Вторая редакция статьи. Исправлены ошибки, синтаксис регулярных выражений описан в более распространенных терминах. Огромное спасибо source777 за помощь в подготовке статьи!

Введение

Регулярные выражения (*regular expressions*) — современная технология поиска текстовых фрагментов в электронных документах, соответствующих определенным образцам. **Правила (*rules*)** — это новое название регулярных выражений. Именно так они именуются в последней версиии языка Perl, признанного лидера по работе со строками.

Образец (*pattern*), задающий правило поиска, по-русски также иногда называют «шаблоном», «маской».

Регулярные выражения произвели прорыв в электронной обработке текста в конце XX века. Сейчас регулярные выражения используются многими текстовыми редакторами и утилитами для поиска и изменения текста на основе выбранных правил. Многие языки программирования уже поддерживают регулярные выражения для работы со строками. На моё удивление, в Delphi не оказалось встроенного модуля/компонента для работы с регулярными выражениями. Это существенное упущение разработчиков Delphi. Однако, как известно, свято место пусто не бывает! Итак, знакомьтесь, **TRegExpr** — класс для работы с регулярными выражениями в Delphi. Берём с официального сайта архив с модулем и примерами.

Составные части регулярных выражений

Следует отметить, что в справке к TRegExpr используется отличная от общепринятой терминология. В этой заметке будет использована терминология более распространенная в мире регулярных выражений, аналоги из справки к TRegExpr приведены в скобках.

Итак, в состав регулярного выражения могут входить:

- **Символы**.
- **Символьные классы** (перечни символов + метасимволы - границы слов).
- **Фиксирующие директивы** (метасимволы - разделители строк)
- **Квантификаторы** (метасимволы - повторения).
- **Альтернативы** (метасимволы - варианты).
- **Группировки** (метасимволы - подвыражения).
- **Модификаторы**.

Символы

Любой символ совпадает с самим собой (если только он не относится к метасимволам). Любая последовательность символов совпадает с такой же во входной строке. Также можно использовать escape-последовательности, например:

```
\t    // табуляция
\n    // новая строка
```

Символьные классы

Символьный класс — просто конечный набор символов. Он ограничивается квадратными скобками и содержит перечисление символов, которые можно вместо него подставить. Заменяется он всего на один символ, входящий в это перечисление. Примеры:

```
[абвгде]      // простое перечисление символов
[а-яА-ЯёЁ]    // все русские буквы
[0-9a-z]      // цифры и строчная латиница
[^0-9]        // все символы, кроме цифр
```

- Можно использовать - (дефис) для указания диапазонов символов.
- Если первый символ класса (идущий сразу после открывающейся квардратной скобки) — ^, то это означает символ, который отсутствует в данном перечислении. Фактически префикс ^ инвертирует список; вместо того, чтобы перечислять символы принадлежащие классу, мы перечисляем символы, не входящие в него.
- Некоторые популярные символьные классы имеют короткую запись.

Стандартные символьные классы

Короткая запись	Полная запись	Описание
\s	[\f\t\n\r]	Пробельный символ
\S	[^\f\t\n\r]	Любой символ, кроме пробельного
\d	[0-9]	Цифра
\D	[^0-9]	Любой символ, кроме цифры
\w	[a-zA-Z0-9_]	Латиница, цифра или подчеркивание (_)
\W	[^a-zA-Z0-9_]	Любой символ, кроме латиницы или цифры
.	[^\n\r]	Любой символ, кроме перевода строки
\b		Граница слова
\B		Не граница слова
\A		Начало строки
\Z		Конец строки

Фиксирующие директивы

Фиксирующие директивы — это символы, которые привязывают правило к некоторому признаку. Например, к концу или началу строки. Наиболее часто используемые:

```
^     // начало строки
$     // конец строки
```

Квантификаторы

Показывают, сколько раз может повторяться предыдущий символ (символьный класс, альтернатива и т.д.) Ограничиваются парой фигурных скобок. Примеры:

```
\w{3}             // три латинских буквы или цифры
\d{1, 3}          // одна, две или три цифры
[а-яА-Я]{3, }     // русское слово длинной три символа и больше
```

Квантификаторы

- С одним параметром называются *точными* и указывают точное количество повторений.
- С двумя агрументами называются *конечными* и указывают конечный диапазон, в котором варьируется количество повторений.
- Без второго параметра (но с запятой) называются *бесконечными* и ограничивают количество повторений лишь снизу.
- Некоторые популярные имеют короткую запись.

