# Новое в синтаксисе Perl 5.10

Андрей Шитов

**18 декабря исполнилось 20 лет с момента выхода первой версии Perl. Наиболее распространенная на сегодня версия – 5.8.8 – вышла в свет в начале 2006 года, при том что сама ветка 5.8 была начата еще почти четырьмя годами раньше, летом 2002-го. В этот же день 18 декабря в 2007 году произошло два важных события: выпущена новая стабильная версия Perl 5.10 [1], которая, по-видимому, станет стандартом на переходный период до создания полноценного компилятора Perl 6, и новая версия виртуальной машины Parrot [2], в составе которой содержится заново переписанный экспериментальный компилятор Perl 6.**

В этой статье описаны основные нововведения, которые появились в Perl 5.10. Читателя не должно смущать, что вместо шестой версии продолжает развиваться пятая. В версии Perl 5.10 включены некоторые операторы, которые как раз и появились при разработке дизайна Perl 6.

## Установка

Новой версии предшествовали две предварительные: RC1 и RC2. Тем не менее для экспериментов с новым языком предпочтительнее установить дистрибутив в отдельный каталог /perl5.10 на UNIX-системах. Пользователям Windows проще воспользоваться готовым инсталлятором ActivePerl 5.10, выпущенным компанией ActiveState.

Процесс установки из исходных кодов стандартен; для упомянутого каталога установки необходимо выполнить следующие команды:

```
./Configure -Dprefix=/perl5.10
make
make install
```

Конфигурационный скрипт традиционно задает множество вопросов. Чтобы их избежать, можно при запуске Configure указать опции -des, чтобы выбрать все значения по умолчанию.

Для удобства экспериментов полезно включить путь к исполнимому файлу perl5.10 в переменную окружения PATH. После успешной установки команда perl5.10 -v должна напечатать информацию о версии; в моем случае вывелось следующее:

```
This is perl, v5.10.0 built for darwin-2level
Copyright 1987-2007, Larry Wall
```

## use feature

Для некоторых новых возможностей потребовалось ввести в язык новые ключевые слова. Поэтому, для того чтобы обеспечить максимально возможную обратную совместимость, необходимо сообщить компилятору о том, что необходимо включить поддержку этих слов. Для их активации предусмотрена инструкция use feature, которая может принимать список идентификаторов. В Perl 5.10 существует три случая, когда это необходимо делать:

```
use feature say;
use feature state;
use feature switch;
```

Кроме того, возможно объединять несколько инструкций в одну строку, например:

```
use feature qw(say state switch);
```

Каждая из этих возможностей детально описана далее.

Чтобы подключить сразу все новые ключевые слова, можно воспользоваться одной из нескольких инструкций, указывающих версию языка:

```
use 5.10.0;
use v5.10;
use feature ":5.10";
```

И, наконец, при выполнении кода из командной строки можно использовать ключ -E вместо традиционного -e.

Важно иметь в виду, что инструкция use feature имеет лексическую область видимости и поэтому ее можно использовать, чтобы включить новые ключевые слова лишь в пределах текущего блока кода. Кроме того, существует инструкция с противоположным действием no feature, которая отменяет соответствующие расширения.

### use feature 'say'

Инструкция use feature 'say' подключает одну-единственную функцию say, которая выполняет те же действия, что и print, но делает перевод строки после вывода списка переданных аргументов. Допустимо также передать файловый дескриптор.

Например:

```
say 2007;
say 'London LHR';
say $air_carrier;
say $weekday_name[($departure_day + 1) % 7];
say $printer "flight coupon No. $current of $total";
```

Функция say в Perl 5.10 частично повторяет поведение одноименной функции из Perl 6, однако есть несоответствия, вызванные тем, что в Perl 5.10 приходится соблюдать соглашения, принятые в более младших версиях.

Различие проявляется в двух случаях. Во-первых, невозможно использовать вызов say как метод некоторого объекта, как это допустимо в Perl 6: $day.say();. Кроме того, в Perl 5.10 функция, вызванная без аргументов, продолжает принимать переменную по умолчанию $_. Поэтому один и тот же код:

```
say for 1..3;
```

даст разные результаты. В Perl 5.10 будут напечатаны три строки с числами от одного до трех, а в Perl 6 всего лишь три перевода строк (чтобы использовать переменную по умолчанию, нужно записать либо $_.say, либо .say).

