ВИКИУЧЕБНИК
Открытые книги для открытого мира

Некоторые сведения о Perl 5/Работа со строками и регулярные выражения

Материал из Викиучебника — открытых книг для открытого мира

< Некоторые сведения о Perl 5

← Пакеты, библиотеки и модули Глава Приложения →

Работа со строками и регулярные выражения

Регулярные выражения в Perl

Ниже представлены метасимволы, которые вводит Perl-диалект регулярных выражений.

```
\ ^ . $ | ( ) [ ] * + ? { } *? +? ??
# Символ запятой ',' используется в квантификаторах
```

Квантификаторы

```
*         Ноль и больше повторений.
+         Одно и больше повторений.
?         Одно повторение.
{n,m}     Не менее n и не более m повторений.
{n,}      n и больше повторений.
{,m}      Не более m повторений.
*?        Ноль и больше повторений. (ленивая квантификация)
+?        Не менее одного символа. (ленивая квантификация)
??        Одно повторение. (ленивая квантификация)
{2,5}?    По меньшей мере 2. (ленивая квантификация)
```

Составные метасимволы

Метасимвол	Аналог	Значение	Метасимвол	Аналог	Значение
Классы литералов					
\t	[\x09]	Управляющая последовательность «горизонтальный табулятор»	\n	[\x0a]	Управляющая последовательность «перевод строки»
\r	[\x0d]	Управляющая последовательность «возврат каретки»	\f	[\x0c]	Управляющая последовательность «разрыв страницы»
\a	[\x07]	Управляющая последовательность «перевод формата/ звонок спикера»	\v	[\x0b]	Управляющая последовательность «вертикальная табуляция»
\e	[\x1b]	Escape-символ	\nnn \0nn \o{nnn..} где n – восьмеричная цифра		Символ в виде восьмеричного кода. Обычно коды используют, когда символ невозможно ввести с клавиатуры. Обратите внимание, что \nnn совпадает с ссылкой на группу с захватом (если она есть в шаблоне), поэтому чтобы отметить восьмеричный код существуют разные варианты
\xnn \x{nnn..} где n – шестнадцатеричная цифра		Символ в виде шестнадцатеричного кода. Обычно коды используют, когда символ невозможно ввести с клавиатуры	\cX где X – ASCII-символ		Сочетание `Ctrl` + `X`, передаваемое в потоке символов. Например \cD означает `Ctrl` + `D`
\uNNN где N – десятичная цифра		Символ в кодировке Unicode	\v	[\x0b]	Управляющая последовательность «вертикальная табуляция»
\w	[a-zA-Z0-9_]	Буквы, цифры и символ нижнего подчеркивания. Другими словами, это все символы, из которых можно составить слово. Обратите внимание, что аналог не	\W	[^a-zA-Z0-9_]	Не символ класса \w

		совсем точно передает смысл метасимвола: современные процессоры поддерживают множество алфавитов, а не только латинский, поэтому более корректным было бы поместить в произвольную группу множество других печатаемых символов			
\s	[\t\n\r\f\v]	Символ-разделитель	\S	[^ \t\n\r\f\v]	Не символ-разделитель
\d	[0-9]	Десятичная цифра	\D	[^0-9]	Не десятичная цифра
\$	[\$]	Символ $	\@	[@]	Символ @
\%	[%]	Символ %			
Мнимые метасимволы					
^		Начало фрагмента	$		Конец фрагмента
\b	\W\w \w\W	Граница слова	\B		Не граница слова
\A		Левая граница текста. В однострочном режиме аналогичен ^	\Z		Правая граница текста. В однострочном режиме аналогичен $
\l		Следующий символ должен быть преобразован к нижнему регистру. Например /\lP/ аналогично /p/	\u		Следующий символ шаблона должен быть преобразован к верхнему регистру
\L...\E		Все символы в шаблоне между \L и \E преобразуются к нижнему регистру	\U...\E		Все символы в шаблоне между \U и \E преобразуются к верхнему регистру
\Q...\E		Все символы в шаблоне между \Q и \E экранируются символом \	\G		Точка в тексте, на которой закончился предыдущий поиск операции m/.../g

- Чтобы использовать некоторый символ, который совпадает с метасимволом регулярного выражения, буквально, его нужно экранировать с помощью обратного слеша \. Например,

символы скобок () являются метасимволами, и чтобы их понимать буквально (как символы скобок), их нужно экранировать следующим образом: \(и \). В диалекте Perl любой экранированный символ, который не совпадает ни с одним метасимволом, будет трактоваться всегда буквально, например, \y – избыточный способ представить символ y.

