TRegExpr Documentation

Release 0.952

Andrey Sorokin

27.05.2019

Inhaltsverzeichnis

This is very old and outdated translation. *If you can read English or Russian please use up-to-date* English version *or* Russian version.

If you want to help to update the translation please contact me. New translation is based on GetText and can be edited with transifex.com. It is already machine-translated and need only proof-reading and may be some copy-pasting from here.

TRegExpr ist eine Sammlung von einfach zu benutzenden Routinen, um mächtige, vorlagenbasierte Zeichenkettenvergleiche durchzuführen, beispielsweise zur Prüfung von strukturierten Dateneingaben in Datenbanken wie beispielsweise Telefonnummern mit Vorwahlen, Sozialversicherungsnummern, Web-Applikationen, komplexere Suchen & Ersetzen-Vorgänge, Werkzeuge zur Durchforstung von Dateibeständen nach regelbasierenden Ausdrücken und so weiter.

Du kannst mit TRegExpr leicht und schnell die korrekte Syntax einer E-Mail-Adresse prüfen, Telefonnummern in einem Text erkennen, URLs aus Qelltexten von Web-Seiten extrahieren, unterschiedliche Schreibweisen eines Ausdruckes finden und durch eine einzige ersetzen. Es bleibt Deiner Fantasie überlassen, wozu Du TRegExpr noch benutzen kannst. Die Suchvorlagen (im folgenden Templates genannt), können zur Laufzeit geändert werden, ohne dass eine Neuübersetzung des Programmes notwendig würde!

Diese Bibliothek, die ich hiermit in die Freeware lege, ist eine Delphi-Portierung der Routinen, die Henry Spencer als V8-Routinen herausbrachte, um damit eine Untermenge der Regulären Ausdrücke von Perl handhaben zu können.

Demgegenüber ist TRegExpr vollständig in einfachem Object-Pascal geschrieben und wird mit dem ganzen Quelltext kostenlos zur Verfügung gestellt.

Der originale C-Quelltext wurde verbessert, in eine Klasse TRegExpr gekapselt und in einer einzigen Datei gespeichert: RegExpr.pas.

Du brauchst also keine DLL mehr für Reguläre Ausdrücke!

Um die Bibliothek zu installieren, kopiere einfach RegExpr.pas in ein Verzeichnis Deiner Wahl und/oder füge den Pfad zu diesem Verzeichnis in Delphis Projekt-Manager hinzu.

Das ist schon alles!

Danach benutze einfach das TregExpr-Objekt oder die globalen Routinen in Deinem Projekt (beachte die Beispiele).

Schaue Dir mal die einfachen Beispiele an und studiere die Syntax der Regulären Ausdrücke (Du kannst natürlich auch das Demo-Projekt für Studienzwecke heranziehen und damit auch Deine eigenen Regulären Ausdrücken ausarbeiten oder debuggen).

Du kannst sogar Unicode (d.h. Delphis WideString) benutzen – weiteres unter „Wie wird Unicode benutzt?“.

Wirf auch einen Blick auf die Was gibt’s Neues web-Sektion für die neuesten Änderungen.

Und natürlich sind Kommentare, Ideen, Vorschläge und sogar Bug Reports *willkommen*.

- Guido Muehlwitz - er fand und behob einen ärgerlichen Fehler bei der Bearbeitung von grossen Strings
- Stephan Klimek - er testete in CPPB und schlug einige Features vorund implementierte sie auch gleich
- Steve Mudford - er implementierte den Offset-Parameter
- Martin Baur (www.mindpower.com) - Deutsche Hilfe,nützliche Vorschläge
- Yury Finkel - er implementierte die UniCode-Unterstützung, fand und behob einige Fehler
- Ralf Junker - er implementierte einige Features, zahlreiche Optimierungsvorschläge
- Simeon Lilov - Bulgarische Hilfe
- Filip Jirsák und Matthew Winter (wintermi@yahoo.com) – Hilfe bei der Implementation des „genügsamen“ Moduls

- Kit Eason - Viele Beispiele un die Einführung im Hilfe-Abschnitt
- Juergen Schroth - bug hunting and usefull suggestions
- Martin Ledoux - French help
- Diego Calp (mail@diegocalp.com), Argentinien – Spanische Hilfe

Und viele andere – für die grosse Arbeit des Fehlerfindens!

KAPITEL 1

Documentation

This is very old and outdated translation. *If you can read English or Russian please use up-to-date* English version *or* Russian version.

If you want to help to update the translation please contact me. New translation is based on GetText and can be edited with transifex.com. It is already machine-translated and need only proof-reading and may be some copy-pasting from here.

1.1 Einführung

Reguläre Ausdrücke werden weitum verwendet, um Textmuster zu beschreiben, nach welchen dann gesucht wird. Spezielle Metazeichen erlauben das Definieren von Bedingungen, beispielsweise soll ein bestimmter gesuchter String am Anfang oder am Ende einer Zeile vorkommen, oder ein bestimmtes Zeichen soll n mal Vorkommen.

Reguläre Ausdrücke sehen üblicherweise für Anfänger ziemlich kryptisch aus, sind aber im Grunde genommen sehr einfache (nun, üblicherweise einfache ;)), handliche und enorm mächtige Werkzeuge.

Ich empfehle Dir wärmstens, dass Du mit dem Demo-Projekt in Windows REStudio – es wird Dir enorm dabei helfen, die hauptsächlichen Konzepte zu erfassen. Darüberhinaus findest Du viele vorgegebene und kommentierte Beispiele in TestRExp.

Also, starten wir in die Lernkurve!

1.2 Einfache Treffer

Jedes einzelne Zeichen findet sich selbst, ausser es sei ein Metazeichen mit einer speziellen Bedeutung (siehe weiter unten).

Eine Sequenz von Zeichen findet genau dieses Sequenz im zu durchsuchenden String (Zielstring). Also findet das Muster (= reguläre Ausdruck) `bluh` genau die Sequenz `bluh` irgendwo im Zielstring. Ganz einfach, nicht wahr?

Damit Du Zeichen, die üblicherweise als Metazeichen oder Escape-Sequenzen dienen, als ganz normale Zeichen ohne jede Bedeutung finden kannst, stelle so einem Zeichen einen \ voran. Diese Techninik nennt man Escaping. Ein Beispiel: das Metazeichen ^ findet den Anfang des Zielstrings, aber \^ findet das Zeichen ^ (Circumflex), \\ findet also \ etc.

1.2.1 Beispiele:

```
foobar       findet den String 'foobar'
\^FooBarPtr findet den String '^FooBarPtr'
```

1.3 Escape-Sequenzen

Zeichen könenn auch angeben werden mittels einer Escape-Sequenz, in der Syntax ähnlich derer, die in C oder Perl benutzt wird: \n findet eine neue Zeile, \t einen Tabulator etc. Etwas allgemeiner: \xnn, wobei nn ein String aus hexadezimalen Ziffern ist, findet das Zeichen, dessen ASCII Code gleich nn ist. Falls Du Unicode-Zeichen (16 Bit breit kodierte Zeichen) angeben möchtest, dann benutze \x{nnnn}, wobei nnnn – eine oder mehrere hexadezimale Ziffern sind.

```
 \xnn      Zeichen mit dem Hex-Code nn (ASCII-Text)
 \x{nnnn} Zeichen mit dem Hex-Code nnnn (ein Byte für ASCII-Text und zwei Bytes für
[Unicode](tregexpr.html#unicode)-Zeichen
 \t        ein Tabulator (HT/TAB), gleichbedeutend wie \x09
 \n        Zeilenvorschub (NL), gleichbedeutend wie \x0a
 \r        Wagenrücklauf (CR), gleichbedeutend wie \x0d
 \f        Seitenvorschub (FF), gleichbedeutend wie \x0c
 \a        Alarm (bell) (BEL), gleichbedeutend wie \x07
 \e        Escape (ESC), gleichbedeutend wie \x1b
```

1.3.1 Beispiele

```
foo\x20bar   findet 'foo bar' (beachte den Leerschlag in der Mitte)
\tfoobar     findet 'foobar', dem unmittelbar ein Tabulator vorangeht
```

1.4 Zeichenklassen

Du kannst sogenannte Zeichenklassen definieren, indem Du eine Liste von Zeichen, eingeschlossen in eckige Klammern [], angibst. So eine Zeichenklasse findet genau eines der aufgelisteten Zeichen Zeichen im Zielstring.

