Регулярные выражения
Коротко

Работа с текстом — это то, в чем Perl превосходит другие программы. Регулярные выражения обеспечивают значительную часть возможностей Perl по обработке текстов. Они позволяют сопоставлять текст с указанным шаблоном (а именно, сравнивать две строки с помощью универсальных символов, интерпретируемых специальным образом) и выполнять замену текста. Таким образом, Perl предоставляет мощный инструмент для манипулирования текстом под управлением программы пользователя.
С другой стороны, не подлежит сомнению, что регулярные выражения Perl — это одна из тех областей, которые требуют серьезных усилий со стороны программиста. Зачастую требуется время для того, чтобы разобраться в том, что делают даже относительно прямолинейные и однозначные конструкции. Например, следующее регулярное выражение осуществляет поиск в тексте маркеров HTML **<A>** и **<IMG>** , а также текста, заключенного внутри разметки вплоть до концевых маркеров </A> и </IMG>):

```
Stext = "<A>Here is an anchor.</A>";
if ($text =~ /<([IM6|A])>[\w\s\.]+<\/\1>/i)
{
print "Found an image or anchor tag."; }
```

Found an image or anchor tag
Это наиболее сокровенная область языка Perl. А потому я постараюсь внести в этот вопрос максимальную ясность[1].

Использование регулярных выражений

В Perl имеются три основных оператора, работающих со строками:
- m/.../ — проверка совпадений (matching),
- s/.../.../ — подстановка текста (substitution),
- tr/.../.../ — замена текста (translation).

Оператор т/.../ анализирует входной текст и ищет в нем подстроку, совпадающую с указанным шаблоном (он задан регулярным выражением). Оператор s/.../.../ выполняет подстановку одних текстовых фрагментов вместо других, используя для этой цели регулярные выражения. Оператор tr/.../.../ также изменяет входной текст, но при этом он не использует регулярные выражения, осуществляя замену посимвольно.

Оператор проверки совпадений m/…/

Оператор m/.../ пытается сопоставить шаблон, указанный в качестве аргумента, и заданный текст (по умолчанию текст берется из переменной Perl $__). В приведенном ниже примере мы ищем во вводимом пользователем тексте строку **exit** (модификатор i после второй наклонной черты делает проверку нечувствительной к регистру):

```
while (<>)
      {if (m/exit/i) {exit;} }
```

Вместо того чтобы использовать переменную $__, можно задать источник проверяемого текста с помощью оператора =~. В нашем примере для этой цели используется переменная **$line** (данный код не меняет содержимого переменной **$line,** хотя оператор =~ и напоминает символ присвоения):

```
while (Sline = <>)
      { if (Sline =~ m/exit/i)  {exit;}  }
```

---

[1] Тем не менее, необходимо отметить, что материал, изложенный здесь, является лишь введением в эту важную область — регулярным выражениям и особенностям работы с ними посвящены целые книги, подробно рассматривающие специальные приемы, трюки и готовые решения. — *Примеч. ред.*

Смысл сравнения можно изменить на противоположный, если вместо оператора =~ использовать оператор !~:

```
while ($line = <>)
      { if ($line !~ /exit/i)  {} else {exit;} }
```

Поскольку в Perl оператор m/.../ используется очень часто, можно использовать его сокращенную форму, опустив начальную букву m. Если же начальная буква m присутствует, то вместо символов наклонной черты (слэша) в качестве ограничителей можно использовать, за редким исключением, любой другой символ (см. далее описание оператора подстановки s/.../.../):

```
while (Sline = <>) {
      if ($1ine =~ /exit/i ) {exit;}
      if ($line =~ m|quit|i) {exit;}
      if ($line =~ m%stop%i) {exit;}
}
```

Подсказка. *Если шаблон содержит символы косой черты, что зачастую встречается, например, при анализе имен файлов с указанием пути и/или меток HTML, то стоит отказаться от использования этих символов в качестве ограничителя. Это позволит не ставить обратную косую черту перед каждым символом косой черты внутри шаблона и сохранит его прозрачность.*

Оператор подстановки s/.../.../

Оператор s/.../.../ выполняет замену одних фрагментов текста на другие. Например, в следующем случае мы заменяем подстроку **young** на подстроку old:

```
$text = "Pretty young."; $text =~ s/young/old/;
print $text; Pretty old.
```

По умолчанию оператор замены работает с переменной Perl $__. Как и в случае оператора m/.../, косую черту использовать не обязательно — годится любой символ, который не вступает в противоречие с заданным выражением. Например, вместо косой черты можно использовать |:

```
$text = "Pretty young."; $text =~ s|young|old|;
print $text; Pretty old.
```

либо просто поставить в качестве ограничителя обычные скобки:

```
$text = "Pretty young."; $text =~ s(young)(old);
print $text; Pretty old.
```

Более подробно о форме записей команд m/.../ и s/.../.../ рассказывается далее в разделе «Особенности работы команд m/.../ и s/.../.../».

Подсказка. *Старайтесь не применять в качестве ограничителей вопросительный знак (?) и апостроф (') — шаблоны, ограниченные этими символами, обрабатываются иначе, чем обычные (см. раздел «Особенности работы команд m/.../ и s/.../.../» далее).*

Обратите внимание, что операторы s/.../.../ и m/.../ ведут поиск с первого символа текстовой строки до первого совпадения. Если оно найдено, без специального указания поиск не продолжается:

```
$text = "Pretty young,   but not very young.";
$text =~ s/young/old/;
print $text;
Pretty old, but not very young.
```

Оператор замены tr/.../.../

Кроме операторов /.../ и s/.../.../ для работы со строками в Perl имеется оператор tr/.../.../. Он также выполняет замену одних фрагментов текста на другие, однако в отличие от s/.../.../, не пытается обрабатывать регулярные выражения, подставляя текст один к одному. В следующем примере мы заменяем с его помощью букву «о» на букву «i»:

```
$text = "His name is Tom."; $text =~ tr/o/i/;
print Stext; His name is Tim.
```

Подсказка. *В Perl операторы tr/.../.../ и y/.../.../ выполняют одинаковые действия. Точнее, tr и y — это два имени одного и того же оператора.*

Итак, мы перечислили операторы, с которыми будем работать в этой главе. Наше введение, однако, лишь едва затронуло рассматриваемую тему. Теперь наступает время изучить создание регулярных выражений всерьез, что позволит полноценно работать с поиском и заменой строк.

Непосредственные решения

Создание регулярных выражений

Регулярные выражения — основа работы с операторами m/.../ и s/.../.../, так как они передаются последним в качестве аргументов. Разберемся, как устроено регулярное выражение **\b([A-Za-z]+)\b,** осуществляющее поиск отдельных слов в строке:

```
$text = "Perl is the subject.";
Stext =~ /\b([A-Za-z]+)\b/;
print $1;
Perl
```

Выражение **\b([A-Za-z]+)\b** включает в себя группирующие метасимволы (и),метасимвол границы слова \b, класс всех латинских букв **[A-Za-z]** (он объединяет заглавные и строчные буквы) и квантификатор +, который указывает на то, что требуется найти один или несколько символов рассматриваемого класса. Поскольку регулярные выражения, как это было в предыдущем примере, могут быть очень сложными, в этой главе они разбираются по частям. В общем случае регулярное выражение состоит из следующих компонентов:

• одиночные символы (characters),
• классы символов (character classes),
• альтернативные шаблоны (alternative match patterns),
• квантификаторы (quantifiers),
• мнимые символы (assertions),
• ссылки на найденный текст (backreferences),
• дополнительные конструкции (regular expression extensions).

Каждый из этих элементов достоин особого изучения. Их обсуждению посвящено несколько следующих разделов.

Одиночные символы в регулярных выражениях

В регулярном выражении любой символ соответствует самому себе, если только он не является метасимволом со специальным значением (такими метасимволами являются \, |, (,), [, {, *, +, ^, $, ? и .). В следующем примере проверяется, не ввел ли пользователь команду «quit» (и если это так, то прекращаем работу программы):

```
while {<>}   {
      if (m/quit/)  {exit;} }
```

Правильнее проверить, что введенное пользователем слово «quit» не имеет соседних слов, изменяющих смысл предложения. (Например, программа выполнит заведомо неверное действие, если вместо «quit» пользователь введет команду «Don't quit!».) Это можно сделать с помощью метасимволов ^ и $. Заодно, чтобы сравнение было нечувствительно к разнице между прописными и заглавными буквами, используем модификатор i:

```
while  {<>}
      { if (m/~quit$/i) {exit;} }
```

(О работе метасимволов ^ и $ рассказывается в разделе «Мнимые символы в регулярных выражениях». О модификаторе i можно подробнее узнать в разделе «Модификаторы команд m/.../ и s/.../.../».)