- Внутри шаблонов конструкции вида \n, где n - число, являются обратными ссылками на группы. При этом конструкция будет трактоваться как обратная ссылка на группу, если число групп в основном шаблоне не меньше, чем указанное число, иначе данная конструкция будет трактоваться как восьмеричный код символа. Вы можете определять обратную ссылку как \g1, т.е. перед номером группы нужно поставить символ g. В этом случае неоднозначности между кодом символа и обратной ссылкой не возникает.
- Как будет показано в примерах, к группам можно обращаться через специальные переменные типа $1, $2, $3 и так далее, где число обозначает номер группы. Область действия этих переменных распространяется на блок, если регулярное выражение в нем записано; до конца строки/блока eval; до следующего регулярного выражения, в котором есть раскрытие в группах.
- В Perl предусмотрено несколько специальных переменных, чтобы иметь возможность обращаться к различным частям фрагмента после сопоставления с образцом. Следует помнить, что если в программе встречается хотя бы одно обращение к любой из них, они будут вычисляться для всех регулярных выражений в программе, что может сказаться на производительности, поэтому не стоит к ним обращаться просто так.
  - $& – после сопоставления с шаблоном, будет хранить найденную часть фрагмента, которая совпала с шаблоном;
  - $` – часть строки, которая стоит в проверяемом фрагменте до той части, которая совпала с шаблоном;
  - $' – часть строки, которая стоит в проверяемом фрагменте после той части, которая совпала с шаблоном.
- Регулярные выражения обычно записываются между двумя разделителями (обычно это //, например /<шаблон>/). Разделители // используются по умолчанию. Ниже будет показано, что они могут быть заменены на другие, более удобные в конкретной ситуации. Смена разделителей обычно нужна, когда внутри шаблона очень много литералов, совпадающих с символом разделителя, которые нужно экранировать, что загромождает регулярное выражение.
- Разделители для регулярных выражений играют для интерпретатора важную роль: они помечают регулярное выражение в коде программы. Когда интерпретатор находит регулярное выражение, он компилирует его в микропрограмму. После второго разделителя может идти модификатор регулярного выражения, который управляет процессором регулярных выражений. Например модификатор /<шаблон>/i включает игнорирование регистра символов во входящем фрагменте.
- Короткие регулярные выражения обычно записываются одной строкой. Используя модификатор x, можно писать многострочные регулярные выражения, и даже писать в них комментарии, например:

```
# Парсит число, записанное в экспоненциальной форме.
/^
   [+-]?          # сначала находим необязательный знак
   (              # потом находим целое или мантиссу - число с плавающей запятой:
       \d+\.\d+   # мантисса типа a.b
```

```
        |\d+\.      # мантисса типа a.
        |\.\d+     # мантисса типа .b
        |\d+       # целое в виде a
    )
    ([eE][+-]?\d+)?  # в конце, опционально, находим экспоненту
$/x;
```

Группы

Для выборки подстрок во фрагментах, в Perl используются группы захвата (или просто группы). Группа представляет собой вложенный шаблон, который раскрывает характерную часть обрабатываемого фрагмента. Кроме групп захвата, есть ещё специальные группы, которые не захватывают результат, но задают условия обработки всего шаблона.

Группы Perl

(<шаблон>) *Простая группа с захватом*

Вложенное в основной шаблон регулярное выражение, на результат раскрытия которого можно сослаться из программы Perl.

(?:<шаблон>) *Группа без захвата*

Похожа на простую группу, но не захватывает результат раскрытия. Это полезно для экономии ресурсов и быстродействия, когда нужны преимущества групп (например, группировка альтернатив), но при этом не нужно ссылаться на результат. Между восклицательным знаком и двоеточием допустимо использование одного из следующих флагов: i (игнорировать регистр), m (обрабатывать фрагмент в многострочном режиме), s (обрабатывать фрагмент в однострочном режиме), x (разрешается использовать в шаблонах пробелы и комментарии через символ #). Если перед модифицирующим флагом поставить -, то это будет означать отключить этот флаг для этой группы.

(?#<текст>) *Комментарий*

Комментарий в шаблоне. Обычно используется, чтобы пояснить для разработчиков некоторую часть в шаблоне. Все комментарии игнорируются процессором регулярных выражений.

(?=<шаблон>) *Специальная группа с положительной опережающей проверкой*

Продолжает обработку шаблона, только если справа от текущей позиции во фрагменте раскрывается указанный шаблон.

(?!<шаблон>) *Специальная группа с отрицательной опережающей проверкой*

Продолжает обработку шаблона, только если справа от текущей позиции во фрагменте не раскрывается указанный шаблон.

(?<=<шаблон>) *Специальная группа с положительной ретроспективной проверкой*

Продолжает обработку шаблона, только если слева от текущей позиции во фрагменте раскрывается указанный шаблон.

(?<!<шаблон>) *Специальная группа с отрицательной ретроспективной проверкой*

Продолжает обработку шаблона, только если слева от текущей позиции во фрагменте не раскрывается указанный шаблон.

