TRegExpr Documentation

Release 0.952

Andrey Sorokin

May 16, 2019

Contents

	English		Deutsch		Français	Español

TRegExpr library implements regular expressions.

Regular expressions are easy to use and powerful tool for sophisticated search and substitution and for template based text check.

It is especially useful for user input validation in input forms - to validate e-mail addresses and so on.

Also you can extract phone numbers, ZIP-codes etc from web-pages or documents, search for complex patterns in log files and all you can imagine. Rules (templates) can be changed without your program recompilation.

TRegExpr is implemented in pure Pascal. It's included into Lazarus (Free Pascal) project. But also it exists as separate library and can be compiled by Delphi 2-7, Borland C++ Builder 3-6.

CHAPTER 1

Reviews

How good the library was met.

CHAPTER 2

Quick start

To use the library just add the sources to you project and use the class TRegExpr.

In the FAQ you can learn from others users problems.

Ready to run Windows application REStudio will help you learn and debug regular expressions.

CHAPTER 3

Feedback

If you see any problems, please create the bug.

CHAPTER 4

Source code

Pure Object Pascal.

- Original version
- FreePascal fork (GitHub mirror of the SubVersion)

CHAPTER 5

Documentation

	English		Deutsch		Français	Español

5.1 Regular expressions (RegEx)

5.1.1 Introduction

Regular expressions are a handy way to specify patterns of text.

With regular expressions you can validate user input, search for some patterns like emails of phone numbers on web pages or in some documents and so on.

Below is complete regular expressions cheat sheet just on one page.

5.1.2 Characters

Simple matches

Any single character matches itself.

A series of characters matches that series of characters in the input string.

RegEx	Matches
`foobar`	`foobar`

Non-Printable Characters (escape-codes)

To represent non-printable character in regular expression you use `\x..`:

RegEx	Matches
\xnn	character with hex code nn
\x{nnnn}	character with hex code nnnn (one byte for plain text and two bytes for Unicode)
foo\x20bar	foo bar (note space in the middle)

There are a number of predefined escape-codes for non-printable characters, just like in C language:

RegEx	Matches
\t	tab (HT/TAB), same as \x09
\n	newline (NL), same as \x0a
\r	car.return (CR), same as \x0d
\f	form feed (FF), same as \x0c
\a	alarm (BEL), same as \x07
\e	escape (ESC), same as \x1b
\tfoobar	foobar preceded by TAB

Escaping

If you want to use character \ by itself, not as part of escape-code, just prefix it with \, like that: \\.

In fact you can prefix (or escape) with \ any character that has special meaning in regular expressions.

RegEx	Matches
\^FooBarPtr	^FooBarPtr this is ^ and not *start of line*
\[a\]	[a] this is not *character class*

5.1.3 Character Classes

User Character Classes

Character class is a list of characters inside []. The class matches any **one** character listed in this class.

RegEx	Matches
foob[aeiou]r	foobar, foober etc but not foobbr, foobcr etc

You can invert the class - if the first character after the [is ^, then the class matches any character **but** characters listed in the class.

RegEx	Matches
foob[^aeiou]r	foobbr, foobcr etc but not foobar, foober etc

Within a list, the - character is used to specify a range, so that a-z represents all characters between a and z, inclusive.

If you want - itself to be a member of a class, put it at the start or end of the list, or *escape* it with a backslash.

If you want] or [you may place it at the start of list or escape it with a backslash.

RegEx	Matches
`[-az]`	`a`, `z` and `-`
`[az-]`	`a`, `z` and `-`
`[a\-z]`	`a`, `z` and `-`
`[a-z]`	characters from `a` to `z`
`[\n-\x0D]`	characters from `#10` to `#13`

Predefined Character Classes

There are a number of predefined character classes that keeps regular expressions more compact.

RegEx	Matches
`\w`	an alphanumeric character (including _)
`\W`	a nonalphanumeric
`\d`	a numeric character (same as `[0123456789]`)
`\D`	a non-numeric
`\s`	any space (same as `[ \t\n\r\f]`)
`\S`	a non space

You may use `\w`, `\d` and `\s` within *user character classes*.

