Глава 5. Операторы

Учебник по Perl

Содержание [скрыть]

Perl является императивным языком программирования: его программа состоит из последовательности операторов, определяющих некоторые действия. *Оператор* – это завершенная инструкция интерпретатору на выполнение определенного действия. Все операторы языка Perl делятся на простые и составные. Простой оператор представляет собой выражение, возможно, снабженное модификатором. Составной оператор определяется в терминах блоков.

Каждый оператор Perl имеет возвращаемое значение. Для простого оператора – это значение вычисляемого в нем выражения, для составного оператора – значение последнего вычисленного в нем оператора.

Операторы обрабатываются в той последовательности, в которой они встречаются в программе. Однако среди множества допустимых операторов языка Perl, есть группа операторов, которая

изменяет последовательность выполнения операторов в программе.

Простые операторы

Простой оператор представляет собой любое выражение, завершенное точкой с запятой ";". Символ точка с запятой обязателен. Он может отсутствовать, только если простой оператор является последним оператором в блоке, специальной синтаксической конструкции, о которой мы поговорим чуть-чуть позже.

Основное назначение простого оператора – вычисление выражения с побочным эффектом, который связан с изменением значения некоторых переменных в выражении при его вычислении. Обычно он реализуется операциями увеличения/уменьшения на единицу (++, --):

```
$n++; # Переменная $n увеличивается на единицу.
–$n**2; # Переменная $n уменьшается на единицу.
```

Никакие другие операции Perl не вызывают побочных эффектов в выражении, если не считать операции присваивания (простое и составное), результатом вычисления которой является изменение значения левого операнда. Вызовы функций также могут приводить к побочным эффектам, но об этом подробнее мы расскажем в *главе 11.*

Простой оператор Perl может содержать выражение и без побочного эффекта. Все равно он будет рассматриваться интерпретатором как допустимый оператор, не выполняющий никакого действия, а только вычисляющий значение выражения. Однако если установлен флаг (-w) отображения интерпретатором предупреждающих сообщений, то будет получено сообщение о бесполезности использования соответствующей операции в void-контексте. Например, при выполнении оператора

```
($n*$m)**4 + 6;
```

будет отображено сообщение

```
Useless use of addition in void context at D:\PERL\EXAMPLE3.PL line 4.
```

Отметим, что в сообщении упоминается о последней выполненной операции в выражении.

> **Замечание**
>
> Подобное сообщение будет отображаться, даже если в сложном выражении присутствует операция, вызывающая побочный эффект. Сообщение не отображается только в случае выражения, составленного из одних операций уменьшения /увеличения, или выражения присваивания.

Читатель спросит, какой прок в операторе, не выполняющем никакого действия. Его можно использовать для задания возвращаемого пользовательской функцией значения. Забегая вперед, скажем, что если в функции явно не указано в операторе return о возвращаемое значение, то по

умолчанию Perl считает таковым значение последнего вычисленного оператора. Именно это обстоятельство и используется многими программистами для определения возвращаемого значения:

```
sub raySub {
какие-то операторы condition == true ? "Успех" : "Провал"; # Последний оператор * '
```

Последний оператор функции mysub вычисляет операцию выбора, в которой второй и третий операнды представлены просто строковыми литералами – выражениями без побочного эффекта. Результат вычисления одного из них и является возвращаемым значением функции.

Модификаторы простых операторов

Каждый простой оператор может быть снабжен *модификатором,* представляющим ключевое СЛОВО if, unless, while, until ИЛИ foreach, за которым следует выражение-условие. В самом операторе модификатор стоит непосредственно за выражением, составляющим простой оператор, перед завершающим символом точка с запятой. Каждый простой оператор может иметь только *один* модификатор. Семантически роль модификатора сводится к тому, что оператор вычисляется при выполнении условия, определяемого модификатором. Например, следующий оператор присваивания

```
$n = $l/$m if $т != 0;
```

с модификатором if будет выполнен при условии, что переменная $т не равна о. Общий синтаксис простого оператора с модификатором имеет следующий вид:

```
ВЫРАЖЕНИЕ ключ_слово_модификатора [(]ВЫРАЖЕНИЕ-УСЛОВИЕ [)];
```

Модификаторы *if* и *unless*

Модификаторы if и unless употребляются в прямом смысле их английского значения. Простой оператор с модификатором If выполняется, если ВЫРАЖЕНИЕ-УСЛОВИЕ истинно. Семантически простой оператор

```
ВЫРАЖЕНИЕ if ВЫРАЖЕНИЕ-УСЛОВИЕ;
```

эквивалентен следующему оператору условия:

```
if(ВЫРАЖЕНИЕ-УСЛОВИЕ) { ВЫРАЖЕНИЕ; }
```

> **Замечание**
>
> В этом разделе мы представляем эквивалентные простым операторам с модификаторами соответствующие составные операторы языка Perl. Их синтаксис и применение будут детально разобраны в следующих параграфах, но мы уверены, что читатель поймет их и без дополнительных объяснений. Во всяком случае всегда можно вернуться к данному разделу, прочитав разделы, посвященные составным операторам языка Perl.

Модификатор unless является прямой противоположностью модификатора if: простой оператор выполняется, если ВЫРАЖЕНИЕ-УСЛОВИЕ не истинно. Общий синтаксис простого оператора с модификатором unless имеет следующий вид:

```
ВЫРАЖЕНИЕ unless ВЫРАЖЕНИЕ-УСЛОВИЕ;
```

Это всего лишь удобная форма записи оператора условия

```
if( ! ВЫРАЖЕНИЕ-УСЛОВИЕ) { ВЫРАЖЕНИЕ; }
```

Замечание

ВЫРАЖЕНИЕ-УСЛОВИЕ вычисляется в булевом контексте: оно трактуется как Ложь, если равно о или пустой строке " ", и Истина – в любом другом случае.

Использование модификаторов if и unless показано в примере 5.1 – простой программе решения квадратного уравнения.

```
# perl -w
# Решение квадратного уравнения a*x**2+b*x+c=0
$a = <STDIN>;
$b = <STDIN>;
$c = <STDIN>;
$d = $b**2 - 4*$a*$c; # Вычисление дискриминанта уравнения
# Вычисление корней, если дискриминант положителен
( $xl =.(-$b+sqrt $d)/$a/2, $x2 = (-$b-sqrt $d)/$a/2 ) unless $d < 0;
# Печать результатов
print "Коэффициенты:\п a = $a b = $b c = $c"; print "\tPenieHHe:\n\t$xl\t$x2" if defined $xl;
print "^Решения нет!" unless defined $xl;
```

Наша программа решения квадратного уравнения, конечно, примитивна. Из всех возможных проверок в ней проверяется на положительность только дискриминант квадратного уравнения, хотя стоило бы проверить на нуль значение вводимого пользователем с клавиатуры старшего коэффициента уравнения, сохраняемого в переменной $a.

Модификатор unless используется в операторах вычисления корней и печати сообщения об отсутствии решения. Обратите внимание, что в операторе печати проверяется, определена ли переменная $xi, а будет она определена только в случае положительности дискриминанта $d. В модификаторе if оператора печати корней уравнения также проверяется, определена ли переменная $xl.