Операции с регулярными выражениями

Операция поиска

Операция поиска

```
m/<шаблон>/[imsxg]   # буква 'm' от слова matched

# Допустима и короткая форма
/<шаблон>/[imsxg]

# Примечание: буква опускается, когда разделитель не изменяется.
```

осуществляет поиск по шаблону регулярного выражения. По умолчанию, она просматривает переменную $_, но если она используется с =~ или !~, то просматривается левый операнд этих операций. Операция поиска возвращает 1 (ИСТИНА), если что-то было найдено, и пустую строку, если ЛОЖЬ.

Символ / (как и в других подобных операциях) является разделителем. Он может быть заменен на любой другой символ, либо пару скобок. Если символ разделителя встречается в шаблоне, то он должен быть экранирован.

```
m!<шаблон>!
m(<шаблон>)
m{<шаблон>}
```

Операцией поиска можно управлять с помощью флагов:

- i – игнорировать регистр;
- m – обрабатывать входящий фрагмент в многострочном режиме;
- s – обрабатывать входящий фрагмент в однострочном режиме;
- x – разрешает использование пробелов и комментариев в шаблоне;
- g – по умолчанию поиск осуществляется только для первого вхождения во фрагменте; данный флаг позволяет искать все вхождения. В этом режиме возвращаемое значение зависит от контекста исполнения. Если в шаблоне есть группы поиска, то в списковом контексте будут возвращаться списки для каждой группы. Если в образце нет групп поиска, то будет возвращен список всех вхождений. В скалярном контексте, каждый вызов этой операции с флагом g будет возвращать следующее вхождение в том же фрагменте.

Примеры

```
# Где может стоять операция
$_ = "A string";
print "Matched\n" if /[a-z]/i;     # В модификаторе

if ("A string" =~ /[a-z]/i) {
    print "Matched\n";             # Фрагмент является строкой
}

if ("A string" !~ /\d+/) {
```

```
    print "Matched\n";              # Фрагмент не является числом
}

# Шаблон регулярного выражения может быть подставлен из скаляра
$template = "[a-z]";
print "Matched\n" if "A string" =~ /$template/i;
```

Примеры регулярных выражений

Следующие примеры призваны показать основы регулярных выражений. В правой колонке показано, какая часть входящего фрагмента захватывается процессором.

Простые регулярные выражения	
`# Точное совпадение` `print "Matched: '$&'\n" if ($input = "bbbbaaaaccccc") =~ /aaaa/;`	bbbbaaaaccccc
`# Жадный квантификатор '+' захватит всю серию букв 'aaaa'` `print "Matched: '$&'\n" if ($input = "bbbbaaaaccccc") =~ /a+/;`	bbbbaaaaccccc

`# Квантификация по {} по умолчанию жадная`
`print "Matched: '$&'\n" if ($input = "bbbbaaaaccccc") =~ /a{1,4}/;`
`print "Matched: '$&'\n" if ($input = "bbbbaaaaccccc") =~ /a{1,3}/;`
	bbbbaaaaccccc
	bbbbaaaaccccc

`# Ленивая квантификация {}? захватывает серию по меньшей длине`
`print "Matched: '$&'\n" if ($input = "bbbbaaaaccccc") =~ /a{1,4}?/;`
`print "Matched: '$&'\n" if ($input = "bbbbaaaaccccc") =~ /a{1,3}?/;`
`# Аналогично`
`print "Matched: '$&'\n" if ($input = "bbbbaaaaccccc") =~ /a+?/;`
	bbbbaaaaccccc
	bbbbaaaaccccc
	bbbbaaaaccccc

Альтернативы	

`print "Matched: '$&'\n" if ($input = "aaaabbbbcccc") =~ /a+|b+|c+/;`
`# Обратите внимание, что из всех альтернатив захватывается та, которая встречается`
`# во фрагменте раньше всех`
`print "Matched: '$&'\n" if ($input = "aaaabbbbcccc") =~ /c+|b+|a+/;`
	aaaabbbbcccc
	aaaabbbbcccc

Использование классов символов	

`# Захватываем только буквы через класс произвольных символов []`
`print "Matched: '$&'\n" if ($input = "1234abcd5678") =~ /[a-z]+/;`
`# В данном случае еще игнорируем регистр через модификатор шаблона`
`print "Matched: '$&'\n" if ($input = "1234AbCd5678") =~ /[a-z]+/i;`
	1234abcd5678
	1234AbCd5678

`# Использование класса \d захватит первые цифры.`
`print "Matched: '$&'\n" if ($input = "1234abcd5678") =~ /\d+/;`
 	
`# Примечание: для захвата всех цифр нужно использовать другие подходы.`
	1234abcd5678