Falls das erste aufgelistete Zeichen, das direkt nach dem [, ein ^ ist, findet die Zeichenklasse jedes Zeichen ausser denjenigen in der Liste.

1.4.1 #### Beispiele:

```
foob\[aeiou\]r   findet die Strings 'foobar', 'foober' etc. aber nicht 'foobbr',
↪'foobcr' etc.
foob\[^aeiou\]r findet die Strings 'foobbr', 'foobcr' etc. aber nicht 'foobar',
↪'foober' etc.
```

Innerhalb der Liste kann das Zeichen – benutzt werden, um einen Bereich oder eine Menge von Zeichen zu definieren. So definiert `a-z` alle Zeichen zwischen `a` and `z` inklusive.

Falls das Zeichen – selbst ein Mitglied der Zeichenklasse sein soll, dann setze es als erstes oder letztes Zeichen in die Liste oder schütze es mit einem vorangestellten `\` (escaping). Wenn das Zeichen `]` ebenfalls Mitglied der Zeichenklasse sein soll, dann setze es als erstes Zeichen in die Liste oder escape es.

1.4.2 Beispiele:

```
[-az]     findet 'a', 'z' und '-'
[az-]     findet 'a', 'z' und '-'
[a\-z]     findet 'a', 'z' und '-'
[a-z]     findet alle 26 Kleinbuchstaben von 'a' bis 'z'
[\n-\x0D] findet eines der Zeichen  #10, #11, #12 oder #13.
[\d-t]     findet irgendeine Ziffer, '-' oder 't'.
[]-a]     findet irgendein Zeichen von '\]'..'a'.
```

1.5 Metazeichen

Metazeichen sind Zeichen mit speziellen Bedeutungen. Sie sind die Essenz der regulären Ausdrücke. Es gibt verschiedene Arten von Metazeichen wie unten beschrieben.

```
^   Beginn einer Zeile
$   Ende einer Zeile
\A start of text
\Z end of text
.   irgendein beliebiges Zeichen
```

1.5.1 Beispiele:

```
^foobar     findet den String 'foobar' nur, wenn es am Zeilenanfang vorkommt
foobar$     findet den String 'foobar' nur, wenn es am Zeilenende vorkommt
^foobar$   findet den String 'foobar' nur, wenn er der einzige String in der Zeile ist
foob.r     findet Strings wie 'foobar', 'foobbr', 'foob1r' etc.
```

Standardmässig garantiert das Metazeichen `^` nur, dass das Suchmuster sich am Anfang des Zielstrings befinden muss, oder am Ende des Zielstrings mit dem Metazeichen `$`. Kommen im Zielstring Zeilenseparatoren vor, so werden diese von `^` oder `$` nicht gefunden.

Du kannst allerdings den Zielstring als mehrzeiligen Puffer behandeln, so dass `^` die Stelle unmittelbar nach, und `$` die Stelle unmittelbar vor irgendeinem Zeilenseparator findet. Du kannst diese Art der Suche einstellen mit dem Modifikator /m.

The `\A` and `\Z` are just like `^` and `$`, except that they won't match multiple times when the modifier /m is used, while `^` and `$` will match at every internal line separator.

Das `.` Metazeichen findet standardmässig irgendein beliebiges Zeichen, also auch Zeilenseparatoren. Wenn Du den Modifikator /s

ausschaltest, dann findet `.` keine Zeilenseparatoren mehr.

TRegExpr geht mit Zeilenseparatoren so um, wie es auf www.unicode.org empfohlen ist:

^ ist am Anfang des Eingabestrings, und, falls der Modifikator /m gesetzt ist, auch unmitelbar folgend einem Vorkommen von \x0D\x0A oder \x0A or \x0D (falls Du die Unicode-Version von TregExpr benutzst, dann auch nach \x2028 oder \x2029 oder \x0B oder \x0C oder \x85). Beachte, dass es keine leere Zeile gibt in den Sequence \x0D\x0A. Diese beiden Zeichen werden atomar behandelt.

$ ist am Anfang des Eingabestrings, und, falls der Modifikator /m gesetzt ist, auch unmitelbar vor einem Vorkommen von \x0D\x0A oder \x0A or \x0D (falls Du die Unicode-Version von TregExpr benutzst, dann auch vor \x2028 oder \x2029 oder \x0B oder \x0C oder \x85). Beachte, dass es keine leere Zeile gibt in den Sequence \x0D\x0A. Diese beiden Zeichen werden atomar behandelt.

. findet ein beliebiges Zeichen. Wenn Du aber den Modifikator /s ausstellst, dann findet . keine Zeilensearaptoren \x0D\x0A und \x0A und \x0D mehr (falls Du die Unicode-Version von TregExpr benutzst, dann auch \x2028 und \x2029 und \x0B und \x0C and \x85).

Beachte, dass ^.*$ (was auch eine leere Zeile findet können sollte) dennoch nicht den leeren String innerhalb der Sequence \x0D\x0A findet, aber es findet den Leerstring innerhalb der Sequenz \x0A\x0D.

Die Behandlung des Zielstrings als mehrzeiliger String kann leicht Deinen Bedürfnissen angepasst werden dank der TregExpr-Eigenschaften LineSeparators und LinePairedSeparator. Du kannst nur den UNIX-Stil Zeilenseparator \n benutzen oder nur DOS-Stil Separatoren \r\n oder beide gelichzeitig (wie schon oben beschrieben und wie es als Standard gesetzt ist). Du kannst auch Deine eigenen Zeilenseparatoren definieren!

Metazeichen – vordefinierte Klassen

```
\w  ein alphanumerisches Zeichen inklusive "_"
\W  kein alphanumerisches Zeichen, auch kein "_"
\d  ein numerisches Zeichen
\D  kein numerisches Zeichen
\s  irgendein wörtertrennendes Zeichen (entspricht [\t\n\r\f])
\S  kein wörtertrennendes Zeichen
```

Du kannst \w, \d und \s innerhalb Deiner selbstdefinierten Zeichenklassen benutzen.

1.5.2 Beispiele:

```
foob\dr     findet Strings wie 'foob1r', ''foob6r' etc., aber not 'foobar', 'foobbr'␣
↪etc.
foob\[\w\s]r findet Strings wie 'foobar', 'foob r', 'foobbr' etc., aber nicht 'foob1r
↪', 'foob=r' etc.
```

TRegExpr benutzt die Eigenschaften SpaceChars und WordChars, um die Zeichenklassen \w, \W, \s, \S zu definieren. Somit kannst Du sie auch leicht umdefinieren.