Модификаторы *while* и *until*

Эти два модификатора немного сложнее модификаторов if и unless. Они реализуют процесс циклического вычисления простого оператора. Их синтаксис таков:

```
ВЫРАЖЕНИЕ while ВЫРАЖЕНИЕ-УСЛОВИЕ; ВЫРАЖЕНИЕ until ВЫРАЖЕНИЕ-УСЛОВИЕ;
```

Модификатор while повторно вычисляет ВЫРАЖЕНИЕ, пока истинно ВЫРАЖЕНИЕ-УСЛОВИЕ. Модификатор until противоположен модификатору while: ВЫРАЖЕНИЕ повторно вычисляется до момента, когда ВЫРАЖЕНИЕ-УСЛОВИЕ станет истинным, иными словами оно вычисляется, пока ВЫРАЖЕНИЕ-УСЛОВИЕ ложно.

Семантически эти модификаторы простых операторов эквивалентны следующим составным операторам цикла:

```
while(ВЫРАЖЕНИЕ-УСЛОВИЕ) { ВЫРАЖЕНИЕ; } until(ВЫРАЖЕНИЕ-УСЛОВИЕ) { ВЫРАЖЕНИЕ; }
```

Пример 5.2 дает представление о том, как работают модификаторы повтора while И until.

```
# perl -w $first = 10;
• $second = 10; $first++ while $first < 15; # $first увеличивается, пока не станет
# равной 15
—$second until $second < 5; # $second уменьшается, пока не станет
# равной 4
print "\$first $first\n"; print "\$second $second\n";
```

Оператор увеличения на единицу переменной $first будет выполняться, пока выражение $ first < 15 остается истинным. Когда значение переменной $first станет равным 15, выражение модификатора while становится ложным, и оператор завершает работу. Аналогично работает и следующий оператор уменьшения переменной $second на единицу. Единственное отличие от предыдущего оператора заключается в том, что он выполняется, пока выражение $ second < 5 модификатора until остается ложным. Два оператора печати выведут на экран значения переменных $first и $second равными соответственно 15 и 4.

Замечание

Модификаторы повтора следует применять к простым операторам, вычисление которых приводит к изменению условий модификаторов. Если это не так, то простой оператор либо вообще не будет выполнен, либо будет выполняться бесконечно. Например, следующий оператор
print $var while $var < 15;
либо ни разу не напечатает значение переменной $var (если $var > =15), либо будет печатать бесконечно (если $var < 15). В последнем случае произойдет так называемое зацикливание и только нажатием комбинации клавиш <Ctrl>+<C> можно будет остановить выполнение этого оператора.

Модификаторы while и until сначала проверяют истинность или ложность своих выражений-условий, а потом, в зависимости от полученного результа, либо выполняют простой оператор, либо нет. Таким

образом, они реализуют *цикл с предусловием,* при котором оператор, для которого они являются модификаторами, может не выполниться ни одного раза.

Существует единственное исключение из этого правила, когда модификаторы повтора применяются к синтаксической конструкции do <}, которая не является оператором (хотя внешне и похожа), а относится к термам *(см. главу 4).* Поэтому, если ее завершить точкой с запятой, то такая конструкция будет являться простым оператором, к которому можно применять все возможные модификаторы Perl. Семантика этой конструкции заключается в том, что она вычисляет операторы, заданные в фигурных скобках {}, и возвращает значение последнего выполненного оператора. Так вот, если к простому оператору do {}; применить модификаторы повтора, то сначала выполнятся операторы конструкции do {}, а потом будет проверено условие модификатора. Это позволяет написать следующий простой оператор, который сначала осуществит ввод с клавиатуры, а потом проверит введенную информацию на совпадение с символом завершения ввода:

```
$string = ""; do{
$line = <STDIN>;
$string .= $line; I until $line eq ".\n";
```

Этот фрагмент кода будет накапливать вводимые пользователем строки в переменной $string до тех пор, пока не будет введена строка, состоящая из единственного символа точки, после чего оператор do{} until; завершит свою работу. Обратите внимание, что проверка в модификаторе until ведется на совпадение со строкой ". \n", в которой присутствует символ перехода на новую строку. Дело в том, что операция ввода <STDIN> передает этот символ в программу, так как пользователь именно этим символом завершает ввод строки (нажатие клавиши <Enter>).

Модификатор *foreach*

Модификатор foreach ВЫРАЖЕНИЕ относится к модификаторам цикла. Он повторно выполняет простой оператор, осуществляя итерации по списку значений, заданному в ВЫРАЖЕНИЕ. На каждой итерации выбранный элемент списка присваивается встроенной переменной $_, которую можно использовать в простом операторе для получения значения выбранного элемента списка. Например, следующий оператор распечатает все элементы массива @т:

```
print "$_ " foreach @т;
```

Общий синтаксис простого оператора с модификатором foreach следующий:

```
ВЫРАЖЕНИЕ foreach ВЫРАЖЕНИЕ-СПИСОК;
```

Простой оператор с модификатором foreach всего лишь удобная форма записи составного оператора jforeach:

```
foreach (ВЫРАЖЕНИЕ-СПИСОК) { ВЫРАЖЕНИЕ; }
```

Эта форма составного оператора foreach в качестве переменной цикла использует встроенную переменную $_ *(см. раздел 5.4.3).* Обратим внимание читателя на то, что ВЫРАЖЕНИЕ-список вычисляется в списковом контексте, поэтому все используемые в нем переменные ведут себя так, как они должны вести в списковом контексте. Например, хеш-массив представляет обычный список, составленный из последовательности его пар ключ/значение. Следующий фрагмент кода

```
%hash = ( one=>6, two=>8, three=>10 ); print "$_ " foreach %hash;
```

напечатает строку

```
three 10 two 8 one 6
```

Эта строка и есть тот простой список, который возвращает хеш в списковом контексте.

Относительно модификатора foreach (это же относится и к его эквивалентному оператору foreach) следует сказать одну важную вещь. Дело в том, что переменная $_ является не просто переменной, в которой хранится значение элемента списка текущей итерации, она является синонимом имени этого элемента. Это означает, что любое изменение переменной $_ в простом операторе приводит к изменению текущего элемента списка в цикле. Пример 5.3 демонстрирует, как просто можно умножить каждый элемент массива на некоторое число:

```
# perl -w
§array = (1, 2, 3);
$_ *= 2 foreach Sarray; # Умножение каждого элемента на 2.
print "@array"; # Напечатает строку: 246
```

Составные операторы

Составные операторы – это второй тип операторов языка Perl. С их помощью реализуют ветвления в программе и организуют циклические вычисления. Эти операторы, в отличие от аналогичных операторов других языков программирования, определяются в терминах блоков – специальном понятии языка Perl, задающим область видимости переменных. Именно с блоков мы и начнем изучение составных операторов.