Границы слов и фрагментов	

`# Захватит первое слово в строке`
`print "Matched: '$&'\n" if ($input = "one two three") =~ /\b\w+\b/;`
`# В этом примере это аналогично следующему выражению (^ означает левую границу фрагмента)`
`print "Matched: '$&'\n" if ($input = "one two three") =~ /^\w+\b/;`
 	
`# В этом примере для захвата второго слова нужна его характерная часть в шаблоне, тем не менее`
`# существуют методы захвата слов по порядку (см. ниже).`
`print "Matched: '$&'\n" if ($input = "one two three") =~ /\bt\w*\b/;`
 	
`# Захват третьего слова ($ означает правую границу фрагмента)`
`print "Matched: '$&'\n" if ($input = "one two three") =~ /\b\w+$/;`
	one two three
	one two three
	one two three
	one two three

```perl
# В однострочном режиме работы процессора регулярных выражений следующие два шаблона
# будут работать одинаково, так как \A совпадает с ^ и $ совпадает с \Z
print "Matched: '$&'\n" if ($input = "one two three") =~ /\A[\w\s]+\Z/;
print "Matched: '$&'\n" if ($input = "one two three") =~ /^[\w\s]+$/;
```

```
one two three
one two three
```

Многострочный и однострочный режим работы процессора регулярных выражений

```perl
# Флаг 'm' включает многострочный режим работы процессора, в котором переносы строк
# во фрагменте делят его на отдельные строки (подфрагменты).
# В этом режиме ^ и $ означают начало и конец подфрагмента.
print "Matched: '$&'\n" if ($input = "one\ntwo\nthree") =~ /^t\w*$/m;
```

```
one
two
three
```

```perl
# В однострочном режиме фрагмент обрабатывается целиком, и значения для пар метасимволов
# ^, \A и $, \Z совпадают.
print "Matched: '$&'\n" if ($input = "one\ntwo\nthree") =~ /\A[\w\s]+\Z/;
# Результат этого выражения аналогичен предыдущему
print "Matched: '$&'\n" if ($input = "one\ntwo\nthree") =~ /\A[\w\s]+\Z/s;
```

```
one
two
three
```

Группы захвата

Группы захвата регулярных выражений глубоко интегрированы в язык Perl, что позволяет вам записывать очень компактные конструкции для извлечения подстрок. Для начала посмотрим, как обращаться к группам.

```perl
print "Matched: '$&'\n" if ($input = "baba") =~ /(\w?)(\w?)\1\2/;
print "Matched: '$&'\n" if ($input = "baba") =~ /(\w?)(\w?)\g1\g2/;   # Аналогично
print "Group 1: '$1'; Group 2: '$2'\n";
```

```
Matched: 'baba'
Matched: 'baba'
Group 1: 'b'; Group 2: 'a'
```

Группа захвата является вложенным шаблоном регулярного выражения, который раскрывается один раз и на результат которого можно сослаться. Вы можете ссылаться на группу в самом регулярном выражении по обратным ссылкам \n (где n – порядковый номер группы), например, чтобы пометить повторяющиеся части фрагмента (это показано в примере), либо вы можете сослаться на последние результаты раскрытия шаблонов в группах через специальные переменные $1, $2, $3 и так далее, где число также означает номер группы. Группы нумеруются слева направо, начиная с единицы, причем для вложенных групп применяется сквозной принцип нумерации, снаружи внутрь. Это можно увидеть на следующем примере:

```perl
print "Matched: '$&'\n" if ($input = "aabbaa") =~ /(\w{2})((b{1}){2})\1/;
print "Group 1: '$1'; Group 2: '$2'; Group 3: '$3'\n";
```

```
Matched: 'aabbaa'
Group 1: 'aa'; Group 2: 'bb'; Group 3: 'b'

# Третья группа вложена во вторую.
```

Начиная с версии интерпретатора 5.10 можно создавать именованные группы, к которым можно обращаться не через числовой идентификатор, а через строковый. Именованные группы захвата автоматически помещаются в хеш %+.

```
# Первый вариант (?'имя-группы'шаблон)
print "Matched: '$&'\n" if ($input = "aaaa") =~ /(?'group_1'\w{2})\g{group_1}/;
print "$+{group_1}\n";

# Второй вариант (?<имя-группы>шаблон)
print "Matched: '$&'\n" if ($input = "bbbb") =~ /(?<group_1>\w{2})\g{group_1}/;
print "$+{group_1}\n";
```

```
Matched: 'aaaa'
aa
Matched: 'bbbb'
bb
```

Также начиная с 5.10 допустима относительная нумерация групп через отрицательные числа. В этом случае относительно позиции в шаблоне ссылка по типу `\g{-1}` будет означать последняя существующая группа, `\g{-2}` — предпоследняя и так далее.

```
print "Matched: '$&'\n" if ($input = "aaa") =~ /(a)\1\g{-1}/;
# Такое регулярное выражение аналогично /(a)\1\1/, т.е. \g{-1} = \g{1} в этом примере
print "Group 1: $1, Group 2: $2\n";
```

```
Matched: 'aaa'
Group 1: a, Group 2:

# Группы 2 не существует.
```