Metazeichen – Wortgrenzen

```
\b     findet eine Wortgrenze
\B    findet alles ausser einer Wortgrenze
```

Eine Wortgrenze (\b) is der Ort zwischen zwei Zeichen, welcher ein \w auf der einen und ein \W auf der anderen Seite hat bzw. umgekehrt. \b bezeichnet alle Zeichen des \w bis vor das erste Zeichen des \W bzw. umgekehrt.

Metazeichen - Iteratoren

Jeder Teil eines regulären Ausdruckes kann gefolgt werden von einer anderen Art von Metazeichen – den Iteratoren. Dank dieser Metazeichen kannst Du die Häufigkeit des Auftretens des Suchmusters im Zielstring definieren. Dies gilt jeweils für das vor diesem Metazeichen stehenden Zeichen, das Metazeichen oder den Teilausdruck.

```
*     kein- oder mehrmaliges Vorkommen ("gierig"), gleichbedeutend wie {0,}
+   ein- oder mehrmaliges Vorkommen ("gierig"), gleichbedeutend wie {1,}
?   kein- oder einmaliges Vorkommen  ("gierig"), gleichbedeutend wie {0,1}
{n}   genau n-maliges Vorkommen ("gierig")
{n,}   mindestens n-maliges Vorkommen ("gierig")
{n,m} mindestens n-, aber höchstens m-maliges Vorkommen ("gierig")
*?     kein- oder mehrmaliges Vorkommen ("genügsam"), gleichbedeutend wie {0,}?
+?     ein oder mehrmaliges Vorkommen ("genügsam"), gleichbedeutend wie {1,}?
??     kein- oder einmaliges Vorkommen ("genügsam"), gleichbedeutend wie {0,1}?
{n}?   genau n-maliges Vorkommen ("genügsam")
{n,}? Mindestens n-maliges Vorkommen ("genügsam")
{n,m}? mindestens n-, aber höchstens m-maliges Vorkommen ("genügsam")
```

Also, die Ziffern in den geshcweiften Klammern in der Form `{n,m}` geben an, wieviele Male das Suchmuster im Zielstring gefunden muss, um einen Treffer zu ergeben. Die Angabe `{n}` ist gleichbedeutend wie `{n,n}` und findet genau `n` Vorkommen. Die Form `{n,}` findet n oder mehre Vorkommen. Es gibt keine Limiten für die Zahlen `n` und `m`. Aber je grösser sie sind, desto mehr Speicher und Zeit wird benötigt, um den regulären Ausdruck auszuwerten.

Falls eine geschweifte Klammer in einem anderen als dem eben vorgestellten Kontext vorkommt, wird es wie ein normales Zeichen behandelt.

1.5.3 Beispiele:

```
foob.*r     findet Strings wie 'foobar',  'foobalkjdflkj9r' und 'foobr'
foob.+r     findet Strings wie 'foobar', 'foobalkjdflkj9r', aber nicht 'foobr'
foob.?r     findet Strings wie 'foobar', 'foobbr' und 'foobr', aber nicht 'foobalkj9r'
fooba{2}r   findet den String 'foobaar'
fooba{2,}r findet Strings wie 'foobaar', 'foobaaar', 'foobaaaar' etc.
fooba{2,3}r findet Strings wie 'foobaar', or 'foobaaar', aber nicht 'foobaaaar'
```

Eine kleine Erklärung zum Thema „gierig“ oder „genügsam“. „Gierig“ nimmt soviel wie möglich, wohingegen „genügsam“ bereits mit dem ersten Erfüllen des Suchmusters zufrieden ist. Beispiel: `b+` und `b*` angewandut auf den Zielstring `abbbbc` findet `bbbb`, `b+?` findet `b`, `b*?` findet den leeren String, `b{2,3}?` findet `bb`, `b{2,3}` findet `bbb`.

Du kannst alle Iteratoren auf den genugsamen Modus umschalten mit dem Modifier /g.

Metazeichen - Alternativen

Du kannst eine Serie von Alternativen für eine Suchmuster angeben, indem Du diese mit einem `|` trennst. Auf diese Art findet das Suchmuster `fee|fie|foe` eines von `fee`, `fie`, oder `foe` im Zielstring – dies würde auch mit `f(e|i|o)e` ereicht.

Die erste Alternative beinhaltet alles vom letzten Muster-Limiter (`(`, `[` oder natürlich der Anfang des Suchmusters) bis zum ersten `|`. Die letzte Alternative beinhaltet alles vom letzten `|` bis zum nächsten Muster-Limiter. Aus diesem Grunde ist es allgemein eine gute Gewohnheit, die Alternativen in Klammern anzugeben, um möglichen Missverständnissen darüber vorzubeugen, wo die Alternativen beginnen und enden.

Alternativen werden von links nach rechts gepürft, so dass der Treffer im Zielstring zusammengesetzt ist aus den jeweils zuerst passenden Alternativen. Das bedeutet, dass Alternativen nicht notwendigerweise „gierig“ sind. Ein Bei-

spiel: Wenn man mit `(foo|foot)` im Zielstring „barefoot“ sucht, so passt bereits die erste Variante. Diese Tatsache mag nicht besonders wichtig erscheinen, aber es ist natürlich wichtig, wenn der gefundene Text weiterverwendet wird. Im Beispiel zuvor würde der Benutzer nicht `foot` erhalten, wie er eventuell beabsichtigt hatte, sondern nur `foo`.

Erinnere Dich auch daran, dass das `|` innerhalb von eckigen Klammern wie ein normales Zeichen behandelt wird, so dass `[fee|fie|foe]` dasselbe bedeutet wie `[feio|]`.

1.5.4 Beispiele:

```
foo(bar|foo) findet die Strings 'foobar' oder 'foofoo'.
```

Metazeichen - Teilausdrücke

Das KLammernkonstrukt `(...)` wird auch dazu benutzt, reguläre Teilausdrücke zu definieren (nach dem Parsen findest Du Positionen, Längen und effektive Inhalte der regulären Teilausdrücke in den TRegExpr-Eigenschaften MatchPos, MatchLen und Match und kannst sie ersetzen mit den Template-Strings in TRegExpr.Substitute).

Teilausdrücke werden nummeriert von links nach recht, jeweils in der Reihenfolge ihrer öffnenden Klammer. Der erste Teilausdruck hat die Nummer `1`, der gesamte reguläre Ausdruck hat die Nummer `0` (der gesamte Ausdruck kann ersetzt werden in TRegExpr.Substitute als `$0` oder `$&`).

1.5.5 Beispiele:

```
(foobar){8,10} findet Strings, die 8, 9 oder 10 Vorkommen von 'foobar' beinhalten
foob(\[0-9\]|a+)r findet 'foob0r', 'foob1r' , 'foobar', 'foobaar', 'foobaar' etc.
```

Metazeichen - Rückwärtsreferenzen

Die Metacharacters `\1` bis `\9` werden in Suchmustern interpretiert als Rückwärtsreferenzen. `\<n>` findet einen zuvor bereits gefundenen Teilausdruck `#<n>`.

1.5.6 Beispiele:

```
(.)\1+          findet 'aaaa' und 'cc'.
(.+)\1+        findet auch 'abab' und '123123'
(\['"]?)(\d+)\\1 findet "13" (innerhalb "), oder '4' (innerhalb ') oder auch 77, etc.
```

1.6 Modifikatoren

Modifikatoren sind dazu da, das Verhalten von TRegExpr zu verändern.

Es gibt viele Wege, die weiter unten beschriebenen Modifikatoren zu nutzen. Jeder der Modifikatoren lann eingebettet werden im Suchmuster des regulären Ausdruckes mittels des Konstruktes (?...).