Блоки

Блок – последовательность операторов, определяющая область видимости переменных. В программе блок обычно ограничен фигурными скобками *{...}.* Определяя синтаксис составных операторов, мы будем иметь в виду именно такой блок – последовательность операторов в фигурных скобках и обозначать его БЛОК. Интерпретатор рассматривает БЛОК как один оператор, вычисляемым значением которого является значение последнего выполненного оператора блока. Это означает, что там, где можно использовать один оператор, можно использовать и БЛОК. Такая ситуация встречается при использовании функции map(). Она выполняет определяемый ее первым параметром оператор для всех элементов списка, заданного вторым параметром. Значение каждого элемента списка при вычислениях

временно присваивается встроенной переменной $_. Возвращает эта функция список вычисленных значений оператора:

```
@rez = map $_ **= 2, @array; # Список квадратов элементов массива.
```

В качестве первого параметра этой функции можно использовать БЛОК. Следующий оператор также вычисляет список квадратов элементов массива @array, одновременно подсчитывая количество его элементов:

```
@rez = map { ++$kol; $_ **= 2} @array; # Список квадратов элементов
# массива и подсчет их количества.
```

Обратите внимание, что возвращаемым значением блока операторов в этом примере является значение последнего оператора блока, которое и попадает в возвращаемый функцией tap () список.

Замечание

Блоки в Perl не ограничиваются только последовательностью операторов в фигурных скобках. Иногда блок может быть ограничен содержащим его файлом. Например, файлом, содержащим используемый в программе модуль Perl, или файлом самой программы.

Как сказано в начале этого раздела, блок определяет *область видимости переменных.* Это означает, что в блоке можно создать переменные, обращаться к которым можно только из операторов, расположенных в этом блоке. Пока мы в блоке, мы можем присваивать им новые значения, использовать в вычислениях и т. п., но как только мы вышли из блока, мы теряем с ними "связь", они становятся "не видимыми". Такие переменные еще называют *локальными* переменными.

Локальные переменные создаются с помощью функции ту (). Ее параметром является список переменных, область видимости которых ограничена блоком, в котором вызывается эта функция. Если список переменных состоит из одной переменной, то скобки не обязательны. Созданные функцией my о переменные называются также *лексическими* переменными, так как область их действия ограничена фрагментом текста программы — блоком операторов.

В языке Perl можно создавать другой тип локальных переменных, область действия которых определяется динамически во время выполнения программы. Они создаются функцией local о и называются локальными *динамическими* переменных. Однако именно переменные, созданные функцией ту (), являются "истинными" локальными переменными: они создаются при входе в блок и уничтожаются при выходе из него (хотя существуют ситуации, когда Perl не уничтожает локальную лексическую переменную при выходе из ее области действия). Функция local о всего лишь временно сохраняет старое значение глобальной переменной при входе в блок и восстанавливает его при выходе из него.

(Более подробно лексические и динамические переменные рассматриваются в главе 11.)

Локальные переменные удобны для создания временных переменных, которые нигде больше не будут использоваться в программе, а только в одном определенном месте. Например, при отладке части кода часто приходится создавать временные переменные и выводить на печать их значения. Локальные переменные могут иметь такие же имена, как и глобально используемые переменные. Это не приводит к конфликту. После завершения операторов блока значение глобальной переменной имеет то же значение, которое она имела до начала выполнения операторов блока (пример 5.4).

```
# perl -w
$var = "outer"; # Глобальная переменная $var $glob = "glob"; # Глобальная переменная $glob my
$lex = "outer_l"; # Лексическая переменная $1ex {
my($var) = "inner"; # Внутренняя переменная $var my($varl) = "inner_l"; # Внутренняя
переменная $varl print "В блоке \$var = $var\n"; # Напечатает inner print "В блоке \$varl =
$varl\n"; # Напечатает inner_l print "В блоке \$lex = $lex\n"; # Напечатает outer_l print "В
блоке \$glob = $glob\n"; # Напечатает glob }
print "Вне блока \$var = $var\n"; # Напечатает outer print "Вне блока \$1ex = $lex\n"; #
Напечатает outer_l print "Вне блока \$varl = $varl\n"; # Напечатает пустую строку ""
```

Программа примера 5.4 демонстрирует области видимости лексических переменных. Внутри блока {...} "видны" переменные, созданные вне блока: и глобальные, и лексические ($giob, $iex), если только они не переопределены внутри блока ($var). При выходе из внутреннего блока восстанавливаются значения переменных, которые были переопределены внутри блока ($var). Доступ к локальным переменным блока извне невозможен ($vari).

Операторы ветвления

Операторы программы Perl выполняются последовательно в порядке их расположения в программе. Для реализации простых алгоритмов этого вполне достаточно. Однако большинство реальных алгоритмов не укладываются в такую линейную схему. Практически всегда при реализации любого алгоритма возникают ситуации, когда одну группу операторов надо выполнить только при выполнении определенных условиях, тогда как другую группу при этих же условиях вообще не следует выполнять. В языке Perl для организации подобного ветвления в программе предусмотрены операторы if, которые мы и будем называть *операторами ветвления.*

Эти операторы вычисляют выражение, называемое условием, и в зависимости от его истинности или ложности выполняют или не выполняют некоторый блок операторов. Это означает, что выражения условия во всех операторах ветвления вычисляются в булевом контексте.

Иногда приходится делать выбор на основе проверки нескольких различных условий. Для подобных цепочек ветвлений существует специальная форма оператора if, реализующая множественные проверки.

В языке существует три формы оператора ветвления if:

```
if (ВЫРАЖЕНИЕ) БЛОК
if (ВЫРАЖЕНИЕ) БЛОК1 else БЛОК2
if (ВЫРАЖЕНИЕ!) БЛОК1 elsif (ВЫРАЖЕНИЕ2) БЛОК2 ... else БЛОКп
```

Обратим внимание читателя еще раз на тот факт, что все они определяются в терминах блоков операторов, заключенных в фигурные скобки, поэтому даже если в блоке содержится один оператор, он должен быть заключен в фигурные скобки. Такой синтаксис составных операторов Perl может оказаться не совсем привычным для программистов на языке C, в котором фигурные скобки в случае одного оператора в блоке не обязательны.

Первый оператор if реализует простейшее ветвление. Если ВЫРАЖЕНИЕ истинно, то выполняются операторы из БЛОК, в противном случае БЛОК просто пропускается:

```
$var = 10;
if ( $var -= 5 ) {
print "Переменная \$var - $var"; }
```

Обратите внимание, что в этом примере ВЫРАЖЕНИЕ представляет операцию составного присваивания. Это может показаться необычным, так как в большинстве языков программирования здесь требуется выражение, возвращающее булево значение. В Perl можно использовать любое выражение, в том числе и присваивание. Результат его вычисления интерпретируется в булевом контексте: если вычисленное значение равно о или пустой строке "", то оно трактуется как Ложь, иначе – Истина. Возвращаемым значением операции присваивания является значение, присвоенное переменной левого операнда. В нашем примере это число 5, следовательно в булевом контексте оно трактуется как Истина, а поэтому оператор печати print будет выполнен. Если перед выполнением оператора if переменная $var будет равняться 5, то выражение условия будет вычислено равным Ложь и все операторы блока будут просто пропущены.

