(/)

Николай Мишин / POD2-RU-5.18.0.1.84 / perlre

Contents [hide]

NAME

perlre - Регулярные выражения Perl

ОПИСАНИЕ

Эта страница описывает синтаксис регулярных выражений в Perl.

Если вы раньше не использовали регулярные выражения, то есть быстрое введение в perlrequick и более длинное в perlretut.

О том, как регулярные выражения используются в поиске по шаблону, а также различные примеры см. обсуждение m// , s/// , qr// и ?? в "Regexp операторы заключения в кавычки" in perlop.

Модификаторы

Операторы поиска по шаблону имеют разные модификаторы. Модификаторы, которые относятся к интерпретации внутри регулярных выражений перечислены ниже. Модификаторы, которые изменяют путь регулярного выражения, использующегося в Perl, детализированы в "Regexp операторы заключения в кавычки" in perlop и "Внутренние детали парсинга конструкций в кавычках" in perlop.

m

Представляет строку поиска, как многострочную. Разрешает метасимволам ^ и $ привязываться к промежуточным символам \n, имеющимся в тексте.

s

Метасимволу "точка" разрешено соответствовать \n.(s - single line)

Используя вместе /ms позволяет "." соответствовать любому символу включая новую строку а символы "^" и "$" становятся началом и концом всего буфера.

i

Игнорирует регистр символов при сопоставлении с образцом.

Если правила поиска локали включены, текущая карта символов берется из текущей локали для символов с кодом менее 255 и используются юникодные правила для больших кодовых точек. Однако поиск, который будет пересекать правила Unicode / не Unicode на границе (Ords 255/256) не удастся. См. perllocale.

Есть целый ряд символов Юникода, которым соответствуют несколько символов под /i Например, LATIN SMALL LIGATURE FI должен найти последовательность /fi Perl в настоящее время не в состоянии сделать

это, когда несколько символов в шаблоне и они разделены между группировками, или когда один или больше квантификаторов. Таким образом (/)

```
use charnames ':full'; #примечание переводчика
```

```
"\N{LATIN SMALL LIGATURE FI}" =~ /fi/i;          # найдет
"\N{LATIN SMALL LIGATURE FI}" =~ /[fi][fi]/i;    # Не найдет!
"\N{LATIN SMALL LIGATURE FI}" =~ /fi*/i;         # Не найдет!

# Ниже не будет соответствия, и не ясно, что было бы в $1 и $2
# будьте честны, если это произойдет
"\N{LATIN SMALL LIGATURE FI}" =~ /(f)(i)/i;      # Не найдет!
```

Perl не находит несколько символов в скобках (образующих класс), если только знак, который отображается на них прямо упоминается, и он не находит вообще, если класс символов инвертирован, что в противном случае могло бы быть очень запутанно. См. "Bracketed Character Classes" in perlrecharclass, and "Negation" in perlrecharclass.

x

Увеличивает читаемость вашего шаблона для поиска, разрешаются пробелы и комментарии. Подробности в "/x"

p

Сохраненные части строки, такие как ${^PREMATCH}, ${^MATCH}, и ${^POSTMATCH} доступны для использования после поиска по шаблону.

g и c

Глобальный поиск совпадений, и сохранение текущей позиции после неудачного поиска. В отличие от i, m, s и x,эти два флага влияют затрагиваем способ, которым regex используется, а не сам regex. См. "Using regular expressions in Perl" in perlretut (Использование регулярных выражений Perl)для получения большей информации о модификаторах g и c.

a, d, l и u

Эти модификаторы, все новые в 5.14, влияют при какая семантике кодировки (Unicode, и т.д.) используются, как описано ниже в L </модификаторы Кодировки>.}

Модификаторы регулярных выражений обычно описываются в документации, например,как " /x модификатор" несмотря на косую черту, разделителем может быть не только слэш. Модификаторы /imsxadlup могут также быть включены в пределах самого регулярного выражения, используя конструкцию (?...) , см. "Extended Patterns" ("Расширенные Образцы") ниже.

/x

/x(/) говорит регулярному анализатору выражения игнорировать большую часть пробелов, перед которыми не стоит обратный слэш и не входящих в класс символов. Вы можете использовать это, чтобы разбить ваше регулярное выражение на части, чтобы сделать его более читаемым. Символ # также рассматривается как метасимвол, являющимся комментарием, как в обычном коде Perl. Это также означает, что если вы хотите реальный пробел или символ # в шаблоне поиска (вне класса характера, где они не затронуты /x), тогда вы должны будете или эскейпить их (используя наклонную черту влево или \Q...\E) или кодировать их используя восьмеричный, шестнадцатеричный или \N{} код. Взятые вместе, эти фичи (features) имеют большое значение для создания более читаемых регулярных выражений в Perl. Обратите внимание, что вы должны быть осторожны, чтобы не включать разделитель шаблона в комментарий --perl не имеет никакого способа узнать, что вы не намеревались закрыть шаблон раньше. См. удаление комментариев в C коде в perlop. Также отметим, что все внутри \Q...\E остается не зависимым от /x. И заметьте, что /x не влияет на интерпретацию пространства в пределах одной многосимвольной конструкции. Например, в \x{...}, независимо от модификатора /x, не может быть пробелов. То же самое для quantifier таких как {3} или {5,}. Аналогично, (?:...) не может иметь пробелов между (, ?, и :. В пределах любых разделителей для такой конструкции, позволенные пробелы не имеют силы из-за /x и зависят от конструкции. Например, \x{...} не должно быть пробелов, потому что шестнадцатеричные числа не имеют в себе пробелов. Но, у свойств Unicode (Unicode properties) могут быть пробелы, таким образом, в \p{...} могут быть пробелы, которые следуют за правилами Unicode, для которых см. "Properties accessible through \p{} and \P{}" in perluniprops (L <perluniprops/Свойства, доступные через \p{} и \P{}>)

Модификаторы Кодировки

/d, /u, /a, и /l, доступные, начиная с 5.14, называют модификаторами кодировки; они затрагивают семантику кодировки, используемую для регулярного выражения.

Модификаторы /d, /u, и /l вряд ли будут иметь большое применения вами, и таким образом, вы не должны о них очень волноваться. Они существуют для внутреннего использования Perl, так, чтобы сложные структуры данных регулярных выражений могли быть автоматически преобразованы в последовательную форму и позже точно воссозданы, включая все их нюансы. Но, так как Perl не может хранить все в тайне, и могут быть редкие случаи, где они полезны, они задокументированы здесь.

Модификатор /a, с другой стороны, может быть полезным. Его цель заключается в том, что он позволяет коду работать главным образом на данных ASCII, чтобы не иметь проблем с Unicode.

Кратко, /l задает кодировку, которая будет работать в любой <L>ocale во время выполнения поиска по шаблону.

/u задает кодировку Юникод.

/a также устанавливает набор символов для Юникода, но добавляет несколько ограничений ASCII-безопасного поиска.

/d является старым, проблематичным, перед-5.14 Набором по умолчанию символов поведение. Используется только для этого старого поведения.

В любой момент времени только один из этих модификаторов действует. Их существование позволяет Perl сохранить первоначально скомпилированное поведение регулярного выражения, независимо от действий правил, когда оно действительно выполняется. И, если он интерполируется в больших регекс, оригинал в правила продолжают применяться к нему и только к нему.

/l и /u модификаторы автоматически выбираются для регулярных выражений, скомпилированных в рамках различных прагм, и мы рекомендуем, в общем, чтобы вы использовали эти прагмы вместо явного указания этих модификаторов. Для одной вещи, модификаторы затрагивают только шаблоны поиска и не распространяются даже на сделанную замену , тогда как использование прагма-директивы даёт постоянные результаты для всех соответствующие операций в пределах их области. Например,

```
s/foo/\Ubar/il
```

найдет "foo", используя локальные правила языкового стандарта для сопоставления без учета регистра, но /l не влияет, когда работает C <\U>. Скорее всего вы хотите, чтобы оба из них использовали локальные правила. Для этого нужно компилировать регулярное выражение в рамках use locale. Это неявно добавляет /l и применяет правила языка к C <\U>. Урок заключается в том, чтобы использовать use locale и не использовать явно /l.

Аналогичным образом было бы лучше использовать use feature 'unicode_strings' вместо

```
s/foo/\Lbar/iu
```

чтобы получить правила Юникода, как в бывшем \L (но не обязательно последнем) будет также использовать правила Юникода.

Более подробно о каждом из модификаторов смотри далее. Скорее всего вам не нужно знать эти детали для /l, /u и C</d> и можете пропустить вперед /a.

/l

используется для правил текущей локали (см. perllocale) когда шаблон найден. Например `\w` будет совпадать с символами "слова" текущей локали и `"/i"` регистронезависимый поиск будет искать согласно текущим правилам регистронезависимого сравнения. Будет использоваться одна локаль во время поиска по шаблону. Это может быть не то же самое, как во время локали времени компиляции и поиск может отличаться один от другого если еще есть промежуточный вызов из setlocale() function.

Perl поддерживает только однобайтовые локали. Это означает, что код символа, который выше 255, рассматриваются как Юникод независимо от того, какая локаль действует. Согласно правилам Юникода есть несколько локалей без учета регистра, которые пересекают границу 255/256 . Они запрещены при `/l` . Например, 0xFF (на ASCII платформах) не соответствует символу 0x178, LATIN CAPITAL LETTER Y WITH DIAERESIS , в подшаблоне , потому что 0xFF не может быть LATIN SMALL LETTER Y WITH DIAERESIS в текущей локали, и Perl не имеет возможности узнать даже существует ли этот символ в этой локали.

Этот модификатор можно определить, используя `use locale` , но посмотрите "Какой набор символьных модификаторов действует?".

/u

означает использования правил Unicode во время поиска по шаблону. На платформах ASCII это означает, что символы с кодом от 128 до 255 такие, как Latin-1 (ISO-8859-1) значения (которые такие же, как в Юникоде). (Иначе Perl считает, что их значение будет неопределенным). Таким образом, под этот модификатором платформа ASCII фактически становится Юникодной платформой; и следовательно, к примеру, `\w` будет соответствовать любому из более чем 100_000 буквенных символов в Юникоде.

В отличие от большинства локалей, которые являются специфическими для пары язык-страна, Юникод классифицирует все символы, как символ для письма, которые используются *где угодно* в мире, как `\w` . Например, ваша локаль может не считать LATIN SMALL LETTER ETH как строковый символ (пока вы не будете говорить на исландском), а Юникод принимает этот символ за строчный. Аналогичным образом все символы, десятичные цифры везде в мире будут соответствовать `\d` ; это сотни, а не 10-ки возможных совпадений. И некоторые из этих цифр выглядят, как некоторые из 10 ASCII цифр, но означают другое число, поэтому человек легко может думать, что это совсем другое число, чем на самом деле. Например, BENGALI DIGIT FOUR (U+09EA) выглядит очень похоже на ASCII DIGIT EIGHT (U+0038). И `\d+` может найти символы , которые могут соответствовать символам цифр, которые представляют собой смесь из различных систем письма, созданных из соображений

безопасности. "num()" in Unicode::UCD может использоваться для сортировки. А модификатор `/a` может использоваться для принудительного поиска шаблоном `\d`(/) только ASCII цифр от 0 до 9.

Кроме того под этим модификатором, регистронезависимый поиск символов работает на полном наборе Юникодных символов. `KELVIN SIGN`, например совпадает с буквами "k" и "K"; и `LATIN SMALL LIGATURE FF` соответствует последовательности "ff", которая, если вы не подготовлены, может выглядеть, как шестнадцатеричная константа, что представляет еще одну потенциальную проблему безопасности. См. http://unicode.org/reports/tr36 для подробного обсуждения вопросов безопасности в Юникоде.

Этот модификатор может определяться по умолчанию, используя `use feature 'unicode_strings`, `use locale ':not_characters'`, или `use 5.012` >> (или выше), но посмотрите главу "Какой набор символьных модификаторов действует?".

/d

Этот модификатор означает использование собственные правила платформы "По умолчанию", за исключением случаев, когда вместо этого есть основания для использования правил Юникода, следующим образом:

1. целевая строка кодируется в UTF-8; или
2. шаблон кодируется в UTF-8; или
3. шаблон явно упоминает код символа, который находится выше 255 (скажем `\x{100}`); или
4. шаблон использует Юникодное имя (`\N{...}`); или
5. шаблон использует Юникодное свойство (`\p{...}`); или
6. шаблон использует " `(?[ ])` "

Другая мнемоника для этого модификатора "Зависит от", и правила, которые на самом деле используются зависят от различных вещей, и в результате вы можете получить неожиданные результаты. Смотрите "<Bug Юникода" in perlunicode. Ошибка Unicode становиться довольно печальной, ведущих к еще одному (версия для печати) имени для этого модификатора, "Dodgy" "Непредсказуемый" или "Рисковый".

Если шаблон или строки не кодируются в UTF-8, то буду найдены только символы ASCII.

Вот некоторые примеры того, как это работает на ASCII платформе:

```
$str =  "\xDF";      # $str не в UTF-8 формате.
$str =~ /^\w/;       # Не найдет, так как $str не в UTF-8 формате.
$str .= "\x{0e0b}"; # Теперь строка $str в UTF-8 формате.
$str =~ /^\w/;       # Найдет! $str теперь в UTF-8 формате.
chop $str;
$str =~ /^\w/;       # Все еще найдет! $str остается в UTF-8 формате.
```

Этот модификатор автоматически выбирается по умолчанию, если ни один другой не указан , таким образом для него другое имя – "По умолчанию" ("Default"). (/)

Из-за неожиданного поведения, связанные с этим модификатором, вам вероятно следует его использовать только, его вам нужно сохранить странные обратные совместимости.