Du kannst allerdings auch die meisten Modifikatoren beeinflussen, indem Du den entsprechenden TRegExpr-Eigenschaften die passenden Werte zuweist (Beispiel: Zuweisung an ModifierX oder ModifierStr für alle Modifikatoren zugleich).

Die Standardwerte für neue Instanzen von TRegExpr-Objekte sind definiert in *globalen Variablen*. Beispielsweise definiert die globale Variable RegExprModifierX das Verhalten des Modifikators X und damit die Einstellung der TRegExpr-Eigenschaft ModifierX bei neu instantiierten TRegExpr-Objekten.

Führe die Suche Schreibweisen-unabhägig durch (allerdings abhängig von den Einstellungen in Deinem System, Lokale Einstellungen), (beachte auch die InvertCase)

Behandle den Zielstring als mehrzeiligen String. Das bedeutet, ändere die Bedeutungen von „^“ und „$“: Statt nur den Anfang oder das Ende des Zielstrings zu finden, wird jeder Zeilenseparator innerhalb eines Strings erkannt (beachte auch die Zeilenseparatoren)

Behandle den Zielstring als einzelne Zeile. Das bedeutet, dass `.` jedes beliebige Zeichen findet, sogar Zeilenseparatoren (beachte auch Zeilenseparatoren), die es normalerweise nicht findet.

Modifikator für den „Genügsam“-Modus. Durch das Ausstellen werden alle folgenden Operatoren in den „Genugsam“-Modus. Standardmassig sind alle Operatoren „gierig“. Wenn also der Modifikator `/g` aus ist, dann arbeitet + wie `+?`, `*` als `*?` etc.

Erweitert die Lesbarkeit des Suchmusters durch Whitespace und Kommentare (beachte die Erklärung unten).

Modifikator. Falls er gesetzt ist, beinhaltet die Zeichenklasse − zusätzliche russissche Buchstaben , − beinhaltet zusätzlich , und − beinhaltet alle russischen Symbole.

Sorry für fremdsprachliche Benutzer, er ist gesetzt standardmässig. Falls Du ihn ausgeschaltet haben willst standardässig, dann setze die globale Variable *RegExprModifierR* auf false.

Der Modifikator /x selbst braucht etwas mehr Erklärung. Er sagt TRegExpr, dass er allen Whitespace ignorieren soll, der nicht escaped oder innerhalb einer Zeichenklasse ist. Du kannst ihn benutzen, um den regulären Ausdruck in kleinere, besser lesbare Teile zu zerlegen. Das Zeichen # wird nun ebenfalls als Metazeichen behandelt und leitet einen Kommentar bis zum Zeilenende ein.

1.6.1 Beispiel:

```
(
(abc) # Kommentar 1
  |   # Du kannst Leerschläge zur Formatierung benutzen - TRegExpr
ignoriert sie
(efg) # Kommentar 2
)
```

Dies bedeutet auch, wenn Du echten Whitespace oder das # im Suchmuster haben möchtest (ausserhalb einer Zeichenklasse, wo sie unbehelligt von `/x` sind), dann muss der entweder escaped oder mit der hexadezimalen Schreibweise angegeben werden. Beides zusammen sorgt dafür, dass reguläre Ausdrücke besser lesbar werden.

1.7 Perl Erweiterungen

Dies kann benutzt werden in Regulären Ausdrücken, um Modifikatoren innerhalb eines Ausdruckes im Flug zu ändern. Wenn dieses Konstrukt innerhalb eines Teilausdruckes erscheint, betriefft er auch nur diesen.

1.7.1 Beispiele:

```
(?i)Saint-Petersburg       findet 'Saint-petersburg' und'Saint-Petersburg'
(?i)Saint-(?-i)Petersburg findet 'Saint-Petersburg', aber nicht'Saint-petersburg'
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
(?i)(Saint-)?Petersburg   findet 'Saint-petersburg' und'saint-petersburg'
((?i)Saint-)?Petersburg   findet 'saint-Petersburg', aber nicht'saint-petersburg'
```

(?#text)

Ein Kommentar, der Text wird ignoriert. Beachte, dass TRegExpr den Kommentar abschliesst, sobald er eine) sieht. Es gibt also keine Möglichkeit, das Zeichen) im Kommentar zu haben.

Bis heute empfehle ich Dir die FAQ (zu lesen, speziell zu den Fragen der Optimierungen beim „genügsamen“ Modus).

> **This is very old and outdated translation.** *If you can read English or Russian please use up-to-date* English version *or* Russian version.
>
> *If you want to help to update the translation please* contact me. New translation is based on GetText and can be edited with transifex.com. It is already machine-translated and need only proof-reading and may be some copy-pasting from here.

1.8 Public Methoden und Eigenschaften von TRegExpr:

```
class function VersionMajor: integer;
class function VersionMinor: integer;
```

Sie geben die grosse und kleine Versionsummer zurück, Beispiel 0.944 ergibt: VersionMajor = 0 und VersionMinor = 944

```
property Expression : string
```

Regulärer Ausdruck

Aus Optimierungsgründen übersetzt TRegExpr den regulären Ausdruck in den P-Code, den Du kannst ihn sehen mittels der Methode Dump. Der P-Code wird in den internen Strukturen gespeichert.

Eine [Neu]Übersetzung findet nur statt, wenn sie wirklich benötigt wird, beim Aufruf von Exec, ExecNext, Substitute, Dump etc. und auch dann nur, wenn der reguläre Ausdruck oder eine ihn betreffende Eigenschaft geändert wurde seit der letzten [Neu]Übersetzung.

Falls ein Übersetzungsfehler auftaucht, wird die Methode Error aufgerufen. Diese erzeugt standardmässig eine Ausnahme vom Typ ERegExpr – siehe unten

```
property ModifierStr : string
```

Setze / hole die Standardwerte für die Modifikatoren. Modifikatoren in Regulären Ausdrücken (?ismx-ismx) ersetzen diese Standardwerte. Falls Du nicht unterstützte Modifikatoren setzst, wird die Methode Error aufgerufen, die standardmässig eine Ausnahme vom Typ EregExpr erzeugt.

```
property ModifierI : boolean
```

Modifikator /i – Gross- oder Kleinschreibweise wird nicht berücksichtigt. Standardmässig False

```
property ModifierR : boolean
```

Modiifikator /r – benutze die für Russen erweiterte Syntax. Standardmässig True. (war die Eigenschaft ExtSyntaxEnabled in früheren Versionen)

```
property ModifierS : boolean
```

Modifikator /s - `.` findet jedes beliebige Zeichen (sonst wie `[^\n]`). Standardmässig `True`.

```
property ModifierG : boolean
```

Modifikator /g - schaltet alle Operatoren in den genügsamen Modus.

Falls ModifierG False ist, dann arbeitet `*` als `*?`, und `+` als `+?` und so weiter. Standardmässig ``True`.

```
property ModifierM : boolean
```

Modifikator /m – Behandelt den Zielstring als mehrzeiligen String. So finden „^“ und „$“ nicht mehr nur den Anfang und das Ende des Zielstringes, sondern auch Zeilenseparatoren innerhalb des Zielstrings. Standardmässig False.

```
property ModifierX : boolean
```

Modifikator /x – Erweiterte Syntax, erlaubt das Formatieren des regulärenm Ausdruckes zur besseren Lesbarkeit. Standardmässig False.

```
function Exec (const AInputString : string) : boolean;
```

Lässt einen Regulären Ausdruck auf einem Zielstring ablaufen. Exec speichert AInputString in der Eigenschaft InputString

For Delphi 5 and higher available overloaded versions:

```
function Exec : boolean;
```

without parameter (uses already assigned to InputString property value)

```
function Exec (AOffset: integer) : boolean;
```

is same as ExecPos

```
function ExecNext : boolean;
```

Findet nächsten Treffer:

```
ExecNext;
```

Arbeitet gleich wie

```
if MatchLen [0] = 0
    then ExecPos (MatchPos [0] + 1)
    else ExecPos (MatchPos [0] + MatchLen [0]);
```

ist aber viel einfacher!

