perlsyn (Источник, CPAN)

Содержание

ИМЯ

синтаксис perlsyn - Perl

ОПИСАНИЕ

Программа на Perl состоит из последовательности объявлений и инструкций, которые выполняются сверху вниз. Циклы, подпрограммы и другие управляющие структуры позволяют вам перемещаться по коду.

Perl - это язык с **свободной формой**: вы можете форматировать его и делать отступы так, как вам нравится. Пробелы служат в основном для разделения токенов, в отличие от таких языков, как Python, где это важная часть синтаксиса, или Fortran, где это несущественно.

Многие синтаксические элементы Perl являются **необязательными**. Вместо того, чтобы требовать от вас заключать в круглые скобки каждый вызов функции и объявлять каждую переменную, вы часто можете не включать такие явные элементы, и Perl поймет, что вы имели в виду. Это известно как **Делай то, что я имею в виду**, сокращенно **DWIM**. Это позволяет программистам быть **ленивыми** и кодировать в стиле, который им удобен.

Perl **заимствует синтаксис** и концепции из многих языков: awk, sed, C, Bourne Shell, Smalltalk, Lisp и даже английского. Другие языки заимствовали синтаксис из Perl, особенно его расширения регулярных выражений. Поэтому, если вы программировали на другом языке, вы увидите знакомые фрагменты в Perl. Часто они работают одинаково, но информацию об их различиях см. в perltrap.

Объявления

Единственное, что вам нужно объявить в Perl, - это форматы отчетов и подпрограммы (а иногда даже не подпрограммы). Скалярная переменная содержит неопределенное значение (undef) до тех пор, пока ей не будет присвоено определенное значение, которое является чем угодно, кроме undef . При использовании в качестве числа, undef обрабатывается как 0 ; при использовании в качестве строки, оно обрабатывается как пустая строка, "" ; и при использовании в качестве ссылки, которой не присваивается, оно обрабатывается как ошибка. Если вы включите предупреждения, вы будете получать уведомления о неинициализированном значении всякий раз, когда вы обрабатываете undef как строку или число. Ну, обычно. Логические контексты, такие как:

```
if ($a) {}
```

освобождены от предупреждений (поскольку их интересует истинность, а не определенность). Такие операторы, как ++ , -- , += -= , .= , и,, которые работают с неопределенными переменными, такими как:

```
undef $a;
$a++;
```

также всегда освобожден от подобных предупреждений.

Объявление может быть помещено в любое место, куда может быть помещен оператор, но не влияет на выполнение первичной последовательности операторов: все объявления вступают в силу во время компиляции. Все объявления обычно помещаются в начале или в конце скрипта. Однако, если вы используете закрытые переменные с лексической областью действия, созданные с помощью `my()`, `state()` или `our()`, вам нужно убедиться, что ваш формат или определение подпрограммы находится в той же области действия блока, что и my, если вы ожидаете получить доступ к этим закрытым переменным.

Объявление подпрограммы позволяет использовать имя подпрограммы так, как если бы оно было оператором списка с этого момента в программе. Вы можете объявить подпрограмму, не определяя ее, сказав `sub name`, таким образом:

```
sub myname;
$me = myname $0             or die "can't get myname";
```

Простое объявление, подобное этому, объявляет функцию оператором списка, а не унарным оператором, поэтому вы должны быть осторожны при использовании круглых скобок (или `or` вместо `||`.) Оператор `||` слишком сильно привязывается для использования после операторов списка; он становится частью последнего элемента. Вы всегда можете заключить аргументы операторов списка в круглые скобки, чтобы снова превратить оператор списка во что-то, что больше похоже на вызов функции. В качестве альтернативы вы можете использовать прототип `($)` для преобразования подпрограммы в унарный оператор:

```
sub myname ($);
$me = myname $0             || die "can't get myname";
```

Теперь он разбирается так, как вы ожидали, но вам все равно следует привыкнуть использовать круглые скобки в этой ситуации. Подробнее о прототипах см. perlsub .

Объявления подпрограмм также могут быть загружены с помощью `require` инструкции или загружены и импортированы в ваше пространство имен с помощью `use` инструкции. Подробнее об этом смотрите в perlmod.

Последовательность инструкций может содержать объявления переменных с лексической областью действия, но, помимо объявления имени переменной, объявление действует как обычная инструкция и разрабатывается внутри последовательности инструкций, как если бы это была обычная инструкция. Это означает, что на самом деле он оказывает влияние как во время компиляции, так и во время выполнения.

Комментарии

Текст от `"#"` символа до конца строки является комментарием и игнорируется. Исключения включают `"#"` внутри строки или регулярного выражения.

Простые инструкции

Единственный вид простого оператора - это выражение, оцениваемое на предмет его побочных эффектов. Каждый простой оператор должен заканчиваться точкой с запятой, если только это не последний оператор в блоке, и в этом случае точка с запятой необязательна. Но в любом случае ставьте точку с запятой, если блок занимает более одной строки, потому что в конечном итоге вы можете добавить еще одну строку. Обратите внимание, что существуют операторы типа `eval {}`, `sub {}` и `do {}` которые *выглядят* как составные операторы, но таковыми не являются - это просто термины в выражении - и, следовательно, нуждаются в явном завершении при использовании в качестве последнего элемента в операторе.

За любым простым оператором может необязательно следовать *ОДИНОЧНЫЙ* модификатор, непосредственно перед завершающей точкой с запятой (или окончанием блока). Возможные модификаторы::

```
if EXPR
unless EXPR
while EXPR
until EXPR
for LIST
foreach LIST
when EXPR
```

То, `EXPR` что следует за модификатором, называется "условием". Его истинность или ложность определяет, как будет вести себя модификатор.

`if` выполняет инструкцию один раз, *если* и только если условие истинно. `unless` наоборот, он выполняет инструкцию, *если* условие не равно true (то есть, если условие равно false). Смотрите "Скалярные значения" в perldata для определения true и false.

```
print "Basset hounds got long ears" if length $ear >= 10;
go_outside() and play() unless $is_raining;
```

`for(each)` Модификатор является итератором: он выполняет инструкцию один раз для каждого элемента в СПИСКЕ (с `$_` псевдонимами для каждого элемента по очереди). В этой форме нет синтаксиса для указания цикла for в стиле C или переменной итерации с лексической областью действия.

```
print "Hello $_!\n" for qw(world Dolly nurse);
```

while повторяет оператор, *пока* условие истинно. Postfix while имеет ту же магическую обработку некоторых видов условий, что и prefix while . until делает обратное, он повторяет оператор *до* тех пор, пока условие не станет истинным (или пока условие не станет ложным):

```
# Both of these count from 0 to 10.
print $i++ while $i <= 10;
print $j++ until $j >  10;
```

Модификаторы while и until имеют обычную семантику "while цикла" (сначала вычисляется условие), за исключением случаев применения к do -БЛОКУ (или к оператору Perl4 do -ПОДПРОГРАММЫ), в этом случае блок выполняется один раз перед вычислением условия.

