perlop (Источник, CPAN)

Содержание

ИМЯ

perlop - операторы и приоритет Perl

ОПИСАНИЕ

В Perl оператор определяет, какая операция выполняется, независимо от типа операндов. Например, `$x + $y` всегда является числовым дополнением, и если `$x` или `$y` не содержат чисел, сначала предпринимается попытка преобразовать их в числа.

Это отличается от многих других динамических языков, где операция определяется типом первого аргумента. Это также означает, что Perl имеет две версии некоторых операторов, одну для числового и одну для сравнения строк. Например, `$x == $y` сравнивает два числа на предмет равенства и `$x eq $y` сравнивает две строки.

Однако есть несколько исключений: `x` может быть либо повторением строки, либо повторением списка, в зависимости от типа левого операнда, и `&`, `|`, `^` и `~` могут быть либо строковыми, либо числовыми битовыми операциями.

Приоритет операторов и ассоциативность

Приоритет операторов и ассоциативность работают в Perl более или менее так же, как в математике.

Приоритет операторов означает, что некоторые операторы группируются более плотно, чем другие. Например, в `2 + 4 * 5` умножение имеет более высокий приоритет, поэтому `4 * 5` группируется вместе как правый операнд сложения, а не `2 + 4` группируется вместе как левый операнд умножения. Это как если бы выражение было написано `2 + (4 * 5)`, а не `(2 + 4) * 5`. Таким образом, выражение выдает `2 + 20 == 22`, а не `6 * 5 == 30`.

Ассоциативность операторов определяет, что произойдет, если последовательность одних и тех же операторов будет использоваться одна за другой: обычно они будут сгруппированы слева или справа. Например, в `9 - 3 - 2` вычитание ассоциативно по левому краю, поэтому `9 - 3` группируется вместе как левый операнд второго вычитания, а не `3 - 2` группируется вместе как правый операнд первого вычитания. Это как если бы выражение было написано `(9 - 3) - 2`, а не `9 - (3 - 2)`. Таким образом, выражение выдает `6 - 2 == 4`, а не `9 - 1 == 8`.

Для простых операторов, которые вычисляют все свои операнды, а затем каким-либо образом объединяют значения, приоритет и ассоциативность (и круглые скобки) подразумевают некоторые требования к порядку выполнения этих операций объединения. Например, в `2 + 4 * 5` группировка, подразумеваемая приоритетом, означает, что умножение чисел 4 и 5 должно быть выполнено до сложения чисел 2 и 20, просто потому, что результат этого умножения требуется в качестве одного из операндов сложения. Но порядок операций этим определяется не полностью: в `2 * 2 + 4 * 5` оба умножения должны выполняться перед сложением, но группировка ничего не говорит о порядке, в котором выполняются два умножения. На самом деле в Perl существует общее правило, согласно которому операнды оператора вычисляются слева направо. Несколько операторов, таких как `&&=`, имеют специальные правила вычисления, которые могут привести к тому, что операнд вообще не будет вычислен; как правило, оператор верхнего уровня в выражении контролирует вычисление операнда.

Некоторые операторы сравнения, как и их ассоциативность, *связываются* с некоторыми операторами с одинаковым приоритетом (но никогда с операторами с разным приоритетом). Эта цепочка означает, что каждое сравнение выполняется для двух окружающих его аргументов, при этом каждый внутренний аргумент участвует в двух сравнениях, и результаты сравнения неявно передаются. Таким образом, `"$x < $y <= $z"` ведет себя точно так же, как `"$x < $y && $y <= $z"`, предполагая, что `"$y"` это такой же простой скаляр, как и кажется. ANDing замыкается точно так же, как это делает `"&&"`, останавливая последовательность сравнений, как только одно выдает false.

При последовательном сравнении каждое выражение аргумента вычисляется не более одного раза, даже если оно участвует в двух сравнениях, но результат вычисления извлекается для каждого сравнения. (Он вообще не вычисляется, если короткое замыкание означает, что он не требуется ни для каких сравнений.) Это имеет значение, если вычисление внутреннего аргумента является дорогостоящим или недетерминированным. Например,

```
if($x < expensive_sub() <= $z) { ...
```

не совсем похож

```
if($x < expensive_sub() && expensive_sub() <= $z) { ...
```

но вместо этого ближе к

```
my $tmp = expensive_sub();
if($x < $tmp && $tmp <= $z) { ...
```

в том смысле, что подпрограмма вызывается только один раз. Однако это также не совсем похоже на этот последний код, потому что цепочечное сравнение фактически не включает в себя какую-либо временную переменную (именованную или иную): присваивания нет. Это не имеет большого значения, когда выражение является вызовом обычной подпрограммы, но имеет большее значение с подпрограммой lvalue, или если выражение-аргумент выдает какой-то необычный вид скаляра другими способами. Например, если выражение-аргумент выдает связанный скаляр, то выражение вычисляется для получения этого скаляра не более одного раза, но значение этого скаляра может быть выбрано до двух раз, по одному разу для каждого сравнения, в котором оно фактически используется.

В этом примере выражение вычисляется только один раз, и связанный скаляр (результат выражения) извлекается для каждого сравнения, в котором оно используется.

```
if ($x < $tied_scalar < $z) { ...
```

В следующем примере выражение вычисляется только один раз, а связанный скаляр извлекается один раз как часть операции внутри выражения. Результат этой операции извлекается для каждого сравнения, что обычно не имеет значения, если только результат выражения также не является волшебным из-за перегрузки оператора.

```
if ($x < $tied_scalar + 42 < $z) { ...
```

Вместо этого некоторые операторы являются неассоциативными, что означает, что использование последовательности этих операторов с одинаковым приоритетом является синтаксической ошибкой. Например, `"$x .. $y .. $z"` является ошибкой.

Операторы Perl имеют следующую ассоциативность и приоритет, перечисленные от наивысшего приоритета к наименьшему. Операторы, заимствованные из C, сохраняют одинаковые отношения приоритета друг с другом, даже если приоритет C немного странный. (Это упрощает изучение Perl для людей, изучающих C.) За очень немногими исключениями, все они работают только со скалярными значениями, а не со значениями массива.

```
left    terms and list operators (leftward)
left    ->
nonassoc        ++ --
right   **
right   ! ~ ~. \ and unary + and -
left    =~ !~
left    * / % x
left    + - .
left    << >>
nonassoc        named unary operators
nonassoc    isa
chained         < > <= >= lt gt le ge
chain/na        == != eq ne <=> cmp ~~
left    & &.
left    | |. ^ ^.
left    &&
left    || //
nonassoc        ..  ...
right   ?:
right   = += -= *= etc. goto last next redo dump
left    , =>
nonassoc        list operators (rightward)
right   not
left    and
left    or xor
```

В следующих разделах эти операторы подробно описаны в том же порядке, в котором они приведены в таблице выше.

Многие операторы могут быть перегружены для объектов. См. раздел Перегрузка.

Термины и операторы списка (слева направо)

ТЕРМИН имеет наивысший приоритет в Perl. К ним относятся переменные, операторы, заключенные в кавычки, любые выражения в круглых скобках и любые функции, аргументы которых заключены в круглые скобки. На самом деле, на самом деле нет функций в этом смысле, просто операторы списка и унарные операторы, ведущие себя как функции, потому что вы заключаете аргументы в круглые скобки. Все это задокументировано в perlfunc.

Если за любым оператором списка (`print()` и т.д.) или любым унарным оператором (`chdir()` и т.д.) следует левая скобка в качестве следующего маркера, оператор и аргументы в круглых скобках считаются имеющими наивысший приоритет, точно так же, как при обычном вызове функции.

При отсутствии круглых скобок приоритет операторов списка, таких как `print` , `sort` или `chmod` , либо очень высокий, либо очень низкий, в зависимости от того, смотрите ли вы на левую или правую сторону оператора. Например, в

```
@ary = (1, 3, sort 4, 2);
print @ary;             # prints 1324
```

запятые справа от `sort` оцениваются перед `sort` , а запятые слева оцениваются после. Другими словами, операторы списков, как правило, поглощают все последующие аргументы, а затем действуют как простой термин по отношению к предыдущему выражению. Будьте осторожны со скобками:

```
# These evaluate exit before doing the print:
print($foo, exit);        # Obviously not what you want.
print $foo, exit; # Nor is this.

# These do the print before evaluating exit:
(print $foo), exit;       # This is what you want.
print($foo), exit;        # Or this.
print ($foo), exit;       # Or even this.
```

Также обратите внимание, что

```
print ($foo & 255) + 1, "\n";
```

возможно, на первый взгляд, он выполняет не то, что вы ожидаете. В скобках заключен список аргументов для `print` который вычисляется (выводится результат для `$foo & 255`). Затем к возвращаемому значению `print` добавляется единица (обычно 1). Результат получается примерно таким:

```
1 + 1, "\n";    # Obviously not what you meant.
```

Чтобы правильно выполнить то, что вы имели в виду, вы должны написать:

```
print(($foo & 255) + 1, "\n");
```

Смотрите "Именованные унарные операторы" для более подробного обсуждения этого.

Также в качестве терминов анализируются конструкции `do {}` и `eval {}`, а также вызовы подпрограмм и методов и анонимные конструкторы `[]` и `{}`.

Смотрите также "Операторы кавычек и подобные им операторы" ближе к концу этого раздела, а также "Операторы ввода-вывода".

Оператор стрелки

" -> " - это оператор инфиксного разыменования, как и в C и C ++. Если правая сторона представляет собой либо `[...]`, `{...}` либо `(...)` нижний индекс, то левая сторона должна быть либо твердой, либо символической ссылкой на массив, хэш или подпрограмму соответственно. (Или, технически говоря, местоположение, способное содержать жесткую ссылку, если это ссылка на массив или хэш, используемая для назначения.) Смотрите perlreftut и perlref.

В противном случае правая сторона представляет собой имя метода или простую скалярную переменную, содержащую либо имя метода, либо ссылку на подпрограмму, и (если это имя метода) левая сторона должна быть либо объектом (благословенная ссылка), либо именем класса (то есть именем пакета). Смотрите perlobj.

Возможности разыменования (в отличие от возможностей вызова методов) несколько расширены за счет `postderef` функции. Для получения подробной информации об этой функции обратитесь к "Синтаксису постфиксного разыменования" в perlref.

Автоматическое увеличение и автоматическое уменьшение

`"++"` и `"--"` работают как в C. То есть, если они помещены перед переменной, они увеличивают или уменьшают переменную на единицу перед возвратом значения, а если помещены после, увеличивают или уменьшают после возврата значения.

```
$i = 0;  $j = 0;
print $i++;  # prints 0
print ++$j;  # prints 1
```

Обратите внимание, что, как и в C, Perl не определяет, **когда** переменная увеличивается или уменьшается. Вы просто знаете, что это будет сделано когда-нибудь до или после возврата значения. Это также означает, что двукратное изменение переменной в одном и том же операторе приведет к неопределенному поведению. Избегайте операторов типа:

```
$i = $i ++;
print ++ $i + $i ++;
```

Perl не гарантирует результат выполнения приведенных выше инструкций.

В операторе автоматического увеличения есть немного дополнительной встроенной магии. Если вы увеличиваете числовую переменную или которая когда-либо использовалась в числовом контексте, вы получаете обычное приращение. Однако, если переменная использовалась только в строковых контекстах с момента ее установки и имеет значение, которое не является пустой строкой и соответствует шаблону `/^[a-zA-Z]*[0-9]*\z/`, приращение выполняется как строка с сохранением каждого символа в пределах его диапазона с переносом:

```
print ++($foo = "99");    # prints "100"
print ++($foo = "a0");    # prints "a1"
print ++($foo = "Az");    # prints "Ba"
print ++($foo = "zz");    # prints "aaa"
```

`undef` всегда обрабатывается как числовое значение и, в частности, изменяется на `0` перед увеличением (так что последующее увеличение значения undef будет возвращать `0`, а не `undef`).

Оператор автоматического уменьшения не является волшебным.

Возведение в степень

Двоичный код `"**"` - это оператор возведения в степень. Он связывает даже более жестко, чем унарный минус, поэтому `-2**4` есть `-(2**4)`, а не `(-2)**4`. (Это реализовано с помощью функции C `pow(3)`, которая на самом деле работает с удвоениями внутри.)

Обратите внимание, что некоторые выражения возведения в степень определены неточно: к ним относятся `0**0`, `1**Inf` и `Inf**0`. Не ожидайте каких-либо конкретных результатов от этих особых случаев, результаты зависят от платформы.

Символьные унарные операторы

Unary `"!"` выполняет логическое отрицание, то есть "не". Смотрите также `not` версию этого с более низким приоритетом.

Unary `"-"` выполняет арифметическое отрицание, если операнд числовой, включая любую строку, похожую на число. Если операндом является идентификатор, возвращается строка, состоящая из знака минус, объединенного с идентификатором. В противном случае, если строка начинается с плюса или минуса, возвращается строка, начинающаяся с противоположного знака. Одним из последствий этих правил является то,

что `-bareword` эквивалентно строке `"-bareword"`. Однако, если строка начинается с неалфавитного символа (исключая `"+"` или `"-"`), Perl попытается преобразовать строку в числовую, и будет выполнено арифметическое отрицание. Если строка не может быть преобразована в числовое значение, Perl выдаст предупреждение **Аргумент "строка" не является числовым в отрицании (-) в ....**

Унарный `"~"` выполняет побитовое отрицание, то есть дополнение к 1. Например, `0666 & ~027` равно 0640. (Смотрите также "Целочисленная арифметика" и "Побитовые строковые операторы".) Обратите внимание, что ширина результата зависит от платформы: `~0` имеет ширину 32 бита на 32-разрядной платформе, но 64 бита на 64-разрядной платформе, поэтому, если вы ожидаете определенную ширину в битах, не забудьте использовать оператор `"&"`, чтобы замаскировать лишние биты.

Начиная с Perl 5.28, попытка дополнить строку, содержащую символ, порядковым значением выше 255, является фатальной ошибкой.

Если "побитовая" функция включена через `use feature 'bitwise'` или `use v5.28`, то унарный оператор `"~"` всегда обрабатывает свой аргумент как число, а альтернативная форма оператора, `"~."`, всегда обрабатывает свой аргумент как строку. Таким образом, `~0` и `~"0"` оба выдадут 2 ** 32-1 на 32-разрядных платформах, тогда как `~.0` и `~."0"` оба выдадут `"\xff"`. До версии Perl 5.28 эта функция выдавала предупреждение в категории `"experimental::bitwise"`.

Унарный `"+"` не имеет никакого эффекта вообще, даже для строк. Синтаксически он полезен для отделения имени функции от заключенного в скобки выражения, которое в противном случае интерпретировалось бы как полный список аргументов функции. (Смотрите примеры выше в разделе "Операторы терминов и списков (слева)".)

Унарный `"\"` создает ссылки. Если его операндом является объект с одним символом, он создает ссылку на этот объект. Если его операндом является список, заключенный в скобки, то он создает ссылки на объекты, упомянутые в списке. В противном случае он помещает свой операнд в контекст списка и создает список ссылок на скаляры в списке, предоставленном операндом. Смотрите perlreftut и perlref. Не путайте это поведение с обратной косой чертой внутри строки, хотя обе формы действительно передают идею защиты следующего элемента от интерполяции.

Операторы привязки

Двоичный файл `"=~"` привязывает скалярное выражение к соответствию шаблону. Определенные операции выполняют поиск или изменение строки `$_` по умолчанию. Этот оператор выполняет такого рода операции с какой-либо другой строкой. Правый аргумент - это шаблон поиска, подстановка или транслитерация. Левый аргумент - это то, что предполагается искать, подставлять или транслитерировать вместо значения по умолчанию `$_`. При использовании в скалярном контексте возвращаемое значение обычно указывает на успешное выполнение операции. Исключениями являются подстановка (`s///`) и транслитерация (`y///`) с помощью `/r` (неразрушающей) опции, которые приводят к тому, что значение return является результатом подстановки. Поведение в контексте списка зависит от конкретного оператора. Смотрите "Операторы, подобные кавычкам в регулярных выражениях" для получения подробной информации и perlretut для примеров использования этих операторов.

Если правильный аргумент является выражением, а не шаблоном поиска, подстановки или транслитерации, он интерпретируется как шаблон поиска во время выполнения. Обратите внимание, что это означает, что его содержимое будет интерполировано дважды, поэтому

```
'\\' =~ q'\\';
```

не подходит, так как механизм регулярных выражений в конечном итоге попытается скомпилировать шаблон `\`, что он сочтет синтаксической ошибкой.

Двоичный код `"!~"` точно такой же, `"=~"` за исключением того, что возвращаемое значение отрицается в логическом смысле.

Двоичный файл `"!~"` с неразрушающей подстановкой (`s///r`) или транслитерацией (`y///r`) является синтаксической ошибкой.

Мультипликативные операторы

Двоичный код `"*"` умножает два числа.

Двоичный код `"/"` делит два числа.

Binary `"%"` - это оператор по модулю, который вычисляет остаток от деления своего первого аргумента по отношению ко второму аргументу. Заданные целочисленные операнды `$m` и `$n`: Если `$n` положительное значение, то `$m % $n` равно `$m` минус наибольшее кратное `$n` меньше или равно `$m`. Если `$n` значение отрицательное, то `$m % $n` равно `$m` минус наименьшее кратное `$n`, которое не меньше `$m` (то есть результат будет меньше или равен нулю). Если операнды `$m` и `$n` являются значениями с плавающей запятой, а абсолютное значение `$n` (то есть `abs($n)`) меньше `(UV_MAX + 1)`, в операции будет использоваться только целочисленная часть `$m` и `$n` (Примечание: здесь UV_MAX означает максимум целого типа без знака). Если абсолютное значение правого операнда (`abs($n)`) больше или равно `(UV_MAX + 1)`, `"%"` вычисляется остаток с плавающей запятой `$r` в уравнении, `($r = $m - $i*$n)` где `$i` это определенное целое число, которое делает `$r` иметь тот же знак, что и правый операнд `$n` (**не** как у левого операнда `$m`, как в C-функции `fmod()`), а абсолютное значение меньше, чем у `$n`. Обратите внимание, что когда `use integer` находится в области видимости, `"%"` предоставляет вам прямой доступ к оператору modulo, реализованному вашим компилятором C. Этот оператор не так хорошо определен для отрицательных операндов, но он будет выполняться быстрее.

Двоичный код `x` является оператором повторения. В скалярном контексте или если левый операнд не заключен ни в круглые скобки, ни в `qw//` список, он выполняет повторение строки. В этом случае он предоставляет скалярный контекст левому операнду и возвращает строку, состоящую из строки левого операнда, повторяющейся столько раз, сколько указано в правом операнде. Если `x` находится в контексте списка, а левый операнд либо заключен в круглые скобки, либо является `qw//` списком, выполняется повторение списка. В этом случае он предоставляет контекст списка левому операнду и возвращает список, состоящий из списка левых операндов, повторяющегося столько раз, сколько указано в правом операнде. Если правый операнд равен нулю или отрицателен (при отрицательном выводится предупреждение), он возвращает пустую строку или пустой список, в зависимости от контекста.

```
print '-' x 80;           # print row of dashes

print "\t" x ($tab/8), ' ' x ($tab%8);    # tab over

@ones = (1) x 80;                 # a list of 80 1's
@ones = (5) x @ones;     # set all elements to 5
```

Аддитивные операторы

Двоичный код `"+"` возвращает сумму двух чисел.