Raises exception if used without preceeding successful call to

Exec* (Exec, ExecPos, ExecNext). So You always must use something like

```
if Exec (InputString) then repeat { proceed results} until not ExecNext;
```

```
function ExecPos (AOffset: integer = 1) : boolean;
```

Findet einen Treffer im Zielstring, jedoch beginnend ab Position Aoffset. (Hinweis: AOffset=1 – das erste Zeichen im Zielstring)

```
property InputString : string;
```

Gibt den aktuellen Zielstring zurück (vom letzten Exec-Aufruf oder der letzten Zuweisung an diese Eigenschaft. Eine Zuweisung an diese Eigenschaft löscht die Match*-Eigenschaften!

```
function Substitute (const ATemplate : string) : string;
```

Gibt ATemplate mit durch $& oder $0 ersetztem Regulären Ausdruck und durch die Vorkommen von Regulären Unterausdrücken ersetzten $n zurück. Seit Version v.0.929 wird das ‚$' anstelle des ‚ verwendet (aus Gründen der künftigen Erweiterbarkeit und der besseren Kompatibilität zu Perl) und es akzeptiert mehr als eine Ziffer. Falls Du die Zeichen $ oder \ als Literale in einem Template verwenden möchtest, nutze das vorangestellte Escape-Zeichen: Beispiel:

```
'1\$ is $2\\rub\\' -> '1$ is <Match[2]>\rub\'
```

Falls Du eine Ziffer als Literal hinter einem $n plazieren möchtest, dann musst Du das n mit geschweiften Klammern {} begrenzen: Beispiel:

```
'a$12bc' -> 'a<Match[12]>bc'
'a${1}2bc' -> 'a<Match[1]>2bc'.
```

procedure Split (AInputStr : string; APieces : TStrings);

Zerlege AInputStr in die Einzelteile APieces mit den Treffern des Regulären Ausdruckes als Trenner

```
function Replace (AInputStr : RegExprString;

 const AReplaceStr : RegExprString;

 AUseSubstitution : boolean = False) : RegExprString;

function Replace (AInputStr : RegExprString;

 AReplaceFunc : TRegExprReplaceFunction) : RegExprString;

function ReplaceEx (AInputStr : RegExprString;

 AReplaceFunc : TRegExprReplaceFunction)  : RegExprString;
```

Gibt AInputStr mit den Treffern des regulären Ausdruckes ersetzt durch AReplaceStr. Wenn AUseSubstitution true ist, wird AReplaceStr genutzt als Vorlage für die Ersetzungsmethoden.

Beispiel:

Expression := ‚({-i}block|var)s*(s*([^]*)s*)s*';

Replace (‚BLOCK(test1)', ‚def „$1" value „$2"', True);

gibt zurück: def ‚BLOCK' value ‚test1'

Replace (‚BLOCK(test1)', ‚def „$1" value „$2"', False)

gibt zurück: def „$1" value „$2"

Ruft intern Exec[Next] auf.

Overloaded version and ReplaceEx operate with call-back function,

so You can implement really complex functionality.

property SubExprMatchCount : integer; // ReadOnly

Die Anzahl der Unterausdrücke, die beim letzten Exec-Aufruf gefunden wurde. Falls keine Unterausdrücke gefunden wurden, aber der gesamte Reguläre Ausdruck schon (Exec gab True zurück), ist dieser Wert 0. Falls weder Unterausdrücke noch der gesamte Reguläre Ausdruck gefunden wurde (Exec gab False zurück), dann ist dieser Wert –1. Beachte, dass einige Unterausdrücke eventuell nicht gefunden werden können und für solche Unterausdrücke gilt:

MathPos=MatchLen=-1 and Match=‘,.

Beispiel: Ausdruck := ‚(1)?2(3)¿;

Exec (‚123‘): SubExprMatchCount=2, Match[0]=‚123‘, [1]=‚1‘, [2]=‚3‘

Exec (‚12‘): SubExprMatchCount=1, Match[0]=‚23‘, [1]=‚1‘

Exec (‚23‘): SubExprMatchCount=2, Match[0]=‚23‘, [1]=‘,, [2]=‚3‘

Exec (‚2‘): SubExprMatchCount=0, Match[0]=‚2‘

Exec (‚7‘) - ergibt False: SubExprMatchCount=-1

property MatchPos [Idx : integer] : integer; // ReadOnly

Position des Starts des Unterausdruckes mit der Nummer Idx, gefunden beim letzten Exec-Aufruf. Der erste Unterausdruck hat Idx=1, der Letzte – MatchCount. Der gesamte Reguläre Ausdruck hat Idx=0. Gibt –1 zurück, wenn entweder der gewünschte Unterausdruck im Regulären Ausdruck nicht vorhanden ist oder im Zielstring nicht gefunden wurde.

property MatchLen [Idx : integer] : integer; // ReadOnly

(* Die Länge des Unterausdruckes mit der Nummer Idx. Numerierung und Rückgabewert wie bei MatchPos. *)

property Match [Idx : integer] : string; // ReadOnly

== copy (InputString, MatchPos [Idx], MatchLen [Idx])

Gibt einen Leerstring zurück, wenn entweder der gewünschte Unterausdruck im Regulären Ausdruck nicht vorhanden ist oder im Zielstring nicht gefunden wurde

function LastError : integer;

Gibt die ID des letzten Fehler zurück, 0 für keinen Fehler. Nicht zu verwenden, wenn die Error Methode eine Ausnahme erzeugt. Setzt den internen Fehlerzustand zurück auf 0.

function ErrorMsg (AErrorID : integer) : string; virtual;

Gibt die Fehlermeldung zur Fehler-ID AErrorID zurück.

property CompilerErrorPos : integer; // ReadOnly

Gibt die Position im Regulären Ausdruck zurück, wo der Compiler beim Übersetzen stoppte. Nützlich bei der Fehlerdiagnose.

property SpaceChars : RegExprString

Beinhaltet die Zeichen, die für das Metazeichen \s verwendet werden. Anfänglich gefüllt mit der globalen Konstanten RegExprSpaceChars.

property WordChars : RegExprString

Beinhaltet die Zeichen, die für das Metazeichen \w verwendet werden. Anfänglich gefüllt mit der globalen Konstanten RegExprWordChars.

```
property LineSeparators : RegExprString
```

Beinhaltet die Zeichen, die für Zeilenseparatoren wie \n in UNIX verwendet werden. Anfänglich gefüllt mit der globalen Konstanten RegExprLineSeparators. Beachte auch Zeilenseparatoren

property LinePairedSeparators : RegExprString

Beinhaltet die Zeichen, die paarweise für Zeilenseparatoren wie \rn in DOS/Windows verwendet werden. Es müssen genau zwei oder gar keine Zeichen sein. Anfänglich gefüllt mit der globalen Konstanten RegExprLinePairedSeparators. Beachte auch Zeilenseparatoren

Beispiel: Wenn Du den UNIX-Stil als Zeilenseparatoren haben möchtest, dann weise LineSeparators := #$a (Newline Zeichen) und LinePairedSeparator := ,‘ (Leerstring) zu. Wenn Du als Zeilenseparatoren nur genau \x0Dx0A akzeptieren möchtest, jedoch nicht \x0D oder \x0A aleine, dann weise LineSeparators := ,‘ (Leerstring) und LinePairedSeparator := #$d#$a zu.

