguläre Ausdrücke auch nicht zu erlernen.

Max Kleiner

Literatur:

Mastering Regular Expressions - Powerful Techniques for Perl and Other Tools

Links:

http://www.softwareschule.ch/examples/309_regex_powertester3.txt

http://sourceforge.net/projects/maxbox/

http://www.myregextester.com/index.php

http://erik.eae.net/playground/regexp/regexp.html