Двоичный код `"-"` возвращает разницу двух чисел.

Двоичный файл `"."` объединяет две строки.

Операторы сдвига

Binary `"<<"` возвращает значение своего левого аргумента, сдвинутое влево на количество бит, указанное правым аргументом. Аргументы должны быть целыми числами. (См. также "Целочисленная арифметика".)

Binary `">>"` возвращает значение левого аргумента, сдвинутое вправо на количество бит, указанное правым аргументом. Аргументы должны быть целыми числами. (См. также "Целочисленная арифметика".)

Если действует `use integer` (см. "Целочисленная арифметика"), то используются целые числа C со знаком (*арифметический сдвиг*), в противном случае используются целые числа C без знака (*логический сдвиг*), даже для отрицательных сдвигов. При арифметическом сдвиге вправо знаковый бит копируется слева, при логическом сдвиге нулевые биты поступают слева.

В любом случае реализация не будет генерировать результаты, превышающие размер целочисленного типа, с которым был создан Perl (32 или 64 бита).

Сдвиг на отрицательное количество бит означает обратный сдвиг: сдвиг влево становится сдвигом вправо, сдвиг вправо становится сдвигом влево. Это не похоже на C, где отрицательный сдвиг не определен.

Сдвиг на большее количество битов, чем размер целых чисел, в большинстве случаев означает ноль (все биты выпадают), за исключением того, что при `use integer` превышении вправо отрицательный сдвиг приводит к -1. Это не похоже на C, где сдвиг на слишком много битов не определен. Обычным поведением C является "сдвиг по модулю битов", так что, например

```
1 >> 64 == 1 >> (64 % 64) == 1 >> 0 == 1  # Common C behavior.
```

но это совершенно случайно.

Если вам надоест зависеть от собственных целых чисел вашей платформы, `use bigint` pragma аккуратно обходит проблему стороной:

```
print 20 << 20;  # 20971520
print 20 << 40;  # 5120 on 32-bit machines,
                  # 21990232555520 on 64-bit machines
use bigint;
print 20 << 100; # 25353012004564588029934064107520
```

Именованные унарные операторы

Различные именованные унарные операторы обрабатываются как функции с одним аргументом, с необязательными круглыми скобками.

Если за любым оператором списка (`print()` и т.д.) или любым унарным оператором (`chdir()` и т.д.) следует левая скобка в качестве следующего маркера, оператор и аргументы в круглых скобках считаются имеющими наивысший приоритет, точно так же, как при обычном вызове функции. Например, поскольку именованные унарные операторы имеют более высокий приоритет, чем `||` :

```
chdir $foo    || die;    # (chdir $foo) || die
chdir($foo)   || die;    # (chdir $foo) || die
chdir ($foo)  || die;    # (chdir $foo) || die
chdir +($foo) || die;    # (chdir $foo) || die
```

но, поскольку `"*"` имеет более высокий приоритет, чем именованные операторы:

```
chdir $foo * 20; # chdir ($foo * 20)
chdir($foo) * 20;         # (chdir $foo) * 20
chdir ($foo) * 20;        # (chdir $foo) * 20
chdir +($foo) * 20;       # chdir ($foo * 20)

rand 10 * 20;     # rand (10 * 20)
rand(10) * 20;    # (rand 10) * 20
rand (10) * 20;   # (rand 10) * 20
rand +(10) * 20; # rand (10 * 20)
```

Что касается приоритета, операторы filetest, такие как `-f`, `-M` и т.д., обрабатываются как именованные унарные операторы, но они не следуют этому правилу функциональных скобок. Это означает, например, что `-f($file).".bak"` эквивалентно `-f "$file.bak"`.

Смотрите также "Операторы терминов и списков (слева)".

Реляционные операторы

Операторы Perl, возвращающие true или false, обычно возвращают значения, которые можно безопасно использовать как числа. Например, операторы отношения в этом разделе и операторы равенства в следующем возвращают `1` значение true и специальную версию определенной пустой строки, `""` которая считается нулевой, но не содержит предупреждений о неправильных числовых преобразованиях, как и `"0 but true"` есть.

Binary `"<"` возвращает true, если левый аргумент численно меньше правого аргумента.

Binary `">"` возвращает true, если левый аргумент численно больше правого аргумента.

Binary `"<="` возвращает true, если левый аргумент численно меньше или равен правому аргументу.

Binary `">="` возвращает true, если левый аргумент численно больше или равен правому аргументу.

Binary `"lt"` возвращает true, если левый аргумент по строке меньше правого аргумента.

Binary `"gt"` возвращает true, если левый аргумент в строке больше правого аргумента.

Binary `"le"` возвращает true, если левый аргумент по строке меньше или равен правому аргументу.

Binary `"ge"` возвращает true, если левый аргумент в строке больше или равен правому аргументу.

Последовательность реляционных операторов, таких как `"$x < $y <= $z"`, выполняет цепные сравнения способом, описанным выше в разделе "Приоритет операторов и ассоциативность". Следите за тем, чтобы они не соединялись в цепочку с операторами равенства, которые имеют более низкий приоритет.

Операторы равенства

Binary `"=="` возвращает true, если левый аргумент численно равен правому аргументу.

Binary `"!="` возвращает true, если левый аргумент численно не равен правому аргументу.

Двоичный файл `"eq"` возвращает true, если левый аргумент в строке равен правому аргументу.

Binary `"ne"` возвращает true, если левый аргумент в строке не равен правому аргументу.

Последовательность вышеупомянутых операторов равенства, таких как `"$x == $y == $z"`, выполняет цепные сравнения способом, описанным выше в разделе "Приоритет операторов и ассоциативность". Следите за тем, чтобы они не соединялись в цепочку с реляционными операторами, которые имеют более высокий приоритет.

Двоичный код `"<=>"` возвращает -1, 0 или 1 в зависимости от того, численно ли левый аргумент меньше, равен или больше правого аргумента. Если ваша платформа поддерживает `NaN`'s (not-a-numbers) в качестве числовых значений, использование их с `"<=>"` возвращает `undef`. `NaN` не `"<"`, `"=="` `">"`, `"<="` `">="` или `NaN` вообще ничего (даже, что угодно), поэтому эти 5 возвращают `false`. `NaN != NaN` возвращает true, как и `NaN !=` *все остальное*. Если ваша платформа не поддерживает `NaN`'s, то `NaN` это просто строка с числовым значением `0`.

```
$ perl -le '$x = "NaN"; print "No NaN support here" if $x == $x'
$ perl -le '$x = "NaN"; print "NaN support here" if $x != $x'
```

(Обратите внимание, что все программы bigint, bigrat и bignum поддерживают `"NaN"`.)

Двоичный файл `"cmp"` возвращает -1, 0 или 1 в зависимости от того, является ли левый аргумент строковым меньше, равным или большим правого аргумента.

Здесь мы можем видеть разницу между <=> и cmp,

```
print 10 <=> 2 #prints 1
print 10 cmp 2 #prints -1
```

(likewise between gt and >, lt and <, etc.)

Binary "~~" does a smartmatch between its arguments. Smart matching is described in the next section.

Операторы двустороннего упорядочивания "<=>" и "cmp" и оператор smartmatch "~~" неассоциативны друг по отношению к другу и по отношению к операторам равенства с одинаковым приоритетом.

"lt", "le", "ge", "gt" и "cmp" используйте порядок сортировки, указанный текущим LC_COLLATE языком, если use locale действует форма, включающая параметры сортировки. Смотрите perllocale. Не смешивайте их с Unicode, используйте их только с устаревшими 8-разрядными языковыми кодировками. Стандартные Unicode::Collate и Unicode::Collate::Locale модули предлагают гораздо более эффективные решения проблем сортировки.

Для сравнений без учета регистра обратите внимание на "fc" в perlfunc функцию сворачивания регистра, доступную в Perl версии 5.16 или более поздней:

```
if ( fc($x) eq fc($y) ) { ... }
```

Оператор экземпляра класса

Двоичный файл isa принимает значение true, когда левый аргумент является экземпляром объекта класса (или подкласса, производного от этого класса), заданного правым аргументом. Если левый аргумент не определен, не является экземпляром blessed object и не является производным от класса, заданного правым аргументом, оператор вычисляется как false . Правильный аргумент может указывать класс либо в виде простого слова, либо в виде скалярного выражения, которое выдает строковое имя класса:

```
if( $obj isa Some::Class ) { ... }

if( $obj isa "Different::Class" ) { ... }
if( $obj isa $name_of_class ) { ... }
```

Эта функция доступна начиная с Perl 5.31.6 при включении с помощью use feature 'isa' . Эта функция включается автоматически с помощью use v5.36 (или выше) объявления в текущей области.

Оператор Smartmatch

Впервые доступный в Perl 5.10.1 (версия 5.10.0 работала по-другому), binary ~~ выполняет "интеллектуальное соответствие" между своими аргументами. В основном это используется неявно в when конструкции, описанной в perlsyn, хотя не все when предложения вызывают оператор smartmatch . Оператор smartmatch, уникальный среди всех операторов Perl, может выполнять рекурсию. Оператор smartmatch является экспериментальным, и его поведение может быть изменено.

Уникальность Perl также в том, что все другие операторы Perl накладывают контекст (обычно строковый или числовой контекст) на свои операнды, автоматически преобразуя эти операнды в эти наложенные контексты. В отличие от этого, smartmatch *выводит* контексты из фактических типов своих операндов и использует информацию о типе для выбора подходящего механизма сравнения.

~~ Оператор сравнивает свои операнды "полиморфно", определяя, как их сравнивать в соответствии с их фактическими типами (числовой, строковый, массивный, хэш и т.д.). Подобно операторам равенства, с которыми он имеет одинаковый приоритет, ~~ возвращает 1 для true и "" для false . Часто лучше читать вслух как "в", "внутри" или "содержится в", потому что левый операнд часто ищут *внутри* правого операнда. Это приводит к тому, что порядок следования операндов в операнде smartmatch часто противоположен порядку следования обычного оператора match. Другими словами, "меньший" элемент обычно помещается в левый операнд, а больший - в правый.

Поведение smartmatch зависит от того, к какому типу относятся его аргументы, как определено в следующей таблице. Первая строка таблицы, типы которой применяются, определяет поведение smartmatch. Поскольку то, что на самом деле происходит, в основном определяется типом второго операнда, таблица сортируется по правому операнду, а не по левому.

```
Left      Right      Description and pseudocode
================================================================
Any       undef      check whether Any is undefined
               like: !defined Any

Any       Object     invoke ~~ overloading on Object, or die

Right operand is an ARRAY:

Left      Right      Description and pseudocode
================================================================
ARRAY1    ARRAY2     recurse on paired elements of ARRAY1 and ARRAY2[2]
               like: (ARRAY1[0] ~~ ARRAY2[0])
                       && (ARRAY1[1] ~~ ARRAY2[1]) && ...
HASH      ARRAY      any ARRAY elements exist as HASH keys
               like: grep { exists HASH->{$_} } ARRAY
Regexp    ARRAY      any ARRAY elements pattern match Regexp
               like: grep { /Regexp/ } ARRAY
undef     ARRAY      undef in ARRAY
               like: grep { !defined } ARRAY
Any       ARRAY      smartmatch each ARRAY element[3]
               like: grep { Any ~~ $_ } ARRAY

Right operand is a HASH:

Left      Right      Description and pseudocode
================================================================
HASH1     HASH2      all same keys in both HASHes
               like: keys HASH1 ==
                        grep { exists HASH2->{$_} } keys HASH1
ARRAY     HASH       any ARRAY elements exist as HASH keys
               like: grep { exists HASH->{$_} } ARRAY
Regexp    HASH       any HASH keys pattern match Regexp
               like: grep { /Regexp/ } keys HASH
undef     HASH       always false (undef cannot be a key)
               like: 0 == 1
Any       HASH       HASH key existence
               like: exists HASH->{Any}

Right operand is CODE:

Left      Right      Description and pseudocode
================================================================
ARRAY     CODE       sub returns true on all ARRAY elements[1]
             like: !grep { !CODE->($_) } ARRAY
HASH      CODE       sub returns true on all HASH keys[1]
             like: !grep { !CODE->($_) } keys HASH
Any       CODE       sub passed Any returns true
             like: CODE->(Any)

Right operand is a Regexp:

Left      Right      Description and pseudocode
================================================================
ARRAY     Regexp     any ARRAY elements match Regexp
              like: grep { /Regexp/ } ARRAY
HASH      Regexp     any HASH keys match Regexp
               like: grep { /Regexp/ } keys HASH
Any       Regexp     pattern match
               like: Any =~ /Regexp/

Other:

Left      Right      Description and pseudocode
================================================================
Object    Any        invoke ~~ overloading on Object,
                     or fall back to...

Any       Num        numeric equality
                like: Any == Num
Num       nummy[4]    numeric equality
                like: Num == nummy
undef     Any        check whether undefined
                like: !defined(Any)
Any       Any        string equality
                like: Any eq Any
```

Notes:

1. Empty hashes or arrays match.
2. That is, each element smartmatches the element of the same index in the other array.[3]
3. Если найдена циклическая ссылка, вернитесь к равенству ссылок.
4. Либо фактическое число, либо строка, похожая на него.

smartmatch неявно разыменовывает любую ссылку на хэш или массив, не являющуюся благословенной, поэтому в этих случаях применяются записи *HASH* и *ARRAY*. Для благословенных ссылок применяются *Object* записи. Интеллектуальные сопоставления, включающие хэши, учитывают только ключи хэша, но никогда хэш-значения.

Ввод кода "нравится" не всегда является точным отображением. Например, оператор smartmatch выполняет короткое замыкание, когда это возможно, но `grep` этого не делает. Кроме того, `grep` в скалярном контексте возвращает количество совпадений, но `~~` возвращает только true или false.

В отличие от большинства операторов, оператор smartmatch умеет обращаться с `undef` особым образом:

```
use v5.10.1;
@array = (1, 2, 3, undef, 4, 5);
say "some elements undefined" if undef ~~ @array;
```

Каждый операнд рассматривается в модифицированном скалярном контексте, модификация заключается в том, что массив и хэш-переменные передаются по ссылке на оператор, который неявно разыменовывает их. Оба элемента каждой пары одинаковы.:

```
use v5.10.1;

my %hash = (red    => 1, blue   => 2, green  => 3,
            orange => 4, yellow => 5, purple => 6,
            black  => 7, grey   => 8, white  => 9);

my @array = qw(red blue green);

say "some array elements in hash keys" if  @array ~~  %hash;
say "some array elements in hash keys" if \@array ~~ \%hash;

say "red in array" if "red" ~~  @array;
say "red in array" if "red" ~~ \@array;

say "some keys end in e" if /e$/ ~~  %hash;
say "some keys end in e" if /e$/ ~~ \%hash;
```

Два массива smartmatching, если каждый элемент в первом массиве smartmatches (то есть находится "в") соответствующему элементу во втором массиве, рекурсивно.

```
use v5.10.1;
my @little = qw(red blue green);
my @bigger = ("red", "blue", [ "orange", "green" ] );
if (@little ~~ @bigger) {  # true!
    say "little is contained in bigger";
}
```

Поскольку оператор smartmatch выполняет рекурсию во вложенных массивах, он все равно сообщит, что в массиве есть "red".

```
use v5.10.1;
my @array = qw(red blue green);
my $nested_array = [[[[[[[ @array ]]]]]]];
say "red in array" if "red" ~~ $nested_array;
```

Если два массива соответствуют друг другу, то они являются глубокими копиями значений друг друга, как показано в этом примере:

```
use v5.12.0;
my @a = (0, 1, 2, [3, [4, 5], 6], 7);
my @b = (0, 1, 2, [3, [4, 5], 6], 7);

if (@a ~~ @b && @b ~~ @a) {
    say "a and b are deep copies of each other";
}
elsif (@a ~~ @b) {
    say "a smartmatches in b";
}
elsif (@b ~~ @a) {
    say "b smartmatches in a";
}
else {
    say "a and b don't smartmatch each other at all";
}
```

Если вы должны были установить `$b[3] = 4`, то вместо сообщения о том, что "a и b являются глубокими копиями друг друга", теперь он сообщает об этом `"b smartmatches in a"`. Это потому, что соответствующая позиция в `@a` содержит массив, в котором (в конечном итоге) есть 4.

Интеллектуальное сопоставление одного хэша с другим сообщает, содержат ли оба одинаковых ключа, не больше и не меньше. Это можно использовать, чтобы увидеть, имеют ли две записи одинаковые имена полей, не заботясь о том, какие значения могут иметь эти поля. Например:

```
use v5.10.1;
sub make_dogtag {
    state $REQUIRED_FIELDS = { name=>1, rank=>1, serial_num=>1 };

    my ($class, $init_fields) = @_;

    die "Must supply (only) name, rank, and serial number"
        unless $init_fields ~~ $REQUIRED_FIELDS;

    ...
}
```

Однако это делает только то, что вы имеете в виду, если `$init_fields` действительно является ссылкой на хэш. Условие `$init_fields ~~ $REQUIRED_FIELDS` также позволяет передавать строки `"name"`, `"rank"`, `"serial_num"` а также любую ссылку на массив, которая содержит `"name"` or `"rank"` или `"serial_num"` где угодно.

Оператор smartmatch чаще всего используется как неявный оператор `when` предложения. Смотрите раздел "Операторы переключения" в perlsyn.

Интеллектуальное сопоставление объектов

Чтобы не полагаться на базовое представление объекта, если правый операнд smartmatch является объектом, который не перегружается `~~`, возникает исключение "`Smartmatching a non-overloaded object breaks encapsulation`". Это потому, что никому не нужно копаться, чтобы увидеть, находится ли что-то "внутри" объекта. Все это незаконно для объектов без `~~` перегрузки:

```
 %hash ~~ $object
    42 ~~ $object
"fred" ~~ $object
```

Однако вы можете изменить способ интеллектуального сопоставления объекта, перегрузив `~~` оператор. Это разрешено для расширения обычной семантики интеллектуального сопоставления. Для объектов, которые действительно имеют `~~` перегрузку, см. раздел Перегрузка.

Использование объекта в качестве левого операнда разрешено, хотя и не очень полезно. Правила Smartmatch имеют приоритет над перегрузкой, поэтому, даже если объект в левом операнде имеет перегрузку smartmatch, это будет проигнорировано. Левый операнд, который не является перегруженным объектом, возвращается к строковому или числовому сравнению всего, что возвращает `ref` оператор. Это означает, что

```
$object ~~ X
```

не вызывает метод перегрузки с *X* в качестве аргумента. Вместо этого к приведенной выше таблице обращаются как обычно, и в зависимости от типа *X* может быть вызвана перегрузка, а может и не быть. Для простых строк или чисел "in" становится эквивалентным этому:

```
$object ~~ $number          ref($object) == $number
$object ~~ $string          ref($object) eq $string
```

Например, здесь сообщается, что дескриптор пахнет IOish (но, пожалуйста, на самом деле не делайте этого!):

```
use IO::Handle;
my $fh = IO::Handle->new();
if ($fh ~~ /\bIO\b/) {
    say "handle smells IOish";
}
```

Это потому, что он обрабатывает `$fh` как строку типа `"IO::Handle=GLOB(0x8039e0)"`, а затем сопоставляет шаблон с этим.