Это сделано для того, чтобы вы могли писать циклы типа:

```
do {
    $line = <STDIN>;
    ...
} until !defined($line) || $line eq ".\n"
```

Смотрите "do" в perlfunc. Обратите также внимание, что инструкции управления циклом, описанные ниже, *НЕ* будут работать в этой конструкции, потому что модификаторы не принимают метки цикла. Извините. Вы всегда можете поместить другой блок внутри него (для next / redo) или вокруг него (для last), чтобы делать подобные вещи.

Для next или redo просто удвойте фигурные скобки:

```
do {{
    next if $x == $y;
    # do something here
}} until $x++ > $z;
```

Для last вам нужно быть более сложным и заключить его в фигурные скобки:

```
{
    do {
        last if $x == $y**2;
        # do something here
    } while $x++ <= $z;
}
```

Если вам нужны оба next и last , вы должны выполнить оба, а также использовать метку цикла:

```
LOOP: {
    do {{
        next if $x == $y;
        last LOOP if $x == $y**2;
        # do something here
    }} until $x++ > $z;
}
```

ПРИМЕЧАНИЕ: Поведение my , state , or, our измененное с помощью модификатора оператора conditional или конструкции цикла (например, my $x if ...), **не определено**. Значение my переменной может быть undef любым ранее присвоенным значением или, возможно, чем угодно другим. Не полагайтесь на это. Будущие версии perl могут делать что-то отличное от версии perl, на которой вы это пробовали. Здесь будут драконы.

Модификатор when - это экспериментальная функция, которая впервые появилась в Perl 5.14. Чтобы использовать ее, вы должны включить use v5.14 объявление. (Технически для этого требуется только switch функция, но этот ее аспект не был доступен до версии 5.14.) Работает только внутри foreach цикла или given блока, выполняет инструкцию, только если smartmatch $_ ~~ *EXPR* имеет значение true. Если выполняется инструкция, за ней следует a next изнутри a foreach и break изнутри a given .

В текущей реализации foreach цикл может находиться в любом месте динамической области действия when модификатора, но должен находиться в пределах given лексической области действия блока. Это ограничение может быть смягчено в будущей версии. Смотрите "Инструкции переключения" ниже.

Составные операторы

В Perl последовательность операторов, определяющая область действия, называется блоком. Иногда блок ограничивается содержащим его файлом (в случае требуемого файла или программы в целом), а иногда блок ограничивается экстентом строки (в случае eval).

Но обычно блок ограничивается фигурными скобками, также известными как фигурные скобки. Мы будем называть эту синтаксическую конструкцию БЛОКОМ. Поскольку заключающие фигурные скобки также являются синтаксисом выражений конструктора ссылок на хэш (см. perlref), иногда может потребоваться устранить неоднозначность, поместив ; сразу после открывающей фигурной скобки, чтобы Perl понял, что фигурная

скобка является началом блока. Вам чаще всего придется устранять неоднозначность другим способом, помещая `+` непосредственно перед открывающей фигурной скобкой, чтобы заставить ее интерпретироваться как выражение конструктора ссылок на хэш. Считается хорошим тоном широко использовать эти механизмы устранения неоднозначностей, а не только тогда, когда Perl в противном случае неправильно угадал бы.

Для управления потоком могут использоваться следующие составные операторы:

```
if (EXPR) BLOCK
if (EXPR) BLOCK else BLOCK
if (EXPR) BLOCK elsif (EXPR) BLOCK ...
if (EXPR) BLOCK elsif (EXPR) BLOCK ... else BLOCK

unless (EXPR) BLOCK
unless (EXPR) BLOCK else BLOCK
unless (EXPR) BLOCK elsif (EXPR) BLOCK ...
unless (EXPR) BLOCK elsif (EXPR) BLOCK ... else BLOCK

given (EXPR) BLOCK

LABEL while (EXPR) BLOCK
LABEL while (EXPR) BLOCK continue BLOCK

LABEL until (EXPR) BLOCK
LABEL until (EXPR) BLOCK continue BLOCK

LABEL for (EXPR; EXPR; EXPR) BLOCK
LABEL for VAR (LIST) BLOCK
LABEL for VAR (LIST) BLOCK continue BLOCK

LABEL foreach (EXPR; EXPR; EXPR) BLOCK
LABEL foreach VAR (LIST) BLOCK
LABEL foreach VAR (LIST) BLOCK continue BLOCK

LABEL BLOCK
LABEL BLOCK continue BLOCK

PHASE BLOCK
```

Начиная с версии Perl 5.36, вы можете выполнять итерацию по нескольким значениям одновременно, указывая список лексических единиц в круглых скобках:

```
no warnings "experimental::for_list";
LABEL for my (VAR, VAR) (LIST) BLOCK
LABEL for my (VAR, VAR) (LIST) BLOCK continue BLOCK
LABEL foreach my (VAR, VAR) (LIST) BLOCK
LABEL foreach my (VAR, VAR) (LIST) BLOCK continue BLOCK
```

Если включена экспериментальная `try` функция, также можно использовать следующее

```
try BLOCK catch (VAR) BLOCK
try BLOCK catch (VAR) BLOCK finally BLOCK
```

Экспериментальная `given` инструкция *не включается автоматически*; смотрите "Инструкции переключения" ниже, как это сделать, и сопутствующие предостережения.

В отличие от C и Pascal, в Perl все это определяется в терминах блоков, а не операторов. Это означает, что фигурные скобки *обязательны* - не допускаются висячие операторы. Если вы хотите написать условные выражения без фигурных скобок, есть несколько других способов сделать это. Все перечисленные ниже делают то же самое:

```
if (!open(FOO)) { die "Can't open $FOO: $!" }
die "Can't open $FOO: $!" unless open(FOO);
open(FOO)  || die "Can't open $FOO: $!";
open(FOO) ? () : die "Can't open $FOO: $!";
    # a bit exotic, that last one
```

`if` Инструкция проста. Поскольку блоки всегда заключены в фигурные скобки, никогда не возникает никакой двусмысленности относительно того, с чем связан `if` an `else`. Если вы используете `unless` вместо `if`, смысл теста меняется на противоположный. За `if`, `unless` может следовать `else`. за `unless` может даже следовать один или несколько `elsif` операторов, хотя вам, возможно, захочется дважды подумать, прежде чем использовать эту конкретную языковую конструкцию, поскольку каждому, кто читает ваш код, придется подумать как минимум дважды, прежде чем они смогут понять, что происходит.