Standardmässig ist der gemsichte Modus aktiv wie er definiert ist in den globalen Konstanten RegExprLine[Paired]Separator[s]: LineSeparators := #$d#$a; LinePairedSeparator := #$d#$a. Das Verhalten dieses Modus wird ausführlich im Abschnitt Syntax besprochen.

class function InvertCaseFunction (const Ch : REChar) : REChar;

Wandelt Ch in Grossschreibweise um, wenn er in Kleinschreibweise vorliegt oder umgekehrt. Die aktuellen lokalen System-Einstellungen werden dafür benutzt.

property InvertCase : TRegExprInvertCaseFunction;

Setze diese Eigenschaft, wenn Du die [Umwandlungsfunktion] zwischen der Gross- oder Kleinschreibung durch eine eigene ersetzen möchtest. Standardmässig auf InvertCaseFunction gesetzt.

procedure Compile;

Übersetzt den regulären Ausdruck [erneut]. Nützlich für das interaktive Erstellen eines regulären Ausdruckes in einem Editor, zur Prüfung der Gültigkeit aller Parameter, etc.

function Dump : string;

Gibt den übersetzten Regulären Ausdruck in knapp verständlicher Form zurück.

1.9 Globale Konstanten

EscChar = ,‘; // ‚Escape‘-char (,‘ in common r.e.) used for escaping metachars (w, \d etc).

// it’s may be usefull to redefine it if You are using C++ Builder - to avoide ugly constructions

// like ‚\w+\\w+.\w+‘ - just define EscChar=‘/‘ and use ‚/w+/w+/./w+‘

Standardmässig für Modifikatoren

```
RegExprModifierI : boolean = False;                // TRegExpr.ModifierI
RegExprModifierR : boolean = True;                // TRegExpr.ModifierR
RegExprModifierS : boolean = True;                // TRegExpr.ModifierS
RegExprModifierG : boolean = True;                // TRegExpr.ModifierG
RegExprModifierM : boolean = False;                //TRegExpr.ModifierM
RegExprModifierX : boolean = False;                //TRegExpr.ModifierX

RegExprSpaceChars : RegExprString = ' '#$9#$A#$D#$C; // Standardbelegung
für die Eigenschaft SpaceChars

RegExprWordChars : RegExprString =     '0123456789'  +
'abcdefghijklmnopqrstuvwxyz'  + 'ABCDEFGHIJKLMNOPQRSTUVWXYZ_';  //
Standardbelegung für die Eigenschaft WordChars
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
RegExprLineSeparators : RegExprString =  #$D#$A{$IFDEF
UniCode}#$B#$C#$2028#$2029#$85{$ENDIF};  // Standardbelegung für die
Eigenschaft LineSeparators

RegExprLinePairedSeparators : RegExprString = ' '#$D#$A;  //
Standardbelegung für die Eigenschaft LinePairedSeparators

RegExprInvertCaseFunction : TRegExprInvertCaseFunction =
TRegExpr.InvertCaseFunction;
```

// Standardbelegung für die Eigenschaft InvertCase

function RegExprSubExpressions (const ARegExpr : string;

ASubExprs : TStrings; AExtendedSyntax : boolean = False) : integer;

Erzeugt eine Liste der Teilausdrücke in einem regulären Ausdruck.

In ASubExps repräsentiert jeder String einen Teilausdruck, beginnend mit dem ersten bis zum letzten, im Format:

String – Teilausdruck-Text (ohne die Klammern ‚()‘)

Low Word (TString.Object) - Startposition im ARegExpr, inklusive ‚(, falls einer existiert (die erste Position ist 1)

High Word (TString.Object) – Länge, inklusive Start-‚(, und End-‚)‘ falls einer existiert.

AExtendedSyntax - must be True if modifier /x will be On while

using the r.e.

Nützlich für GUIs für Editoren für reguläre Ausdrücke etc. (Du findest ein Beispiel davon im Projekt)

Result code Meaning

---

0 Success. No unbalanced brackets was found;

-1 there are not enough closing brackets ‚)‘;

-(n+1) at position n was found opening ‚[, without corresponding closing ‚]‘;

n at position n was found closing bracket ‚)‘ without corresponding opening ‚(,.

// Falls Result <> 0, dann könnten in ASubExprs auch leere Items enthalten sein.

1.10 Nützliche globale Functionen

function ExecRegExpr (const ARegExpr, AInputStr : string) : boolean;

True, wenn in AInputString der Reguläre Ausdruck AregExpr gefunden wird. Erzeugt eine Ausnahme, wenn es Syntaxfehler hat in AregExpr

procedure SplitRegExpr (const ARegExpr, AInputStr : string; APieces : TStrings);

Zerlegt AInputStr in die Einzelteile APieces getrennt durch die Treffer des Regulären Ausdruckes ARegExpr.

function ReplaceRegExpr (const ARegExpr, AInputStr, AReplaceStr : string) : string;

Gibt AInputStr mit den Treffern des regulären Audruckes ersetzt durch AReplaceStr. Wenn AUseSubstitution true ist, wird AReplaceStr genutzt als Vorlage für die Ersetzungsmethoden.

Beispiel:

ReplaceRegExpr (‚({-i}block|var)s*(s*([^]*)s*)s*‘,

‚BLOCK(test1)‘, ‚def „$1“ value „$2“‘, True)

gibt zurück: def ‚BLOCK‘ value ‚test1‘

ReplaceRegExpr (‚({-i}block|var)s*(s*([^]*)s*)s*‘,

‚BLOCK(test1)‘, ‚def „$1“ value „$2“‘)

gibt zurück: def „$1“ value „$2“

ReplaceRegExpr (‚({-i}block|var)s*(s*([^]*)s*)s*‘,

‚BLOCK(test1)‘, ‚def „$1“ value „$2“‘, True)

gibt zurück: def ‚BLOCK‘ value ‚test1‘

ReplaceRegExpr (‚({-i}block|var)s*(s*([^]*)s*)s*‘,

‚BLOCK(test1)‘, ‚def „$1“ value „$2“‘)

gibt zurück: def „$1“ value „$2“

function QuoteRegExprMetaChars (const AStr : string) : string;

Ersetze alle Metazeichen durch deren sichere Repräsentationen. Beispiel:

‚abc$cd.(‘ wird gewandelt in ‚abc$cd.(‘

Diese Funktion ist nützlich, wenn ein Benutzer einen Regulären Ausdruck selbst zusammenstellen darf, ohne sich um das Escaping kümmern zu müssen.

Ausnahme Typ

Die standardmässige Fehlerbehandlungsroutine erzeugt folgende Ausnahme:

ERegExpr = class (Exception)

public

ErrorCode : integer; // Error-Code. Übersetzungsfehler haben Codes < 1000.

CompilerErrorPos : integer; // Position im Regulären Ausdruck, wo der Übersetzungsfehler auftauchte

end;

1.11 Wie wird Unicode benutzt?

TRegExpr unterstützt nun UniCode, aber leider sehr langsam :(

Wer möchte dies optimieren? ;)

Benütze es nur, wenn Du wirklich nicht auf Unicode-Unterstützung verzichten kannst!