Побитовый и

Двоичный файл `"&"` возвращает свои операнды, объединенные по битам. Хотя в данный момент предупреждение не выдается, результат не совсем определен, когда эта операция выполняется над операндами, которые не являются ни числами (см. "Целочисленная арифметика"), ни битовыми строками (см. "Операторы побитовой строки").

Обратите внимание, что `"&"` имеет более низкий приоритет, чем реляционные операторы, поэтому, например, круглые скобки необходимы в таком тесте, как

```
print "Even\n" if ($x & 1) == 0;
```

Если функция "побитовой" включена с помощью `use feature 'bitwise'` или `use v5.28`, то этот оператор всегда обрабатывает свои операнды как числа. До версии Perl 5.28 эта функция выдавала предупреждение в категории `"experimental::bitwise"`.

Побитовое Или и исключительное Или

Двоичный файл `"|"` возвращает свои операнды, собранные вместе, бит за битом.

Двоичный файл `"^"` возвращает свои операнды, преобразованные вместе, бит за битом.

Хотя в настоящее время предупреждение не выдается, результаты не совсем определены, когда эти операции выполняются над операндами, которые не являются ни числами (см. "Целочисленная арифметика"), ни битовыми строками (см. "Операторы побитовой строки").

Обратите внимание, что `"|"` и `"^"` имеют более низкий приоритет, чем операторы отношения, поэтому, например, круглые скобки необходимы в таком тесте, как

```
print "false\n" if (8 | 2) != 10;
```

Если функция "побитовой" включена с помощью `use feature 'bitwise'` или `use v5.28`, то этот оператор всегда обрабатывает свои операнды как числа. До версии Perl 5.28 эта функция выдавала предупреждение в категории `"experimental::bitwise"`.

Логические и

Двоичный файл `"&&"` выполняет логическую операцию с коротким замыканием И. То есть, если левый операнд равен false, правый операнд даже не вычисляется. Скалярный контекст или контекст списка распространяется вниз к правому операнду, если он вычисляется.

Логическое или

Binary `"||"` performs a short-circuit logical OR operation. That is, if the left operand is true, the right operand is not even evaluated. Scalar or list context propagates down to the right operand if it is evaluated.

Logical Defined-Or

Although it has no direct equivalent in C, Perl's `//` operator is related to its C-style "or". In fact, it's exactly the same as `||`, except that it tests the left hand side's definedness instead of its truth. Thus, `EXPR1 // EXPR2` returns the value of `EXPR1` if it's defined, otherwise, the value of `EXPR2` is returned. (`EXPR1` is evaluated in scalar context, `EXPR2` in the context of `//` itself). Usually, this is the same result as `defined(EXPR1) ? EXPR1 : EXPR2` (except that the ternary-operator form can be used as a lvalue, while `EXPR1 // EXPR2` cannot). This is very useful for providing default values for variables. If you actually want to test if at least one of `$x` and `$y` is defined, use `defined($x // $y)`.

The `||`, `//` and `&&` operators return the last value evaluated (unlike C's `||` and `&&`, which return 0 or 1). Thus, a reasonably portable way to find out the home directory might be:

```
    $home =  $ENV{HOME}
          // $ENV{LOGDIR}
          // (getpwuid($<))[7]
          // die "You're homeless!\n";
```

В частности, это означает, что вам не следует использовать это для выбора между двумя агрегатами для назначения:

```
@a = @b || @c;            # This doesn't do the right thing
@a = scalar(@b) || @c;    # because it really means this.
@a = @b ? @b : @c;        # This works fine, though.
```

В качестве альтернативы && и || при использовании для потока управления Perl предоставляет операторы and и or (см. Ниже). Поведение при коротком замыкании идентично. Однако приоритет "and" и "or" намного ниже, так что вы можете безопасно использовать их после оператора списка без необходимости использования круглых скобок:

```
    unlink "alpha", "beta", "gamma"
            or gripe(), next LINE;
```

С операторами в стиле C, которые были бы написаны следующим образом:

```
    unlink("alpha", "beta", "gamma")
            || (gripe(), next LINE);
```

Было бы еще удобнее написать это таким образом:

```
unless(unlink("alpha", "beta", "gamma")) {
    gripe();
    next LINE;
}
```

Использование "or" для назначения вряд ли приведет к тому, что вы хотите; смотрите Ниже.

Операторы диапазона

Binary ".." - это оператор диапазона, который на самом деле является двумя разными операторами в зависимости от контекста. В контексте списка он возвращает список значений, считая (с увеличением на единицы) от левого значения к правому. Если левое значение больше правого, то возвращается пустой список. Оператор range полезен для написания foreach (1..10) циклов и для выполнения операций среза над массивами. В текущей реализации временный массив не создается, когда оператор range используется в качестве выражения в foreach циклах, но более старые версии Perl могут занимать много памяти при написании чего-то подобного:

```
    for (1 .. 1_000_000) {
        # code
    }
```

Оператор range также работает со строками, используя волшебное автоматическое увеличение, см. Ниже.

В скалярном контексте ".." возвращает логическое значение. Оператор является бистабильным, подобно триггеру, и эмулирует оператор диапазона строк (запятой) из **sed**, **awk** и различных редакторов. Каждый ".." оператор поддерживает свое собственное логическое состояние даже при вызовах подпрограммы, которая его содержит. Это значение равно false, пока его левый операнд равен false . Как только левый операнд принимает значение true, оператор диапазона остается true до тех пор, пока правый операнд не примет значение true, *ПОСЛЕ* чего оператор диапазона снова становится false. Значение не становится ложным до следующего вычисления оператора диапазона. Он может протестировать правильный операнд и стать false при той же оценке, при которой он стал true (как в **awk**), но все равно возвращает true один раз. Если вы не хотите, чтобы он проверял правильный операнд до следующего вычисления, как в **sed**, просто используйте три точки ("...") вместо двух. Во всех остальных отношениях "..." ведет себя точно так же, как ".." делает.

Правый операнд не вычисляется, пока оператор находится в состоянии "false", а левый операнд не вычисляется, пока оператор находится в состоянии "true". Приоритет немного ниже, чем || и && . Возвращаемое значение представляет собой либо пустую строку для false, либо порядковый номер (начинающийся с 1) для true. Порядковый номер сбрасывается для каждого встречающегося диапазона. К конечному порядковому номеру в диапазоне добавляется строка "E0" , которая не влияет на его числовое значение, но дает вам что-то для поиска, если вы хотите исключить конечную точку. Вы можете исключить начальную точку, дождавшись, пока порядковый номер будет больше 1.

Если любой из операндов scalar ".." является постоянным выражением, этот операнд считается истинным, если он равен (==) текущему номеру строки ввода ($. переменной).

Если быть педантичным, сравнение на самом деле выполняется int(EXPR) == int(EXPR) , но это проблема только в том случае, если вы используете выражение с плавающей запятой; при неявном использовании $. , как описано в предыдущем параграфе, сравнение выполняется, int(EXPR) == int($.) что является проблемой только тогда, когда $. установлено значение с плавающей запятой, и вы не читаете из файла. Кроме того, "span" .. "spat" or 2.18 .. 3.14 не будет делать то, что вы хотите в скалярном контексте, потому что каждый из операндов вычисляется с использованием их целочисленного представления.

Примеры:

Как скалярный оператор:

```
if (101 .. 200) { print; } # print 2nd hundred lines, short for
                           #  if ($. == 101 .. $. == 200) { print; }

next LINE if (1 .. /^$/);  # skip header lines, short for
                           #   next LINE if ($. == 1 .. /^$/);
                           # (typically in a loop labeled LINE)

s/^/> / if (/^$/ .. eof());  # quote body

# parse mail messages
while (<>) {
    $in_header =   1  .. /^$/;
    $in_body   = /^$/ .. eof;
    if ($in_header) {
        # do something
    } else { # in body
        # do something else
    }
} continue {
    close ARGV if eof;              # reset $. each file
}
```

Вот простой пример, иллюстрирующий разницу между двумя операторами диапазона:

```
@lines = ("   - Foo",
          "01 - Bar",
          "1  - Baz",
          "   - Quux");

foreach (@lines) {
    if (/0/ .. /1/) {
        print "$_\n";
    }
}
```

Эта программа напечатает только строку, содержащую "Bar". Если оператор диапазона изменить на `...`, она также напечатает строку "Baz".

А теперь несколько примеров в качестве оператора списка:

```
for (101 .. 200) { print }      # print $_ 100 times
@foo = @foo[0 .. $#foo];        # an expensive no-op
@foo = @foo[$#foo-4 .. $#foo];  # slice last 5 items
```

Поскольку каждый операнд вычисляется в целочисленной форме, `2.18 .. 3.14` вернет два элемента в контексте списка.

```
@list = (2.18 .. 3.14); # same as @list = (2 .. 3);
```

Оператор диапазона в контексте списка может использовать волшебный алгоритм автоматического увеличения, если оба операнда являются строками, при соблюдении следующих правил:

- За одним исключением (приведенным ниже), если обе строки выглядят как числа в Perl, магическое приращение применяться не будет, и вместо этого строки будут обрабатываться как числа (более конкретно, целые числа).
  Например, `"-2".."2"` это то же самое, что `-2..2`, и `"2.18".."3.14"` выдает `2, 3`.
- Исключением из приведенного выше правила является случай, когда левая строка начинается с `0` и длиннее одного символа, в этом случае *будет* применено магическое приращение, хотя строки типа `"01"` обычно выглядят в Perl как числа.
  Например, `"01".."04"` производит `"01", "02", "03", "04"` и `"00".."-1"` производит `"00"` через `"99"` - это может показаться удивительным, но смотрите следующие правила, почему это работает именно так. Чтобы получить даты с начальными нулями, вы можете сказать:

  ```
  @z2 = ("01" .. "31");
  print $z2[$mday];
  ```

  Если вы хотите принудительно интерпретировать строки как числа, вы могли бы сказать

  ```
  @numbers = ( 0+$first .. 0+$last );
  ```

  Примечание: В Perl версий 5.30 и ниже, *любая* строка в левой части, начинающаяся с `"0"`, включая саму строку `"0"`, будет вызывать поведение приращения волшебной строки. Это означает, что в этих версиях Perl, `"0".."-1"` будет выдавать `"0"` through `"99"`, что несовместимо с `0..-1`, который выдает пустой список. Это также означает, что `"0".."9"` теперь выдает список целых чисел вместо списка строк.

- Если указанное начальное значение не является частью магической последовательности приращений (то есть непустой строки, соответствующей `/^[a-zA-Z]*[0-9]*\z/`), будет возвращено только начальное значение.
  Например, `"ax".."az"` производит `"ax"`, `"ay"`, `"az"`, но `"*x".."az"` производит только `"*x"`.
- Для других начальных значений, представляющих собой строки, которые действительно следуют правилам магического приращения, будет возвращена соответствующая последовательность.
  Например, вы можете сказать

  ```
  @alphabet = ("A" .. "Z");
  ```

  to get all normal letters of the English alphabet, or

  ```
  $hexdigit = (0 .. 9, "a" .. "f")[$num & 15];
  ```

  чтобы получить шестнадцатеричную цифру.
- Если указанное конечное значение не входит в последовательность, которую произвело бы магическое приращение, последовательность продолжается до тех пор, пока следующее значение не будет длиннее указанного конечного значения. Если длина последней строки короче первой, возвращается пустой список.
  Например, `"a".."--"` это то же самое, что `"a".."zz"`, `"0".."xx"` выдает `"0"` через `"99"`, и `"aaa".."--"` возвращает пустой список.

Начиная с Perl 5.26, оператор list-context range для строк работает так, как ожидалось в области `"use feature 'unicode strings"`. В предыдущих версиях и вне рамок этой функции он демонстрирует "Ошибку Unicode"" в perlunicode": его поведение зависит от внутренней кодировки конечной точки диапазона.

Поскольку магическое приращение работает только с непустыми строками, совпадающими `/^[a-zA-Z]*[0-9]*\z/`, следующее вернет только альфа-символ:

```
use charnames "greek";
my @greek_small =  ("\N{alpha}" .. "\N{omega}");
```

Чтобы получить 25 традиционных строчных греческих букв, включая оба знака, вы могли бы использовать это вместо:

```
use charnames "greek";
my @greek_small =  map { chr } ( ord("\N{alpha}")
                                 ..
                                 ord("\N{omega}")
                               );
```

Однако, поскольку существует *много* других строчных греческих символов, кроме этих, для сопоставления строчных греческих символов в регулярном выражении вы могли бы использовать шаблон `/(?:(?=\p{Greek})\p{Lower})+/` (или экспериментальную функцию `/(?[ \p{Greek} & \p{Lower} ])+/`).

Условный оператор

Тернарный `"?:"` является условным оператором, как и в C. Он работает во многом как if-then-else. Если аргумент перед `?` равен true, возвращается аргумент перед `:`, в противном случае возвращается аргумент после `:`. Например:

```
    printf "I have %d dog%s.\n", $n,
            ($n == 1) ? "" : "s";
```

Скалярный контекст или контекст списка распространяется вниз до 2-го или 3-го аргумента, в зависимости от того, что выбрано.

```
$x = $ok ? $y : $z;  # get a scalar
@x = $ok ? @y : @z;  # get an array
$x = $ok ? @y : @z;  # oops, that's just a count!
```

Оператор может быть назначен, если оба аргумента 2-й и 3-й являются допустимыми значениями (что означает, что вы можете назначить им):

```
($x_or_y ? $x : $y) = $z;
```

Поскольку этот оператор выдает присваиваемый результат, использование присваиваний без круглых скобок приведет к проблемам. Например, это:

```
$x % 2 ? $x += 10 : $x += 2
```

На самом деле это означает:

```
((($x % 2) ? ($x += 10) : $x) += 2
```

Вместо этого:

```
($x % 2) ? ($x += 10) : ($x += 2)
```

Вероятно, это следует записать проще как:

```
$x += ($x % 2) ? 10 : 2;
```

Операторы присваивания

"=" это обычный оператор присваивания.

Операторы присваивания работают так же, как в C. То есть,

```
$x += 2;
```

эквивалентно

```
$x = $x + 2;
```

хотя и без дублирования каких-либо побочных эффектов, которые может вызвать разыменование lvalue, например from tie() . Другие операторы присваивания работают аналогично. Распознаются следующие:

```
**=    +=    *=    &=    &.=    <<=    &&=
       -=    /=    |=    |.=    >>=    ||=
       .=    %=    ^=    ^.=           //=
             x=
```

Хотя они сгруппированы по семействам, все они имеют приоритет присваивания. Эти комбинированные операторы присваивания могут работать только со скалярами, тогда как обычный оператор присваивания может присваивать массивы, хэши, списки и даже ссылки. (См. "Контекст" и "Конструкторы значений списка" в perldata и "Присвоение ссылкам" в perlref.)

В отличие от C, скалярный оператор присваивания выдает допустимое значение lvalue . Изменение присваивания эквивалентно выполнению присваивания с последующим изменением переменной, которой было присвоено значение. Это полезно для изменения копии чего-либо, например, этого:

```
($tmp = $global) =~ tr/13579/24680/;
```

Хотя, начиная с версии 5.14, это также может быть достигнуто таким способом:

```
use v5.14;
$tmp = ($global =~  tr/13579/24680/r);
```

Аналогично,

```
($x += 2) *= 3;
```

эквивалентно

```
$x += 2;
$x *= 3;
```

Аналогично, назначение списка в контексте списка создает список назначенных значений, а назначение списка в скалярном контексте возвращает количество элементов, созданных выражением в правой части назначения.

Три точечных оператора побитового присваивания (&.= |.= ^.=) являются новыми в Perl 5.22. Смотрите "Операторы побитовой строки".

Оператор запятой

Двоичный код "," - это оператор запятой. В скалярном контексте он вычисляет свой левый аргумент, отбрасывает это значение, затем вычисляет свой правый аргумент и возвращает это значение. Это точно так же, как оператор запятой в C.

В контексте списка это просто разделитель аргументов списка, который вставляет оба своих аргумента в список. Эти аргументы также вычисляются слева направо.

Оператор `=>` (иногда произносится как "жирная запятая") является синонимом запятой, за исключением того, что из-за него слово слева от него интерпретируется как строка, если оно начинается с буквы или подчеркивания и состоит только из букв, цифр и подчеркиваний. Сюда входят операнды, которые в противном случае могли бы интерпретироваться как операторы, константы, v-строки с одним числом или вызовы функций. Если вы сомневаетесь в таком поведении, левый операнд можно явно заключить в кавычки.

В противном случае, `=>` оператор ведет себя точно так же, как оператор запятой или разделитель аргументов списка, в зависимости от контекста.

Например:

```
use constant FOO => "something";

my %h = ( FOO => 23 );
```

эквивалентно:

```
my %h = ("FOO", 23);
```

Это *НЕ*:

```
my %h = ("something", 23);
```

Оператор `=>` полезен для документирования соответствия между ключами и значениями в хэшах и другими сопряженными элементами в списках.

```
%hash = ( $key => $value );
login( $username => $password );
```

Специальное поведение при цитировании игнорирует приоритет и, следовательно, может применяться к *части* левого операнда:

```
print time.shift => "bbb";
```

Этот пример выводит что-то вроде `"1314363215shiftbbb"`, потому что `=>` неявно заключает в кавычки `shift` сразу слева, игнорируя тот факт, что `time.shift` это весь левый операнд.

Список операторов (справа)

В правой части оператора списка запятая имеет очень низкий приоритет, так что она управляет всеми найденными там выражениями, разделенными запятыми. Единственными операторами с более низким приоритетом являются логические операторы `"and"`, `"or"` и `"not"`, которые могут использоваться для оценки вызовов операторов списка без использования круглых скобок:

```
open HANDLE, "< :encoding(UTF-8)", "filename"
    or die "Can't open: $!\n";
```

Однако некоторым людям читать такой код сложнее, чем писать его в круглых скобках:

```
open(HANDLE, "< :encoding(UTF-8)", "filename")
    or die "Can't open: $!\n";
```

в этом случае вы могли бы просто использовать более привычный `"||"` оператор:

```
open(HANDLE, "< :encoding(UTF-8)", "filename")
    || die "Can't open: $!\n";
```

Смотрите также обсуждение операторов списков в "Термины и операторы списков (слева направо)".

Логическое не

Unary `"not"` возвращает логическое отрицание выражения справа от него. Это эквивалент `"!"` за исключением очень низкого приоритета.

Логические и

Двоичный файл `"and"` возвращает логическое соединение двух окружающих выражений. Это эквивалентно `&&` за исключением очень низкого приоритета. Это означает, что происходит короткое замыкание: правое выражение вычисляется только в том случае, если левое выражение равно true.