The `while` statement executes the block as long as the expression is true. The `until` statement executes the block as long as the expression is false. The LABEL is optional, and if present, consists of an identifier followed by a colon. The LABEL identifies the loop for the loop control statements `next`, `last`, and `redo`. If the LABEL is omitted, the loop control statement refers to the innermost enclosing loop. This may include dynamically searching through your call-stack at run time to find the LABEL. Such desperate behavior triggers a warning if you use the `use warnings` pragma or the **-w** flag.

If the condition expression of a `while` statement is based on any of a group of iterative expression types then it gets some magic treatment. The affected iterative expression types are `readline`, the `<FILEHANDLE>` input operator, `readdir`, `glob`, the `<PATTERN>` globbing operator, and `each`. If the condition expression is one of these expression types, then the value yielded by the iterative operator will be implicitly assigned to `$_`. If the condition expression is one of these expression types or an explicit assignment of one of them to a scalar, then the condition actually tests for definedness of the expression's value, not for its regular truth value.

If there is a `continue` BLOCK, it is always executed just before the conditional is about to be evaluated again. Thus it can be used to increment a loop variable, even when the loop has been continued via the `next` statement.

Когда блоку предшествует ключевое слово фазы компиляции, такое как `BEGIN`, `END`, `INIT`, `CHECK`, `UNITCHECK` или,, тогда блок будет выполняться только на соответствующем этапе выполнения. Смотрите `perlmod` для получения более подробной информации.

Модули расширения также могут подключаться к анализатору Perl для определения новых типов составных операторов. Они вводятся ключевым словом, которое распознает расширение, а синтаксис, следующий за ключевым словом, полностью определяется расширением. Если вы разработчик, смотрите `"PL keyword plugin" в perlapi`, чтобы узнать о механизме. Если вы используете такой модуль, обратитесь к документации модуля для получения подробной информации о синтаксисе, который он определяет.

Управление циклом

Команда `next` запускает следующую итерацию цикла:

```
LINE: while (<STDIN>) {
    next LINE if /^#/;      # discard comments
    ...
}
```

`last` Команда немедленно завершает рассматриваемый цикл. `continue` Блок, если таковой имеется, не выполняется:

```
LINE: while (<STDIN>) {
    last LINE if /^$/;       # exit when done with header
    ...
}
```

Команда `redo` перезапускает блок цикла без повторного вычисления условия. `continue` Блок, если таковой имеется, *не* выполняется. Эта команда обычно используется программами, которые хотят солгать самим себе о том, что только что было введено.

Например, при обработке файла типа */etc/termcap*. Если ваши входные строки могут заканчиваться обратной косой чертой, указывающей на продолжение, вы хотите пропустить вперед и получить следующую запись.

```
while (<>) {
    chomp;
    if (s/\\$//) {
        $_ .= <>;
        redo unless eof();
    }
    # now process $_
}
```

который является сокращением Perl для более явно написанной версии:

```
LINE: while (defined($line = <ARGV>)) {
    chomp($line);
    if ($line =~ s/\\$//) {
        $line .= <ARGV>;
        redo LINE unless eof(); # not eof(ARGV)!
    }
    # now process $line
}
```

Обратите внимание, что если бы в приведенном выше коде был `continue` блок, он выполнялся бы только в строках, отбрасываемых регулярным выражением (поскольку redo пропускает блок продолжения). Блок продолжения часто используется для сброса счетчиков строк или `m?pat?` одноразовых совпадений:

```
# inspired by :1,$g/fred/s//WILMA/
while (<>) {
    m?(fred)?    && s//WILMA $1 WILMA/;
    m?(barney)?  && s//BETTY $1 BETTY/;
    m?(homer)?   && s//MARGE $1 MARGE/;
} continue {
    print "$ARGV $.: $_";
    close ARGV  if eof;             # reset $.
    reset       if eof;             # reset ?pat?
}
```

Если слово `while` заменить словом `until`, смысл теста меняется на противоположный, но условие все равно проверяется перед первой итерацией.

Операторы управления циклом не работают в `if` или `unless`, поскольку они не являются циклами. Однако вы можете удвоить фигурные скобки, чтобы сделать их таковыми.

```
if (/pattern/) {{
    last if /fred/;
    next if /barney/; # same effect as "last",
                      # but doesn't document as well
    # do something here
}}
```

Это вызвано тем фактом, что блок сам по себе действует как цикл, который выполняется один раз, см. "Основные блоки".

Форма `while/if BLOCK BLOCK`, доступная в Perl 4, больше недоступна. Замените любое вхождение `if BLOCK` на `if (do BLOCK)`.

Для циклов

`for` Цикл Perl в стиле C работает как соответствующий `while` цикл; это означает, что этот:

```
for ($i = 1; $i < 10; $i++) {
    ...
}
```

такой же, как этот:

```
$i = 1;
while ($i < 10) {
    ...
} continue {
    $i++;
}
```

Есть одно небольшое отличие: если переменные объявляются с помощью `my` в разделе инициализации `for`, лексическая область видимости этих переменных в точности соответствует `for` циклу (телу цикла и разделам управления). Для иллюстрации:

```
my $i = 'samba';
for (my $i = 1; $i <= 4; $i++) {
    print "$i\n";
}
print "$i\n";
```

при выполнении выдает:

```
1
2
3
4
samba
```

В качестве особого случая, если тест в `for` цикле (или соответствующем `while` цикле) пуст, он обрабатывается как `true`. То есть оба

```
for (;;) {
    ...
}
```

и

```
while () {
    ...
}
```

обрабатываются как бесконечные циклы.

Помимо обычного зацикливания индекса массива, `for` может использоваться во многих других интересных приложениях. Вот способ, позволяющий избежать проблемы, с которой вы сталкиваетесь, если вы явно проверяете конец файла в интерактивном файловом дескрипторе, из-за чего кажется, что ваша программа зависает.

```
$on_a_tty = -t STDIN && -t STDOUT;
sub prompt { print "yes? " if $on_a_tty }
for ( prompt(); <STDIN>; prompt() ) {
    # do something
}
```

Выражение условия `for` цикла получает ту же магическую обработку `readline` и др., что и выражение условия `while` цикла.

Циклы Foreach

`foreach` Цикл выполняет итерацию по обычному значению списка и устанавливает скалярную переменную VAR в качестве каждого элемента списка по очереди. Если переменной предшествует ключевое слово `my` , то она имеет лексическую область видимости и, следовательно, видна только внутри цикла. В противном случае переменная неявно является локальной для цикла и восстанавливает свое прежнее значение при выходе из цикла. Если переменная была ранее объявлена с помощью `my` , она использует эту переменную вместо глобальной, но она по-прежнему локализована в цикле. Эта неявная локализация происходит *только* для циклов, отличных от C-стиля.

Ключевое слово `foreach` на самом деле является синонимом `for` ключевого слова, поэтому вы можете использовать любое из них. Если параметр VAR опущен, `$_` устанавливается в каждое значение.

