TRegExpr Documentation

Release 0.952

Andrey Sorokin

27.05.2020

Inhaltsverzeichnis

	English		Deutsch		Français	Español

Die TRegExpr-Bibliothek implementiert reguläre Ausdrücke.

Reguläre Ausdrücke sind ein benutzerfreundliches und leistungsfähiges Werkzeug für anspruchsvollere Such- und Ersetzungsaufgaben sowie für vorlagenbasiertes Überprüfen von Text.

Besonders nützlich sind sie zum Prüfen von Benutzereingaben in Eingabemasken, zum Validieren von E-Mail-Adressen usw.

Auch können Sie damit Telefonnummern, Postleitzahlen usw. aus Webseiten oder Dokumenten extrahieren, nach komplexen Mustern in Protokolldateien suchen und was Sie sich sonst noch in der Art vorstellen können. Die Regeln (Vorlagen) lassen sich ändern, ohne die gesamte Anwendung neu kompilieren zu müssen.

TRegExpr ist 100% in Pascal implementiert. Es ist Bestandteil von Lazarus (Free Pascal) , aber auch als separate Bibliothek nutzbar und kann auch mit Delphi 2-7 oder dem Borland C ++ Builder 3-6 kompiliert werden.

KAPITEL 1

Rezensionen

Machen Sie sich ein Bild von der Resonanz bei den Anwendern.

KAPITEL 2

Schnellstart

Um die Bibliothek zu verwenden, fügen Sie einfach den Quelltext Ihrem Projekt hinzu und nutzen Sie die Klasse TRegExpr.

In den FAQ können Sie aus den Problemen anderer Nutzer lernen.

Die benutzerfreundliche Windows-Anwendung REStudio hilft Ihnen dabei, reguläre Ausdrücke zu erlernen und zu debuggen.

KAPITEL 3

Feedback

Wenn Sie auf ein Problem stoßen, erstellen Sie bitte einen Fehlerbericht.

KAPITEL 4

Quellcode

Ausschließlich Object Pascal.

- Originalversion
- FreePascal-Fork (GitHub-Spiegel der Subversion)

KAPITEL 5

Dokumentation

	Englisch		Deutsch		Français	Español

5.1 Reguläre Ausdrücke (RegEx)

5.1.1 Einführung

Reguläre Ausdrücke sind eine praktische Möglichkeit, Textmuster anzugeben.

Mit regulären Ausdrücken können Sie Benutzereingaben validieren, nach Mustern wie E-Mail-Adressen oder Telefonnummern auf Webseiten oder in Dokumenten suchen und so weiter.

Nachfolgend auf einer einzigen Seite ein Spickzettel zum Thema reguläre Ausdrücke.

5.1.2 Zeichen

Einfache Übereinstimmungen

Jedes einzelne Zeichen passt zu sich selbst.

Eine Reihe von Zeichen stimmt mit dieser Reihe von Zeichen in der Eingabezeichenfolge überein.

RegEx	passt auf
`foobar`	`foobar`

Nicht druckbare Zeichen (Escape-Codes)

Um nicht druckbare Zeichen in regulären Ausdrücken darzustellen, verwenden Sie `\x..`:

RegEx	passt auf
`\xnn`	Zeichen mit Hex-Code `nn`
`\x {nnnn}`	Zeichen mit Hex-Code `nnnn` (ein Byte für Klartext und zwei Byte für Unicode)
`foo\x20bar`	`foo bar` (Beachten Sie das Leerzeichen in der Mitte)

Es gibt eine Reihe vordefinierter `Escape-Codes` für nicht druckbare Zeichen, genau wie in der Sprache `C`:

RegEx	passt auf
`\t`	(Horizontal-)Tabulator (HT / TAB), identisch mit `\x09`
`\n`	Zeilenschaltung (Newline, NL), wie `\x0a`
`\r`	Wagenrücklauf (carrier return, CR), das gleiche wie ‘‘ x0d‘‘
`\f`	Formularvorschub (FF), wie `\x0c`
`\a`	Alarm (BEL), gleichbedeutend mit `\x07`
`\e`	Escape (ESC), genauso wie `\x1b`
`\cx`	Kontrolltasten-Kombination (`Ctrl-x`) Beispielsweise passt `\ci` auf die Zielsequenz `\x09`, weil `ctrl-i` dem Wert `0x09` entspricht

Maskierung (Escaping)

Wenn Sie das Zeichen `\` als solches, also nicht als Teil eines `escape code` darstellen wollen, setzen Sie einfach noch ein `\` davor, also: `\\`.

Tatsächlich können Sie jedem Zeichen, das in regulären Ausdrücken eine Sonderbedeutung hat, ein `\` als „„„„Flucht-symbol““““ (`escape`) voranstellen.

RegEx	passt auf
`\^FooBarPtr`	`^FooBarPtr` (Dies ist ein literales `^` und nicht das Metazeichen für *Zeilenanfang*)
`\[a\]`	`[a]` (Dies ist ein a in eckigen Klammern, nicht eine aus a bestehende *Zeichenklasse*)

5.1.3 Zeichenklassen

Benutzerdefinierte Zeichenklassen

Eine Zeichenklasse ist eine Liste von Zeichen in `[]`. Die Klasse steht für genau **ein** beliebiges Zeichen aus dieser Liste.

RegEx	passt auf
`foob[aeiou]r`	`foobar`, `foober` usw., nicht aber `foobbr`, `foobcr` etc.

Sie können die Klasse „invertieren“ - wenn als erstes Zeichen nach dem `[` ein `^` steht, dann steht die Klasse ebenfalls für **ein** beliebiges Zeichen, allerdings eines, das **nicht** in der Liste vorkommt.

RegEx	passt auf
`foob[^aeiou]r`	`foobbr`, `foobcr` usw., aber nicht `foobar`, `foober` etc.

Innerhalb einer Liste wird der Bindestrich − verwendet, um einen Bereich anzugeben, z.B. passt `a−z` für jedes Zeichen von `a` bis `z` (einschließlich).

Wenn ein Bindestrich − selbst Bestandteil der Klasse sein soll, schreiben Sie ihn an den Anfang oder das Ende der Liste oder *maskieren* ihn per vorangestelltem Rückwärtsschrägstrich (Backslash) `\` .

If you want `]` as part of the class you may place it at the start of list or *escape* it with a backslash.

RegEx	passt auf
`[-az]`	`a`, `z` und −
`[az-]`	`a`, `z` und −
`[a\-z]`	`a`, `z` und −
`[az]`	Zeichen von `a` bis `z`
`[\n-\x0D]`	Zeichen von `#10` bis `#13`

Vordefinierte Zeichenklassen

Es gibt eine Reihe vordefinierter Zeichenklassen, die es erlauben, reguläre Ausdrücke übersichtlich zu halten.

