

Главная > Учебники > Учебник по Perl > Глава 4. Операции и выражения

Глава 4. Операции и выражения

 Учебник по Perl

[Содержание](#) [скрыть]

- [1 Арифметические операции](#)
 - [1.1 Бинарные арифметические операции](#)
 - [1.2 Унарные арифметические операции](#)
 - [1.3 Операции увеличения и уменьшения](#)
- [2 Операции конкатенации и повторения](#)
- [3 Операции отношения](#)
 - [3.1 Числовые операции отношения](#)
 - [3.2 Строковые операции отношения](#)
- [4 Логические операции](#)
- [5 Побитовые операции](#)
 - [5.1 Числовые operandы](#)
 - [5.2 Строковые operandы](#)
- [6 Операции присваивания](#)
- [7 Ссылки и операция разыменования](#)
- [8 Операции связывания](#)
- [9 Именованные унарные операции](#)
- [10 Операции ввода/вывода](#)
 - [10.1 Операция `print`](#)
 - [10.2 Выполнение системных команд](#)
 - [10.3 Операция `<>`](#)
- [11 Разные операции](#)
 - [11.1 Операция диапазон](#)
 - [11.2 Операция запятая](#)
 - [11.3 Операция выбора](#)
- [12 Списковые операции](#)
- [13 Операции заключения в кавычки](#)
 - [13.1 Операция `q{}`](#)
 - [13.2 Операция `qq{}`](#)
 - [13.3 Операция `qx{}`](#)
 - [13.4 Операция `qw{}`](#)
 - [13.5 Операция "документ здесь"](#)
- [14 Выражения](#)
 - [14.1 Термы](#)
 - [14.2 Приоритет операций](#)
 - [14.3 Контекст](#)
- [15 Вопросы для самоконтроля](#)

16 Упражнения

Язык программирования, предоставляя возможность определения разнообразных типов данных, должен обеспечивать их обработку, т. к. его основной целью является реализация алгоритмов обработки данных. Выполнение допустимых действий над данными осуществляется с помощью набора определенных в языке программирования операций. *Операция* – это выполнение определенного действия над операндами, результатом которого является новое значение. С точки зрения математики операцию можно рассматривать как некую функцию, которая по заданным переменным (операндам) вычисляет новое значение. Все знают со школьной скамьи четыре основных арифметических действия, выполняемых над числами: сложение (+), вычитание (-), умножение (*) и деление (/). Эти действия (операции) мы всегда выполняем над двумя числами (операндами), получая в результате новое число. Язык программирования определяет не только арифметические операции над числовыми данными, но и операции, применимые к другим допустимым типам данных. Это могут быть операции над строками, массивами и т. д. Важно только одно, что есть операция, определяемая своим знаком, и есть участвующие в ней операнды, в совокупности позволяющие получить (вычислить) новое значение, которое может принадлежать к одному из допустимых типов. В качестве operandов можно использовать литералы, переменные и выражения, представляющие собой комбинации основных операций. Общий синтаксис операции можно представить следующим образом:

```
операнд знак_операции операнд
```

Для некоторых операций один из operandов может быть опущен. В этом случае операция называется одноместной или унарной, например, вычисление противоположного по знаку значения operand - $\$t$. Если в операции участвуют два operand, то операция называется двуместной или бинарной. Практически все допустимые операции являются унарными или бинарными, но в некоторых современных языках программирования определена од-на-единственная условная тернарная операция, требующая три operand. Значением этой операции является второй или третий operand, в зависимости от истинности и ложности первого:

```
operand_1 ? operand_2 : operand_3
```

Синтаксически *выражение* представляется в виде последовательности operandов и знаков операций, которая обычно интерпретируется слева направо учетом порядка старшинства операций, т. е. правил, определяющих, в какой последовательности выполняются присутствующие в выражении операции. Для изменения порядка выполнения используются круглые скобки. Результатом обработки интерпретатором выражения является некоторое вычисленное значение, которое используется в других конструкциях языка, реализующих алгоритм обработки данных.

В этой главе подробно рассматриваются операции языка Perl и их использование при конструировании выражений. В языке Perl определено несколько десятков операций. Их можно сгруппировать по типу выполняемых действий (арифметические, побитовые, логические, отношения,

присваивания), на какие данные они воздействуют (строковые, списковые, файловые), и не относящиеся ни к одному из перечисленных типов (операции запятая, ссылки, разыменования и выбора). Следуя именно такой классификации операций языка Perl, мы и познакомим с ними нашего читателя. Одни операции будут разобраны подробно, о других мы только дадим общее представление, отнеся более детальное описание в другие главы нашей книги.