RegEx	Matches
`foob\dr`	`foob1r`, `foob6r` and so on but not `foobar`, `foobbr` and so on
`foob[\w\s]r`	`foobar`, `foob r`, `foobbr` and so on but not `foob1r`, `foob=r` and so on

Note: TRegExpr

Properties SpaceChars and WordChars define character classes `\w`, `\W`, `\s`, `\S`.

So you can redefine this classes.

5.1.4 Boundaries

Line Boundaries

RegEx	Matches
`^`	start of line
`$`	end of line
`\A`	start of text
`\Z`	end of text
`.`	any character in line
`^foobar`	`foobar` only if it's at the beginning of line
`foobar$`	`foobar` only if it's at the end of line
`^foobar$`	`foobar` only if it's the only string in line
`foob.r`	`foobar`, `foobbr`, `foob1r` and so on

`^` metacharacter by default match the beginning of the input string. `$` - the end.

You may, however, wish to treat a string as a multi-line text, so ^ will match after any line separator within the string, and $ will match before any line separator. You can do this by switching *modifier /m*.

Note that there is no empty line within the sequence \x0D\x0A.

Note: TRegExpr

If you are using Unicode version, then ^/$ also matches \x2028, \x2029, \x0B, \x0C or \x85.

The \A and \Z are just like ^ and $, except that they won't match multiple times when the *modifier /m* is used.

The . metacharacter by default matches any character, but if you switch Off the *modifier /s*, then . won't match line separators inside the string.

Note that ^.*$ does not match a string between \x0D\x0A, because this is unbreakable line separator. But it matches the empty string within the sequence \x0A\x0D because this is just wrong order to be treated as line separator.

Note: TRegExpr

Multiline processing can be tuned with of properties LineSeparators and LinePairedSeparator.

So you can use Unix style separators \n or DOS/Windows style \r\n or mix them together (as in described above default behaviour).

If you prefer mathematically correct description you can find it on www.unicode.org.

Word Boundaries

RegEx	Matches
\b	a word boundary
\B	a non-word boundary

A word boundary \b is a spot between two characters that has a \w on one side of it and a \W on the other side of it (in either order).

5.1.5 Quantification

Quantifier

Any item of a regular expression may be followed by quantifier. Quantifier specifies number of repetition of the item.

RegEx	Matches
{n}	exactly n times
{n,}	at least n times
{n,m}	at least n but not more than m times
*	zero or more, similar to {0,}
+	one or more, similar to {1,}
?	zero or one, similar to {0,1}

So, digits in curly brackets {n,m}, specify the minimum number of times to match n and the maximum m.

The {n} is equivalent to {n,n} and matches exactly n times.

The `{n,}` matches `n` or more times.

There is no limit to the size of `n` or `m`.

If a curly bracket occurs in any other context, it is treated as a regular character.

RegEx	Matches
`foob.*r`	`foobar`, `foobalkjdflkj9r` and `foobr`
`foob.+r`	`foobar`, `foobalkjdflkj9r` but not `foobr`
`foob.?r`	`foobar`, `foobbr` and `foobr` but not `foobalkj9r`
`fooba{2}r`	`foobaar`
`fooba{2,}r`	`foobaar'`, `foobaaar`, `foobaaaar` etc.
`fooba{2,3}r`	`foobaar`, or `foobaaar` but not `foobaaaar`
`(foobar){8,10}`	`8`, `9` or `10` instances of the `foobar` (`()` is *Subexpression*)

Greediness

Quantifiers in `greedy` mode takes as many as possible, in `non-greedy` mode - as few as possible.

By default all quantifiers are `greedy`. Use `?` to make any quantifier `non-greedy`.

For string `abbbbc`:

RegEx	Matches
`b+`	`bbbb`
`b+?`	`b`
`b*?`	empty string
`b{2,3}?`	`bb`
`b{2,3}`	`bbb`

You can switch all quantifiers into `non-greedy` mode (*modifier /g*, below we use *in-line modifier change*).

