

---

# TRegExpr Documentation

*Выпуск 1.147*

Andrey Sorokin

сент. 16, 2021



<b>1</b>	<b>Отзывы</b>	<b>3</b>
<b>2</b>	<b>Быстрый старт</b>	<b>5</b>
<b>3</b>	<b>Обратная связь</b>	<b>7</b>
<b>4</b>	<b>Исходный код</b>	<b>9</b>
<b>5</b>	<b>Документация</b>	<b>11</b>
5.1	Регулярные выражения (RegEx) . . . . .	11
5.2	TRegExpr . . . . .	22
5.3	Часто задаваемые вопросы . . . . .	31
5.4	Demos . . . . .	34
<b>6</b>	<b>Переводы</b>	<b>37</b>
<b>7</b>	<b>Благодарности</b>	<b>39</b>



	<a href="#">English</a>	<a href="#">Русский</a>	<a href="#">Deutsch</a>	<a href="#">Български</a>	<a href="#">Français</a>	<a href="#">Español</a>
--	-------------------------	-------------------------	-------------------------	---------------------------	--------------------------	-------------------------

Библиотека TRegExpr реализует [регулярные выражения](#).

Регулярные выражения - простой в использовании и мощный инструмент для сложного поиска и замены, а также для проверки текста на основе шаблонов.

Это особенно полезно для проверки пользовательского ввода в формах ввода - для проверки адресов электронной почты и так далее.

Также вы можете извлекать номера телефонов, почтовые индексы и т.д. из веб-страниц или документов, искать сложные шаблоны в файлах журналов и все, что вы можете себе представить. Правила (шаблоны) могут быть изменены без перекомпиляции вашей программы.

TRegExpr реализован на чистом Паскале. Является частью [Lazarus \(Free Pascal\)](#) проекта. Но также существует как отдельная библиотека, которая может быть скомпилирована Delphi 2-7, Borland C ++ Builder 3-6.



## Глава 1

---

### Отзывы

---

Как библиотека была встречена.





---

### Быстрый старт

---

Чтобы использовать библиотеку, просто добавьте [исходники](#) в ваш проект и далее используйте класс `TRegExpr`.

Благодаря [FAQ](#) вы можете учиться на чужих ошибках.

Готовое к запуску приложение Windows [REStudio](#) поможет вам выучить и отладить регулярные выражения.



Если вы видите какие-либо проблемы, пожалуйста, [создайте баг](#).



Чистый Object Pascal.

- Оригинальная версия
- FreePascal fork (GitHub зеркало SubVersion)



	English	Русский	Deutsch	Български	Français	Español
--	---------	---------	---------	-----------	----------	---------

## 5.1 Регулярные выражения (RegEx)

### 5.1.1 Вступление

Регулярные выражения - удобный способ описывать шаблоны текстов.

С помощью регулярных выражений вы можете проверять пользовательский ввод, искать некоторые шаблоны, такие как электронные письма телефонных номеров на веб-страницах или в некоторых документах и так далее.

Ниже приведена исчерпывающая шпаргалка по регулярным выражениям всего на одной странице.

### 5.1.2 Символы

#### Простые совпадения

Серия символов соответствует этой серии символов во входной строке.

RegEx	Находит
foobar	foobar

#### Непечатаемые символы (escape-коды)

Для представления непечатаемого символа в регулярном выражении используется `\x` с шестнадцатеричным кодом. Если код длиннее 2 цифр (более `U+00FF`), то он обрамляется в фигурные скобки.

RegEx	Находит
\xAB	символ с 2-значным шестнадцатеричным кодом AB
\x{AB20}	символ с 1-4 значным шестнадцатеричным кодом AB20
foo\x20bar	foo bar (обратите внимание на пробел в середине)

Существует ряд predefined `escape`-кодов для непечатных символов, как в языке C:

RegEx	Находит
\t	tab (HT/TAB), тоже что \x09
\n	символ новой строки (LF), то же что \x0a
\r	возврат каретки (CR), тоже что \x0d
\f	form feed (FF), то же что \x0c
\a	звонок (BEL), тоже что \x07
\e	escape (ESC), то же что \x1b
\cA ... \cZ	chr(0) по chr(25). Например \cI соответствует табуляции. Также поддерживаются буквы в нижнем регистре «a»...»z».

## Эскейпинг

Для представления спецсимволов (`.+*?|\()\[\{\}^$`), перед ними надо поставить `\`. Чтобы вставить сам обратный слэш его надо удвоить.

RegEx	Находит
\^FooBarPtr	^FooBarPtr здесь ^ не означает <i>начало строки</i>
[a\\]	[a] это не <i>класс символов</i>

## 5.1.3 Классы символов

### Пользовательские классы

Символьный класс - это список символов внутри `[]`. Класс соответствует любому **одному** символу, указанному в этом классе.

RegEx	Находит
foob[aeiou]r	foobar, foobar и т. д., но не foobbr, foobcr и т. д.

Вы можете **инвертировать** класс - если первый символ после `[` является `^`, то класс соответствует любому символу, **кроме** символов, перечисленных в классе.

RegEx	Находит
foob[^aeiou]r	foobbr, foobcr и т. д., но не foobar, foobar и т. д.

Внутри списка символ - используется для указания диапазона, так что `a-z` представляет все символы между `a` и `z` включительно.



Если вы хотите, чтобы `-` сам был членом класса, поместите его в начало или конец списка или предварите его обратной косой чертой (*escape*).

Если вы хотите буквально использовать символ `]` поместите его в начало списка или *escape* обратной косой чертой.