/a (и /aa)

Этот модификатор выступает за ASCII строгость (или ASCII-безопасность). Этот модификатор, в отличие от других, может в два раза расширить свой эффект.

Когда он появляется один, то он применяется к последовательности `\d`, `\s`, `\w`, и классам символов Posix для соответствия только в диапазоне ASCII. Они, таким образом, возвращаются к версиям до 5.6 (pre-5.6), бывшим еще до значений Юникода (pre-Unicode). Под модификатором `/a`, `\d` всегда означает только цифры от `"0"` до `"9"`; `\s` означает пять символов `[ \f\n\r\t]`, и начиная с Perl v5.18, экспериментально, вертикальный табулятор; `\w` означает 63 символа `[A-Za-z0-9_]`; кроме того, все Posix классы такие как `[[:print:]]` найдут только соответствующие символы ASCII-диапазона.

Этот модификатор является полезным для людей, которые только изредка используют Юникод, и, которые не желают обременяться его сложностью и проблемами безопасности.

С модификатором `/a`, вы можете написать `\d` с уверенностью, что он точно найдет только ASCII символы, и, если вам нужно найти что-то за пределами символов ASCII, вы можете вместо этого использовать `\p{Digit}` (или `\p{Word}` для `\w`). Есть аналогичные конструкции `\p{...}`, которые могут соответствовать символам за пределами ASCII как для пробельных символов (см. "Whitespace" in perlrecharclass), так и для классов Posix (см. "POSIX Character Classes" in perlrecharclass. Таким образом этот модификатор не означает, что вы не можете использовать Unicode, это означает, что для поиска Юникодных символов вы должны явно использовать конструкцию (`\p{}`, `\P{}`), которая говорит о том, что это Юникод.

Как и следовало ожидать, этот модификатор вызывает, например, что `\D` означает то же, что и `[^0-9]`; в самом деле все не ASCII находят `\D`, `\S`, и `\W`. `\b` по-прежнему соответствует границе между `\w` и `\W`, используя свое определение при модификаторе `/a` (аналогично для `\B`).

В противном случае `/a` ведет себя как модификатор `/u`, в регистронезависимом поиске семантики Юникодных символов; к примеру будет `"k"` найдет юникодный `\N{KELVIN SIGN}` при поиске с модификатором `/i` и код символов в диапазоне Latin1, выше ASCII будет иметь правила Юникода, как они действуют при поиске без учета регистра.

Для запрещения поиска ASCII/не-ASCII (например, как "k" с \N{KELVIN SIGN}), поставьте "a" дважды, например /aai или /aia. (Первой появление "a" ограничивает \d, и так далее, а второе появление добавляет ограничение для /i.) Но, нужно заметить, что для символов вне диапазона ASCII будет использоваться правила Юникода для /i поиска, таким образом, модификатор, в действительности, не ограничивает символы только ASCII диапазоном; он просто запрещает смешение символов ASCII и не ASCII диапазона.

Подводя итог, этот модификатор обеспечивает защиту для приложений, которые не хотят быть подвержены влиянию Юникода. Указав его дважды вы получаете дополнительную защиту.

Этот модификатор может быть указан по умолчанию с помощью use re '/a' или use re '/aa'. Если вы так сделаете, вы можете иметь возможность использовать модификатор /u явно, если существует несколько регулярных выражений, где вы хотите использовать полные правила Юникода (но даже здесь, лучше, если бы работала функция "unicode_strings", наряду с use re '/aa'). Также смотри "Какой набор символьных модификаторов действует?".

Какой набор символьных модификаторов действует?

Какой набор символьных модификаторов действует в любой месте регулярного выражения зависит от набора довольно сложных взаимодействий. Они разработаны таким образом, что в целом вам не придется беспокоиться об этом, но этот раздел даст вам кровавые подробности. Как объясняется ниже в "Расширенных шаблонах" это возможно явно указав модификаторы, которые применяются только к части регулярного выражения. Внутреннее всегда имеет приоритет над любым внешним и одно применение для целого выражения имеет приоритет над любым из параметров по умолчанию, описанному в оставшейся части этого раздела.

Прагма use re '/foo' может использоваться для установки модификаторов по умолчанию (включая эти) для скомпилированных регулярных выражений в пределах своей области. Эта прагма имеет приоритет над другими прагмами, перечисленные ниже, что также изменяет значения по умолчанию.

Иначе, use locale устанавливает модификатор по умолчанию в /l; и use feature 'unicode_strings, или use 5.012 (или выше) установка умолчания в /u когда не в той же области, либо как use locale или use bytes. (use locale ':not_characters' также устанавливают умолчание в /u, переопределяет и простой use locale.) В отличие от механизмов, упомянутых выше, эти операнды влияют на операции помимо поиска регулярного выражения по шаблону и поэтому дают более последовательные результаты с другими операторами, в том числе с использованием \U, \l и т.д. в операциях замены.

Если ни один из перечисленных выше вариантов не применяется, то, для обратной совместимости модификатор /d является в сущности по умолчанию. Так как это может привести к неожиданным результатам, то лучше указать

какой другой набор правил должен использоваться.

Модификатор поведения набора символов до Perl 5.14

(/)

До 5.14, были не было явных модификаторов, но /l подразумевается для регулярных выражений, скомпилированные в пределах действия прагм use locale и /d подразумевался иначе.Однако интерполяция регексов в больших регексах будет игнорировать оригинальную компиляцию в пользу всего, что было в действительности во время второй компиляции. Существует ряд несоответствий (ошибок) с модификатором /d , где правила Юникода будут неуместны и наоборот.С <\p{}> не подразумевает правил Юникода и не делает все вхождения \N{} , до 5.12.

Регулярные выражения

Метасимволы

Шаблоны, используемые в Perl эволюционировали от тех, которые были в Процедурах Регулярных Выражениях Версии 8 (Version 8 regex routines). Процедуры являются производными (отдаленно) от свободно распространяемой повторной реализации подпрограмм версии V8 Генри Спенсера . См. "Версия 8 Регулярных Выражений" для подробностей.

В частности следующие метасимволы имеют значения по *egrep*-ному стандарту:

```
\       Экранирование следующего метасимвола
^       Найдет начало строки
.       Найдет любой символ (кроме символа новой строки)
$       Найдет конец строки (перед символом новой строки в конце)
|       Альтернатива
()      Группировка
[]      Класс символов в квадратных скобках
```

По умолчанию, символ "^" гарантирует, что найдется только начало строки , а символ "$" найдет только конец (или перед символом новой строки в конце), и Perl делает определенные оптимизации, предполагая, что набор символов содержит только одну строку. Внутренние новые строки не будут найдены "^" или "$". Вы можете, тем не менее пожелать представить символы для поиска как буфер из многих строк, тогда "^" будет найдет после любого символа новой строки внутри символов для поиска (исключением является случай, когда символ новой строки является последним символом в строке поиска), и "$" найдет перед любым символом новой строки. Стоимость накладных расходов немного больше, если вы применяете к шаблону поиска модификатор /m. (Старые программы делали это, устанавливая $* , но эта возможность удалена в perl 5.10.)

Для упрощения замены в многострочном буфере, символ "." никогда не найдет символ новой строки, только, если вы не используете модификатор `/s`, который фактически сообщает Perl, что весь буфер для поиска - это одна строка --даже если это не так.

Повторители (Квантификаторы, умножители, символы, указывающие количество)

Существуют следующие стандартные повторители:

```
*           Найдет 0 или больше раз
+           Найдет 1 или больше раз
?           Найдет 1 или 0 раз
{n}         Найдет точно n раз
{n,}        Найдет по крайней мере n раз
{n,m}       Найдет по крайней мере n раз, но не более m раз
```

(Если фигурная скобка находится в любом другом контексте и не являются частью эскейп последовательности такой, как `\x{...}`, то она обрабатывается как обычный символ. В частности, нижняя граница повторителя не является обязательной, опечатка в повторителе молча рассматривает его, как буквенные символы. Например,

```
/o{4,3}/
```

выглядит как повторитель, который найдется 0 раз, так как 4 больше 3, но это в действительности это означает соответствие последовательности из шести символов `"o { 4 , 3 }"`. Запланировано в конечном счете потребовать буквального использования из фигурных скобок, которых будут заэкранированы, скажите, поставив перед ними обратную косую черту или поставив их квадратные скобки, , (`"\{"` или `"[{]"`). Это изменение будет допускать будущее расширения синтаксиса (как создание нижней границы повторителя, как опциональное), и лучшей проверки на ошибки. Тем временем вы должны привыкнуть избегать всех случаев, где вы имеете в виду символ "{".)

Повторитель "*" означает тоже, что и `{0,}`, повторитель "+" это `{1,}`, и "?" это `{0,1}`. n и m ограничены неотрицательными значениями меньше, чем заданный предел, определенный во время сборки perl. Это обычно 32766 на наиболее распространенных платформах. Фактический предел может будьте найден в сообщении об ошибке, произведенном следующим кодом:

```
$_ **= $_ , / {$_} / for 2 .. 42;
```

По умолчанию повторительный подшаблон "жадный", это означает, что он будет искать максимальное число совпадений из возможного (начиная с указанной стартовой позиции) пока еще соответствие шаблону остается. Если вы хотите найти минимальное число совпадений , укажите после повторителя "?". Следует что значение шаблона не меняется, а только его "жадность":

```
*?       Найдёт 0 или больше раз, не жаден
+?       Найдёт 1 или больше раз, не жаден
??(/)    Найдёт 0 или 1 раз, не жаден
{n}?     Найдёт точно n раз, не жаден (излишне)
{n,}?    Найдёт как минимум n раз, не жаден
{n,m}?   Найдёт как минимум n раз но не более m раз, не жаден
```

По умолчанию, когда повторенный вложенный шаблон не позволяет совпасть остальной части общего шаблона, Perl делает шаг назад (backtrack). Однако такое поведение является иногда нежелательно. Тогда Perl предоставляет форму "притяжательного" повторителя.

```
*+     Найдёт 0 или больше раз и не дает возврата
++     Найдёт 1 или больше раз и не дает возврата
?+     Найдёт 0 или 1 раз и не дает возврата
{n}+   Найдёт точно n раз и не дает возврата (излишне)
{n,}+  Найдёт как минимум n раз и не дает возврата
{n,m}+ Найдёт как минимум n раз но не более m раз и не дает возврата
```

Например,

```
'aaaa' =~ /a++a/
```

никогда не найдется, так как `a++` будет съедать все `a` в строке и не оставит ни одного символа для оставшейся части шаблона. Эта особенность может быть экстремально полезна, давая перлу сигнал, где не должно быть возврата назад. Например, типичная проблема "поиска строки в двойных кавычках" может решиться наиболее эффективно, когда написано так:

```
/"(?:[^"\\]++|\\.)*+"/
```

как мы знаем, если последняя кавычка не находится, то поиск с возвратом не поможет.

Смотри независимое выражение "`(?>pattern)`" для деталей; притяжательные повторители являются только синтаксическим сахаром для этой конструкции. Например, пример, приведенный выше, может быть записан следующим образом:

```
/"(?>(?:(?>[^"\\]+)|\\.)*)"/
```

Эскейп последовательности (или последовательности обратной косой черты)

Поскольку шаблоны обрабатываются, как строки в двойных кавычках, то следующие последовательности также будут работать:

```
\t          табуляция                   (HT, TAB)
\n          новая строка               (LF, NL)
\r          ввод, перенос строки                (CR)
\f          Перевод формата (Formfeed)             (FF)
\a          Тревога (звуковой сигнал)          (BEL)
\e          Клавиша ESC (Escape) (think troff)  (ESC)
\cK         символ control          (пример: VT)
\x{}, \x00  порядковый номер символа, данный в шестнадцатеричном формате
\N{name}    Именованный символ Юникода или последовательность символов
\N{U+263D}  Юникодный символ     (пример: FIRST QUARTER MOON)
\o{}, \000  порядковый номер символа, данный в восьмеричном формате
\l          Следующий символ в нижнем регистре (думая vi)
\u          Следующий символ в верхнем регистре (думая vi)
\L          Нижний регистр до \E (думая vi)
\U          Верхний регистр до \E (думая vi)
\Q          (quote) Отключает действие метасимволов в шаблоне до \E
\E          Завершение модификации, конец случае модификации или заквотированной секции, думая
```

Подробнее смотри "Квотирование и Операторы заключения в кавычки" in perlop ("Quote and Quote-like Operators" in perlop).

