TRegExpr Documentation

Release 1.147

Andrey Sorokin

Sep 20, 2023

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 the complete regular expressions cheat sheet.

5.1.2 Characters

Simple matches

Any single character (except special regex characters) matches itself. A series of (not special) characters matches that series of characters in the input string.

RegEx	Matches
`foobar`	`foobar`

Non-Printable Characters (escape-codes)

To specify character by its Unicode code, use the prefix `\x` followed by the hex code. For 3-4 digits code (after U+00FF), enclose the code into braces.

RegEx	Matches
`\xAB`	character with 2-digit hex code `AB`
`\x{AB20}`	character with 1..4-digit hex code `AB20`
`foo\x20bar`	`foo bar` (note space in the middle)

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

RegEx	Matches
`\t`	tab (HT/TAB), same as `\x09`
`\n`	line feed (LF), same as `\x0a`
`\r`	carriage return (CR), same as `\x0d`
`\f`	form feed (FF), same as `\x0c`
`\a`	alarm (BEL), same as `\x07`
`\e`	escape (ESC), same as `\x1b`
`\cA ... \cZ`	chr(0) to chr(25). For example, `\cI` matches the tab-char. Lower-case letters "a"..."z" are also supported.

Escaping

To represent special regex character (one of `.+*?|\()[]{}^$`), prefix it with a backslash `\`. The literal backslash must be escaped too.

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 square brackets `[]`. The class matches any **single** 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 **except** the characters listed in the class.

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

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

If you want the dash − 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 `]` as part of the class 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 chr(10) to chr(13)

Dot Meta-Char

Meta-char `.` (dot) by default matches any character. But if you turn **off** the *modifier /s*, then it won't match line-break characters.

The `.` does not act as meta-class inside *user character classes*. `[.]` means a literal ".".

Meta-Classes

There are a number of predefined character classes that keeps regular expressions more compact, "meta-classes":

RegEx	Matches
`\w`	an alphanumeric character, including _
`\W`	a non-alphanumeric
`\d`	a numeric character (same as `[0-9]`)
`\D`	a non-numeric
`\s`	any space (same as `[ \t\n\r\f]`)
`\S`	a non-space
`\h`	horizontal whitespace: the tab and all characters in the "space separator" Unicode category
`\H`	not a horizontal whitespace
`\v`	vertical whitespace: all characters treated as line-breaks in the Unicode standard
`\V`	not a vertical whitespace
`\R`	unicode line break: LF, pair CR LF, CR, FF (form feed), VT (vertical tab), U+0085, U+2028, U+2029

You may use all meta-classes, mentioned in the table above, 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 these classes.

5.1.4 Boundaries

Line Boundaries

Meta-char	Matches
^	zero-length match at start of line
$	zero-length match at end of line
\A	zero-length match at the very beginning
\z	zero-length match at the very end
\Z	like \z but also matches before the final line-break
\G	zero-length match at the end pos of the previous match

Examples:

RegEx	Matches
^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

Meta-char ^ matches zero-length position at the beginning of the input string. $ - at the ending. If *modifier /m* is **on**, they also match at the beginning/ending of individual lines in the multi-line text.

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.

Meta-char \A matches zero-length position at the very beginning of the input string, \z - at the very ending. They ignore *modifier /m*. \Z is like \z but also matches before the final line-break (LF and CR LF). Behaviour of \A, \z, \Z is made like in most of major regex engines (Perl, PCRE, etc).

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 2 line-breaks in the wrong order.

Note: TRegExpr

Multi-line processing can be tuned by properties LineSeparators and UseLinePairedBreak.

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

Quantifiers

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

RegEx	Matches
`{n}`	exactly `n` times
`{n,}`	at least `n` times
`{,m}`	not more than `m` times (only with AllowBraceWithoutMin)
`{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.

The variant `{,m}` is only supported if the property AllowBraceWithoutMin is set.

There is no practical limit to the values n and m (limit is maximal signed 32-bit value).

Using `{` without a correct range will give an error. This behaviour can be changed by setting the property AllowLiteralBraceWithoutRange, which will accept `{` as a literal char, if not followed by a range. A range with a low value bigger than the high value will always give an error.

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...10 instances of `foobar` (`()` is *group*)

Greediness

Quantifiers in "greedy" mode takes as many as possible, in "lazy" mode - as few as possible.

By default all quantifiers are "greedy". Append the character `?` to make any quantifier "lazy".