RegEx	Находит
<code>[-az]</code>	<code>a</code> , <code>z</code> и <code>-</code>
<code>[az-]</code>	<code>a</code> , <code>z</code> и <code>-</code>
<code>[A\ -z]</code>	<code>a</code> , <code>z</code> и <code>-</code>
<code>[a-z]</code>	символы от <code>a</code> до <code>z</code>
<code>[\n-\x0D]</code>	символы от <code>#10</code> до <code>#13</code>

## Метаклассы

Существует ряд predefined классов символов, которые делают регулярные выражения более компактными. Их называют метаклассы:

RegEx	Находит
<code>\w</code>	буквенно-цифровой символ (включая <code>_</code> )
<code>\W</code>	не буквенно-цифровой
<code>\d</code>	числовой символ (тоже, что <code>[0-9]</code> )
<code>\D</code>	нечисловой
<code>\s</code>	любой пробел (такой же как <code>[\t\n\r\f]</code> )
<code>\S</code>	не пробел
<code>\h</code>	горизонтальный разделитель. Табуляция, пробел и все символы в Unicode категории «разделители» (space separator Unicode category)
<code>\H</code>	не горизонтальный разделитель
<code>\v</code>	вертикальные разделители. новая строка и все символы «разделители строк» в Unicode
<code>\V</code>	не вертикальный разделитель

Все указанные в таблице метаклассы можно использовать внутри *пользовательских классов*.

RegEx	Находит
<code>foob\dr</code>	<code>foob1r</code> , <code>foob6r</code> и т. д., но не <code>foobar</code> , <code>foobbr</code> и т. д.
<code>foob[\w\s]r</code>	<code>foobar</code> , <code>foob r</code> , <code>foobbr</code> и т. д., но не <code>foob1r</code> , <code>foob=r</code> и т. д.

### Примечание: TRegExpr

Свойства `SpaceChars` и `WordChars` определяют, какие символы входят в классы `\w`, `\W`, `\s`, `\S`.

Таким образом, вы можете переопределить эти классы.

## 5.1.4 Разделители

### Разделители строк

Метасимвол	Находит
.	любой символ в строке, может включать разделители строк
^	совпадение нулевой длины в начале строки
\$	совпадение нулевой длины в конце строки
\A	совпадение нулевой длины в начале строки
\z	совпадение нулевой длины в конце строки
\Z	похож на \z но совпадает перед разделителем строки, а не сразу после него, как \z

Примеры:

RegEx	Находит
^foobar	foobar только если он находится в начале строки
foobar\$	foobar, только если он в конце строки
^foobar\$	foobar только если это единственная строка в строке
foob.r	foobar, foobbr, foob1r и так далее

Метасимвол `^` совпадает с точкой начала строки (нулевой длины). `$` - в конце строки. Если включен *modifier /m*, они совпадают с началами или концами строк внутри текста.

Обратите внимание, что в последовательности `\x0D\x0A` нет пустой строки.

---

#### Примечание: TRegExpr

Если вы используете [Unicode версию](#), то `^/$` также соответствует `\x2028`, `\x2029`, `\x0B`, `\x0C` или `\x85`.

---

Метасимвол `\A` совпадает с точкой нулевой длины в начале строки, `\z` - в конце (после символов завершения строки). Модификатор *modifier /m* на них не влияет. `\Z` тоже самое что `\z` но совпадает с точкой перед символами завершения строки (LF and CR LF).

Метасимвол `.` по умолчанию соответствует любому символу, но если вы выключите *modifier /s*, то `.` не будет совпадать с разделителями строк внутри строки.

Обратите внимание, что выражение `^.*$` не соответствует точке между `\x0D\x0A`, потому что это неразрывный разделитель строк. Но оно соответствует пустой строке в последовательности `\x0A\x0D`, поэтому из-за неправильного порядка кодов он не воспринимается как разделитель строк и считается просто двумя символами.

---

#### Примечание: TRegExpr

Многострочная обработка может быть настроена с помощью свойств [LineSeparators](#) и [LinePairedSeparator](#).

Таким образом, вы можете использовать разделители стиля Unix `\n` или стиль DOS / Windows `\r\n` или смешивать их вместе (как описано выше по умолчанию).

---

Если вы предпочитаете математически правильное описание, вы можете найти его на сайте [www.unicode.org](http://www.unicode.org).

## Разделители слов

RegEx	Находит
\b	разделитель слов
\B	разделитель с <b>не</b> -словом

Граница слова \b - это точка между двумя символами, у которой \w с одной стороны от нее и \W с другой стороны (в любом порядке).

## 5.1.5 Повторы

### Повтор

За любым элементом регулярного выражения может следовать допустимое число повторений элемента.

RegEx	Находит
{n}	ровно n раз
{n,}	по крайней мере n раз
{n,m}	по крайней мере n, но не более чем m раз
*	ноль или более, аналогично {0,}
+	один или несколько, похожие на {1,}
?	ноль или единица, похожая на {0,1}

То есть цифры в фигурных скобках {n,m} определяют минимальное n и максимальное m количество повторов (совпадений во входном тексте).

{n} эквивалентно {n,n} и означает **точно n раз**. {n,} совпадает n или более раз.

Теоретически значение n и m не ограничены (можно использовать максимальное значение для 32-х битного числа).

RegEx	Находит
foob.*r	foobar, foobalkjdfklkj9r и foobr
foob.+r	foobar, foobalkjdfklkj9r, но не foobr
foob.?r	foobar, foobbr и foobr, но не foobalkj9r
fooba{2}r	foobaar
fooba{2}r	foobaar, foobaaar, foobaaaar и т. д.
fooba{2,3}r	foobaar, или foobaaar, но не foobaaaar
(foobar){8,10}	8, 9 или 10 экземпляров foobar (()) это <i>Группа</i> )

### Жадность

*Повторы* в **жадном** режиме захватывают как можно больше из входного текста, в **не жадном** режиме - как можно меньше.