Кроме обычных символов Perl определяет специальные символы. Они вводятся с помощью обратной косой черты (escape-последовательности) и также могут встречаться в регулярном выражении:

- \077 — восьмеричный символ,
- \a — символ BEL (звонок),
- \c[— управляющие символы (комбинация Ctrl + *символ,* в данном случае это управляющий символ ESC),
- \d — соответствует цифре,
- \D — соответствует любому символу, кроме цифры,
- \e — символ escape (ESC),
- \E — конец действия команд \L, \U и \Q,
- \f — символ прогона страницы (FF),
- \l — следующая литера становится строчной (lowercase),
- \L — все последующие литеры становятся строчными вплоть до команды \E,
- \n — символ новой строки (LF, NL),

- \Q — вплоть до команды \E все последующие метасимволы становятся обычными символами,
- \r — символ перевода каретки (CR),
- \s — соответствует любому из «пробельных символов» (пробел, вертикальная или горизонтальная табуляция, символ новой строки и т. д.),
- \S — любой символ, кроме «пробельного»,
- \t — символ горизонтальной табуляции (HT, TAB),
- \u — следующая литера становится заглавной (uppercase),
- \U — все последующие литеры становятся заглавными вплоть до команды \E,
- \v — символ вертикальной табуляции (VT),
- \w — алфавитно-цифровой символ (любая буква, цифра или символ подчеркивания),
- \W — любой символ, кроме букв, цифр и символа подчеркивания,
- \x1B — шестнадцатеричный символ.

Вы также можете «защитить» любой метасимвол, то есть заставить Perl рассматривать его как обыкновенный символ, а не как команду, поставив перед метасимволом обратную косую черту \. Обратите внимание на символы типа \w, \d и \s, которые соответствуют не одному, а любому символу из некоторой группы. Также заметьте, что один такой символ, указанный в шаблоне, соответствует ровно одному символу проверяемой строки. Поэтому для задания шаблона, соответствующего, например, слову из букв, цифр и символов подчеркивания, надо использовать конструкцию \w+, как это сделано в следующем примере:

```
$text = "Here is some text."; text =~ s/\w*/There/;
print $text;
```

There is some text.

(Знак + означает «один или более символов, соответствующих шаблону». Более подробно о нем рассказано в разделе «Квантификаторы в регулярных выражениях».)

Совпадение с любым символом

В Perl имеется еще один мощный символ — а именно, точка (.). В шаблоне он соответствует любому знаку, кроме символа новой строки. Например, следующая / команда заменяет в строке все символы на звездочки (использован модификатор g, обеспечивающий глобальную замену):

```
$text = "Now is the time,";
$text =~ s/./*/g;
print $text;
******************
```

А что делать, если требуется проверить совпадение именно с точкой? Символы вроде точки (конкретно, \|()[{^A$*+?.)> играющие в регулярном выражении особую роль) называются, как уже было сказано выше, *метасимволами,* и если вы хотите, чтобы они внутри шаблона интерпретировались как обычные символы, метасимволу должна предшествовать обратная косая черта. Точно так же обратная косая черта предшествует символу, используемому в качестве ограничителя для команды m/.../, s/…/.../ или tr/.../.../, если он встречается внутри шаблона и не должен рассматриваться как ограничитель. Рассмотрим пример:

```
$line = ".Hello!";
if ($line =~ m/^\./) {
      print "Shouldn't start a sentence with a period!\n"; }
```

Shouldn't start a sentence with a period!

Классы символов в регулярных выражениях

Символы могут быть сгруппированы в классы. Указанный в шаблоне класс символов сопоставляется с любым из символов, входящим в этот класс. Класс — это список символов, заключенный в квадратные скобки [и]. Можно указывать как отдельные символы, так и их диапазон (диапазон задается двумя крайними символами, соединенными тире).

Например, следующий код производит поиск гласных:

```
$text = "Here is the text.";
if ($text =~ /[aeiou]/) {
      print "Vowels:  we got 'em.\n";
}
```

Vowels: we got 'em.

Другой пример: с помощью шаблона **[A-Za-z]+** (метасимвол + означает утверждение: «один или более таких символов» — см. далее раздел «Квантификаторы в регулярных выражениях») ищется и заменяется первое слово:

```
$text = "What is the subject.";
$text =~ s/[A-Za-z]+/Perl/;
print $text;
```

Perl is the subject.

Подсказка. *Если требуется задать минус как символ, входящий в класс символов, перед ним надо поставить обратную косую черту (\-).*

Если сразу после открывающей квадратной скобки стоит символ ^, то смысл меняется на противоположный. А именно, этот класс сопоставляется любому символу, *кроме* перечисленных в квадратных скобках. В следующем примере производится замена фрагмента текста, составленного не из букв и не из пробелов:

```
$text - "Perl is the subject on page 493 of the book.";
$text =~ s/[A-Za-z\s]+/500/;
print $text;
```

Perl is the subject on page 500 of the book.

Альтернативные шаблоны в регулярных выражениях

Вы можете задать несколько альтернативных шаблонов, используя символ | как разделитель. Альтернативные шаблоны позволяют превратить процедуру поиска из однонаправленного процесса в разветвленный: если не подходит один шаблон, Perl подставляет другой и повторяет сравнение, и так до тех пор, пока не иссякнут все возможные альтернативные комбинации. Например, следующий фрагмент проверяет, не ввел ли пользователь «exit», «quit» или «stop»:

```
while (<>)  {
      if (m/exit|quit|stop/) {exit;}
 }
```

Чтобы было ясно, где начинается и где заканчивается набор альтернативных шаблонов, их заключают в круглые скобки — иначе символы, расположенные справа и слева от группы шаблонов, могут смешаться с альтернативными шаблонами.

В следующем примере метасимволы ^ и $ обозначают начало и конец строки (см. раздел «Мнимые символы в регулярных выражениях») и отделяются от набора альтернативных шаблонов с помощью скобок:

```
while (<>)
      { if (m/^(exit|quit|stop)$/) {exit;}  }
```

Альтернативные варианты перебираются слева направо. Как только найдена первая альтернатива, для которой выполняется совпадение с шаблоном, перебор прекращается.

Подсказка 1. *Участки шаблона, заключенные в круглые скобки, выполняют специальную роль при выполнении операций поиска и замены. Об этой особенности круглых скобок рассказывается в разделах «Ссылки на найденный текст» и «Особенности работы команд* m/.../ *и* s/.../…/».

Подсказка 2. *Если символ \ находится в квадратных скобках, он интерпретируется как обычный символ. Поэтому если вы используете конструкцию шаблона вида* ***[Tim | Tom | Tam],*** *то она будет эквивалентна классу символов* ***[Tioam|].*** *Точно так же большинство других метасимволов и команд, специфичных для регулярных выражений — в частности, квантификаторы и мнимые символы, описанные в двух последующих разделах, — внутри квадратных скобок превращаются в обычные символы или escape-последовательности текстовых строк.*

Квантификаторы в регулярных выражениях

Квантификаторы указывают на то, что тот или иной шаблон в строке может повторяться определенное количество раз. Например, можно использовать квантификатор + для поиска мест неоднократного повторения подряд латинской буквы e и их замены на одиночную букву e:

```
$text = "Hello from Peeeeeeeeeeeeeeerl.";
$text =~ s/e+/e/;
print $text;
```

Hello from Perl.

Квантификатор + соответствует фразе «один или несколько». Перечислим все доступные в Perl квантификаторы:

• * — ноль или несколько совпадений,

• + — одно или несколько совпадений,
• ? — ноль совпадений или одно совпадение,
• *{n}* — ровно *n* совпадений,
• *{n,}* — по крайней мере *n* совпадений,
• ***{n,m}*** — от *n* до *m* совпадений.
Например, вот как проверить, что пользователь ввел не менее двадцати символов:

```
while (<>){
      if (!m/.{20,}/)
            {print "Please type longer lines!\n";}
}
```

Подсказка. *Квантификатор действует только на предшествующий ему элемент шаблона. Например, конструкция* ***\d[a-z]+*** *будет соответствовать последовательности из одной или нескольких строчных латинских букв, начинающейся с цифры, а не последовательности, составленной из чередующихся цифр и букв. Чтобы выделить группу элементов, на которую действует квантификатор, нужно использовать круглые скобки:* ***(\d[a-z]) +.***

«Жадность» квантификаторов

Учтите, что квантификаторы количества по умолчанию являются «жадными», то есть возвращают самый длинный фрагмент текста, соответствующий указанному шаблону, начиная с текущей позиции строки. Например, вы хотите заменить фразу «That is some text, isn't it?» на «That's some text, isn't it?», подставив «That's» вместо «That is». Посмотрим, что получится, если использовать команду

```
$text = "That is some text,   isn't it?";
$text =~ s/. *is/That's/;
print $text;
```

В силу «жадности» квантификатора * конструкция **.*is** будет сопоставлена максимально возможному фрагменту текста. То есть Perl соотнесет с ней все символы, предшествующие последнему «is» (включая и сам «is»). В результате выполнения команды получится:
That'sn't it?
Проблеме, как заставить квантификаторы количества быть менее жадными, посвящен отдельный раздел этой главы (см. далее раздел «Как ограничить «жадность» квантификаторов»).
Регулярные выражения, использующие квантификаторы, могут порождать процесс, который называется *перебор с возвратом* (backtracking). Чтобы произошло совпадение текста с шаблоном, надо построить соответствие между текстом и *всем* регулярным выражением, а не его *частью.* Начало шаблона может содержать квантификатор, который поначалу срабатывает, но впоследствии приводит к тому, что для части шаблона не хватает текста или возникает несоответствие между текстом и шаблоном. В таких случаях Perl возвращается назад и начинает построение соответствия между текстом и шаблоном с самого начала, ограничивая «жадность» квантификатора (именно поэтому процесс и называется «перебор с возвратом»).