### use feature 'state'

Ключевое слово state, которое становится доступным после инструкции use featuer 'state', позволяет создавать переменные с лексической областью видимости, которые, однако, сохраняют свое значение даже после выхода программы из области видимости этой переменной. Как только выполнение программы вновь окажется в той же области, переменная будет снова доступна с прежним значением.

Например, переменные, объявленные как state, могут быть использованы для подсчета числа вызовов функции, для генерации последовательных номеров, для подсчета числа созданных объектов или для накопления статистики.

Два примера. В первом из них функция next_serial() при каждом вызове возвращает последовательно увеличивающиеся номера.

```
sub next_serial {
    state $serial = 0;
    return ++$serial;
}

say next_serial();
say next_serial();
say next_serial();
```

Во втором примере вычисляется среднее значение случайной последовательности чисел: для каждого нового значения вызывается функция register_value(), которая накапливает сумму всех полученных величин и их число.

```
sub register_value {
    state $sum = 0;
    state $num = 0;

    my $value = shift;
```

```
    $sum += $value;
    $num++;

    return $num ? $sum / $num : 0;
}

for (1..1000) {
    say register_value (rand 10);
}
```

При большом числе повторов программа в итоге начинает печатать среднее значение, близкое к пяти, как и ожидалось.

Обратите внимание, что инициализация статических переменных происходит в момент объявления и выполняется однократно.

## use feature 'switch'

С одной стороны, название switch говорит само за себя, с другой, не совпадает ни с одним новым ключевым словом. Инструкция use feature 'switch' вводит в обращение ключевые слова given, when и default, которые предназначены для реализации блоков выбора. Синтаксис почти совпадает с принятым в Perl 6, за тем исключением, что в Perl 5.10 по-прежнему необходимы круглые скобки вокруг условия.

Работа блока выбора given/when напоминает традиционный блок switch в других языках программирования, однако имеет больше возможностей при анализе аргумента. Блок выбора открывается ключевым словом given, отдельные ветви словом when, а действие по умолчанию – default. Например:

```
given ($day) {
    when (6) {say 'Saturday'}
    when (7) {say 'Sunday'}
    default  {say 'Weekday'}
}
```

В отличие от C и C++, после найденного совпадения выполняется только один блок кода, и «проваливания» в следующие блоки when не происходит. Первый успешный when выполняет соответствующий блок кода и прекращает выполнение блока given, не допуская, таким образом, остальные попытки проверить условия в блоках when, записанных ниже.

Если поведение требуется изменить, следует указать это инструкцией continue в конце соответствующего блока.

```
given ($day) {
    when ($today) {say 'Today'; continue}
    when (6)      {say 'Saturday'}
    when (7)      {say 'Sunday'}
    default       {say 'Weekday'}
}
```

Аргументом в ветвях when может быть не только константа или переменная. Там, где в программе встречается вызов when, происходит сопоставление переменной $_ с выражением, которое является аргументом when(). Явно указывать эту переменную обычно не требуется, поскольку она автоматически устанавливается при входе в блок given.

Сам по себе блок given не обязателен, чтобы вызвать when. Например, допустимо воспользоваться циклом for, которые тоже устанавливает переменную по умолчанию $_:

```
for ('a'..'z') {
    when (/[aeiou]/) {say "$_ is vowel"}
}
```

Обратите внимание на два момента (помимо того, что when используется внутри for). Во-первых, в блоке кода использована переменная $_. Во-вторых, в условии when записано регулярное выражение, которое сопоставляется с переменной $_. Показанный цикл печатает сообщение для каждой гласной буквы.

В некоторых случаях необходимо явно указывать переменную по умолчанию, например: when ($_ > 0) или when (test_me ($_)). В последнем примере возможно также передать ссылку на функцию: when (\&test_me).