RegEx	Matches
`(?-g)b+`	`b`

5.1.6 The choice

Expressions in the choice are separated by `|`.

So `fee|fie|foe` will match any of `fee`, `fie`, or `foe` in the target string (as would `f(e|i|o)e`).

The first expression includes everything from the last pattern delimiter (`(`, `[`, or the beginning of the pattern) up to the first `|`, and the last expression contains everything from the last `|` to the next pattern delimiter.

Sounds a little complicated, so it's common practice to include the choice in parentheses, to minimize confusion about where it starts and ends.

Expressions in the choice are tried from left to right, so the first expression that matches, is the one that is chosen.

For example, regular expression `foo|foot` in string `barefoot` will match `foo`. Just a first expression that matches.

Also remember that `|` is interpreted as a literal within square brackets, so if you write `[fee|fie|foe]` you're really only matching `[feio|]`.

RegEx	Matches
`foo(bar|foo)`	`foobar` or `foofoo`

5.1.7 Subexpressions

The brackets `( ... )` may also be used to define regular expression subexpressions.

Note: TRegExpr

Subexpression positions, lengths and actual values will be in MatchPos, MatchLen and Match.

You can substitute them with Substitute.

Subexpressions are numbered from left to right by their opening parenthesis (including nested subexpressions).

First subexpression has number `1`. Whole regular expression has number `0`.

For example for input string `foobar` regular expression `(foo(bar))` will find:

subexpression	value
0	`foobar`
1	`foobar`
2	`bar`

5.1.8 Backreferences

Metacharacters `\1` through `\9` are interpreted as backreferences. `\n` matches previously matched subexpression `n`.

RegEx	Matches
`(.)\1+`	`aaaa` and `cc`
`(.+)\1+`	also `abab` and `123123`

`(['"]?)(\d+)\1` matchs `"13"` (in double quotes), or `'4'` (in single quotes) or `77` (without quotes) etc

5.1.9 Modifiers

Modifiers are for changing behaviour of regular expressions.

You can set modifiers globally in your system or change inside the the regular expression using the *(?imsxr-imsxr)*.

Note: TRegExpr

To change modifiers use ModifierStr or appropriate `TRegExpr` properties Modifier*.

The default values are defined in global variables. For example global variable `RegExprModifierX` defines default value for `ModifierX` property.

---

i, case-insensitive

Case-insensitive. Use installed in you system locale settings, see also InvertCase.

m, multi-line strings

Treat string as multiple lines. So `^` and `$` matches the start or end of any line anywhere within the string.

See also *Line Boundaries*.

s, single line strings

Treat string as single line. So `.` matches any character whatsoever, even a line separators.

See also *Line Boundaries*, which it normally would not match.

g, greediness

Note: TRegExpr only modifier.

Switching it `Off` you'll switch *quantifiers* into *non-greedy* mode.

So, if modifier `/g` is `Off` then `+` works as `+?`, `*` as `*?` and so on.

By default this modifier is `On`.

x, eXtended syntax

Allows to comment regular expression and break them up into multiple lines.

If the modifier is `On` we ignore all whitespaces that is neither backslashed nor within a character class.

And the `#` character separates comments.

Notice that you can use empty lines to format regular expression for better readability:

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

This also means that if you want real whitespace or `#` characters in the pattern (outside a character class, where they are unaffected by `/x`), you'll either have to escape them or encode them using octal or hex escapes.

r, Russian ranges

Note: TRegExpr only modifier.

In Russian ASCII table characters **ё** are placed separately from others.

Big and small Russian characters are in separated ranges, this is the same as with English characters but nevertheless I wanted some short form.

With this modifier instead of `[а-я]` you can write `[а-ё]` if you need all Russian characters.

When the modifier is `On`:

RegEx	Matches
–	chars from to and
–	chars from to and
–	all russian symbols

The modifier is set *On* by default.