Арифметические операции

Все *арифметические операции* можно разбить на три группы: бинарные, унарные и увеличения/уменьшения. Их основное назначение – выполнить определенные вычисления над числовыми данными, но во всех арифметических операциях в качестве operandов могут выступать и строковые данные, причем не обязательно, чтобы они конвертировались в числовые данные.

Бинарные арифметические операции

Бинарные арифметические операции – это известные всем четыре арифметических действия: сложение (+), вычитание (-), умножение (*) и деление (/), к которым добавляются еще два: остаток от деления двух целых чисел (%) и возведение в степень (**). Примененные к числовым данным или строковым, которые содержат .правильные литералы десятичных чисел, они выполняют соответствующие арифметические действия (пример 4.1).

```
3.14 + 123; # Результат: 126.14
"3.14" + "123"; # Результат: 126.14
"3.14" + 123; # Результат: 126.14
"3.14" * 10; # Результат: 31.4
300 - 200; # Результат: 100
300 / 200; # Результат: 1.5
3 % 2; # Результат: 1
2 ** 3; . # Результат: 8
(-2) ** 3; # Результат: -8 '
2 ** (-3); # Результат: 0.125
2.5 ** 1.5; # Результат: -23.95284707521047
```

Как видим, бинарные арифметические операции "работают" именно так, как мы привыкли их использовать в обычных арифметических вычислениях в нашей повседневной жизни.

Замечание

Если operand в операции получения остатка от деления целых чисел (%) является вещественным числом с дробной частью, то он преобразуется к целому простым отбрасыванием дробной части, после чего операция выполняется над целыми числами.

Замечание

Нельзя возводить отрицательное число не в целую степень. Если такое случается, то интерпретатор не выдает никакой ошибки, но результатом такой операции является нуль: (-2.5)
 $** (1.3) = 0.$

В качестве operandов бинарных арифметических операций можно использовать строки, не содержащие правильные числовые литералы. В этом случае интерпретатор попытается выделить, начиная с первого символа, из содержимого строки число и использовать его в качестве соответствующего операнда заданной операции. Если не удается выделить правильный числовой литерал, то operand принимает значение, равное 0. Подобные ситуации демонстрируются в примере 4.2.

```
"3f14" .+ "12-30"; # Результат: 15 ("3" + "12")
"a!20" + "12-30"; # Результат: 12 ("0" + "12")
."a!20" + "-0012-30"; # Результат: -12 ("0" + "-12")
```

Замечание

Если установить режим отображения предупреждающих сообщений интерпретатора (ключ `-w`), то при попытке использовать в бинарных арифметических операциях строки, не содержащей правильные числовые литералы, будет отображено сообщение вида:

```
Argument "a!20" isn't numeric in add at D:\EXAMPLE1.PL line 2.
```

Бинарные арифметические операции выполняются в скалярном контексте. Это означает, что operandами должны быть скалярные переменные, а переменные массивов скаляров и хеш-массивов принимают значения, равные, соответственно, количеству элементов массивов скаляров или количеству использованных в хеш-таблице записей в соответствии с требованиями скалярного контекста (пример 4.3).

```
@m = (2, 4, 6, 8, 10);
%ml = ( 1 => "a", 2 => "b");
$n = 100;
$n + @m; # Результат: 105 (100 + 5)
@m + %ml; # Результат: 7 (5+2)
```

Замечание

В скалярном контексте хеш-массив принимает строковое значение, состоящее из числа использованных участков записей в хеш-таблице и числа выделенных участков записей, разделенных символом "/" (см. главу 3). Используемое в арифметических операциях число получается выделением из этой строки числового литерала, который как раз и соответствует количеству использованных в хеш-таблице записей.

Унарные арифметические операции

В языке Perl есть только две унарные арифметические операции (+) и (-). Унарный плюс +, примененный к данным любого типа, представленным литералами или своими переменными, не имеет никакого семантического эффекта. Он полезен перед выражением в круглых скобках, стоящим непосредственно после имени функции, если необходимо чисто визуально акцентировать тот факт, что функция фактически является списковой операцией.

(Об использовании унарного плюса в вызовах функций см. раздел 4.14.2 и главу 11.)