Мнимые символы в регулярных выражениях

В Perl имеются символы (метасимволы), которые соответствуют не какой-либо литере или литерам, а означают выполнение определенного условия (поэтому в английском языке их называют assertions, или *утверждениями).* Их можно рассматривать как мнимые символы нулевого размера, расположенные на границе между реальными символами в точке, соответствующей определенному условию:
• ^ — начало строки текста,
• $ - конец строки или позиция перед символом начала новой строки, расположенного в конце,
• \b — граница слова,
• \B — отсутствие границы слова,
• \A — «истинное» начало строки,
• \Z — «истинный» конец строки или позиция перед символом начала новой строки, расположенного в «истинном» конце строки,
• \z — «истинный» конец строки,
• \G — граница, на которой остановился предыдущий глобальный поиск, выполняемый командой m/.../g,
• (?= *шаблон)* — после этой точки есть фрагмент текста, который соответствует указанному регулярному выражению,

• (?! *шаблон)* — после этой точки *нет* текста, который бы соответствовал указанному регулярному выражению,
• (?<= *шаблон)* — *перед* этой точкой есть фрагмент текста, соответствующий указанному регулярному выражению,
• (?<! *шаблон)* — перед этой точкой *нет* фрагмента текста, соответствующего указанному регулярному выражению.

Подсказка. *Утверждения вида* (?= ...), *(?! ...), (?<= ...) и (?<! ...) рассматриваются более детально далее в этой главе в разделах «Дополнительные конструкции в регулярных выражениях» и «Утверждения, проверяющие текст до и после шаблона».*

Например, вот как выполнить поиск и замену слова, используя метасимволы границы слов:

```
$text = "Here is some text.";
$text =~ s/\b([A-Za-z]+)\b/There/;
print $text;
```

There is some text.

(Perl считает границей слова точку, расположенную между \w и \W, независимо от того, в каком порядке следуют эти символы.)

В следующем примере выводится сообщение о том, что пользователь ввел слово «yes», при условии, что оно единственное, что ввел пользователь. Для этого шаблон включает мнимые символы начала и конца строки:

```
while (<>) {
if (m/^yes$/) {
      print "Thank you for being agreeable. \n";}
}
```

Приведенный выше пример требует комментария. Прежде всего, бросается в глаза наличие двух групп метасимволов для начала и конца строки. В большинстве случаев они означают одно и то же, так как обычно символы новой строки (то есть \n), встречающиеся внутри текстового выражения, не рассматриваются как вложенные строки. Однако если для команды m/.../ или s/.../.../ указан модификатор m (см. далее раздел «Модификаторы команд m/.../ и s/.. ./.../»), то текстовое выражение будет рассматриваться как многострочный текст, в котором границами строк выступают символы новой строки \n.

В случае многострочного текста метасимвол ^ сопоставляется с позицией после *любого* символа новой строки, а не только с началом текстового выражения. Точно также метасимвол $ — это позиция перед *любым* символом новой строки, расположенным внутри текстового выражения, а не обязательно конец текстового выражения или же позиция перед концевым символом \n. Однако метасимвол \A — начало текстового выражения, а метасимвол \Z — конец текстового выражения или позиция перед концевым символом \n, даже если в текстовом выражении имеются вложенные символы \n и при выполнении операции поиска или замены указан модификатор т.

Подсказка. *Метасимвол «точка» (.) соответствует любому символу, кроме символа новой строки \n. Независимо от того, задан ли модификатор т, она не будет сопоставляться ни с внутренними, ни с концевыми символами \n. Единственный способ заставить точку рассматривать \n как обычный символ — использовать модификатор s (см. раздел «Модификаторы команд т/.../ и s/.../.../» далее).*

Отсюда понятна разница между метасимволами \Z и \z. Если в качестве текстового выражения используется результат чтения входного потока данных, то с большой вероятностью данное выражение заканчивается символом \n, за исключением того варианта, когда программа предусмотрительно «отщипнула» его с помощью функции **chop** или **chomp.** Метасимвол \Z игнорирует концевой символ \n, если он случайно остался на месте, рассматривая обе ситуации как «конец строки». В отличие от него метасимвол \z оказывается более пунктуальным и рассматривает концевой символ \n как неотъемлемую часть проверяемого текстового выражения, если только пользователь не позаботился об удалении этого символа.

Отдельно следует остановиться на метасимволе \G. Он может указываться в регулярном выражении только в том случае, если выполняется глобальный поиск (то есть если команда m/.../ имеет модификатор g — см. раздел «Модификаторы команд m/.../ и s/.../.../»). Метасимвол \G, указанный в шаблоне, соответствует точке, на которой остановилась предыдущая операция поиска. Более подробно о работе алгоритма поиска при наличии модификатора g и метасимвола \G рассказывается далее в разделе «Особенности работы команд m/.../ и s/.../.../».

Предупреждение. *В текущей версии Perl метасимвол \G эквивалентен метасимволу \A, если при выполнении операции поиска не указан модификатор g. Однако это — случайное и недокументированное свойство, которое легко может измениться в будущем.*

Ссылки на найденный текст

Иногда нужно сослаться на подстроку текста, для которой получено совпадение с некоторой частью шаблона. Например, при обработке файла HTML может потребоваться выделять фрагменты текста, ограниченные открывающими и закрывающими метками HTML (например, <A> и </A>). В начале этой главы уже приводился пример, в котором выделялся текст, ограниченный метками HTML **<A>** и **<IMG>.** Следующий пример позволяет выделять текст, расположенный между любыми правильно закрытыми метками:

```
$text = "<A>Here is an anchor.</A>";
if ($text =~ m%<([A-Za-z]+)>[\w\s\.]+</\1>%i)
      { print "HTML tag with some text inside it is found."; }
```

HTML tag with some text inside it is found.

(Обратите внимание, что вместо косой черты в качестве ограничителя шаблона использован другой символ. Это позволяет использовать символ косой черты внутри шаблона без предшествующей ему обратной косой черты.)

Каждому фрагменту шаблона, заключенному в круглые скобки, соответствует определенная внутренняя переменная. Переменные пронумерованы, так что на них можно ссылаться внутри шаблона, поставив перед номером обратную косую черту (\1, \2, \3, ...). На значения переменных можно ссылаться внутри шаблона, как на обычный текст, поэтому </\i> соответствует </A>, если открывающей меткой служит **<A>,** и **</IMG>,** если открывающей меткой служит **</IMG>.**

Эти же самые внутренние переменные можно использовать и вне шаблона, ссылаясь на них как на скаляры с именами $1, $2, $3, ..., *$n:*

```
$text = "I have 4 apples.";
if ($text =~ /(\d+)/) {
      print "Here is the number of apples: $1.\n";
}
```

Here is the number of apples: 4.

Каждой паре скобок внутри шаблона после завершения операции поиска будет соответствовать скалярная переменная с соответствующим номером. Это можно использовать при выделении нужных для последующей работы фрагментов анализируемой строки. В следующем примере мы изменяем порядок трех слов в текстовой строке с помощью команды s/.../.../:

```
$text = "I see you.";
$text =~ s/~(\w+)*(\w*)*(\w*)/$3 $2 $1/;
print $text;
you see I.
```

Переменные, соответствующие фрагментам шаблона, нумеруются слева направо с учетом вложенности скобок. Например, после следующей операции поиска будут проинициализированы шесть переменных, соответствующих шести парам скобок:

```
$text - "ABCDEFGH";
$text =~ m/(\w(\w)(\w))((\w)(\w))/;
print "$1/$2/$3/$4/$5/$6/"; ABC/B/C/DE/D/E
```

Подсказка 1. *Кроме переменных, ссылающихся на найденный текст, можно использовать специальные переменные Perl. Так, $& содержит найденное совпадение (то есть фрагмент текста, для которого найдено соответствие между шаблоном и текстом при последней операции поиска или замены), $` содержит текст перед найденный совпадением, $' - текст после найденного совпадения,* $+ *— совпадение для обработанного последним фрагмента шаблона, заключенного в круглые скобки (если у шаблона нет фрагментов, заключенных в круглые скобки, она получает неопределенное значение).*

Подсказка 2. *Пары скобок, используемые в синтаксических конструкциях вида (?- ...), (?! ...), (?<= ...) и т. д. (см. раздел «Дополнительные конструкции в регулярных выражениях»), не порождают нумерованных переменных.*

Дополнительные конструкции в регулярных выражениях

В регулярных выражениях Perl версии 5 имеются дополнительные синтаксические конструкции, использующие скобки в комбинации с вопросительным знаком. Имеют смысл следующие конструкции:

- *(?#текст)* — комментарий. Текст комментария игнорируется.