Обычно выражение условия представляет собой сложное выражение, составленное из операций отношения, связанных логическими операциями. Использование операции присваивания в выражении условия оператора if не совсем типично. Здесь мы его использовали, чтобы подчеркнуть то обстоятельство, что в Perl любое правильное выражение может быть использовано в качестве выражения условия, которое вычисляется в булевом контексте. Вторая форма оператора if используется, когда необходимо выполнить одну группу операторов (БЛОК!) в случае истинности некоторого выражения (ВЫРАЖЕНИЕ), а в случае его ложности – другую группу операторов (влок2):

```
if ($var >= 0} # ВЫРАЖЕНИЕ {
print "Переменная неотрицательна."; # БЛОК1, если ВЫРАЖЕНИЕ истинно } else {
print "Переменная отрицательна."; # БЛОК2, если ВЫРАЖЕНИЕ ложно }
```

По существу, первая форма оператора if эквивалентна второй форме, если БЛОК2 не содержит ни одного оператора.

Последняя, третья форма оператора if реализует цепочку ветвлений. Семантика этого оператора такова. Выполняются операторы из БЛОК!, если **ИСТИННО** ВЫРАЖЕНИЕ!. **ЕСЛИ ОНО ЛОЖНО, ТО ВЫПОЛНЯЮТСЯ Операторы** ИЗ БЛОК2

в случае истинности вырлжЕНИЕ2. Если и оно ложно, то проверяется ВЫРАЖЕ-НИЕ3 и т. д. Если ни одно из выражений условия оператора if не истинно, то выполняются операторы блока, определяемого после ключевого слова else в случае его наличия. В противном случае выполняется следующий после оператора if оператор программы. При выполнении следующего оператора ветвления if

```
if( $var < 0) { # ВЫРАЖЕНИЕ!
print "Переменная отрицательна"; i БЛОК! } elsif ( $var == 0) { # ВЫРАЖЕНИЕ2
print "Переменная равна нулю"; # БЛОК2 } else {
print "Переменная положительна"; # БЛОК3 }
```

сначала проверяется условие отрицательности переменной $var. Если значение переменной строго меньше нуля (ВЫРАЖЕНИЕ1), то печатается сообщение из БЛОК! и оператор завершает свою работу. Если значение переменной не меньше нуля, то оно проверяется на равенство (ВЫРАЖЕНИЕ2) и в случае истинности выполняется оператор печати из блока операторов elsif (влок2). Если проверка на равенство нулю дала ложный результат, то выполняется оператор печати из блока операторов else (БЛОК3).

> **Замечание**
>
> Ключевое слово else вместе со своим блоком операторов может быть опущено.

В операторе if со множественными проверками может быть сколько угодно **блоков** elsif, **НО ТОЛЬКО ОДИН БЛОК** else.

Так как все операторы ветвления определяются в терминах блоков операторов, то не возникает двусмысленности при определении, какие операторы в какой части выполняются.

При работе с операторами ветвления важно помнить, что только один блок операторов будет выполнен – тот, для которого истинно соответствующее выражение условия.

Во всех операторах ветвления ключевое слово if может быть заменено на unless. В этом случае проверка выражения условия осуществляется на его ложность. Последний оператор if можно записать и так:

```
unless( $var >= 0} { # ВЫРАЖЕНИЕ!
print "Переменная отрицательна"; # БЛОК! } elsif ( $var == 0) { # ВЫРАЖЕНИЕ2
print "Переменная равна нулю"; # БЛОК2 } else {
print "Переменная положительна"; # БЛОК3 }
```

При этом нам пришлось заменить ВЫРАЖЕНИЕ! на противоположное по смыслу.

> **Замечание**
>
> Все операторы if (unless) могут быть вложенными, т. е. в любом их блоке можно свободно использовать другие операторы ветвления.

Операторы цикла

Известно, что для реализации любого алгоритма достаточно трех структур управления: последовательного выполнения, ветвления по условию и цикла с предусловием. Любой язык программирования предоставляет в распоряжение программиста набор всех трех управляющих конструкций, дополняя их для удобства программирования другими конструкциями: цепочки ветвления и разнообразные формы цикла с предусловием, а также циклы с постусловием.

Мы уже познакомились с операторами ветвления Perl, а теперь пришло время узнать, какие конструкции цикла можно применять в Perl. Их всего три: while, for и foreach. Все они относятся к классу *составных* операторов и, естественно, определяются в терминах блоков БЛОК.

Циклы *while* и *until*

Цикл while предназначен для повторного вычисления блока операторов, пока остается истинным задаваемое в нем выражение-условие. Его общий синтаксис имеет две формы:

```
МЕТКА while (ВЫРАЖЕНИЕ) БЛОК
МЕТКА while (ВЫРАЖЕНИЕ) БЛОК continue БЛОК1
```

Все операторы цикла могут быть снабжены не обязательными метками. В Perl метка представляет правильный идентификатор, завершающийся двоеточием ":". Она важна для команды перехода next, о которой мы поговорим в следующем разделе.

Оператор while выполняется по следующей схеме. Вычисляется выражения-условия ВЫРАЖЕНИЕ. Если оно истинно, то выполняются операторы БЛОК. В противном случае оператор цикла завершает свою работу и передает управление следующему после него оператору программы (цикл 1 примера 5.5). Таким образом, оператор цикла while является управляющей конструкцией цикла с предусловием: сначала проверяется условие завершения цикла, а потом только тело цикла, определяемое операторами БЛОК. Поэтому может оказаться, что тело цикла не будет выполнено ни одного раза, если при первом вхождении в цикл условие окажется ложным (цикл 3 примера 5.5).

Вместо ключевого слова while можно использовать ключевое слово until. В этом случае управляющая конструкция называется циклом until, который отличается от разобранного цикла while тем, что его тело выполняется, только если выражение условия *ложно* (цикл 2 примера 5.5).

```
# perl -w
# цикл 1
```

```
$i = 1;
while <$i•<= 3) {
$a[$i] = l/$i; # Присвоить значение элементу массива
++$i; >
print "Переменная цикла \$i = $i\n"; # $i = 4 print "Массив \@a: @a\n"; # @a = (I, 0.5.
0.333333333333333)
# цикл 2, эквивалентный предыдущему
$i = 1;
until ($i > 3) {
$a[$i] = l/$i; # Присвоить значение элементу массива
++$i; }
print "Переменная цикла \$i = $i\n"; # $i = 4 print "Массив \@a: @a\n"; # @a = (1, 0.5.
0.333333333333333)
# цикл 3, тело цикла не выполняется ни одного раза
$i = 5;
while ($i-<= 3) {
$a[$i] = l/$i;
++$i; } print "Переменная цикла \$i = $i\n"; # $i = 5
# цикл 4, бесконечный цикл (не изменяется выражение условия)
$i = 1;
while ($i <= 3) {
$a[$i] = l/$i; } .
```

Обратим внимание на то, что в теле цикла должны присутствовать операторы, вычисление которых приводит к изменению выражения условия. Обычно это операторы, изменяющие значения переменных, используемых в выражении условия. Если этого не происходит, то цикл while или until будет выполняться бесконечно (цикл 4 примера 5.5).

Замечание

Цикл с постусловием реализуется применением модификатора while к конструкции do{), и рассматривался нами в разделе 5.2.2 " Модификаторы while и until".