Если какой-либо элемент LIST является значением lvalue, вы можете изменить его, изменив VAR внутри цикла. И наоборот, если какой-либо элемент LIST НЕ является значением lvalue, любая попытка изменить этот элемент завершится неудачей. Другими словами, `foreach` переменная индекса цикла является неявным псевдонимом для каждого элемента в списке, по которому вы выполняете цикл.

Если какая-либо часть СПИСКА является массивом, `foreach` вы сильно запутаетесь, если добавите или удалите элементы в теле цикла, например, с помощью `splice` . Так что не делайте этого.

`foreach` вероятно, не будет делать то, что вы ожидаете, если VAR является привязанной или другой специальной переменной. Этого тоже не делайте.

Начиная с Perl 5.22, существует экспериментальный вариант этого цикла, который принимает переменную, перед которой стоит обратная косая черта для VAR, и в этом случае элементы в СПИСКЕ должны быть ссылками. Переменная с обратной косой чертой станет псевдонимом для каждого элемента в СПИСКЕ, на который ссылается ССЫЛКА, который должен быть правильного типа. В этом случае переменная не обязательно должна быть скалярной, и за обратной косой чертой может следовать `my` . Чтобы использовать эту форму, вы должны включить `refaliasing` функцию через `use feature` . (Смотрите Функцию. Смотрите также "Назначение ссылок" в perlref.)

Начиная с Perl 5.36, вы можете выполнять итерацию по нескольким значениям одновременно. Вы можете выполнять итерации только с лексическими скалярами в качестве переменных итератора - в отличие от назначения списка, его невозможно использовать `undef` для обозначения нежелательного значения. Это ограничение текущей реализации, и оно может быть изменено в будущем.

Если размер СПИСКА не является точным кратным числу переменных итератора, то на последней итерации "лишние" переменные итератора являются псевдонимами для `undef` , как если бы СПИСОК был `, undef` добавлен столько раз, сколько необходимо, чтобы его длина стала точным кратным. Это происходит независимо от того, является ли LIST литеральным СПИСКОМ или массивом - т.е. массивы не расширяются, если их размер не кратен размеру итерации, что согласуется с повторением массива по очереди. Поскольку эти элементы заполнения не являются значениями lvalues, попытка их изменения завершится неудачей, что соответствует поведению при повторении списка с литералом `undef` s. Если это не то поведение, которое вы хотите, то перед запуском цикла либо явно расширьте свой массив, чтобы он был точным кратным, либо явно создайте исключение.

Примеры:

```
for (@ary) { s/foo/bar/ }

for my $elem (@elements) {
    $elem *= 2;
}

for $count (reverse(1..10), "BOOM") {
    print $count, "\n";
    sleep(1);
}

for (1..15) { print "Merry Christmas\n"; }

foreach $item (split(/:[\\\n:]*/, $ENV{TERMCAP})) {
    print "Item: $item\n";
}

use feature "refaliasing";
no warnings "experimental::refaliasing";
foreach \my %hash (@array_of_hash_references) {
    # do something with each %hash
}

foreach my ($foo, $bar, $baz) (@list) {
    # do something three-at-a-time
}

foreach my ($key, $value) (%hash) {
    # iterate over the hash
    # The hash is immediately copied to a flat list before the loop
    # starts. The list contains copies of keys but aliases of values.
    # This is the same behaviour as for $var (%hash) {...}
}
```

Вот как программист на C может закодировать определенный алгоритм на Perl:

```
for (my $i = 0; $i < @ary1; $i++) {
    for (my $j = 0; $j < @ary2; $j++) {
        if ($ary1[$i] > $ary2[$j]) {
            last; # can't go to outer :-(
        }
        $ary1[$i] += $ary2[$j];
    }
    # this is where that last takes me
}
```

Принимая во внимание, что вот как программист Perl, более знакомый с идиомой, мог бы это сделать:

```
OUTER: for my $wid (@ary1) {
INNER:   for my $jet (@ary2) {
            next OUTER if $wid > $jet;
            $wid += $jet;
         }
      }
```

Видите, насколько это проще? Это чище, безопаснее и быстрее. Это чище, потому что меньше шума. Это безопаснее, потому что, если код будет добавлен между внутренним и внешним циклами позже, новый код не будет выполнен случайно. `next` явно повторяет другой цикл, а не просто завершает внутренний. И это быстрее, потому что Perl выполняет `foreach` инструкцию быстрее, чем это было бы в эквивалентном цикле в стиле C `for` .

Проницательные хакеры Perl, возможно, заметили, что `for` цикл имеет возвращаемое значение, и что это значение можно получить, заключив цикл в `do` блок. Наградой за это открытие является следующий предостерегающий совет: возвращаемое значение `for` цикла не указано и может измениться без предварительного уведомления. Не полагайтесь на это.

Попробуйте обработать исключение Catch

Синтаксис `try` / `catch` обеспечивает поток управления, связанный с обработкой исключений. Ключевое слово `try` вводит блок, который будет выполняться при его обнаружении, и `catch` блок предоставляет код для обработки любого исключения, которое может быть вызвано первым.

```
try {
    my $x = call_a_function();
    $x < 100 or die "Too big";
    send_output($x);
}
catch ($e) {
    warn "Unable to output a value; $e";
}
print "Finished\n";
```

Here, the body of the `catch` block (i.e. the `warn` statement) will be executed if the initial block invokes the conditional `die`, or if either of the functions it invokes throws an uncaught exception. The `catch` block can inspect the `$e` lexical variable in this case to see what the exception was. If no exception was thrown then the `catch` block does not happen. In either case, execution will then continue from the following statement - in this example the `print`.

The `catch` keyword must be immediately followed by a variable declaration in parentheses, which introduces a new variable visible to the body of the subsequent block. Inside the block this variable will contain the exception value that was thrown by the code in the `try` block. It is not necessary to use the `my` keyword to declare this variable; this is implied (similar as it is for subroutine signatures).

Как в `try`, так и в `catch` блоках разрешено содержать выражения потока управления, такие как `return`, `goto` или `next` / `last` / `redo`. Во всех случаях они ведут себя так, как ожидалось, без предупреждений. В частности, `return` выражение внутри `try` блока возвращает всю содержащуюся в нем функцию - это контрастирует с его поведением внутри `eval` блока, где оно возвращает только этот блок.

Как и другой синтаксис потока управления, `try` and `catch` выдает последнее вычисленное значение при размещении в качестве последнего оператора в функции или `do` блоке. Это позволяет использовать синтаксис для создания значения. В этом случае помните, что не следует использовать `return` выражение, иначе это приведет к возврату содержащей функции.

```
my $value = do {
    try {
        get_thing(@args);
    }
    catch ($e) {
        warn "Unable to get thing - $e";
        $DEFAULT_THING;
    }
};
```

Как и в случае с другим синтаксисом потока управления, `try` блоки не видны `caller()` (так же, как, например, не видны циклы `while` или `foreach`). Последующие уровни `caller` результата могут видеть вызовы подпрограмм и `eval` блоки, потому что они влияют на то, как `return` будет работать. Поскольку `try` блоки не перехватываются `return`, они не представляют интереса для `caller`.