По умолчанию все повторы являются **жадными**. Используйте ? Чтобы сделать любой повтор **не жадным**.

Для строки abbbbc:

RegEx	Находит
<code>b+</code>	<code>bbbb</code>
<code>ᑭ+?</code>	<code>b</code>
<code>b*?</code>	пустую строку
<code>b{2,3}?</code>	<code>bb</code>
<code>b{2,3}</code>	<code>bbb</code>

Вы можете переключить все повторы в не жадный режим (*modifier* `/g`, ниже мы используем *in-line модификатор* `change`).

RegEx	Находит
<code>(?-g)ᑭ+</code>	<code>b</code>

### Сверхжадные повторы (Possessive Quantifier)

Синтаксис: `a++`, `a*+`, `a?+`, `a{2,4}+`. В настоящее время реализован только для простых групп и не будет работать для сложных, как например `(foo|bar){3,5}+`.

[Полное описание \(на английском\)](#) Вкратце, сверхжадный повтор ускоряет работу в сложных случаях.

## 5.1.6 Альтернативы

Выражения в списке альтернатив разделяются `|`.

Таким образом, `fee|fie|foe` будет соответствовать любому из `fee`, `fie` или `foe` (также как и `f(e|i|o)e`).

Первое выражение включает в себя все от последнего разделителя шаблона (`(`, `[` или начало шаблона) до первого `|`, а последнее выражение содержит все от последнего `|` к следующему разделителю шаблона.

Звучит сложно, поэтому обычной практикой является заключение списка альтернатив в скобки, чтобы минимизировать путаницу относительно того, где он начинается и заканчивается.

Выражения в списке альтернатив пробуются слева направо, принимается первое же совпадение.

Например, регулярное выражение `foo|foot` в строке `barefoot` будет соответствовать `foo` - первое же совпадение.

Также помните, что `|` в квадратных скобках воспринимается просто как символ, поэтому, если вы напишите `[fee|fie|foe]`, это тоже самое что `[feio]`.

RegEx	Находит
<code>foo(bar foo)</code>	<code>foobar</code> или <code>foofoo</code>

## 5.1.7 Группы (подвыражения)

Скобки `(...)` также могут использоваться для определения групп (подвыражений) регулярного выражения.

---

**Примечание:** TRegExpr

Позиция, длина и фактические значения подвыражений будут в `MatchPos`, `MatchLen` и `Match`.

Вы можете заменить их с помощью функции [Substitute](#).

Подвыражения нумеруются слева направо по открывающим их скобкам (включая вложенные группы (подвыражения)). У первой группы номер 1. У выражения в целом - 0.

Например, для входной строки `foobar` регулярное выражение `(foo(bar))` найдет:

Группы (подвыражения)	значение
0	<code>foobar</code>
1	<code>foobar</code>
2	<code>bar</code>

### 5.1.8 Ссылки на группы (Backreferences)

Метасимволы от `\1` до `\9` интерпретируются как ссылки на группы (подвыражения в `()`). `\n` соответствует ранее найденному подвыражению `n`.

RegEx	Находит
<code>(.)\1+</code>	<code>aaaa</code> и <code>cc</code>
<code>(.+)\1+</code>	также <code>abab</code> и <code>123123</code>

`(["']?)(\d+)\1` соответствует `"13"` (в двойных кавычках) или `'4'` (в одинарных кавычках) или `77` (без кавычек) и т. д.

### 5.1.9 Именованные группы (подвыражения) и ссылки на них

Чтобы присвоить имя группе используйте `(?P<name>expr)` или `(?'name'expr)`.

Имя группы должно начинаться с буквы или `_`, далее следуют буквы, цифры или `_`. Именованные и не именованные группы имеют общую нумерацию от 1 до 9.

Чтобы сослаться на именованную группу используйте `(?P=name)`. Или, как и для не именованных, цифры от `\1` до `\9`.

RegEx	Находит
<code>(?P&lt;qq&gt;["'])\w+(?P=qq)</code>	<code>"word"</code> и <code>'word'</code>

### 5.1.10 Модификаторы

Модификаторы предназначены для изменения поведения регулярных выражений.

Вы можете установить модификаторы глобально в вашей системе или изменить их внутри регулярного выражения, используя `(?imsxr-imsxr)`.

#### Примечание: TRegExpr

Для изменения модификаторов используйте [ModifierStr](#) или соответствующие TRegExpr свойства [Модификатор \\*](#).

Значения по умолчанию определены в [глобальных переменных](#). Скажем, глобальная переменная `RegExprModifierX` определяет значение по умолчанию для свойства `ModifierX`.

### i, без учета регистра

Регистро-независимые сравнения. Использует установленные в вашей системе языковые настройки, см. также [InvertCase](#).

### m, многострочные строки

Обрабатывать строку как несколько строк. Таким образом, `^` и `$` соответствуют началу или концу любой строки в любом месте строки.

Смотрите также *Разделители строк*.

### s, одиночные строки

Обрабатывать строку как одну строку. Так что `.` соответствует любому символу, даже разделителям строк.

Смотрите также *Разделители строк*, которые обычно не совпадают.

### г, жадность

---

**Примечание:** Специфичный для [TRegExpr](#) модификатор.

---

Отключив его `Off`, вы переключите *повторитель* в *не-жадный* режим.

Итак, если модификатор `/g` имеет значение `Off`, то `+` работает как `+`, `*` как `*` и так далее.

По умолчанию этот модификатор имеет значение `Вкл.`

### x, расширенный синтаксис

Позволяет комментировать регулярные выражения и разбивать их на несколько строк.

Если модификатор включен, мы игнорируем все пробелы, которые не заэскейплены обратной косой чертой, и не включены в класс символов.

Также символ `#` отделяет комментарии.