Символьные типы и другие специальные эскейп символы

Кроме того Perl определяет следующее:

Последовательность	Замечание	Описание
[...] (/)	[1]	Найдет символ, который соответствует правилу в квадратных скобках, определенный "...". Например: [a-z] найдет "a" или "b" или "c" ... или "z"
[[:...:]]	[2]	Найдет символ, который соответствует правилам POSIX "..." класса символов в двойных квадратных скобках. Например: [[:upper:]] ha символ в верхнем регистре.
(?[...])	[8]	Расширенный класс символов в скобках
\w	[3]	Найдет символ "слова" (буквацифра плюс "_", плюс другие соединительные знаки пунктуации, отметки Юникода)
\W	[3]	Найдет символ не-"слова"
\s	[3]	Найдет пробельный символ
\S	[3]	Найдет не пробельный символ
\d	[3]	Найдет символ десятичной цифры
\D	[3]	Найдет отрицание символа десятичной цифры
\pP	[3]	Найдет P, именованное свойство. Используйте \p{Prop} для длинных имен
\PP	[3]	Найдет не-P
\X	[4]	Найдет Юникодный "расширенный графема кластер"
\C		Найдет один символ из языка C (октет - 8 бит, 1 байт) даже, если он часть большого UTF-8 символа. Таким образом он ломает символы в их байты из UTF-8, получить искаженные кусочки(malformed pieces) UTF-8. Не поддерживается в поиске вперед.
\1	[5]	Обратная ссылка на конкретную группу или буфер. '1' на самом деле может быть любое положительное целое число.
\g1	[5]	Обратная ссылка на конкретную или предыдущую группу,
\g{-1}	[5]	Числа могут быть отрицательными, указывать относительно предыдущей группы и при необходимости могут быть обернуты в фигурные скобки для безопасного синтаксического разбора.
\g{name}	[5]	Именованная обратная ссылка
\k<name>	[5]	Именованная обратная ссылка
\K	[6]	Сохраняет найденный материал слева от \K, не включая его в $&
\N	[7]	Любой символ, кроме \n. Не влияет на модификатор /s
\v	[3]	Вертикальный пробел
\V	[3]	Невертикальный пробел
\h	[3]	Горизонтальный пробел
\H	[3]	Негоризонтальный пробел
\R	[4]	Универсальная новая строка

[1]
См. "Bracketed Character Classes" in perlrecharclass для деталей.

[2]
См. "POSIX Character Classes" in perlrecharclass для деталей.

[3]
См. "Backslash sequences" in perlrecharclass для деталей.

[4]
См. "Misc" in perlrebackslash для деталей.

[5]

См. "Группы захвата" ниже для деталей.

[6]

(/)

См. "Расширенные шаблоны поиска" ниже для деталей.

[7]

Обратите внимание, что `\N` имеет два значения. Когда он в форме `\N{NAME}`, он соответствует символу или последовательности символов, которые называются `NAME`; и так же, когда форма `\N{U+`*hex*`}`, она соответствует символу Юникода, в шестнадцатеричном коде — <hex>. В противном случае он соответствует любому символу, кроме `\n`.

[8]

See "Расширенная классы скобочных символов (Extended Bracketed Character Classes)" in perlrecharclass для деталей.

Утверждения

Perl определяет следующие утверждения нулевой длины:

```
\b  Найдет границу слова
\B  Найдет все, кроме границы слова
\A  Найдет начало строки
\Z  Найдет только в конце строки или перед новой строкой в конце
\z  Найдет только в конце строки
\G  Найдет только в pos() (т.е. в конце последней найденной позиции m//g)
```

Граница слова (`\b`) это место между двумя символами, которые содержат `\w` на одной их стороне и `\W` на другой (в любом порядке), считая воображаемые символы начала и конца строки, как `\W`. (Внутри сивольных классов `\b` представляет клавишу backspace, а не границу слова,так, как это обычно бывает в любой строке в двойных кавычках.) `\A` и `\Z` работают также, как и "^" и "$", за исключением того, что они не будут искать несколько раз, когда используется модификатор `/m`, тогда как "^" и "$" будут соответствовать каждой внутренней линии границы. Чтобы найти актуальный конец строки и не игнорировать дополнительные конечные строки, используйте `\z`.

Утверждение `\G` можно использовать для глобального поиска (используя `m//g`), как описано в "Квотирование и Операторы заключения в кавычки" in perlop ("Quote and Quote-like Operators" in perlop). Это также удобно во время написания `lex`-подобных сканеров, когда у вас есть несколько паттернов, которым вы хотите сопоставить последующие подстроки строки; смотрите предыдущие ссылки. Фактическое местонахождение, где `\G` будет находить, также может определяться с помощью `pos()` как lvalue(левое значение): см. "pos" in perlfunc. Обратите внимание, что правило нулевой длины (см. "Повторяющиеся шаблоны, находящие подстроки нулевой длины"

("Repeated Patterns Matching a Zero-length Substring")) изменяется несколько в том, что содержимое слева от \G не учитываются при определении длины найденного. Таким образом следующее никогда не будет соответствовать:

```
my $string = 'ABC';
pos($string) = 1;
while ($string =~ /(.\G)/g) {
    print $1;
}
```

Он напечатает 'A' и остановиться, т.к. будет считаться, что был поиск нулевой ширины и таким образом - он не будет соответствовать одной и той же позиции дважды в строке.

Стоит отметить, что \G при не правильном использовании может привести к бесконечному циклу. Будьте осторожны при использовании шаблонов, которые включают \G как альтернативу.

Группы захвата

Скобочная конструкция (...) создаёт группы захвата (также именуется как буферы захвата). Чтобы сослаться на текущее содержимое группы позже в пределах текущего шаблона поиска нужно использовать \g1 (или \g{1}) для первой, \g2 (или \g{2}) для второй группы и так далее. Это называется *обратной ссылкой* (*backreference*).

Количество захваченных подстрок, которые вы можете использовать не ограничено. Группы нумеруются с левой открывающей круглой скобки номер 1, и т.д. Если группа не найдена, связанные обратные ссылки тоже не будут совпадать. (Это может случиться, если группа не является обязательной, или поиск идет в альтернативной ветке). Вы можете опустить "g" и написать "\1", и т.д., но с этим есть некоторые проблемы, описанные ниже.

Также можно захватывать группы относительно друг друга, используя отрицательное число, так C<\g-1> и C<\g{-1}> оба относятся к группе непосредственно перед захватом и \g-2 и \g{-2} относятся к группе перед ней. Например:

```
/
 (Y)           # группа 1
 (             # группа 2
    (X)        # группа 3
    \g{-1}     # обратная ссылка на группу 3
    \g{-3}     # обратная ссылка на группу 1
 )
/x
```

поиск будет таким же, как `/(Y) ( (X) \g3 \g1 )/x`. Это позволяет интерполировать регулярные выражения в большие регулярные выражения и тогда не придется беспокоиться о перенумерации групп захвата.

Можно вообще обойтись числами и создать именованные группы захвата. Нужно объявить `(?<name>...)` и `\g{name}` для такой ссылки. (Чтобы быть совместимым с регулярные выражения .Net, `\g{name}` также может быть написано как `\k{name}`, `\k<name>` или `\k'name'`.) *name* не должно начинаться с цифры или содержать дефисы. Если несколько групп в рамках шаблона имеют одинаковое имя, то любая ссылка на это имя подразумевает самую левую определяемую группу. Именованные группы учитываются и в абсолютной и в относительной нумерации и на них также можно сослаться с помощью номера.

(Это делает возможным делать вещи с именованными группами захвата, которые бы в противном случае требовали бы `(??{})`.)

Содержимое захваченной группы определяется динамически и оно доступно для вас вне шаблона поиска до конца закрывающего блока или до следующего успешного поиска, что наступит раньше. (См. "Составные операторы" in perlsyn.) Вы можете обращаться к ним по абсолютному номеру (используя `"$1"` вместо `"\g1"`, и т.д.); или по имени через хэш `%+`, с помощью `"$+{name}"`.

Скобки требуются для ссылки на именованные группы захвата, но являются опциональными для абсолютной или относительной нумерации групп. Скобки являются более безопасными, когда создается регулярное выражение объединение меньших строк. Например, если у вас есть `qr/$a$b/` и `$a` содержит `"\g1"`, и `$b` содержит `"37"`, вы получите `/\g137/`, что является, вероятно, не тем, что вы хотели.

Обозначения `\g` и `\k` были введены в Perl 5.10.0. До этого не было ни именованных, ни относительно пронумерованных групп захвата. Абсолютная нумерация группы передавалась с помощью `\1`, `\2` и т.д., и эта запись до сих пор принимается (и скорее всего всегда будет). Но это приводит к некоторым неясностям, если есть более чем 9 групп захвата, так `\10` может означать либо десятую группу захвата, или символ, чей порядковый номер восьмеричное-010 (бэкспейс (пробел назад) в ASCII). Perl разрешает эту неопределенность путем интерпретации `\10` как обратной ссылки только, если по крайней мере 10 левых скобок были открыты перед ней. Аналогичным образом, `\11` - это обратная ссылка только, если по крайней мере 11 левых скобок открыты перед ней. И так далее. От `\1` до `\9` всегда интерпретируются как обратные ссылки. Есть несколько примеров ниже, которые иллюстрируют эти риски. Вы можете избежать двусмысленности всегда используя `\g{}` или `\g`, если вы имеете в виду захватываемые группы; и для восьмеричных констант всегда использовать `\o{}`, или для `\077` и ниже, используя 3 цифры с ведущими нулями, поскольку лидирующий ноль означает восьмеричную константу.

Обозначения *\digit* также работает в определенных обстоятельствах за пределами шаблона. Смотри "Предупреждение на \1 вместо $1" ниже для деталей.

Examples:

```
s/^([^ ]*) *([^ ]*)/$2 $1/;     # поменять местами первые два слова

/(.)\g1/                        # найдет первый дублирующий символ
     and print "'$1' is the first doubled character\n";

/(?<char>.)\k<char>/            # ... другой способ
     and print "'$+{char}' is the first doubled character\n";

/(?'char'.)\g1/                 # ... смешанный поиск
     and print "'$1' is the first doubled character\n";

if (/Time: (..):(..):(..)/) {   # разобрать значения
    $hours = $1;
    $minutes = $2;
    $seconds = $3;
}

/(.)(.)(.)(.)(.)(.)(.)(.)(.)\g10/   # \g10 - это обратная ссылка
/(.)(.)(.)(.)(.)(.)(.)(.)(.)\10/    # \10 - восьмеричное число
/((.)(.)(.)(.)(.)(.)(.)(.)(.))\10/  # \10 - это обратная ссылка
/((.)(.)(.)(.)(.)(.)(.)(.)(.))\010/ # \010 - восьмеричное число

$a = '(.)\1';        # Создает проблемы, когда объединяются.
$b = '(.)\g{1}';     # Позволяет избежать проблем.
"aa" =~ /${a}/;      # Правда
"aa" =~ /${b}/;      # Правда
"aa0" =~ /${a}0/;    # Ложь!
"aa0" =~ /${b}0/;    # Правда
"aa\x08" =~ /${a}0/;  # Правда!
"aa\x08" =~ /${b}0/;  # Ложь
```

Несколько специальных переменных также вернут порцию предыдущего поиска. `$+` вернет результат найденного в последней скобке. `$&` вернет всю совпадающую строку. (Раньше это делал `$0`, но теперь он возвращает имя программы.) `$`` вернет все перед совпавшей строкой. `$'` вернет все после совпавшей строки. И `$^N` содержит все, что было найдено в самой недавно закрытой группе (подстроку). `$^N` может быть использован в расширенные шаблонах (см. ниже), например назначить найденную подстроку переменной.

Существуют специальные переменные такие как хеш `%+` и пронумерованные переменные поиска (`$1`, `$2`, `$3`, и т.д.) динамически распространяющиеся до конца закрывающего блока или до следующего успешного поиска, что наступит раньше. (См. "Составные операторы" in perlsyn.) (См. "Compound Statements" in perlsyn.)

ЗАМЕЧАНИЕ: неудачные поиски в Perl не сбрасывают переменные поиска, что упрощает написание кода, который проверяет серию конкретных случаев и запоминает лучший поиск.

ПРЕДУПРЕЖДЕНИЕ: как только Perl видит, что вам нужна одна из переменных `$&`, `$`` , или `$'` где-либо в программе, он должен предоставить их для каждого поиска по шаблону. Это может существенно замедлить вашу программу. Perl использует тот же механизм для производства `$1`, `$2`, и т.д., так, что вы также платите существенную цену для каждого шаблона, который содержит захватывающие скобки. (Чтобы избежать этой платы при сохранении группирующего поведения, используйте вместо этого расширенное регулярное выражение `(?: ... )` .) Но если вы никогда не используете `$&`, `$`` или `$'`, тогда шаблоны *без* захватывающих скобок не будут наказаны. Поэтому избегайте `$&`, `$'`, или `$``, если можете, но если вы не можете (и я очень ценю некоторые такие алгоритмы), после того как вы использовали их один раз, использовать их по своему желанию, потому что вы уже за низ заплатили. По состоянию на 5.17.4, присутствие каждой из трех переменных в программе регистрируется отдельно и, в зависимости от обстоятельства, perl может быть в состоянии быть более эффективным зная что только `$&` , а не все три будут использованы, например.

Для того, чтобы обойти эту проблему Perl 5.10.0 представляет переменные `${^PREMATCH}`, `${^MATCH}` и `${^POSTMATCH}`, которые эквивалентны `$``, `$&` и `$'`, **исключая** то, что только они будут гарантированно определены после успешного поиска, который был выполнен с модификатором `/p` (preserve) (сохранять) . Использование этих переменных не влечет к штрафу глобальной производительности, в отличие от их эквивалента в виде знаков препинания, однако здесь вы идете на компромисс того, что вы должны сказать perl, когда вы хотите их использовать.