RegEx	passt auf
`\w`	ein alphanumerisches Zeichen (einschließlich _)
`\W`	ein nichtalphanumerisches Zeichen
`\d`	ein numerisches Zeichen (dasselbe wie `[0123456789]`)
`\D`	ein nichtnumerisches Zeichen
`s`	ein beliebiger Leerraum (dasselbe wie `[ \t\n\r\f]`)
`\S`	ein Zeichen, das keinen Leerraum ergibt
`\h`	Horizontaler Leerraum, d.h. ein Horizontaltabulator oder jedes Zeichen in der Unicode-Kategorie „space separator“.
`\H`	jedes Zeichen, das kein horizontaler Leerraum ist
`\v`	vertikaler Leerraum, jedes Zeichen, das in die Unicode-Kategorie `Trenner, Abstand (space separator)` gehört.
`\V`	jedes Zeichen, das *nicht* einen vertikalen Abstand liefert

Sie können `\w`, `\d` und `\s` innerhalb von *benutzerdefinierten Zeichenklassen* verwenden.

RegEx	passt auf
`foob\dr`	`foob1r`, `foob6r` usw., aber nicht `foobar`, `foobbr` usw.
`foob[\w\s]r`	`foobar`,‘‘foob r‘‘, `foobbr` und so weiter, aber nicht‘‘foob1r‘‘, `foob=r` usw.

Bemerkung: TRegExpr

Die Zeichenklassen `\w`, `\W`, `\s` und `\S` sind in den Properties SpaceChars und WordChars definiert.

Sie können diese Zeichenklassen also auch neu definieren.

5.1.4 Grenzen

Zeilengrenzen

RegEx	passt auf
`^`	Zeilenanfang
`$`	Zeilenende
`\A`	Beginn des Textes (Stringanfang)
`\Z`	Ende des Textes (Stringende)
`.`	ein beliebiges Zeichen (innerhalb derselben Zeile)
`^foobar`	`foobar` nur, wenn es am Anfang der Zeile steht
`foobar$`	`foobar` nur wenn es am Zeilenende steht
`^foobar$`	`foobar` nur wenn davor und danach kein sonstiges Zeichen innerhalb der Zeile steht
`foob.r`	`foobar`, `foobbr`, `foob1r` usw.

Das Metazeichen `^` passt auf den Anfang ,‘‘$‘‘ auf das Ende der Zeichenkette.

Eventuell wollen Sie eine Zeichenfolge als mehrzeiligen Text behandeln, in dem `^` auch nach jedem internen Zeilenende und `$` vor jedem Zeilenende passt. Dieses Verhalten erhalten Sie mit dem Schalter *modifier / m* .

Beachten Sie, dass sich in der Sequenz `\x0D\x0A` keine leere Zeile befindet.

Bemerkung: TRegExpr

Wenn Sie die Unicode-Version verwenden, dann stimmen `^`/`$` auch mit `\x2028`, `\x2029`, `\x0B`, `\x0C` or `\x85` überein.

Die Anker `\A` und `\Z` funktionieren genau wie `^` and `$`, außer dass sie auch dann nicht mehrmals übereinstimmen, wenn die Option‘modifier /m <#m>‘_ eingeschaltet ist.

Das Metazeichen `.` passt von Haus aus auf jedes Zeichen, solange jedoch der Schalter *Modifier / s* `Off` ist, dann mit Ausnahme von Zeilentrennzeichen.

Beachten Sie, dass `^.*$` nicht auf eine Zeichenfolge inmitten von `\x0D\x0A` (CRLF) passt, denn die CRLF-Sequenz gilt als untrennbare Zeilenschaltung. Er passt hingegen auf die leeren Zeichenfolge in der Sequenz `\x0A\x0D`, weil dies einfach die falsche Reihenfolge ist, um als Zeilenschaltung interpretiert zu werden.

Bemerkung: TRegExpr

Feinheiten der mehrzeiligen Verarbeitung können mit den Eigenschaften LineSeparators und LinePairedSeparator justiert werden.

Sie können also Zeilenschaltungen im Unix-Stil `\n` oder im DOS/Windows-Stil `\r\n` verwenden oder beides (Standardverhalten wie oben beschrieben).

Wenn Sie eine mathematisch korrekte Beschreibung bevorzugen, können Sie sie unter www.unicode.org finden <http://www.unicode.org/unicode/reports/tr18/>__.

Wortgrenzen (Ankerpunkte)

RegEx	passt auf
`\b`	eine Wortgrenze
`\B`	keine Wortgrenze (Ankerpunkt, der entweder beidseits von w oder W umgeben ist)

Eine Wortgrenze `\b` ist eine Stelle zwischen zwei Zeichen, von denen eines `\w` und das andere ein `\W` ist (Reihenfolge egal).

5.1.5 Quantifizierung

Quantifizierer

Auf jedes Element eines regulären Ausdrucks kann ein Quantifizierer folgen. Dieser gibt die Anzahl der Wiederholungen des Elements an.

RegEx	passt auf
`{n}`	genau `n` mal
`{n,}`	mindestens `n` mal
`{n,m}`	zumindest "n" aber nicht mehr als "m" mal
`*`	nicht bis beliebig oft vorkommend, alternative Notation: `{0,}`
`+`	mindestens einmal vorkommend, alternative Notation: `{1,}`
`?`	höchstens einmal vorkommend, alternative Notation: `{0,1}`

Innerhalb der geschweiften Klammern `{n, m}` geben Sie also die Mindest-(`n`) und Höchstanzahl (`m`) für das Vorkommen des voranstehenden Elements an.

Die Kurzform `{n}` entspricht `{n, n}` und stimmt mit genau `n` Vorkommen überein.

Das `{n,}` entspricht „mindestens `n` Mal“.

Es gibt keine Obergrenze für `n` oder `m`.

Wenn eine geschweifte Klammer in einem anderen Kontext auftritt, wird sie als normales Zeichen behandelt.

RegEx	passt auf
`foob.*r`	`foobar`, `foobalkjdflkj9r` und `foobr`
`foob.+r`	`foobar`, `foobalkjdflkj9r`` aber nicht `foobr`
`foob.?r`	`foobar`, `foobbr` und `foobr`, aber nicht `foobalkj9r`
`fooba{2}r`	`foobaar`
`fooba{2,}r`	`foobaar`, `foobaaar`, `foobaaaar` usw.
`fooba{2,3}r`	`foobaar` oder `foobaaar`, aber nicht `foobaaaar`
`(foobar){8,10}`	`8`, `9` oder `10` Vorkommen des Worts `foobar` (`()` ist ein *Unterausdruck (subexpression)*)

Gier (greediness)

Im `gierigen` (greedy) Modus versuchen *Quantifizierer* so viele Übereinstimmungen wie möglich zu treffen, im `trägen` Modus so wenig wie möglich.

Standardmäßig verhalten sich alle Quantifizierer `gierig`. Durch Anhängen eines Fragezeichens `?` kann man sie jedoch `träge` machen.

Für den String `abbbbc`:

RegEx	passt auf
`b+`	`bbbb`
`b+?`	`b`
`b*?`	leerer String
`b{2,3}?`	`bb`
`b{2,3}`	`bbb`

Alle Quantifizierer können Sie in den `trägen` (`non-greedy`) Modus umschalten (*modifier / g*), im nachfolgenden Beispiel als *Inline-Modifikation*) gezeigt.