Логическое или и исключительное или

Двоичный файл "or" возвращает логическую дизъюнкцию двух окружающих выражений. Это эквивалентно || за исключением очень низкого приоритета. Это делает его полезным для потока управления.:

```
print FH $data           or die "Can't write to FH: $!";
```

Это означает, что происходит короткое замыкание: правое выражение вычисляется только в том случае, если левое выражение равно false . Из-за его приоритета вы должны быть осторожны, чтобы не использовать его в качестве замены оператора || . Обычно это работает лучше для управления потоком, чем при назначениях:

```
$x = $y or $z;              # bug: this is wrong
($x = $y) or $z;            # really means this
$x = $y || $z;              # better written this way
```

Однако, когда это назначение в контексте списка и вы пытаетесь использовать || для потока управления, вам, вероятно, нужно "or" чтобы назначение имело более высокий приоритет.

```
@info = stat($file) || die;     # oops, scalar sense of stat!
@info = stat($file) or die;     # better, now @info gets its due
```

Опять же, вы всегда можете использовать круглые скобки.

Двоичный файл "xor" возвращает исключающее-ИЛИ из двух окружающих выражений. Это не может привести к короткому замыканию (конечно).

Для defined-OR не существует оператора с низким приоритетом.

Операторы C Отсутствуют в Perl

Вот что есть в C, чего нет в Perl:

унарный &

Адрес оператора. (Но смотрите "\" оператор для получения ссылки.)

унарный *

Оператор адреса разыменования. (Операторы разыменования префиксов Perl вводятся следующим образом: $, @ , % и & .)

(ТИП)

Оператор приведения типов.

Кавычки и операторы, подобные кавычкам

Хотя мы обычно думаем о кавычках как о литеральных значениях, в Perl они функционируют как операторы, предоставляя различные виды интерполяции и возможностей сопоставления с образцом. Perl предоставляет обычные символы кавычек для этих действий, но также предоставляет вам возможность выбрать свой символ кавычек для любого из них. В следующей таблице a {} представляет любую пару выбранных вами разделителей.

```
    Customary  Generic        Meaning        Interpolates
        ''       q{}          Literal             no
        ""      qq{}          Literal             yes
        ``      qx{}          Command             yes*
                qw{}         Word list            no
        //       m{}       Pattern match          yes*
                qr{}          Pattern             yes*
                 s{}{}      Substitution          yes*
                tr{}{}    Transliteration         no (but see below)
                 y{}{}    Transliteration         no (but see below)
        <<EOF                 here-doc            yes*

        * unless the delimiter is ''.
```

Разделители, не заключающие в квадратные скобки, используют один и тот же символ спереди и сзади, но все четыре вида скобок ASCII (круглые, угловые, квадратные, фигурные) являются вложенными, что означает, что

```
q{foo{bar}baz}
```

совпадает с

```
'foo{bar}baz'
```

Обратите внимание, однако, что это не всегда работает для цитирования кода Perl:

```
$s = q{ if($x eq "}") ... }; # WRONG
```

синтаксическая ошибка. Text::Balanced Модуль (стандартный начиная с версии 5.8 и из CPAN до этого) способен выполнять это должным образом.

Между оператором и символами, заключенными в кавычки, может (а в некоторых случаях и должен) быть пробел, за исключением случаев, когда # используется в качестве символа, заключающего в кавычки. q#foo# анализируется как строка foo , в то время как q #foo# это оператор q , за которым следует комментарий. Его аргумент будет взят из следующей строки. Это позволяет вам писать:

```
s {foo}  # Replace foo
  {bar}  # with bar.
```

В случаях, когда необходимо использовать пробел, это когда символ, заключающий в кавычки, является символом слова (что означает, что он совпадает /\w/):

```
q XfooX # Works: means the string 'foo'
qXfooX  # WRONG!
```

Следующие управляющие последовательности доступны в конструкциях с интерполяцией и в транслитерациях, разделителями которых не являются одинарные кавычки ("'"). Во всех операторах с фигурными скобками допускается (и игнорируется) любое количество пробелов и / или табуляций, примыкающих к фигурным скобкам и находящихся внутри них.

```
Sequence     Note  Description
\t                 tab              (HT, TAB)
\n                 newline          (NL)
\r                 return           (CR)
\f                 form feed        (FF)
\b                 backspace        (BS)
\a                 alarm (bell)     (BEL)
\e                 escape           (ESC)
\x{263A}     [1,8] hex char         (example shown: SMILEY)
\x{ 263A }         Same, but shows optional blanks inside and
                   adjoining the braces
\x1b         [2,8] restricted range hex char (example: ESC)
\N{name}     [3]   named Unicode character or character sequence
\N{U+263D}   [4,8] Unicode character (example: FIRST QUARTER MOON)
\c[          [5]   control char     (example: chr(27))
\o{23072}    [6,8] octal char       (example: SMILEY)
\033         [7,8] restricted range octal char  (example: ESC)
```

Обратите внимание, что любая управляющая последовательность, использующая фигурные скобки внутри интерполированных конструкций, может содержать необязательные пробелы (символы табуляции или пробела), примыкающие к фигурным скобкам и находящиеся внутри них, как показано выше на втором \x{ } примере.

[1]

Результатом является символ, указанный шестнадцатеричным числом между фигурными скобками. Смотрите "[8]" ниже, чтобы узнать, какой именно символ.

Пробелы (символы табуляции или пробела) могут отделять число от одной или обеих фигурных скобок.

В противном случае между фигурными скобками допустимы только шестнадцатеричные цифры. Если встречается недопустимый символ, будет выдано предупреждение, и недопустимый символ и все последующие символы (допустимые или недействительные) внутри фигурных скобок будут удалены.

Если между фигурными скобками нет допустимых цифр, сгенерированный символ является нулевым символом (\x{00}). Однако явная пустая фигурная скобка (\x{}) не вызовет предупреждения (в настоящее время).

[2]

Результатом является символ, указанный шестнадцатеричным числом в диапазоне от 0x00 до 0xFF. Смотрите "[8]" ниже, чтобы узнать, какой именно символ.

После \x допустимы только шестнадцатеричные цифры. Когда за \x следует менее двух допустимых цифр, все допустимые цифры будут дополнены нулем. Это означает, что \x7 будет интерпретироваться как \x07 , а одиночное значение "\x" будет интерпретироваться как \x00 . За исключением конца строки, наличие менее двух допустимых цифр приведет к предупреждению. Обратите внимание, что, хотя в

предупреждении говорится, что недопустимый символ игнорируется, он игнорируется только как часть escape и все равно будет использоваться в качестве последующего символа в строке. Например:

```
Original    Result    Warns?
"\x7"       "\x07"    no
"\x"        "\x00"    no
"\x7q"      "\x07q"   yes
"\xq"       "\x00q"   yes
```

[3]

Результатом является символ Юникода или последовательность символов, заданная с помощью *name*. Смотрите charnames .

[4]

`\N{U+`*hexadecimal number*`}` означает символ Юникода, кодовой точкой которого является *шестнадцатеричное число*.

[5]

Следующий символ `\c` сопоставляется с каким-либо другим символом, как показано в таблице:

```
Sequence   Value
  \c@      chr(0)
  \cA      chr(1)
  \ca      chr(1)
  \cB      chr(2)
  \cb      chr(2)
  ...
  \cZ      chr(26)
  \cz      chr(26)
  \c[      chr(27)
                   # See below for chr(28)
  \c]      chr(29)
  \c^      chr(30)
  \c_      chr(31)
  \c?      chr(127) # (on ASCII platforms; see below for link to
                   #  EBCDIC discussion)
```

Другими словами, это символ, кодовая точка которого имеет 64 xor'd с прописными буквами. `\c?` является DELETE на платформах ASCII, потому что `ord("?") ^ 64` равно 127, и `\c@` равно NULL, потому что порядок `"@"` равен 64, поэтому сам xor'ing 64 выдает 0.

Кроме того, `\c\`*X* выдает `chr(28) . "`*X*`"` результат для любого *X*, но не может быть в конце строки, потому что обратная косая черта будет проанализирована как экранирующая конечную кавычку.

На платформах ASCII результирующие символы из приведенного выше списка представляют собой полный набор элементов управления ASCII. Это не относится к платформам EBCDIC; см. "РАЗЛИЧИЯ ОПЕРАТОРОВ" в perlebcdic для полного обсуждения различий между ними для платформ ASCII и EBCDIC.

Использование любых других символов, следующих за `"c"` помимо перечисленных выше, не рекомендуется, и начиная с Perl версии 5.20, фактически разрешены только символы ASCII для печати, за вычетом левой фигурной скобки `"{"` . Что происходит для любого из разрешенных других символов, так это то, что значение выводится путем xor'ing с седьмым битом, который равен 64, и выдается предупреждение, если включено. Использование запрещенных символов приводит к фатальной ошибке.

Чтобы получить независимые от платформы элементы управления, вы можете использовать `\N{...}` .

[6]

Результатом является символ, указанный восьмеричным числом между фигурными скобками. Смотрите "[8]" ниже, чтобы узнать, какой именно символ.

Пробелы (символы табуляции или пробела) могут отделять число от одной или обеих фигурных скобок.

В противном случае, если встречается символ, который не является восьмеричной цифрой, выдается предупреждение, и значение основывается на восьмеричных цифрах перед ним, отбрасывая его и все последующие символы вплоть до закрывающей фигурной скобки. Это фатальная ошибка, если восьмеричные цифры вообще отсутствуют.

[7]

Результатом является символ, указанный трехзначным восьмеричным числом в диапазоне от 000 до 777 (но лучше не использовать выше 077, см. Следующий параграф). Смотрите "[8]" ниже для получения подробной информации о том, какой символ.

Some contexts allow 2 or even 1 digit, but any usage without exactly three digits, the first being a zero, may give unintended results. (For example, in a regular expression it may be confused with a backreference; see "Octal escapes" in perlrebackslash.) Starting in Perl 5.14, you may use `\o{}` instead, which avoids all these problems. Otherwise, it is best to use this construct only

for ordinals `\077` and below, remembering to pad to the left with zeros to make three digits. For larger ordinals, either use `\o{}`, or convert to something else, such as to hex and use `\N{U+}` (which is portable between platforms with different character sets) or `\x{}` instead.

[8]

Several constructs above specify a character by a number. That number gives the character's position in the character set encoding (indexed from 0). This is called synonymously its ordinal, code position, or code point. Perl works on platforms that have a native encoding currently of either ASCII/Latin1 or EBCDIC, each of which allow specification of 256 characters. In general, if the number is 255 (0xFF, 0377) or below, Perl interprets this in the platform's native encoding. If the number is 256 (0x100, 0400) or above, Perl interprets it as a Unicode code point and the result is the corresponding Unicode character. For example `\x{50}` and `\o{120}` both are the number 80 in decimal, which is less than 256, so the number is interpreted in the native character set encoding. In ASCII the character in the 80th position (indexed from 0) is the letter `"P"`, and in EBCDIC it is the ampersand symbol `"&"`. `\x{100}` and `\o{400}` are both 256 in decimal, so the number is interpreted as a Unicode code point no matter what the native encoding is. The name of the character in the 256th position (indexed by 0) in Unicode is `LATIN CAPITAL LETTER A WITH MACRON`.

An exception to the above rule is that `\N{U+`*hex number*`}` is always interpreted as a Unicode code point, so that `\N{U+0050}` is `"P"` even on EBCDIC platforms.

ПРИМЕЧАНИЕ: В отличие от C и других языков, в Perl нет `\v` управляющей последовательности для вертикальной вкладки (VT, которая равна 11 как в ASCII, так и в EBCDIC), но вы можете использовать `\N{VT}`, `\ck`, `\N{U+0b}` или `\x0b`. (`\v` имеет значение в шаблонах регулярных выражений в Perl, см. perlre.)

Следующие управляющие последовательности доступны в конструкциях с интерполяцией, но не в транслитерациях.

```
\l          lowercase next character only
\u          titlecase (not uppercase!) next character only
\L          lowercase all characters till \E or end of string
\U          uppercase all characters till \E or end of string
\F          foldcase all characters till \E or end of string
\Q          quote (disable) pattern metacharacters till \E or
            end of string
\E          end either case modification or quoted section
            (whichever was last seen)
```

Смотрите "quotemeta" в perlfunc для точного определения символов, которые заключаются в кавычки с помощью `\Q`.

`\L`, `\U`, `\F`, и `\Q` могут стекироваться, и в этом случае вам понадобится по одному `\E` для каждого. Например:

```
say "This \Qquoting \ubusiness \Uhere isn't quite\E done yet,\E is it?";
This quoting\ Business\ HERE\ ISN\'T\ QUITE\ done\ yet\, is it?
```

Если действует `use locale` форма, включающая `LC_CTYPE` (см. perllocale), карта регистров, используемая `\l`, `\L`, `\u`, и `\U`, берется из текущей локали. Если используется Unicode (например, `\N{}` или кодовые точки 0x100 или выше), то регистровая карта, используемая `\l`, `\L`, `\u`, и `\U` соответствует определению Unicode. Это означает, что при сопоставлении регистра для одного символа иногда может получаться последовательность из нескольких символов. В соответствии с `use locale`, `\F` выдает те же результаты, что и `\L` для всех локалей, кроме UTF-8, где вместо этого используется определение Unicode.

Все системы используют virtual `"\n"` для представления символа окончания строки, называемого "новой строкой". Не существует такого понятия, как неизменяемый физический символ новой строки. Это всего лишь иллюзия, что операционная система, драйверы устройств, библиотеки C и Perl сговорились сохранять. Не все системы читают `"\r"` как ASCII CR и `"\n"` как ASCII LF. Например, на старых компьютерах Mac (до macOS X) прошлых лет они менялись местами, а в системах без разделителя строк печать `"\n"` может не выдавать фактических данных. Как правило, используйте `"\n"`, когда вы подразумеваете "перевод строки" для вашей системы, но используйте буквенный ASCII-код, когда вам нужен точный символ. Например, большинство сетевых протоколов ожидают и предпочитают CR + LF (`"\015\012"` или `"\cM\cJ"`) для линейных терминаторов, и хотя они часто принимают just `"\012"`, они редко допускают just `"\015"`. Если у вас войдет в привычку использовать `"\n"` для работы в сети, вы можете однажды обжечься.

For constructs that do interpolate, variables beginning with "`$`" or "`@`" are interpolated. Subscripted variables such as `$a[3]` or `$href->{key}[0]` are also interpolated, as are array and hash slices. But method calls such as `$obj->meth` are not.

Интерполяция массива или фрагмента интерполирует элементы по порядку, разделенные значением `$"`, поэтому эквивалентно интерполяции `join $", @array`. Массивы "знаков препинания", такие как `@*` обычно интерполируются, только если имя заключено в фигурные скобки `@{*}`, но массивы `@_`, `@+` и `@-` интерполируются даже без фигурных скобок.

Для строк, заключенных в двойные кавычки, цитирование из `\Q` применяется после интерполяции и обработки экранирования.

```
"abc\Qfoo\tbar$s\Exyz"
```

эквивалентно

```
"abc" . quotemeta("foo\tbar$s") . "xyz"
```

Для шаблона операторов регулярных выражений (qr// , m// и s///) цитирование из \Q применяется после обработки интерполяции, но до обработки экранирования. Это позволяет шаблону совпадать буквально (за исключением $ и @). Например, следующие совпадения:

```
'\s\t' =~ /\Q\s\t/
```

Поскольку $ или @ запускает интерполяцию, вам нужно будет использовать что-то вроде /\Quser\E\@\Qhost/ , чтобы соответствовать им буквально.

Шаблоны могут интерпретироваться на дополнительном уровне как регулярные выражения. Это делается на втором этапе, после интерполяции переменных, так что регулярные выражения могут быть включены в шаблон из переменных. Если это не то, что вы хотите, используйте \Q для буквальной интерполяции переменной.

Помимо поведения, описанного выше, Perl не расширяет несколько уровней интерполяции. В частности, вопреки ожиданиям программистов оболочки, обратные кавычки *НЕ* интерполируют внутри двойных кавычек, а одинарные кавычки не препятствуют вычислению переменных при использовании внутри двойных кавычек.

Regexp

Вот операторы, подобные кавычкам, которые применяются к сопоставлению с шаблоном и связанным с ним действиям.

qr/*STRING*/msixpodualn

Этот оператор заключает в кавычки (и, возможно, компилирует) свою *СТРОКУ* как регулярное выражение. *СТРОКА* интерполируется так же, как *ШАБЛОН* в m/*PATTERN*/ . Если "'" используется в качестве разделителя, интерполяция переменных не выполняется. Возвращает значение Perl, которое может использоваться вместо соответствующего /*STRING*/msixpodualn выражения. Возвращаемое значение представляет собой нормализованную версию исходного шаблона. Он волшебным образом отличается от строки, содержащей те же символы: ref(qr/x/) возвращает "Регулярное выражение"; однако его разыменование четко не определено (в настоящее время вы получаете нормализованную версию исходного шаблона, но это может измениться).

Например,

```
$rex = qr/my.STRING/is;
print $rex;                  # prints (?si-xm:my.STRING)
s/$rex/foo/;
```

эквивалентно

```
s/my.STRING/foo/is;
```

Результат может быть использован в качестве подшаблона при сопоставлении:

```
$re = qr/$pattern/;
$string =~ /foo${re}bar/; # can be interpolated in other
                          # patterns
$string =~ $re;           # or used standalone
$string =~ /$re/;                 # or this way
```

Поскольку Perl может компилировать шаблон в момент выполнения qr() оператора, использование qr() может иметь преимущества в скорости в некоторых ситуациях, особенно если результат qr() используется автономно:

```
sub match {
    my $patterns = shift;
    my @compiled = map qr/$_/i, @$patterns;
    grep {
        my $success = 0;
        foreach my $pat (@compiled) {
            $success = 1, last if /$pat/;
        }
        $success;
    } @_;
}
```

Предварительная компиляция шаблона во внутреннее представление в момент qr() позволяет избежать необходимости перекомпиляции шаблона каждый раз при попытке сопоставления /$pat/ . (В Perl есть много других внутренних оптимизаций, но ни одна из них не сработала бы в приведенном выше примере, если бы мы не использовали qr() operator .)

Параметрами (заданными следующими модификаторами) являются:

```
m       Treat string as multiple lines.
s       Treat string as single line. (Make . match a newline)
i       Do case-insensitive pattern matching.
x   Use extended regular expressions; specifying two
    x's means \t and the SPACE character are ignored within
    square-bracketed character classes
p       When matching preserve a copy of the matched string so
    that ${^PREMATCH}, ${^MATCH}, ${^POSTMATCH} will be
    defined (ignored starting in v5.20 as these are always
    defined starting in that release)
o       Compile pattern only once.
a   ASCII-restrict: Use ASCII for \d, \s, \w and [[:posix:]]
    character classes; specifying two a's adds the further
    restriction that no ASCII character will match a
    non-ASCII one under /i.
l   Use the current run-time locale's rules.
u   Use Unicode rules.
d   Use Unicode or native charset, as in 5.12 and earlier.
n   Non-capture mode. Don't let () fill in $1, $2, etc...
```

Если предварительно скомпилированный шаблон встроен в более крупный шаблон, то эффект "msixpluadn" будет распространен соответствующим образом. Эффект, который имеет модификатор /o, не распространяется, поскольку ограничен теми шаблонами, которые явно его используют.

Модификаторы /a, /d, /l, и /u (добавлены в Perl 5.14) управляют правилами набора символов, но /a это единственный, который вы, вероятно, захотите указать явно; остальные три выбираются автоматически различными прагмами.