• *(?шаблон)* или *(?модификаторы:шаблон)* — группирует элементы шаблона. В отличие от обычных круглых скобок, не создает нумерованной переменной. Модификаторы, которые можно указывать в этой конструкции, описаны далее в разделе «Модификаторы команд m/.../ и s/.../.../». Например, модификатор i не будет делать различия между строчными и заглавными буквами, однако область действия этого модификатора будет ограничена только указанным шаблоном (см. также далее конструкцию *(?имя))*.
• *(?=шаблон)* — «заглядывание вперед». Требует, чтобы после текущей точки находился текст, соответствующий данному шаблону. Такая конструкция обрабатывается как условие или мнимый символ, поскольку не включается в результат поиска. Например, поиск с помощью команды /w+(?=\s+)/ найдет слово, за которым следуют один или несколько «пробельных символов», однако сами они в результат не войдут.
• *(?!шаблон)* — случай, противоположный предыдущему. После текущей точки *не должно быть* текста, соотносимого с заданным шаблоном. Так, если шаблон w+(?=\s) — это слово, за которым следует «пробельный символ», то шаблон w+(?!\s) — это слово, за которым *нет* «пробельного символа».
• *(?<=шаблон)* — «заглядывание назад». Требует, чтобы *перед* текущей точкой находился соответствующий текст. Так, шаблон (?<=\s)w+ интерпретируется как слово, *перед* которым имеется «пробельный символ» (в отличие от «заглядывания вперед», «заглядывание назад» может работать только с фиксированным числом проверяемых символов).
• *(?<!шаблон)* — отрицание предыдущего условия. Перед текущей точкой *не должно быть* текста, соотносимого с заданным шаблоном. Соответственно, от команды /(?<!\s)w+/ требуется найти слово, перед которым *нет* «пробельного символа».
• *(?{код})* — условие (мнимый символ), которое всегда выполняется. Сводится к выполнению команд Perl в фигурных скобках. Вы можете использовать эту конструкцию, только если в начале сценария указана команда use re 'eval'. При последовательном соотнесении текста и шаблона, когда Perl доходит до такой конструкции, выполняется указанный код. Если полного соответствия для оставшихся элементов найти не удалось, то при возврате левее данной точки шаблона вычисления, проделанные с локальными переменными, откатываются назад. (Условие является экспериментальным. В документации, прилагаемой к Perl, можно найти довольно детальное рассмотрение (с примерами) работы этого условия и возможных трудностей в случае его применения.)
• *(?>шаблон)* — «независимый» или «автономный» шаблон. Используется для оптимизации процесса поиска, поскольку запрещает «поиск с возвратом». Такая конструкция соответствует подстроке, на которую налагается заданный шаблон, если его закрепить в текущей точке без учета последующих элементов шаблона. Например, шаблон **(?>a*)ab** в отличие от a*ab не может соответствовать никакой строке. Если поставить в любом месте шаблон a*, он «съест» все буквы a, не оставив ни одной шаблону ab. (Для шаблона a*ab «аппетит» квантификатор * будет ограничен за счет работы поиска с возвратами: после того как на первом этапе не удастся найти соответствие между шаблоном и текстом, Perl сделает шаг назад и уменьшит количество букв a, захватываемых конструкцией a*.)
• *(?(условие)шаблон-да\шаблон-нет)* или *(?(условие)шаблон-да)* — условный оператор, который подставляет тот или иной шаблон в зависимости от выполнения заданного условия. Более подробно описан в документации Perl.
• *(? модификаторы)* — задает модификаторы (они описаны далее в разделе «Модификаторы команд m/.../ и s/.../.../»), которые *локальным* образом меняют работу процедуры поиска. В отличие от глобальных модификаторов, имеют силу только для текущего блока, то есть для ближайшей группы круглых скобок, охватывающих конструкцию. Например, шаблон **((?i)text)** соответствует слову «text» без учета регистра.
Примеры использования дополнительных конструкций в регулярных выражениях приводятся в разделе «Утверждения, проверяющие текст перед и после шаблона» в конце этой главы.

Модификаторы команд m/.../ и s/.../.../

В Perl имеется несколько модификаторов, используемых с командами m/.../ и s/.../.../:
• i — игнорирует различие между заглавными и строчными буквами.
• s — метасимволу «точка» разрешено соответствовать символам \n.
• m — разрешает метасимволам ^ и $ привязываться к промежуточным символам \n, имеющимся в тексте. Не влияет на работу метасимволов \A, \Z и \z.
• x — игнорирует «пробельные символы» в шаблоне (имеются в виду «истинные» пробелы, а не метасимволы \s и пробелы, созданные через escape-последовательности). Разрешает использовать внутри шаблона комментарии.
• g — выполняет глобальный поиск и глобальную замену (подробнее — см. следующий раздел).

• c — после того как в скалярном контексте при поиске с модификатором g не удалось найти очередное совпадение, *не* позволяет сбрасывать текущую позицию поиска (подробнее — см. следующий раздел). Работает только для команды m/.../ и только вместе с модификатором g.
• o — запрещает повторную компиляцию шаблона при каждом обращении к данному оператору поиска или замены. Пользователь, однако, должен гарантировать, что шаблон не меняется между вызовами данного фрагмента кода.
• e — показывает, что правый аргумент команды s/.../…/ — это фрагмент выполняемого кода. В качестве текста для подстановки будет использовано возвращаемое значение — возможно, после процесса интерполяции (подробнее — см. следующий раздел).
• ee — показывает, что правый аргумент команды s/.../…/ — это строковое выражение, которое надо вычислить и выполнить как фрагмент кода (через функцию **eval).** В качестве текста для подстановки используется возвращаемое значение — возможно, после процесса интерполяции (подробнее — см. следующий раздел).
(Некоторые модификаторы — например, i, s, m, x — могут находиться в дополнительных конструкциях, рассмотренных в предыдущем разделе.)
В качестве примера рассмотрим сценарий, в котором пользователь выполняет команду выхода, вводя слово «stop», «STOP» или даже «StOp», то есть без учета регистра:

```
while (<>)  {
      if (m/^stop$/i)
      {exit;}
}
```

Особенности работы команд m/.../ и s/.../.../

До сих пор мы рассматривали регулярные выражения, используемые в качестве шаблонов для команд m/.../ и s/.../.../, и не особо интересовались, как работают эти команды. Настало время восполнить пробелы.
Команда m/.../ ищет текст по заданному шаблону. Ее работа и возвращаемое значение сильно зависят от того, в скалярном или списковом контексте она используется и имеет ли модификатор g (глобальный поиск). Более подробно, работа команды m/.../ рассматривается ниже.
Команда s/.../.../ ищет прототип, соответствующий шаблону, и, если поиск оказывается успешным, заменяет его на новый текст. Без модификатора g замена производится только для первого найденного совпадения, с модификатором g выполняются замены для всех совпадений во входном тексте. Команда возвращает в качестве результата число успешных замен или пустую строку (условие *ложь —false),* если ни одной замены сделано не было.
В качестве анализируемого текста используется специальная переменная Perl $_ (режим по умолчанию) или выражение, присоединенное к шаблону с помощью оператора =~ или !~. В случае поиска (команда m/.../) конструкция, расположенная слева от операторов =~ или !~, может и не быть переменной. В случае замены (команда s/.../.../) в левой части должна стоять скалярная переменная, или элемент массива, или элемент хэша, или же команда присвоения одному из указанных объектов
Вместо косой черты в качестве ограничителя для аргументов команд m/.../ и s/.../.../ можно использовать любой символ, за исключением «пробельного символа», буквы или цифры. Например, в этом качестве можно использовать символ комментария, который будет работать как ограничитель:

```
$text="ABC-abc";
$text =" s#B#xxx#ig;
print Stext;
AxxxC-axxxc
```

Подсказка. *В качестве ограничителей не стоит использовать вопросительный знак и апостроф (одинарную кавычку) — шаблоны с такими ограничителями обрабатываются специальным образом.*

Если команда m/.../ использует символ косой черты в качестве разделителя,"то букву m можно опустить:

```
while (defined($text = <>))
      { if ($text =~ /^exit$/i) {exit;} }
```

Если в качестве ограничителя для команды m/.../ используется вопросительный знак, то букву m также можно опустить. Однако шаблоны, ограниченные символом ?, в случае поиска работают особым образом (независимо от наличия или отсутствия начальной m). А именно, они ведут себя как триггеры, которые срабатывают один раз и потом выдают состояние *ложь (false),* пока их не взведут снова,

вызвав функцию **reset** (она очищает статус блокировки сразу всех конструкций 7...?, локальных для данного пакета). Например, следующий фрагмент сценария проверяет, есть ли в файле пустые строки:

```
while (<>)  {
      if (?^$?)
            {print "There is an empty line here.\n";}
}
continue {
      reset if eof;     # почистить для следующего файла
}
```

Диагностическое сообщение будет напечатано только один раз, даже если в файле присутствует несколько пустых строк.