Квотирующие метасимволы (Quoting metacharacters)

Метасимволы с обратным слэшем в Perl являются буквенно-цифровыми, например `\b`, `\w`, `\n`. В отличие от некоторых других языков с регулярными выражениями, не существует символов с обратным слэшем, которые не являются буквенно-цифровыми. Так что все, что выглядит как `\\`, `\(`, `\)`, `\ [`, `\]`, `\ {`, или `\}` всегда интерпретируется как буквенный символ, а не метасимвол. Это было некогда общей идеей для отключения или квотирования (оборачивание в кавычки) метасимволов, имеющих специальное значение в регулярных выражениях, которых вы хотите использовать для шаблоне поиска. Теперь заквотируем (поставим обратный слэш) для всех не-"словесных" символов:

```
$pattern =~ s/(\W)/\\$1/g;
```

(Если установлено `use locale`, то это зависит от текущей локали.) Сегодня чаще используют функцию quotemeta() или C<\Q> метаквотированный эскейп символ для отключения специальных значений всех метасимволов следующим образом:

```
/$unquoted\Q$quoted\E$unquoted/
```

Учтите, что если вы положите символ косой черты (который не внутри переменных с интерполяцией) между `\Q` и `\E`, двойное квотирование интерполяции обратной косой черты может привести к нежелательным результатам. Если вам *нужно* использовать символ косой черты в пределах `\Q...\E`, проконсультируйтесь с ("Кровавые подробности разбора закавыченных(квотированных) конструкций" in perlop) ("Gory details of parsing quoted constructs" in perlop).

`quotemeta()` и `\Q` полностью описаны в "quotemeta" in perlfunc.

Расширенные шаблоны

Perl также определяет синтаксис последовательного расширения функций не входящих в стандартные инструменты, такие как **awk** и **lex**. Синтаксис для большинства из них - это пара скобок с вопросительным знаком идущим сразу за первой скобкой. Символ идущий за вопросительным знаком указывает расширение.

Стабильность этих расширений колеблется в широких пределах. Некоторые из них были частью основного языка на протяжении многих лет. Другие экспериментальные и могут меняться без предупреждения или быть полностью удалены. Проверяйте документацию для каждого компонента для проверки его текущего статуса.

Вопросительный знак был выбран для этого и для минимального соответствия конструкции потому что 1) вопросительные знаки встречаются редко в старых регулярных выражения и 2) всякий раз, когда вы его видите, вы должны остановиться и "задаться вопросом" точности того, что происходит. Это психология...

(?#text)
: Комментарий. Текст игнорируется. Если модификатор `/x` позволяет использование пробельных символов для форматирования, то простого `#` будет достаточно. Обратите внимание, что Perl закрывает комментарий, как только он видит `)`, так что нет никакого способа положить литерал (символ) `)` в комментарий.

(?adlupimsx-imsx)
(?^alupimsx)

Один или несколько встроенных модификаторов в шаблон, чтобы быть включенным (или выключенным, если предшествует `-`) на оставшуюся часть (?) шаблона или оставшуюся часть закрытой группы поиска(если таковые имеются).

Это особенно полезно для динамических шаблонов, например, которые читаются из файла конфигурации, взяты из аргумента, или указаны в таблице где-либо. Рассмотрим случай, когда некоторые шаблоны хотят быть с учетом регистра,а другие нет: для того, чтобы поиска части шаблона был без учета регистра, то необходимо включить `(?i)` перед шаблоном. Например:

```
$pattern = "foobar";
if ( /$pattern/i ) { }

# более гибко:

$pattern = "(?i)foobar";
if ( /$pattern/ ) { }
```

These modifiers are restored at the end of the enclosing group. For example, Эти модификаторы восстанавливаются в конце внешней группы. Например,

```
( (?i) blah ) \s+ \g1
```

который найдет `blah` в любом регистре, некоторые пробелы и точное (*включая регистр*!) повторение предыдущего слова, предполагая модификатор `/x`, а теперь нет модификатора `/i` за пределами этой группы.

Эти модификаторы не переносятся в именованные подмаски, вызванные в закрытой группе. Другими словами шаблон например `((?i)(?&NAME))` не изменяет чувствительности регистра для шаблона "NAME".

Любой из этих модификаторов можно установить, что применить его глобально для всех регулярных выражений, скомпилированные в сфере `use re`. См. "'/flags' mode" in re.

Начиная с Perl 5.14, `"^"` (курсор или диакритический акцент) сразу за `"?"` это сокращение соответствует `d-imsx`. Флаги (кроме `"d"`) могут следовать после курсора для переопределения. Но минус не является легальным с ним.

Учтите, что модификаторы `a`, `d`, `l`, `p`, и `u` являются специальными здесь они могут быть включены, не не выключены, а модификаторы `a`, `d`, `l`, и `u` являются взаимоисключающими: указав один вы отрицаете другой, и максимум

один из них (или два a) может появится в этой конструкции. Таким образом, (?-p) предупредит при компиляции при use warnings; (?-d:...) и (?dl:...) дадут фатальную ошибку.

Также, обратите внимание, что модификатор p имеет особое значение в том, что его присутствие в любом месте в шаблоне имеет глобальный эффект.

(?:pattern)

(?adluimsx-imsx:pattern)

(?^aluimsx:pattern)

Это для кластеризации, но не для захвата; он группирует подвыражения как "()", но не делает обратной ссылки, как это делает "()". Так что

```
@fields = split(/\b(?:a|b|c)\b/)
```

подобно

```
@fields = split(/\b(a|b|c)\b/)
```

но не выплевывает дополнительные поля. Также дешевле не захватывать группы, если вам не нужно.

Любые символы между ? и : действуют как флаги модификаторов, такие как (?adluimsx-imsx). Например,

```
/(?s-i:more.*than).*million/i
```

эквивалентно более подробному

```
/(?:(?s-i)more.*than).*million/i
```

Начиная с Perl 5.14, "^" (курсор или диакритический акцент) сразу за "?" это сокращение соответствует d-imsx. Любые позитивные флаги (кроме "d") могут следовать после курсора, таким образом

```
(?^x:foo)
```

эквивалентно

```
(?x-ims:foo)
```

Курсор рассказывает Perl, что этот кластер не наследует флаги любых окружающих шаблонов, но использует системные умолчания (d-imsx), измененные любыми указанными флагами.

Курсор позволяет упростить создание строчек (stringification) из скомпилированных регулярных выражений. Они выглядят как

```
(?^:pattern)
(/)
```

с любыми флагами не по умолчанию, появляющимися между курсором и двоеточием. Тест, который выглядит, как перевод в строку (stringification), таким образом, не нуждается в в флагах по умолчанию, жестко зашитых в нем, только курсор. Если новые флаги будут добавлены в Perl, смысл курсора расширения изменится включением флагов по умолчанию, поэтому тест все еще будет работать, без изменений.

Указание отрицательных флага, после курсора `^`, является ошибкой, т.к. как флаг является излишним.

Мнемоника для `(?^...)`: свежее начало, поскольку галка обычно используется для поиска начала.

`(?|pattern)`

Это "сброс ветки" шаблона, который имеет специальные свойства, когда захватываемые группы нумеруются с той же начальной точки в каждой чередующейся ветке . Он доступен начиная с perl 5.10.0.

Группы захвата нумеруются слева направо, но внутри этой конструкции нумерация возобновляется для каждой ветви.

Нумерация в пределах каждой ветки будет нормальной и все группы следующие за этой конструкцией будут пронумерованы так,как-будто бы конструкция содержится только одну ветвь, ту, которая попадет в группу захвата.

Эта конструкция является полезной, когда вы хотите захватить одну из нескольких альтернативных групп.

Рассмотрим следующий шаблон. Цифры над под выражением показывают номера групп захвата.

```
# before  ---------------branch-reset----------- after
/ ( a )  (?| x ( y ) z | (p (q) r) | (t) u (v) ) ( z ) /x
# 1           2         2  3        2     3     4
```

Будьте внимательны при использовании сброса ветки шаблона в сочетании с именем захвата. Именованные захваты реализованы как псевдонимы нумерованным группам, что мешает осуществлению сброса ветки шаблона. Если вы используете именованный захват в сбросе ветки шаблона, то лучше использовать те же имена, в том же порядке, в каждом из альтернатив:

```
/(?|  (?<a> x ) (?<b> y )
   |  (?<a> z ) (?<b> w )) /x
```

Не делайте так, это может привести к сюрпризам:

```
"12" =~ /(?| (?<a> \d+ ) | (?<b> \D+))/x;
say $+ {a};   # Prints '12'
say $+ {b};   # *Also* prints '12'.
```

Проблема здесь в том, что обе группы назвали `a`, а группа с именем `b` является алиасом для группы, принадлежащей к `$1`.

Утверждения Осмотра Вокруг (Look-Around Assertions)

Утверждения Осмотра Вокруг представляют собой шаблоны нулевой ширины, которые находя специфические элементы без включения результата в `$&`. Позитивные утверждения находятся, когда подмаски находятся, отрицательные утверждения находятся, когда поиск по подмаске не удачный. Осмотр-До(look-behind) находит текст до текущей позиции, Осмотр-После(look-ahead) находит текст следующий за позицией поиска.

(?=pattern)

Позитивное нулевой ширины утверждение поиска вперед. Например, `/\w+(?=\t)/` найдет слово, за которым следует табуляция без включения таба в `$&`.

(?!pattern)

Негативное нулевой ширины утверждение поиска вперед. Например `/foo(?!bar)/` найдет любое вхождение "foo" за которым не следует "bar". Замечание. Однако поиск вперед и назад не то же самое. Вы не можете использовать это для поиска назад.

Если вы ищете "bar" перед которым нет "foo", `/(?!foo)bar/`, то будет искаться не то, что вы хотите. Потому что `(?!foo)` означает, что следующая вещь не может быть "foo"--но это не так, это "bar", таким образом "foobar" будет найден. Используйте вместо этого поиск назад (смотрите ниже).

(?<=pattern) \K

Позитивное нулевой ширины утверждение поиска назад. Например, `/(?<=\t)\w+/` найдет слово, перед которым стоит таб, без включения таба в `$&`. Работает только для поиска назад фиксированной длины.

Существует специальная форма этой конструкции, называемая `\K`, которая влияет на регекс движок таким образом, что он "сохраняет" все, что было найдено перед `\K` и не включает это в `$&`. Это обеспечивает эффективный поиск назад переменной длины. Использования `\K` внутри другого утверждения Осмотра Вокруг возможно, но это поведение еще хорошо не определено.

Для различных причин `\K` может быть значительно более эффективным, чем эквивалентная конструкция `(?<=...)` и он особенно полезен в ситуации, где вы хотите эффективно удалить что-то следующее после строки. К примеру

```
s/(foo)bar/$1/g;
```

(/)можно переписать гораздо более эффективно

```
s/foo\Kbar//g;
```

(?<!pattern)

Отрицающее утверждение нулевой ширины поиска назад. Например /(?<!bar)foo/ найдет вхождения "foo" , которые не следуют за "bar". Работает только для поиска назад фиксированной длины.

(?'NAME'pattern)

(?<NAME>pattern)

Имя захваченной группы. Идентично во всех отношениях для нормального захвата скобками () , но есть для дополнительной факт то, что к группе можно обращаться по имени в различных конструкциях регулярных выражений (например, \g{NAME}) и может быть доступен по имени после успешного поиска через %+ или %- . См. perlvar для более подробной информации о хэшах %+ и %- .

Если несколько различных захватываемых групп имеют одинаковые имена, то $+{NAME} будет ссылаться на самую левую и этих групп.

Формы (?'NAME'pattern) и (?<NAME>pattern) эквивалентны.

ПРИМЕЧАНИЕ: Хотя нотация этой конструкции такая же, как в регексах .NET , но поведение не совпадает. В Perl являются группы нумеруются последовательно, независимо от того именованные они или нет. Таким образом, в шаблоне

```
/(x)(?<foo>y)(z)/
```

$+{foo} будет тоже, что и $2, и $3 будет содержать 'z' вместо того, что хакер .NET регекса мог бы ожидать.

В настоящее время имя ограничивается простым идентификатором. Другими словами, он должен совпадать с /^[_A-Za-z][_A-Za-z0-9]*\z/ или его Юникодным расширением (см. utf8) хотя он не поддерживает докали (locale) (см. perllocale).

ПРИМЕЧАНИЕ: для того, чтобы сделать вещи проще для программистов с опытом работы с Python или движком регексов PCRE, шаблон (?P<NAME>pattern) может быть использован вместо (?<NAME>pattern) ; Однако эта форма не поддерживает использование одинарных кавычек в качестве разделителя для имени.

\k<NAME>

\k'NAME'

Именованная обратная ссылка. Похожа на цифровую обратную ссылку за исключением случаев , когда группа обозначается именем, а не числом. Если несколько групп имеют то же имя, то оно относится к самой левой найденной группе в текущем поиске.

Будет сообщение об ошибке, если будет ссылка на имя не определенное С << (? <NAME>) >> ранее в шаблоне.

Обе формы являются эквивалентными.

ПРИМЕЧАНИЕ: для того, чтобы сделать вещи проще для программистов с опытом работы с Python или движком регексов PCRE, шаблон (?P=NAME) может быть использован вместо \k<NAME> .

(?{ code })

ПРЕДУПРЕЖДЕНИЕ: Эта функция расширенных регулярных выражений считается экспериментальной и может быть изменена без предварительного уведомления. Выполняемый код имеет побочные эффекты, которые могут быть не одинаковыми от версии к версии из-за эффекта будущего оптимизаций в движке регексов (regex engine). Для осуществление этой функции был радикально пересмотрен 5.18.0 релиз и его поведение в более ранних версиях perl было намного более бажное (с большим числом багов), особенно в отношении разбора, лексических переменных, их действия, рекурсии и повторного входа (to parsing, lexical vars, scoping, recursion and reentrancy).