Короткие записи популярных квантификаторов

Короткая запись	Полная запись	Описание
`*`	`{0, }`	Любое количество
`+`	`{1, }`	Один или более
`?`	`{0, 1}`	Есть или нет

Жадность

Речь пойдёт о жадности среди квантификаторов. Квантификаторам в регулярных выражениях соответствует максимально длинная строка из возможных (они являются **«жадными»**, *greedy*). Это может оказаться значительной проблемой.

Например, часто ожидают, что выражение `<.*>` найдёт в тексте теги HTML. Однако этому выражению соответствует целиком строка:

Википедия — свободная энциклопедия, в которой каждый может изменить или дополнить любую статью

Эту проблему можно решить двумя способами. Первый состоит в том, что в регулярном выражении учитываются символы, не соответствующие желаемому образцу (`[^>]*` для вышеописанного случая). Второй заключается в определении повторителя как **«нежадного»** («ленивого», *lazy*), добавив после него знак вопроса.

Например, выражению `<.*?>` соответствует не вся показанная выше строка, а отдельные теги (выделены цветом):

Википедия — свободная энциклопедия, в которой каждый может изменить или дополнить любую статью

"Жадные" варианты регулярных выражений пытаются захватить как можно большую

часть входного текста, в то время как "не жадные" - как можно меньшую. Например, `b+` как и `b*` примененные к входной строке 'abbbbc' найдут 'bbbb', в то время как `b+?` найдет только 'b', а `b*?` - вообще - пустую строку; `b{2,3}?` найдет 'bb', в то время как `b{2,3}` найдет 'bbb'.

Альтернативы

Нужны, когда необходимо объединить несколько правил в одно. При этом совпадение засчитывается, когда есть совпадение хотя бы с одним правилом. Желательно альтернативы заключать внутрь группировки (круглые скобки). Правила, входящие в вариант, разделяются `|` (вертикальной чертой). Примеры:

```
(жы|шы)                         // или "жы", или "шы"
([a-zA-Z]+|[a-яA-ЯёЁ]+)     // или слово на латинице, или русское
```

В данном примере продемонстрирована альтернатива в группировке. В принципе альтернатива может существовать и вне группировки, но так возникает больше ошибок.

Группировки

Используются, когда необходимо обрабатывать результат частями. Например, при обработке ссылок в HTML-документе удобно отдельно обрабатывать текст ссылки и URL. Группировки заключаются в круглые скобки.

Модификаторы

Модификаторы предназначены для изменения поведения правила. Назначение и примеры - смотри в справке к TRegExpr.

Использование в Delphi

Пример использования

Использовать регулярные выражения в Delphi просто.