Смотрите perlre для получения дополнительной информации о допустимом синтаксисе для *СТРОКИ*, а также для подробного ознакомления с семантикой регулярных выражений. В частности, все модификаторы, за исключением в значительной степени устаревших, /o более подробно объясняются в "Модификаторах" в perlre. /o описано в следующем разделе.

m/*PATTERN*/msixpodualngc

/*PATTERN*/msixpodualngc

Выполняет поиск в строке соответствия шаблону и в скалярном контексте возвращает true, если это удается, false, если это не удается. Если с помощью оператора =~ or !~ не указана строка, выполняется поиск по $_ строке. (Строка, указанная с помощью =~ не обязательно должна быть значением lvalue - это может быть результатом вычисления выражения, но помните, что =~ привязки довольно жесткие.) Смотрите также perlre.

Параметры, как описано в qr// выше; кроме того, доступны следующие модификаторы процесса сопоставления:

```
g  Match globally, i.e., find all occurrences.
c  Do not reset search position on a failed match when /g is
   in effect.
```

Если "/" является разделителем, то инициал m необязателен. С помощью m вы можете использовать любую пару символов без пробелов (ASCII) в качестве разделителей. Это особенно полезно для сопоставления имен путей, содержащих "/", чтобы избежать LTS (синдрома наклоняющейся зубочистки). Если "?" является разделителем, то применяется правило совпадения только один раз, описанное в m?*PATTERN*? ниже. Если "'" (одинарная кавычка) является разделителем, интерполяция переменных в *ШАБЛОНЕ* не выполняется. При использовании символа-разделителя, допустимого в идентификаторе, после m требуется пробел.

ШАБЛОН может содержать переменные, которые будут интерполироваться каждый раз при выполнении поиска по шаблону, за исключением случаев, когда разделителем является одинарная кавычка. (Обратите внимание, что $(, $) и $| не интерполируются, поскольку они выглядят как тесты конца строки.) Perl не будет перекомпилировать шаблон, если не изменится содержащаяся в нем интерполированная переменная. Вы можете заставить Perl пропустить тест и никогда не перекомпилировать, добавив /o (что означает "один раз") после конечного разделителя. Когда-то Perl перекомпилировал регулярные выражения без необходимости, и этот модификатор был полезен, чтобы указать ему не делать этого в интересах скорости. Но сейчас единственными причинами для использования /o являются одна из:

1. Переменные имеют длину в тысячи символов, и вы знаете, что они не меняются, и вам нужно выжать последнюю каплю скорости, пропустив тестирование Perl для этого. (За это взимается штраф за обслуживание, поскольку упоминание /o означает обещание, что вы не будете изменять переменные в шаблоне. Если вы их измените, Perl даже не заметит.)
2. вы хотите, чтобы шаблон использовал начальные значения переменных независимо от того, изменяются они или нет. (Но есть более разумные способы добиться этого, чем использование /o.)
3. Если шаблон содержит встроенный код, например

   ```
   use re 'eval';
   $code = 'foo(?{ $x })';
   /$code/
   ```

   затем perl будет перекомпилировать каждый раз, даже если строка шаблона не изменилась, чтобы гарантировать, что текущее значение $x отображается каждый раз. Используйте /o, если хотите избежать этого.

Суть в том, что использование `/o` почти никогда не бывает хорошей идеей.

Пустой шаблон `//`

Если *ШАБЛОН* выдает пустую строку, вместо нее используется последнее *успешно* подобранное регулярное выражение. В этом случае учитываются только флаги `g` и `c` в пустом шаблоне; остальные флаги взяты из исходного шаблона. Если ранее совпадение не было успешным, это будет (автоматически) действовать как подлинный пустой шаблон (который всегда будет совпадать). Использование предоставленной пользователем строки в качестве шаблона сопряжено с риском того, что, если строка пуста, это вызовет поведение "последнее успешное совпадение", которое может привести к большой путанице. В таких случаях рекомендуется заменить `m/$pattern/` на `m/(?:$pattern)/`, чтобы избежать подобного поведения.

К последнему успешному шаблону можно получить доступ как к переменной через `${^LAST_SUCCESSFUL_PATTERN}`. Сопоставление с ним или пустым шаблоном должно иметь тот же эффект, за исключением того, что при отсутствии последнего успешного шаблона пустой шаблон будет автоматически соответствовать, тогда как использование переменной `${^LAST_SUCCESSFUL_PATTERN}` приведет к появлению неопределенных предупреждений (если предупреждения включены). Вы можете проверить, `defined(${^LAST_SUCCESSFUL_PATTERN})` есть ли "последнее успешное совпадение" в текущей области.

Обратите внимание, что Perl можно запутать, заставив думать, что `//` (пустое регулярное выражение) на самом деле `//` (оператор defined-or). Обычно Perl неплохо справляется с этим, но это могут вызвать некоторые патологические случаи, такие как `$x///` (это `($x) / (//)` или `$x // /`?) и `print $fh //` (`print $fh(//` или `print($fh //`?). Во всех этих примерах Perl предположит, что вы имели в виду defined-или. Если вы имели в виду пустое регулярное выражение, просто используйте круглые скобки или пробелы для устранения неоднозначности или даже префикс пустого регулярного выражения с `m` (так `//` становится `m//`).

в контексте списка

Если `/g` опция не используется, `m//` в контексте списка возвращается список, состоящий из подвыражений, которым соответствуют круглые скобки в шаблоне, то есть (`$1`, `$2`, `$3` ...) (Обратите внимание, что здесь также заданы `$1` и т.д.). Когда в шаблоне нет круглых скобок, возвращаемым значением является список `(1)` для успешного выполнения. С круглыми скобками или без них при сбое возвращается пустой список.

Примеры:

```
open(TTY, "+</dev/tty")
   || die "can't access /dev/tty: $!";

<TTY> =~ /^y/i && foo(); # do foo if desired

if (/Version: *([0-9.]*)/) { $version = $1; }

next if m#^/usr/spool/uucp#;

# poor man's grep
$arg = shift;
while (<>) {
   print if /$arg/o; # compile only once (no longer needed!)
}

if (($F1, $F2, $Etc) = ($foo =~ /^(\S+)\s+(\S+)\s*(.*)/))
```

Этот последний пример разбивается `$foo` на первые два слова и оставшуюся часть строки и присваивает этим трем полям значение `$F1`, `$F2` и `$Etc`. Условие имеет значение true, если были назначены какие-либо переменные; то есть, если шаблон совпал.

Модификатор `/g` задает глобальное сопоставление с шаблоном, то есть сопоставление как можно большего количества раз в строке. Поведение модификатора зависит от контекста. В контексте списка он возвращает список подстрок, которым соответствуют любые фиксирующие скобки в регулярном выражении. Если скобки отсутствуют, он возвращает список всех совпадающих строк, как если бы вокруг всего шаблона были круглые скобки.

В скалярном контексте каждое выполнение `m//g` находит следующее совпадение, возвращая true, если оно совпадает, и false, если дальнейшего совпадения нет. Позицию после последнего совпадения можно прочитать или установить с помощью `pos()` функции; см. "pos" в perlfunc. Неудачное совпадение обычно приводит к возврату позиции поиска в начало строки, но этого можно избежать, добавив `/c` модификатор (например, `m//gc`). Изменение целевой строки также сбрасывает позицию поиска.

`\G` *assertion*

Вы можете смешивать `m//g` совпадения с `m/\G.../g`, где `\G` это утверждение нулевой ширины, которое точно соответствует позиции, на которой остановилось предыдущее `m//g`, если таковое было. Без `/g` модификатора `\G` утверждение по-прежнему привязывается к `pos()`, как это было в начале операции (см. "pos" в perlfunc), но сопоставление, конечно, выполняется только один раз. Использование `\G` without `/g` для целевой строки, к которой ранее не применялось `/g` соответствие, аналогично использованию `\A` утверждения для сопоставления с началом строки. Обратите также внимание, что в настоящее время `\G` должным образом поддерживается только при привязке в самом начале шаблона.

Примеры:

```
    # list context
    ($one,$five,$fifteen) = (`uptime` =~ /(\d+\.\d+)/g);

    # scalar context
    local $/ = "";
    while ($paragraph = <>) {
        while ($paragraph =~ /\p{Ll}['")]*[.!?]+['")]*\s/g) {
            $sentences++;
        }
    }
    say $sentences;
```

Вот еще один способ проверить наличие предложений в абзаце:

```
my $sentence_rx = qr{
   (?: (?<= ^ ) | (?<= \s ) )  # after start-of-string or
                               # whitespace
   \p{Lu}                         # capital letter
    .*?                            # a bunch of anything
    (?<= \S )                      # that ends in non-
                                   # whitespace
    (?<! \b [DMS]r  )              # but isn't a common abbr.
    (?<! \b Mrs )
    (?<! \b Sra )
    (?<! \b St  )
    [.?!]                          # followed by a sentence
                                   # ender
    (?= $ | \s )                   # in front of end-of-string
                                   # or whitespace
}sx;
local $/ = "";
while (my $paragraph = <>) {
    say "NEW PARAGRAPH";
    my $count = 0;
    while ($paragraph =~ /($sentence_rx)/g) {
        printf "\tgot sentence %d: <%s>\n", ++$count, $1;
    }
}
```

Вот как использовать `m//gc` с `\G`:

```
$_ = "ppooqppqq";
while ($i++ < 2) {
    print "1: '";
    print $1 while /(o)/gc; print "', pos=", pos, "\n";
    print "2: '";
    print $1 if /\G(q)/gc;  print "', pos=", pos, "\n";
    print "3: '";
    print $1 while /(p)/gc; print "', pos=", pos, "\n";
}
print "Final: '$1', pos=",pos,"\n" if /\G(.)/;
```

Последний пример должен выводить:

```
1: 'oo', pos=4
2: 'q', pos=5
3: 'pp', pos=7
1: '', pos=7
2: 'q', pos=8
3: '', pos=8
Final: 'q', pos=8
```

Обратите внимание, что окончательное совпадение совпало `q` вместо `p`, что было бы при совпадении без `\G` привязки. Также обратите внимание, что окончательное совпадение не обновлялось `pos`. `pos` обновляется только при `/g` совпадении. Если финальное совпадение действительно совпало `p`, можно поспорить, что вы используете старую (до 5.6.0) версию Perl.

Полезная идиома для `lex` подобных сканеров - `/\G.../gc`. Вы можете комбинировать несколько подобных регулярных выражений для обработки строки по частям, выполняя разные действия в зависимости от того, какое регулярное выражение соответствует. Каждое регулярное выражение пытается соответствовать тому месту, где заканчивается предыдущее.

```
$_ = <<'EOL';
   $url = URI::URL->new( "http://example.com/" );
   die if $url eq "xXx";
EOL

LOOP: {
    print(" digits"),        redo LOOP if /\G\d+\b[,.;]?\s*/gc;
    print(" lowercase"),     redo LOOP
                                 if /\G\p{Ll}+\b[,.;]?\s*/gc;
    print(" UPPERCASE"),     redo LOOP
                                 if /\G\p{Lu}+\b[,.;]?\s*/gc;
    print(" Capitalized"),   redo LOOP
                                 if /\G\p{Lu}\p{Ll}+\b[,.;]?\s*/gc;
    print(" MiXeD"),         redo LOOP if /\G\pL+\b[,.;]?\s*/gc;
    print(" alphanumeric"),  redo LOOP
                                 if /\G[\p{Alpha}\pN]+\b[,.;]?\s*/gc;
    print(" line-noise"),    redo LOOP if /\G\W+/gc;
    print ". That's all!\n";
}
```

Вот результат (разделенный на несколько строк):

```
line-noise lowercase line-noise UPPERCASE line-noise UPPERCASE
line-noise lowercase line-noise lowercase line-noise lowercase
lowercase line-noise lowercase lowercase line-noise lowercase
lowercase line-noise MiXeD line-noise. That's all!
```

m?*PATTERN*?msixpodualngc

Это похоже на `m/PATTERN/` поиск, за исключением того, что он совпадает только один раз между вызовами `reset()` оператора. Это полезная оптимизация, например, когда вы хотите видеть только первое вхождение чего-либо в каждом файле из набора файлов. Сбрасываются только `m??` шаблоны, локальные для текущего пакета.

```
    while (<>) {
        if (m?^$?) {
                                # blank line between header and body
        }
    } continue {
        reset if eof;           # clear m?? status for next file
    }
```

Другой пример заменил первую найденную кодировку "latin1" на "utf8" в pod-файле:

```
s//utf8/ if m? ^ =encoding \h+ \K latin1 ?x;
```

Поведение при однократном совпадении контролируется разделителем совпадения, равным `?` ; с любым другим разделителем это обычный `m//` оператор.

В прошлом начальный параметр `m` в m?*PATTERN*? был необязательным, но его отсутствие привело бы к появлению предупреждения об устаревании. Начиная с версии 5.22.0, его отсутствие приводит к синтаксической ошибке. Если вы встретите эту конструкцию в более старом коде, вы можете просто добавить `m` .

s/*PATTERN*/*REPLACEMENT*/msixpodualngcer

Выполняет поиск шаблона в строке и, если он найден, заменяет этот шаблон текстом замены и возвращает количество произведенных замен. В противном случае он возвращает false (значение, которое одновременно является пустой строкой ("") и числовым нулем (0), как описано в "Операторы отношения").

Если используется параметр `/r` (неразрушающий), то он выполняет подстановку для копии строки и вместо возврата количества подстановок возвращает копию независимо от того, произошла подстановка или нет. Исходная строка никогда не изменяется при использовании `/r` . Копия всегда будет простой строкой, даже если входными данными является объект или связанная переменная.

Если с помощью оператора `=~` or `!~` не указана строка, выполняется поиск и изменение `$_` переменной. Если не используется параметр `/r` , указанная строка должна быть скалярной переменной, элементом массива, элементом хэша или присвоением одному из них; то есть, своего рода скалярным значением lvalue.

Если выбран разделитель в виде одинарных кавычек, интерполяция переменных ни в *ШАБЛОНЕ*, ни в *ЗАМЕНЕ* не выполняется. В противном случае, если *ШАБЛОН* содержит `$` , который выглядит как переменная, а не как тест конца строки, переменная будет интерполирована в шаблон во время выполнения. Если вы хотите, чтобы шаблон компилировался только один раз при первой интерполяции переменной, используйте опцию `/o` . Если шаблон выдает пустую строку, вместо нее используется последнее успешно выполненное регулярное выражение. Смотрите perlre для получения дополнительных разъяснений по этому поводу.

Параметры такие же, как в `m//` с добавлением следующих параметров, специфичных для замены:

```
e       Evaluate the right side as an expression.
ee  Evaluate the right side as a string then eval the
    result.
r   Return substitution and leave the original string
    untouched.
```

Любой разделитель, не содержащий пробелов, может заменять косую черту. Добавляйте пробел после `s` при использовании символа, разрешенного в идентификаторах. Если используются одинарные кавычки, интерпретация заменяющей строки не выполняется (однако модификатор `/e` переопределяет это). Обратите внимание, что Perl обрабатывает обратные метки как обычные разделители; заменяющий текст не оценивается как команда. Если *ШАБЛОН* ограничен кавычками, заключенными в квадратные скобки, у *ЗАМЕНЫ* есть своя пара кавычек, которые могут быть, а могут и не быть кавычками, заключенными в квадратные скобки, например, `s(foo)(bar)` или `s<foo>/bar/`. A `/e` приведет к тому, что заменяющая часть будет обрабатываться как полноценное выражение Perl и вычисляться прямо здесь и сейчас. Однако синтаксис проверяется во время компиляции. Второй `e` модификатор приведет к тому, что заменяющая часть будет `eval` отредактирована перед запуском как выражение Perl.

Примеры:

```
    s/\bgreen\b/mauve/g;               # don't change wintergreen

    $path =~ s|/usr/bin|/usr/local/bin|;

    s/Login: $foo/Login: $bar/; # run-time pattern

    ($foo = $bar) =~ s/this/that/;        # copy first, then
                                          # change
    ($foo = "$bar") =~ s/this/that/;      # convert to string,
                                          # copy, then change
    $foo = $bar =~ s/this/that/r; # Same as above using /r
    $foo = $bar =~ s/this/that/r
                =~ s/that/the other/r;    # Chained substitutes
                                          # using /r
    @foo = map { s/this/that/r } @bar     # /r is very useful in
                                          # maps

    $count = ($paragraph =~ s/Mister\b/Mr./g);  # get change-cnt

    $_ = 'abc123xyz';
    s/\d+/$&*2/e;                 # yields 'abc246xyz'
    s/\d+/sprintf("%5d",$&)/e;    # yields 'abc  246xyz'
    s/\w/$& x 2/eg;               # yields 'aabbcc  224466xxyyzz'

    s/%(.)/$percent{$1}/g;        # change percent escapes; no /e
    s/%(.)/$percent{$1} || $&/ge;       # expr now, so /e
    s/^=(\w+)/pod($1)/ge;         # use function call

    $_ = 'abc123xyz';
    $x = s/abc/def/r;            # $x is 'def123xyz' and
                                 # $_ remains 'abc123xyz'.

    # expand variables in $_, but dynamics only, using
    # symbolic dereferencing
    s/\$(\w+)/${$1}/g;

    # Add one to the value of any numbers in the string
    s/(\d+)/1 + $1/eg;

    # Titlecase words in the last 30 characters only (presuming
    # that the substring doesn't start in the middle of a word)
    substr($str, -30) =~ s/\b(\p{Alpha})(\p{Alpha}*)\b/\u$1\L$2/g;

    # This will expand any embedded scalar variable
    # (including lexicals) in $_ : First $1 is interpolated
    # to the variable name, and then evaluated
    s/(\$\w+)/$1/eeg;

    # Delete (most) C comments.
    $program =~ s {
        /\*    # Match the opening delimiter.
       .*?     # Match a minimal number of characters.
       \*/     # Match the closing delimiter.
    } []gsx;

    s/^\s*(.*?)\s*$/$1/;   # trim whitespace in $_,
                           # expensively

    for ($variable) {      # trim whitespace in $variable,
                           # cheap
        s/^\s+//;
        s/\s+$//;
    }

    s/([^ ]*) *([^ ]*)/$2 $1/;  # reverse 1st two fields

    $foo !~ s/A/a/g;    # Lowercase all A's in $foo; return
                        # 0 if any were found and changed;
                        # otherwise return 1
```

Note the use of `$` instead of `\` in the last example. Unlike **sed**, we use the \<*digit*> form only in the left hand side. Anywhere else it's $<*digit*>.

Occasionally, you can't use just a `/g` to get all the changes to occur that you might want. Here are two common cases:

```
# put commas in the right places in an integer
1 while s/(\d)(\d\d\d)(?!\d)/$1,$2/g;

# expand tabs to 8-column spacing
1 while s/\t+/' ' x (length($&)*8 - length($`)%8)/e;
```

Хотя флаг `s///` принимает `/c`, он не имеет никакого эффекта, кроме выдачи предупреждения, если предупреждения включены.