Это утверждение нулевой ширины выполняет любой встроенный код Perl. Оно всегда завершается успешно, и ее возвращаемое значение устанавливается как $^R .

В буквенных шаблонах, код парсится в то же время, как окружающий код. Во время обработки шаблона управление временно передаётся анализатору perl пока не находится закрывающая скобка. Это похоже путь, которым индекс массива вытаскивает строку, например

```
"abc$array[ 1 + f('[') + g()]def"
```

В частности, скобки не обязательно должны быть сбалансированы:

```
s/abc(?{ f('{'); })/def/
```

Даже, если шаблон является интерполяцией и компилируется во время выполнения, литеральные блоки кода компилируются один раз, во время компиляции perl; следующее печатает "ABCD":

```
print "D";
my $qr = qr/(?{ BEGIN { print "A" } })/;
(/)my $foo = "foo";
/$foo$qr(?{ BEGIN { print "B" } })/;
BEGIN { print "C" }
```

В шаблонах, где текст кода является производным от информации времени выполнения , а не появляется буквально в исходном коде /шаблоне/, код компилируется в то же время, что и шаблон, и, по соображениям безопасности, `use re 'eval'` должен быть в области. Это нужно для остановки пользовательских шаблонов, содержащие фрагменты кода, которые будут выполнены.

В ситуациях, когда вам нужно включить это с `use re 'eval'` вы должны также включить проверку на чужеродные примеси в коде(taint) Еще лучше, используйте аккуратно ограничения выполнения в рамках безопасного отсека (Safe compartment.). Смотрите perlsec для деталей работы обоих этих механизмов.

С точки зрения анализа, лексическая область видимости переменной и замыканий (closures),

```
/AAA(?{ BBB })CCC/
```

ведет себя примерно как

```
/AAA/ && do { BBB } && /CCC/
```

Аналогичным образом,

```
qr/AAA(?{ BBB })CCC/
```

ведет себя примерно как

```
sub { /AAA/ && do { BBB } && /CCC/ }
```

В частности:

```
{ my $i = 1; $r = qr/(?{ print $i })/ }
my $i = 2;
/$r/; # prints "1"
```

Внутри блока `(?{...})`, `$_` относится к строке регулярного выражения стоящего напротив. Вы также можете использовать `pos()`, чтобы узнать текущую позицию соответствия внутри этой строки.

> Блок кода вводит новую область с точки зрения лексического объявления переменных, но **не** с точки зрения локального и аналогичного локализованного поведения. Чуть позже блоки кода в том же шаблоне будут (^) по-прежнему видеть значения, которые были локализованы в предыдущих блоках. Эти накопленные локализации отменяются либо в конце успешного поиска, или, если утверждение возвратилось (assertion is backtracked) (сравните "Backtracking"). Например,

```
$_ = 'a' x 8;
m<
   (?{ $cnt = 0 })             # Инициализация $cnt.
   (
     a
     (?{
         local $cnt = $cnt + 1;  # Обновляем $cnt,
                                 # обратная ссылка безопасна (backtracking-safe).
     })
   )*
   aaaa
   (?{ $res = $cnt })          # При удачном поиске копирование в
                               # не-локализованную локацию.
 >x;
```

первоначально увеличит $cnt до 8; затем во время отступает, его значение будет развернуто обратно до 4, это значение назначается C <$res>. В конце выполнения регекса $cnt будет уменьшено обратно в свое первоначальное значение 0.

Это утверждение может использоваться как условие в

```
(?(condition)yes-pattern|no-pattern)
```

свиче (переключателе). Если *не* использовать таким образом, то результат выполнения кода помещается в специальную переменную $^R. Это происходит сразу же, так только $^R может быть использован с другими (?{ code }) утверждениями внутри того же самого регулярного выражения.

Присваивание C<$^R> выше правильно локализовано, так, что старое значение $^R восстанавливается, если утверждение возвратилось при поиске; сравните "Поиск с возвратом".

Обратите внимание, что специальная переменная $^N особенно полезна с блоками кода для записи результатов подсовпадений в переменные без необходимости отслеживать количество вложенных скобок. Например:

```
$_ = "The brown fox jumps over the lazy dog";
/the (\S+)(?{ $color = $^N }) (\S+)(?{ $animal = $^N })/i;
print "color = $color, animal = $animal\n";
```

(??{ code })

ПРЕДУПРЕЖДЕНИЕ: Эта функция расширенных регулярных выражений считается экспериментальной и может быть изменена без предварительного уведомления. Выполняемый код имеет побочные эффекты и он может не выполняться одинаково от версии к версии из-за эффекта будущих оптимизаций в регекс движке.

Это «отложенное» регулярное выражение. Оно ведет себя *точно* так же, как и (?{код}) , как описано выше, за исключением того, что возвращаемое значение не присваиваются $^R , а рассматривается как шаблон, компилируется, если он является строкой (или используется как есть, если это qr / / объект), затем ищется соответствие, как если бы оно был вставлено вместо этой конструкции.

В ходе сопоставления этого вложенного шаблона, он имеет свой собственный набор захваченных скобок (или захватов дальше), которые являются действительными во время подпоиска (промежуточного поиска), но удаляются, как только управление возвращается в основной шаблон. Например, следующий поиск, с внутренним шаблоном захвата "B" и находящий "BB", пока внешний шаблон захватывает "A";

```
my $inner = '(.)\1';
"ABBA" =~ /^(.)(??{ $inner })\1/;
print $1; # prints "A";
```

Обратите внимание, что это означает, что нет никакого способа, чтобы внутренний шаблон ссылался на группу захвата, определенную вне. (Сам блок кода можно использовать $1 , и т.д., для ссылки на группу захвата включенного шаблона.) Таким образом, хотя

```
('a' x 100)=~/(??{'(.)' x 100})/
```

найдет он *не* установит $1 при выходе.

Следующий шаблон найдет группу в круглых скобках:

```
$re = qr{
           \(
           (?:
              (?> [^()]+ )  # Не-парные без поиска с возвратом
            |
              (??{ $re })   # Группы с соответствующими парными скобками
           )*
           \)
       }x;
```

Смотрите также (?*PARNO*) для другого, более эффективного способа выполнения этой задачи.

Выполнение отложенного регулярного выражения 50 раз без получения какой-либо входной строки приведет к фатальной ошибке. Максимальная глубина (6) компилируется в perl поэтому, чтобы изменить её требуется кастомный билд(custom build).

(?*PARNO*) (?-*PARNO*) (?+*PARNO*) (?R) (?0)

Аналогично работает (??{ code }) за исключением того, что он не предполагает выполнение любого кода или потенциальной компиляции строки возвращенного шаблона; вместо этого он представляет часть текущего шаблона, содержащийся в указанной группе захвата как независимый шаблон, который должен совпадать с текущей позиции. Группы захвата, содержащиеся в шаблоне, будут иметь значение, как это определено во внешней рекурсии.

PARNO является последовательность цифр (не начинающихся с нуля), значение которого отражает число пар группы захвата для рекурсии. (?R) рекурсивно проходит по по шаблону с самого начала. (?0) -это альтернативный синтаксис для (?R). Если *PARNO* предшествует плюс или минус, то можно предположить, чтобы быть относительным, с отрицательными числами указывает предыдущие группы захвата , а знак плюс группы после. Таким образом (?-1) относится к наиболее недавно объявленной группе и (?+1) указывает следующую объявленную группу. Обратите внимание, что подсчет относительной рекурсии отличается от относительных обратных ссылок, в том, что в рекурсию **включаются** не закрытые группы.

Следующий шаблон ищет функцию foo(), которая может содержать сбалансированные скобки в качестве аргумента.

```
$re = qr{ (                   # парная группа 1 (полная функция)
            foo
            (                 # парная группа 2 (пары)
              \(
                (             # парная группа 3 (содержимое парных скобок)
                (?:
                 (?> [^()]+ ) # Не парные без поиска с возвратом
                |
                 (?2)         # Рекурсия на начало 2-й парной группы
                )*
                )
              \)
            )
          )
        }x;
```

Если шаблон был использован следующим образом

```
'foo(bar(baz)+baz(bop))'=~/$re/
    and print "\$1 = $1\n",
              "\$2 = $2\n",
              "\$3 = $3\n";
```

результаты должны быть следующие:

```
$1 = foo(bar(baz)+baz(bop))
$2 = (bar(baz)+baz(bop))
$3 = bar(baz)+baz(bop)
```

Если соответствующая группа захвата не определена , то будет фатальная ошибка. Рекурсии глубже, чем 50 без добавления на вход новой строки приведет также к фатальной ошибке. Максимальная глубина компилируется в perl, поэтому для его изменения его требуется сделать кастомный билд (requires a custom build).

Ниже показано, как с помощью отрицательной индексации облегчить внедрение рекурсивных структур внутри конструкции qr// для дальнейшего использования:

```
my $parens = qr/(\((?:[^()]++|(?-1))*+\))/;
if (/foo $parens \s+ \+ \s+ bar $parens/x) {
   # сделать что-то здесь...
}
```

Примечание, что этот шаблон не ведёт себя так же, как эквивалент в PCRE или Python такой же формы. В Perl можно делать поиск с возвратом в рекурсируемой группе, в PCRE и Python рекурсия в группе рассматривается как атомарная. Кроме того модификаторы разрешаются во время компиляции, так что такие конструкции как (?i:(?1)) или (?:(?i)(?1)) не влияют на то, как вложенный шаблон будет обрабатываться.

(?&NAME)

Рекурсия для именованных шаблонов. Идентичен (?*PARNO*) за исключением того, что на скобки для рекурсии указывает имя. Если несколько скобок имеют то же имя, то она рекурсивно берет самую левую.

Возникнет сообщение об ошибке, если будет ссылка на имя, которое не объявлено где-то в шаблоне.

ПРИМЕЧАНИЕ: для того, чтобы облегчить жизнь программистов с опытом работы с Python или движком регексов PCRE, шаблон (?P>NAME) может быть использован вместо (?&NAME).

(?(condition)yes-pattern|no-pattern)

(?(condition)yes-pattern)

Условное выражение. Найдет yes-pattern если condition дает значение true, в противном случае ищет no-pattern . Отсутствует шаблон по которому ищется всегда.

С <(condition)> должно быть одним из: 1) целое число в скобках (которое валидно, если соответствующая пара скобок найдена); 2) поиск впереди/поиск назад/выполнение утверждения нулевой ширины; 3) имя в угловых

скобках или одинарных кавычках (который валидно, если найдена группа с заданным именем); или 4) Специальный символ (R) (true когда выполнение внутри рекурсии или eval). Дополнительно после R могут быть числа, (1) (которые будут true, когда выполняться рекурсия внутри соответствующей группы) или &NAME, в этом случае она будет иметь значение true только при выполнении во время рекурсии именованной группы.

Вот краткое изложение возможных предикатов:

(1) (2) ...

Проверяет, если что-то пронумерованные группы захвата что-то нашли.

(<NAME>) ('NAME')

Проверяет, если что группа с заданным именем что-то нашла.

(?=...) (?!...) (?<=...) (?<!...)

Проверяет, найден ли шаблон (или не найден, для вариантов с '!').

(?{ CODE })

Обрабатывает возвращаемое значение блока кода как условие.

(R)

Проверяет, что выражение выполняется внутри рекурсии.

(R1) (R2) ...

Проверяет, если выражение выполняется во время n-ной группы захвата. Эта проверка является эквивалентом следующему регексу

```
if ((caller(0))[3] eq 'subname') { ... }
```

Другими словами он не проверяет полный стек рекурсии.

(R&NAME)

Аналогично (R1), этот предикат проверяет при выполнении непосредственно внутри левой группы с заданным именем (это та же самая логика используемая (?&NAME) для устранения неоднозначности). Он не проверяет полный стек, но только имя внутренней активной рекурсии.

(DEFINE)

В этом случае, да-шаблон выполняется не напрямую и нет никаких нет-шабонов. Аналогично в духе (?{0}) но более эффективно. Подробнее смотрите ниже.

For example:

```
m{ ( \( )?
   [^()]+
   (?(1) \) )
 }x
```

найдет часть с непарными скобками, возможно, включенные в парные скобки.

Особая форма - это предикат (DEFINE) , который никогда не выполняет (?) своих yes-шаблонов непосредственно и не разрешает no-шаблоны. Это позволяет определить подмаски, которые будут выполняться только механизмом рекурсии. Таким образом, можно определить набор регулярных выражений, которые могут быть использованы в любом шаблоне, который вы выберете.

Рекомендуется, что для данного использования вы определяете блок DEFINE в конце шаблона, и что вы задаете имена любым подшаблонам, определенным внутри него.

Кроме того, стоит отметить, что шаблоны определенные таким образом, вероятно, не могут быть эффективными, так как оптимизатор их обрабатывающий не достаточно умный.

Вот пример, как это может быть использовано:

```
/(?<NAME>(?&NAME_PAT))(?<ADDR>(?&ADDRESS_PAT))
 (?(DEFINE)
   (?<NAME_PAT>....)
   (?<ADRESS_PAT>....)
 )/x
```

Обратите внимание, что группы захвата внутри соответствующей рекурсии не доступны после возврата из нее, так как для этого нужен дополнительный слой из группы захвата. Таким образом $+{NAME_PAT} не будет определяться, хотя $+{NAME} будет.