С версии 5.10 также можно вмешиваться в нумерацию вложенных групп, если внутри есть альтернатива:

```
# Благодаря (?|...), номера групп внутри каждой альтернативы сохранятся, как 1 и 2. Без этого была бы
# применена сквозная нумерация. Следующая группа за (?|...) будет иметь номер 3.
print "Matched: '$&'\n" if ($input = "13:30") =~ /(?|(\d\d):(\d\d)|(\d\d) (\d\d))/;
print "Hours: $1, Minutes: $2\n";

print "Matched: '$&'\n" if ($input = "12 15") =~ /(?|(\d\d):(\d\d)|(\d\d) (\d\d))/;
print "Hours: $1, Minutes: $2\n";
```

```
Matched: '13:30'
Hours: 13, Minutes: 30
Matched: '12 15'
Hours: 12, Minutes: 15
```

Процессор регулярных выражений ведет для групп захвата два массива. В первый массив @- записываются начальные позиции найденных подстрок в оригинальном фрагменте, а во втором массиве @+ — конечные позиции, причем вы сопоставляете начальные и конечные позиции по индексам. Если в шаблоне есть группы захвата, то первая группа будет ассоциироваться с индексом 1, вторая — 2 и так далее. В элементе 0 лежат позиции для всего шаблона.

```
$str = "aabbaa";
$str =~ /(a){2}(b){2}\1*/;
foreach $expr (0..$#-) {
    $group = $expr == 0 ? "" : "'${$expr}', ";
    print "Found $expr: $group positions ($-[$expr],$+[$expr]), ";
    print "using positions: " . substr( $str, $-[$expr], $+[$expr]-$-[$expr] ) . "\n";
}
```

```
Found 0:  positions (0,6), using positions: aabbaa
Found 1: 'a',  positions (1,2), using positions: a
Found 2: 'b',  positions (3,4), using positions: b
```

Таким образом, используя значения позиций, можно извлекать подстроки с помощью функции substr():

```
substr( $input, 0, $-[0] )             # аналогична $`
substr( $input, $-[0], $+[0]-$-[0] )   # аналогична $&
substr( $input, $+[0] )                # аналогична $'
```

Глобальный поиск

По умолчанию операция поиска находит только первое вхождение по регулярному выражению. Модификаторы g и c служат для глобального поиска во фрагменте.

Регулярное выражение с модификатором g будет последовательно возвращать результаты раскрытия шаблона, перемещая позицию начала поиска за последний удачный результат. Получить или задать позицию вы можете с помощью функции pos(). Неудачный поиск или изменение целевого фрагмента сбрасывают позицию, т.е. текущая позиция связывается именно с фрагментом, в котором происходит поиск, а не с регулярным выражением. Чтобы позиция в этих случаях не сбрасывалась, используйте дополнительно модификатор c (//gc).

```
$input = "one two three";
# В скалярном контексте каждый новый поиск возвращает новое вхождение.
while ($input =~ /(\w+)/g) {
    print "Found '$1', position " . pos($input) . "\n";
}
```

```
Found 'one', position 3
Found 'two', position 7
Found 'three', position 13
```

В списковом контексте глобальный поиск возвращает массив всех найденных групп, либо массив всех совпадений по основному шаблону.

```
$input = "John-1 Monty-2 Alice-3";

@words = ($input =~ /(\w+)-(\d+)/g);   # Списковый контекст

foreach $element (@words) {
    print "$element\n";
}
```

```
John
1
Monty
2
Alice
3
```

Глобальный поиск очень тесно связан с метасимволом \G, который отмечает позицию в шаблоне, с которой должен начаться следующий поиск. Очень часто это нужно, когда в одной строке нужно сопоставлять шаблоны только с определёнными токенами, а остальные нужно игнорировать. Конечно можно попробовать составить регулярное выражение и без этого метасимвола, но оно будет в разы сложнее и медленнее. Рассмотрим такой пример:

```
$input = "Point 1: x=10 y=15 z=28; Point 2: x=23 y=25 z=39; Point 3: x=85 y=5 z=16; Point 4: x=5 y=6 z=15;";

@words = ($input =~ /(Point [23]:|\G) [xyz]?=(\d+)/g);

print "@words\n";
```

Во входящем фрагменте у нас есть 4 точки с тремя координатами для каждой. В этой задаче нам хотелось бы выделить вторую и третью точки, а все остальные опустить. Данное выражение справляется с такой ситуацией:

```
Point 2: 23  25  39 Point 3: 85  5  16
```

Когда процессор начинает работу, то шаблон первой группы вернет альтернативу \G для самого начала фрагмента, но, так как следующий шаблон не удовлетворяет следующему символу во фрагменте, то произойдет откат и процессор начнет заново. Рано или поздно процессор доберется до совпадения по первой группе `Point 2:` и вместе со второй группой захват будет `Point 2: x=23`. Следующая позиция за `Point 2: x=23` это пробельный символ, который совпадает с метасимволом \G, поэтому произойдет захват по второй группе `y=25` и аналогично для следующего прохода `z=39`. Аналогичные измышления применяются для `Point 3:`.