Обратите внимание, что вы можете использовать пустые строки для форматирования регулярного выражения для лучшей читаемости:

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

Это также означает, что если вам нужно вставить пробел или символ `#` в шаблон (вне класса символов, где они не затрагиваются `/x`), вам придется либо эскейпить их, либо кодировать, используя шестнадцатеричный код.

## г, русские диапазоны

---

**Примечание:** Специфичный для TRegExpr модификатор.

---

В русской таблице ASCII символы ё / Ё размещаются отдельно от других.

Большие и маленькие русские символы находятся в отдельных диапазонах, это не отличается от ситуации с английскими символами, но, тем не менее, я хотел иметь краткую форму.

С этим модификатором вместо [а-яА-ЯёЁ] вы можете написать [а-Я], если вам нужны все русские символы.

Когда модификатор включен:

RegEx	Находит
а-я	символы от а до я и ё
А-Я	символы от А до Я и Ё
а-Я	все русские символы

Модификатор по умолчанию установлен на Вкл.

### 5.1.11 Проверки или заглядывания вперед и назад (Assertions)

Заглядывание вперед (lookahead assertion) `foo(?=bar)` совпадает «foo» только перед «bar», при этом сама строка «bar» не войдет в найденный текст.

Отрицательное заглядывание вперед (negative lookahead assertion): `foo(?!bar)` совпадает «foo» только если после этой строки не следует «bar».

Ретроспективная проверка (lookbehind assertion): `(?<=foo)bar` совпадает «bar» только после «foo», при этом сама строка «foo» не войдет в найденный текст.

Отрицательное заглядывание вперед (negative lookahead assertion): `foo(?!bar)` совпадает «bar» только если перед этой строки нет «foo».

Ограничения:

- Скобки для заглядываний вперед должны быть в самом начале выражения. Не поддерживаются заглядывания внутри альтернатив (|) или групп.
- Для заглядывания назад `(?<!foo)bar`, выражение «foo» должно быть фиксированной длины. Допустимы повторы только с фиксированным числом {n} или {n,n}. Разрешено использование классов символов и точки. Не разрешены группы и альтернативы.
- Для остальных трех типов заглядываний, выражение в скобках может быть сколь угодно сложным.

### 5.1.12 Не захватываемые группы (подвыражения)

Синтаксис: `(?:subexpression)`.

У этих групп (подвыражений) нет номера, их нельзя указать в ссылке на группу. Эти группы используют чтобы за счет группировки сделать регулярное выражение более читаемым, но нет необходимости расходовать ресурсы на то, чтобы реально отдельно захватывать то, с чем такие группы совпадут:

RegEx	Находит
<code>(https? ftp)://([~/\r\n]+)</code>	в <code>https://sorokin.engineer</code> захватит подвыражения <code>https</code> и <code>sorokin.engineer</code>
<code>(?:https? ftp)://([~/\r\n]+)</code>	в <code>https://sorokin.engineer</code> захватит только <code>sorokin.engineer</code>

### 5.1.13 Атомарные группы

Синтаксис: `(?>expr|expr|...)`.

Атомарные группы это специальный случай незахватывающих групп. [Подробнее](#)

### 5.1.14 Модификаторы

Синтаксис для одного модификатора: `(?i)` чтобы включить, и `(?-i)` чтобы выключить. Для большого числа модификаторов используется синтаксис: `(?msgxr-imsgrx)`.

Можно использовать внутри регулярного выражения. Это может быть особенно удобно, поскольку оно имеет локальную область видимости. Оно влияет только на ту часть регулярного выражения, которая следует за оператором `(?imsgrx-imsgrx)`.

И если оно находится внутри подвыражения, оно будет влиять только на это подвыражение, а именно на ту часть подвыражения, которая следует за оператором. Таким образом, в `((?i)Saint)-Petersburg` это влияет только на подвыражение `((?i)Saint)`, поэтому оно будет соответствовать `saint-Petersburg`, но не `saint-petersburg`.

RegEx	Находит
<code>(?i)Saint-Petersburg</code>	<code>Saint-petersburg</code> и <code>Saint-Petersburg</code>
<code>(?i)Saint-(?-i)Petersburg</code>	<code>Saint-Petersburg</code> , но не <code>Saint-petersburg</code>
<code>(?i)(Saint-)?Petersburg</code>	<code>Saint-petersburg</code> и <code>saint-petersburg</code>
<code>((?i)Saint-)?Petersburg</code>	<code>saint-Petersburg</code> , но не <code>saint-petersburg</code>

### 5.1.15 Комментарии

Синтаксис: `(?#text)`. Все, что внутри скобок, игнорируется.

Обратите внимание, что комментарий закрывается ближайшим `)`, поэтому нет способа вставить литерал `)` в комментарий.

### 5.1.16 Рекурсия

Синтаксис `(?R)`, синоним `(?0)`.

Выражение `a(?R)?z` совпадает с одним или более символом «a» за которым следует точно такое же число символов «z».

Основное назначение рекурсии - сбалансировать обрамление вложенного текста. Общий вид `b(?:m|(?R))*e` где «b» это то что начинает обрамляемый текст, «m» это собственно текст, и «e» это то, что завершает обрамление.

Если же обрамляемый текст также может встречаться без обрамления то выражение будет `b(?R)*e|m`.



### 5.1.17 Вызовы подвыражений

Нумерованные группы (подвыражения) обозначают (?1) ... (?90) (максимальное число групп определяется константой в TRegExpr).

Синтаксис для именованных групп : (?P>name). Поддерживается также Perl вариант синтаксиса: (?&name).

Это похоже на рекурсию, но повторяет только указанную группу (подвыражение).