Предупреждение. *Команда поиска с вопросительным знаком относится к «подозрительным» командам, а потому может не войти в новые версии Perl*

В качестве ограничителей можно также использовать различные (парные) конструкции скобок:

```
while (<>) {
if (m/^quit$/i) {exit;}
if (m(^stop$)i) {exit;}
if (m[^end$]i) {exit;}
if (m{^bye$}i) {exit;}
if (m<^exit$>i) {exit;}
```

В случае команды s/.../.../ и использования скобок как ограничителей для первого аргумента, ограничители второго аргумента могут выбираться независимо:

```
Stext = "Perl is wonderful.";
Stext =~ s/is/is very/;
Stext =~ s[wonderful]{beautiful};
Stext =~ s(\.)/!/;
print $text;
```

Perl is very *beautiful!*

Предварительная обработка регулярных выражений

Аргументами команд m/.../ и s/.../.../ являются регулярные выражения, которые перед началом работы интерполируются подобно строкам, заключенным в двойные кавычки (см. раздел «Подстановка переменных (интерполяция строк)»). В отличие от текстовых строк, для шаблона не выполняется интерполяция имен типа $), $| и одиночного $ — Perl считает, что такие конструкции соответствуют метасимволу конца строки, а не специальной переменной. Если же в результате интерполяции шаблон поиска оказался пустой строкой, Perl использует последний шаблон, который применялся им для поиска или замены. Если вы не хотите, чтобы Perl выполнял интерполяцию регулярного выражения, в качестве ограничителя надо использовать Апостроф (одиночную кавычку), тогда шаблон будет вести себя, как текстовая строка, заключенная в апострофы. Однако, например, в случае команды замены s/.../.../ с модификатором e или ee (их работа описывается чуть дальше) для второго аргумента будет выполняться интерполяция даже в том случае, если он заключен в апострофы. Если вы уверены, что при любом обращении к команде поиска или замены шаблон остается неизменным (например, несмотря на интерполяцию, скалярные переменные внутри шаблона не будут менять своего значения), то можно задать модификатор o (см. выше раздел «Модификаторы команд m/.../ и s/.../.../»). Тогда Perl компилирует шаблон в свое внутреннее представление только при первой встрече с данной командой поиска или замены. При остальных обращениях к команде будет использовать откомпилированное значение. Однако, если внезапно изменить значение переменных, задействованных в шаблоне, Perl этого даже не заметит.

Команда замены s/.../.../ использует регулярное выражение, указанное в качестве второго аргумента, для замены текста. Поскольку оно обрабатывается (интерполируется) *после* того, как выполнена очередная операция поиска, в нем можно, в частности, использовать временные переменные, созданные на этапе поиска. В следующем примере мы последовательно заменим местами пары слов, заданных во входном тексте, оставив между ними по одному пробелу:

```
$text = "One     Two     Three      Four      Five      Six";
$text =~ s/(\w+)\s*(\w+)/$2 $1 /g;
Two One Four Three Six Five
```

Однако Perl допускает и более сложные способы определения заменяющего текста. Так, если для команды s/.../.../ указать модификатор e, to в качестве второго аргумента надо указать код, который необходимо выполнить (например, вызвать функцию). Полученное выражение будет использовано как текст для подстановки. При этом после вычисления текстового значения, но *перед* его подстановкой

будет выполнен процесс интерполяции, аналогичный процессу интерполяции текстовых строк, заключенных в двойные кавычки (см. раздел «Подстановка переменных (интерполяция строк)» в главе 2).
Еще более сложная схема реализуется, если задан модификатор ее. В этом случае второй аргумент команды s/.../.../ — это строковое выражение, которое сперва надо *вычислить (то* есть интерполировать), затем *выполнить* в качестве кода (вызвав встроенную функцию Perl eval), и только после второй *интерполяции* полученный результат подставляется вместо найденного текста.

Работа команды m/…/ в режиме однократного поиска

В скалярном контексте и без модификатора g команда m/.../ возвращает логическое значение — целое число 1 *(истина (true)),* если поиск оказался успешным, и пустую строку "" *(ложь (false)),* если нужный фрагмент текста найти не удалось. Если внутри шаблона имеются группы элементов, заключенные в круглые скобки, то после операции поиска создаются нумерованные переменные $1, $2, … в которых содержится текст, соответствующий круглым скобкам. В частности, если весь шаблон заключить в круглые скобки, то в случае успешного поиска переменная $1 будет содержать текст, соотнесенный с шаблоном. (После успешного поиска можно также использовать специальные переменные $&, $\ $' и $+ см. ранее раздел «Ссылки на найденный текст».) Пример:

```
$text = "---one---two---three---";
$scalar = ($text =~ m/(\w+)/);
print "Result:  $scalar ($1).";
```

Result: 1 (one).

Если вы используете команду m/.../ в списковом контексте, то возвращаемое значение сильно зависит от того, есть ли группы из круглых скобок в вашем шаблоне. Если они есть (то есть если создаются нумерованные переменные), то после успешного поиска в качестве результата будет получен список, составленный из нумерованных переменных ($1, $2, ...):

```
$text = "---one,      two,      three---";
@array = ($text =~ m/(\w+),\s+(\w+),\s+(\w+)/);
print join ("=", @array);
```

one=two=three

В отличие от ранних версий, Perl 5 присваивает значения нумерованным переменным, даже если команда поиска работает в списковом контексте:

```
$text = "---one,      two,      three---";
($Fa,   $Fb,   $Fc) = ($text =~ m/(\w+).\s+(\w+)As+(\w+)/);
print "/$Fa/$Fb/$Fc/\n";
print "$1=$2=$3.\n";
```

/one/two/three/
one-two-three.

Если же в шаблоне' нет групп, выделенных круглыми скобками, то в случае успешного поиска возвращается список, состоящий из одного элемента — числа 1. При неудачном поиске независимо от того, были ли в шаблоне круглые скобки, возвращается пустой список:

```
$text = "---one,      two,      three---";
@array = ($text =~ m/z\w+/);
print "Result:   /",  @array.   "An";
print "Size:   ",  $#array+1,   ".\n";
```

Result://
Size: 0.

(Обратите внимание на разницу между *пустым* и *неопределенным* списками.)

Работа команды m/…/ в режиме глобального поиска

Команда m/.../ работает иначе, если указан модификатор g, задающий глобальный поиск всех вхождений шаблона по всему тексту. Если оператор используется в списковом контексте и в шаблоне есть группы круглых скобок, то в случае удачного поиска возвращается список, состоящий из *всех* найденных групп, расположенных друг за другом:

```
$text = "---one---two---three---";
@array = (Stext =" m/(-(\w+))/);
print "Single:   [",   join("s   ",  @array), "].\n";
@array = (Stext =~ m/(-(\w+))/g);
print "Global:   [",   join(",   ",  @array),   "].\n";
```

Single: [-one, one].
Global: [-one, one, -two, two, -three, three].

Если же в шаблоне нет групп круглых скобок, то оператор поиска возвращает список всех найденных прототипов шаблона, то есть ведет себя так, как если бы весь шаблон был заключен в круглые скобки:

```
$text = "---one---two---three---";
@array = ($text =~ m/\w+/);
print "Result:   (",   join(",   ",  @array),   ").\n";
```

Result: (one, two, three).

В случае неудачного поиска, как и в предыдущих вариантах, возвращается пустой список.

В скалярном контексте и с модификатором **g** команда **m/.../** ведет себя совершенно особым образом. Специальная переменная **$_** или переменная, стоящая слева от оператора **=~** или **!~**, при поиске с модификатором **g** получает дополнительные свойства — в нее записывается последнее состояние. При каждом последующем обращении к данному фрагменту кода поиск будет продолжаться с того места, на котором он остановился в последний раз. Например, следующая команда подсчитывает количество букв x в заданной строке текста:

```
$text = "Here is texxxxxt.";
$counter = 0;
while ($text =~ m/x/g) {
      print "Found another x.\n";
      $counter++
}
print "Total amount = $counter. \n";
```

Found another x.
Found another x
Found another x.
Found another x.
Found another x.
Total amount = 5.