RegEx	passt auf
`(?-g)b+`	`b`

5.1.6 Alternative Ausdrücke

Alternative Ausdrücke werden durch `|` getrennt.

`fee|fie|foe` passt also zu einem `fee`, `fie` oder `foe` in der Zielzeichenfolge (ebenso wie `fee|fie|foe`).

Zum ersten Ausdruck zählt alles vom letzten Musterbegrenzer (`(`, `[` oder dem Anfang des Musters) bis vor das erste `|` ``, und der letzte Ausdruck umfasst alles nach dem letzten ``| zum nächsten Musterbegrenzer.

Klingt etwas kompliziert, daher setzt man üblicherweise die Alternativausdrücke in Klammern, um damit ihren Anfang und das Ende deutlicher erkennbar zu machen.

Die alternativen Ausdrücke werden von links nach rechts durchgetestet, und der erste passende wird verwendet.

Beispielsweise wird der reguläre Ausdruck `foo|foot` in der Zeichenfolge `barefoot` nur `foo` ergeben. Dies war nämlich die erste passende Alternative.

Denken Sie auch daran, dass `|` innerhalb eckiger Klammern als Literal interpretiert wird. Wenn Sie also `[fee|fie|foe]` schreiben, bedeutet das wirklich nur die Zeichenklasse `[efio |]`.

RegEx	passt auf
`foo(bar|foo)`	`foobar` oder `foofoo`

5.1.7 Unterausdrücke (subexpressions)

Die Klammern `(...)` können auch verwendet werden, um Unterausdrücke in einem regulären Ausdruck zu definieren.

Bemerkung: TRegExpr

Positionen, Längen und tatsächliche Werte des Unterausdrucks werden in MatchPos,‘MatchLen <tregexpr.html#matchlen>‘_ und‘Match <tregexpr.html#match>‘_ angegeben.

Sie können sie durch Substitute ersetzen.

Unterausdrücke werden von links nach rechts durch ihre öffnenden Klammern nummeriert (einschließlich verschachtelter Unterausdrücke).

Der erste Unterausdruck hat die Nummer `1`. Der gesamte reguläre Ausdruck hat die Nummer `0`.

Zum Beispiel wird für die Eingabezeichenfolge `foobar` der reguläre Ausdruck `(foo(bar))` folgendes finden:

Unterausdruck	Wert
0	foobar
1	foobar
2	bar

5.1.8 Rückreferenzen

Die Metazeichen `\1` bis `\9` werden als Rückreferenzen interpretiert. `\n` stimmt dabei mit dem vorher übereinstimmenden Unterausdruck `n` überein.

RegEx	passt auf
`(.)\1+`	`aaaa` und `cc`
`(.+)\1+`	auch `abab` und `123123`

`(['"]?)(\d+)\1` passt auf `"13"` (in doppelten Anführungszeichen) genauso wie auch auf `'4'` (in einfachen Anführungszeichen) oder `77` (ohne Anführungszeichen) usw.

5.1.9 Modifikatoren

Modifikatoren gestatten eine Verhaltensänderung regulärer Ausdrücke.

Sie können Modifikatoren sowohl global in Ihrem System festlegen als auch das Verhalten eines regulären Ausdrucks quasi an Ort und Stelle mithilfe von *Inline-Modifikatoren* ändern.

Bemerkung: TRegExpr

Um den Wert eines Modifikators festzulegen, verwenden Sie entweder ModifierStr oder die entsprechende `TRegExpr`-Eigenschaft namens ‘Modifier* <tregexpr.html#modifieri> __.

Die Standardwerte sind in den globalen Variablen definiert. Beispielsweise legt die globale Variable ‘‘ RegExprModifierX‘‘ den Standardwert für die `ModifierX`-Eigenschaft fest.

---

i: Groß- und Kleinschreibung ignorieren

Groß- und Kleinschreibung wird nicht berücksichtigt. Verwendet ansonsten die in Ihrem System eingestellten Spracheinstellungen, siehe auch InvertCase.

m: mehrzeilige Zeichenketten

String zeilenweise als mehrzeiligen Text behandeln. `^` und `$` finden damit Anfang und Ende in jeder beliebigen Zeile innerhalb des Strings.

Siehe auch *Zeilengrenzen*.

s: einzeilige Zeichenfolgen

Gesamte Zeichenfolge als einzelne Zeile behandeln. `.` passt dann auf *jedes* beliebige Zeichen, insbesondere auch auf Zeilenwechsel.

Siehe auch *Zeilengrenzen*, die normalerweise nicht gefunden würden.

g: Gierigkeit

Bemerkung: In TRegExpr nur als Modifikator verfügbar.

Wenn Sie auf `Off` umschalten, bringen Sie alle *Quantifier* vom gierigen (greedy) in den ‘‘trägen (non-greedy)<#greedy>‘__-Modus.

Wenn also der Modifikator `/g Off` ist, funktioniert `+` als `+?`, `*` als `*?` und so weiter.

Standardmäßig ist dieser Modifikator `On`.

x: erweiterte Syntax

Ermöglicht den regulären Ausdruck zu kommentieren und in mehrere Zeilen aufzuteilen.

Wenn dieser Modifikator `On` ist, ignorieren wir alle Leerräume, sofern sie weder maskiert noch innerhalb einer Zeichenklasse stehen.

Und das Zeichen ‘‘ #‘‘ trennt Kommentare ab.

Eine mehrzeilige Darstellung macht übrigens reguläre Ausdrücke oft besser lesbar:

```
(
(abc) # Kommentar 1
#
(efg) # Kommentar 2
)
```

Dies bedeutet auch: wenn Sie literale Whitespace- oder `#` Zeichen in dem Muster (außerhalb einer Zeichenklasse, wo sie nicht von `/x` betroffen sind) angeben möchten, müssen Sie diese entweder maskieren oder als Oktal- oder Hex-Codes schreiben.

r: russische Zeichenbereiche

Bemerkung: In TRegExpr nur als Modifikator verfügbar.

In der russischen ASCII-Tabelle sind die Zeichen `ё` separat untergebracht.

Die übrigen großen und kleinen russische Schriftzeichen liegen jeweils in getrennten Bereichen, analog wie bei den englischen. Ich wollte eine praktische Kurzform.

Mit diesem Modifikator können Sie also statt `[ёа-я]` einfach `[а-я]` schreiben, wenn Sie sämtliche russischen Zeichen benötigen.

Wenn dieser Modifikator `On` ist:

RegEx	passt auf
–	Kleinbuchstaben, von bis sowie
–	Großbuchstaben, von A bis sowie
–	Sämtliche russischen Schriftzeichen

Dieser Modifikator ist standardmäßig Ein.

5.1.10 Erweiterungen

(?=<lookahead>)

Eine Vorauschauannahme (look-ahead assertion) prüft auf ein nachfolgendes Vorkommen des Ausdrucks <look-ahead>, aber erfasst diese nicht mit.