Блок операторов БЛОК!, задаваемый после ключевого слова continue, выполняется всякий раз, когда осуществляется переход на выполнение новой итерации. Это происходит после выполнения последнего оператора тела цикла или при явном переходе на следующую итерацию цикла командой next. Блок continue на практике используется редко, но с его помощью можно строго определить цикл for через оператор цикла while.

Пример 5.6 демонстрирует использование цикла while для вычисления степеней двойки не выше шестнадцатой. В этом примере оператор цикла while функционально эквивалентен циклу for. Блок

continue выполняется всякий раз по завершении очередной итерации цикла, увеличивая переменную $i на единицу. Он эквивалентен выражению увеличения/уменьшения оператора for *(см. следующий раздел).*

```
# perl -w
# Вычисление степеней числа 2 $1 = I;
while ($i <= 16) {
print "2 в степени $i: ", 2**$i, "\n"; } continue {
++$i; f Увеличение переменной $i перед выполнением следующей итерации }
```

Цикл *for*

При выполнении циклов while и until заранее не известно, сколько итераций необходимо выполнить. Их количество зависит от многих факторов: значений переменных в выражении условия до начала выполнения цикла, их изменении в теле цикла, виде самого выражения условия и т. п. Но иногда в программе необходимо выполнить заранее известное количество повторений определенной группы операторов. Например, прочитать из файла 5 строк и видоизменить их по определенным правилам. Конечно, можно такую задачу запрограммировать операторами цикла while и until, но это может выглядеть не совсем выразительно. В том смысле, что при прочтении программы придется немного "пошевелить" мозгами, прежде чем понять смысл оператора цикла. Для решения подобных задач с заранее известным числом повторений язык Perl предлагает специальную конструкцию цикла – цикл for:

```
МЕТКА for (ВЫРАЖЕНИЕ!; ВЫРАЖЕНИЕ2; ВЫРАЖЕНИЕ3) БЛОК
```

ВЫРАЖЕНИЕ! используется для установки начальных значений переменных, управляющих циклом, поэтому его называют *инициализирующим* выражением. Обычно это одна или несколько операций присваивания, разделенных запятыми.

ВЫРАЖЕНИЕ2 определяет условие, при котором будут повторяться итерации цикла. Оно, как и выражение-условие цикла while, должно быть истинным, чтобы началась следующая итерация. Как только это выражение становится ложным, цикл for прекращает выполняться и передает управление следующему за ним в программе оператору.

ВЫРАЖЕНИЕ3 отвечает за увеличение/уменьшение значений переменных цикла после завершения очередной итерации. Обычно оно представляет собой список выражений с побочным эффектом или список операций присваивания переменным цикла новых значений. Его иногда называют *изменяющим* выражением.

Алгоритм выполнения цикла for следующий:

1. Вычисляется инициализирующее выражение (ВЫРАЖЕНИЕ!).

2. Вычисляется выражение условия (выражение2). Если оно истинно, то выполняются операторы блока БЛОК, иначе цикл завершает свое выполнение.

3. После выполнения очередной итерации вычисляется выражение увеличения/уменьшения (ВЫРАЖЕНИЕ3) и повторяется пункт 2.

Как отмечалось в предыдущем разделе, цикл for эквивалентен циклу while с блоком continue. Например, следующий цикл

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

эквивалентен циклу while

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

Существует единственное отличие между этими двумя циклами. Цикл for определяет лексическую область видимости для переменной цикла. Это позволяет использовать в качестве переменных цикла локальные переменные, объявленные с помощью функции ту:

```
$i = "global";
for (my $i = 1; $i <= 3; $i++) {
print "Внутри цикла \$i: $i\n"; } print "Вне цикла \$i: $i\n ";
```

При выполнении этого фрагмента программы оператор печати будет последовательно отображать значения 1, 2 и з локальной переменной цикла $1. При выходе из цикла локальная переменная $i будет уничтожена и оператор печати вне цикла напечатает строку global — значение глобальной переменной $1, определенной вне цикла for.

Все три выражения цикла for являются необязательными и могут быть опущены, но соответствующие разделители ",-" должны быть оставлены. Если опущено выражение условия, то по умолчанию оно принимается равным Истина. Это позволяет организовать бесконечный цикл:

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

Выход из такого цикла осуществляется командами управления, о которых речь пойдет в следующем параграфе.

Инициализировать переменную цикла можно и вне цикла, а изменять значение переменной цикла можно и внутри тела цикла. В этом случае инициализирующее и изменяющее выражения не обязательны:

```
$i = 1;
for (•; $i <= 3;) {
$i++; }
```

Совет

Хотя существует возможность изменения переменной цикла в теле цикла, не рекомендуется ею пользоваться. Цикл for был введен в язык именно для того, чтобы собрать в одном месте все операторы, управляющие работой цикла, что позволяет достаточно быстро изменить его поведение.

Цикл for позволяет использовать несколько переменных для управления работой цикла. В этом случае в инициализирующем и изменяющем выражениях используется операция запятая. Например, если мы хотим создать хеш, в котором ключам, представляющим цифры от 1 до 9, соответствуют значения этих же цифр в обратном порядке от 9 до 1, то эту задачу можно решить с помощью цикла for с двумя переменными цикла:

```
for ($j = 1, $k = 9; $k >0; $j++, $k—) {
$hash{$j} = $k; }
```

Этот же пример показывает, что в цикле for переменная цикла может как увеличиваться, так и уменьшаться. Главное, чтобы выражение условия правильно отслеживало условия продолжения итераций цикла.

Цикл for достаточно гибкая конструкция, которую можно использовать не только для реализации цикла с заранее заданным числом итераций. Он позволяет в инициализирующем и изменяющем выражениях использовать вызовы встроенных и пользовательских функций, а не только определять и изменять переменные цикла. Основное — чтобы изменялось выражение условия завершения цикла. Пример 5.7 демонстрирует именно такое использование цикла for.

```
# perl -w
for (print "Введите данные, для завершения ввода нажмите <Enter>\n"; <STDIN>;
print "Введите данные, для завершения ввода нажмите <Enter>\n") {
last if $_ eq "\n"; print "Ввели строку: $_"; }
```

В этом примере пользователь вводит в цикле строки данных. Перед вводом новой строки отображается подсказка с помощью функции print (), которая определена в изменяющем выражении цикла. Выражение условия представляет операцию ввода из файла стандартного ввода <STDIN>. Так как это выражение вычисляется всякий раз, когда цикл переходит на очередную итерацию, то на каждом шаге цикла программа будет ожидать ввода с клавиатуры. Выход из цикла осуществляется командой last, вызываемой в случае ввода пользователем пустой строки. Введенные данные сохраняются во встроенной переменной $_, причем в ней сохраняется и символ перехода на новую строку, являющийся признаком завершения операции ввода данных. Поэтому при вводе пустой строки

на самом деле в переменной $_ хранится управляющая последовательность "\n", с которой и осуществляется сравнение для реализации выхода из цикла.