Состояние (точнее, позиция) поиска сохраняется даже в случае перехода к следующему оператору поиска, имеющему модификатор **g**. Неудачный поиск сбрасывает значение в исходное состояние, если только для команды **m/.../** не указан модификатор **c** (то есть команда должна иметь вид **m/.../gc**). Изменение текстового буфера, для которого выполняется поиск, также сбрасывает позицию поиска в исходное состояние. В следующем примере из текстовой строки последовательно извлекаются и выводятся пары имя/значение до тех пор, пока строка не закончится:

```
$text = "X=5; z117e=3. 1416;  temp=1024;";
$docycle = 1; $counter = 0;
while ($docycle) {
      undef $name;  undef $value;
      if ($text =~ m/(\w*)\s*=\s*/g) {$name = $1;}
      if ($text =~ m/([\d\.\+\-]*)\s*;/g) {$value = $1;}
      if (defined($name) and defined($value))  {
            print "Name=$name,  Value=$value.\n"; $counter++;
      } else {
            $docycle = 0;
      }
}
print "I have found $conter values. \n";
```

Name-X, Value=5.
Name=z117e, Value=3. 1416.
Name=temp, Value-1024.
I have found 3 values.

Позиция, на которой остановился поиск, может быть прочитана и даже переустановлена с помощью встроенной функции Perl **pos**. В шаблоне на текущую позицию поиска можно ссылаться с помощью метасимвола **\G**. В следующем примере из строки последовательно извлекаются буквы **p**, **o** и **q** и выводится текущая позиция поиска:

```
$index = 0;
$_ = "ppooqppqq";
while ($index++ < 2)  {
      print  "1:  '";
      print $1 while /(o)/gc; print "', pcs=", pos, "\n";
      print "2: '";
      print $1 if /\G(q)/gc; print "', pos=", pos, "\n";
```

```
      print "3: '";
      print $1 while /(p)/gc;   print '",   pos=",   pos,   "\n";
}
```
1: 'oo', pos-4
2: 'q', pos-5
3: 'pp', pos=7
1: '', pos=7
2: 'q', pos=8
3: '', pos=8

В документации Perl приводится основанный на этом механизме интересный пример последовательного лексического разбора текста. В нем каждая последующая команда поиска очередной лексической единицы начинает выполняться с того места, где завершила свою работу предыдущая. Советую внимательно разобраться с этим примером (страница руководства **perlop**, раздел «Regexp Quote-Like Operators», описание команды **m/PATTERN/**), если вы хотите расширить доступный вам инструментарий Perl!

Замена строк с помощью команды tr/.

Кроме команд **m/.../** и **s/.../.../** строки можно обрабатывать с помощью команды **tr/.../.../** (она же — команда **y/.../.../**):

```
tr/список1/список2/модификаторы
y/список1/список2/модификаторы
```

В отличие от **m/.../** и **s/.../.../**, эта команда не использует шаблоны и регулярные выражения, а выполняет посимвольную замену, подставляя в текст вместо литер из первого списка соответствующие им литеры из второго списка. Например, в следующем случае производится замена литер “**i**” на “**o**”:

```
$text = "My name is Tim.";
$text =~ tr/i/o/;
print $text;
```
My name is Tom.

В качестве списков используются идущие друг за другом символы, не разделяемые запятыми (то есть это скорее строки, чем списки). В отличие от шаблонов команд **m/.../** и **s/.../.../**, аргументы команды **tr/.../.../** *не интерполируются* (то есть подстановки значений вместо имен переменных не происходит), хотя escape-последовательности, указанные внутри аргументов, обрабатываются правильно.

Подобно **m/.../** и **s/.../.../**, команда **tr/.../.../** по умолчанию работает с переменной **$_**:

```
while (<>) {
      tr/il/jJ/;
      print;
}
```

В качестве списков можно указывать диапазоны символов — как, например, в следующем фрагменте кода, заменяющем строчные буквы на заглавные:

```
$text = "Here is the text.";
$text =~ tr/a-z/A-Z/;
print $text;
```
HERE IS THE TEXT.

Как и в случае **m/.../** и **s/.../.../**, команда **tr/.../.../** не требует использовать именно знаки косой черты в качестве ограничителей. Можно использовать практически любой символ, отличный от «пробельных», букв и цифр, а также парные скобочные конструкции, описанные ранее в разделе «Особенности работы команд **m/.../** и **s/.../.../**».

Команда **tr/.../.../** возвращает число успешных замен. В частности, если не было сделано никаких замен, она возвращает число ноль. Это позволяет, например, подсчитать с помощью команды **tr/.../.../** количество вхождений буквы **x** в строку **$text**, не меняя содержимого этой переменной:

```
$text = "Here is the text.";
Sxcount = ($text =~ tr/x/x/);
print $xcount;
1
```

Если у команды **tr/.../.../** нет модификаторов (см. далее раздел «Модификаторы команды **tr/.../.../**»), то ее аргументы при обычных условиях должны быт одинаковой длины. Если второй аргумент длиннее

первого, то он усекается до длины первого аргумента, — так, команда **tr/abc/0-9/** эквивалентна команде **tr/abc/012/**. Если первый аргумент длиннее второго и второй не пуст, то для второго аргумента необходимое число раз повторяется его последний символ, — так, команда **tr/0-9/abc/** эквивалентна команде **tr/0123456789/abccccccccc/.** Если же второй аргумент пуст, то команда **tr/.../.../** подставляет вместо него первый аргумент.
Как легко заметить, если второй аргумент пуст, то (при отсутствии модификаторов) команда **tr/.../.../** не производит никаких действий, а возвращаемое ею значение равно числу совпадений между первым аргументом и обрабатываемым текстом. Например, следующая команда подсчитывает количество цифр в строке:

```
$text = "Pi=3.1415926536, e=2.7182";
$digit_counter = ($text =~ tr/0-9//);
print $digit_counter;
16
```

Команда **tr/.../.../** работает без рекурсии, просто последовательно заменяет символы входного текста. Например, для замены заглавных букв на строчные, и наоборот, достаточно выполнить команду:

```
$text = "MS Windows 95/98/NT";
$text =~ tr/A-Za-z/a-zA_Z/;
print Stext;
ms WINDOWS 95/98/nt
```

Если в списке, указанном в качестве первого аргумента, есть повторяющиеся символы, то для замены используется первое вхождение символа:

```
$text = "Billy Gates";
$text =~ tr/ttt/mvd/;
print $text;
Billy Games
```

Модификаторы команды tr/…/…/

Команда **tr/.../.../** допускает использование следующих модификаторов:

- **d** — удаляет непарные символы, не выравнивая аргументы по длине.
- **c** — в качестве первого аргумента использует полный список из 256 символов за вычетом указанных в списке символов.
- **s** — удаляет образовавшиеся в результате замены повторяющиеся символы.

Если указан модификатор **d**, а первый аргумент команды длиннее второго, то все символы из первого списка, не имеющие соответствия со вторым списком, удаляются из обрабатываемого текста. Пример: удаляем строчные латинские буквы и заменяем пробелы на слэши:

```
$text = "Here is the text.";
$text =~ tr[ a-z][/]d;
print $text;
```

(Наличие модификатора **d** — единственный случай, когда первый и второй аргументы не выравниваются друг относительно друга. В остальных вариантах второй аргумент либо усекается, либо последний символ в нем повторяется до тех пор, пока аргументы не сравняются, либо, если второй аргумент пуст, вместо второго аргумента берется копия первого.)
Если указан модификатор **c**, то в качестве первого аргумента рассматриваются все символы, *кроме* указанных. Например, заменим на звездочки все символы, кроме строчных латинских букв:

```
$text = "Here is the text.";
$text =~ tr/a-z/*/c;
print $text;
*ere*is*the*text*
```

Если указан модификатор **s**, то в случае если замещаемые символы образуют цепочки из одинаковых символов, они сокращаются до одного. Например, заменим слова, состоящие из латинских букв, на однократные символы косой черты:

```
$text = "Here is the text.”;
$text =~ tr(A-Za-z)(/)s;
print $text;
////.
```

Без модификатора **s** результат был бы другим:

```
$text = "Here is the text.";
$text =~ tr(A-Za-z)(/);
print $text;
//// / /// ////.
```

Примеры:

1. Заменить множественные пробелы и нетекстовые символы на одиночные пробелы:

```
$text = "Here   is  the  text.";
$text =~ tr[\000-\040\177\377][\040]s;
print $text;
```

Here is the text.

2. Сократить удвоенные, утроенные и т. д. буквы:

```
$text = "Here is the texxxxxxt.";
$text =~ tr/a-zA-Z//s; print $text;
```

Here is the text.