Действия, выполняемые в каждой ветви when, зависят от типов переменных, участвующих в сравнении. Подробнее об этом рассказано в следующем разделе.

В частности, условие when (5) внутри блока given ($day) эквивалентно проверке if ($day == 5), а when (2 + $period) – if ($day == 2 + $period). Но если в условии when записан вызов функции или передана ссылка на нее, то сравнения исходного значения с результатом функции не происходит: указанная в заголовке блока given переменная передается этой функции как аргумент, а решение об успешности текущей ветви when принимается на основании значения, возвращаемого функцией. Это значение интерпретируется как булево, то есть если функция вернула ненулевое значение, блок when считается успешным.

В следующих двух блоках given происходит вызов функции func(), и поскольку она возвращает ненулевое значение 2 или 1, первая же проверка when оказывается успешной:

```
sub func {
    my $arg = shift;
    return $arg;
}

given (2) {
    when (\&func) {say "func"}
    when (2)      {say "2"}
}

given (2) {
    when (func(1)) {say "func"}
    when (2)       {say "2"}
}
```

Чтобы сравнить переменную с возвращаемым функцией значением, необходимо записать это явно:

```
given (2) {
    when ($_ == func(0)) {say "func"}
    when (2)             {say "2"}
}
```

Обратите также внимание, что значение, возвращаемое функцией, преобразуется в булево по традиционным правилам: например, строка «0» считается ложью, а «0.0», «00» или «0E0» истиной.

## ~~ (smart matching)

Сопоставление, о котором было сказано в предыдущем разделе, на самом деле является не традиционным сопоставлением с регулярным выражением, а так называемым smart matching, которое имеет собственный символ в грамматике языка: ~~.

Предыдущие примеры можно было бы переписать, явно используя переменную по умолчанию, и оператор ~~, которые неявно подразумеваются при обращении к when:

```
given ($day) {
    when ($_ ~~ $today) {say 'Today'; continue}
    when ($_ ~~ 6) {say 'Saturday'}
    when ($_ ~~ a7) {say 'Sunday'}
    default  {say 'Weekday'}
}

for ('a'..'z') {
    when ($_ ~~ /[aeiou]/) {say "$_ is vowel"}
}
```

Оператор ~~ всегда коммутативен, то есть выражения $a ~~ $b и $b ~~ $a дают одинаковый результат.

Кроме того, оператор ~~ допускает в качестве аргументов не только переменные, константы или регулярные выражения. Например:

```
if ($x ~~ 100)        {say "constant"}
if ($x ~~ /\d+/)      {say "regexp"}
if ($x ~~ $y)         {say "variable"}
if ($x ~~ [50..150]) {say "array ref"}
if ($x ~~ @a)         {say "array"}
unless ($x ~~ undef) {say "undef"}
```

Название smart matching обусловлено тем, что оператор ~~ самостоятельно выбирает, как именно сравнивать аргументы, основываясь на их типе.