5.1.10 Extensions

(?=<lookahead>)

`Look ahead` assertion. It checks input for the regular expression `<look-ahead>`, but do not capture it.

Note: TRegExpr

Look-ahead is not implemented in TRegExpr.

In many cases you can replace `look ahead` with *Sub-expression* and just ignore what will be captured in this subexpression.

For example `(blah)(?=foobar)(blah)` is the same as `(blah)(foobar)(blah)`. But in the latter version you have to exclude the middle sub-expression manually - use `Match[1] + Match[3]` and ignore `Match[2]`.

This is just not so convenient as in the former version where you can use whole `Match[0]` because captured by `look ahead` part would not be included in the regular expression match.

(?imsgxr-imsgxr)

You may use it inside regular expression for modifying modifiers by the fly.

This can be especially handy because it has local scope in a regular expression. It affects only that part of regular expression that follows `(?imsgxr-imsgxr)` operator.

And if it's inside subexpression it will affect only this subexpression - specifically the part of the subexpression that follows after the operator. So in `((?i)Saint)-Petersburg` it affects only subexpression `((?i)Saint)` so it will match `saint-Petersburg` but not `saint-petersburg`.

RegEx	Matches
`(?i)Saint-Petersburg`	`Saint-petersburg` and `Saint-Petersburg`
`(?i)Saint-(?-i)Petersburg`	`Saint-Petersburg` but not `Saint-petersburg`
`(?i)(Saint-)?Petersburg`	`Saint-petersburg` and `saint-petersburg`
`((?i)Saint-)?Petersburg`	`saint-Petersburg`, but not `saint-petersburg`

(?#text)

A comment, the text is ignored.

Note that the comment is closed by the nearest `)`, so there is no way to put a literal `)` in the comment.

5.1.11 Afterword

In this ancient blog post from previous century I illustrate some usages of regular expressions.

	English		Deutsch		Français	Español

5.2 TRegExpr

Implements regular expressions in pure pascal. Is compatible with Free Pascal, Delphi 2-7, Borland C++ Builder 3-6.

To use it just copy source code into your project.

The library had already included into Lazarus (Free Pascal) project so you do not need to copy anything if you use Lazarus.

5.2.1 TRegExpr class

VersionMajor, VersionMinor

Return major and minor version, for example, for `version 0.944`

```
VersionMajor = 0
VersionMinor = 944
```

Expression

Regular expression.

For optimization regular expression is automatically compiled into `P-code`. Human-readable form of the `P-code` returns by *Dump*.

In case of any errors in compilation, `Error` method is called (by default `Error` raises exception *ERegExpr*)

ModifierStr

Set or get values of regular expression modifiers.

Format of the string is similar as in (?ismx-ismx). For example `ModifierStr := 'i-x'` will switch on modifier /i, switch off /x and leave unchanged others.

If you try to set unsupported modifier, `Error` will be called.

ModifierI

Modifier /i, "case-insensitive" <regular_expressions.html#i>, initialized with *RegExprModifierI* value.

ModifierR

Modifier /r, "Russian range extension", initialized with *RegExprModifierR* value.

ModifierS

Modifier /s, "single line strings", initialized with *RegExprModifierS* value.

ModifierG

Modifier /g, "greediness", initialized with *RegExprModifierG* value.

ModifierM

Modifier /m, "multi-line strings", initialized with *RegExprModifierM* value.

ModifierX

Modifier /x, "eXtended syntax", initialized with *RegExprModifierX* value.

Exec

Match the regular expression against `AInputString`.

Available overloaded `Exec` version without `AInputString` - it uses `AInputString` from previous call.

See also global function *ExecRegExpr* that you can use without explicit `TRegExpr` object creation.

ExecNext

Find next match.

Without parameter works the same as

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

Raises exception if used without preceeding successful call to *Exec*, *ExecPos* or *ExecNext*.

So you always must use something like

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

ExecPos

Finds match for `InputString` starting from `AOffset` position

```
AOffset = 1 // first char of InputString
```

InputString

Returns current input string (from last *Exec* call or last assign to this property).