### 5.1.18 Unicode категории (category)

В стандарте Unicode есть именованные категории символов (Unicode category). Категория обозначается одной буквой, и еще одна добавляется, чтобы указать подкатеорию. Например «L» это буква в любом регистре, «Lu» - буквы в верхнем регистре, «Ll» - в нижнем.

- Cc - Control
- Cf - Формат
- Co - Частное использование
- Cs - Заменитель (Surrogate)
- Ll - Буква нижнего регистра
- Lm - Буква-модификатор
- Lo - Прочие буквы
- Lt - Titlecase Letter
- Lu - Буква в верхнем регистре
- Mc - Разделитель
- Me - Закрывающий знак (Enclosing Mark)
- Mn - Несамостоятельный символ, как умляут над буквой (Nonspacing Mark)
- Nd - Десятичная цифра
- Nl - Буквенная цифра - например, китайская, римская, руническая и т.д. (Letter Number)
- No - Другие цифры
- 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 - Математический символ
- So - Прочие символы

- Zl - Разделитель строк
- Zp - Разделитель параграфов
- Zs - Space Separator

Метасимвол `\p` это один символ указанной Unicode категории (category). Синтаксис: `\pL` или `\p{L}` если категория обозначается одним символом, `\p{Lu}` для 2-символьных категорий.

Метасимвол `\P` это символ **не** из Unicode категории (category).

Эти метасимволы также поддерживаются внутри пользовательских классов.

### 5.1.19 Послесловие

В этой [древней статье](#) из [прошлого века](#) есть примеры использования регулярных выражений.

	<a href="#">English</a>	<a href="#">Русский</a>	<a href="#">Deutsch</a>	<a href="#">Български</a>	<a href="#">Français</a>	<a href="#">Español</a>
--	-------------------------	-------------------------	-------------------------	---------------------------	--------------------------	-------------------------

## 5.2 TRegExpr

Реализует [регулярные выражения](#) в чистом паскале. Совместим с Free Pascal, Delphi 2-7, Borland C++ Builder 3-6.

Для использования скопируйте в свой проект файлы «[regexpr.pas](#)», «[regexpr\\_unicodedata.pas](#)», «[regexpr\\_compilers.inc](#)».

Библиотека уже включена в [Lazarus](#) (Free Pascal) проект, поэтому вам не нужно ничего копировать, если вы используете [Lazarus](#).

### 5.2.1 Класс TRegExpr

#### VersionMajor, VersionMinor

Вернуть мажорную и минорную версию, например, `version 0.944`

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

#### Expression

Регулярное выражение.

Для оптимизации регулярное выражение автоматически компилируется в **Р-код**. Читаемую человеком форму **Р-кода** возвращает *Dump*.

В случае каких-либо ошибок при компиляции вызывается метод **Error** (по умолчанию **Error** вызывает исключение *ERegExpr*)

## ModifierStr

Установить или получить значения модификаторов регулярных выражений.

Формат строки такой же, как в (? Ismx-ismx). Например, ModifierStr: = &#39;i-x&#39; включит модификатор /i, выключит /x и оставит без изменений другие.

Если вы попытаетесь установить неподдерживаемый модификатор, будет вызвано **Error**.

## ModifierI

Модификатор /i, «регистро-нечувствительный», инициализируется значением *RegExprModifierI*.

## ModifierR

Модификатор /г, «расширение русского диапазона», инициализируется значением *RegExprModifierR*.

## модификаторы

Модификатор /s, «одноточный текст», инициализируется значением *RegExprModifierS*.

## ModifierG

Модификатор /g, «жадность», инициализируется значением *RegExprModifierG*.

## ModifierM

Модификатор /m, «многострочный текст», инициализируется значением *RegExprModifierM*.

## ModifierX

Модификатор /x, «расширенный синтаксис», инициализируется значением *RegExprModifierX*.

## Exec

Ищет регулярное выражение в *AInputString*.

Доступна перегруженная версия *Exec* без *AInputString* - она использует *AInputString* из предыдущего вызова.

Смотрите также глобальную функцию *ExecRegExpr*, которую вы можете использовать без явного создания объекта *TRegExpr*.

## ExecNext

Ищет следующее совпадение. Если *ABackward* = *True* то ищет в обратном направлении.

Без параметра работает так же:

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

Вызывает исключение, если используется без предшествующего успешного вызова *Exec*, *ExecPos* или *ExecNext*.

Таким образом, вы всегда должны использовать что-то вроде

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

## ExecPos

Находит совпадение для *InputString*, начиная с позиции *AOffset* (нумерация с 1)

Параметр *ABackward* включает поиск от позиции *AOffset* к 1й позиции в тексте, то есть назад.

*ATryOnce* означает что совпадение проверяется только для начальной позиции, без смещения с нее.

## Строка ввода

Возвращает текущую входную строку (из последнего вызова *Exec* или последнего присвоения этому свойству).

Любое присвоение этому свойству очищает *Match*, *MatchPos* и *MatchLen*.

## Substitute

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

Возвращает *ATemplate*, где *\$&* или “\$0” заменяются на совпавший текст, а \$1 по \$9 заменяются на группу (подвыражение) с соответствующим номером с 1 по 9.

Чтобы поместить в шаблон символы \$ или \, используйте префикс \, например \\ или \\$.

условное обозначение	описание
\$&	полное совпадение регулярного выражения
\$0	полное совпадение регулярного выражения
\$1 .. \$9	совпадение нумерованных групп 1 .. 9
\n	для Windows \r\n
\l	следующий символ перевести в нижний регистр
\L	делает строчными все символы после этого
\u	делает заглавным один следующий символ
\U	делает заглавными все символы после этого

```
'1\ $ is $2\\rub\\' -> '1$ это <Match[2]>\rub\'  
'\U$1\\r' преобразуется '<Match[1] в верхнем регистре>\r'
```

Если вы хотите поместить необработанную цифру после `n`, вы должны разделить `n` фигурными скобками `{}`.