Не захватывающие и специальные группы

Как уже было упомянуто, не захватывающие группы (?:<шаблон>) работают быстрее захватывающих за счёт того, что процессору регулярных выражений не требуется инициализировать переменные в памяти под группы. Не захватывающие группы используются в одном из следующих случаев:

- когда нужно сгруппировать несколько альтернатив, но захват не нужен;
- когда нужны управляющие флаги для конкретной группы, но захват при этом не нужен;
- когда использование группы в данной ситуации удобно, но захват не нужен, например, чтобы он не попадал в результирующий массив в списковом контексте.

```
# Регистр хотим игнорировать, но захватывать слово не хотим
@words = (($input = "PERL: 5.32.0 Perl: 5.34.0") =~ /(?i:perl: )([\d\.]+)/g);
print "@words\n";

# Группировкой мы задаем разделители, но сами разделители захватывать не хотим
@numbers = split /(?:a|b)+/, "12aba34ba5";
print "@numbers\n";
```

```
5.32.0 5.34.0
12 34 5
```

Специальные группы все являются не захватывающими. Они являются аналогом условных выражений в шаблоне, т.е. они задают условие продолжения поиска в зависимости от существования или не существования во фрагменте определенной характерной части в зависимости от позиции в шаблоне.

Различают специальные группы с *опережающей* и *ретроспективной* проверками. При опережающей проверке шаблон проверяется для части фрагмента, который отстоит от группы справа, соответственно для ретроспективной – которая отстоит слева. В таких проверках важен не результат раскрытия, а факт этого раскрытия. Опережающие и ретроспективные проверки могут быть *положительными* и *отрицательными*. Соответственно условным успехом для положительной проверки считается раскрытие по шаблону в этой группе, а для отрицательной проверки успехом считается не раскрытие по шаблону в группе. Когда условный успех специальной группы достигнут, то она разрешает процессору регулярных выражений продолжать обработку фрагмента по оставшейся части шаблона (именно поэтому проверки и называются опережающими).

```
# Опережающая положительная проверка здесь означает "ищи серию [\d\.]+, если справа от нее стоит ' kg'".
@words = ("12 Mb; 45 kg; 135 m; 64 kg; 19 secs; 15.65 kg;" =~ /[\d\.]+(?=\skg)/g);
print "Kilograms: @words\n";
```

```
# Если сделать опережающую отрицательную проверку над тем же фрагментом, то найдем все остальное.
@words = ("12 Mb; 45 kg; 135 m; 64 kg; 19 secs; 15.65 kg;" =~ /[\d\.]+(?!\skg)/g);
print "Others: @words\n";

# Данный шаблон выдаст все карты Nvidia во фрагменте. Данный шаблон говорит "ищи серию ([\w\s]+?), если
# слева от нее стоит 'Nvidia: '".
@nvidias = ("Nvidia: GeForce GT 230M; Radeon: RX500;  Nvidia: GeForce GT 130M; Radeon: RX7000" =~ /(?<=Nvidia: )
([\w\s]+?);/g);
print "@nvidias\n";

# Выбираем цены только в российской валюте.
@rubles = ('₽30 ¥40 $50 £100 ₽100' =~ /(?<=₽)\d+/g );
print "@rubles\n";

# Выбираем остальные валюты
@rubles = ('₽30 ¥40 $50 £100 ₽100' =~ /\b(?<!₽{1})\d+\b/g );
print "@rubles\n";
```

```
Kilograms: 45 64 15.65
Others: 12 4 135 6 19 15.6
GeForce GT 230M GeForce GT 130M
30 100
40 50 100
```

Шаблоны в опережающих проверках имеют много ограничений, особенно на квантификацию, поэтому их следует использовать осторожно и в очень простых ситуациях. Рекомендуется использовать другие механизмы фильтрации, если дело доходит до опережающих проверок.

Рекурсивные шаблоны

Начиная с версии 5.10 в Perl можно использовать рекурсивные шаблоны, т.е. специальные группы ссылающиеся на другие шаблоны, чтобы повторить их в глубину (по подобию рекурсивного вызова процедуры). Это позволяет захватывать повторяющиеся серии за один вызов регулярного выражения, не прибегая к глобальному поиску.

Рекурсивно шаблоны могут вызываться так:

- (?R) – рекурсивно ссылается на внешний шаблон;
- (?<номер-группы>) – рекурсивно ссылается на группу, указанную по номеру;
- (?&имя-группы) – рекурсивно ссылается на группу по имени.