Bemerkung: TRegExpr

Look-Ahead ist in TRegExpr nicht implementiert.

In vielen Fällen können Sie look ahead durch einen *Unterausdruck* ersetzen und einfach ignorieren, was in diesem Unterausdruck erfasst wird.

Zum Beispiel arbeitet (blah) (? = foobar) (blah) prinzipiell genauso wie (blah) (foobar) (blah). In der letzteren Variante müssen Sie lediglich den mittleren Unterausdruck ausschließen - verwenden Sie also den Ausdruck Match [1] + Match [3] und ignorieren Sie Match[2].

Dies ist lediglich ein bisschen unpraktischer als in einer look-ahead-Version, in welcher Sie den gesamten Match[0] verwenden könnten, weil der vom look ahead erfasste Teil zwar gesucht, aber nicht in das Ergebnis mit einbezogen würde.

(?:<non-capturing group>)

?: wird genutzt, wenn man einen Ausdruck zwar gruppieren, aber keine Rückwärtsreferenz auf den passenden Textabschnitt speichern möchte.

Auf diese Weise können Sie Ihren Regex in Unterausdrücke gliedern, ohne unnötig Speicherplatz und Verarbeitungszeit für die Ergebniserfassung zu verschwenden:

RegEx	passt auf
(https?|ftp)://([^/\r\n]+)	https und sorokin.engineer in https://sorokin.engineer
(?:https?|ftp)://([^/\r\n]+)	nur sorokin.engineer in https://sorokin.engineer

(? imsgxr-imsgxr)

Damit kann man innerhalb von regulären Ausdrücken Modifikatoren „im Handumdrehen“ ändern.

Das Praktische daran kann sein, dass die Änderung direkt im Regex nur lokalen Gültigkeitsbereich hat. Sie betrifft sogar nur denjenigen Anteil des regulären Ausdrucks, der auf den Operator (?imsgxr-imsgxr) folgt.

Innerhalb eines Unterausdrucks gesetzt, wirkt sich der Operator nur innerhalb dessen aus, wiederum auch nur auf den Teil des Teilausdrucks, der auf den Operator folgt. In `((?i)Saint)-Petersburg` betrifft es nur den Teilausdruck `((?i) Saint)`, so dass `sankt-Petersburg`, aber nicht `Sankt-petersburg` passt .

RegEx	passt auf
`(?i)Sankt-Petersburg`	‘‘ Sankt-petersburg‘‘ und ‘‘ Sankt-Petersburg‘‘
`(?i)Sankt-(?-i)Petersburg`	`Sankt Petersburg` aber nicht `Sankt petersburg`
`(?i)(Sankt-)?Petersburg`	`Sankt-petersburg` und `sankt-petersburg`
`((?i)Sankt-)?Petersburg`	`saint-Petersburg`, aber nicht `saint-petersburg`

(?#Text)

Ein Kommentar („Text“ wird ignoriert).

Beachten Sie, dass der Kommentar durch die nächstfolgende) geschlossen wird. Es gibt also keine Möglichkeit, eine schließende runde Klammer) in den Kommentar einzufügen.

5.1.11 Nachwort

In diesem alten Blogeintrag aus dem vorigen Jahrhundert erläutere ich einige Anwendungsfälle von regulären Ausdrücken.

	Englisch		Deutsch		Français	Spanisch

5.2 TRegExpr

Implementiert reguläre Ausdrücke in reinem Pascal, kompatibel zu Free Pascal, Delphi 2 bis 7 und Borland C++ Builder 3 bis 6.

Um es zu nutzen, kopiere einfach den Quellcode der unit RegExpr in Dein Projekt.

In der IDE Lazarus (Free Pascal) ist diese Bibliothek bereits enthalten . Falls Sie also Lazarus verwenden, müssen Sie gar nichts kopieren.

5.2.1 Klasse TRegExpr

Haupt- und Nebenversion

Gibt die Haupt- und Nebenversionsnummer zurück, zum Beispiel für `Version 0.944`

```
Hauptversion = 0
Nebenversion = 944
```

Ausdruck

Ein regulärer Ausdruck.

Aus Optimierungsgründen wird der reguläre Ausdruck automatisch in `P-Code` kompiliert. Eine menschenlesbare Form des `P-Codes` wird gibt die Methode *Dump* aus.

Bei einem Kompilierfehler wird die `Error`-Methode aufgerufen (standardmäßig löst `Error` die Ausnahme *ERegExpr* aus)

ModifierStr

Werte für reguläre Ausdrücke setzen oder auslesen.

Die Syntax ist ähnlich wie in (?ismx-ismx). Zum Beispiel schaltet ‘‘ModifierStr := ‘i-x’‘ den Modifikator / i ein, ‘/x <regular_expressions.html#x> ‘__ aus und lässt die übrigen (nicht aufgeführten) unverändert.

Wenn Sie versuchen, einen nicht unterstützten Modifikator einzustellen, wird `Error` aufgerufen.

ModifierI

Modifier /i, „Groß- und Kleinschreibung wird nicht berücksichtigt“ <regular_expressions.html#i>, Initialwert ist *RegExprModifierI* value.

ModifierR

Modifier /r, “ russische Zeichenbereiche“, initialisiert mit dem Wert von *RegExprModifierR*.

ModifierS

Modifier /s, „einzeilige Zeichenketten „, initialisiert mit dem Wert von *RegExprModifierS* .

ModifierG

Modifier /g, „Gier (greediness)“, initialisiert mit dem Wert von *RegExprModifierG* .

ModifierM

Modifier /m, „mehrzeilige Zeichenkette“, initialisiert mit dem Wert von *RegExprModifierM* .

ModifierX

Modifier /x, „erweiterte Syntax“, initialisiert mit dem Wert von *RegExprModifierX* .

Exec

Wendet den regulären Ausdruck auf die Eingabezeichenfolge `AInputString` an.

Eine überladene `Exec`-Version ohne `AInputString` ist ebenfalls vorhanden, diese verwendet implizit den zuletzt benutzten `AInputString`.

Siehe auch die globale Funktion *ExecRegExpr*, die sich ohne explizite Erstellung eines `TRegExpr`-Objekts nutzen lässt.

ExecNext

Die nächste Übereinstimmung suchen.

Benötigt keinen Parameter, und funktioniert in der Art von

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

Löst eine Ausnahme aus, falls sie ohne vorherigen erfolgreichen Aufruf von *Exec*, *ExecPos* oder *ExecNext* verwendet wird.

Man muss also stets nach folgendem Schema vorgehen:

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

ExecPos

Durchsucht `InputString` ab der Position `AOffset`

```
AOffset = 1 // erstes Zeichen von InputString
```

InputString

Gibt die aktuelle Eingabezeichenfolge zurück (aus dem letzten *Exec*-Aufruf oder der letzten Zuweisung an diese Eigenschaft stammend).

Jede Zuweisung zu dieser Eigenschaft setzt *Match*, *MatchPos* und *MatchLen* zurück.