Any assignment to this property clears *Match*, *MatchPos* and *MatchLen*.

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`.

To place into template characters `$` or `\`, use prefix `\`, like `\\` or `\$`.

symbol	description
`$&`	whole regular expression match
`$0`	whole regular expression match
`$n`	regular subexpression `n` match
`\n`	in Windows replaced with `\r\n`
`\l`	lowcase one next char
`\L`	lowercase all chars after that
`\u`	uppcase one next char
`\U`	uppercase all chars after that

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

If you want to place raw digit after '$n' you must delimit `n` with curly braces `{}`.

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

Split

Split AInputStr into APieces by r.e. occurencies

Internally calls *Exec* / *ExecNext*

See also global function *SplitRegExpr* that you can use without explicit `TRegExpr` object creation.

Replace, ReplaceEx

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

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

(continues on next page)

(continued from previous page)

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

Returns the string with r.e. occurencies replaced by the replace string.

If last argument (`AUseSubstitution`) is true, then `AReplaceStr` will be used as template for Substitution methods.

```
Expression := '((?i)block|var)\s*(\s*\([^ ]*\)\s*)\s*';
Replace ('BLOCK( test1)', 'def "$1" value "$2"', True);
```

Returns `def "BLOCK" value "test1"`

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

Returns `def "$1" value "$2"`

Internally calls *Exec* / *ExecNext*

Overloaded version and `ReplaceEx` operate with call-back function, so you can implement really complex functionality.

See also global function *ReplaceRegExpr* that you can use without explicit `TRegExpr` object creation.

SubExprMatchCount

Number of subexpressions has been found in last *Exec* / *ExecNext* call.

If there are no subexpr. but whole expr was found (Exec* returned True), then `SubExprMatchCount=0`, if no subexpressions nor whole r.e. found (*Exec* / *ExecNext* returned false) then `SubExprMatchCount=-1`.

Note, that some subexpr. may be not found and for such subexpr. `MathPos=MatchLen=-1` and `Match=''`.

```
Expression := '(1)?2(3)?';
Exec ('123'): SubExprMatchCount=2, Match[0]='123', [1]='1', [2]='3'

Exec ('12'): SubExprMatchCount=1, Match[0]='12', [1]='1'

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

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

Exec ('7') - return False: SubExprMatchCount=-1
```

MatchPos

pos of entrance subexpr. `#Idx` into tested in last `Exec*` string. First subexpr. have `Idx=1`, last - `MatchCount`, whole r.e. have `Idx=0`.

Returns `-1` if in r.e. no such subexpr. or this subexpr. not found in input string.

MatchLen

len of entrance subexpr. `#Idx` r.e. into tested in last `Exec*` string. First subexpr. have `Idx=1`, last - MatchCount, whole r.e. have `Idx=0`.

Returns -1 if in r.e. no such subexpr. or this subexpr. not found in input string.

Match

Returns '' if in r.e. no such subexpression or this subexpression was not found in the input string.

LastError

Returns `ID` of last error, `0` if no errors (unusable if `Error` method raises exception) and clear internal status into `0` (no errors).

ErrorMsg

Returns `Error` message for error with `ID = AErrorID`.

CompilerErrorPos

Returns pos in r.e. there compiler stopped.

Useful for error diagnostics

SpaceChars

Contains chars, treated as `\s` (initially filled with *RegExprSpaceChars* global constant)

WordChars

Contains chars, treated as `\w` (initially filled with *RegExprWordChars* global constant)

LineSeparators

line separators (like `\n` in Unix), initially filled with *RegExprLineSeparators* global constant)

see also Line Boundaries

LinePairedSeparator

paired line separator (like `\r\n` in DOS and Windows).

must contain exactly two chars or no chars at all, initially filled with RegExprLinePairedSeparator global constant)

see also Line Boundaries

For example, if you need Unix-style behaviour, assign `LineSeparators := #$a` and `LinePairedSeparator := ''` (empty string).