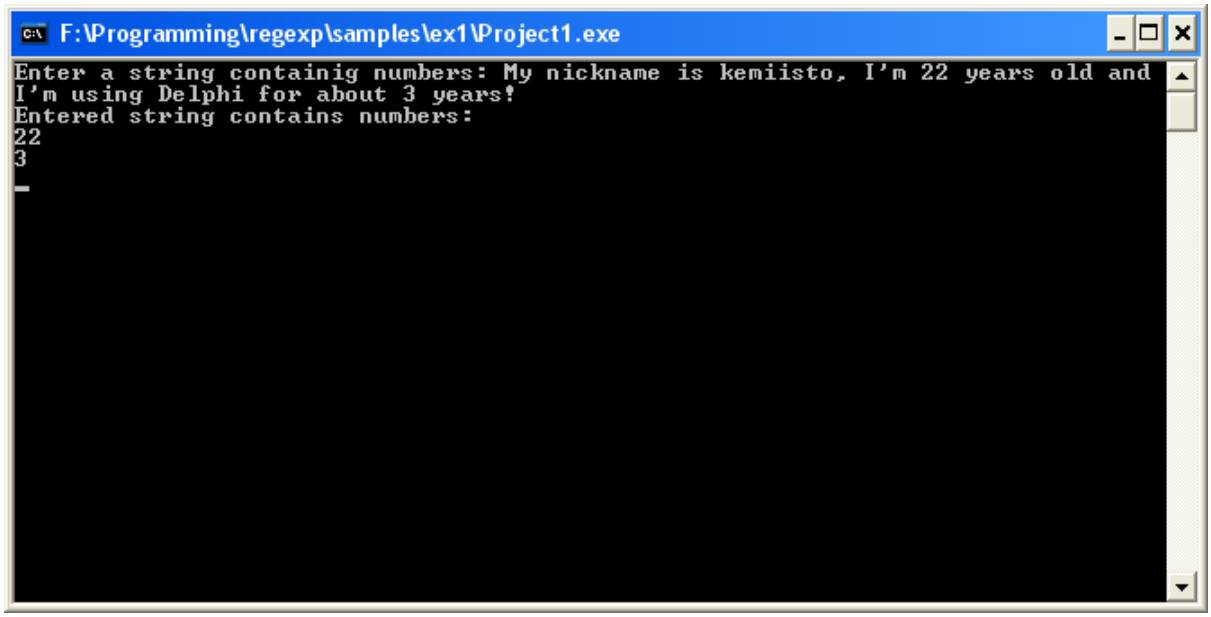
- Распаковываем архив в любой каталог.
- Добавляем RegExpr.pas (размещен в подкаталоге Source) в список файлов нашего проекта (главное меню Delphi Project -> Add to project...)
- Используем класс TRegExpr в нашем проекте. Не забудьте добавить 'uses RegExpr' в соответствующие модули проекта.

Итак, небольшой пример. Попробуем найти в строке все целые числа. Регулярное выражение для этого случая будет выглядеть так: `-?\d+`, т.е. содержать соответствующий символьный класс `\d` (т.е. цифры) и квантификатор `+`, означающий любое количество. Думаю, то, что целое число есть произвольное количество цифр понятно. ;-) Также перед числом возможно будет стоять знак «минус». Остальное должно быть понятно из коментариев.

```
// Очень простой пример - извлечение чисел из введённой строки.
program Project1;
```

```
{$APPTYPE CONSOLE}

uses
  SysUtils,
  // добавляем нужный модуль
  RegExpr in 'RegExpr.pas';

var
  // нам необходим экземпляр класса TRegExpr
  RegExp: TRegExpr;
  s: string;

begin
  // выводим запрос и считываем строку
  Write('Enter a string containing numbers: ');
  Readln(s);

  // создаём объект
  RegExp := TRegExpr.Create;

  // гарантирует освобождение занятой объектом памяти
  try
    // регулярное выражение находится в свойстве Expression
    RegExp.Expression := '-?\d+';
    // ищем первое совпадение с помощью функции
    // Exec(const AInputString : string) : boolean, которая
вернет true,
    // если в строке AInputString будет найдено совпадение с
    // регулярным выражением, хранящимся в свойтве Expression
    if RegExp.Exec(s) then
    // если находим
    begin
      Writeln('Entered string contains numbers: ');
      repeat
        // выводим найденное выражение, которое хранится в
Match[0]
        Writeln(RegExp.Match[0]);
        // и продолжаем поиск
      until not RegExp.ExecNext;
    end
    else
      // иначе - сообщаем, что ничего не нашли
      Writeln('Entered string contains no numbers!');
  finally
    RegExp.Free;
  end;
  Readln;
end.
```

Результат работы программы:



Пример использования вариантов и подвыражений

```
// Чуть более сложный пример - извлечение целой и дробной части числа.
// Регулярное выражение содержит группировки и альтернативу.
program Project1;

{$APPTYPE CONSOLE}

uses
  SysUtils,
  // добавляем нужный модуль
  RegExpr in 'RegExpr.pas';

var
  // нам необходим экземпляр класса TRegExpr
  RegExp: TRegExpr;
  s: string;

begin
  // выводим запрос и считываем строку
  Write('Enter a string containing numbers: ');
  Readln(s);

  // создаём объект
  RegExp := TRegExpr.Create;

  // гарантирует освобождение занятой объектом памяти
  try
    // регулярное выражение находится в свойстве Expression
```

```
    RegExp.Expression := '(\d+)([.,])(\d+)';
    if RegExp.Exec(s) then
    // если находим
    begin
      Writeln('Whole number: ', RegExp.Match[0]);
      Writeln('Integer part: ', RegExp.Match[1]);
      Writeln('Divider : ', RegExp.Match[2]);
      Writeln('Fractional part: ', RegExp.Match[3]);
      Writeln(RegExp.Substitute('$1$2$3'));
    end
    else
    // иначе - сообщаем, что ничего не нашли
      Writeln('Entered string doesn''t contain any number!');
  finally
    RegExp.Free;
  end;
  Readln;
end.
```