Следующая программа демонстрирует примеры эквивалентных проверок, либо не использующих оператор ~~ вовсе, либо сводящихся к использованию более простого варианта. Массив @test содержит попарные инструкции, которые при выполнении дают одинаковый результат (в большинстве случаев использование smart matching позволяет написать более короткий и прозрачный код). Тесты выполняются автоматически в цикле do {...} while @test, для каждого из них выводится сообщение ok или not ok в зависимости от того, успешно было сопоставление или нет. Эквивалентные пары дают одинаковый результат, однако следует иметь в виду, что фактическая реализация не обязательно совпадает с кодом во втором столбце. В частности, компилятор делает оптимизации, чтобы не вычислять величины, которые не повлияют на результат.

```
use v5.10;

my $a = 1;
my $b;
my $c = 'abc';

my @a = (1..3);
my @b = (1..3);
my @c = (3..5);

my @d = (123, 'abc');
my @e = (qr/\d/, qr/\w/);

my @f = ('a'..'f');
my @g = (1..10);

my %h = (a => 'alpha', b => 'beta');
my $h_ref = \%h;

my %hh = (b => 1, a => 2);

sub subA {say "subA"; return 2}
sub subB {say "subB"; return 2}
sub subC {say "subC"; return 3}
sub subD {say shift; return 1}

my $subA1_ref = \&subA;
my $subA2_ref = \&subA;
my $subD_ref  = \&subD;

my @test = (
    '   $b ~~ undef      ', '   !defined $b                  ',
    '   $c ~~ "abc"      ', '   $c eq "abc"                  ',
    '   $c ~~ /b/        ', '   $c =~ /b/                    ',

    '   @a ~~ @b         ', '   1 == 1 && 2 == 2 && 3 == 3   ',
    '   @a ~~ @c         ', '   1 == 3 && 2 == 4 && 3 == 5   ',

    '   @d ~~ @e         ', '   123 ~~ /\d/ && "abc" ~~ /\w/ ',

    '   @f ~~ "d"        ', '   grep {$_ eq "d"} @f          ',

    '   @g ~~ 7          ', '   grep {$_ == 7} @g            ',
    '   @g ~~ /^\d$/     ', '   grep {$_ =~ /^\d\d$/} @g     ',

    '   3.14 ~~ "3.14"   ', '   3.14 == "3.14"               ',

    '   $subA1_ref ~~ $subA2_ref ',
                           '   $subA1_ref == $subA2_ref     ',
    '   subA() ~~ subB() ', '   subA() == subB()             ',
    '   subA() ~~ subC() ', '   subA() == subC()             ',

    '   $a ~~ $subA_ref  ', '   $subA_ref->()                ',
    '   -1 ~~ $subA_ref  ', '   $subA_ref->()                ',
    '   $c ~~ $subD_ref  ', '   $subD_ref->($c)              ',

    '   %h ~~ "a"        ', '   exists $h{"a"}               ',
    '   $h_ref ~~ "a"    ', '   exists $h_ref->{"a"}         ',
    '   %h ~~ /[A-F]/i   ', '   grep {/[A-F]/i} keys %h      ',
    '   %h ~~ %hh        ', '   [sort keys %h] ~~ [sort keys %hh] ',
);


do {
    my $smart_match = shift @test;
    my $equivalent  = shift @test;

    say eval $smart_match ? 'ok' : 'not ok', " $smart_match";
    say eval $equivalent  ? 'ok' : 'not ok', " $equivalent\n";
} while @test;
```

Результат выполнения этого скрипта содержит пары строк, начинающихся с ok или not ok, причем в пределах каждой пары результаты должны совпадать:

```
ok    $b ~~ undef
ok    !defined $b

ok    $c ~~ "abc"
ok    $c eq "abc"

ok    $c ~~ /b/
ok    $c =~ /b/

ok    @a ~~ @b
ok    1 == 1 && 2 == 2 && 3 == 3

not ok    @a ~~ @c
not ok    1 == 3 && 2 == 4 && 3 == 5

ok    @d ~~ @e
ok    123 ~~ /\d/ && "abc" ~~ /\w/

ok    @f ~~ "d"
ok    grep {$_ eq "d"} @f

ok    @g ~~ 7
ok    grep {$_ == 7} @g

ok    @g ~~ /^\d$/
ok    grep {$_ =~ /^\d\d$/} @g

ok    3.14 ~~ "3.14"
ok    3.14 == "3.14"

ok    $subA1_ref ~~ $subA2_ref
ok    $subA1_ref == $subA2_ref
```

```
subA
subB
ok    subA() ~~ subB()
subA
subB
ok    subA() == subB()

subA
subC
not ok    subA() ~~ subC()
subA
subC
not ok    subA() == subC()

not ok    $a ~~ $subA_ref
not ok    $subA_ref->()

not ok    -1 ~~ $subA_ref
not ok    $subA_ref->()

abc
ok    $c ~~ $subD_ref
abc
ok    $subD_ref->($c)

ok    %h ~~ "a"
ok    exists $h{"a"}

ok    $h_ref ~~ "a"
ok    exists $h_ref->{"a"}

ok    %h ~~ /[A-F]/i
ok    grep {/[A-F]/i} keys %h

ok    %h ~~ %hh
ok    [sort keys %h] ~~ [sort keys %hh]
```

Полезно также самостоятельно поэкспериментировать с показанным скриптом, записывая новые условия проверок [3].