If you want to accept as line separators only `\x0D\x0A` but not `\x0D` or `\x0A` alone, then assign `LineSeparators := ''` (empty string) and `LinePairedSeparator := #$d#$a`.

By default 'mixed' mode is used (defined in RegExprLine[Paired]Separator[s] global constants):

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

Behaviour of this mode is detailed described in the Line Boundaries.

InvertCase

Invertion of character case. Redefine it if you want different behaviour.

Compile

Compiles regular expression.

Useful for example for GUI regular expressions editors - to check regular expression without using it.

Dump

Show `P-code` (compiled regular expression) as human-readable string.

5.2.2 Global constants

EscChar

Escape-char, by default \.

RegExprModifierI

Modifier i default value

RegExprModifierR

Modifier r default value

RegExprModifierS

Modifier s default value

RegExprModifierG

Modifier g default value

RegExprModifierM

Modifier m default value

RegExprModifierX

Modifier x default value

RegExprSpaceChars

Default for *SpaceChars* property

RegExprWordChars

Default value for *WordChars* property

RegExprLineSeparators

Default value for *LineSeparators* property

RegExprLinePairedSeparator

Default value for *LinePairedSeparator* property

RegExprInvertCaseFunction

Default for *InvertCase* property

5.2.3 Global functions

All this functionality is available as methods of `TRegExpr`, but with global functions you do not need to create `TReExpr` instance so your code would be more simple if you just need one function.

ExecRegExpr

true if the string matches the regular expression. Just as *Exec* in `TRegExpr`.

SplitRegExpr

Splits the string by regular expressions. See also *Split* if you prefer to create `TRegExpr` instance explicitly.

ReplaceRegExpr

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

Type
  TRegexReplaceOption = (rroModifierI,
                         rroModifierR,
```

(continues on next page)

(continued from previous page)

```
                          rroModifierS,
                          rroModifierG,
                          rroModifierM,
                          rroModifierX,
                          rroUseSubstitution,
                          rroUseOsLineEnd);
  TRegexReplaceOptions = Set of TRegexReplaceOption;

function ReplaceRegExpr (
    const ARegExpr, AInputStr, AReplaceStr : RegExprString;
    Options :TRegexReplaceOptions
) : RegExprString; overload;
```

Returns the string with regular expressions replaced by the `AReplaceStr`. See also *Replace* if you prefer to create TRegExpr instance explicitly.

If last argument (`AUseSubstitution`) is true, then `AReplaceStr` will be used as template for `Substitution` methods:

```
ReplaceRegExpr (
  '((?i)block|var)\s*(\s*\([^ ]*\)\s*)\s*',
  'BLOCK(test1)',
  'def "$1" value "$2"',
  True
)
```

Returns `def 'BLOCK' value 'test1'`

But this one (note there is no last argument):

```
ReplaceRegExpr (
  '((?i)block|var)\s*(\s*\([^ ]*\)\s*)\s*',
  'BLOCK(test1)',
  'def "$1" value "$2"'
)
```

Returns `def "$1" value "$2"`

Version with options

With `Options` you control `\n` behaviour (if `rroUseOsLineEnd` then `\n` is replaced with `\n\r` in Windows and `\n` in Linux). And so on.

```
Type
  TRegexReplaceOption = (rroModifierI,
                         rroModifierR,
                         rroModifierS,
                         rroModifierG,
                         rroModifierM,
                         rroModifierX,
                         rroUseSubstitution,
                         rroUseOsLineEnd);
```

QuoteRegExprMetaChars

Replace all metachars with its safe representation, for example `abc'cd.(` converts into `abc\'cd\.\(`

This function usefull for r.e. autogeneration from user input

RegExprSubExpressions

Makes list of subexpressions found in `ARegExpr`

In `ASubExps` every item represent subexpression, from first to last, in format:

String - subexpression text (without '()')

low word of Object - starting position in ARegExpr, including '(' if exists! (first position is 1)

high word of Object - length, including starting '(' and ending ')' if exist!

`AExtendedSyntax` - must be `True` if modifier `/x` will be `On` while using the r.e.

Usefull for GUI editors of r.e. etc (you can find example of using in REStudioMain.pas)

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 `(`

If `Result <> 0`, then `ASubExprs` can contain empty items or illegal ones

5.2.4 ERegExpr