Наконец имейте в виду, что подшаблоны созданные внутри блока DEFINE считают абсолютное и относительное число захватов, таким образом:

```
my @captures = "a" =~ /(.)                # Первый захват
                      (?(DEFINE)
                         (?<EXAMPLE> 1 )  # Второй захват
                      )/x;
say scalar @captures;
```

Будет выводить 2, не 1. Это особенно важно, если вы намерены компилировать регулярное выражение с помощью qr// и позднее интерполировать их в другой шаблон.

(?>pattern)

"Независимое" подвыражение, которой соответствует подстрока являющаяся *автономным* (*standalone*) шаблоном , который найдется, если есть якорь на данной позиции, и он найдет *ничего другого, кроме этой строки*. Это конструкция полезна для оптимизации в противном случае были бы "вечные"

поиски, потому что он не будет делать поиск с возвратом (см. "Backtracking"). Он также может быть полезным в местах, где "захватить все можно и не дать что-нибудь обратно" семантической желательно.

Например: `^(?>a*)ab` никогда не будет совпадать, тогда как `(?>a*)` (якорь в начале строки, как указано выше) будет соответствовать *всем* символам `a` в начале строки, оставляя не C <a>для поиска `ab`. В противоположность этому `a*ab` будет соответствовать тому же, что и `a+b`, поскольку поиск подгруппы `a*` находится под влиянием следующей группы `ab` (см. "Backtracking"). В частности, `a*` внутри `a*ab` будет соответствовать меньше символов, чем автономное `a*`, так как оно делает хвостовой поиск.

`(?>pattern)` не отключает поиск в возвратом всего один раз, пока не найдет. Еще возможен поиск с возвратом, проходящий мимо конструкции, но не входящий в неё. Так что `((?>a*)|(?>b*))ar` будет по-прежнему соответствовать "bar".

Эффект, аналогичный `(?>pattern)` может быть достигнут путем написания `(?=(pattern))\g{-1}`. Это соответствует той же подстроке, как автономная `a+` и следующий C <\g{-1}> ест совпавшие строки; он поэтому делает утверждение нулевой длины в аналогичное `(?>...)`. (Разница между этих двумя конструкциями является тем, что во второй используется группа захвата (записи), таким образом смещаются порядковые числа обратных ссылок в остальной части регулярного выражения.)

Рассмотрим этот шаблон:

```
m{ \(
     (
       [^()]+          # x+
     |
       \( [^()]* \)
     )+
   \)
 }x
```

Он будет эффективно соответствовать непустой группе с соответствующими скобками до двух уровней или меньше. Однако, если нет такой группы, он практически навсегда возьмет длинную строку. Это потому, что так много различных способов разбить длинную строку на несколько подстрок. Это то, что делает `(.+)+` и `(.+)+` подобен подшаблону выше шаблона. Рассмотрим, как шаблон выше обнаруживает не нахождение `(()aaaaaaaaaaaaaaaaa` в несколько секунд, но каждая дополнительная буква удваивает это время. Это экспоненциальное повышение затрат сделает так, что ваша программа зависнет. Однако крошечные изменения в этот шаблоне

```
m{ \(
     (
(/)    (?> [^()]+ )        # изменим x+ выше на (?> x+ )
     |
       \( [^()]* \)
     )+
   \)
 }x
```

которая использует (?>...) находит точно,когда один выше делает (проверка этого самостоятельно является хорошим упражнением), но заканчивается в четвертом времени при использовании подобной строки с 1000000 a s. Будьте осторожны, однако, что, когда за этой конструкцией следует умножитель (квантификатор), то он в настоящее время вызывает предупреждение под прагмой use warnings или ключа **-W**, который говорит, что "много раз найдены null строки в регексе" ("matches null string many times in regex").

На простых группах, таких, как шаблон (?> [^()]+) , сопоставимый эффект может быть достигнут путем негативного поиска вперед, как в [^()]+ (?! [^()]) . Это было только в 4 раза медленнее на строке с 1000000 a s.

Сема нтика (от др.-греч. σημαντικός — обозначающий) "Возьми все, что можешь и ничего не отдавать обратно" желательна во многих ситуациях, когда на первый взгляд простой ()* выглядит как правильное решение. Предположим, что мы анализировали текст с комментариями с разделителями от # за которыми следуют некоторые необязательные (горизонтальные) пробелы. В отличие от его внешнего вида, #[\t]* *Это не* правильное подвыражение для поиска разделителей комментариев, потому что она может "отказаться" от некоторых пробелов если остальная часть шаблона совпадет. Правильный ответ на это - ни один из них:

```
(?>#[ \t]*)
#[ \t]*(?![ \t])
```

Например чтобы захватить не-пустые комментарии в $1, следует использовать что-либо одно из этого:

```
/ (?> \# [ \t]* ) (       .+ ) /x;
/    \# [ \t]*   ( [^ \t] .* ) /x;
```

От того, что вы выбираете, зависит, какие из этих выражений лучше отражают выше спецификации комментариев.

В некоторой литературе эта конструкция называется "атомарным поиском" или "притяжательным поиском"("possessive matching").

Притяжательные умножители эквивалентны поставленному элементу, к которому они применяются внутри одной из этих конструкций. Применяются следующие эквиваленты:

```
Форма с умножителями  Форма со скобками
---------------       ---------------
PAT*+                 (?>PAT*)
PAT++                 (?>PAT+)
PAT?+                 (?>PAT?)
PAT{min,max}+         (?>PAT{min,max})
```

(?[ ])

See "Расширенные классы символов в квадратных скобках" in perlrecharclass.

Специальные глаголы для управления поиска с возвратом (Special Backtracking Control Verbs)

ПРЕДУПРЕЖДЕНИЕ: эти шаблоны являются экспериментальными и могут быть изменены или удалены в будущей версии Perl. Их использование в продуктивном коде следует специально отметить, чтобы избежать проблем при обновлении.

Эти специальные шаблоны, как правило, в форме (*VERB:ARG). Иногда аргумент ARG является необязательным; в некоторых случаях это запрещено.

Любой шаблон, содержащий специальный глагол возврата, который позволяет аргументу иметь специальное поведение, когда выполнен он устанавливает в текущем пакете $REGERROR и $REGMARK переменные. При этом применяются следующие правила:

В случае неудачи, переменной $REGERROR будет присвоено значение ARG глагольного шаблона, если глагол принимал участие в этом поиске. Если значение ARG шаблона было опущено, то $REGERROR будет присвоено имя последнего исполненного шаблона (*MARK:NAME), или в значение TRUE, если такого не было. Кроме того, переменная $REGMARK будет установлена в FALSE.

В случае успешного поиска, переменная $REGERROR будет установлена в FALSE, а переменной $REGMARK будет присвоено имя последнего исполненного шаблона (*MARK:NAME). Смотрите объяснение для глагола (*MARK:NAME) ниже.

ПРИМЕЧАНИЕ: $REGERROR и C<$REGMARK> не являются магическими переменными такими, как $1 и большинство других переменных, связанных с регексами. Они не являются локальными для области, ни ReadOnly, но вместо этого они аналогичны летучим переменным пакета $AUTOLOAD. Используйте local для локализации изменений в их конкретной области при необходимости.

Если шаблон не содержит специального глагола для поиска с возвратом, который позволяет аргумент, то $REGERROR и $REGMARK не затрагиваются вообще.

Глаголы, которые принимают аргумент

(*PRUNE) (*PRUNE:NAME)

Этот шаблон нулевой ширины удаляет дерево с возвратом в текущей точке когда возврат был неудачным. Рассмотрим шаблон A (*PRUNE) B , где A и B являются сложными шаблонами. Пока не достигается глагола <(*PRUNE)> , A может возвращаться назад, если это нужно для сопоставления. После того, как оно будет достигнуто, поиск продолжается к B, который также может возвращаться в случае необходимости; Однако, если B не найдено, то без дальнейших возвратов мы займет место и поиск по шаблону будет неудачным прямо в текущей начальной позиции.

В следующем примере подсчитывается все возможные соответствия строк в шаблоне (без фактического сопоставления любой из них).

```
'aaab' =~ /a+b?(?{print "$&\n"; $count++})(*FAIL)/;
print "Count=$count\n";
```

что производит:

```
aaab
aaa
aa
a
aab
aa
a
ab
a
Count=9
```

Если мы добавим (*PRUNE) перед count следующим образом

```
'aaab' =~ /a+b?(*PRUNE)(?{print "$&\n"; $count++})(*FAIL)/;
print "Count=$count\n";
```

мы предотвратили поиск с возвратом и нашли число самой длинной найденной строки в каждой соответствующей отправной точке следующим образом:

```
aaab
aab
ab
Count=3
```

Любое число утверждений (*PRUNE) может использоваться в шаблоне.

Смотрите также (?>pattern) и притяжательные квантификаторы для других способов управления возвратом. В некоторых случаях использование (*PRUNE) может быть заменено на (?>pattern) без функциональной разницы; однако, (*PRUNE) может использоваться для обработки случаев, которые не могут быть выражены с помощью одного (?>pattern)

(*SKIP) (*SKIP:NAME)

Этот шаблон нулевой ширины подобен (*PRUNE), за исключением случая неудачного поиска, это также означает, что любой текст, который стоит перед шаблоном (*SKIP) не может быть частью, *любого* матча текущего шаблона. Это фактически означает, что обработчик регулярных выражений "пропускает" вперед на эту позицию в случае неудачного поиска и пытается снова продолжить поиск(предполагая, что имеется достаточно места для соответствия).

Имя шаблона (*SKIP:NAME) имеет особое значение. Если (*MARK:NAME) обнаружен при сопоставлении, то это эту позицию используют в качестве "точки пропуска». Если не было обнаружено <(*MARK)>, то оператор (*SKIP) не имеет никакого эффекта. При использовании без имени "точки пропуска" там, где была точка поиска, когда выполнялся (*SKIP) шаблон.

Сравните следующие примеры в (*PRUNE); обратите внимание строки, что строка в два раза длиннее:

```
'aaabaaab' =~ /a+b?(*SKIP)(?{print "$&\n"; $count++})(*FAIL)/;
print "Count=$count\n";
```

выведет

```
aaab
aaab
Count=2
```

После того, как "aaab" в начале строки найден и выполнен (*SKIP), следующий отправной точкой будет место, где выполнен (*SKIP).

(*MARK:NAME) (*:NAME)

Это нулевой ширины шаблон может быть использован, чтобы отметить точку, достигнутое в строке когда определенная часть шаблона успешно найдена. Этому знак может быть дано имя. Через C<(* SKIP)> шаблон будет пропускать вперед к этой точке, если он отступил в случае неудачи. Любое количество шаблонов (*MARK) разрешено, а ИМЯ может быть продублировано.

Помимо взаимодействия с шаблоном (*SKIP), (*MARK:NAME) может использоваться как "метка" разветвления шаблона, так что после сопоставления, программа может определить, какие ветви шаблона были вовлечены в поиск (матч).

Если совпадение успешно, переменной $REGMARK будет присвоено имя последнего выполненного (*MARK:NAME), который участвовал в поиске. Это может использоваться для определения, какая ветвь шаблона сработала без использования отдельной группы для каждой ветви, которые в свою очередь может привести к улучшению производительности, так как perl не может оптимизировать /(?:(x)|(y)|(z))/ также эффективно, как что-то типа /(?:x(*MARK:x)|y(*MARK:y)|z(*MARK:z))/.

(/)

Когда поиск не удался, и пока другой глагол был вовлечен в ошибочный поиск и предоставил свое собственное имя для использования, переменной $REGERROR будет присвоено имя последнего выполненного (*MARK:NAME).

См. "(*SKIP)" для более подробной информации.

В качестве быстрого имени (шортката) (*MARK:NAME) может быть написан, как (*:NAME).

(*THEN) (*THEN:NAME)

Это похоже на оператор :: "вырезать группу" в Perl 6. Также, как и (*PRUNE), эта команда всегда совпадает и когда поиск с возвратом неудачен, он вызывает обработчик регексов, чтобы попробовать следующее чередование во внутренней, включающей группе (захвата или иным образом), которая имеет альтернативы. Две ветви (?(condition)yes-pattern|no-pattern) не считают альтернативу так, как это делает (*THEN)

Его название происходит от наблюдения, что эта операция в сочетании с оператором чередования (|) может использоваться для создания по сути шаблоно-блоков если/то/иначе:

```
( COND (*THEN) FOO | COND2 (*THEN) BAR | COND3 (*THEN) BAZ )
```

Обратите внимание, что если используется этот оператор не внутри чередование, то он действует так же, как оператор (*PRUNE).

```
/ A (*PRUNE) B /
```

Это то же самое, что и

```
/ A (*THEN) B /
```

но

```
/ ( A (*THEN) B | C ) /
```

не тоже самое, что и

```
/ ( A (*PRUNE) B | C ) /
```

так как после нахождения A, но при неудачном поиске B глагол (*THEN) сделает возврат и попробует C; но глагол (*PRUNE) просто закончится (/)неудачей.

Глаголы без аргументов

(*COMMIT)

Это "шаблон фиксации" из Perl 6 <commit> или ::: . Это нулевой ширины шаблон подобен (*SKIP) , за исключением случаев, когда поиск с возвратом будет неудачным он вызывает неудачу не сразу. Никаких дальнейших попыток чтобы найти правильны матч, продвигая указатель в начало будет происходить снова. Например,

```
'aaabaaab' =~ /a+b?(*COMMIT)(?{print "$&\n"; $count++})(*FAIL)/;
print "Count=$count\n";
```

выведет

```
aaab
Count=1
```

Другими словами после того как был введен (*COMMIT) и, если шаблон не совпадает, то движок регексов не будет попробовать любые дополнительные поиск на остальной части строки.