Пример 5.8 демонстрирует программу, читающую 3 строки файла err.err. Операция чтения из файла задается в инициализирующем и изменяющем выражении вместе с определением и изменением переменной цикла $1.

```
# perl -w
open (FF, "err.err") or die "Ошибка открытия файла";
for ($line=<FF>, $count = 1; $count <=3; $line=<FF>, $count++)
{
print "Строка $count:\n $line\n";
t ' . }
close(FILE);
```

Цикл *foreach*

Одно из наиболее частых применений циклов в языках программирования – организация вычислений с элементами массивов: найти максимальный элемент, распечатать элементы массива, выяснить, существует ли элемент массива, равный заданному значению, и т. п. Подобные задачи легко решаются с помощью циклов while и for. В примере 5.9 определяется максимальный элемент массива (в предположении, что он содержит числовые данные).

```
#! perl -w
@array = (1,-6,9,18,0,-10);
$max = $array[0]; ,
for ($i = 1; $i <= $farray; $i++) {
$max = $array[$i] if $array[$i] > $max; }
```

После выполнения программы примера 5.9 переменная $max будет иметь значение 18, равное максимальному элементу массива $array. Обратим внимание читателя на то, что в цикле for (как и в цикле while) доступ к элементам массива организуется с помощью индекса.

В Perl списки и массивы, являющиеся, по существу, также списками, являются столь полезными и часто используемыми конструкциями, что для организации цикла по их элементам в языке предусмотрен специальный оператор foreach, имеющий следующий синтаксис:

```
МЕТКА foreach ПЕРЕМЕННАЯ (СПИСОК) БЛОК
МЕТКА foreach ПЕРЕМЕННАЯ (СПИСОК) БЛОК continue БЛОК
```

Он реализует цикл по элементам списка список, присваивая на каждом шаге цикла переменной ПЕРЕМЕННАЯ значение выбранного элемента списка. Блок операторов continue выполняется всякий раз, как начинается очередная итерация, за исключением первой итерации, когда переменная $temp

равна первому элементу списка. Список можно задавать или последовательностью значений, разделенных запятыми, или массивом скаляров, или функцией, возвращаемым значением которой является список. Определение максимального элемента массива можно переписать с циклом foreach (пример 5.10).

```
#! perl, -w
Sarray = (1,-6,9,18,0,-10) ; $max = $array[0]; foreach $temp (Sarray) (
$max = $temp if $temp > $max; } print "$max";
```

На каждом шаге цикла переменная $temp последовательно принимает значения элементов массива $array. Обратите внимание на внешний вид программы – в отсутствии индексов массива она стала лучше читаемой.

Отметим несколько особенностей цикла foreach. Прежде всего следует сказать, что ключевое слово foreach является синонимом ключевого слова for. Цикл из примера 5.10 можно было бы записать и так:

```
for $temp (@array) { # Ключевое слово foreach синоним for.
$max = $temp if $temp > $max; }
```

Однако, как нам кажется, использование foreach лучше отражает семантику этого оператора цикла, так как в самом ключевом слове уже отражена его сущность (for each – для каждого).

Следующая особенность оператора foreach связана с переменной цикла. По умолчанию эта переменная является локальной, область видимости которой ограничена телом цикла. Она создается только на время выполнения оператора foreach, доступна внутри тела цикла и уничтожается при выходе из цикла.

Обычно программисты, работающие на языке Perl, вообще не применяют в циклах foreach переменную цикла. Это связано с тем обстоятельством, что в отсутствии явно заданной переменной цикла Perl по умолчанию использует специальную переменную $_. На каждом шаге цикла именно она будет содержать значение элемента списка или массива. С учетом этого факта цикл foreach примера 5.10 можно переписать так:

```
foreach (@array) { # В качестве переменной цикла используется $_.
$roax = $_ if $_ > $max;
} . l
```

Последняя особенность оператора foreach, которая также связана с переменной цикла, заключается в том, что фактически на каждом шаге выполнения цикла эта переменная является синонимом того элемента списка, значение которого она содержит. Это означает, что ее изменение в цикле приводит к изменению значения соответствующего элемента списка. Это свойство цикла foreach удобно для изменения значений элементов списка. Отметим, что его можно применять к спискам,

хранящимся в массивах. Например, возвести в квадрат каждый элемент списка можно следующим оператором foreach:

```
foreach $temp (@array) {
$temp **= 2; }
```

Список, по элементам которого организуется цикл, может быть задан не только явно конструктором или переменной массива, но и функцией, возвращаемым значением которой является список. Канонический способ печати хеш-массива в упорядоченном порядке представлен в примере 5.11.

```
# perl -w %array = {
blue => 1,
red => 2,
green => 3,
yellow => 3 ); foreach (sort keys %array) {
print "$_\t => $array{$_}\n"; } '
```

Эта программа напечатает пары ключ/значение хеш-массива %array в соответствии с алфавитным порядком его ключей:

```
blue => 1
green => 3
red => 2
yellow => 3
```

Замечание

Цикл foreach выполняется быстрее аналогичного цикла for, так как не требует дополнительных затрат на вычисление индекса элемента списка.

Команды управления циклом

Каждый цикл в программе завершается при достижении некоторого условия, определяемого самим оператором. В циклах while и for это связано с ложностью выражения-условия, а в цикле foreach с окончанием перебора всех элементов списка. Иногда возникает необходимость при возникновении некоторых условий завершить выполнение всего цикла, либо прервать выполнение операторов цикла и перейти на очередную итерации. Для подобных целей в языке Perl предусмотрены три команды last, next и redo, которые и называют *командами управления циклом.*

Синтаксис этих команд прост – ключевое слово, за которым может следовать необязательный идентификатор метки:

```
last ИДЕНТИФИКАТОР_МЕТКИ; next ИДЕНТИФИКАТОР_МЕТКИ; redo ИДЕНТИФИКАТОР_МЕТКИ;
```

Семантика этих команд также проста. Они изменяют порядок выполнения циклов, принятый по умолчанию в языке, и передают управление в определенное место программы, завершая выполнение цикла (last), переходя на следующую итерацию цикла (next) или повторяя выполнение операторов тела цикла при тех же значениях переменных цикла (redo). Место перехода задается *меткой.* Помните синтаксис операторов цикла? Каждый из них может быть помечен. Именно идентификаторы меток операторов цикла и используются в командах управления для указания места передачи управления.

Метка в программе Perl задается идентификатором, за которым следует двоеточие. В командах управления циклом используется именно *идентификатор метки,* а не *метка.*

Несколько слов о терминологии. Читатель, наверное, обратил внимание, что мы не называем команды управления циклом операторами. И это справедливо. Они не являются операторами, хотя могут использоваться как операторы. Их следует считать *унарными операциями,* результатом вычисления которых является изменение последовательности выполнения операторов. Поэтому команды управления циклом можно использовать в любом выражении Perl. Заметим, что их следует использовать в таких выражениях, где имеет смысл их использовать, например в выражениях с операцией "запятая":

```
open (INPUT_FILE, $file)
or warn ("Невозможно открыть $file: $!\n"), next FILE;
```

Приведенный оператор может являться частью программы, которая в цикле последовательно открывает и обрабатывает файлы. Команда next инициирует очередную итерацию цикла с меткой FILE, если не удалось открыть файл в текущей итерации. Обратите внимание, что она используется в качестве операнда операции "запятая". В таком контексте эта команда имеет смысл. Следующий оператор является синтаксически правильным, но использование в нем команды redo не имеет никакого смысла:

```
print "qu-qu", 5 * redo OUT, "hi-hi\n";
```

Результатом выполнения этого оператора будет повторение вычислений операторов цикла с меткой ODT, т. е. простое выполнение команды redo OUT.