Простейший пример рекурсивного вызова:

```
"aaa111bbb222" =~ /\w{3}\d{3}(?R)?/;
print "$&\n";
```

```
'aaa111bbb222'

# Рекурсивный вызов захватил всю серию, т.е. рекурсивный вызов аналогичен в этом примере
# такому шаблону:
#
#     \w{3}\d{3}\w{3}\d{3}
#
```

Ещё один пример, в котором используется рекурсивный вызов шаблона, это поиск палиндрома (т.е. такой строки, которая одинаково читается как слева направо, так и справа налево):

```
# Примечание:
#   Saippuakivikauppias по-фински означает "продавец мыльного камня".
"Saippuakivikauppias" =~
    /^(?'palindrome'
        \W*
```

```
            (?:
                (\w) (?&palindrome) \g{-1} | \w?
            )
        \W*
    )$/ix;
print "'$1'";
```

Шаблон выглядит сложно, но основной идеей в нём является то, что палиндромами считаются такие строки, в которых крайние символы должны совпадать, а между ними должен стоять палиндром, либо строка должна состоять из одного символа. Рекурсивным вызовом мы добиваемся того, что процессор проверит все возможные серии из символов, оставив такую (если она есть), которая удовлетворяет приведенному условию.

Операция замены

Операция замены

```
s/<шаблон>/<замена>/[egimosx]
```

осуществляет поиск по шаблону регулярного выражения и, в случае нахождения фрагмента, заменяет его текстом из <замена>. Операция возвращает число сделанных замен или пустую строку, если замен не было. По умолчанию поиск происходит из $_. Данную операцию можно использовать с =~ и !~, тогда в качестве строки берется левый операнд этих операций.

Флаги операции:

- g включает глобальный режим замены;
- e подсказывает операции, что часть <замена> является Perl-выражением, которое нужно предварительно вычислить.

```
print "$input\n" if ($input = "one two") =~ s/(\S+) (\S+)/$2 $1/;   # Меняем местами два слова в строке.

# Заменяем слово звездочками под размер строки
$password = "secret word";
$password =~ s/$password/'*' x length($password)/e;
print "Replace password with stars: $password\n";
```

```
two one
Replace password with stars: ***********
```

Операция транслитерации

Операция транслитерации

```
tr/<список-поиска>/<список-замены>/[cds]
y/<список-поиска>/<список-замены>/[cds]  # как в Sed
```

преобразует каждый символ из списка поиска в соответствующий символ из списка замены и возвращает число преобразованных символов. По умолчанию преобразования осуществляются для переменной $_. Для операций =~ и !~ используется левый операнд.

Оба списка операции задаются через серию символов и могут содержать диапазоны с разделяющим знаком -.

Флаги:

- c – использовать вместо списка поиска его дополнение до основного множества символов;
- d – удаляет из списка поиска все символы, для которых нет соответствия в списке замены. Если этот флаг не установлен и список замены короче списка поиска, то вместо недостающих символов в списке замены используется последний символ этого списка;
- s – все последовательности символов, которые были преобразованы в один и тот же символ, заменяются одним экземпляром этого символа.

```
$string = "hello world";

$string =~ tr/[a-z]/[A-Z]/;    # Преобразуем все слова в верхний регистр
print "$string\n";

$string =~ tr/[A-Za-z]/*/;     # Заменяем буквы звездочками
print "$string\n";

$string = "A string with spaces";
$string =~ tr/ /_/;            # Заменяем пробелы символом нижнего подчеркивания
print "$string\n";
```

```
HELLO WORLD
***** *****
A_string_with_spaces
```

Операция заключения в кавычки

Как мы уже говорили выше, регулярные выражения компилируются интерпретатором в микропрограммы. Операция qr// позволяет объявить в произвольном месте программы регулярное выражение, которое скомпилируется один раз, и которое можно использовать сколько угодно раз. Преимуществом от этого является ускорение компиляции и следовательно выполнение программы интерпретатора.

```
# Это регулярное выражение компилируется один раз и ...
$regex = qr/[\w\s]+/i;

$input = "A string";

print "Matched\n" if $input =~ $regex;    # ... используется здесь, ...
print "Matched\n" if $input =~ /$regex/;  # ... здесь (аналогично предыдущему оператору) ...
print "Matched, $1\n" if $input =~ /($regex)+/; # ... и здесь, в составе другого регулярного выражения.
```

Функции для работы со строками

В этом разделе перечислены самые часто используемые функции для работы со строками. Некоторые из них в качестве аргумента способны принимать регулярные выражения.

```
chop [<список>]
```