Substitute

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

Returns `ATemplate` with `$&` or `$0` replaced by whole regular expression and `$n` replaced by occurence of subexpression number `n`.

Um ein Dollarzeichen `$` oder einen Rückwärtsschrägstrich `\` in ATemplate einzufügen, sind sie mit `\`` zu maskieren (also ``\\` oder `\$`).

Symbol	Beschreibung
`$&`	ganze Übereinstimmung des regulären Ausdrucks
`$0`	ganze Übereinstimmung des regulären Ausdrucks
`$n`	regulärer Unterausdruck `n`
`\n`	in Windows durch `\r\n` ersetzt
`\l`	Kleinbuchstabe ein nächstes Zeichen
`\L`	Kleinbuchstaben alle Zeichen danach
`\u`	ein weiteres Zeichen
`\U`	Großbuchstaben alle Zeichen danach

```
'1\$ is $2\\rub\\' -> '1$ is <Match[2]>\rub\'
'\U$1\\r' wird zu '<Match[1] in uppercase>\r'
```

Wenn Sie eine Ziffer unmittelbar hinter `$n` schreiben möchten, müssen Sie `n` mit geschweiften Klammern `{}` abgrenzen.

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

Split

Splittet den Eingabestring AInputStr an den Regex-Fundstellen in die Stringliste APieces

Ruft intern *Exec* / *ExecNext* auf

Siehe alternativ die globale Funktion *SplitRegExpr*, die sich verwenden läßt, ohne explizit ein‘‘TRegExpr‘‘-Objekt zu erstellen.

Replace, ReplaceEx

```
function Replace (Const AInputStr : RegExprString;
  const AReplaceStr : RegExprString;
  AUseSubstitution : boolean= False)
 : RegExprString; overload;

function Replace (Const AInputStr : RegExprString;
  AReplaceFunc : TRegExprReplaceFunction)
 : RegExprString; overload;

function ReplaceEx (Const AInputStr : RegExprString;
  AReplaceFunc : TRegExprReplaceFunction):
  RegExprString;
```

Gibt die Zeichenfolge mit erneuten Vorkommen durch die Ersetzungszeichenfolge zurück.

Wenn das letzte Argument ("AUseSubstitution") wahr ist, wird "AReplaceStr" als Vorlage für Substitutionsmethoden verwendet.

```
Ausdruck: = &#39;((? I) block | var) \ s * (\ s * \ ([^] * \) \ s *) \ s *&#39;;␣
↪Ersetzen Sie (&#39;BLOCK (test1)&#39;, &#39;def &quot;$ 1&quot; -Wert &quot;$ 2&
↪quot;&#39;, True);
```

Liefert ‘‘def „BLOCK“ Wert "test1" ‘‘

```
Ersetzen (&#39;BLOCK (test1)&#39;, &#39;def &quot;$ 1&quot; Wert &quot;$ 2&quot;&#39;,
↪ False)
```

Liefert ‘‘ def "$ 1" Wert "$ 2" ‘‘

Ruft intern *Exec* / *ExecNext* auf

Die überladene Version und `ReplaceEx` arbeiten mit der Rückruffunktion, sodass Sie sehr komplexe Funktionen implementieren können.

Siehe auch die globale Funktion *ReplaceRegExpr*, die Sie ohne explizite `TRegExpr`-Objekterstellung verwenden können.

SubExprMatchCount

Die Anzahl der Unterausdrücke wurde im letzten *Exec* / *ExecNext*-Aufruf gefunden.

Wenn es keine Subexpressions gibt, jedoch ein vollständiger Ausdruck gefunden wurde (Exec * hat True zurückgegeben), dann `SubExprMatchCount = 0`, falls weder Unterausdrücke noch vollständige gefunden wurden (*Exec* / *ExecNext* lieferte False), dann `SubExprMatchCount = -1`.

Beachten Sie, dass einige Unterausdrücke. kann nicht gefunden werden und für einen solchen Unterausdruck. ‘‘ MathPos = MatchLen = -1‘‘ und ‘‘ Match = ” ‘‘.

```
Ausdruck: = &#39;(1) 2 (3)&#39; &#39;; Exec (&#39;123&#39;): SubExprMatchCount = 2,␣
↪Match [0] = &#39;123&#39;, [1] = &#39;1&#39;, [2] = &#39;3&#39; Exec (&#39;12&#39;
↪): SubExprMatchCount = 1, Match [0] = &#39;12 &#39;, [1] =&#39; 1 &#39;Exec (&#39;␣
↪23 &#39;): SubExprMatchCount = 2, Match [0] =&#39; 23 &#39;, [1] =&#39; &#39;, [2]␣
↪=&#39; 3 &#39;Exec (&#39; 2 &#39;): SubExprMatchCount = 0, Match [0] =&#39; 2 &#39;
↪Exec (&#39; 7 &#39;) - Rückgabe False: SubExprMatchCount = -1
```

MatchPos

pos von Eingang subexpr. `#Idx` wurde in der letzten ‘‘ Exec * ‘‘ Zeichenfolge getestet. Erster Unterausdruck habe `Idx=1`, zuletzt - ‘‘ MatchCount‘‘, ganze hat `Idx=0`.

Gibt `-1` zurück, wenn kein solcher Subexpr vorliegt. oder dieser subexpr. nicht in Eingabezeichenfolge gefunden.

MatchLen

len von entry subexpr. `#Idx` wird in der letzten ‘‘ Exec * ‘‘ Zeichenfolge getestet. Erster Unterausdruck habe `Idx=1`, letzte - MatchCount, ganze hat `Idx=0`.

Gibt -1 zurück, wenn kein solcher Unterausdruck vorhanden ist. oder dieser subexpr. nicht in Eingabezeichenfolge gefunden.

Spiel

Gibt ‘‘ ” ‘‘ zurück, wenn in der Eingabezeichenfolge kein solcher Unterausdruck oder dieser Unterausdruck gefunden wurde.

LastError

Gibt `ID` des letzten Fehlers zurück, `0`, wenn keine Fehler vorliegen (unbrauchbar, wenn die ’Fehler’-Methode eine Ausnahmebedingung auslöst) und den internen Status in `0` löschen (keine Fehler).

ErrorMsg

Gibt die `Error`-Nachricht für einen Fehler mit ‘‘ ID = AErrorID‘‘ zurück.

CompilerErrorPos

Gibt pos in re zurück, wo der Compiler gestoppt wurde.

Nützlich für die Fehlerdiagnose

SpaceChars

Enthält Zeichen, die als `\s` behandelt werden (anfänglich mit der globalen Konstante *RegExprSpaceChars* gefüllt)

WordChars

Enthält Zeichen, die als `\w` behandelt werden (anfänglich mit der globalen Konstante *RegExprWordChars* gefüllt)

LineSeparators

Zeilentrennzeichen (wie `\n` in Unix), anfänglich gefüllt mit *RegExprLineSeparators* (globale Konstante)

Siehe auch ‘Zeilengrenzen <regular_expressions.html#lineseparators> __

LinePairedSeparator

gepaartes Trennzeichen (wie `\r\n` in DOS und Windows).

muss genau zwei oder gar keine Zeichen enthalten und ist anfangs mit dem Inhalt der globalen Konstante RegExprLinePairedSeparator vorbefüllt

Siehe auch ‘Zeilengrenzen <regular_expressions.html#lineseparators> __

Wenn Sie beispielsweise ein Verhalten im Unix-Stil benötigen, weisen Sie ‘‘ LineSeparators: = # $ a‘‘ und ‘‘ LinePairedSeparator: = ” ‘‘ (leere Zeichenfolge) zu.