Относительно команд управления циклом следует сказать, что к ним можно применять модификаторы, так как употребленные самостоятельно с завершающей точкой с запятой они рассматриваются как *простые* операторы: next if $a – 2;

Переход на следующую итерацию цикла осуществится только, если переменная $a равна 2.

Команда *last*

Команда last немедленно прекращает выполнение цикла, в котором она задана, и передает управление на оператор, непосредственно следующий за оператором цикла. Ее целесообразно использовать для нахождения одного определенного значения в массиве (пример 5.12).

```
#! perl -w
@letters = ("A".."Z");
for ($index=0; $index<01etters; $index++) {
last if $letters[$index] eq "M"; } print $index;
```

Цикл в программе примера 5.12 будет выполняться, пока перебор элементов массива $ letters не достигнет элемента, содержащего символ "м". После чего будет выполнен первый после оператора for оператор программы. В результате будет напечатано число 12 – индекс элемента, содержащего символ "м".

Метка используется для конкретизации передачи управления в случае вложенных циклов: управление передается непосредственно на оператор, следующий за оператором цикла с указанной меткой (пример 5.13).

```
CYCLE_1: while (...){ , CYCLE_2: for (...) {
CYCLE_3: foreach (...) { last CYCLE_2; }
```

Операторы цикла CYCLE_2 } Операторы цикла CYCLE_1 # Сюда передает управление

```
t оператор last CYCLE_2; }
```

Если в команде last указать метку CYCLE_I, то управление будет передано на первый после самого внешнего цикла оператор программы. Если в команде last задать метку CYCLE_S (или задать ее вообще без метки), то управление будет передано на первый оператор группы операторы цикла CYCLE_2.

Передача управления командой last осуществляется не *на* оператор цикла с соответствующей меткой, а на оператор, непосредственно следующий *за* ним.

Команда last осуществляет выход из цикла, не выполняя никаких блоков **операторов** continue.

Команда *next*

Команда next позволяет пропустить расположенные после нее в теле цикла операторы и перейти на следующую итерацию цикла. Если оператор цикла содержит блок continue, то его операторы выполняются до проверки условия окончания цикла, с которой начинается следующая итерация. Одно из применений этой команды – обработать определенные элементы массива, ничего не делая с другими. Программа примера 5.14 присваивает всем элементам массива, содержащим четные числа, символ звездочка "*".

```
#! perl -w
@array = (2, 5, 8, 4, 7,, 9); print "До: @array\n"; foreach (Sarray) { next if $_ % 2;
$_ = »*»;
```

```
}
print "После: @array\n";
```

Результат выполнения программы примера 5.14 показан ниже:

```
До: 2 5 8 4 7 9 После: * 5 * * 7 9
```

Если элемент массива нечетное число, то результат операции $_ % 2 равен *i* (Истина) и команда next инициирует следующую итерацию цикла foreach, не изменяя значение текущего элемента массива. Если значением элемента массива является четное число, то команда next не выполняется и значение элемента меняется на символ "*".

Команда next, употребленная совместно с идентификатором метки прерывает выполнение цикла, в теле которого она находится, и начинает новую итерацию цикла с указанной меткой, выполнив предварительно его блок continue, если таковой имеется (пример 5.15).

```
#! perl -w
$out =- 0;
OUT: while ($out < 2) {
print "Начало внешнего цикла\n";
for($in=0; $in<=2; $in++) {
print "\$out: $out\t\$in: $in\n"; next OUT if $in =1; }
print "\$out: $out\n"; # Никогда не выполняется! } continue {
print "Блок continue внешнего цикла\n"; $out++; }
```

Вывод этой программы будет следующим:

```
Начало внешнего цикла
$out: 0 $in: O
$out: 0 $in: 1
Блок continue внешнего цикла
Начало внешнего цикла
$out: I $in: O
$out: I $in: 1
Блок continue внешнего цикла
```

Обратите внимание, что количество итераций внутреннего цикла for равно двум, так как на второй его итерации выполняется команда next OUT, прекращающая его выполнение и инициализирующая выполнение очередной итерации внешнего цикла OUT. Оператор печати этого цикла пропускается, выполняется блок операторов continue, проверяется условие и если оно истинно, то тело цикла выполняется. Таким образом, оператор печати внешнего цикла OUT не выполняется ни одного раза, что подтверждается приведенным выводом из программы примера 5.15.

Команда *redo*

Команда redo повторно выполняет операторы тела цикла, не инициализируя следующую итерацию. Это означает, что ни выражение изменения цикла for, ни операторы блока continue, если он присутствует, ни выражение условия не вычисляются. Операторы тела цикла, расположенные за оператором redo, пропускаются и снова начинается выполнение тела цикла со значениями переменных, которые они имели перед выполнением этой передачи управления. Программа примера 5.16 демонстрирует использование команды redo.

```
I! perl -w $notempty·= 0;
$total = 0;
for (;;) { tt Бесконечный цикл
$line=<STDIN>; # Ввод строки
chop($line);
last if $line eq "END"; # Выход из цикла
++$total;
redo if $line eq ""; # Возврат на чтение строки
++$notempty; } print "Всего прочитано строк: $total\nM3 них не пустых: $notempty\n";
```

Эта программа в бесконечном цикле ожидает ввода пользователем на клавиатуре строки данных и в переменной $ total подсчитывает количество введенных строк. В переменной $notempty вычисляется количество введенных не пустых строк. Если введена пустая строка, то команда redo начинает повторное выполнение операторов тела цикла, не увеличивая на единицу переменную $notempty. Для завершения бесконечного цикла следует ввести строку END. В этом случае выполняется команда last.

Функция chop используется для удаления из введенной пользователем строки символа перехода на новую строку "\n", поэтому в программе она сравнивается со строками без завершающего символа перехода на новую строку (сравни с примером 5.7).

Если команда redo используется с идентификатором метки, то ее действие аналогично действию команды next с той лишь разницей, что она просто передает управление на первый оператор тела цикла с указанной меткой, не инициируя следующей итерации и не вычисляя операторов блока continue. В качестве иллюстрации такого использования команды redo перепишем программу примера 5.16 следующим образом:

```
#! perl -w $notempty = 0; $total = 0; OUT: while (1) {
print "Введи строки\n"; # Сюда передает управление команда redo OUT; for {;;) {
$line=<STDIN>;
chop($line);
last OUT if-$line eq "END"; I Выход из всех циклов
++$total;
```

```
redo OUT if $line eq "";
++$notempty; } } print "Всего прочитано строк: $tptal\nИЗ них не пустых: $notempty\n";
```

В примере 5.17 мы ввели внешний бесконечный цикл OUT и изменили команды redo и last, добавив к ним метку на внешний цикл. Теперь в случае, если пользователь вводит пустую строку, команда redo OUT передает управление на первый оператор внешнего цикла, и программа печатает приглашение ввести строки.