Операторы, подобные кавычкам

q/*STRING*/

'*STRING*'

Буквальная строка, заключенная в одинарные кавычки. Обратная косая черта представляет обратную косую черту, если за ней не следует разделитель или другая обратная косая черта, и в этом случае разделитель или обратная косая черта интерполируются.

```
$foo = q!I said, "You said, 'She said it.'"!;
$bar = q('This is it.');
$baz = '\n';             # a two-character string
```

qq/*STRING*/

"*STRING*"

Строка, заключенная в двойные кавычки и интерполированная.

```
    $_ .= qq
    (*** The previous line contains the naughty word "$1".\n)
                if /\b(tcl|java|python)\b/i;      # :-)
    $baz = "\n";         # a one-character string
```

qx/*STRING*/

`*STRING*`

Строка, которая (возможно) интерполируется и затем выполняется как системная команда через */bin/sh* или ее эквивалент, если требуется. Подстановочные знаки, каналы и перенаправления оболочки будут соблюдены. Аналогично `system`, если строка не содержит метасимволов оболочки, то она будет выполняться напрямую. Возвращается собранный стандартный вывод команды; стандартная ошибка не изменяется. В скалярном контексте он возвращается в виде одиночной (потенциально многострочной) строки или `undef` если оболочку (или команду) запустить не удалось. В контексте списка возвращает список строк (однако вы определили строки с помощью `$/` или `$INPUT_RECORD_SEPARATOR`) или пустой список, если оболочку (или команду) запустить не удалось.

Поскольку обратные ссылки не влияют на стандартную ошибку, используйте синтаксис файлового дескриптора оболочки (при условии, что оболочка поддерживает это), если вы хотите решить эту проблему. Чтобы записать STDERR и STDOUT команды вместе.:

```
$output = `cmd 2>&1`;
```

Чтобы зафиксировать стандартный вывод команды, но удалить его стандартный код:

```
$output = `cmd 2>/dev/null`;
```

Чтобы зафиксировать STDERR команды, но отбросить ее стандартный вывод (здесь важен порядок):

```
$output = `cmd 2>&1 1>/dev/null`;
```

Для замены стандартного вывода команды и STDERR, чтобы захватить STDERR, но оставить его стандартный вывод, чтобы вывести старый STDERR:

```
$output = `cmd 3>&1 1>&2 2>&3 3>&-`;
```

Чтобы прочитать стандартный вывод команды и ее STDERR отдельно, проще всего перенаправить их отдельно в файлы, а затем прочитать из этих файлов по завершении работы программы:

```
system("program args 1>program.stdout 2>program.stderr");
```

Дескриптор файла STDIN, используемый командой, унаследован от STDIN Perl. Например:

```
open(SPLAT, "stuff")   || die "can't open stuff: $!";
open(STDIN, "<&SPLAT") || die "can't dupe SPLAT: $!";
print STDOUT `sort`;
```

напечатает отсортированное содержимое файла с именем *"stuff"*.

Использование одинарных кавычек в качестве разделителя защищает команду от интерполяции Perl в двойные кавычки, передавая ее в командную оболочку вместо этого:

```
$perl_info  = qx(ps $$);            # that's Perl's $$
$shell_info = qx'ps $$';            # that's the new shell's $$
```

Способ вычисления этой строки полностью зависит от интерпретатора команд в вашей системе. На большинстве платформ вам придется защищать метасимволы оболочки, если вы хотите, чтобы они обрабатывались буквально. На практике это сделать сложно, поскольку неясно, как экранировать какие символы. Смотрите в perlsec чистый и безопасный пример руководства `fork()` и `exec()` как безопасно эмулировать обратные ссылки.

На некоторых платформах (особенно DOS-подобных) оболочка может быть не способна обрабатывать многострочные команды, поэтому перевод строк в строку может не дать вам желаемого. Возможно, вы сможете вычислять несколько команд в одной строке, разделяя их символом-разделителем команд, если ваша оболочка поддерживает это (например, `;` во многих оболочках Unix и `&` в оболочке Windows NT `cmd`).

Perl will attempt to flush all files opened for output before starting the child process, but this may not be supported on some platforms (see perlport). To be safe, you may need to set `$|` (`$AUTOFLUSH` in English) or call the `autoflush()` method of `IO::Handle` on any open handles.

Beware that some command shells may place restrictions on the length of the command line. You must ensure your strings don't exceed this limit after any necessary interpolations. See the platform-specific release notes for more details about your particular environment.

Использование этого оператора может привести к созданию программ, которые трудно переносить, поскольку вызываемые команды оболочки различаются в разных системах и фактически могут отсутствовать вообще. В качестве одного из примеров, `type` команда в командной оболочке POSIX сильно отличается от `type` команды в DOS. Это не значит, что вы должны изо всех сил избегать обратных ссылок, когда они являются правильным способом добиться чего-либо. Perl был создан как язык склеивания, и одна из вещей, которые он склеивает, - это команды. Просто поймите, во что вы ввязываетесь.

Например, `system` кнопки возврата помещают код выхода дочернего процесса в `$?`. Если вы хотите вручную проверить сбой, вы можете проверить все возможные режимы сбоя, выполнив проверку `$?` следующим образом:

```
if ($? == -1) {
     print "failed to execute: $!\n";
}
elsif ($? & 127) {
     printf "child died with signal %d, %s coredump\n",
          ($? & 127),  ($? & 128) ? 'with' : 'without';
}
else {
     printf "child exited with value %d\n", $? >> 8;
}
```

Используйте прагму open для управления уровнями ввода-вывода, используемыми при чтении выходных данных команды, например:

```
use open IN => ":encoding(UTF-8)";
my $x = `cmd-producing-utf-8`;
```

`qx//` также может вызываться как функция с "readpipe" в perlfunc.

Смотрите "Операторы ввода-вывода" для более подробного обсуждения.

qw/*STRING*/

Вычисляется как список слов, извлеченных из *СТРОКИ*, используя встроенные пробелы в качестве разделителей слов. Это можно понимать как примерно эквивалентное:

```
split(" ", q/STRING/);
```

отличия в том, что он разбивается только на пробелы ASCII, генерирует реальный список во время компиляции, а в скалярном контексте возвращает последний элемент в списке. Итак, это выражение:

```
qw(foo bar baz)
```

семантически эквивалентен списку:

```
"foo", "bar", "baz"
```

Некоторые часто встречающиеся примеры:

```
use POSIX qw( setlocale localeconv )
@EXPORT = qw( foo bar baz );
```

Распространенной ошибкой является попытка разделять слова запятыми или помещать комментарии в многострочную `qw` строку. По этой причине `use warnings` прагма и переключатель **-w** (то есть `$^W` переменная) выдают предупреждения, если *СТРОКА* содержит символ `","` или `"#"`.

`tr`/*SEARCHLIST*/*REPLACEMENTLIST*/`cdsr`

`y`/*SEARCHLIST*/*REPLACEMENTLIST*/`cdsr`

Транслитерирует все вхождения символов, найденных (или не найденных, если указан модификатор `/c`) в списке поиска, соответствующим по положению символом в списке замены, возможно, удаляя некоторые, в зависимости от указанных модификаторов. Возвращает количество замененных или удаленных символов. Если строка не указана с помощью оператора `=~` или `!~`, то `$_` строка транслитерируется.

Для приверженцев **sed**, `y` предоставляется как синоним `tr`.

Если присутствует опция `/r` (неразрушающий), создается новая копия строки и ее символы транслитерируются, и эта копия возвращается независимо от того, была ли она изменена или нет: исходная строка всегда остается неизменной. Новая копия всегда представляет собой простую строку, даже если входной строкой является объект или связанная переменная.

Если не используется параметр `/r`, строка, указанная с помощью `=~`, должна быть скалярной переменной, элементом массива, элементом хэша или присвоением одному из них; другими словами, значением lvalue.

Символы, разделяющие *СПИСОК ПОИСКА* и *СПИСОК ЗАМЕНЫ*, могут быть любыми печатаемыми символами, а не только косыми чертами. Если они заключены в одинарные кавычки (`tr'`*SEARCHLIST*`'`*REPLACEMENTLIST*`'`), единственной интерполяцией является удаление `\` из пар `\\`; таким образом, дефисы интерпретируются буквально, а не указывают диапазон символов.

В противном случае диапазон символов может быть указан через дефис, поэтому `tr/A-J/0-9/` выполняется та же замена, что и `tr/ACEGIBDFHJ/0246813579/`.

Если *СПИСОК ПОИСКА* ограничен кавычками, заключенными в квадратные скобки, то в *СПИСКЕ ЗАМЕНЫ* должна быть своя пара кавычек, которые могут быть, а могут и не быть кавычками, заключенными в квадратные скобки; например, `tr(aeiouy)(yuoiea)` или `tr[+\-*/]"ABCD"`. Этот последний пример показывает способ визуально прояснить происходящее для людей, которые больше знакомы с шаблонами регулярных выражений, чем с `tr`, и которые могут подумать, что разделители косой черты подразумевают, что `tr` это больше похоже на шаблон регулярных выражений, чем есть на самом деле. (Другим вариантом может быть использование `tr[...][...]`.)

`tr` не полностью похож на классы символов, заключенные в квадратные скобки, просто (значительно) больше похож на них, чем на полные шаблоны. Например, символы, появляющиеся более одного раза в любом списке, ведут себя здесь иначе, чем в шаблонах, и `tr` списки не допускают классы символов с обратной косой чертой, такие как `\d` или `\pL`, а также интерполяцию переменных, поэтому `"$"` и `"@"` всегда обрабатываются как литералы.

Разрешенными элементами являются литералы плюс `\'` (что означает одинарную кавычку). Если разделители не заключены в одинарные кавычки, также разрешены любые управляющие последовательности, принятые в строках, заключаемых в двойные кавычки. Подробная информация о escape-последовательности приведена в таблице в начале этого раздела.

A hyphen at the beginning or end, or preceded by a backslash is also always considered a literal. Precede a delimiter character with a backslash to allow it.

The `tr` operator is not equivalent to the tr(1) utility. `tr[a-z][A-Z]` will uppercase the 26 letters "a" through "z", but for case changing not confined to ASCII, use `lc`, `uc`, `lcfirst`, `ucfirst` (all documented in perlfunc), or the substitution operator `s/`*PATTERN*`/`*REPLACEMENT*`/` (with `\U`, `\u`, `\L`, and `\l` string-interpolation escapes in the *REPLACEMENT* portion).

Most ranges are unportable between character sets, but certain ones signal Perl to do special handling to make them portable. There are two classes of portable ranges. The first are any subsets of the ranges `A-Z`, `a-z`, and `0-9`, when expressed as literal characters.

```
tr/h-k/H-K/
```

заглавные буквы `"h"`, `"i"`, `"j"`, и `"k"` и ничего больше, независимо от набора символов платформы. Напротив, все

```
tr/\x68-\x6B/\x48-\x4B/
tr/h-\x6B/H-\x4B/
tr/\x68-k/\x48-K/
```

используйте те же заглавные буквы, что и в предыдущем примере, при запуске на платформах ASCII, но что-то совершенно другое на платформах EBCDIC.

Второй класс переносимых диапазонов вызывается, когда одна или обе конечные точки диапазона выражаются как `\N{...}`

```
$string =~ tr/\N{U+20}-\N{U+7E}//d;
```

удаляет из `$string` всех символов платформы, эквивалентных любому из Unicode U+0020, U+0021, ... U+007D, U+007E. Это переносимый диапазон, который имеет одинаковый эффект на каждой платформе, на которой он запущен. В этом примере это символы для печати в формате ASCII. Итак, после запуска, `$string` содержит только элементы управления и символы, которые не имеют эквивалентов ASCII.

Но даже для переносимых диапазонов обычно не очевидно, что включено, без необходимости смотреть в руководстве. Разумный принцип заключается в использовании только диапазонов, которые начинаются и заканчиваются либо алфавитами ASCII с одинаковым регистром (`b-e` , `B-E`), либо цифрами (`1-4`). Все остальное неясно (и непереносимо, если не используется `\N{...}`). Если сомневаетесь, укажите наборы символов полностью.

Опции:

```
    c   Complement the SEARCHLIST.
    d   Delete found but unreplaced characters.
    r   Return the modified string and leave the original string
        untouched.
    s   Squash duplicate replaced characters.
```

Если указан `/d` модификатор, все символы, указанные в *СПИСКЕ ПОИСКА*, которые не найдены в *СПИСКЕ ЗАМЕНЫ*, удаляются. (Обратите внимание, что это немного более гибко, чем поведение некоторых **tr** программ, которые удаляют все, что находят в *СПИСКЕ ПОИСКА*, точка.)

If the `/s` modifier is specified, sequences of characters, all in a row, that were transliterated to the same character are squashed down to a single instance of that character.

```
my $a = "aaabbbca";
$a =~ tr/ab/dd/s;      # $a now is "dcd"
```

If the `/d` modifier is used, the *REPLACEMENTLIST* is always interpreted exactly as specified. Otherwise, if the *REPLACEMENTLIST* is shorter than the *SEARCHLIST*, the final character, if any, is replicated until it is long enough. There won't be a final character if and only if the *REPLACEMENTLIST* is empty, in which case *REPLACEMENTLIST* is copied from *SEARCHLIST*. An empty *REPLACEMENTLIST* is useful for counting characters in a class, or for squashing character sequences in a class.

```
tr/abcd//           tr/abcd/abcd/
tr/abcd/AB/         tr/abcd/ABBB/
tr/abcd//d          s/[abcd]//g
tr/abcd/AB/d        (tr/ab/AB/ + s/[cd]//g)  - but run together
```

Если указан `/c` модификатор, транслитерируемые символы ОТСУТСТВУЮТ в *СПИСКЕ ПОИСКА*, то есть он дополняется. Если также указаны `/d` и/или `/s` , они применяются к дополняемому *СПИСКУ ПОИСКА*. Напомним, что если *REPLACEMENTLIST* пуст (за исключением под `/d`), вместо этого используется копия *SEARCHLIST*. Эта копия сделана после дополнения в разделе `/c` . После дополнения *СПИСОК ПОИСКА* сортируется по порядку кодовых точек, и любой *СПИСОК ЗАМЕНЫ* применяется к этому отсортированному результату. Это означает, что в соответствии с `/c` порядок символов, указанный в *СПИСКЕ ПОИСКА*, не имеет значения. Это может привести к разным результатам в системах EBCDIC, если *СПИСОК ЗАМЕН* содержит более одного символа, следовательно, его обычно нельзя переносить `/c` с таким *СПИСКОМ ЗАМЕН*.

Другой способ описания операции заключается в следующем: Если `/c` указан, *СПИСОК ПОИСКА* сортируется по порядку кодовых точек, а затем дополняется. Если *СПИСОК ЗАМЕН* пуст и `/d` не указан, *СПИСОК ЗАМЕН* заменяется копией *СПИСКА ПОИСКА* (как изменено в разделе `/c`), и эти потенциально измененные списки используются в качестве основы для дальнейшего. Любой символ в целевой строке, которого нет в *СПИСКЕ ПОИСКА*, передается без изменений. Каждый второй символ в целевой строке заменяется символом в *СПИСКЕ ЗАМЕНЫ*, который позиционно соответствует своему партнеру в *СПИСКЕ ПОИСКА*, за исключением того, что в разделе `/s` второй и последующие символы выдавливаются в последовательности символов в строке, которые все преобразуются в один и тот же символ. Если *СПИСОК ПОИСКА* длиннее, чем *СПИСОК ЗАМЕНЫ*, символы в целевой строке, которые соответствуют символу в *СПИСКЕ ПОИСКА*, который не имеет соответствия в *СПИСКЕ ЗАМЕНЫ*, либо удаляются из целевой строки, если `/d` указан; или заменяются последним символом в *СПИСКЕ ЗАМЕНЫ*, если `/d` не указан.

Несколько примеров:

```
$ARGV[1] =~ tr/A-Z/a-z/;   # canonicalize to lower case ASCII

$cnt = tr/*/*/;             # count the stars in $_
$cnt = tr/*//;              # same thing

$cnt = $sky =~ tr/*/*/;     # count the stars in $sky
$cnt = $sky =~ tr/*//;      # same thing

$cnt = $sky =~ tr/*//c;     # count all the non-stars in $sky
$cnt = $sky =~ tr/*/*/c;    # same, but transliterate each non-star
                             # into a star, leaving the already-stars
                             # alone.  Afterwards, everything in $sky
                             # is a star.

$cnt = tr/0-9//;            # count the ASCII digits in $_

tr/a-zA-Z//s;                # bookkeeper -> bokeper
tr/o/o/s;                    # bookkeeper -> bokkeeper
tr/oe/oe/s;                  # bookkeeper -> bokkeper
tr/oe//s;                    # bookkeeper -> bokkeper
tr/oe/o/s;                   # bookkeeper -> bokkopor

($HOST = $host) =~ tr/a-z/A-Z/;
 $HOST = $host  =~ tr/a-z/A-Z/r; # same thing

$HOST = $host =~ tr/a-z/A-Z/r   # chained with s///r
              =~ s/:/ -p/r;

tr/a-zA-Z/ /cs;                 # change non-alphas to single space

@stripped = map tr/a-zA-Z/ /csr, @original;
                            # /r with map

tr [\200-\377]
   [\000-\177];                  # wickedly delete 8th bit

$foo !~ tr/A/a/    # transliterate all the A's in $foo to 'a',
                    # return 0 if any were found and changed.
                    # Otherwise return 1
```

Если для символа задано несколько транслитераций, используется только первая:

```
tr/AAA/XYZ/
```

будет транслитерировать любые A в X.

Поскольку таблица транслитерации создается во время компиляции, ни *СПИСОК ПОИСКА*, ни *СПИСОК ЗАМЕНЫ* не подвергаются интерполяции двойными кавычками. Это означает, что если вы хотите использовать переменные, вы должны использовать `eval()`:

```
eval "tr/$oldlist/$newlist/";
die $@ if $@;

eval "tr/$oldlist/$newlist/, 1" or die $@;
```

<<EOF

Ориентированная на строку форма цитирования основана на синтаксисе оболочки "here-document". После а `<<` вы указываете строку, завершающую материал, заключенный в кавычки, и все строки, следующие за текущей строкой вплоть до завершающей строки, являются значением элемента.

Добавление в завершающую строку префикса `~` указывает, что вы хотите использовать "С отступом здесь-docs" (см. Ниже).

Завершающей строкой может быть либо идентификатор (слово), либо некоторый текст, заключенный в кавычки. Идентификатор без кавычек работает подобно двойным кавычкам. Между `<<` и идентификатором может не быть пробела, если только идентификатор явно не заключен в кавычки. Завершающая строка должна отображаться сама по себе (без кавычек и без окружающих пробелов) в завершающей строке.

Если конечная строка заключена в кавычки, тип используемых кавычек определяет обработку текста.

Двойные кавычки

Двойные кавычки указывают, что текст будет интерполирован с использованием точно таких же правил, что и обычные строки, заключенные в двойные кавычки.

```
    print <<EOF;
The price is $Price.
EOF

    print << "EOF"; # same as above
The price is $Price.
EOF
```

Одинарные кавычки

Одинарные кавычки указывают на то, что текст должен обрабатываться буквально, без интерполяции его содержимого. Это похоже на строки, заключенные в одинарные кавычки, за исключением того, что обратная косая черта не имеет особого значения, поскольку `\\` обрабатывается как две обратные косые черты, а не одна, как в любой другой конструкции, заключающей в кавычки.