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 "lazy" mode (*modifier /g*, below we use *in-line modifier change*).

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

Possessive Quantifier

The syntax is: `a++`, `a*+`, `a?+`, `a{2,4}+`. Currently it's supported only for simple braces, but not for braces after group like `(foo|bar){3,5}+`.

This regex feature is described here. In short, possessive quantifier speeds up matching in complex cases.

5.1.6 Choice

Expressions in the choice are separated by vertical bar `|`.

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 Groups

The brackets `()` are used to define groups (ie subexpressions).

Note: TRegExpr

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

You can substitute them with Substitute.

Groups are numbered from left to right by their opening parenthesis (including nested groups). First group has index 1. The entire regex has index 0.

For string `foobar`, the regex `(foo(bar))` will find:

Group	Value
0	`foobar`
1	`foobar`
2	`bar`

5.1.8 Backreferences

Meta-chars `\1` through `\9` are interpreted as backreferences to capture groups. They match the previously found group with the specified index.

The meta char `\g` followed by a number is also interpreted as backreferences to capture groups. It can be followed by a multi-digit number.

RegEx	Matches
`(.)\1+`	`aaaa` and `cc`
`(.+)\1+`	also `abab` and `123123`
`(.)\g1+`	`aaaa` and `cc`

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

5.1.9 Named Groups and Backreferences

To make some group named, use this syntax: `(?P<name>expr)`. Also Perl syntax is supported: `(?'name'expr)`. And further: `(?<name>expr)`

Name of group must be valid identifier: first char is letter or "_", other chars are alphanumeric or "_". All named groups are also usual groups and share the same numbers 1 to 9.

Backreferences to named groups are `(?P=name)`, the numbers `\1` to `\9` can also be used. As well as the example `\g` and `\k` in the table below.

5.2 Supported syntax are

`(?P=name) \g{name} \k{name} \k<name> \k'name'` ==========

Example

RegEx	Matches
`(?P<qq>['"])\w+(?P=qq)`	`"word"` and `'word'`

5.2.1 Modifiers

Modifiers are for changing behaviour of regular expressions.

You can set modifiers globally in your system or change inside 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-яё]` you can write `[а-я]` if you need all Russian characters.

When the modifier is `On`:

RegEx	Matches
а-я	chars from а to я and
А-Я	chars from А to Я and
а-яА-ЯёЁ	all russian symbols

The modifier is set *On* by default.

5.2.2 Assertions

Positive lookahead assertion: `foo(?=bar)` matches "foo" only before "bar", and "bar" is excluded from the match.

Negative lookahead assertion: `foo(?!bar)` matches "foo" only if it's not followed by "bar".

Positive lookbehind assertion: `(?<=foo)bar` matches "bar" only after "foo", and "foo" is excluded from the match.

Negative lookbehind assertion: `(?<!foo)bar` matches "bar" only if it's not prefixed with "foo".

Limitations:

- Variable length lookbehind are not allowed to contain capture groups. This can be allowed by setting the property `AllowUnsafeLookBehind`. If this is enabled and there is more than one match in the text that the group might capture, then the wrong match may be captured. This does not affect the correctness of the overall assertion. (I.e., the lookbehind will correctly return if the text before matched the pattern).
- Variable length lookbehind may be slow to execute, if they do not match.

5.2.3 Non-capturing Groups

Syntax is like this: `(?:expr)`.

Such groups do not have the "index" and are invisible for backreferences. Non-capturing groups are used when you want to group a subexpression, but you do not want to save it as a matched/captured portion of the string. So this is just a way to organize your regex into subexpressions without overhead of capturing result:

RegEx	Matches
`(https?|ftp)://([^/\r\n]+)`	in `https://sorokin.engineer` matches `https` and `sorokin.engineer`
`(?:https?|ftp)://([^/\r\n]+)`	in `https://sorokin.engineer` matches only `sorokin.engineer`

5.2.4 Atomic Groups

Syntax is like this: `(?>expr|expr|...)`.

Atomic groups are special case of non-capturing groups. Description of them.

5.2.5 Inline Modifiers

Syntax for one modifier: `(?i)` to turn on, and `(?-i)` to turn off. Many modifiers are allowed like this: `(?msgxr-imsgxr)`.