Entferne ‚.‘ aus {.$DEFINE UniCode} in regexpr.pas. Danach werden alle Strings als Delphis WideString (= Unicode) behandelt

> **This is very old and outdated translation.** *If you can read English or Russian please use up-to-date* English version *or* Russian version.
>
> *If you want to help to update the translation please* contact me. New translation is based on GetText and can be edited with transifex.com. It is already machine-translated and need only proof-reading and may be some copy-pasting from here.

1.12 F. Wie kann ich TRegExpr mit Borland C++ Builder benutzen?

Ich habe ein Problem, weil offensichtlich die Header Datei fehlt (.h or .hpp).

1.12.1 A.

- Füge RegExpr.pas zum BCB Projekt hinzu.
- Kompiliere das Projekt. Dies generiert die Header Datei RegExpr.hpp.
- Nun kann Programmcode geschrieben werden, der die RegExpr unit benutzt.Nicht vergessen den Verweis auf die Header Datei (#include „RegExpr.hpp“) einzufügen, wo dies nötig ist.
- Dont `forget to replace all\in regular expressions with`‘.

1.13 F. Why many r.e. (including r.e. from TRegExpr help and demo) work wrong in Borland C++ Builder?

1.13.1 A.

Please, reread answer to previous question ;) Symbol `\\` has special treting in C++, so You have to `escape` it (as described in prev.answer). But if You dont `like r.e. like\w+\\w+.\w+You can redefine constant EscChar (RegExpr.pas), for example EscChar=/- then r.e. will be`/w+/w+/./w+‘, sligtly unusual but more readable..

1.14 F. Weshalb gibt TRegExpr mehr als eine Zeile zurück?

Beispiel sei der reguläre Ausdruck `<font .\*>`, der das erste `<font` und danach den ganzen Rest des Eingabefiles zurückbringt inklusive dem letzten `</html>`.

1.14.1 A.

Aus Grunden der Abwarstkompatibilitat ist der Modifikator /s standardmassig eingeschaltet.

Schalte ihn aus und `.` findet alles ausser Zeilenseparatoren – Wie Du es wunschst.

BTW Ich schlage vor, Du nimmst `<font (\[^\\n>\]\*)>`, dann hast Du in Match[1] die URL.

1.15 F. Weshalb gibt TRegExpr mehr zurück als ich erwarte?

Beispiel sei der reguläre Ausdruck `<p>(.+)</p>`, angewandt auf den Zielstring `<p>a</p><p>b</p>`, der `a</p><p>b` zurückgibt, aber nicht nur das `a` wie erwartet.

1.15.1 A.

Standardmässig arbeiten alle Operatoren im „gierig“ Modus. Sie finden also soviel wie möglich.

Falls Du den „genügsamen“ Modus benutzen möchtest, so geht das nun ab Version 0.940. Da funktionieren Operatoren wie `+?` etc. in diesem minimalen Match-Modus. Du kannst auch alle Operatoren standardmässig in diesem Modus arbeiten lassen mit dem Einsatz des Modifikators `g` (benutze dazu die entsprechenden TRegExpr-Eigenschaften oder Inline-Konstrukte wie `?(-g)` im regulären Ausdruck).

1.16 F. Wie parse ich Quelltexte wie HTML mit Hilfe von TRegExpr?

1.16.1 A.

Sorry folks, aber das ist fast unmöglich!

Natürlich kann man TRegExpr benutzen, um Teile aus einem HTML-File zu extrahieren, wie ich ja auch zeige in den Beispielen. Aber wenn effektiv ein ganzes File geparsed (d.h. in seine Elemente erlegt werden soll), dann braucht man einen ausgewachsenen Parser, nicht nur einen Regulären-Ausdruck-Matcher. Eine umfassende Lektüre bietet beispielsweise das `Perl Cookbook` von Tom Christiansen und Nathan Torkington. In kurzen Worten, es gibt viele Konstruktionen, die ganz leicht mit echten Parsern, aber nicht mit regulären Ausdrücken zerlegt werden können. Zudem ist ein echter Parser viel schneller beim Zerlegen des Files, weil ein Regulärer-Ausdruck-Matcher nicht nur den Eingabetext liest, sondern ein optimiertes Suchmuster aufbaut, was viel Zeit in Anspruch nehmen kann.

1.17 F. Gibt es einen Weg, mehrere Treffer eines Suchmusters zu erhalten?

1.17.1 A.

Du kannst eine Schleife mittels der ExecNext-Methode schreiben und so einen Treffer nach dem anderen herausholen.

Leichter kann es nicht gemacht werden, weil Delphi nicht wie Perl ein Interpreter ist. Als Compiler ist Delphi dafür schneller.

Falls Du ein Beispiel suchst, schaue Dir doch die Implementation von TRegExpr.Replace an oder das Beispiel in *HyperLinksDecorator.pas*

1.18 F. Ich überpfüfe die Eingabe des Benutzers. Weshalb gibt TRegExpr `True`

zurück für falsche Eingabestrings?

1.18.1 A.

In vielen Fällen vergessen die Benutzer von TRegExpr, dass er gemacht ist zur Suche im Eingabestring. Wenn Du also den Benutzer dazubringen möchtest, dass er nur 4 Ziffern eingibt und Du dazu den regulären Ausdruck `\\d{4,4}` benutzst, so wird dieser Ausdruck schon die 4 Ziffern in Eingaben wie `12345` oder `irgendwas 1234 und nochwas` erkennen. Eventuell hast Du nur vergessen, dass die 4 Ziffern alleine vorkommen sollen. Du müsstest also den regulären Ausdruck so schreiben: `^\\d{4,4}$`.

1.19 F. Weshalb arbeiten genügsame Operatoren manchmal wie ihre gierigen Gegenstücke?

Beispiel sei der reguläre Ausdruck `a+?,b+?` angewandt auf den String `aaa,bbb` findet `aaa,b`. Sollte er nicht `a,b` finden wegen des genügsamen ersten Iterators?

1.19.1 A.

Dies ist eine Einschränkung der von TRegExpr (und Perl und vielen UNIXen) verwandten Mathematik – reguläre Ausdrücke verwenden nur „einfache“ Suchoptimierungen, nicht unbedingt die beste Optimierung. In seltenen Fällen ist das nicht ausreichend, aber in den meisten Fällen ist es wohl eher ein Vorteil denn ein Nachteil: Und zwar aus Gründen der Performance und Vorhersagbarkeit des Resultats. Die Hauptregel ist: Die regulären Ausdrücke versuchen zuerst, von der aktuellen Stelle im Zielstring alle Varianten zu finden und nur wenn es absolut keinen Treffer gibt, wird vom Zielstring ein Zeichen vorwärtsgelesen und alles wiederholt. Wenn Du also `a,b+?` benutzt, dann findet es `a,b`. Im Falle von `a+?,b+?` ist es zwar nicht wünschenswert (wegen der genügsamen Iteratoren) aber möglich, mehrere `a`s zu finden, also findet TRegExpr sie auch und gibt einen korrekten, aber nicht unbedingt optimalen Treffer zurück. Genauso wie die regulären Ausdrücke Perl oder UNIX geht TRegExpr nicht so weit, dass es nach einem Zeichen weitergeht im Zielstring und erneut prüft, ob es einen „noch besseren Treffer“ gäbe. Zudem kann man hierbei überhaupt nicht von „schlechteren oder besseren Treffern“ sprechen. Bitte lies den Abschnitt für Syntax weitere Erläuterungen.

> **This is very old and outdated translation.** *If you can read English or Russian please use up-to-date* English version *or* Russian version.
>
> *If you want to help to update the translation please* contact me. New translation is based on GetText and can be edited with transifex.com. It is already machine-translated and need only proof-reading and may be some copy-pasting from here.