Регулярное выражение для этого случая будет выглядеть несколько сложнее: (\d+)([.,])(\d+), т.е. содержать соотв. группировки (целая часть, разделитель и дробная части) и вариант (в качестве разделителя может выступать как точка, так и запятая).

Выражение, соотв. всему регулярному выражению по-прежнему находится в `Match[0]`, а вот `Match[i]` содержит `i`-ую группировку.

Также обратите своё внимание на достаточно интересную функцию

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

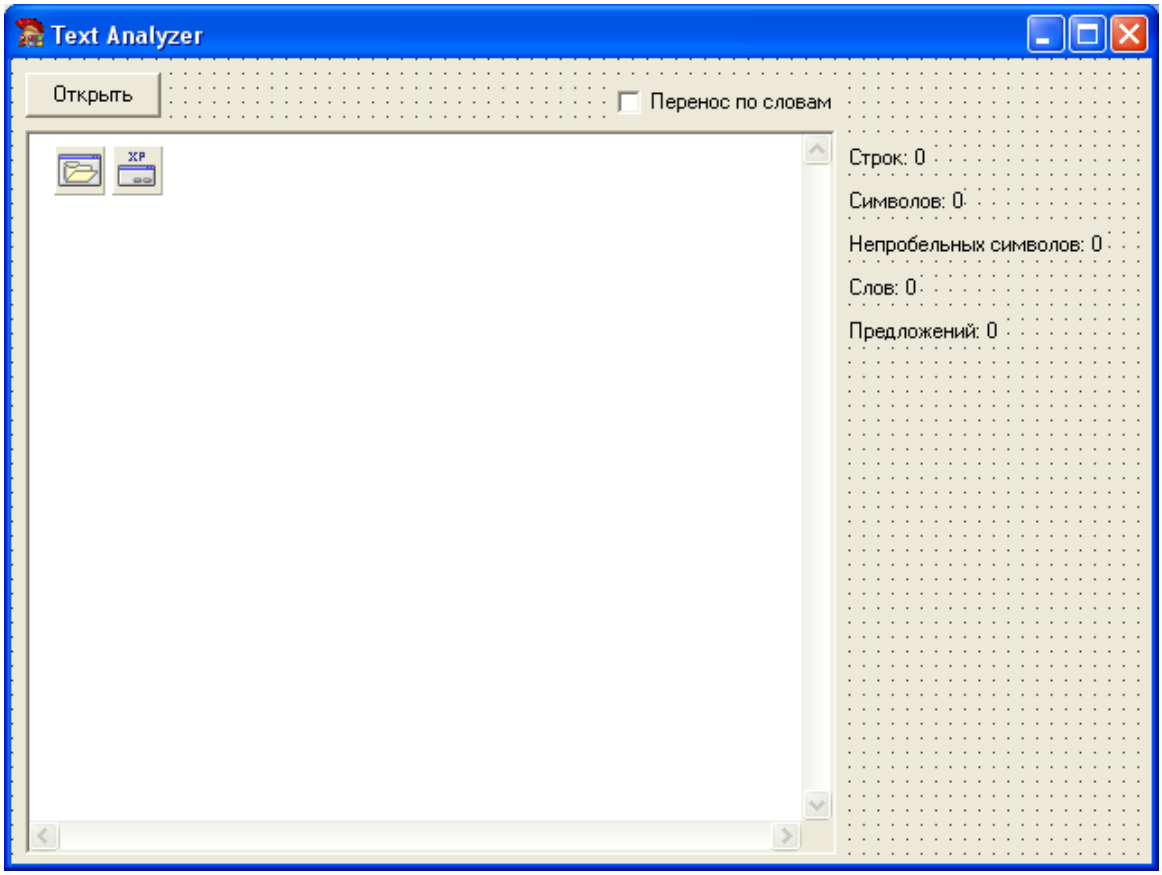
которая возвращает `ATemplate`, в котором все '$&' и '$0' заменены на найденное регулярное выражение, а '$n' на n-ое подвыражение. Например, оператор

```
Writeln(RegExp.Substitute('$1$2$3'));
```

добавленный в предидущий пример, выведет исходное число.