```
ERegExpr = class (Exception)
  public
   ErrorCode : integer; // error code. Compilation error codes are before 1000
   CompilerErrorPos : integer; // Position in r.e. where compilation error occured
 end;
```

5.2.5 Unicode

UniCode slows down performance so use it only if you really need Unicode support.

To use Unicode uncomment `{$DEFINE UniCode}` in regexpr.pas (remove `off`).

After that all strings will be treated as WideString.

	English		Deutsch		Français	Español

5.3 FAQ

5.3.1 I found a terrible bug: TRegExpr raises Access Violation exception!

Answer

You must create the object before usage. So, after you declared something like:

```
r : TRegExpr
```

do not forget to create the object instance:

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

5.3.2 Regular expressions with (?=...) do not work

Look ahead is not implemented in the TRegExpr. But in many cases you can easily replace it with simple subexpressions.

5.3.3 Does it support Unicode?

Answer

How to use Unicode

5.3.4 Why does TRegExpr return more then one line?

For example, r.e. `<font .\*>` returns the first `<font`, then the rest of the file including last `</html>`.

Answer

For backward compatibility, modifier /s is `On` by default.

Switch it Off and `.` will match any but Line separators - exactly as you wish.

BTW I suggest `<font ([^\n>]*)>`, in `Match[1]` will be the URL.

5.3.5 Why does TRegExpr return more then I expect?

For example r.e. `<p>(.+)</p>` applyed to string `<p>a</p><p>b</p>` returns `a</p><p>b` but not `a` as I expected.

Answer

By default all operators works in `greedy` mode, so they match as more as it possible.

If you want `non-greedy` mode you can use `non-greedy` operators like `+?` and so on or switch all operators into `non-greedy` mode with help of modifier `g` (use appropriate TRegExpr properties or operator `?(-g)` in r.e.).

5.3.6 How to parse sources like HTML with help of TRegExpr

Answer

Sorry folks, but it's nearly impossible!

Of course, you can easily use TRegExpr for extracting some information from HTML, as shown in my examples, but if you want accurate parsing you have to use real parser, not r.e.

You can read full explanation in Tom Christiansen and Nathan Torkington `Perl Cookbook`, for example.

In short - there are many structures that can be easy parsed by real parser but cannot at all by r.e., and real parser is much faster to do the parsing, because r.e. doesn't simply scan input stream, it performs optimization search that can take a lot of time.

5.3.7 Is there a way to get multiple matches of a pattern on TRegExpr?

Answer

You can iterate matches with ExecNext method.

If you want some example, please take a look at `TRegExpr.Replace` method implementation or at the examples for HyperLinksDecorator

5.3.8 I am checking user input. Why does TRegExpr return `True` for wrong input strings?

Answer

In many cases TRegExpr users forget that regular expression is for **search** in input string.

So, for example if you use `\d{4,4}` expression, you will get success for wrong user inputs like `12345` or `any letters 1234`.

You have to check from line start to line end to ensure there are no anything else around: `^\d{4,4}$`.

5.3.9 Why does non-greedy iterators sometimes work as in greedy mode?

For example, the r.e. `a+?,b+?` applied to string `aaa,bbb` matches `aaa,b`, but should it not match `a,b` because of non-greediness of first iterator?

Answer

This is because of TRegExpr way to work. In fact many others r.e. engines work exactly the same: they performe only `simple` search optimization, and do not try to do the best optimization.

In some cases it's bad, but in common it's rather advantage then limitation, because of performance and predictability reasons.

The main rule - r.e. first of all try to match from current place and only if that's completely impossible move forward by one char and try again from next position in the text.

So, if you use `a,b+?` it'll match `a,b`. In case of `a+?,b+?` it's now not recommended (we add non-greedy modifyer) but still possible to match more then one `a`, so TRegExpr will do it.

TRegExpr like Perl's or Unix's r.e. doesn't attempt to move forward and check - would it will be "better" match. Fisrt of all, just because there is no way to say it's more or less good match.

5.3.10 How can I use TRegExpr with Borland C++ Builder?

I have a problem since no header file (`.h` or `.hpp`) is available.

Answer

- Add `RegExpr.pas` to `bcb` project.
- Compile project. This generates the header file `RegExpr.hpp`.
- Now you can write code which uses the `RegExpr` unit.

- Don't forget to add `#include "RegExpr.hpp"` where needed.
- Don't forget to replace all `\` in regular expressions with `\\` or redefined EscChar const.

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

Answer

The hint is in the previous question ;) Symbol `\` has special meaning in `C++`, so you have to `escape` it (as described in previous answer). But if you don't like r.e. like `\\w+\\\\w+\\.\\w+` you can redefine the constant `EscChar` (in `RegExpr.pas`). For example `EscChar = "/"`. Then you can write `/w+/w+/./w+`, looks unusual but more readable.

	English		Deutsch		Français	Español

5.4 Demos

Demo code for TRegExpr

5.4.1 Introduction

If you don't familiar with regular expression, please, take a look at the r.e.syntax.

TRegExpr interface described in TRegExpr interface.

5.4.2 Text2HTML

Text2HTML sources

Publish plain text as HTML

Uses unit HyperLinksDecorator that is based on TRegExpr.

This unit contains functions to decorate hyper-links.

For example, replaces `www.masterAndrey.com` with `<a href="http://www.masterAndrey.com">www.masterAndrey.com</a>` or `filbert@yandex.ru` with `<a href="mailto:filbert@yandex.ru">filbert@yandex.ru</a>`.

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

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

TDecorateURLsFlagSet = set of TDecorateURLsFlags;

function DecorateEMails (const AText : string) : string;
```