Wenn Sie als Linientrennzeichen nur *x0D x0A* ‘ aber nicht‘ *x0D* ‘ oder‘ *x0A* ‘ allein akzeptieren möchten, weisen Sie‘ *LineSeparators: = ” (leere Zeichenfolge) und* ‘ LinePairedSeparator: = # $ d # $ a‘‘.

Standardmäßig wird der gemischte Modus verwendet (definiert in RegExprLine [Paired] Separator [s] globale Konstanten):

```
LineSeparators: = # $ d # $ a; LinePairedSeparator: = # $ d # $ a
```

Das Verhalten dieses Modus wird ausführlich in den Zeilengrenzen beschrieben <regular_expressions.html#lineseparators> __.

InvertCase

Invertierung des Zeichenkastens. Definieren Sie es neu, wenn Sie ein anderes Verhalten wünschen.

Kompilieren

Kompiliert regulären Ausdruck.

Nützlich zum Beispiel für GUI-Editoren für reguläre Ausdrücke, um reguläre Ausdrücke zu überprüfen, ohne ihn zu verwenden.

Dump

Zeigen Sie `P-Code` (kompilierter regulärer Ausdruck) als vom Menschen lesbare Zeichenfolge.

5.2.2 Globale Konstanten

EscChar

Escape-char, standardmäßig \.

RegExprModifierI

Modifier i <regular_expressions.html#i> _ Standardwert

RegExprModifierR

Modifier r <regular_expressions.html#r> _ Standardwert

RegExprModifierS

‘Modifier s <regular_expressions.html#s> _ Standardwert

RegExprModifierG

‘Modifikator g <regular_expressions.html#g> _ Standardwert

RegExprModifierM

Modifier m <regular_expressions.html#m> _ Standardwert

RegExprModifierX

‘Modifikator x <regular_expressions.html#x> _ Standardwert

RegExprSpaceChars

Standardwert für die *SpaceChars*-Eigenschaft

RegExprWordChars

Standardwert für die Eigenschaft *WordChars*

RegExprLineSeparators

Standardwert für die *LineSeparators*-Eigenschaft

RegExprLinePairedSeparator

Standardwert für die *LinePairedSeparator*-Eigenschaft

RegExprInvertCaseFunction

Standard für die Eigenschaft *InvertCase*

5.2.3 Globale Funktionen

Alle diese Funktionen stehen als Methoden von `TRegExpr` zur Verfügung, aber mit globalen Funktionen müssen Sie keine `TReExpr`-Instanz erstellen, sodass Ihr Code einfacher wäre, wenn Sie nur eine Funktion benötigen.

ExecRegExpr

true, wenn die Zeichenfolge mit dem regulären Ausdruck übereinstimmt. So wie *Exec* in `TRegExpr`.

SplitRegExpr

Teilt den String mit regulären Ausdrücken. Siehe auch *Split*, wenn Sie die `TRegExpr`-Instanz explizit erstellen möchten.

ReplaceRegExpr

```
Funktion ReplaceRegExpr (const ARegExpr, AInputStr, AReplaceStr: RegExprString;␣
↪AUseSubstitution: boolean = False): RegExprString; Überlast; Typ␣
↪TRegexReplaceOption = (rroModifierI, rroModifierR, rroModifierS, rroModifierG,␣
↪rroModifierM, rroModifierX, rroUseSubstitution, rroUseOsLineEnd);␣
↪TRegexReplaceOptions = Set von TRegexReplaceOption; Funktion ReplaceRegExpr (const␣
↪ARegExpr, AInputStr, AReplaceStr: RegExprString; Optionen: TRegexReplaceOptions):␣
↪RegExprString; Überlast;
```

Gibt den String mit regulären Ausdrücken zurück, die durch `AReplaceStr` ersetzt werden. Siehe auch *Replace*, wenn Sie es vorziehen, die TRegExpr-Instanz explizit zu erstellen.

Wenn das letzte Argument (`AUseSubstitution`) wahr ist, wird ‘‘ AReplaceStr‘‘ als Vorlage für ‘‘ Substitutionsmethoden‘‘ verwendet:

```
ReplaceRegExpr (&#39;((?)) Block | var) \ s * (\ s * \ ([^] * \) \ s *) \ s *&#39;, &
↪#39;BLOCK (test1)&#39;, &#39;def &quot;$ 1&quot; value &quot; $ 2 &quot;, wahr)
```

Gibt den ‘‘ def ’BLOCK’-Wert’ test1 ’‘zurück

Aber dieses hier (Anmerkung: Es gibt kein letztes Argument):

```
ReplaceRegExpr (&#39;((?)) Block | var) \ s * (\ s * \ ([^] * \) \ s *) \ s *&#39;, &
↪#39;BLOCK (test1)&#39;, &#39;def &quot;$ 1&quot; value &quot; $ 2 &quot;&#39;)
```

Liefert ‘‘ def "$ 1" Wert "$ 2" ‘‘

Version mit Optionen

Mit `Options` steuert man das Verhalten von `\n` (mit der Option `rroUseOsLineEnd` wird jedes `\n` unter Windows durch `\n\r` ersetzt, unter Linux bleibt es ein `\n`). Und so weiter.

```
Typ TRegexReplaceOption = (rroModifierI, rroModifierR, rroModifierS, rroModifierG,␣
↪rroModifierM, rroModifierX, rroUseSubstitution, rroUseOsLineEnd);
```

QuoteRegExprMetaChars

Ersetzen Sie alle Metachars durch ihre sichere Darstellung, zum Beispiel ‘‘ abc’cd. (‘‘ Konvertiert in `abc\'cd\.\(.`)

Diese Funktion ist nützlich für die erneute Generierung von Benutzereingaben

RegExprSubExpressions

Erstellt eine Liste der Unterausdrücke, die in `ARegExpr` gefunden werden

In `ASubExps` repräsentiert jedes Element einen ersten Ausdruck vom ersten bis zum letzten Format im Format:

String - Unterausdruck (ohne ’()’)

Low word of Object - Startposition in ARegExpr, einschließlich ’(’ falls vorhanden! (erste Position ist 1)

hohes Wort der Objektlänge, einschließlich "(" und "Endung") "falls vorhanden!

`AExtendedSyntax` - muss `True` sein, wenn der Schalter `/x` mit dem regulären Ausdruck verwendet werden soll.