Как и в командной оболочке, простое слово с обратной косой чертой, следующее за `<<`, означает то же самое, что и строка, заключенная в одинарные кавычки:

```
    $cost = <<'VISTA';  # hasta la ...
That'll be $10 please, ma'am.
VISTA

    $cost = <<\VISTA;   # Same thing!
That'll be $10 please, ma'am.
VISTA
```

Это единственная форма цитирования в perl, при которой нет необходимости беспокоиться об экранировании содержимого, что генераторы кода могут эффективно использовать.

Обратные ссылки

Содержимое документа here обрабатывается точно так же, как это было бы, если бы строка была встроена в обратные ссылки. Таким образом, содержимое интерполируется так, как если бы оно было заключено в двойные кавычки, а затем выполняется через оболочку с возвращением результатов выполнения.

```
    print << `EOC`; # execute command and get results
echo hi there
EOC
```

С отступом здесь-docs

Модификатор here-doc `~` позволяет вам делать отступы в ваших документах here-docs, чтобы сделать код более читаемым:

```
if ($some_var) {
  print <<~EOF;
    This is a here-doc
    EOF
}
```

Это будет напечатано...

```
This is a here-doc
```

... без начальных пробелов.

Строка, содержащая разделитель, который отмечает конец документа here-doc, определяет шаблон отступа для всего документа. Компиляция прерывается, если какая-либо непустая строка внутри here-doc не начинается с точного отступа конечной строки. (Пустая строка состоит из одного символа "\ n".) Например, предположим, что конечная строка начинается с символа табуляции, за которым следуют 4 пробела. Каждая непустая строка в here-doc должна начинаться с символа табуляции, за которым следуют 4 пробела. Они удаляются из каждой строки, и любой оставшийся пробел в начале строки служит отступом для этой строки. В настоящее время только символы ТАБУЛЯЦИИ и ПРОБЕЛА обрабатываются как пробелы для этой цели. Табуляции и пробелы могут быть смешаны, но совпадают в точности; вкладки остаются вкладками и не раскрываются.

Дополнительный начальный пробел (помимо того, что предшествовало разделителю) будет сохранен:

```
print <<~EOF;
  This text is not indented
    This text is indented with two spaces
                This text is indented with two tabs
  EOF
```

Наконец, модификатор может использоваться со всеми упомянутыми выше формами:

```
<<~\EOF;
<<~'EOF'
<<~"EOF"
<<~`EOF`
```

Между разделителями `~` и в кавычках могут использоваться пробелы:

```
<<~ 'EOF'; # ... "EOF", `EOF`
```

Здесь можно расположить несколько документов подряд:

```
   print <<"foo", <<"bar"; # you can stack them
I said foo.
foo
I said bar.
bar

   myfunc(<< "THIS", 23, <<'THAT');
Here's a line
or two.
THIS
and here's another.
THAT
```

Только не забывайте, что вы должны поставить точку с запятой в конце, чтобы завершить инструкцию, поскольку Perl не знает, что вы не собираетесь пытаться это сделать:

```
   print <<ABC
179231
ABC
   + 20;
```

Если вы хотите удалить разделитель строк из ваших here-документов, используйте `chomp()`.

```
chomp($string = <<'END');
This is a string.
END
```

Если вы хотите, чтобы ваши here-docs имели отступы вместе с остальным кодом, используйте `<<~FOO` конструкцию, описанную в разделе "Here-docs с отступами":

```
$quote = <<~'FINIS';
   The Road goes ever on and on,
   down from the door where it began.
   FINIS
```

Если вы используете here-doc в конструкции с разделителями, такой как в `s///eg`, материал, заключенный в кавычки, все равно должен находиться в строке после `<<FOO` маркера, что означает, что он может находиться внутри конструкции с разделителями:

```
s/this/<<E . 'that'
the other
E
 . 'more '/eg;
```

Это работает таким образом с Perl 5.18. Исторически это было непоследовательно, и вам пришлось бы написать

```
s/this/<<E . 'that'
 . 'more '/eg;
the other
E
```

вне строковых оценок.

Кроме того, правила кавычек для идентификатора конца строки не связаны с правилами кавычек Perl. `q()`, `qq()` и т.п. Не поддерживаются вместо `''` и `""`, и единственная интерполяция - это обратная косая черта символа, заключенного в кавычки:

```
print << "abc\"def";
testing...
abc"def
```

Наконец, строки, заключенные в кавычки, не могут занимать несколько строк. Общее правило заключается в том, что идентификатор должен быть строковым литералом. Придерживайтесь этого, и вы будете в безопасности.

Кровавые подробности синтаксического анализа конструкций в кавычках

При представлении чего-либо, что может иметь несколько различных интерпретаций, Perl использует принцип **DWIM** (это "Делай то, что я имею в виду"), чтобы выбрать наиболее вероятную интерпретацию. Эта стратегия настолько успешна, что программисты Perl часто не подозревают о двойственности того, что они пишут. Но время от времени представления Perl существенно отличаются от того, что честно имел в виду автор.

В этом разделе мы надеемся прояснить, как Perl обрабатывает конструкции, заключенные в кавычки. Хотя наиболее распространенной причиной для изучения этого является необходимость разобраться в запутанных регулярных выражениях, поскольку начальные шаги синтаксического анализа одинаковы для всех операторов, заключающих в кавычки, все они обсуждаются вместе.

The most important Perl parsing rule is the first one discussed below: when processing a quoted construct, Perl first finds the end of that construct, then interprets its contents. If you understand this rule, you may skip the rest of this section on the first reading. The other rules are likely to contradict the user's expectations much less frequently than this first one.

Some passes discussed below are performed concurrently, but because their results are the same, we consider them individually. For different quoting constructs, Perl performs different numbers of passes, from one to four, but these passes are always performed in the same order.

Finding the end

The first pass is finding the end of the quoted construct. This results in saving to a safe location a copy of the text (between the starting and ending delimiters), normalized as necessary to avoid needing to know what the original delimiters were.

Если конструкция является here-doc, конечным разделителем является строка, которая имеет завершающую строку в качестве содержимого. Следовательно, `<<EOF` завершается на `EOF`, за которым сразу следует `"\n"` и начиная с первого столбца завершающей строки. При поиске конечной строки here-doc ничего не пропускается. Другими словами, строки после синтаксиса here-doc сравниваются с конечной строкой построчно.

Для конструкций, за исключением here-docs, в качестве начальных и конечных разделителей используются одиночные символы. Если начальным разделителем является открывающий знак препинания (то есть `(`, `[`, `{` или `<`), конечным разделителем является соответствующий закрывающий знак препинания (то есть `)`, `]`, `}` или `>`). Если начальным разделителем является непарный символ, такой как `/` или завершающий знак препинания, конечный разделитель совпадает с начальным разделителем. Следовательно, а `/` завершает `qq//` конструкцию, в то время как а `]` завершает обе конструкции `qq[]` и `qq]]`.

При поиске односимвольных разделителей пропускаются экранированные разделители и `\\`. Например, при поиске терминации `/` пропускаются комбинации `\\` и `\/`. Если разделители заключены в квадратные скобки, вложенные пары также пропускаются. Например, при поиске закрывающего `]` в паре с открывающим `[` все комбинации `\\`, `\]`, и `\[` пропускаются, а также все вложенные `[` и `]` пропускаются. Однако, когда в качестве разделителей используются обратные косые черты (например, `qq\\` и `tr\\\`), ничего не пропускается. Во время поиска конца удаляются обратные косые черты, экранирующие разделители, или другие обратные черты (точнее говоря, они не копируются в безопасное место).

Для конструкций с разделителями из трех частей (`s///`, `y///` и `tr///`) поиск повторяется еще раз. Если первый разделитель не является вводным знаком препинания, то три разделителя должны быть одинаковыми, такими как `s!!!` и `tr)))`, и в этом случае второй разделитель завершает левую часть и сразу начинает правую. Если левая часть разделена знаками препинания, заключенными в квадратные скобки (то есть `()`, `[]`, `{}`, или `<>`), для правой части нужна другая пара разделителей, таких как `s(){}` и `tr[]//`. В этих случаях допускаются пробелы и комментарии между двумя частями, хотя комментарий должен следовать по крайней мере за одним символом пробела; в противном случае символ, ожидаемый в качестве начала комментария, может рассматриваться как начальный разделитель правой части.

Во время этого поиска не уделяется внимания семантике конструкции. Таким образом:

```
"$hash{"$foo/$bar"}"
```

или:

```
m/
  bar   # NOT a comment, this slash / terminated m//!
 /x
```

не образуют допустимых выражений, заключенных в кавычки. Часть, заключенная в кавычки, заканчивается на первом `"` и `/`, а остальное является синтаксической ошибкой. Поскольку за косой чертой, которая завершалась `m//`, следовала SPACE, приведенный выше пример не является `m//x`, а скорее `m//` без `/x` модификатора. Таким образом, встроенный `#` интерпретируется как литерал `#`.

Also no attention is paid to `\c\` (multichar control char syntax) during this search. Thus the second `\` in `qq/\c\/` is interpreted as a part of `\/`, and the following `/` is not recognized as a delimiter. Instead, use `\034` or `\x1c` at the end of quoted constructs.

Interpolation

The next step is interpolation in the text obtained, which is now delimiter-independent. There are multiple cases.

`<<'EOF'`

No interpolation is performed. Note that the combination `\\` is left intact, since escaped delimiters are not available for here-docs.

`m''`, **шаблон** `s'''`

На этом этапе интерполяция не выполняется. Любые последовательности с обратной косой чертой, включая `\\` обрабатываются на этапе "Синтаксического анализа регулярных выражений".

`''`, `q//`, `tr'''` `y'''`, **замена** `s'''`

Единственная интерполяция - это удаление `\` из пар `\\`. Следовательно, `"-"` в `tr'''` и `y'''` трактуется буквально как дефис, и диапазон символов недоступен. `\1` при замене `s'''` не работает как `$1`.

`tr///`, `y///`

Интерполяция переменных не происходит. Комбинации строк, изменяющие регистр и кавычки, такие как `\Q`, `\U` и `\E`, не распознаются. Другие управляющие последовательности, такие как `\200` и `\t`, и символы с обратной косой чертой, такие как `\\` и `\-`, преобразуются в соответствующие литералы. Символ `"-"` обрабатывается особым образом и, следовательно, `\-` рассматривается как литерал `"-"`.

`""`, `` `` ``, `qq//`, `qx//`, `<file*glob>`, `<<"EOF"`

`\Q`, `\U`, `\u` `\L`, `\l` `\F`, `\E` (возможно в паре c,,,) преобразуются в соответствующие конструкции Perl. Таким образом, `"$foo\Qbaz$bar"` преобразуется в `$foo . (quotemeta("baz" . $bar))` внутренне. Другие управляющие последовательности, такие как `\200` и `\t`, и символы с обратной косой чертой, такие как `\\` и `\-`, заменяются соответствующими расширениями.

Следует подчеркнуть, что *все, что находится между* `\Q` *и* `\E`, интерполируется обычным способом. Что-то вроде `"\Q\\E"` не имеет `\E` внутри. Вместо этого в нем есть `\Q`, `\\` и `E`, поэтому результат тот же, что и для `"\\\\E"`. Как правило, обратная косая черта между `\Q` и `\E` может приводить к нелогичным результатам. Итак, `"\Q\t\E"` преобразуется в `quotemeta("\t")`, что совпадает с `"\\\t"` (поскольку ТАБУЛЯЦИЯ не буквенно-цифровая). Также обратите внимание, что:

```
$str = '\t';
return "\Q$str";
```

может быть ближе к предполагаемому *намерению* автора `"\Q\t\E"`.

Интерполированные скаляры и массивы преобразуются внутренне в `join` и `"."` операции привязки. Таким образом, `"$foo XXX '@arr'"` становится:

```
$foo . " XXX '" . (join $", @arr) . "'";
```

Все операции, описанные выше, выполняются одновременно слева направо.

Поскольку в результате `"\Q` *STRING* `\E"` все метасимволы заключены в кавычки, невозможно вставить литерал `$` or `@` внутри `\Q\E` пары. Если защищено с помощью `\`, `$` будет заключено в кавычки, чтобы стать `"\\\$"`; если нет, это интерпретируется как начало интерполированного скаляра.

Обратите также внимание, что код интерполяции должен принимать решение о том, где заканчивается интерполированный скаляр. Например, действительно ли `"a $x -> {c}"` означает:

```
"a " . $x . " -> {c}";
```

или:

```
"a " . $x -> {c};
```

В большинстве случаев используется максимально длинный текст, который не содержит пробелов между компонентами и который содержит соответствующие фигурные скобки. поскольку результат может определяться голосованием на основе эвристических оценок, результат не является строго предсказуемым. К счастью, это обычно правильно для неоднозначных случаев.

Замена `s///`

Обработка `\Q`, `\U`, `\u`, `\L`, `\l`, `\F` и интерполяция происходит как с `qq//` конструкции.

Именно на этом этапе `\1` неохотно преобразуется в `$1` в тексте замены `s///`, чтобы исправить неисправимых *хакеров sed*, которые еще не освоили более разумную идиому. Если был установлен флаг `use warnings` pragma или **-W** командной строки (то есть `$^W` переменная), выдается предупреждение.

RE **в** `m?RE?`, `/RE/`, `m/RE/`, `s/RE/foo/`,

Processing of `\Q`, `\U`, `\u`, `\L`, `\l`, `\F`, `\E`, and interpolation happens (almost) as with `qq//` constructs.

Processing of `\N{...}` is also done here, and compiled into an intermediate form for the regex compiler. (This is because, as mentioned below, the regex compilation may be done at execution time, and `\N{...}` is a compile-time construct.)

However any other combinations of `\` followed by a character are not substituted but only skipped, in order to parse them as regular expressions at the following step. As `\c` is skipped at this step, `@` of `\c@` in RE is possibly treated as an array symbol (for example `@foo`), even though the same text in `qq//` gives interpolation of `\c@`.

Code blocks such as `(?{BLOCK})` are handled by temporarily passing control back to the perl parser, in a similar way that an interpolated array subscript expression such as `"foo$array[1+f("[xyz")]bar"` would be.

Moreover, inside `(?{BLOCK})`, `(?# comment )`, and a `#`-comment in a `/x`-regular expression, no processing is performed whatsoever. This is the first step at which the presence of the `/x` modifier is relevant.

Интерполяция в шаблонах имеет несколько особенностей: `$|`, `$(`, `$)` `@+` и `@-` не интерполируются, а конструкции `$var[SOMETHING]` выбираются (несколькими разными оценщиками) либо как элемент массива, либо `$var` за которым следует альтернатива RE. Вот тут-то и пригодится обозначение `${arr[$bar]}`: `/${arr[0-9]}/` интерпретируется как элемент массива `-9`, а не как регулярное выражение из переменной, за `$arr` которой следует цифра, что было бы интерпретацией `/$arr[0-9]/`. Поскольку может происходить голосование между разными оценщиками, результат непредсказуем.

Отсутствие обработки `\\` создает определенные ограничения на постобработанный текст. Если разделителем является `/`, невозможно получить комбинацию `\/` в результате этого шага. `/` завершит регулярное выражение, `\/` будет заменено на `/` на предыдущем шаге и `\\/` будет оставлено как есть. Поскольку `/` эквивалентно `\/` внутри регулярного выражения, это не имеет значения, если только разделитель не является символом, специальным для механизма RE, например, в `s*foo*bar*`, `m[foo]` или `m?foo?`; или буквенно-цифровым символом, как в:

```
m m ^ a \s* b mmx;
```

В приведенной выше ссылке, которая намеренно запутана для иллюстрации, разделителем является `m`, модификатором является `mx`, и после удаления разделителя ссылка такая же, как для `m/ ^ a \s* b /mx`. Существует несколько причин, по которым рекомендуется ограничивать использование разделителей не буквенно-цифровыми, без пробелов.

Этот шаг является последним для всех конструкций, за исключением регулярных выражений, которые обрабатываются далее.

Синтаксический анализ регулярных выражений

Предыдущие шаги выполнялись во время компиляции кода Perl, но этот выполняется во время выполнения, хотя при необходимости он может быть оптимизирован для вычисления во время компиляции. После предварительной обработки, описанной выше, и, возможно, после оценки, если задействованы конкатенация, объединение, преобразование оболочки или метаквотинг, результирующая *строка* передается в RE engine для компиляции.

Все, что происходит в движке RE, лучше обсудить в perlre, но ради непрерывности мы сделаем это здесь.

Это еще один шаг, на котором важно наличие модификатора `/x`. Механизм RE сканирует строку слева направо и преобразует ее в конечный автомат.

Символы с обратной косой чертой либо заменяются соответствующими строками литералов (как в `\{`), либо они генерируют специальные узлы в конечном автомате (как в `\b`). Специальные символы для движка RE (такие как `|`) генерируют соответствующие узлы или группы узлов. `(?#...)` комментарии игнорируются. Все остальное либо преобразуется в литеральные строки для соответствия, либо игнорируется (как и пробелы и комментарии в `#` стиле, если `/x` присутствуют).

Синтаксический анализ конструкции символьного класса, заключенной в квадратные скобки, `[...]` несколько отличается от правила, используемого для остальной части шаблона. Терминатор этой конструкции определяется с использованием тех же правил, что и для поиска терминатора конструкции с `{}` разделителями, единственным исключением является то, что `]` непосредственно следующая `[` черта обрабатывается так, как если бы ей предшествовала обратная косая черта.

The terminator of runtime `(?{...})` is found by temporarily switching control to the perl parser, which should stop at the point where the logically balancing terminating `}` is found.

It is possible to inspect both the string given to RE engine and the resulting finite automaton. See the arguments `debug` / `debugcolor` in the `use re` pragma, as well as Perl's **-Dr** command-line switch documented in "Command Switches" in perlrun.

Optimization of regular expressions

This step is listed for completeness only. Since it does not change semantics, details of this step are not documented and are subject to change without notice. This step is performed over the finite automaton that was generated during the previous pass.

It is at this stage that `split()` silently optimizes `/^/` to mean `/^/m`.

Операторы ввода-вывода

Есть несколько операторов ввода-вывода, о которых вам следует знать.

Строка, заключенная в обратные метки (серьезные акценты), сначала подвергается интерполяции двойными кавычками. Затем он интерпретируется как внешняя команда, и результатом этой команды является значение строки возврата, как в командной оболочке. В скалярном контексте возвращается единственная строка, состоящая из всех выходных данных. В контексте списка возвращается список значений, по одному на строку вывода. (Вы можете настроить `$/` на использование другого разделителя строк.) Команда выполняется каждый раз, когда вычисляется псевдобуквал. Значение состояния команды возвращается в `$?` (см. perlvar для интерпретации `$?`). В отличие от **csh**, перевод возвращаемых данных не выполняется - новые строки остаются новыми строками. В отличие от любой из оболочек, одинарные кавычки не скрывают имена переменных в команде от интерпретации. Чтобы передать буквенный знак доллара в оболочку, вам нужно скрыть его обратной косой чертой. Обобщенной формой обратных ссылок является `qx//`, или вы можете вызвать "readpipe" в perlfunc функция. (Поскольку обратные ссылки также всегда подвергаются расширению оболочки, см. perlsec по вопросам безопасности.)