Удаляет последний символ из всех слов, переданных списком скаляров, и возвращает последний удаленный символ. Если список не был передан, то используется переменная $_. Функция изменяет строки по месту их хранения, поэтому функции не имеет смысл передавать строковые литералы. Функция не выводит никаких ошибок, если она неправильно используется, и нормально реагирует на пустые строки. Данная функция часто используется при разделении сложной строки по разделителю, когда сам символ разделителя остается в разделенных словах последним.

```
chomp [<список>]
```

Функция похожа на chop(), но удаляет только символ, хранящийся на текущий момент в переменной $/ (обычно это символ переноса строки), если конечно он есть во входящей строке.

```
length <выражение>
```

Возвращает длину вычисленного в выражении скаляра в байтах.

```
lc <выражение>
```

Возвращает строку, полученную в результате вычисления выражения, в которой все символы приведены к нижнему регистру.

```
uc <выражение>
```

Возвращает строку, полученную в результате вычисления выражения, в которой все символы приведены к верхнему регистру.

```
lcfirst <выражение>
```

Возвращает строку, полученную в результате вычисления выражения, в которой только первый символ приведен к нижнему регистру.

```
ucfirst <выражение>
```

Возвращает строку, полученную в результате вычисления выражения, в которой только первый символ приведен к верхнему регистру.

```
join <выражение>, <список>
```

Объединяет слова из списка в одну строку, используя результат выражения в качестве разделителя.

```
split [/<шаблон>/[, <выражение>[, <предел>]]]
```

Разбивает строку, полученную в результате вычисления выражения, на отдельные строки, используя в качестве разделителя шаблон регулярного выражения. В списковом контексте возвращает массив из полученных слов, а в скалярном — количество полученных слов. В скалярном контексте результат разбиения не теряется, а помещается в массив @_. Об этом следует помнить, когда функция вызывается внутри процедуры, так как она нечаянно может затереть массив с аргументами процедуры. Если выражение опущено, то функция использует переменную $_. Если опущен шаблон, то используется неявно /\s+/.

Если задан <предел>, то общее число слов в результате не будет превышать этого числа, причем отрицательные числа будут означать отсутствие предела.

```
index <строка>, <подстрока>[, <позиция>]
```

Находит первое вхождение подстроки в строке, начиная с указанной позиции. Если позиция не задана, то по умолчанию берется значение из переменной $[, которое обычно равно 0. Вообще переменная $[используется, чтобы указать на то, что считать самым первым элементом в любом массиве и первым символом в любой строке, поэтому не рекомендуется её изменять, чтобы не допустить скрытых ошибок. Если функция ничего не находит, то возвращает -1.

```
rindex <строка>, <подстрока>, <позиция>
```

Похожа на index(), но возвращает последнее вхождение подстроки в строке. Причем позиция будет играть ограничителем справа.

```
substr <выражение>, <смещение> [, <длина> [, <замена>]]
```

Возвращает из строки, полученной выражением, подстроки, ограниченной слева смещением, а справа суммой смещения и длины. Если длина опущена, то справа строка ограничивается всей её фактической длиной. Отрицательное число для длины задаёт число символов отсчитываемых от конца, которые не попадут в результирующую строку. Отрицательное смещение также будет отсчитывать начальный символ от фактического конца строки. Аргумент <замена> указывает скаляр, на значение которого нужно заменить выделенную подстроку. Причем последнего можно достигнуть и через lvalue выражение, сравните:

```perl
$input = "A TOKEN";
substr($input, 2, 6, "string");
print "$input\n"; # "A string"

# Аналогично через lvalue-выражение
$input = "A TOKEN";
substr($input, 2) = "string";
print "$input\n"; # "A string"
```

```
pos [<скаляр>]
```

Эту функцию мы уже упоминали, когда говорили о глобальном поиске. Она возвращает позицию, на которой завершился последний поиск в той строке, с которой связывалась операция поиска:

```perl
$scalar =~ m/.../g;
$position = pos($scalar);

# Значение в $position будет вычислено аналогично следующему выражению
#
#   $position = length($`) + length($&);
#
```

Если аргумент у функции отсутствует, то по умолчанию берется переменная $_. Функцию можно использовать и в lvalue-выражении для принудительного задания

новой позиции, с которой продолжится глобальный поиск:

```
$words = "one two three four";
pos $words = 4;

while ($words =~ m/\w+/g) {
    print pos $words, " - " , substr( $words, $-[0], $+[0]-$-[0] ) , "\n";
}
# Вывод:
#
# 7 - two
# 13 - three
# 18 - four
```

```
quotemeta [<выражение>]
```

Возвращает строку, вычисленную выражением, в которой все символы, кроме алфавитно-цифровых и символа нижнего подчеркивания, экранированы обратным слешем \. Обычно используется над строками, которые передаются транзитом в другой интерпретатор, чтобы экранировать те символы, которые нужно понимать буквально.

← Пакеты, библиотеки и модули Приложения →