Nützlich für GUI-Editoren von re etc (ein Beispiel für die Verwendung finden Sie in ‘REStudio-Main.pas) <https://github.com/andgineer/TRegExpr/blob/74ab342b639fc51941a4eea9c7aa53dcdf783592/restudio/REStudioMain.pas#L474> _)

Ergebniscode	Bedeutung
0	Erfolg. Es wurden keine unausgeglichenen Klammern gefunden
-1	Es gibt nicht genug schließende Klammern `)`
-(n+1)	An Position n wurde das Öffnen von `[` ohne entsprechende Schließung ‘‘] ‘‘ gefunden
n	In Position n wurde das Schließen der Klammer `)` ohne entsprechende Öffnung `()` gefunden

Wenn `Result <> 0` `ASubExprs` kann leere oder illegale Elemente enthalten

5.2.4 ERegExpr

```
ERegExpr = Klasse (Ausnahme) public ErrorCode: integer; // Fehlercode.␣
↪Kompilierungsfehlercodes liegen vor 1000 CompilerErrorPos: integer; // Position in␣
↪re, an der der Kompilierungsfehler aufgetreten ist end;
```

5.2.5 Unicode

UniCode verlangsamt die Leistung. Verwenden Sie ihn daher nur, wenn Sie wirklich Unicode-Unterstützung benötigen.

Um Unicode zu verwenden, kommentieren Sie `{$DEFINE UniCode}` *in* regexpr.pas <https://github.com/andgineer/TRegExpr/blob/29ec3367f8309ba2ecde7d68d5f14a514de94511/src/RegExpr.pas#L86>‘__ (entferne `off`).

Danach werden alle Zeichenfolgen als WideString behandelt.

	Englisch		Deutsch		Français	Español

5.3 FAQ

5.3.1 Ich habe einen schrecklichen Fehler gefunden: TRegExpr löst Zugriffsverletzung aus!

Antworten

Sie müssen das Objekt vor der Verwendung erstellen. Nachdem Sie also etwas erklärt haben:

```
r: TRegExpr
```

Vergessen Sie nicht, die Objektinstanz zu erstellen:

```
r: = TRegExpr.Create.
```

5.3.2 Reguläre Ausdrücke mit (? = . . .) funktionieren nicht

Look Ahead ist in TRegExpr nicht implementiert. In vielen Fällen können Sie sie jedoch durch einfache Unteraus-drücke leicht ersetzen.

5.3.3 Unterstützt es Unicode?

Antworten

‘Wie verwende ich Unicode? <tregexpr.html#unicode> __

5.3.4 Warum gibt TRegExpr mehr als eine Zeile zurück?

Zum Beispiel gibt re ‘‘ ‘‘ das erste ‘‘ zurück <font‘‘, then the rest of the file including last `</html>` `<font .\*>`.

Antworten

Für Rückwärtskompatibilität Modifier / s ist standardmäßig‘ *Ein*‘.

Schalten Sie es aus und . passt zu allen anderen als ’Trennzeichen’ <regu-lar_expressions.html#syntax_line_separators> __ - genau wie Sie es wünschen.

Übrigens empfehle ich ‘‘] *)> ‘‘, in ‘‘ Match [1] ‘‘ wird die URL.

5.3.5 Warum gibt TRegExpr mehr als ich erwarte?

Zum Beispiel: ‘‘ (. +) ‘‘ auf String ‘‘ angewendet ein b ‘‘ gibt ‘‘ a zurück b‘‘ aber nicht `a` wie ich erwartet hatte.

Antworten

Standardmäßig arbeiten alle Operatoren im `gierigen` Modus, so dass sie so weit wie möglich zusammenpassen.

Wenn Sie den `nicht-gierigen` Modus wollen, können Sie ‘‘ nicht-gierige‘‘-Operatoren wie `+?` Usw. verwenden, oder alle Operatoren mit Hilfe des Modifikators ‘‘ `g (verwenden Sie die entsprechenden TRegExpr-Eigenschaften oder den Operator‘ *? (- g)* ‘in re).

5.3.6 Wie kann man mit TRegExpr Quellen wie HTML analysieren?

Antworten

Sorry Leute, aber es ist fast unmöglich!

Natürlich können Sie TRegExpr problemlos zum Extrahieren einiger Informationen aus HTML verwenden, wie in meinen Beispielen gezeigt. Wenn Sie jedoch eine genaue Analyse wünschen, müssen Sie einen echten Parser verwenden, nicht re

Die vollständige Erklärung finden Sie beispielsweise in Tom Christiansen und Nathan Torkington, ‘‘ Perl Cookbook‘‘.

Kurz gesagt, es gibt viele Strukturen, die mit echtem Parser einfach geparst werden können, von re aber gar nicht, und realer Parser ist beim Parsing viel schneller, da re nicht nur den Eingabestrom scannt, sondern eine Optimierungssuche ausführt, die dauern kann viel Zeit.

5.3.7 Gibt es eine Möglichkeit, mehrere Übereinstimmungen eines Musters auf TRegExpr abzurufen?

Antworten

Sie können Übereinstimmungen mit der ExecNext-Methode wiederholen.

Wenn Sie ein Beispiel wünschen, werfen Sie einen Blick auf die Implementierung der *TRegExpr.Replace‘*-Methode oder auf die Beispiele für‘ HyperLinksDecorator <demos.html> _

5.3.8 Ich überprüfe die Benutzereingaben. Warum gibt TRegExpr für falsche Eingabezeichenfolgen "True" zurück?

Antworten

In vielen Fällen vergessen TRegExpr-Benutzer, dass reguläre Ausdrücke für ** Suche ** in der Eingabezeichenfolge sind.

Wenn Sie beispielsweise ‘‘ d {4,4} ‘‘ Ausdruck verwenden, erhalten Sie Erfolg bei falschen Benutzereingaben wie `12345` oder ‘‘ any letters 1234‘‘.

Sie müssen vom Anfang der Zeile bis zum Ende der Zeile überprüfen, um sicherzustellen, dass nichts anderes vorhanden ist: ‘‘ ^ d {4,4} $ ‘‘.

5.3.9 Warum funktionieren nichtgierige Iteratoren manchmal wie im gierigen Modus?

Zum Beispiel entspricht das re `a+?,b+?`, Das auf den String `aaa,bbb` angewendet wird, `aaa,b`, sollte aber nicht mit `a,b` wegen Nicht-Gier passen des ersten Iterators?

Antworten

Dies liegt an TRegExprs Arbeitsweise. Tatsächlich arbeiten viele andere re-Engines genau gleich: Sie führen nur eine "einfache" Suchoptimierung durch und versuchen nicht, die beste Optimierung zu erreichen.

In manchen Fällen ist es schlecht, aber im Allgemeinen ist es aus Gründen der Leistung und Vorhersagbarkeit eher ein Vorteil als eine Einschränkung.

Die Hauptregel - versuchen Sie zuerst, vom aktuellen Ort aus zu passen. Nur wenn dies völlig unmöglich ist, bewegen Sie sich um ein Zeichen vorwärts und versuchen Sie es erneut von der nächsten Position im Text.