You may use it inside regular expression for modifying modifiers on-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 group, it will affect only this group - specifically the part of the group that follows the modifiers. So in `((?i)Saint)-Petersburg` it affects only group `((?i)Saint)` so it will match `saint-Petersburg` but not `saint-petersburg`.

Inline modifiers can also be given as part of a non-capturing group: `(?i:pattern)`.

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`

5.2.6 Comments

Syntax is like this: `(?#text)`. Text inside brackets is ignored.

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

5.2.7 Recursion

Syntax is `(?R)`, the alias is `(?0)`.

The regex `a(?R)?z` matches one or more letters "a" followed by exactly the same number of letters "z".

The main purpose of recursion is to match balanced constructs or nested constructs. The generic regex is `b(?:m|(?R))*e` where "b" is what begins the construct, "m" is what can occur in the middle of the construct, and "e" is what occurs at the end of the construct.

If what may appear in the middle of the balanced construct may also appear on its own without the beginning and ending parts then the generic regex is `b(?R)*e|m`.

5.2.8 Subroutine calls

Syntax for call to numbered groups: `(?1)` ... `(?90)` (maximal index is limited by code).

Syntax for call to named groups: `(?P>name)`. Also Perl syntax is supported: `(?&name)`.

5.3 Supported syntax are

`(?number) (?P>name) (?&name) \g<name> \g'name' ==========`

This is like recursion but calls only code of capturing group with specified index.

5.3.1 Unicode Categories

Unicode standard has names for character categories. These are 2-letter strings. For example “Lu” is uppercase letters, “Ll” is lowercase letters. And 1-letter bigger category “L” is all letters.

- Cc - Control
- Cf - Format
- Co - Private Use
- Cs - Surrrogate
- Ll - Lowercase Letter
- Lm - Modifier Letter
- Lo - Other Letter
- Lt - Titlecase Letter
- Lu - Uppercase Letter
- Mc - Spacing Mark
- Me - Enclosing Mark
- Mn - Nonspacing Mark
- Nd - Decimal Number
- Nl - Letter Number
- No - Other Number
- Pc - Connector Punctuation
- Pd - Dash Punctuation
- Pe - Close Punctuation
- Pf - Final Punctuation
- Pi - Initial Punctuation
- Po - Other Punctuation
- Ps - Open Punctuation
- Sc - Currency Symbol
- Sk - Modifier Symbol
- Sm - Math Symbol
- So - Other Symbol
- Zl - Line Separator
- Zp - Paragraph Separator

- Zs - Space Separator

Meta-character `\p` denotes one Unicode char of specified category. Syntax: `\pL` and `\p{L}` for 1-letter name, `\p{Lu}` for 2-letter names.

Meta-character `\P` is inverted, it denotes one Unicode char **not** in the specified category.

These meta-characters are supported within character classes too.

5.3.2 Afterword

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

	English		Deutsch		Français	Español

5.4 TRegExpr

Implements regular expressions in pure Pascal. Compatible with Free Pascal, Delphi 2-7, C++Builder 3-6.

To use it, copy files "regexpr.pas", "regexpr_unicodedata.pas", "regexpr_compilers.inc", to your project folder.

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

5.4.1 TRegExpr class

VersionMajor, VersionMinor

Return major and minor version of the component.

```
VersionMajor = 1
VersionMinor = 101
```

Expression

Regular expression.

For optimization, regular expression is automatically compiled into P-code. Human-readable form of the P-code is returned 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 to (?ismx-ismx). For example `ModifierStr := 'i-x'` will switch on the 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", 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

Finds regular expression against `AInputString`, starting from the beginning.

The overloaded `Exec` version without `AInputString` exists, it uses `AInputString` from previous call.

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

ExecNext

Finds next match. If parameter `ABackward` is True, it goes downto position 1, ie runs backward search.

Without parameter it 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 `AInputString` starting from `AOffset` position (1-based).

Parameter `ABackward` means going from `AOffset` downto 1, ie backward search.

Parameter `ATryOnce` means that testing for regex will be only at the initial position, without going to next/previous positions.

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`, where `$&` or `$0` are replaced with the found match, and `$1` to `$9` are replaced with found groups 1 to 9.

To use in template the characters `$` or `\`, escape them with a backslash `\`, like `\\` or `\$`.