## // и //=

Оператор //, называемый defined-or, пришел из Perl 6. Этот бинарный оператор возвращает значение левого операнда, если он определен, и правого в противоположном случае.

Важно отличать оператор // от ||. Запись «$a = $b // $c» эквивалентна «$a = defined $b ? $b : $c», в то время как «$a = $b || $c $a = $b ? $b : $c».

Поэтому если предположить, что переменная $a содержит нулевое значение, то выражение «$a // $b» всегда вернет 0, а «$a || $b» – значение переменной $b.

Для записи «$a = $a // $b» предусмотрен сокращенный вариант «$a //= $b».

## my $_

В Perl 5.10 переменную по умолчанию $_ возможно объявить локально, присвоив ей новое значение:

```
for (1..5) {
      my $_ = '*';
      print;
}
```

Этот код напечатает пять символов '*'. Чтобы обратиться к одноименной переменной, которая устанавливается при входе в цикл, следует указать имя пакета main, записав $main::$_ либо просто $::_.

Глобальную переменную $::_ возможно изменить с помощью объявления our $_. Следующий код тоже печатает пять звездочек:

```
for (1..5) {
    our $_ = '*';
    print $::_;
}
```

## _ в прототипах

При создании функций теперь возможно использовать символ подчеркивания для обозначения скалярного аргумента, который по умолчанию принимает значение переменной $_. Например, вызовы функции name() в показанном ниже цикле приведут к вызовам с аргументом $_.

```
sub name(_) {
    say shift;
}

for (1..3) {
    name();
}
```

Поскольку переменная, прототип которой объявлен с помощью символа «_», по определению не является обязательной, она должна стоять либо последней в списке, либо после точки с запятой, отделяющей обязательные формальные аргументы от необязательных.

# Регулярные выражения

В синтаксисе регулярных выражений, доступных в Perl 5.10, появилось много дополнений, которые могут существенно облегчить выполнение многих практических задач.

### Именованные сохраняющие скобки

Значения, попавшие в сохраняющие круглые скобки, по-прежнему хранятся в переменных типа $1, $2 и т. д. Однако теперь есть возможность в самом регулярном выражении присвоить каждой паре скобок имя, используя конструкцию (?<имя>...), например:

```
my $date = 'Tue 1 January 2008';
$date =~ /
    (?<wday>  \w+  ) \s+
    (?<day>   \d+  ) \s+
    (?<month> \w+  ) \s+
    (?<year>  \d{4})
/x;
```

Сохраненные значения доступны в массиве %+:

```
say $+{wday};
say $+{year};
```

Именованные значения сохраняются фактически как ссылки на соответствующие переменные $1, $2 и т. д. В программе допустимо одновременно использовать оба типа сохраняющих скобок.

Элементы массива %+ доступны и при использовании оператора s/// в выражении для замены:

```
$date =~ s/(?<year>\d{4})/$+{year} + 1/e;
```

### Обратные ссылки

Обратные ссылки (традиционно использующие для записи обратный слеш и номер сохраняющей пары) получили но-

вый синтаксис. Для именованных скобок обратная ссылка имеет вид \k<имя>. Например, следующий код использует обратные ссылки, чтобы отыскать в полученной строке определения переменных и заменить повторное использование переменной ее значением:

```
my $code = 'my $value = 100; say $value;';
$code =~ s/
    my                      \s*
    (?<variable> \$[a-z]+) \s*
    =                       \s*
    (?<value>    [^;]+   ) \s*
    ;                       \s*

    (?<other_code>.*?)

    \k<variable>
/${other_code}${value}/x;
```

Показанное регулярное выражение вначале отыскивает подстроку my $value = 100 и сохраняет подстроки $value и 100 соответственно в переменных $+{variable} и $+{value}, а затем ищет второе вхождение $value (этого требует метапоследовательность \k<variable>). После выполненной замены в переменной $code окажется строка say 100;.

Именованные сохраняющие скобки помимо удобства использования позволяют избегать неоднозначности, когда число скобок превышает 10: теперь нет необходимости использовать выражения вида \11.