3. Пересчитать количество небуквенных символов:

```
$xcount=($text =~ tr/A-Za-z//c);
```

4. Обнулить восьмой бит символов, удалить нетекстовые символы:

```
$text =~ tr{\200-\377}{\000-\177};
$text =~ tr[\000-\037\177][]d;
```

5. Заменить нетекстовые и 8-битные символы на одиночный пробел:

```
$text =~ tr/\021-\176/ /cs;
```

Поиск отдельных слов

Чтобы выделить слово, можно использовать метасимвол **\S**, соответствующий символам, отличным от «пробельных»:

```
$text = "Now is the time.";
$text =~ /(\S+)/;
print $1;
```

Now

Однако метасимвол **\S** соответствует также и символам, обычно не используемым для идентификаторов. Чтобы отобрать слова, составленные из латинских букв, цифр и символов подчеркивания, нужно использовать метасимвол **\w**:

```
$text = "Now is the time.";
$text =~ /(w+)/;
print $1;
```

Now

Если требуется включить в поиск только латинские буквы, надо использовать класс символов:

```
$text = "Now is the time.";
$text =~ /([A-Za-z]+)/;
print $1;
```

Now

Более безопасный метод состоит в том, чтобы включить в шаблон мнимые символы границы слова:

```
$text = "Now is the time.";
$text =~ /\b([A-Za-z]+)\b/;
print $1;
```

Now

Привязка к началу строки

Началу строки соответствует метасимвол (мнимый символ) ^. Чтобы привязать шаблон к началу строки, надо задать этот символ в начале регулярного выражения. Например, вот так можно проверить, что текст не начинается с точки:

```
$line = ".Hello!";
if ($line =~ m/^\./) {
      print "Shouldn't start a sentence with a period!\n"; }
```

Shouldn't start a sentence with a period!

Подсказка. *Чтобы точка, указанная в шаблоне, не интерпретировалась как метасимвол, перед ней пришлось поставить обратную косую черту.*

Привязка к концу строки

Чтобы привязать шаблон к концу строки, используется метасимвол (мнимый символ) **$**. В нашем примере мы используем привязку шаблона к началу и к концу строки, чтобы убедиться, что пользователь ввел только слово «exit»:

```
while (<>)  {
      if (m/^exit$/) {exit;}
}
```

Поиск чисел

Для проверки того, действительно ли пользователь ввел число, можно использовать метасимволы **\d** и **\D**. Метасимвол **\D** соответствует любому символу, кроме цифр. Например, следующий код проверяет, действительно ли введенный текст представляет собой целое значение без знака и паразитных пробелов:

```
$text = "Hello!";
if ($text = ~ /W) {
      print "It is not a number. \n"; }
```

It is not a number.

То же самое можно проделать, использовав метасимвол **\d**:

```
$text = "333";
if ($text = ~ /~\d+$/) {
      print "It is a number. \n"; }
```

It is a number.

Вы можете потребовать, чтобы число соответствовало привычному формату. То есть число может содержать десятичную точку, перед которой стоит по крайней мере одна цифра и, возможно, какие-то цифры после нее:

```
$text = "3.1415926";
if ($text =~ /^(\d+\.\d*|\d+)$/){
      print "It is a number. \n"; }
```

It is a number.

Подсказка. *Чтобы точка, указанная в шаблоне, не интерпретировалась как метасимвол, не забудьте поставить перед ней обратную косую черту.*

Кроме того, при проверке можно учитывать тот факт, что перед числом может стоять как плюс, так и минус (или пустое место):

```
$text = "-2. 7182";
if ($text =~ /^([+-]*\d+(\.\d*| )$/) {
      print "It is a number. \n"; }
```

It is a number.

Подсказка. *Поскольку «плюс» является метасимволом, его надо защищать обратной косой чертой. Однако внутри квадратных скобок, то есть класса символов, он не может быть квантификатором. Знак «минус» внутри класса символов обычно играет роль оператора диапазона и поэтому должен защищаться обратной косой чертой. Однако в начале или в конце шаблона он никак не может обозначать диапазон, и поэтому обратная косая черта необязательна.*

Наконец, более строгая проверка требует, чтобы знак, если он присутствует, был только один:

```
$text = "+0.142857142857142857";
if ($text =~ /^(+|-| )\d+(\.\d*| )$/) {
      print "It is a number. \n"; }
```

It is a number.

Подсказка. *Альтернативные шаблоны, если они присутствуют, проверяются слева направо. Перебор вариантов обрывается, как только найдено соответствие между текстом и шаблоном. Поэтому, например, порядок альтернатив в шаблоне* ***(\.\d*|)*** *мог бы стать критичным, если бы не привязка к концу строки.*

Наконец, вот как можно произвести проверку того, что текст является шестнадцатеричным числом без знака и остальных атрибутов:

```
$text = "1A0";
unless ($text =~ m/^[a-fA-F\d]+$/)
      { print "It is not a hex number. \n";}
```

Подсказка. *Придумайте сами, как добавить десятичную мантиссу* ***Enn*** *к формату десятичного числа, а также знак и префикс* ***0x*** *к формату шестнадцатеричного числа.*

Проверка идентификаторов

С помощью метасимвола \w можно проверить, состоит ли текст только из букв, цифр и символов подчеркивания (это те символы, которые Perl называет *слоеными* (word characters)):

```
$text = "abc";
if ($text =~ /^\w+$/)  {
      print "Only word characters found. \n"; }
```

Only word characters found.

Однако, если вы хотите убедиться, что текст содержит латинские буквы и не содержит цифр или символов подчеркивания, придется использовать другой шаблон:

```
$text = "aBc";
if ($text =~ /^[A-Za-z]+$/) {
      print "Only letter characters found. \n"; }
```

Only letter characters found.

Наконец, для проверки, что текст является идентификатором, то есть начинается с буквы и содержит буквы, цифры и символы подчеркивания, можно использовать команду:

```
$text = "X125c";
if ($text =~ /^[A-Za-z]\w+$/) {
      print "This is identifier.\n"; }
```

This is identifier.

Подсказка. *Придумайте, как исключить из символов, используемых для построения идентификаторов, знак подчеркивания.*

Как найти множественные совпадения

Для поиска нескольких вхождений шаблона можно использовать модификатор **g**. Следующий пример, который мы уже видели ранее, использует команду **m/.../.../** с модификатором **g** для поиска всех вхождений буквы **x** в тексте:

```
$text = "Here is texxxxxt.";
while ($text =~ m/x/g) {
      print  "Found another x.\n"; }
```

Found another x.
Found another x.
Found another x.
Found another x.
Found another x.

Модификатор **g** делает поиск глобальным. В данном (скалярном) контексте Perl помнит, где он остановился в строке при предыдущем поиске. Следующий поиск продолжается с отложенной точки. Без модификатора **g** команда **m/.../** будет упорно находить первое вхождение буквы **x**, и цикл будет продолжаться бесконечно.

В отличие от команды **m/.../** команда **s/.../.../** с модификатором **g** выполняет глобальную замену за один раз, работая так, будто внутри нее уже имеется встроенный цикл поиска, подобный приведенному выше. Следующий пример за один раз заменяет все вхождения **x** на **z**:

```
$text = "Here is texxxxxt.";
$text =~ s/x/z/g;
print Stext;
```

Here is tezzzzzt.

Без модификатора **g** команда **s/.../.../** заменит только первую букву **x**.

Команда **s/.../.../** возвращает в качестве значения число сделанных подстановок, что может оказаться полезным:

```
$text = "Here is texxxxxt.";
print (text =~ s/x/z/g);
```

5

Поиск нечувствительных к регистру совпадений

Вы можете использовать модификатор **i**, чтобы сделать поиск нечувствительным к разнице между заглавными и строчными буквами. В следующем примере программа повторяет на экране введенный пользователем текст до тех пор, пока не будет введено **Q** или **q** (сокращение для QUIT или quit), после чего программа прекращает работу:

```
while (<>) {
      chomp;
      unless (/~q$/i) {
            print; }
      else
            { exit; }
}
```

Выделение подстроки

Чтобы получить найденную подстроку текста, можно использовать круглые скобки в теле шаблона. (Если это более удобно, можно также использовать встроенную функцию **substr.)** В следующем примере мы вырезаем из текстовой строки нужный нам тип изделия:

```
$record = "Product number:12345
           Product type:   printer
           Product price:   $325";
if ($record =~ /Product type:\s*([a-z]+)/i) {
      print "The product's type is $1.\n";
}
```

The product's type is printer.

Вызов функций и вычисление выражений при подстановке текста

Используя для команды **s/.../.../** модификатор **e**, вы то тем самым показываете, что правый операнд (то есть подставляемый текст) — это то выражение Perl, которое надо вычислить. Например, с помощью встроенной функции Perl **uc** (uppercase) можно заменить все строчные буквы слов строки на заглавные:

```
$text = "Now is the time.";
$text =~ s/(\w+)/uc($1)/ge;
print $text;
```

NOW IS THE TIME.

Вместо функции **uc($1)** можно поместить произвольный код, включая вызовы подпрограмм.