Пример анализа текста

Давайте напишем простенький анализатор текста. На это раз сделаем GUI-приложение. На форме расположим один экземпляр `TButton`, один экземпляр `TMemo` и пять экземпляров `TLabel`.



При нажатии на кнопку, реализуем показ диалога выбора текстового файла и загрузку его в `Memo1`.

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  if OpenDialog1.Execute then
    Memo1.Lines.LoadFromFile(OpenDialog1.FileName);
end;
```

Теперь давайте реализуем сбор статистики по мере изменения текста в `Memo1`:

```
procedure TForm1.Memo1Change(Sender: TObject);
var
  Count, i: Integer;
  RegExp: TRegExpr;
begin
  RegExp := TRegExpr.Create;

  // кол-во строк
  Label1.Caption := 'Строк: ' + IntToStr(Memo1.Lines.Count);

  // кол-во символов
  Label2.Caption := 'Символов: ' + IntToStr(Length(Memo1.Text));
```

```
  // кол-во символов, исключая пробельные
  Count := 0;
  RegExp.Expression := '\s';
  Count := Count + Length(RegExp.Replace(Memo1.Text, '',
False));
  Label3.Caption := 'Непробельных символов: ' + IntToStr(Count);

  // кол-во слов
  Count := 0;
  RegExp.Expression := '\s*[^\s.-]+-?[^\s.-]*';
  if RegExp.Exec(Memo1.Text) then
  repeat
    Count := Count + 1;
  until not RegExp.ExecNext;
  Label4.Caption := 'Слов: ' + IntToStr(Count);

  // кол-во предложений
  Count := 0;
  RegExp.Expression := '[.!?]+(\s|$)';
  if RegExp.Exec(Memo1.Text) then
  repeat
    Count := Count + 1;
  until not RegExp.ExecNext;
  Label5.Caption := 'Предложений: ' + IntToStr(Count);
end;
```

Во-первых, остановимся на подсчёте количества символов, исключая пробельные. Здесь был использован метод

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

заменяющий в `AInputStr` все вхождения регулярного выражения на `AReplaceStr`. Вычисляя сумму длин полученных таким образом строк, получим искомую величину.

Теперь взглянем на подсчёт количества предложений в нашем примере. Ответом на вопрос «Что надо найти?», будет, скорее всего, «Количество точек, знаков восклицания и знаков вопроса, стоящих в конце слова или строки.» Кроме того, следует учитывать возможность наличия неединичных знаков препинания (…, !?). Составляем регулярное выражение '[.!?]+(\s|$)', и собственно всё! Задача решена!

При подсчёте количества слов можно, например, использовать регулярное выражение вида '\s*[^\s.-]+-?[^\s.-]*'. Слово может следовать за пробелом, а может и нет (если стоит в начале текста). Поэтому вначале нашеговыражения стоит '\s*'. Собственно слово (написанное без орфографических ошибок) — последовательность непробельных символов. То есть мы могли бы записать регулярное вражение так: '\s*[\S]'. Но под такой шаблон попадут и некоторые другие символы и их последовательности, которые врядли можно назвать словами — многоточие, тире. Чтобы исправить ситуацию мы пишем '\s*[^\s.-]'. Ну и, наконец, необходимо учесть наличие слов, которые пишутся через дефис и получим '\s*[^\s.-]+-?[^\s.-]*'.

Важно понимать, что основное преимущество при использовании регулярных выражений заключается в экономии времени! Вам необходимо лишь задаться вопросом «Что надо найти?» и составить регулярное выражение, которое является ответом на этот вопрос, а не разрабатывать «с нуля» алгоритм поиска.

Успехов! ;-)