Wenn Sie also `a,b+?` Wird es `a,b` passen. Im Falle von `a+?,b+?` Wird es jetzt nicht empfohlen (wir fügen einen nicht-gierigen Modifikator hinzu), aber es ist immer noch möglich, mehr als einen `a` zu finden, also macht TRegExpr dies.

TRegExpr wie Perl oder Unix versucht nicht, sich vorwärts zu bewegen und zu prüfen - würde es "besser" passen. Fisrt von allem, nur weil es keine Möglichkeit gibt zu sagen, dass es mehr oder weniger gut passt.

5.3.10 Wie kann ich TRegExpr mit Borland C ++ Builder verwenden?

Ich habe ein Problem, da keine Header-Datei (`.h` oder `.hpp`) verfügbar ist.

Antworten

- Fügen Sie `RegExpr.pas` zu `bcb` hinzu.
- Projekt kompilieren Dadurch wird die Header-Datei `RegExpr.hpp` generiert.
- Jetzt können Sie Code schreiben, der die `RegExpr`-Einheit verwendet.
- Vergessen Sie nicht, ‘‘ #include “RegExpr.hpp” ‘‘ bei Bedarf hinzuzufügen.
- Vergessen Sie nicht, alle \ in regulären Ausdrücken durch \\ oder neu definierte ‘EscChar zu ersetzen <tregexpr.html#escchar> __ const.

5.3.11 Warum arbeiten viele (einschließlich TRegExpr-Hilfe und -Demo) in Borland C ++ Builder falsch?

Antworten

Der Hinweis ist in der vorherigen Frage;) Das Symbol \ hat eine besondere Bedeutung in ‘‘ C ++ , `also müssen Sie es ``entkommen` (wie in der vorherigen Antwort beschrieben). Wenn Sie nicht gerne `\\w + \\\\ w + \\. \\ w + `` mögen, können Sie die Konstante ``EscChar` (in `RegExpr.pas`) neu definieren. Zum Beispiel ‘‘ EscChar = "/" ‘‘. Dann können Sie ‘‘ / w + / w + /. / W + ‘‘ schreiben, sieht ungewöhnlich aus, ist aber lesbarer.

	English		Deutsch		Français	Español

5.4 Demos

Demo-Code für ‘TRegExpr <index.html> __

5.4.1 Einführung

Wenn Sie sich mit dem regulären Ausdruck nicht auskennen, werfen Sie einen Blick auf die Resyntax <regular_expressions.html> __.

TRegExpr-Schnittstelle, beschrieben in TRegExpr-Schnittstelle.

5.4.2 Text2HTML

‘Text2HTML-Quellen <https://github.com/andgineer/TRegExpr/tree/master/examples/Text2HTML> _

Veröffentlichen Sie Nur-Text als HTML

Verwendet die Einheit HyperLinksDecorator, das auf TRegExpr basiert.

Diese Einheit enthält Funktionen zum Verzieren von Hyperlinks.

Ersetzt beispielsweise ‘‘ www.masterAndrey.com‘‘ durch ‘‘ www.masterAndrey.com ‘‘ oder `filbert@yandex.ru` durch `<a href="mailto:filbert@yandex.ru">filbert@yandex.ru</a>`.

```
Funktion DecorateURLs (const AText: string; AFlags: TDecorateURLsFlagSet = [durlAddr,␣
↪durlPath]): string; type TDecorateURLsFlags = (durlProto, durlAddr, durlPort,␣
↪durlPath, durlBMark, durlParam); TDecorateURLsFlagSet = Satz von TDecorateURLsFlags;
↪ Funktion DecorateEMails (const AText: string): string;
```

Wert	Bedeutung
durlProto	Protokoll (wie ‘‘ ftp: // *oder http: //* ‘)
durlAddr	TCP-Adresse oder Domänenname (wie `masterAndrey.com`)
durlPort	Portnummer, falls angegeben (wie `: 8080`)
DurlPath	Pfad zum Dokument (wie `index.html`)
durlBMark	Buchmarke (wie `#mark`)
DurlParam	URL-Parameter (wie `? ID = 2 & User = 13`)

Gibt den eingegebenen Text "AText" mit verzierten Hyperlinks zurück.

"AFlags" beschreibt, welche Teile des Hyperlinks in den sichtbaren Teil des Links eingefügt werden müssen.

Wenn zum Beispiel AFlags ‘‘ [durlAddr] ‘‘ ist, wird der Hyperlink `www.masterAndrey.com/contact.htm` als ``<a href="www.masterAndrey.com/contacts.htm">www.masterAndrey.com</a>`` verziert.

5.4.3 ‘TRegExprRoutines <https://github.com/andgineer/TRegExpr/tree/master/examples/TRegE

—

Sehr einfache Beispiele, siehe Kommentare im Gerät

5.4.4 ‘TRegExprClass <https://github.com/andgineer/TRegExpr/tree/master/examples/TRegExp

—

Etwas komplexere Beispiele, siehe Kommentare innerhalb der Einheit

KAPITEL 6

Übersetzungen

Die Dokumentation ist in Englisch verfügbar.

Es gibt auch alte Übersetzungen in Deutsch, Bulgarisch, Französisch und Spanisch. Wenn Sie bei der Aktualisierung dieser alten Übersetzungen mithelfen möchten, kontaktieren Sie mich bitte <https://github.com/andgineer>‘_.

Neue Übersetzungen basieren auf GetText <https://en.wikipedia.org/wiki/Gettext>‘_ und kann mit‘ transifex.com bearbeitet werden <https://www.transifex.com/masterAndrey/tregexpr/dashboard/>‘_.

Sie sind bereits maschinell übersetzt und müssen nur korrigiert werden. Möglicherweise werden auch alte Übersetzungen kopiert.

KAPITEL 7

Dankbarkeit

Viele Funktionen wurden vorgeschlagen und viele Fehler wurden von TRegExpr-Mitarbeitern begründet (und sogar behoben).

Ich kann hier nicht alle aufführen, aber ich freue mich über alle Fehlerberichte, Vorschläge und Fragen, die ich von Ihnen bekomme.

- Guido Muehlwitz - hässlicher Fehler in der Verarbeitung großer Seiten gefunden und behoben
- Stephan Klimek - Testen in CPPB und Vorschlagen / Implementieren vieler Funktionen
- Steve Mudford - Offset-Parameter implementiert
- Martin Baur (www.mindpower.com)) - deutsche Übersetzung, nützliche Vorschläge
- Yury Finkel - Unterstützung für UniCode implementiert, einige Fehler gefunden und behoben
- Ralf Junker - Einige Funktionen implementiert, viele Optimierungsvorschläge
- Simeon Lilov - Bulgarische Übersetzung
- Filip Jirsk und Matthew Winter - Hilfe bei der Implementierung des nichtgierigen Modus
- Kit Eason viele Beispiele für die Einführung
- Jürgen Schroth - Käferjagd und nützliche Vorschläge
- Martin Ledoux - französische Übersetzung
- Diego Calp, Argentinien - spanische Übersetzung