(*FAIL) (*F)

Этот шаблон соответствует ничему и всегда терпит неудачу. Он может использоваться для принудительного движка с поиском с возвратом. Это эквивалентно (?!) , но легче читать. Фактически, (?!) получает оптимизированное (*FAIL) внутренне.

Это вероятно полезно только при сочетании с (?{}) или (??{}) .

(*ACCEPT)

В <ПРЕДУПРЕЖДЕНИЕ:> эта функция является весьма экспериментальной. Не рекомендуется для кода в продакшене.

Этот шаблон ничего не ищет и возникает в конце успешного поиска в точке, в которой шаблон (*ACCEPT) был обнаружена, независимо от того, есть ли на самом деле больше искать в строке. Когда внутри вложенный шаблон, например рекурсии, или в подшаблоне динамически генерируемые данные через (??{}) , только самый внутренний шаблон завершается немедленно.

Если (*ACCEPT) внутри группы захвата тогда группа помечает конец в точке, в которой был обнаружен (*ACCEPT) . К примеру:

```
'AB' =~ /(A (A|B(*ACCEPT)|C) D)(E)/x;
```

будет найден и $1 будет AB и $2 будет B, а $3 не будет установлен. Если другая ветка во внутренних скобках будет найдена, например, как (/)строка "ACDE", тогда D и E будут найдены также.

Поиск с возвратом (Backtracking)

Примечание: В этом разделе абстрактно представлено приблизительное поведение регулярных выражений. Для просмотра более строгого (и сложного) видения правил участия в выборе найденного среди возможных альтернатив, смотрите "Сочетая частей RE".

Фундаментальная возможностью поиска регулярными выражениями включает в себя понятие под названием *Поиск с возвратом* (*backtracking*), который используется в настоящее время (при необходимости) всеми регулярными выражениями с не притяжательными повторителями, а именно *, *?, +, +?, {n,m}, и {n,m}?. Часто поиск с возвратом оптимизирован внутренне, но общий принцип, изложенный здесь является допустимым.

Для поиска регулярным выражением *всё* регулярное выражение должно совпасть, а не только часть его. Так что, если начало шаблона, содержащего повторитель успешно совпало способом, который вызывает позже части шаблона для неудачного поиска, поисковый движок возвращается и пересчитывает начальную часть - вот почему он называется поиском с возвратом.

Вот пример поиска с возвратом: скажем, вы хотите найти слово после "foo" в строке "Food is on the foo table.":

```
$_ = "Food is on the foo table.";
if ( /\b(foo)\s+(\w+)/i ) {
    print "$2 следует за $1.\n";
}
```

Когда поиск начинается, первая часть регулярного выражения (\b(foo)) находит возможное возможное справа, в самом начале строки и загружает $1 в "Foo". Однако, как только поисковый движок видит, что нет пробелов после "Foo", который он положил в $1, он понимает свою ошибку и начинается новый поиск после знака, где он делал предварительный поиск. На этот раз он идет вплоть до следующего вхождения "foo". Полное регулярное выражение сопоставляется в этот раз, и вы получаете ожидаемый результат "table следует за foo."

Иногда минимальный поиск может сильно помочь. Представьте, что вы хотели бы найти все, что между "foo" и"bar". Изначально вы напишете что-то типа этого:

```
$_ =  "The food is under the bar in the barn.";
if ( /foo(.*)bar/ ) {
  (/)print "got <$1>\n";
}
```

Который, возможно, неожиданно дает:

```
got <d is under the bar in the >
```

Это потому, что .* жадный, так что вы получите все, что между *первым* "foo" и *последним* "bar". Здесь более эффективным является использование минимального соответствия, чтобы убедиться, вы получите текст между "foo" и первым "bar" после этого.

```
  if ( /foo(.*?)bar/ ) { print "got <$1>\n" }
got <d is under the >
```

Вот еще один пример. Допустим, вы хотите найти число в конце строки и вы также хотите сохранить предыдущую часть. То вы напишете следующее:

```
$_ = "I have 2 numbers: 53147";
if ( /(.*)(\d*)/ ) {                                # Неправильно!
    print "Beginning is <$1>, number is <$2>.\n";
}
```

Это не будет работать совсем, потому что .* жадный и съест вверх всю строку. Так как \d* может соответствовать пустой строке, то полное регулярное выражение будет успешно найдено.

```
Beginning is <I have 2 numbers: 53147>, number is <>.
```

Вот некоторые варианты, большинство из которых не работают:

```
$_ = "I have 2 numbers: 53147";
@pats = qw{
  (/)(.*)(\d*)
    (.*)(\d+)
    (.*?)(\d*)
    (.*?)(\d+)
    (.*)(\d+)$
    (.*?)(\d+)$
    (.*)\b(\d+)$
    (.*\D)(\d+)$
};

for $pat (@pats) {
    printf "%-12s ", $pat;
    if ( /$pat/ ) {
        print "<$1> <$2>\n";
    } else {
        print "FAIL\n";
    }
}
```

Это напечатает:

```
(.*)(\d*)    <I have 2 numbers: 53147> <>
(.*)(\d+)    <I have 2 numbers: 5314> <7>
(.*?)(\d*)   <> <>
(.*?)(\d+)   <I have > <2>
(.*)(\d+)$   <I have 2 numbers: 5314> <7>
(.*?)(\d+)$  <I have 2 numbers: > <53147>
(.*)\b(\d+)$ <I have 2 numbers: > <53147>
(.*\D)(\d+)$ <I have 2 numbers: > <53147>
```

Как вы видите, это может быть немного сложнее. Важно понимать, что регулярное выражение-это просто набор утверждений, которые определяют успех. Может быть 0, 1 или несколько различных способов определений, который могут быть успешны против конкретной строки. И если есть несколько способов, которыми можно добиться успеха, то вам нужно понять поиск с возвратом, чтобы узнать, насколько разнообразного успеха вы добьетесь.

При использовании утверждений осмотра вперед утвердительного или отрицательного, это может получиться даже сложнее. Представьте, что вы хотели бы найти последовательность не-цифр, за которыми не следует "123". Вы можете попробовать написать так

```
$_ = "ABC123";
if ( /^\D*(?!123)/ ) {              # Неправильно!
    print "Yup, no 123 in $_\n";
}
```

Но это не будет найдено; по крайней мере не так, как вы надеетесь. Здесь утверждается, что в строке нет никаких 123. Вот более четкая картина о том, почему этот шаблон совпадает, вопреки всеобщим ожиданиям:

```
$x = 'ABC123';
$y = 'ABC445';

print "1: got $1\n" if $x =~ /^(ABC)(?!123)/;
print "2: got $1\n" if $y =~ /^(ABC)(?!123)/;

print "3: got $1\n" if $x =~ /^(\D*)(?!123)/;
print "4: got $1\n" if $y =~ /^(\D*)(?!123)/;
```

Это напечатает

```
2: got ABC
3: got AB
4: got ABC
```

Вы могли рассчитывать, что 3 тест будет провален, потому что он кажется более общим назначением теста 1. Важное различие между ними является то, что тест 3 содержит квантификатор (`\D*`) и поэтому он может использовать поиск с возвратом, тогда как тест 1 не будет. Что происходит, когда вы спросили "это правда, что в начале $x, после 0 или больше не цифр, у вас есть что-то не 123?" Если совпадений шаблона пусть `\D*` расширяется до "ABC", это привело бы к тому, что весь шаблон был бы провальным.

Поисковый движок изначально будет искать шаблон `\D*` в "ABC". Затем он попробует найти `(?!123)` в "123", что не удастся. А потому, что повторитель (`\D*`) был использован в регулярном выражении, поисковый движок может искать с возвратом и повторить поиск по-разному в надежде на полное соответствие регулярного выражения.

Шаблон действительно, *действительно* хочет добиться успеха, поэтому он использует стандартный шаблон "возвратись-и-повтори" и позволяет `\D*` расширится до просто "AB" в это время. Теперь это действительно что-то после "AB" это не "123". Это "C123", которого достаточно.

Мы можем иметь дело с этим с помощью утверждения и отрицания. Мы будем говорить, что за первой частью в $1 должна идти цифра что-то не "123". Помните, что осмотр-вперед является выражением нулевой ширины -- они только так выглядят, но не потребляют никакой строки во время поиска. Так что переписывания таким образом производит то, что можно было бы ожидать; то есть 5 случай неуспешен, а 6 случай успешен:

```
print "5: got $1\n" if $x =~ /^(\D*)(?=\d)(?!123)/;
print "6: got $1\n" if $y =~ /^(\D*)(?=\d)(?!123)/;
```

```
(/)
6: got ABC
```

Другими словами,работают два утверждения нулевой ширины рядом друг с другом они объединяются вместе, так же, как и любые встроенные утверждения: /^$/ соответствует только если вы в начале строки и в конце строки одновременно. Более глубокая правда в том, что соседство в регулярные выражения всегда означает И, за исключением случаев, когда вы пишете явно ИЛИ , используя вертикальную черту. /ab/ означает поиск "a" И (потом) поиск "b", хотя попытки поиска сделаны на разных позициях, потому что "a" это не утверждение нулевой ширины, а утверждение в ширину один символ.

ПРЕДУПРЕЖДЕНИЕ: особенно сложные регулярные выражения могут занимать экспоненциальное время для их решения из-за огромного количества возможных способов, которыми они могут использовать поиск с возвратом для поиска. Например, без внутренней оптимизации, сделанной обработчиком регулярных выражений, это будет занимать много времени для работы:

```
'aaaaaaaaaaaa' =~ /((a{0,5}){0,5})*[c]/
```

И если вы использовали * на внутренние группы вместо их ограничения с 0 до 5 раз, то он будет считать вечно --или до тех пор, пока вы не достигнете переполнения стека. Кроме того эти внутренние оптимизации являются не всегда применимыми. Например, если вы поставили {0,5} вместо * на внешней группе, не применяется текущая оптимизация и поиск занимает много времени для завершения.

Мощный инструмент для оптимизации таких зверских выражений является то, что известно как "Независимая группа", которая не использует поиск с возвратом (см. "(?>pattern)"). Обратите внимание, что утверждения нулевой длины такие как осмотр-вперед/осмотр-назад не будет делать поиск с возвратом, чтобы найти хвостовой поиск, так как они находятся в "логическом" контексте: только то место, где они найдутся считается соответствующим. В качестве примера где побочные эффекты осмотра-вперед *могут* иметь влияние на последующий поиск, см "(?>pattern)".

8 версия регулярных выражений

В случае, если вы не знакомы с "обычной" 8 версией регекс процедур, вот правила сопоставления шаблонов, не описанные выше.

Любой один символ соответствует самому себе, если это не *метасимвол*, которые имеют особое значение, описанное здесь или выше. Вы можете вызывать символы, которые обычно являются метасимволами буквально, предваряя их "\"

(например, поиск "\." найдет ".",а не какой-либо другой символ; "\\" найдет "\"). Этот механизм экранирования нужен также для символа, используемого в качестве разделителя шаблона. (/)

Серия символов соответствует этой серии символов в целевой строке, поэтому шаблон C <blurfl> будет соответствовать "blurfl" в целевой строке.

Можно указать класс символов, заключив список символов в `[]`, который будет соответствовать любому символу из списка. Если первый символ после "[" "^",класс соответствует любому знаку, не в списке. В списке символ "-" указывает на диапазон, так что `a-z` представляет все символы между "a" и "z" включительно. Если вы хотите, чтобы либо "-" либо "]" сами собой были членами класса, поместите их в начале списка (возможно, после "^"), или поставьте перед символами обратный слеш. "-" также воспринимается буквально, когда оно находится в конце списка, непосредственно рядом с "]". (следующие выражения ниже указывают на класс из трех символов: `[-az]`, `[az-]` и `[a\-z]`. Все они отличаются от `[a-z]`, который указывает класс, содержащий двадцать шесть символов, даже на основе EBCDIC набора символов.) Кроме того, если вы попытаетесь использовать символ класса `\w`, `\W`, `\s`, `\S`, `\d`, или `\D` как конечные точки диапазона, то "-" понимается буквально.

Также, обратите внимание, что идея со всем диапазоном довольно непереносима между наборами символов -- и даже в пределах наборов символов она может привести к результатам, которые вы, вероятно, не ожидали. Здесь принцип заключается в том, чтобы использовать только диапазоны, начинающиеся и заканчивается либо в одном алфавите одинакового регистра ([a-e], [A-E]), или цифр ([0-9]). Все остальное является небезопасным. Если есть сомнения, приведите набор символов в полном объеме.

Символы могут быть указаны с помощью синтаксиса метасимволов такого, какой используется в C: "\n" найдет новую строку, "\t" - табулятор, "\r" - возврат каретки, "\f" подача страницы на принтере (form feed), и т.д. В целом *ппп*, где *ппп* является строкой из трех восьмеричных цифр соответствует символу, чьи закодированное символьное значение это *ппп*. Аналогично для \x*пп*, где *пп* являются шестнадцатеричными цифрами, соответствует символу, чей порядковый номер это *пп*. Выражение \c*x* соответствует элементу управления control-*x*. Наконец метасимвол "." соответствует любому знаку, кроме "\n" (если вы используете `/s`).