Поиск n-го совпадения

С помощью модификатора **g** перебираются все вхождения заданного шаблона. Но что делать, если нужна вполне определенная точка совпадения с шаблоном — например, вторая или третья? Оператор цикла **while** в сочетании с круглыми скобками, выделяющими нужный образец, поможет вам:

```
$text = "Name: Anne Name:Burkart Name: Claire Name: Dan";
$match = 0;
while ($text =~ /Name:\s*(\w+)/g)
      { ++$match;
      print "Match number $match is $1.\n";  }
```

Match number 1 is Anne
Match number 2 is Burkart
Match number 3 is Claire
Match number 4 is Dan

Этот пример можно переписать, используя цикл **for:**

```
$text = "Name:Anne Name:Burkart Name:Claire Name:Dan";
for ($match = 0;
      $text =~ /Name:\s*(\w+)/g;
      print "Match number ${\++$match} is $1,\n")
{}
```

Match number 1 is Anne
Match number 2 is Burkart
Match number 3 is Claire
Match number 4 is Dan

Если же вам требуется определить нужное совпадение не по номеру, а по содержанию (например, по первой букве имени пользователя), то вместо счетчика **$match** можно анализировать содержимое переменной **$1**, обновляемой при каждом найденном совпадении.
Когда требуется не найти, а заменить второе или третье вхождение текста, можно применить ту же схему, использовав в качестве тела цикла выражение Perl, вызываемое для вычисления заменяющей строки:

```
$text = "Name:Anne Name:Burkart Name:Claire Name:Dan";
$match = 0;
$text =~ s/(Name:\s*(\w+))/          ### начинается код Perl
      if (++$match == 2)             # увеличить счетчик
            {"Name:John ($2)"}       # вернуть новое значение
      else {$1}                      # оставить старое значение
/gex;
print $text;
```

Name:Anne
Name:John (Burkartl
Name:Claire
Name:Dan

В процессе глобального поиска при каждом найденном совпадении вычисляется выражение, указанное в качестве второго операнда. При его вычислении увеличивается значение счетчика, и в зависимости от него в качестве замены подставляется либо старое значение текста, либо новое.
Модификатор **x** позволяет добавить в поле шаблона комментарии, делая код более прозрачным. (Обратите внимание, что нам пришлось заключить весь шаблон в круглые скобки, чтобы получить значение найденного текста и подставить его на прежнее место полностью.)

Как ограничить «жадность» квантификаторов

По умолчанию квантификаторы ведут себя как «жадные» объекты. Начиная с текущей позиции поиска, они захватывают самую длинную строку, которой может соответствовать регулярное выражение, стоящее перед квантификатором. Алгоритм перебора с возвратами, используемый Perl, способен ограничивать аппетит квантификаторов, возвращаясь назад и уменьшая длину захваченной строки, если не удалось найти соответствия между текстом и шаблоном (детали см. в подразделе «„Жадность" квантификаторов» раздела «Квантификаторы в регулярных выражениях»). Однако этот механизм не всегда работает так, как хотелось бы.
Рассмотрим следующий пример. Мы хотим заменить текст «That is» текстом «That's». Однако в силу «жадности» квантификатора регулярное выражение «.*is» сопоставляется фрагменту текста от начала строки и до последнего найденного «is»:

```
$text = "That is some text, isn't it?";
$text =~ s/,*is/That's/;
print $texts;
```

That'sn't it?

Чтобы сделать квантификаторы не столь жадными, а именно, заставить их захватывать минимальную строку, с которой сопоставимо регулярное выражение, — после квантификатора нужно поставить вопросительный знак. Тем самым квантификаторы, перечисленные в разделе «Квантификаторы в регулярных выражениях», принимают следующий вид:
• ***?*** — ноль или несколько совпадений,
• **+?** — одно или несколько совпадений,
• **??** — ноль совпадений или одно совпадение,
• ***{n}?*** — ровно ***n*** совпадений,
• **{n,}?** — по крайней мере ***n*** совпадений,
• **{n,m}?** — совпадений по крайней мере ***n***, но не более, чем ***m***.
Обратите внимание, что смысл квантификатора от этого не меняется, меняется только поведение алгоритма поиска. Если в процессе сопоставления шаблона и текста прототип определяется однозначно, то алгоритм поиска с возвратами увеличит «жадность» такого квантификатора точно

так же, как он ограничивает аппетит его собрата. Однако если выбор неоднозначен, то результат поиска будет другим:

```
$text = "That is some text, isn't it?";
$text =~ s/.*?is/That's/;
print Stexts;
```

That's some text, isn't it?

Как удалить ведущие и завершающие пробелы

Чтобы отсечь от строки начальные «пробельные символы», можно использовать следующую команду:

```
$text = "        Now is the time.";
$text =~ s/^\s+//:
print $text;
```

Now is the time.

Чтобы отсечь «хвостовые» пробелы, годится команда:

```
$text = "Now is the time.        ";
$text =~ s/\s+$//;
print $text;
```

Now is the time.

Чтобы отсечь и начальные, и хвостовые пробелы, лучше вызвать последовательно эти две команды, чем использовать шаблон, делающий отсечение ненужных пробелов за один раз. Поскольку процедура сопоставления шаблона и текста достаточно сложна, на эту простую операцию может уйти гораздо больше времени, чем хотелось бы.

Утверждения, проверяющие текст до и после шаблона

Как уже упоминалось, в регулярных выражениях Perl есть конструкции (мнимые символы нулевой длины), позволяющие проверять утверждения о фрагментах текста, расположенных перед проверяемым шаблоном или после него:

- ***(?=шаблон)*** — после текущей точки находится фрагмент текста, соответствующий указанному регулярному выражению,
- ***(?!шаблон)*** — после текущей точки *лет* текста, соответствующего указанному регулярному выражению,
- ***(?<=шаблон)*** — *перед* текущей точкой есть фрагмент текста, соответствующего указанному регулярному выражению,
- ***(?<!шаблон)*** — перед текущей точкой н*ет* фрагмента текста, соответствующего указанному регулярному выражению.

(Конструкции ***(?<=шаблон)*** и ***(?<!шаблон)*** работают только с шаблонами, соответствующими фиксированному числу символов. Иными словами, в шаблонах, указываемых для **(?<=...)** и **(?<!....)**, не должно быть квантификаторов.)

Эти условия полезны, если нужно проверить, что перед определенным фрагментом текста или после него находится нужная строка, однако ее не требуется включать в результат поиска. Это бывает необходимо, если в коде используются специальные переменные **$&** (фрагмент, для которого найдено соответствие между текстом и регулярным выражением), **$`** (текст, предшествующий найденному! фрагменту) и **$'** (текст, следующий за найденным фрагментом). (Более гибким представляется применение нумерованных переменных **$1, $2, $3, ...**, в которые заносятся отдельные части найденного фрагмента.)

В следующем примере ищется слово, за которым следует пробел, но сам пробел не включается в результат поиска:

```
Stext = "Mary Tom Frank ";
while ($text =~ /\w+(?=\s)/g) {
      print $& .   "\n";
}
```

Mary
Tom
Frank

Того же результата можно добиться, если заключить в круглые скобки интересующую нас часть шаблона и затем использовать ее как переменную **$1:**

```
$text/= "Mary Tom Frank ";
```

```
while ($text =~ /(\w+)\s/g) {
        print $1 .  "\n";
}
```
Mary Tom Frank

Следует четко понимать, что вы имеете в виду, когда используете то или иное условие. Рассмотрим следующий пример:

```
$text = "Mary+Tom ";
if ($text =~ m|(?!Mary\+)Tom|) {
      print "Tom is without Mary!\n";
} else {
      print "Tom is busy...\n";
```

Вопреки нашим ожиданиям, Perl напечатает:

```
$text = "Mary+Tom ";
if ($text =~ m|(?!Mary\+)Tom|) {
      print "Tom is without Mary!\n";
} else {
      print "Tom is busy...\n"; }
```

Tom is without Mary!

Это произойдет по следующей причине. Пробуя различные начальные точки входной строки, от которой начинается сопоставление шаблона и текста, Perl рано или поздно доберется до позиции, расположенной прямо перед именем «Tom». Условие (?!Mary\+) требует, чтобы *после* текущей точки не находился текст «Mary+», и это условие для *рассматриваемой* точки будет выполнено. Далее, Perl последовательно проверяет, что после текущей точки следуют буквы «T», «o» и «m», и это требование также в силе (после проверки условия (?!Mary\+) текущая точка остается на месте). Тем самым найдено соответствие между подстрокой «Tom» и шаблоном, поэтому команда поиска возвращает значение *истина.*

Регулярное выражение (?!Mary\+)....Tom, резервирующее четыре символа под текст «Mary+», для приведенного выше случая выведет то, что требовалось, но выдаст ошибочный ответ, если перед именем «Tom» *нет* четырех символов:

```
$text = "O,  Tom!   ";
if ($text =~ m|(?!Mary\+)....Tom|) {
      print "Tom is without Mary!\n"; }
else {
      print "Tom is busy...\n"; }
```

Tom is busy. . .

Наконец, если более точно сформулировать, чего требуется, получится нужный результат:

```
$text = "Mary+Tom ";
if ($text =~ m|(?<|Mary\+)Tom|) {
      print "Tom is without Mary!\n";
} else {
      print "Tom is busy...\n"; }
```

Tom is busy...