За блоками `try` и `catch` необязательно может следовать третий блок, вводимый с помощью `finally` ключевого слова. Этот третий блок выполняется после завершения остальной части конструкции.

```
try {
    call_a_function();
}
catch ($e) {
    warn "Unable to call; $e";
}
finally {
    print "Finished\n";
}
```

`finally` Блок эквивалентен использованию `defer` блока и будет вызываться в тех же ситуациях; независимо от того, завершается ли `try` блок успешно, создает исключение или передает управление в другое место с помощью `return`, элемента управления циклом или `goto`.

В отличие от блоков `try` и `catch`, `finally` блоку не разрешается `return`, `goto` или использовать какие-либо элементы управления циклом. Конечное значение выражения игнорируется и не влияет на возвращаемое значение содержащей функции, даже если оно помещено последним в функцию.

В настоящее время этот синтаксис является экспериментальным и должен быть включен с помощью `use feature 'try'`. Он выдает предупреждение в категории `experimental::try`.

Основные блоки

БЛОК сам по себе (помеченный или нет) семантически эквивалентен циклу, который выполняется один раз. Таким образом, вы можете использовать любой из операторов управления циклом в нем, чтобы выйти из блока или перезапустить его. (Обратите внимание, что это *НЕ* верно в `eval{}`, `sub{}` или вопреки распространенному мнению `do{}` блоках, которые *НЕ* считаются циклами.) `continue` Блок необязателен.

Блочная конструкция может использоваться для эмуляции регистровых структур.

```
SWITCH: {
    if (/^abc/) { $abc = 1; last SWITCH; }
    if (/^def/) { $def = 1; last SWITCH; }
    if (/^xyz/) { $xyz = 1; last SWITCH; }
    $nothing = 1;
}
```

Вы также обнаружите, что `foreach` цикл используется для создания топикализатора и переключателя:

```
SWITCH:
for ($var) {
    if (/^abc/) { $abc = 1; last SWITCH; }
    if (/^def/) { $def = 1; last SWITCH; }
    if (/^xyz/) { $xyz = 1; last SWITCH; }
    $nothing = 1;
}
```

Такие конструкции используются довольно часто, как потому, что в старых версиях Perl не было официального `switch` заявления, так и потому, что новая версия, описанная непосредственно ниже, остается экспериментальной и иногда может сбивать с толку.

отложенные блоки

Блок с префиксом модификатора `defer` предоставляет раздел кода, который выполняется позже при выходе из области видимости.

`defer` Блок может появиться в любой точке, где разрешен обычный блок или другая инструкция. Если поток выполнения достигает этой инструкции, тело блока сохраняется на потом, но не вызывается немедленно. Когда поток управления по какой-либо причине покидает содержащий блок, этот сохраненный блок выполняется по пути мимо. Это обеспечивает средство отсрочки выполнения до более позднего времени. Это работает аналогично синтаксису, предоставляемому некоторыми другими языками, часто с использованием ключевых слов с именами `try` / `finally` .

Этот синтаксис доступен, если он включен `defer` указанной функцией, и в настоящее время является экспериментальным. Если включены экспериментальные предупреждения, при использовании он будет выдавать предупреждение.

```
use feature 'defer';

{
    say "This happens first";
    defer { say "This happens last"; }

    say "And this happens inbetween";
}
```

Если несколько `defer` блоков содержатся в одной области видимости, они выполняются в порядке LIFO; последний достигнутый блок выполняется первым.

Код, хранящийся в `defer` блоке, будет вызван, когда элемент управления покинет содержащий его блок из-за регулярного аварийного завершения, явных `return` исключений, создаваемых `die` или распространяемых вызываемыми им функциями, `goto` или любыми операторами управления циклом `next` , `last` или `redo` .

Если поток управления не достигает самого `defer` оператора, то его тело не сохраняется для последующего выполнения. (Это прямо противоположно коду, предоставляемому `END` блоком phaser, который всегда помещается компилятором в очередь, независимо от того, достигало ли выполнение когда-либо строки, в которой он был задан.)

```
use feature 'defer';

{
    defer { say "This will run"; }
    return;
    defer { say "This will not"; }
}
```

Исключения, создаваемые кодом внутри `defer` блока, будут передаваться вызывающей стороне таким же образом, как и любое другое исключение, создаваемое обычным кодом.

Если `defer` блок выполняется из-за сгенерированного исключения и выдает другое, не указано, что происходит, кроме того, вызывающий объект обязательно получит исключение.

Помимо создания исключения, `defer` блоку не разрешается иным образом изменять поток управления окружающим его кодом. В частности, он может не вызывать содержащуюся в нем функцию `return` , также он не может `goto` создавать метку или управлять содержащим циклом с помощью `next` , `last` или `redo` . Однако эти конструкции полностью разрешены в теле `defer` .

```
use feature 'defer';

{
    defer {
        foreach ( 1 .. 5 ) {
            last if $_ == 3;      # this is permitted
        }
    }
}

{
    foreach ( 6 .. 10 ) {
        defer {
            last if $_ == 8;      # this is not
        }
    }
}
```

Начиная с Perl 5.10.1 (ну, 5.10.0, но это сработало неправильно), вы можете сказать

```
use feature "switch";
```

чтобы включить экспериментальную функцию переключения. Это в общих чертах основано на старой версии предложения Raku, но больше не похоже на конструкцию Raku. Вы также получаете функцию переключения всякий раз, когда заявляете, что ваш код предпочитает выполняться под версией Perl между 5.10 и 5.34. Например:

```
use v5.14;
```

С помощью функции "переключения" Perl получает экспериментальные ключевые слова `given`, `when`, `default`, `continue` `break`, и,,,. Начиная с Perl 5.16, к ключевым словам switch можно добавить префикс `CORE::` для доступа к функции без `use feature` инструкции. Ключевые слова `given` и `when` аналогичны `switch` и `case` в других языках, хотя `continue` это не так, поэтому код в предыдущем разделе можно переписать как

```
use v5.10.1;
for ($var) {
    when (/^abc/) { $abc = 1 }
    when (/^def/) { $def = 1 }
    when (/^xyz/) { $xyz = 1 }
    default       { $nothing = 1 }
}
```

`foreach` - это неэкспериментальный способ настройки топикализатора. Если вы хотите использовать высокоэкспериментальный `given`, это можно записать следующим образом:

```
use v5.10.1;
given ($var) {
    when (/^abc/) { $abc = 1 }
    when (/^def/) { $def = 1 }
    when (/^xyz/) { $xyz = 1 }
    default       { $nothing = 1 }
}
```

Начиная с версии 5.14, это также можно записать таким образом:

```
use v5.14;
for ($var) {
    $abc = 1 when /^abc/;
    $def = 1 when /^def/;
    $xyz = 1 when /^xyz/;
    default { $nothing = 1 }
}
```

Или, если вы не хотите перестраховаться, вот так:

```
use v5.14;
given ($var) {
    $abc = 1 when /^abc/;
    $def = 1 when /^def/;
    $xyz = 1 when /^xyz/;
    default { $nothing = 1 }
}
```

Аргументы для given и when находятся в скалярном контексте и given присваивают $_ переменной ее тематическое значение.