Именованные блоки

В Perl блок операторов, заключенный в фигурные скобки, семантически эквивалентен циклу, выполняющемуся один раз. В связи с этим обстоятельством можно использовать команду last для выхода из него, а команду redo для повторного вычисления операторов блока. Команда next также осуществляет выход из блока, но отличается от команды last тем, что вычисляются операторы блока continue, который может задаваться для блока операторов в фигурных скобках:

```
BLOCK1: {
$i = 1;
last BLOCK1; } continue {
++$i; }
print "Переменная \$i после BLOCK1: $i\n"; BLOCK2: {
$i = 1;
next BLOCK2; } continue {
++$i; } print "Переменная \$i после BLOCK2: $i\n";
```

Первый оператор print этого фрагмента кода напечатает значение переменной $i равным тогда как второй оператор print напечатает 2, так как при *г,* выходе из блока BLOCK2 будет выполнен оператор увеличения на единицу переменной $i из блока continue.

Замечание

Если в простом блоке операторов задан блок continue, то при нормальном завершении простого блока (без использования команд управления циклом) блок операторов continue также будет выполнен.

Блоки могут иметь метки, и в этом случае их называют Подобные конструкции используются для реализации переключателей — конструкций, которые не определены в синтаксисе языка Perl. Существует множество способов создания переключателей средствами языка Perl. Один из них представлен в примере 5.18. *именованными блоками.*

```
#! perl -w $var = 3; SWITCH: {
$casel = 1, last SWITCH if $var == 1;
$case2 = 1, last SWITCH if $var == 2;
```

```
$case3 = 1, last SWITCH if $var = 3;
$nothing = 1; }
```

После выполнения именованного блока операторов SWITCH переменная $casei будет равна 1, если $var равна i, $case2 будет равна 2, если $var равна 2 и, наконец, $case3 будет равна з, если $var равна з. В случае, если переменная $var не равна ни одному из перечисленных значений, то переменная $nothing будет равна i. Конечно, это простейший переключатель, разработанный всего лишь для демонстрации возможности быстрого создания переключателя в Perl. Для выполнения группы операторов в переключателе можно использовать не модификатор if, а оператор выбора if.

Блоки могут вложенными друг в друга. Именованные блоки и команды управления циклом, используемые для выхода из внутренних блоков, позволяют создавать достаточно прозрачные конструкции, реализующие сложные алгоритмы. Например, можно организовать бесконечный цикл без использования какого-либо оператора цикла:

```
$notempty = 0; $total = 0; INPUT: {
$line=<STDIN>; chop($line);
last INPUT if $line eq "END"; # Выход из бесконечного цикла ++$total;
redo INPUT if $line eq ""; ++$notempty; redo INPUT; }
```

Узнаете программу примера 5.16? Действительно, это реализация без оператора цикла программы ввода строк и подсчета общего числа введенных, а также непустых строк. Единственное, что нам пришлось добавить — еще одну команду redo в конце блока операторов.

Оператор безусловного перехода

Оператор безусловного перехода goto, возможно, самый спорный оператор. Много копий было поломано в дебатах о его целесообразности и полезности. Однако практически в любом языке программирования можно обнаружить оператор безусловного перехода. Не является исключением и язык Perl. В нем есть три формы этого оператора:

```
goto МЕТКА; goto ВЫРАЖЕНИЕ; goto &ПОДПРОГРАММА;
```

Первая форма goto МЕТКА передает управление на оператор с меткой МЕТКА, который может быть расположен в любом месте программы, за исключением конструкций, требующих определенных инициирующих действий перед их выполнением. К ним относятся цикл foreach и определение подпрограммы sub.

Замечание

Компилятор Perl не генерирует никаких ошибок, если в операторе goto задана не существующая метка, или он передает управление в конструкцию foreach или sub. Все ошибки, связанные с этим оператором, возникают во время выполнения программы.

Во второй форме оператора безусловного перехода goto ВЫРАЖЕНИЕ возвращаемым значением выражения должен быть метка, на которую и будет передано управление в программе. Эта форма оператора goto является аналогом вычисляемого goto языка FORTRAN:

```
@label = ("OUT", "IN");
goto $label[l]; ·
```

В приведенном фрагменте кода выражение в операторе goto будет вычислено равным строке IN и именно на оператор с этой меткой будет передано управление.

Последняя форма оператора goto «ПОДПРОГРАММА обладает магическим свойством, как отмечают авторы языка. Она подставляет вызов указанной в операторе подпрограммы для выполняемой в данной момент подпрограммы. Эта процедура осуществляется подпрограммами AUTOLOAD (), которые загружают одну подпрограмму, скрывая затем, что на самом деле сначала была вызвана другая подпрограмма.

Описание оператора goto приведено нами исключительно для полноты изложения. В программах его следует избегать, так как он делает логику программы более сложной и запутанной. Намного лучше использовать структурированные команды управления потоком вычислений next, last и redo. Если в процессе программирования выяснится, что не обойтись без оператора безусловного перехода, то это будет означать только одно: на этапе проектирования программы она была не достаточно хорошо структурирована. Вернитесь снова к этапу проектирования и постарайтесь реструктурировать ее таким образом, чтобы не требовалось использовать оператор goto.

В этом разделе мы познакомились с основными операторами языка Perl, которые используются для написания программ. Узнали, что операторы могут быть простыми и составными.

Выполнением простых операторов можно управлять с помощью модификаторов, которые вычисляют простой оператор при выполнении некоторого условия. Некоторые модификаторы организуют повторное вычисление в цикле оператора, к которому они применены.

Составные операторы представляют собой операторы, управляющие потоком вычислений в программе. Они определяются в терминах блоков. К ним относятся операторы выбора и цикла. Команды управления циклом позволяют изменить порядок выполнения операторов цикла.

Вопросы для самоконтроля

- Как определяются простые операторы Perl?
- Что такое модификаторы простых операторов и как они влияют на выполнение простых операторов?
- Перечислите составные операторы языка Perl.
- Что такое блок операторов и что он определяет в программе?
- Определите лексическую переменную.
- Какой оператор цикла удобнее для перебора всех элементов списка и почему?
- Какие команды используются в Perl для управления выполнением циклов?

- Как реализуются в Perl переключатели?

Упражнения

1. Какие из следующих операторов являются простыми, а какие составными: "abc" if 1; if ($a) { print $a;} do{ $a++; $b–;} until $b; while($a eq "a") { $a–;}

2. Найдите ошибку в программе:
perl -w $a = "true"; $b = "false"; if ($a)
$a = $b; elsif ($b) $b == $a;

3. Напишите программу, которая по заданному числу STEP печатает лесенку из STEP ступеней (каждая следующая ступень на один символ "-" шире предыдущей):
I (первая ·· ступень}
I (вторая ступень)
I (третья ступень)

4. Напишите программу, которая во вводимой пользователем строке подсчитывает количество слов, количество не пробельных символов и количество пробельных символов. Словом считать непрерывную последовательность алфавитно-цифровых символов, ограниченных пробельными символами ("\n", "\t", " "). Для завершения программы пользователь должен ввести пустую строку.

5. Напишите программу, которая читает целую величину ROW и печатает первые ROW строк треугольника Паскаля:

```
1
1 1
1 2 1
1 3 3 1
1 4 6 4 1
```