Value	Meaning
durlProto	Protocol (like `ftp://` or `http://`)
durlAddr	TCP address or domain name (like `masterAndrey.com`)
durlPort	Port number if specified (like `:8080`)
durlPath	Path to document (like `index.html`)
durlBMark	Book mark (like `#mark`)
durlParam	URL params (like `?ID=2&User=13`)

Returns input text `AText` with decorated hyper links.

`AFlags` describes, which parts of hyper-link must be included into visible part of the link.

For example, if *AFlags* is `[durlAddr]` then hyper link `www.masterAndrey.com/contacts.htm` will be decorated as `<a href="www.masterAndrey.com/contacts.htm">www.masterAndrey.com</a>`.

5.4.3 TRegExprRoutines

Very simple examples, see comments inside the unit

5.4.4 TRegExprClass

Slightly more complex examples, see comments inside the unit

CHAPTER 6

Translations

The documentation is available in English and Russian.

There are also old translations to German, Bulgarian, French and Spanish. If you want to help to update this old translations please contact me.

New translations are based on GetText and can be edited with transifex.com.

They are already machine-translated and need only proof-reading and may be some copy-pasting from old translations.

CHAPTER 7

Gratitude

Many features suggested and a lot of bugs founded (and even fixed) by TRegExpr's contributors.

I cannot list here all of them, but I do appreciate all bug-reports, features suggestions and questions that I am receiving from you.

- Guido Muehlwitz - found and fixed ugly bug in big string processing
- Stephan Klimek - testing in CPPB and suggesting/implementing many features
- Steve Mudford - implemented Offset parameter
- Martin Baur (www.mindpower.com) - German translation, usefull suggestions
- Yury Finkel - implemented UniCode support, found and fixed some bugs
- Ralf Junker - Implemented some features, many optimization suggestions
- Simeon Lilov - Bulgarian translation
- Filip Jirsk and Matthew Winter - help in Implementation non-greedy mode
- Kit Eason many examples for introduction help section
- Juergen Schroth - bug hunting and useful suggestions
- Martin Ledoux - French translation
- Diego Calp, Argentina - Spanish translation