Трудно точно описать, что именно делает аргумент *EXPR* для when , но в целом он пытается угадать, что вы хотите сделать. Иногда он интерпретируется как $_ ~~ *EXPR* , а иногда нет. Он также ведет себя иначе, когда лексически заключен в given блок, чем когда он динамически заключен в foreach цикл. Правила слишком сложны для понимания, чтобы описывать их здесь. Смотрите "Подробные сведения об эксперименте, когда" позже.

Из-за досадной ошибки в том, как given было реализовано между Perl 5.10 и 5.16, в этих реализациях версия $_ , управляемая given , является просто лексически ограниченной копией оригинала, а не динамически изменяемым псевдонимом оригинала, как это было бы, если бы это был foreach или как в оригинальной, так и в текущей спецификации языка Raku. Эта ошибка была исправлена в Perl 5.18 (а сам lexicalized $_ был удален в Perl 5.24).

Если ваш код по-прежнему должен выполняться в старых версиях, придерживайтесь foreach для вашего топикализатора, и вы будете менее недовольны.

Переход

Хотя Perl и не для слабонервных, он поддерживает оператор goto . Существует три формы: goto -LABEL, goto -EXPR и goto -&NAME. METKA цикла на самом деле не является допустимой целью для goto ; это просто название цикла.

Форма goto -LABEL находит инструкцию, помеченную LABEL, и возобновляет выполнение там. Его нельзя использовать для перехода к любой конструкции, требующей инициализации, такой как подпрограмма или foreach цикл. Его также нельзя использовать для перехода к конструкции, которая полностью оптимизирована. Его можно использовать практически для перехода в любое другое место динамической области, в том числе за пределы подпрограмм, но обычно лучше использовать какую-либо другую конструкцию, такую как last или die . Автор Perl никогда не чувствовал необходимости использовать эту форму goto (в Perl это другое дело - C).

Форма goto -EXPR ожидает имя метки, область видимости которой будет разрешена динамически. Это позволяет вычислять goto s для FORTRAN, но не обязательно рекомендуется, если вы оптимизируете для удобства обслуживания:

```
goto(("FOO", "BAR", "GLARCH")[$i]);
```

Форма goto -&NAME в высшей степени волшебна и заменяет вызов именованной подпрограммы для текущей запущенной подпрограммы. Это используется AUTOLOAD() подпрограммами, которые хотят загрузить другую подпрограмму, а затем притворяются, что другая подпрограмма была вызвана в первую очередь (за исключением того, что любые изменения @_ в текущей подпрограмме распространяются на другую подпрограмму.) После goto даже caller() не удастся определить, была ли эта процедура вызвана первой.

Почти во всех случаях, подобных этому, обычно намного, намного лучше использовать механизмы структурированного потока управления next , last или redo вместо того, чтобы прибегать к goto . Для определенных приложений пара catch и throw из eval{} и die() для обработки исключений также может быть разумным подходом.

Оператор с многоточием

Начиная с Perl 5.12, Perl принимает многоточие " ... " в качестве заполнителя для кода, который вы еще не реализовали. Когда Perl 5.12 или более поздней версии обнаруживает оператор с многоточием, он анализирует его без ошибок, но если и когда вы действительно должны попытаться выполнить его, Perl выдает исключение с текстом Unimplemented :

```
use v5.12;
sub unimplemented { ... }
eval { unimplemented() };
if ($@ =~ /^Unimplemented at /) {
    say "I found an ellipsis!";
}
```

Вы можете использовать эллиптический оператор только для замены полного оператора. Синтаксически " ...; " является завершенным оператором, но, как и в случае с другими типами операторов, заканчивающихся точкой с запятой, точка с запятой может быть опущена, если " ... " появляется непосредственно перед закрывающей фигурной скобкой. Эти примеры показывают, как работает многоточие:

```
use v5.12;
{ ... }
sub foo { ... }
...;
eval { ... };
sub somemeth {
    my $self = shift;
    ...;
}
$x = do {
    my $n;
    ...;
    say "Hurrah!";
    $n;
};
```

Оператор с многоточием не может заменять выражение, являющееся частью более крупного оператора. Эти примеры попыток использовать многоточие являются синтаксическими ошибками:

```
use v5.12;

print ...;
open(my $fh, ">", "/dev/passwd") or ...;
if ($condition && ... ) { say "Howdy" };
... if $a > $b;
say "Cromulent" if ...;
$flub = 5 + ...;
```

В некоторых случаях Perl не может сразу определить разницу между выражением и оператором. Например, синтаксис блока и анонимного конструктора ссылок на хэш выглядят одинаково, если в фигурных скобках нет чего-либо, что дает Perl подсказку. Многоточие является синтаксической ошибкой, если Perl не догадывается, что `{ ... }` является блоком. Внутри вашего блока вы можете использовать `;` перед многоточием, чтобы обозначить, что `{ ... }` это блок, а не конструктор ссылок на хэш.

Примечание: Некоторые люди в разговорной речи называют этот знак препинания "бла-бла-бла" или "тройная точка", но на самом деле его истинное название - многоточие.

Модули: встроенная документация

В Perl есть механизм для смешивания документации с исходным кодом. В то время как он ожидает начало нового оператора, если компилятор обнаруживает строку, начинающуюся со знака равенства и слова, подобного этому

```
=head1 Here There Be Pods!
```

Тогда этот текст и весь оставшийся текст вплоть до строки, начинающейся с `=cut`, включая строку, начинающуюся с. Формат промежуточного текста описан в perlpod.

Это позволяет вам свободно смешивать исходный код и текст документации, как в

```
=item snazzle($)

The snazzle() function will behave in the most spectacular
form that you can possibly imagine, not even excepting
cybernetic pyrotechnics.

=cut back to the compiler, nuff of this pod stuff!

sub snazzle($) {
    my $thingie = shift;
    .........
}
```

Обратите внимание, что переводчики pod должны просматривать только абзацы, начинающиеся с директивы pod (это упрощает синтаксический анализ), тогда как компилятор на самом деле знает, что нужно искать экранирующие элементы pod даже в середине абзаца. Это означает, что следующий секретный материал будет проигнорирован как компилятором, так и трансляторами.

```
$a=3;
=secret stuff
 warn "Neither POD nor CODE!?"
=cut back
print "got $a\n";
```

Вероятно, вам не стоит полагаться на то, что `warn()` будет удален навсегда. Не все pod-трансляторы хорошо себя ведут в этом отношении, и, возможно, компилятор станет более разборчивым.