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

Чтобы использовать найденные группы используется синтаксис `${name}`, where `name` это номер найденной группы.

## Split

Разделяет `AInputStr` на `APieces` по найденным регулярным выражениям

Внутренне вызывает *Exec / ExecNext*

Смотрите также глобальную функцию *SplitRegExpr*, которую вы можете использовать без явного создания объекта `TRegExpr`.

## 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;
```

Возвращает строку с повторениями, замененными строкой замены.

Если последний аргумент (`AUseSubstitution`) равен `true`, то `AReplaceStr` будет использоваться в качестве шаблона для методов подстановки.

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

Возвращает `def "BLOCK" value "test1"`

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

Возвращает `def "$1" value "$2"`

Внутренне вызывает *Exec / ExecNext*

Перегруженная версия `ReplaceEx` работающая с функцией обратного вызова, чтобы реализовать сложную функциональность.

Смотрите также глобальную функцию *ReplaceRegExpr*, которую вы можете использовать без явного создания объекта `TRegExpr`.

## SubExprMatchCount

Количество подвыражений, которое было найдено в последнем вызове *Exec* / *ExecNext*.

Если нет подвыражений но было найдено все выражение (*Exec*\* вернул *True*), то *SubExprMatchCount*=0, если ни подвыражений, ни всего выражения не найдено (*Exec* / *ExecNext* вернул *false*), тогда *SubExprMatchCount*=-1.

Обратите внимание, что некоторые группы (подвыражения) могут быть не найдены и для таких групп *MathPos*=*MatchLen*=-1 и *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

Позиция группы (начиная с 1). Результат имеет смысл только после того как выражение было найдено. Первая группа имеет номер 1. Совпадение регулярного выражения в целом имеет номер 0.

Возвращает -1, если группа с указанным номером не была найдена.

## MatchLen

Длина группы с указанным номером. Результат имеет смысл только после того, как выражение было найдено. У первой группы номер 1, у выражения в целом номер 0.

Возвращает -1, если группа с указанным номером не была найдена.

## Match

Строка совпавшая с группой с указанным номером. Первая группа имеет номер 1, все выражение имеет номер 0. Возвращает пустую строку, если указанная группа не была найдена.

## MatchIndexFromName

Возвращает индекс группы (начиная с 1) по имени группы, необходимом для именованных групп. Или -1 если такой именованной группы не найдено.

## LastError

Возвращает ID последней ошибки, 0, если ошибок нет (невозможно использовать, если метод *Error* вызывает исключение) и очищает внутренний статус в 0 (без ошибок).

## ErrorMsg

Возвращает сообщение об ошибке `Error` с `ID = AErrorID`.

## CompilerErrorPos

Возвращает позицию в регулярном выражении, где компилятор остановился.

Полезно для диагностики ошибок.

## SpaceChars

Содержит символы, которые рассматриваются как `\s` (изначально заполнены глобальной константой *RegExprSpaceChars*)

## WordChars

Содержит символы, которые рассматриваются как `\w` (изначально заполнены глобальной константой *RegExprWordChars*)

## LineSeparators

разделители строк (например, `\n` в Unix), изначально заполненные глобальной константой *RegExprLineSeparators*)

смотрите также [Разделители строк](#)

## UseLinePairedBreak

Булево свойство, включает поиск парных разделителей строк CR LF.

смотрите также [Разделители строк](#)

Например, если вам нужно поведение в стиле Unix, присвойте `LineSeparators := #\n` и `LinePairedSeparator := ''` (пустая строка).

Если вы хотите принять в качестве разделителей строк только `\r` \ `\n`, но не `\r` или `\n`, тогда присвойте `LineSeparators := ''` (пустая строка) и `UseLinePairedBreak := True`.

По умолчанию используется смешанный режим (определенный в глобальных константах *RegExprLineSeparators*):

```
LineSeparators := #\r\n;\nUseLinePairedBreak := True;
```

Поведение этого режима подробно описано в [Разделителях строк](#).

## Compile

Компилирует регулярное выражение.

Полезно, например, для редакторов регулярных выражений в графическом интерфейсе - для проверки регулярных выражений без их использования.

## Dump

Показать Р-код (скомпилированное регулярное выражение) в виде удобочитаемой строки.

## 5.2.2 Глобальные константы

### EscChar

Escape-char, по умолчанию \.

### SubstituteGroupChar

Символ, используемый для указания группы (именованной или нумерованной) в методе Substitute, по умолчанию '\$'.

### RegExprModifierI

Модификатор i значение по умолчанию

### RegExprModifierR

Модификатор r значение по умолчанию

### RegExprModifierS

Модификатор s значение по умолчанию

### RegExprModifierG

Модификатор g значение по умолчанию

### RegExprModifierM

Модификатор m значение по умолчанию

### RegExprModifierX

Модификатор x значение по умолчанию

### RegExprSpaceChars

По умолчанию для свойства *SpaceChars*

### RegExprWordChars

Значение по умолчанию для свойства *WordChars*



## RegExprLineSeparators

Значение по умолчанию для свойства *LineSeparators*

### 5.2.3 Глобальные функции

Вся эта функциональность доступна как методы TRegExpr, но с глобальными функциями вам не нужно создавать экземпляр TRegExpr, поэтому ваш код будет более простым, если вам просто понадобится одна функция.

## ExecRegExpr

Значение True, если строка соответствует регулярному выражению. Так же, как *Exec* в TRegExpr.

## SplitRegExpr

Разбивает строку по регулярным выражениям. Смотрите также *Split*, если вы предпочитаете явно создавать экземпляр TRegExpr.

## 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;
```

Возвращает строку с регулярными выражениями, замененными на AReplaceStr. Смотрите также *Replace*, если вы предпочитаете создавать экземпляр TRegExpr явно.