Для обратных ссылок предусмотрен альтернативный метасимвол \g, после которого Perl ожидает номер сохраняющих скобок, причем для исключения неоднозначности номер рекомендуется заключать в фигурные скобки, например:

```
my $re = qr/
    (?<what> \w+) board
    .*?
    \g{1}
/x;

'keyboard made of keys' =~ $re;
say $+{what};

'snowboarding assumes snow' =~ $re;
say $+{what};
```

В приведенном примере метапоследовательность \g{1} полностью аналогична \g, \1, \k<what> и \g{what}.

Кроме того, \g принимает и отрицательные аргументы. В таком случае вся конструкция должна совпасть с сохраняющими скобками, нумерация которых начинается в текущем месте и идет к началу строки. Например, \g{-1} соответствуют предыдущим скобкам. В следующем фрагменте используется последовательность \g{-2}. Результат сопоставления со строками из предыдущего примера (обратите внимание на дополнительные скобки вокруг слова board):

```
my $re = qr/
    (?<what> \w+) (board)
    .*?
    \g{-2}
/x;
```

## Повторяющиеся имена

В том случае, когда в регулярном выражении встречается несколько одинаково поименованных сохраняющих скобок, доступ к совпавшим подстрокам возможен через хеш %-, в котором для каждого имени хранится ссылка на массив значений.

Например, следующий фрагмент выбирает из списка високосных лет два года, приходящихся на границу тысячелетия:

```
my $leap_years = '1992 1996 2004 2008';
$leap_years =~ /
    (?<year> 1 \d{3})
    \s*
    (?<year> 2 \d{3})
/x;
```

Здесь дважды встречается имя <year>, и элемент хеша $+{year} хранит первое совпавшее значение, в то время как в $-{year} появился массив, содержащий значения 1996 и 2004:

```
say $_ for @{$-{year}};
```

При использовании имени для сохраняющих скобок более чем один раз следует быть особенно осторожным, поскольку поведение программы в этом случае может отличаться от того, что, возможно, имел в виду программист.

В частности, модификатор g не добавляет новые элементы в хеш %-:

```
my $leap_years = '1992 1996 2004 2008';
$leap_years =~ m/(?<year>\d{4})/g;
say $_ for @{$-{year}};
```

Регулярное выражение /(?<year>\d{4})/g сохранит только первое найденное значение: 1992. Более того, поскольку для успешного совпадения достаточно найти только первый год, дальнейший поиск компилятор выполнять не обязан.

В том случае, если это же выражение используется для замены, Perl должен пройтись по всей строке и найти все четыре подходящих подстроки:

```
$leap_years =~ s/(?<year>\d{4})/*/g;
```

Строка теперь будет содержать значение «* * * *», а в полях $+{year} и @$-{year} сохранится только последнее значение 2008.

Несколько иное поведение будет при наличии квантификатора +:

```
$leap_years =~ m/(?<year>\d{4}\s*)+/g;
```

В этом случае и $+{year}, и @$-{year} содержат по одному последнему значению 2008.

## «Завладевающие» квантификаторы

Работа квантификаторов ?, *, + и {min, max} может быть изменена вторичным квантификатором + таким образом, что их поведение будет более чем «жадным»: дополнительный + запрещает выполнять откат, если выражение уже захватило какие-либо символы.

В частности, такое поведение удобно, чтобы сделать подсказку компилятору о том, что если совпадения не нашлось с первой попытки, не имеет смысла выполнять от-

каты и искать другие варианты. Например, для выделения выражения, заключенного в кавычки:

```
my $re = qr/
    (
        "
            (?:
                    [^"\\]++
                |
                    \\.
            )*+
        "
    )
/x;
```

Это выражение помещают в переменную $1 строку в кавычках, правильно обрабатывая экранированные кавычки \» внутри строки. Обратите внимание на два случая применения «завладевающих» квантификаторов.

## (?|...)

Шаблон (?|...), называемый branch reset, принуждает регулярное выражение заново начать нумерацию сохраняющих скобок в каждой ветви альтернативных подвыражений, записанных через символ “|”.