1.20 Einfache Beispiele

Falls Du nicht mit Regulären Ausdrücken bekannt bist, dann könntest Du etwas unter dem Abschnitt Syntax dazulernen oder schaue in ein gutes Perl oder Unix Buch.

1.21 Globale Routinen verwenden

Das ist zwar einfach, aber nicht besonders flexibel oder effektiv.

```
ExecRegExpr ('\\d{3}-(\\d{2}-\\d{2}|\\d{4})', 'Phone: 555-1234');
```

Rückgabewert: True

```
ExecRegExpr ('^\\d{3}-(\\d{2}-\\d{2}|\\d{4})', 'Phone: 555-1234');
```

Rückgabewert: False, weil es einige Zeichen vor der Telefonnummer hat und weil wir das Metazeichen ‚^‘ benutzen (Bedeutung von ‚^‘: BeginningOfLine)

```
ReplaceRegExpr ('product', 'Take a look at product. product is the best!', 'TRegExpr
↪');
```

Rückgabewert: ‚Take a look at TRegExpr. TRegExpr is the best ¡; ;)

1.22 Die TRegExpr-Klasse verwenden

Damit hast Du alle Möglichkeiten der Bibliothek zur Verfügung.

{% highlight pascal linenos %} // Diese einfache Funktion extrahiert alle E-Mail-Adressen aus dem InputString // und legt eine Liste dieser Adressen in den Rückgabewert function ExtractEmails (const AInputString : string) : string; const EmailRE = ‚[_a-zA-Zd-.]+@[_a-zA-Zd-]+(.[_a-zA-Zd-]+)+‘ var r : TRegExpr; begin Result := ‚‘; r := TRegExpr.Create; // Erzeuge Objekt try // garantiere Speicherfreigabe r.Expression := EmailRE; // der R.A. wird automatisch in die interne Struktur übersetzt // innerhalb der Zuweisung an diese Eigenschaft if r.Exec (AInputString) then REPEAT Result := Result + r.Match [0] + ‚‚ ‚; UNTIL not r.ExecNext; finally r.Free; end; end begin ExctractEmails (‚My e-mails is anso@mail.ru and anso@usa.net‘); // gibt zurück: ‚anso@mail.ru, anso@usa.net, , end. {% endhighlight %}

Beachte: Die Übersetzung eines Regulären Ausdruckes beansprucht etwas Zeit während der Zuweisung. Wenn Du also diese Funktion öfters ausführst, dann erzeugst Du damit unnötigen Aufwand. Wenn der Reguläre Ausdruck also konstant bleibt, dann kannst Du dies beträchtlich optimieren, indem Du diese Zuweisung nur während der Initialisierungsphase Deines Projektes ausführst.

{% highlight pascal linenos %} // Dieses einfache Beispie extrahiert Telefonnummern und // zerlegt sie in Teile (Stadt- und Landesvorwahl, interne Nummer). // Danach ersetzt es diese Teile im Template. function ParsePhone (const AInputString, ATemplate : string) : string; const IntPhoneRE = ‚(+d *)?((d+) *)?d+(-d*)*‘; var r : TRegExpr; begin r := TRegExpr.Create; try r.Expression := IntPhoneRE; if r.Exec (AInputString) then Result := r.Substitute (ATemplate) else Result := ‚‘; finally r.Free; end; end; begin ParsePhone (‚Phone of AlkorSoft (project PayCash) is +7(812)329-44-69‘, ‚Zone code $1, city code $2. Whole phone number is $&.‘); // Rückgabe: ‚Zone code +7, city code (812) . Whole phone number is +7(812) 329-44-69.‘ end. {% endhighlight %}

1.23 Etwas komplexere Beispiele

Du findest komplexere Beispiele für den Gebrauch von TRegExpr in den Projekten TestRExp.dpr und HyperLinkDecorator.pas.

Beachte bitte auch meine kleinen Artikel.

Ausführliche Erklärung

Bitte beachte dazu die TRegExpr-Interface-Beschreibung.

> **This is very old and outdated translation.** *If you can read English or Russian please use up-to-date* English version *or* Russian version.
>
> *If you want to help to update the translation please* contact me. New translation is based on GetText and can be edited with transifex.com. It is already machine-translated and need only proof-reading and may be some copy-pasting from here.

Die Demo ist ein einfaches Programm, um mit den Regulären Ausdrücken zu spielen, sie kennenzulernen und eigene Ausdrücke zu testen Es ist im Quelltext vorhanden im Projekt TestRExp.dpr) und als TestRExp.exe.

Beachte, dass es einige VCL-Eigenschaften verwendet, die nur in Delphi 4 oder höher vorhanden sind. Wenn Du es unter Delphi 2 oder 3 übersetzst, wirst Du Fehlermeldungen erhalten über unbekannte Eigenschaften. Du kannst diese Meldungen ignorieren – diese Eigenschaften werden nur fürs Anpassen der Komponenten auf dem Formular beim Ändern der Grösse des Formulars verwendet.

Mit Hilfe dieses Programmens kannst Du unter anderem leicht die Anzahl der Teilausdrücke eines komplexeren regulären Ausdruckes bestimmen. Du kannst zu jedem dieser Teilausdrücke springen, sowohl im Eingabe- wie auch im Zielstring, kannst mit den Funktionen Substitute, Replace und Split functions etc. spielen.

Darüberhinaus sind im Demo-Projekt zahlreiche Beispiele – benutze sie als Lernobjekte, um entweder die Syntax der regulären Ausdrücke oder TRegExpr kennenzulernen

1.24 Example: Hyper Links Decorator

- *DecorateURLs*
- *DecorateEMails*

Funktionen um reine URLs in HTML-Anker (aka Links) umzuwandeln.

Beispiel: Ersetze ‚http://regexpstudio.com' mit href="https://regexpstudio.com"'

oder ‚anso@mail.ru' mit ‚anso@mail.ru'.

1.24.1 Funktionen DecorateURLs

Findet URLs wie ‚http://. . . ' or ‚ftp://..', aber auch solche, die mit ‚www.' Beginnen. E-Mailadressen werden mit der Function *DecorateEMails* umgewandelt.

function DecorateURLs (const AText : string; AFlags : TDecorateURLsFlagSet = [durlAddr, durlPath]) : string;

Beschreibung

Gibt den Eingabetext AText mit umgewandelten URLs zurück

AFlags welche gefundenen Teile der URL müssen in den sichtbaren Teil. Beispiel: Wenn [durlAddr] dann wird die URL ‚http://anso.da.ru/index.htm' zu ‚anso.da.ru'

type

TDecorateURLsFlags = (durlProto, durlAddr, durlPort, durlPath, durlBMark, durlParam);

TDecorateURLsFlagSet = set of TDecorateURLsFlags;

Beschreibung

Dies sind die möglichen Werte:

Benennung Bedeutungen

---

durlProto Protokoll (wie ‚ftp://' or ‚http://')

durlAddr TCP Adresse oder Domain-Name (wie ‚anso.da.ru')

durlPort Port falls angeben (wie ‚:8080')

durlPath Pfad zum Dokument (wie ‚index.htm')

durlBMark Bookmark (wie ‚#mark')

durlParam URL Parameters (wie ‚?ID=2&User=13')

1.24.2 Funktionen DecorateEMails

Ersetzt alle korrekten E-Mails-URLs mit ‚ADDR' Beispiel: Ersetze ‚anso@mail.ru' mit ‚anso@mail.ru'.

function DecorateEMails (const AText : string) : string;

Beschreibung

Gibt den Eingabetext AText mit umgewandelten E-Mail-Links zurück