Можно указать ряд альтернатив для шаблона с помощью "|" для их разделения, так что `fee|fie|foe` найдет любое из "fee", "fie", или "foe" в целевой строке (как бы `f(e|i|o)e`). Первый вариант включает в себя все, от последнего разделителя шаблона ("(", "(?:", и т.д. или в начале шаблона) до первого "|", и последний вариант содержит все, что от последнего "|" к

следующему закрывающему шаблон разделителю. Вот почему это обычная практика для включения альтернативы в скобках: для минимизации путаницы о том, где у них начало и конец.

Альтернативы применяются слева направо, так что, если первая альтернатива найдена, то есть все выражение совпадает, это то, что выбирается. Это означает, что альтернативы не обязательно жадные. Для примера: при сопоставлении `foo|foot` против "barefoot", только часть "foo" будет найдена, так как пытался первый вариант и он был успешно найден в строке целевого объекта. (Это может показаться не важным, но это важно, когда вы захватываете совпадающий текст, используя скобки.)

Также помните, что "|" интерпретируется как литерал в квадратных скобках, так что если вы пишете `[fee|fie|foe]` вы действительно ищете совпадение `[feio|]`.

Внутри шаблона вы можете определять подшаблоны для последующего использования заключив их в скобки и вы можете обратиться к *n*ному подшаблону позже в шаблоне, используя метасимвол `\`*n* или `\g`*n*. Подмаски нумеруются начиная слева направо по очереди открывающих скобок. Обратная ссылка совпадает независимо от того найдется ли на самом деле соответствие подшаблону в рассматриваемой строке, не от правил для этого подшаблона. Таким образом будет `(0|0x)\d*\s\g1\d*` найдет "0x1234 0x4321", но не "0x1234 01234",потому что подшаблон 1 соответствует "0x", даже несмотря на то, что правило `0|0x` потенциально может соответствовать ведущему 0 во второй цифре.

Предупреждение на \1 вместо $1

Некоторые люди слишком привыкли писать такие вещи, как:

```
$pattern =~ s/(\W)/\\\1/g;
```

Это старая привычка (для ссылок от \1 до \9) для RHS замены для избежания шокирующей для **sed**фанатов, но это также грязная привычка попасть. Это потому, что в PerlThink, правой стороне `s///` представляет собой строку с двойными кавычками. `\1` в обычные двойных кавычках строка означает управления A. Для обычного Unix значение `\1` является хаком для `s///`. Однако если вы получите привычку сделать это, вы получить себе в беду, если вы затем добавите модификатор `/e`.

```
s/(\d+)/ \1 + 1 /eg;            # приведет к предупреждению под флагом -w
```

Или, если вы попытаетесь сделать

```
s/(\d+)/\1000/;
```

Вы не сможете устранить неоднозначность, сказав `\{1}000`, в то время как вы можете исправить это только так `${1}000`. Не следует путать операции интерполяции с операцией поиска обратной ссылки. Конечно они значат две разные вещи на *левой* стороне `s///`.

Повторяющиеся шаблоны поиска подстроки нулевой длины

ПРЕДУПРЕЖДЕНИЕ: трудный материал (и проза) будут впереди. Этот раздел необходимо переписать.

Регулярные выражения предоставляют лаконичный и мощный язык программирования. Как и многие другие инструменты власти власть приходит вместе со способностью сеять хаос.

Общие злоупотребления этой властью проистекает из способность сделать бесконечные циклы с помощью регулярных выражений, с чем-то настолько безобидным, как:

```
'foo' =~ m{ ( o? )* }x;
```

`o?` находит в начале `'foo'` и с тех пор позиция в строке не перемещается, поиск, `o?` будет искать снова и снова из-за квантификатора `*`. Еще один распространенный способ создания аналогичного цикла является циклирующий модификатор `//g`:

```
@matches = ( 'foo' =~ m{ o? }xg );
```

или

```
print "match: <$&>\n" while 'foo' =~ m{ o? }xg;
```

или под циклом подразумевается split().

Однако многолетний опыт показал, что многие задачи программирования можно значительно упростить с помощью повторяющихся подвыражений , которым могут соответствовать подстроки нулевой длины. Вот простой пример того:

```
@chars = split //, $string;           # // нет магии split()
($whitewashed = $string) =~ s/()/ /g; # закрытые скобки избегают магии s// /
```

Таким образом Perl позволяет такие конструкции, чтобы *насильственно разорвать бесконечный цикл*. Правила для этого различны для низко-уровневых циклов предоставлены "жадные" повторители кванторы `*+{}`, а для более высокого уровня циклов такие модификаторы, как `/g` или оператор split().

Циклы нижнего уровня являются *прерываемыми*(то есть цикл является сломанный) когда Perl обнаруживает, что повторяемое находит подстроку нулевой длины. Таким образом

```
m{ (?: NON_ZERO_LENGTH | ZERO_LENGTH )* }x;
```

производит эквивалентно

```
m{ (?: NON_ZERO_LENGTH )* (?: ZERO_LENGTH )? }x;
```

Например, эта программа

```
#!perl -l
"aaaaab" =~ /
  (?:
     a                # не равно нулю
     |                # или
    (?{print "hello"}) # печатать hello всякий раз, когда эта
                       #    ветка выбирается
    (?=(b))           # утверждение нулевой длины
  )*  # любое число раз
 /x;
print $&;
print $1;
```

напечатает

```
hello
aaaaa
b
```

Обратите внимание, что "hello" будет напечатан один раз, когда Perl увидит, что шестая итерация внешнего `(?:)*` найдет строку нулевой длины,и он останавливает `*`.

Внешний цикл сохраняет дополнительное состояния между итерациями: в зависимости о того является ли последний поиск нулевой длины. Чтобы разорвать цикл, следующий поиск после совпадения нулевой длины запрещается иметь нулевой длины. Этот запрет взаимодействует с поиском с возвратом (см. "Поиск с возвратом"), и поэтому *второй лучший* поиск выбирается, если *лучший* поиск нулевой длины.

Например:

```
$_ = 'bar';
s/\w??/<$&>/g;
```

результаты в `<><b><><a><><r><>`. В каждой позиции строки лучший поиск дает не-жадное `??`, это совпадения нулевой длины и *вторым лучший* поиском является то, что находится `\w`. Таким образом, поиск нулевой длины является альтернативой поиска односимвольной строки.

Аналогично, для повторного m/()/g вторым лучшим поиском является поиск в позиции на одну ступень далее в строке.

(/)

Дополнительное состояние *поиска нулевой длины* связано с найденной строкой и сброса каждого назначения для pos(). Поиски нулевой длины в конце предыдущего поиска игнорируются во время split.

Сочетание кусков РЕГЕКСПОВ

Каждой из элементарных частей регулярного выражения, которые были описаны ранее (например, ab или \Z) могут соответствовать более чем одной подстроки в данной позиции входной строки. Однако в типичных регулярных выражениях эти элементарные части объединяются в более сложные шаблоны, с помощью сочетания операторов ST, S|T, S* и т.д. (в этих примерах S и T - регулярные выражения).

Такие комбинации могут включать альтернативы, ведущих к проблеме выбора: если мы ищем совпадение для регулярного выражения a|ab в строке "abc", чему оно будет соответствовать подстроке "a" или "ab"? Один из способов описать какие подстроки на самом деле находятся является концепция Поиска с возвратом (см. "Поиск с возвратом"). Однако это описание слишком низкого уровня и заставляет вас думать с точки зрения конкретной реализации.

Другое Описание начинается с понятий "лучше"/"хуже". Все подстроки, которые могут быть сопоставлены с заданным регулярным выражением могут быть отсортированы от "лучшего" совпадения до "худшего", и это "лучшее" сопоставление и будет выбрано. Это заменяет вопрос "что выбрали?" на вопрос "какие сопоставления лучше, а какие хуже?».

Опять же для элементарных частей не существует такого вопроса, поскольку в большинстве случаев возможно только одно совпадение в заданной позиции. В этом разделе описывается понятие лучше/хуже для объединения операторов. В описании ниже S и T - это регулярные выражения.

ST

Рассмотрим два возможных поиска, AB и A'B', A и A' - это подстроки, которые могут быть найдены S, B и B' подстроки которые могут быть найдены T.

Если A является лучшим поиском для S,чем A', то AB будет лучшим поиском, чем A'B'.

Если A и A' совпадают: AB - лучший поиск, чем AB', если B лучший поиск для T чем B'.

S|T

Когда S может найтись, это лучшее совпадение, чем когда может найтись только C<T>.

Последовательность двух поисков для S является такой же и для S. Аналогично для двух поисков для T.

S{ЧИСЛО_ПОВТОРОВ}

Найдет, как SSS...S (повторяя столько раз сколько нужно).

S{min,max}

Найдет S{max}|S{max-1}|...|S{min+1}|S{min}.

S{min,max}?

Найдет S{min}|S{min+1}|...|S{max-1}|S{max}.

S?, S*, S+

Тоже, что и S{0,1}, S{0,BIG_NUMBER}, S{1,BIG_NUMBER} соответственно.

S??, S*?, S+?

Тоже, что и S{0,1}?, S{0,BIG_NUMBER}?, S{1,BIG_NUMBER}? соответственно.

(?>S)

Соответствует лучшему поиску для S и только так.

(?=S), (?<=S)

Только лучший поиск для S рассматривается. (Это имеет значение только, если S имеет скобки, и где-то используются обратные ссылки где-то еще в целом регулярном выражении.)

(?!S), (?<!S)

Для этого оператора группирования нам не нужно описать порядок следования, так как только поиск по не S имеет важное значение.

(??{ EXPR }), (?*PARNO*)

Порядок такой же, как для регулярного выражения, которым является результат EXPR, или шаблон, содержащий группу захвата *PARNO* (*НОМЕРСКОБКИ*).

(?(condition)yes-pattern|no-pattern)

Вызов да-шаблона или нет-шаблона уже определен своим совпадением. Последовательность (ordering) поиска такая же, как и выбранные подвыражения.

Рецепты выше описывают последовательность поиска *в заданной позиции*. Еще одно правило необходимо понять, как поиск определяется для всего регулярного выражения: поиск на ранней позиции всегда лучше, чем успешный поиск в поздней позиции.

Создание пользовательских движков RE (РЕГЕКСПОВ)

По состоянию на Perl 5.10.0 можно создать пользовательский движок регулярных выражений. Это не для слабонервных, так как они должны подключаться на уровне C. См.perlreapi для получения более подробной

информации.

В качестве альтернативы можно перегружать константы (см. overload) (7) предоставляют простой путь для расширения функциональности движка PE, заменяя один шаблон другим.

Предположим, что мы хотим, чтобы включить новую RE эскейп-последовательность \Y| , которая ищет на границе между знаками пробела и непробельных символов. Обратите внимание что (?=\S)(?<!\S)|(?!\S)(?<=\S) находит в точности на этих позициях, итак мы хотим иметь каждый \Y| в месте более сложной версии. Мы можем создать модуль customre для того, чтобы делать это:

```
package customre;
use overload;

sub import {
  shift;
  die "No argument to customre::import allowed" if @_;
  overload::constant 'qr' => \&convert;
}

sub invalid { die "/$_[0]/: invalid escape '\\$_[1]'"}

# Мы должны также позаботиться о том, чтобы неэскейпить законную \\Y|
# последовательность, поэтому присутствие '\\' в правилах преобразования.
my %rules = ( '\\' => '\\\\',
              'Y|' => qr/(?=\S)(?<!\S)|(?!\S)(?<=\S)/ );
sub convert {
  my $re = shift;
  $re =~ s{
            \\ ( \\ | Y . )
          }
          { $rules{$1} or invalid($re,$1) }sgex;
  return $re;
}
```

Теперь use customre включает новую эсейп-последовательность в постоянных регулярных выражениях, то есть без какого-либо выполнения переменных интерполяции. Как описано в overload, это преобразование будет работать только над литеральными (символьными) частями регулярных выражений. Для \Y|$re\Y| переменная часть этого регулярного выражения необходимо явно преобразовать (но только если особое значение \Y| должно быть включено внутри $re):

```
use customre;
$re = <>;
chomp $re;
$re = customre::convert $re;
/\Y|$re\Y|/;
```

Поддержка PCRE/Python

По состоянию на Perl 5.10.0 Perl поддерживает несколько расширений Python/PCRE-специфичного синтаксиса регекс. В то время, как Perl программистов, рекомендуется использовать Perl специфический синтаксис, также принимаются следующие:

(?P<NAME>pattern)
  Определение именованную группу. Эквивалент (?<NAME>pattern).

(?P=NAME)
  Обратная ссылка на именованную группу. Эквивалент \g{NAME}.

(?P>NAME)
  Вызов подпрограммы как именованной группы. Эквивалент (?&NAME).

ОШИБКИ (BUGS)

Многие конструкции регулярных выражений не работают на платформах EBCDIC.

Существует целый ряд проблем (issues) в отношении поиска без учета регистра в правилах Юникода. Смотрите i в статье "Модификаторы" выше.

Этот документ варьируется от сложного для понимания до полностью и совершенно непрозрачного. Блуждающие проза, изобилует жаргоном трудно понимаемым в нескольких местах.

Этот документ нужно переписать, выделяя содержимое учебника из ссылки на содержание.

СМОТРИТЕ ТАКЖЕ

perlrequick.

perlretut.

"Квотирование и Операторы заключения в кавычки" in perlop.

"Внутренние детали парсинга конструкций в кавычках" in perlop.

perlfaq6.

"pos" in perlfunc.

perllocale.

perlebcdic.

Mastering Regular Expressions by Jeffrey Friedl, published by O'Reilly and Associates.

ПЕРЕВОДЧИКИ

- Николай Мишин <mi@ya.ru>