Например, регулярное выражение для чтения дат, записанных в разных форматах:

```
my $re = qr/
        (\d{4}) (\d\d) (\d\d)
    |
        (\w+) \s+ (\d+) , \s+ (\d+)
/x;
```

сохранит подстроки в переменных $1, $2 и $3 для строки '20080101', но для строки 'January 1, 2008' подстроки окажутся в переменных с другими номерами, а именно $4, $5 и $6.

Скобки (?|...), поставленные вокруг всего выражения с двумя ветками, начнут нумерацию с единицы и во втором случае:

```
my $re = qr/
    (?|
        (\d{4}) (\d\d) (\d\d)
    |
        (\w+) \s+ (\d+) , \s+ (\d+)
    )
/x;
```

Нужно проявлять осторожность, когда шаблон (?|...) используется совместно с именованными сохраняющими скобками. По возможности не следует повторять имена в разных ветвях регулярного выражения. Как упоминалось ранее, именованные переменные, хранимые в хешах %+ и %-, фактически являются ссылками на одни из нумерованных скобок. Поэтому, если одно и то же имя используется в скобках, номер которых не совпадает в разных подвыражениях, результат окажется не тем, какой ожидался.

## \K

Новый метасимвол \K сообщает, что при замене нужно отбросить часть выражения, находящуюся слева от \K, и заменить только то, что совпало справа.

Например, регулярное выражение $url =~ s/\bwww\.\w+\.\Ksu\b/ru/ заменяет доменную зону su зоной ru в адресах, начинающихся с www. Без метасимвола \K пришлось бы использовать, например, сохраняющие скобки и переменную $1, чтобы сохранить часть адреса, которую не требуется заменять.

## \h, \H, \v, \V и \R

Метасимволы \h и \v совпадают, соответственно, с горизонтальными и вертикальными пропусками. Пара \H и \V совпадает с тем, что не является горизонтальным или вертикальным пропуском.

Еще один новый полезный метасимвол \R совпадает с переводом строки, причем независимо от формата, принятого в операционной системе. Иными словами, теперь не придется писать громоздкие конструкции вида /\n\r?/, которые к тому же всегда приходится составлять, испытывая дискомфорт из-за опасения пропустить какой-нибудь формат.

Метасимвол \R не совпадет с неверной последовательностью \n\r:

```
my @strings = (
    "a\nb",   "a\rb",
    "a\r\nb", "a\n\rb"
);
for (@strings) {
    when (/a\Rb/) {say 'ok'    }
    default       {say 'not ok'}
}
```

Эта программа трижды напечатает «ok» для строк с допустимым форматом перевода строки и «not ok» для последней тестовой строки. (Обратите внимание на использование ключевых слов when и default внутри цикла for.)

## (?N)

В регулярных выражениях возможно рекурсивно использовать части выражения, заключенные в скобки, используя метапоследовательность (?N) и указав номер соответствующих скобок. Например:

```
'Mon-Fri' =~ /
    (?<from>
        Mon|Tue|Wed|Thu|Fri|Sat|Sun
    )
    -
    (?<to>
        (?1)
    )
/x;
```

Сокращенные названия дней недели перечислены только один раз, а для второго совпадения используется последовательность (?1), которая делает те же проверки. В переменных $+{from} и $+{to} окажутся соответственно строки Mon и Fri.

Более нетривиальное применение возможностей рекурсии в регулярных выражениях – обработка строк со вложенными скобками, число которых заранее неизвестно. Пример такого регулярного выражения показан в следующей программе.

```
use feature ":5.10";

my $re = qr
     /^
         (
             \(
```

```
                    (?:
                        [^()]
                        |
                        (?1)
                    )*
                \)
            )
        $/x;

my @tests = (
        "()",
        "(",
        "(1+2)",
        "1+2)",
        "(1-(2+3))",
        "(1-(2+3)",
        "(1+2+3*(4-5)+6/(2+3-(4*5*(6-7)))-8)",
);

for (@tests) {
        say /$re/ ? "ok" : "not ok", " $_";
}
```