В скалярном контексте вычисление дескриптора файла в угловых скобках приводит к следующей строке из этого файла (включая перевод строки, если таковой имеется), или `undef` в конце файла или при ошибке. Когда `$/` установлено значение `undef` (иногда известное как режим загрузки файла) и файл пуст, он возвращается `''` в первый раз, за которым следует `undef` впоследствии.

Обычно вы должны присвоить возвращаемое значение переменной, но есть одна ситуация, когда происходит автоматическое присвоение. Тогда и только тогда, когда входной символ является единственным элементом внутри условия `while` оператора (даже если он замаскирован под `for(;;)` цикл), значение автоматически присваивается глобальной переменной `$_`, уничтожая все, что было там ранее. (Это может показаться вам странным, но вы будете использовать конструкцию почти в каждом написанном вами скрипте Perl.) `$_` Переменная неявно локализована. Вам придется поставить `local $_;` перед циклом, если вы хотите, чтобы это произошло. Более того, если входной символ или явное присвоение входного символа скаляру используется в качестве условия `while` / `for`, то условие фактически проверяет определенность значения выражения, а не его обычное значение истинности.

Таким образом, следующие строки эквивалентны:

```
while (defined($_ = <STDIN>)) { print; }
while ($_ = <STDIN>) { print; }
while (<STDIN>) { print; }
for (;<STDIN>;) { print; }
print while defined($_ = <STDIN>);
print while ($_ = <STDIN>);
print while <STDIN>;
```

Это также ведет себя аналогично, но присваивается лексической переменной вместо `$_`:

```
while (my $line = <STDIN>) { print $line }
```

В этих конструкциях цикла присвоенное значение (независимо от того, является ли присвоение автоматическим или явным) затем проверяется, определено ли оно. Определенный тест позволяет избежать проблем, когда строка имеет строковое значение, которое было бы обработано Perl как false; например, "" или `"0"` без перевода строки в конце. Если вы действительно хотите, чтобы такие значения завершали цикл, их следует проверить явно:

```
while (($_ = <STDIN>) ne '0') { ... }
while (<STDIN>) { last unless $_; ... }
```

В других логических контекстах `<`*FILEHANDLE*`>` без явного `defined` тестирования или сравнения выдается предупреждение, если действует `use warnings` pragma или параметр командной строки **-w** (`$^W` переменная).

Дескрипторы файлов STDIN, STDOUT и STDERR предопределены. (Дескрипторы файлов `stdin`, `stdout` и `stderr` также будут работать, за исключением пакетов, где они будут интерпретироваться как локальные идентификаторы, а не глобальные.) Дополнительные дескрипторы файлов могут быть созданы, среди прочего, с помощью `open()` функции. Смотрите perlopentut и "открыть" в perlfunc для получения подробной информации об этом.

Если а `<`*FILEHANDLE*`>` используется в контексте, который ищет список, возвращается список, содержащий все входные строки, по одной строке на элемент списка. Таким образом легко расширить пространство данных до довольно большого, поэтому используйте его с осторожностью.

`<`*FILEHANDLE*`>` также может быть написано `readline(*`*FILEHANDLE*`)`. Смотрите "readline" в perlfunc.

Нулевой дескриптор файла `<>` (иногда называемый ромбовидным оператором) является специальным: его можно использовать для эмуляции поведения **sed** и **awk**, а также любой другой программы фильтрации Unix, которая принимает список имен файлов, проделывая то же самое с каждой строкой ввода из всех них. Ввод из `<>` осуществляется либо из стандартного ввода, либо из каждого файла, указанного в командной строке. Вот как это работает: при первом вычислении `<>` проверяется `@ARGV` массив, и если он пустой, `$ARGV[0]` устанавливается значение `"-"`, которое при открытии выдает стандартный ввод. Затем массив `@ARGV` обрабатывается как список имен файлов. Цикл

```
    while (<>) {
        ...                     # code for each line
    }
```

эквивалентно следующему псевдокоду, подобному Perl:

```
    unshift(@ARGV, '-') unless @ARGV;
    while ($ARGV = shift) {
        open(ARGV, $ARGV);
        while (<ARGV>) {
            ...         # code for each line
        }
    }
```

за исключением того, что это не так сложно сказать, и на самом деле будет работать. Это действительно сдвигает `@ARGV` массив и помещает текущее имя файла в `$ARGV` переменную. Он также использует внутри filehandle *ARGV*. `<>` это просто синоним `<ARGV>`, который является волшебным. (Приведенный выше псевдокод не работает, потому что он рассматривает `<ARGV>` как немагический.)

Поскольку нулевой дескриптор файла использует форму с двумя аргументами "open" в perlfunc, он интерпретирует специальные символы, поэтому, если у вас есть сценарий, подобный этому:

```
while (<>) {
    print;
}
```

и вызываем его с помощью `perl dangerous.pl 'rm -rfv *|'`, он фактически открывает канал, выполняет `rm` команду и считывает `rm` выходные данные из этого канала. Если вы хотите, чтобы все элементы в `@ARGV` интерпретировались как имена файлов, вы можете использовать модуль `ARGV::readonly` из CPAN или использовать двойную ромбовидную скобку:

```
while (<<>>) {
    print;
}
```

Использование двойных угловых скобок внутри a while приводит к тому, что open использует форму с тремя аргументами (со вторым аргументом, равным `<`), поэтому все аргументы в `ARGV` обрабатываются как буквальные имена файлов (включая `"-"`). (Обратите внимание, что для удобства, если вы используете `<<>>` и если `@ARGV` значение пусто, оно все равно будет считываться из стандартного ввода.)

Вы можете изменять `@ARGV` перед первым `<>`, если в конечном итоге массив будет содержать список имен файлов, которые вам действительно нужны. Номера строк (`$.`) сохраняются, как если бы входные данные представляли собой один большой файл happy. Смотрите пример в "eof" в perlfunc, чтобы узнать, как сбросить номера строк в каждом файле.

Если вы хотите установить для `@ARGV` свой собственный список файлов, продолжайте. Это устанавливает для `@ARGV` всех обычных текстовых файлов, если не было указано значение `@ARGV`:

```
@ARGV = grep { -f && -T } glob('*') unless @ARGV;
```

Вы даже можете установить их в качестве команд канала. Например, это автоматически фильтрует сжатые аргументы с помощью **gzip**:

```
@ARGV = map { /\.(gz|Z)$/ ? "gzip -dc < $_ |" : $_ } @ARGV;
```

Если вы хотите передавать переключатели в свой скрипт, вы можете использовать один из `Getopts` модулей или поместить цикл спереди, например, так:

```
    while ($_ = $ARGV[0], /^-/) {
        shift;
        last if /^--$/;
        if (/^-D(.*)/) { $debug = $1 }
        if (/^-v/)     { $verbose++  }
        # ...          # other switches
    }

    while (<>) {
        # ...          # code for each line
    }
```

`<>` Символ будет возвращен `undef` для обозначения конца файла только один раз. Если вы вызовете его снова после этого, он предположит, что вы обрабатываете другой `@ARGV` список, и, если вы не установили `@ARGV`, будет считывать входные данные из STDIN.

Если угловые скобки содержат простую скалярную переменную (например, `$foo`), то эта переменная содержит имя дескриптора файла для ввода, или его typeglob, или ссылку на него. Например:

```
$fh = \*STDIN;
$line = <$fh>;
```

Если то, что находится внутри угловых скобок, не является ни дескриптором файла, ни простой скалярной переменной, содержащей имя дескриптора файла, typeglob или ссылку на typeglob, это интерпретируется как шаблон имени файла, подлежащий глобированию, и в зависимости от контекста возвращается либо список имен файлов, либо следующее имя файла в списке. Это различие определяется исключительно по синтаксическим признакам. Это означает, что `<$x>` всегда является `readline()` из косвенного дескриптора, но `<$hash{key}>` всегда является `glob()`. Это потому, что `$x` это простая скалярная переменная, но `$hash{key}` это не так - это хэш-элемент. Даже `<$x >` (обратите внимание на лишний пробел) обрабатывается как `glob("$x ")`, а не `readline($x)`.

Сначала выполняется один уровень интерпретации двойных кавычек, но вы не можете сказать, `<$foo>` потому что это косвенный дескриптор файла, как объяснялось в предыдущем параграфе. (В более старых версиях Perl программисты вставляли фигурные скобки, чтобы принудительно интерпретировать как glob имени файла: `<${foo}>`. В наши дни считается более чистым вызывать внутреннюю функцию напрямую как `glob($foo)`, что, вероятно, было правильным способом сделать это в первую очередь.) Например:

```
while (<*.c>) {
    chmod 0644, $_;
}
```

примерно эквивалентно:

```
open(FOO, "echo *.c | tr -s ' \t\r\f' '\\012\\012\\012\\012'|");
while (<FOO>) {
    chomp;
    chmod 0644, $_;
}
```

за исключением того, что глобализация фактически выполняется внутренне с использованием стандартного `File::Glob` расширения. Конечно, самый короткий способ сделать это -:

```
chmod 0644, <*.c>;
```

Глобус (file) вычисляет свой (встроенный) аргумент только при запуске нового списка. Все значения должны быть прочитаны, прежде чем он начнется заново. В контексте списка это не важно, потому что вы все равно автоматически получаете их все. Однако в скалярном контексте оператор возвращает следующее значение каждый раз, когда он вызывается, или `undef` когда список заканчивается. Как и при чтении дескриптора файла, автоматически `defined` генерируется, когда в тестовой части a происходит большой объем `while`, потому что возврат допустимого большого объема (например, файла с именем *0*) в противном случае завершил бы цикл. Опять же, `undef` возвращается только один раз. Так что, если вы ожидаете от глобуса одно значение, гораздо лучше сказать

```
($file) = <blurch*>;
```

чем

```
$file = <blurch*>;
```

потому что последний будет чередоваться между возвратом имени файла и возвратом false.

Если вы пытаетесь выполнить интерполяцию переменных, определенно лучше использовать функцию `glob()`, потому что более старая нотация может привести к путанице с косвенной нотацией дескриптора файла.

```
@files = glob("$dir/*.[ch]");
@files = glob($files[$i]);
```

Если в качестве условия цикла `while` or `for` используется выражение глобулирования на основе угловых скобок, то оно будет неявно присвоено `$_`. Если в качестве условия `while` / `for` используется либо расширяющееся выражение, либо явное присвоение расширяющегося выражения скаляру, то условие фактически проверяет определенность значения выражения, а не его обычное значение истинности.

Сворачивание константы

Как и C, Perl выполняет определенное количество вычислений выражений во время компиляции всякий раз, когда определяет, что все аргументы оператора статичны и не имеют побочных эффектов. В частности, конкатенация строк происходит во время компиляции между литералами, которые не выполняют замену переменных. Интерполяция обратной косой черты также происходит во время компиляции. Вы можете сказать

```
  'Now is the time for all'
. "\n"
.  'good men to come to.'
```

и все это внутренне сводится к одной строке. Аналогично, если вы скажете

```
    foreach $file (@filenames) {
        if (-s $file > 5 + 100 * 2**16) {   }
    }
```

компилятор предварительно вычисляет число, которое представляет это выражение, чтобы интерпретатору не приходилось этого делать.

Отсутствие операций

В Perl официально нет оператора no-op, но простые константы `0` и `1` имеют специальный регистр, чтобы не выдавать предупреждение в контексте void, поэтому вы можете, например, безопасно выполнить

```
1 while foo();
```

Побитовые строковые операторы

Побитовые операторы (`~ | & ^`) могут манипулировать битовыми строками любого размера.

Если операнды двоичной побитовой операции представляют собой строки разного размера, то операции **|** и **^** действуют так, как если бы более короткий операнд имел дополнительные нулевые биты справа, в то время как операция **&** действует так, как если бы более длинный операнд был урезан до длины более короткого. Степень детализации для такого расширения или усечения составляет один или несколько байтов.

```
# ASCII-based examples
print "j p \n" ^ " a h";           # prints "JAPH\n"
print "JA" | "  ph\n";             # prints "japh\n"
print "japh\nJunk" & '_____';      # prints "JAPH\n";
print 'p N$' ^ " E<H\n";           # prints "Perl\n";
```

Если вы собираетесь манипулировать битовыми строками, убедитесь, что вы предоставляете битовые строки: если операндом является число, это будет означать **числовую** побитовую операцию. Вы можете явно указать, какой тип операции вы собираетесь выполнять, используя `""` или `0+` , как в примерах ниже.

```
$foo =  150  |  105;     # yields 255  (0x96 | 0x69 is 0xFF)
$foo = '150' |  105;     # yields 255
$foo =  150  | '105';    # yields 255
$foo = '150' | '105';    # yields string '155' (under ASCII)

$baz = 0+$foo & 0+$bar;  # both ops explicitly numeric
$biz = "$foo" ^ "$bar";  # both ops explicitly stringy
```

Этого несколько непредсказуемого поведения можно избежать с помощью функции "побитового", новой в Perl 5.22. Вы можете включить ее с помощью `use feature 'bitwise'` или `use v5.28` . До Perl 5.28 он выдавал предупреждение в категории `"experimental::bitwise"` . В соответствии с этой функцией четыре стандартных побитовых оператора (`~ | & ^`) всегда являются числовыми. Добавление точки после каждого оператора (`~. |. &. ^.`) заставляет его обрабатывать свои операнды как строки:

```
use feature "bitwise";
$foo =  150  |  105;     # yields 255  (0x96 | 0x69 is 0xFF)
$foo = '150' |  105;     # yields 255
$foo =  150  | '105';    # yields 255
$foo = '150' | '105';    # yields 255
$foo =  150  |. 105;     # yields string '155'
$foo = '150' |. 105;     # yields string '155'
$foo =  150  |.'105';    # yields string '155'
$foo = '150' |.'105';    # yields string '155'

$baz = $foo &  $bar;     # both operands numeric
$biz = $foo ^. $bar;     # both operands stringy
```

Варианты назначения этих операторов (`&= |= ^= &.= |.= ^.=`) ведут себя аналогично в рамках этой функции.

Это фатальная ошибка, если операнд содержит символ, порядковое значение которого выше 0xFF и, следовательно, его невозможно выразить, кроме как в UTF-8. Операция выполняется с копией, отличной от UTF-8, для других операндов, закодированных в UTF-8. Смотрите "Семантика байтов и символов" в perlunicode.

Смотрите "vec" в perlfunc для получения информации о том, как манипулировать отдельными битами в битовом векторе.

Целочисленная арифметика

По умолчанию Perl предполагает, что большую часть своей арифметики он должен выполнять с плавающей запятой. Но, говоря

```
use integer;
```

вы можете указать компилятору использовать целочисленные операции (подробное объяснение см. в integer) отсюда и до конца заключающего БЛОКА. Внутренний БЛОК может отменить это, сказав

```
no integer;
```

который длится до конца этого блока. Обратите внимание, что это не означает, что все является целым числом, просто Perl будет использовать целочисленные операции для арифметики, сравнения и побитовых операторов. Например, даже под `use integer` , если вы возьмете `sqrt(2)` , вы все равно получите `1.4142135623731` или около того.

Используемые с числами побитовые операторы (`&` `|` `^` `~` `<<` `>>`) всегда дают целочисленные результаты. (Но смотрите также "Побитовые строковые операторы".) Тем не менее, `use integer` для них все еще имеет значение. По умолчанию их результаты интерпретируются как целые числа без знака, но если `use integer` действует, их результаты интерпретируются как целые числа со знаком. Например, `~0` обычно вычисляется как большое целое значение. Однако, `use integer; ~0` это `-1` на машинах с двумя дополнениями.

Арифметика с плавающей запятой

Хотя `use integer` предоставляет арифметику только для целых чисел, аналогичного механизма для автоматического округления или усечения до определенного количества знаков после запятой не существует. Для округления до определенного количества цифр обычно проще всего использовать `sprintf()` или `printf()` . Смотрите perlfaq4.

Числа с плавающей запятой являются лишь приближением к тому, что математик назвал бы действительными числами. Реальных чисел бесконечно больше, чем чисел с плавающей запятой, поэтому некоторые углы необходимо обрезать. Например:

```
printf "%.20g\n", 123456789123456789;
#        produces 123456789123456784
```

Тестирование на точное равенство или неравенство с плавающей запятой - плохая идея. Вот (относительно дорогостоящий) обходной путь для сравнения того, равны ли два числа с плавающей запятой определенному количеству знаков после запятой. Смотрите Knuth, том II, для более подробного рассмотрения этого вопроса.

```
    sub fp_equal {
        my ($X, $Y, $POINTS) = @_;
        my ($tX, $tY);
        $tX = sprintf("%.${POINTS}g", $X);
        $tY = sprintf("%.${POINTS}g", $Y);
        return $tX eq $tY;
    }
```

Модуль POSIX (часть стандартного дистрибутива perl) реализует `ceil()` , `floor()` и другие математические и тригонометрические функции. Модуль Math::Complex (часть стандартного дистрибутива perl) определяет математические функции, которые работают как с действительными, так и с мнимыми числами. `Math::Complex` не так эффективен, как POSIX, но POSIX не может работать с комплексными числами.

Округление в финансовых приложениях может иметь серьезные последствия, и используемый метод округления должен быть точно указан. В этих случаях, вероятно, имеет смысл не доверять той системе округления, которая используется Perl, а вместо этого реализовать нужную вам функцию округления самостоятельно.

Большие числа

Стандартные модули Math::BigInt , Math::BigRat и Math::BigFloat , наряду с `bignum` , `bigint` и `bigrat` прагмами, обеспечивают арифметику переменной точности и перегруженные операторы, хотя в настоящее время они работают довольно медленно. За счет некоторого пространства и значительной скорости они позволяют избежать обычных ошибок, связанных с представлениями ограниченной точности.

```
        use 5.010;
        use bigint;  # easy interface to Math::BigInt
        $x = 123456789123456789;
        say $x * $x;
    +15241578780673678515622620750190521
```

Или с помощью rationals:

```
use 5.010;
use bigrat;
$x = 3/22;
$y = 4/6;
say "x/y is ", $x/$y;
say "x*y is ", $x*$y;
x/y is 9/44
x*y is 1/11
```

Несколько модулей позволяют выполнять вычисления с неограниченной или фиксированной точностью (ограниченной только объемом памяти и процессорным временем). Также есть несколько нестандартных модулей, которые обеспечивают более быструю реализацию с помощью внешних библиотек C.

Вот краткое, но неполное резюме:

```
Math::String           treat string sequences like numbers
Math::FixedPrecision   calculate with a fixed precision
Math::Currency         for currency calculations
Bit::Vector            manipulate bit vectors fast (uses C)
Math::BigIntFast       Bit::Vector wrapper for big numbers
Math::Pari             provides access to the Pari C library
Math::Cephes           uses the external Cephes C library (no
                       big numbers)
Math::Cephes::Fraction fractions via the Cephes library
Math::GMP              another one using an external C library
Math::GMPz             an alternative interface to libgmp's big ints
Math::GMPq             an interface to libgmp's fraction numbers
Math::GMPf             an interface to libgmp's floating point numbers
```

Выбирайте с умом.