Можно также использовать директивы pod для быстрого закомментирования раздела кода.

Простые старые комментарии (нет!)

Perl может обрабатывать строковые директивы, во многом подобно препроцессору C. Используя это, можно управлять представлением Perl об именах файлов и номерах строк в сообщениях об ошибках или предупреждениях (особенно для строк, которые обрабатываются с помощью `eval()`). Синтаксис этого механизма почти такой же, как у большинства препроцессоров C: он соответствует регулярному выражению

```
# example: '# line 42 "new_filename.plx"'
/^\#   \s*
  line \s+ (\d+)   \s*
  (?:\s("?)([^"]+)\g2)? \s*
 $/x
```

где `$1` - номер строки для следующей строки, а `$3` - необязательное имя файла (указывается в кавычках или без них). Обратите внимание, что перед `#` не должно быть пробелов, в отличие от современных препроцессоров C.

В директиву `line` включена довольно очевидная ошибка: отладчики и профилировщики будут показывать только последнюю исходную строку с определенным номером строки в данном файле. Следует соблюдать осторожность, чтобы не вызвать коллизии номеров строк в коде, который вы хотите отладить позже.

Вот несколько примеров, которые вы должны иметь возможность ввести в свою командную оболочку:

```
% perl
# line 200 "bzzzt"
# the '#' on the previous line must be the first char on line
die 'foo';
__END__
foo at bzzzt line 201.

% perl
# line 200 "bzzzt"
eval qq[\n#line 2001 ""\ndie 'foo']; print $@;
__END__
foo at - line 2001.

% perl
eval qq[\n#line 200 "foo bar"\ndie 'foo']; print $@;
__END__
foo at foo bar line 200.

% perl
# line 345 "goop"
eval "\n#line " . __LINE__ . ' "' . __FILE__ ."\"\ndie 'foo'";
print $@;
__END__
foo at goop line 345.
```

Подробности эксперимента по заданию и когда

Как упоминалось ранее, функция "переключения" считается экспериментальной (ее также планируется удалить в perl 5.42.0); она может быть изменена без предварительного уведомления. В частности, `when` имеет сложное поведение, которое, как ожидается, изменится и станет менее сложным в будущем. Не полагайтесь на его текущую (неправильную) реализацию. До Perl 5.18 `given` также имел сложное поведение, которого вам все равно следует остерегаться, если ваш код должен выполняться на более старых версиях Perl.

Вот более длинный пример `given`:

```
use feature ":5.10";
given ($foo) {
    when (undef) {
        say '$foo is undefined';
    }
    when ("foo") {
        say '$foo is the string "foo"';
    }
    when ([1,3,5,7,9]) {
        say '$foo is an odd digit';
        continue; # Fall through
    }
    when ($_ < 100) {
        say '$foo is numerically less than 100';
    }
    when (\&complicated_check) {
        say 'a complicated check for $foo is true';
    }
    default {
        die q(I don't know what to do with $foo);
    }
}
```

До версии Perl 5.18, `given(EXPR)` присваивалось значение *EXPR* просто ***копии*** с лексическим расширением `$_` (!) `foreach`, а не псевдониму с динамическим расширением, как это делается. Это сделало его похожим на

```
do { my $_ = EXPR; ... }
```

за исключением того, что блок был автоматически разорван успешным `when` или явным `break`. Поскольку это была всего лишь копия и поскольку она имела только лексическую, а не динамическую область видимости, вы не могли делать с ней то, к чему вы привыкли в `foreach` цикле. В частности, он не работал для произвольных вызовов функций, если эти функции могли пытаться получить доступ к `$_`. Для этого лучше придерживаться `foreach`.

Большая часть возможностей исходит от неявного интеллектуального сопоставления, которое иногда может применяться. Большую часть времени `when(EXPR)` обрабатывается как неявное интеллектуальное сопоставление `$_`, то есть `$_ ~~ EXPR`. (См. "Оператор интеллектуального сопоставления" в perlop для получения дополнительной информации о интеллектуальном сопоставлении.) Но когда *EXPR* является одним из 10 исключительных случаев (или похожих на них), перечисленных ниже, он используется непосредственно как логическое значение.

1. Вызов пользовательской подпрограммы или метода.

2. Соответствие регулярному выражению в форме `/REGEX/`, `$foo =~ /REGEX/` или `$foo =~ EXPR`. Кроме того, отрицаемое регулярное выражение совпадает в форме `!/REGEX/`, `$foo !~ /REGEX/` или `$foo !~ EXPR`.

3. Интеллектуальное сопоставление, использующее явный `~~` оператор, такой как `EXPR ~~ EXPR`.

   ПРИМЕЧАНИЕ: Вам часто придется использовать `$c ~~ $_` потому что по умолчанию используется регистр `$_ ~~ $c`, который часто противоположен тому, что вы хотите.

4. Логический оператор сравнения, такой как `$_ < 10` или `$x eq "abc"`. Реляционные операторы, что это касается шесть сравнение (`<`, `>`, `<=`, `>=`, `==`, и `!=`), и шесть сравнения строк (`lt`, `gt`, `le`, `ge`, `eq`, и `ne`).

5. По крайней мере, три встроенные функции `defined(...)`, `exists(...)` и `eof(...)`. Возможно, когда-нибудь мы добавим еще что-нибудь из них, если подумаем о них.

6. Отрицаемое выражение, будь то `!(EXPR)` или `not(EXPR)`, или логическое исключающее-или, `(EXPR1) xor (EXPR2)`. Побитовые версии (`~` и `^`) не включены.

7. Оператор filetest ровно с 4 исключениями: `-s`, `-M`, `-A`, и `-C`, поскольку они возвращают числовые значения, а не логические. Оператор `-z` filetest не включен в список исключений.

8.

Операторы `..` и `...` триггера. Обратите внимание, что `...` оператор триггера полностью отличается от `...` только что описанного эллиптического оператора.

В этих 8 приведенных выше случаях значение EXPR используется непосредственно как логическое значение, поэтому интеллектуальное сопоставление не выполняется. Вы можете думать о `when` как о smartsmartmatch .

Кроме того, Perl проверяет операнды логических операторов, чтобы решить, использовать ли интеллектуальное сопоставление для каждого из них, применяя приведенный выше тест к операндам:

9.

If EXPR is `EXPR1 && EXPR2` or `EXPR1 and EXPR2` , the test is applied *recursively* to both EXPR1 and EXPR2. Only if *both* operands also pass the test, *recursively*, will the expression be treated as boolean. Otherwise, smartmatching is used.

10.