Регулярное выражение $re требует, чтобы строка начиналась и заканчивалась круглыми скобками:

```
/^(\((?:[^()]|(?1))*\))$/
```

внутри которых либо не должно быть скобок:

```
/^(\((?:[^()]|(?1))*\))$/
```

либо должно содержаться то, что описывается выражением в первых сохраняющих скобках:

```
/^(\((?:[^()]|(?1))*\))$/
```

которые, в свою очередь, охватывают все регулярное выражение:

```
/^(\((?:[^()]|(?1))*\))$/
```

Таким образом, выражение рекурсивно содержит само себя и может совпадать с любым числом вложенных (и при этом парных) скобок.

Результат работы подтверждает, что невозможное в прежних версиях языка теперь возможно:

```
ok ()
not ok (
ok (1+2)
not ok 1+2)
ok (1-(2+3))
not ok (1-(2+3)
ok (1+2+3*(4-5)+6/(2+3-(4*5*(6-7)))-8)
```

## Другие изменения

В Perl 5.10 есть еще много менее значимых изменений. Подробности о нововведениях можно найти в документации, а именно в документе perl5100delta.pod [4]. Там же упомянуто и о несовместимостях с предыдущими версиями Perl (но это в основном касается нетривиальных случаев, использующих внутренности языка, поэтому большинство пользователей различий не заметят).

Вот краткий список некоторых новшеств, не описанных в статье:

- возможность «нанизывать» операторы проверки состояния файла, например: if -f -w $filename;
- функция readline() читает из *ARGV при вызове ее без аргументов;
- функцию readpipe() и операторы qx// и `` разрешено переопределять;
- появился новый тип блока специального кода UNITCHECK, который вызывается сразу после того, как скомпилирован соответствующий фрагмент кода;
- добавлена прагма mro (Method resolution order), изменяющая порядок просмотра дерева классов при множественном наследовании;
- в классе UNIVERSAL появился метод DOES(), который программист может переопределять, если ему не хватает проверки, выполняемой функцией isa() (например, когда классы не наследуются);
- расширен набор директив для форматированного вывода;
- функции pack() и unpack() распознают модификаторы > и <, указывающие порядок байт; unpack() будучи вызванной без аргументов, работает с переменной $_;
- инструкцией no N допустимо указать максимальную версию Perl, с которой разрешено выполнять программу, например no 5 в начале программы приведет к ошибке: «Perls since v5.0.0 too modern--this is v5.10.0»;
- функции chdir(), chmod() и chown() работатают не только с именами, но и с дескрипторами файлов (если позволяет операционная система); mkdir() берет аргумент по умолчанию $_;
- в регулярных выражениях появились экспериментальные управляющие конструкции (*THEN), (*PRUNE), (*MARK), (*SKIP), (*COMMIT), (*FAIL) и (*ACCEPT), которые, не поглощая символов, управляют ходом выполнения выражения, например позволяют поставить метку, откатить выполнение к следующей ветви, пропустить ветвление, либо вручную сообщить об ошибке [5].

## Несовместимости

Обновление языка не обошлось без исключения некоторых устаревших возможностей и исправления прежних ошибок, что может изменить работу старого кода или вообще не позволит откомпилировать старую программу. В числе устаревших конструкций, от которых решено отказаться:

- псевдохеши;
- переменные $* и $#;
- инструкция (?p{}) в регулярных выражениях;
- интерпретация массивов @- и @+ внутри регулярного выражения;
- компилятор perlcc и сопутствующие модули.

Тем, кто собирается переносить хитро написанные скрипты на Perl новой версии, необходимо ознакомиться с разделом Incompatible Changes документа perl5100delta.pod [4]. Однако большинство программ скорее всего будут работать без изменений.


1. http://search.cpan.org/~rgarcia/perl-5.10.0.
2. http://www.parrotcode.org/source.html.
3. http://talks.shitov.ru/ppt/moscow.pm/2/smart-matching.pdf.
4. http://search.cpan.org/~rgarcia/perl-5.10.0/pod/perl5100delta.pod.
5. http://www.regex-engineer.org/slides/perl510_regex.html.