Если последний аргумент (AUseSubstitution) равен True, то AReplaceStr будет использоваться в качестве шаблона для методов подстановки:

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

(continues on next page)

(продолжение с предыдущей страницы)

```
True
)
```

Возвращает `def 'BLOCK' value 'test1'`

Но этот (обратите внимание, что нет последнего аргумента):

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

Возвращает `def "$1" value "$2"`

### Версия с опциями

С `Options` вы управляете поведением `\n` (если `rroUseOsLineEnd`, то `\n` заменяется на `\n\r` в Windows и `\n` в Linux). И так далее.

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

### QuoteRegExprMetaChars

Замените все мета-символы их безопасным представлением, например, `abc'cd.(` преобразуется в `abc\'cd\.\(`

Эта функция полезна для повторной автогенерации из пользовательского ввода

### RegExprSubExpressions

Составляет список подвыражений, найденных в `ARegExpr`

В `ASubExps` каждый элемент представляет подвыражение, от первого до последнего, в формате:

`String` - текст подвыражения (без „()“)

Младшее слово `Object` - начальная позиция в `ARegExpr`, включая ‘(’ если существует! (первая позиция 1)

Старшее слово `Object` - длина, включая начало ‘(’ и окончание‘)’, если существует!

`AExtendedSyntax` - должно быть `True`, если модификатор `/x` будет `On` при использовании г.е.

Полезно для графических редакторов и т. д. (пример использования можно найти в [REStudioMain.pas](#))

Код результата	Имея в виду
0	Успех. Не найдено несбалансированных скобок
-1	Недостаточно закрывающих скобок )
-(n+1)	В позиции n было найдено открытие [ без соответствующего закрытия ]
N	В позиции n была найдена закрывающая скобка ) без соответствующего открывания (

Если `Result <> 0`, то `ASubExprs` может содержать пустые или недопустимые элементы.

## 5.2.4 ERegExpr

```
ERegExpr = class (Exception)
public
  ErrorCode : integer; // error code. Ошибка компиляции менее 1000
  CompilerErrorPos : integer; // Позиция в которой найдена ошибка
end;
```

## 5.2.5 Unicode

В режиме юникода все строки (`InputString`, `Expression` и пр.) будут типа `UnicodeString/WideString`, а не «string». Режим юникода замедляет работу, если он вам не нужен, лучше его не включать.

Чтобы использовать Unicode раскомментируйте `{ $DEFINE UniCode }` в `regexpr.pas` (удалить `off`).

	English	Русский	Deutsch	Български	Français	Español
--	---------	---------	---------	-----------	----------	---------

## 5.3 Часто задаваемые вопросы

### 5.3.1 Я обнаружил ужасную ошибку: TRegExpr вызывает исключение Access Violation!

#### Ответ

Вы должны создать объект перед использованием. Итак, после того, как вы объявили что-то вроде:

```
r: TRegExpr
```

не забудьте создать экземпляр объекта:

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

### 5.3.2 Регулярные выражения с (? = ...) не работают

Look-ahead не реализованы в `TRegExpr`. Но во многих случаях вы можете легко заменить их простыми подвыражениями.

### 5.3.3 Поддерживает ли он Юникод?

#### Ответ

Как использовать Юникод

### 5.3.4 Почему TRegExpr возвращает более одной строки?

Например, регулярное выражение `<font .*>` возвращает первый же `<font`, далее весь последующий текст до финального `</html>`.

#### Ответ

Для обратной совместимости модификатор `/s` по умолчанию Вкл.

Выключите его, и `.` будет соответствовать любому символу, кроме Разделителей строк - именно так, как вы хотите.

Я лично предлагаю `<font ([^\n] *)>`, тогда в `Match [1]` будет URL.

### 5.3.5 Почему TRegExpr возвращает больше, чем я ожидаю?

Например `<p> (. +) </p>` для строки `<p>a </p><p> b </p>` возвращает `a </p><p> b` но не `a`, как ожидается.

#### Ответ

По умолчанию все операторы работают в жадном режиме, поэтому они совпадают как можно больше.

Если вам нужен режим не жадный режим, вы можете использовать не жадные варианты операторов, такие как `+` и т. д., или переключить все операторы в не жадный режим с помощью модификатора `g` (используйте соответствующие свойства TRegExpr или оператор `?(-g)` внутри выражения).

### 5.3.6 Как анализировать HTML, с помощью TRegExpr?

#### Ответ

Извините, ребята, но это почти невозможно!

Конечно, вы можете легко использовать TRegExpr для извлечения некоторой информации из HTML, как показано в моих примерах, но если вам нужен точный синтаксический анализ, вы должны использовать полноценный парсер, а не регулярные выражения

Вы можете прочитать полное объяснение в Том Кристиансен и Натан Торкингтон Perl Cookbook, например.

Вкратце - есть много структур, которые могут быть легко проанализированы реальным парсером, но не могут быть проанализированы регулярными выражениями. Полноценный парсер намного быстрее выполнит синтаксический анализ.

### 5.3.7 Есть ли способ получить несколько совпадений шаблона на TRegExpr?

#### Ответ

Вы искать последующие совпадения с помощью метода `ExecNext`.

Если вам нужен какой-то пример, посмотрите на реализацию метода `TRegExpr.Replace` или на примеры для `HyperLinksDecorator`

### 5.3.8 Я проверяю пользовательский ввод. Почему TRegExpr возвращает True для неправильных входных строк?

#### Ответ

Во многих случаях пользователи TRegExpr забывают, что регулярное выражение предназначено для **поиска** во входной строке.

Так, например, выражение `\d{4,4}` совпадет и с 12345 и с **любые** буквы 1234.