Symbol	Description
`$&`	whole regular expression match
`$0`	whole regular expression match
`$1 .. $9`	contents of numbered group 1 .. 9
`\n`	in Windows replaced with `\r\n`
`\l`	lowercase one next char
`\L`	lowercase all chars after that
`\u`	uppercase 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'.
```

To use found named groups, use syntax `${name}`, where "name" is valid identifier of previously found named group (starting with non-digit).

Split

Splits `AInputStr` into `APieces` by regex occurrences.

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;

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

Returns the string with regex 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 callback function, so you can implement really complex functionality.

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

SubExprMatchCount

Count of groups (subexpressions) found in last *Exec* / *ExecNext* call.

If there are no groups found, but some string was found (Exec* returned True), it returns 0. If no groups nor some string were found (*Exec* / *ExecNext* returned false), it returns -1.

Note, that some group may be not found, and for such group `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

Position (1-based) of group with specified index. Result is valid only after some match was found. First group has index 1, the entire match has index 0.

Returns -1 if no group with specified index was found.

MatchLen

Length of group with specified index. Result is valid only after some match was found. First group has index 1, the entire match has index 0.

Returns -1 if no group with specified index was found.

Match

String of group with specified index. First group has index 1, the entire match has index 0. Returns empty string, if no such group was found.

MatchIndexFromName

Returns group index (1-based) from group name, which is needed for "named groups". Returns -1 if no such named group was found.

LastError

Returns Id of last error, or 0 if no errors occured (unusable if `Error` method raises exception). It also clears internal status to 0 (no errors).

ErrorMsg

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

CompilerErrorPos

Returns position in regex, where P-code compilation was 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

UseLinePairedBreak

Boolean property, enables to detect paired line separator CR LF.

See also Line Boundaries

For example, if you need only Unix-style separator LF, assign `LineSeparators := #$a` and `UseLinePairedBreak := False`.

If you want to accept as line separators only CR LF but not CR or LF alone, then assign `LineSeparators := ''` (empty string) and `UseLinePairedBreak := True`.

By default, "mixed" mode is used (defined in RegExprLineSeparators global constant):

```
LineSeparators := #$d#$a;
UseLinePairedBreak := True;
```

Behaviour of this mode is described in the Line Boundaries.

Compile

Compiles regular expression to internal P-code.

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

Dump

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

5.4.2 Global constants

EscChar

Escape character, by default backslash `'\'`.

SubstituteGroupChar

Char used to prefix groups (numbered and named) in Substitute method, 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.

5.4.3 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

Returns True if the string matches the regular expression. Just like *Exec* in `TRegExpr`.

SplitRegExpr

Splits the string by regular expression occurences. 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,
                         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,
```

(continues on next page)

(continued from previous page)

```
                        rroModifierM,
                        rroModifierX,
                        rroUseSubstitution,
                        rroUseOsLineEnd);
```

QuoteRegExprMetaChars

Replace all metachars with its safe representation, for example `abc'cd.(` is converted to `abc\'cd\.\(`

This function is useful for regex auto-generation from user input.

RegExprSubExpressions

Makes list of subexpressions found in `ARegExpr`.

In `ASubExps` every item represents 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` os on, while using the regex.

Usefull for GUI editors of regex (you can find example of usage in REStudioMain.pas)

Result code	Meaning
0	Success. No unbalanced brackets were found.
-1	Not enough closing brackets `)`.
-(n+1)	At position n it was found opening `[` without corresponding closing `]`.
n	At position n it was found closing bracket `)` without corresponding opening `(`.

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

5.4.4 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;
```

5.4.5 Unicode

In Unicode mode, all strings (InputString, Expression, internal strings) are of type UnicodeString/WideString, instead of simple "string". 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`).

	English		Deutsch		Français	Español

5.5 FAQ

5.5.1 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.
```

5.5.2 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.

5.5.3 Does it support Unicode?

Answer

How to use Unicode

5.5.4 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.

5.5.5 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.).

5.5.6 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.

5.5.7 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

5.5.8 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}$`.

5.5.9 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.

5.5.10 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.

5.5.11 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

5.6 Demos

Demo code for TRegExpr

5.6.1 Introduction

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

TRegExpr interface described in TRegExpr interface.

5.6.2 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>`.

5.6.3 TRegExprRoutines

Very simple examples, see comments inside the unit

5.6.4 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 Weblate.

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.

- Alexey Torgashin - added many features in 2019-2020, e.g. named groups, non-capturing groups, assertions, backward search
- Guido Muehlwitz - found and fixed ugly bug in big string processing
- Stephan Klimek - testing in C++Builder 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