Если EXPR имеет значение `EXPR1 || EXPR2` , `EXPR1 // EXPR2` или `EXPR1 or EXPR2` , тест применяется *рекурсивно* только к EXPR1 (который сам по себе может быть оператором AND с более высоким приоритетом, например, и, следовательно, подчиняется предыдущему правилу), но не к EXPR2. Если EXPR1 должен использовать интеллектуальное сопоставление, то EXPR2 также делает это, независимо от того, что содержит EXPR2. Но если EXPR2 не может использовать smartmatching, то второй аргумент также не будет использоваться. Это сильно отличается от `&&` только что описанного случая, поэтому будьте осторожны.

Эти правила сложны, но цель состоит в том, чтобы они выполняли то, что вы хотите (даже если вы не совсем понимаете, почему они это делают). Например:

```
when (/^\d+$/ && $_ < 75) { ... }
```

будет обрабатываться как логическое соответствие, потому что в правилах указано, что и совпадение регулярных выражений, и явный тест на `$_` будут обрабатываться как логические значения.

Также:

```
when ([qw(foo bar)] && /baz/) { ... }
```

будет использоваться интеллектуальное сопоставление, потому что только *один* из операндов является логическим значением: другой использует интеллектуальное сопоставление, и это выигрывает.

Далее:

```
when ([qw(foo bar)] || /^baz/) { ... }
```

будет использовать интеллектуальное сопоставление (учитывается только первый операнд), тогда как

```
when (/^baz/ || [qw(foo bar)]) { ... }
```

будет тестироваться только регулярное выражение, которое заставляет оба операнда обрабатываться как логические. Тогда обратите внимание на это, потому что arrayref всегда является истинным значением, что делает его фактически избыточным. Не очень хорошая идея.

Тавтологичные логические операторы по-прежнему будут оптимизированы. Не поддавайтесь искушению писать

```
when ("foo" or "bar") { ... }
```

Это будет оптимизировано до `"foo"` , поэтому `"bar"` никогда не будет рассматриваться (хотя в правилах указано использовать smartmatch на `"foo"`). Для подобного чередования будет работать ссылка на массив, потому что это вызовет интеллектуальное сопоставление:

```
when ([qw(foo bar)] { ... }
```

Это в некоторой степени эквивалентно сквозной функциональности оператора switch в стиле C (не путать с сквозной функциональностью *Perl* - см. Ниже), в которой один и тот же блок используется для нескольких `case` инструкций.

Другой полезный ярлык заключается в том, что, если вы используете литеральный массив или хэш в качестве аргумента для `given` , он превращается в ссылку. Так `given(@foo)` это то же самое, что `given(\@foo)` , например.

`default` ведет себя точно так же, как `when(1 == 1)` , то есть всегда совпадает.

Прорыв

Вы можете использовать `break` ключевое слово, чтобы вырваться из окружающего `given` блока. Каждый `when` блок неявно заканчивается `break` .

Аварийный вариант

Вы можете использовать `continue` ключевое слово для немедленного перехода от одного обращения к следующему `when` или `default`:

```
given($foo) {
    when (/x/) { say '$foo contains an x'; continue }
    when (/y/) { say '$foo contains a y'             }
    default    { say '$foo does not contain a y'     }
}
```

Возвращаемое значение

Когда оператор `given` также является допустимым выражением (например, когда это последний оператор блока), он принимает значение:

- Пустой список, как только встречается явное `break`.
- Значение последнего вычисленного выражения успешного предложения `when` / `default`, если таковое имеется.
- Значение последнего вычисленного выражения `given` блока, если ни одно условие не выполняется.

В обоих последних случаях последнее выражение вычисляется в контексте, который был применен к `given` блоку.

Обратите внимание, что, в отличие от `if` и `unless`, операторы с ошибкой `when` всегда выдают пустой список.

```
my $price = do {
    given ($item) {
        when (["pear", "apple"]) { 1 }
        break when "vote";       # My vote cannot be bought
        1e10  when /Mona Lisa/;
        "unknown";
    }
};
```

В настоящее время `given` блоки не всегда можно использовать как правильные выражения. Это может быть исправлено в будущей версии Perl.

Переключение в цикле

Вместо использования `given()` вы можете использовать `foreach()` цикл. Например, вот один из способов подсчитать, сколько раз определенная строка встречается в массиве:

```
use v5.10.1;
my $count = 0;
for (@array) {
    when ("foo") { ++$count }
}
print "\@array contains $count copies of 'foo'\n";
```

Или в более поздней версии:

```
use v5.14;
my $count = 0;
for (@array) {
    ++$count when "foo";
}
print "\@array contains $count copies of 'foo'\n";
```

В конце всех `when` блоков есть неявное значение `next`. Вы можете переопределить это явным значением `last`, если вас интересует только первое совпадение.

Это не сработает, если вы явно укажете переменную цикла, как в `for $item (@array)`. Вы должны использовать переменную по умолчанию `$_`.

Отличия от Raku

Конструкции Perl 5 smartmatch и `given` / `when` несовместимы с их аналогами Raku. Наиболее заметное отличие и наименее важное отличие заключается в том, что в Perl 5 требуется заключать в круглые скобки аргумент to `given()` и `when()` (за исключением случаев, когда этот последний используется в качестве модификатора оператора). Круглые скобки в Raku всегда необязательны в конструкции управления, такой как `if()`, `while()` или `when()`; их нельзя сделать необязательными в Perl 5 без большой потенциальной путаницы, потому что Perl 5 будет анализировать выражение

```
given $foo {
    ...
}
```

как если бы аргумент в `given` был элементом хэша `%foo`, интерпретируя фигурные скобки как синтаксис хэш-элемента.

Однако у них есть много-много других отличий. Например, это работает в Perl 5:

```
use v5.12;
my @primary = ("red", "blue", "green");

if (@primary ~~ "red") {
    say "primary smartmatches red";
}

if ("red" ~~ @primary) {
    say "red smartmatches primary";
}

say "that's all, folks!";
```

Но в Raku это вообще не работает. Вместо этого вам следует использовать оператор (распараллеливаемый) `any`:

```
if any(@primary) eq "red" {
    say "primary smartmatches red";
}

if "red" eq any(@primary) {
    say "red smartmatches primary";
}
```

Таблица smartmatches в "Операторе Smartmatch" в perlop не идентична таблице, предложенной спецификацией Raku, в основном из-за различий между моделями данных Raku и Perl 5, но также и потому, что спецификация Raku изменилась с тех пор, как Perl 5 был быстро внедрен.

В Raku `when()` всегда будет выполнять неявное интеллектуальное сопоставление со своим аргументом, в то время как в Perl 5 удобно (хотя и потенциально запутанно) подавлять это неявное интеллектуальное сопоставление в различных довольно слабо определенных ситуациях, как в общих чертах описано выше. (Разница во многом заключается в том, что Perl 5 не имеет, даже внутренне, логического типа.)