Вы должны проверить от начала строки до конца строки, чтобы убедиться, что вокруг ничего больше нет: `^\d{4,4}$`.

### 5.3.9 Почему не жадные итераторы иногда работают в жадном режиме?

Например, `a+?, b+?`, для строки `aaa,bbb`, найдет `aaa,b`, но не `a,b` хотя первый итератор не жаден?

#### Ответ

Регулярные выражения только ищут первое же совпадение и не пытаются найти «наилучшее» совпадение.

В некоторых случаях это плохо, но в целом это скорее преимущество, чем ограничение, по причинам производительности и предсказуемости.

Основное правило - сначала пытаемся найти соответствие, начиная с текущей позиции в строке и, только если это невозможно, продвигаемся на один символ вперед и попробуем снова со следующей позиции в тексте.

Если вы используете `a, b+?` то это будет соответствовать `a, b`. В случае `a+?, b+?`, не смотря на не жадный модификатор, все же возможно захватить более одного `a`, поэтому TRegExpr сделает это.

Регулярные выражения, не пытаются двигаться дальше по тексту и проверять - удастся ли найти «лучшее» совпадение. Хотя бы потому, что нельзя сказать, что такое «лучше».

### 5.3.10 Как использовать TRegExpr с Borland C ++ Builder?

У меня проблема, нет файла заголовка (`.h` или `.hpp`).

#### Ответ

- Добавьте `RegExpr.pas` к проекту `bcb`.
- Скомпилировать проект. Это создает заголовочный файл `RegExpr.hpp`.
- Теперь вы можете писать код, использующий модуль `RegExpr`.
- Не забудьте добавить `#include 'RegExpr.hpp'` там, где это необходимо.
- Не забудьте заменить все `\` в регулярных выражениях на `\\` или переопределить `EscChar` const.

### 5.3.11 Почему многие примеры (включая примеры из документации) работают неправильно в Borland C ++ Builder?

#### Ответ

Подсказка есть в предыдущем вопросе;) Символ \ имеет особое значение в C ++, поэтому вы должны эскейпить его (как описано в предыдущем ответе). Но если вам не нравится, как выглядит \w+\w+\. \w+, вы можете переопределить константу EscChar (в RegExpr.pas). Скажем, EscChar = "/". Затем вы можете написать /w+/w+/. /W+ - выглядит необычно, но более читабельно.

	English	Русский	Deutsch	Български	Français	Español
--	---------	---------	---------	-----------	----------	---------

## 5.4 Demos

Демо-код для TRegExpr

### 5.4.1 Вступление

Если вы не знакомы с регулярными выражениями, посмотрите на синтаксис регулярных выражений. Интерфейс TRegExpr описан в TRegExpr.

### 5.4.2 Text2HTML

Text2HTML исходники

Преобразует текст в HTML

Использует блок HyperLinksDecorator, основанный на TRegExpr.

Этот блок содержит функции для оформления гиперссылок.

Например, замените “ www.masterAndrey.com “ на “ <a href=»http://www.masterAndrey.com»>www.masterAndrey.com</a> “ или filbert@yandex.ru на <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;
```

Значение	Имея в виду
durlProto	Протокол (например, ftp:// или http://)
durlAddr	ТСР-адрес или доменное имя (например, masterAndrey.com)
durlPort	Номер порта, если указан (например, : 8080)
durlPath	Путь к документу (например, index.html)
durlBMark	Закладка (например, “ # mark “)
durlParam	Параметры URL (например, ? ID = 2 & User = 13)

Возвращает введенный текст AText с оформленными гиперссылками.

*AFlags* описывает, какие части гиперссылки должны быть включены в видимую часть ссылки.

Например, если *AFlags* равно `[durlAddr]`, то гиперссылка `www.masterAndrey.com / contacts.htm` будет оформлена `<a href="www.masterAndrey.com/contacts.htm">www.masterAndrey.com</a>`.

### 5.4.3 TRegExprRoutines

Очень простые примеры, см. Комментарии внутри блока

### 5.4.4 TRegExprClass

Чуть более сложные примеры, см. Комментарии внутри блока





Документация доступна на [английском](#) и русском языках.

Есть также старые переводы на немецкий, болгарский, французский и испанский языки. Если вы хотите помочь обновить эти старые переводы, пожалуйста, [свяжитесь со мной](#).

Новые переводы основаны на [GetText](#) и могут быть отредактированы с помощью [transifex.com](#).

Они уже переведены автоматически и нуждаются только в корректуре, и, возможно, копировании каких-то частей из старых переводов.



---

### Благодарности

---

Сообществом предложено и реализовано множество функций TRegExpr.

Я не могу перечислить здесь всех, но я ценю все сообщения об ошибках, предложения функций и вопросы, которые я получаю от вас.

- Alexey Torgashin - основной контрибутор 2019-2020. реализовал именованные группы, не захватывающие группы, заглядывания вперед и назад, обратный поиск
- Guido Muehlwitz - обнаружена и исправлена ошибка в обработке больших строк
- Stephan Klimek - тестирование в CPPB и предложение / реализация многих функций
- Steve Mudford - реализован параметр Offset
- Martin Baur ([www.mindpower.com](http://www.mindpower.com)) - немецкий перевод, полезные предложения
- Yury Finkel - реализовал поддержку UniCode, нашел и исправил некоторые ошибки
- Ralf Junker - Реализованы некоторые функции, много предложений по оптимизации
- Симеон Лилов - болгарский перевод
- Филип Джирсбк и Мэтью Винтер - помогли в реализации не жадного режима
- Kit Eason много примеров для документации
- Juergen Schroth - поиск ошибок и полезные советы
- Martin Ledoux - французский перевод
- Diego Calp, Аргентина - испанский перевод