Унарный минус (-) выполняет арифметическое отрицание числового операнда. Это означает, что если число было отрицательным, то оно станет положительным, и наоборот. Если операндом является идентификатор, то результатом выполнения этой операции будет строка, состоящая из символа "-", за которым следует идентификатор. Если операндом является строка, начинающаяся с символа минус или плюс, то результатом также будет строка, в которой минус заменен на плюс и наоборот. Для строк, не начинающихся с плюса или минуса, операция унарного минуса добавляет его первым символом в строку. Все перечисленные случаи употребления унарного минуса показаны в примере 4.4.

```
-'12.09'; # Результат: -12.09
-(-12.09); # Результат: 12.09
-id; # Результат: '-id'
-'+id"; # Результат: '-id' 
-"-id"; # Результат: "+id" 
-'a!20'; # Результат: '-a!20'
```

Операции увеличения и уменьшения

Операции увеличения (++) и уменьшения (--) аналогичны таким же операциям в языке С. (Авторы языка Perl не скрывают, что они многое заимствовали из этого языка.) Результат этих операций зависит от того, стоят ли они перед (префиксная форма) или после переменной (постфиксная форма). При использовании префиксной формы они, соответственно, увеличивают или уменьшают числовое значение переменной на единицу до возвращения значения. Постфиксная форма этих операций изменяет числовое значение переменной после возвращения ими значения. Действие этих операций на числовые переменные иллюстрируется примером 4.5 (операторы фрагмента программы выполняются последовательно).

```
$п = 10.7; # Начальное значение
$infl = -$п; # Результат:$infl = 9.7 и $п=9.7
$inf2 = ++$п; # Результат: $inf2 = 10.7 и $п = 10.7
$postl = $п--; # Результат: $postl = 10.7 но $п = 9.7
$post2 = $п++; # Результат: $post2 = 9.7 но $п = 10.7
```

Операция увеличения (префиксная и постфиксная), примененная к переменной, содержащей строку определенного вида, выполняется несколько необычно. Если строка состоит только из латинских букв, то возвращаемым значением операции увеличении будет строка, в которой последняя буква заменена на следующую по порядку букву алфавита, причем строчная заменяется строчной, а прописная прописной. Если строка завершается идущими подряд буквами "z" или "Z", то все они заменяются соответственно на "a" или "A", а стоящая перед ними в строке буква заменяется на следующую букву алфавита. Если вся строка состоит из букв "z" и "Z", то кроме замены этих букв в соответствии с *предыдущим* правилом, перед ними добавляется строчная или прописная буква "a" в зависимости от того, строчная или прописная буква "z" стояла первой в строке.

Аналогичные действия осуществляются, если строка завершается последовательностью цифр: последняя цифра увеличивается на единицу. Если строка завершается идущими подряд цифрами 9, то все они заменяются на 0, а примыкающий к ним символ "увеличивается" на единицу: для буквы он переходит в следующий по алфавиту, а для цифры в следующую по порядку цифру. Если последовательность целиком состоит из девяток, то все они заменяются на нули, перед которыми добавляется единица. Префиксная и постфиксная формы операции действуют как обычно. Несколько иллюстраций этих операций представлены в примере 4.6.

```
$s = "abc"
$s1 = ++$s; # Результат: $s1 = "abd"
$s = "abC";
$s1 = ++$s; # Результат: $s1 = "abD"
$s = "abz";
$s1 = ++$s; # Результат: $s1 = "aca"
$s = "abzzz";
$s1 = ++$s; # Результат: $s1 = "acaAa"
$s = "ab09";
$s1 = ++$s; # Результат: $s1 = "ab10"
$s = "99"; .
$s1 = ++$s; # Результат: $s1 = "100"
```

Замечание

Операция уменьшения (-) работает со специальными строками так же, как и с обычными. Осуществляется попытка выделить числовой литерал, начиная с первого символа. Если такое оказывается возможным, то числовое значение строки приравнивается выделенному числовому литералу, если нет – ее значение считается равным 0. После этого применяется операция уменьшения к вычисленному числовому значению строки.

Операции конкатенации и повторения

Бинарная операция *конкатенации*, или *соединения* объединяет два строковых операнда в одну строку. Знаком этой операции служит точка ":":

```
"one_string"."two_string"; # Результат: "one_stringtwo_string"
```

В новой строке содержимое первого операнда и содержимое второго операнда соединяются без пробела между ними. Обычно эта операция используется для присваивания переменной некоторого нового значения. Если необходимо соединить две или более строки со вставкой пробелов между ними, то следует воспользоваться операцией *join* (см. гл. 10 "Работа со строками"). Можно, однако, для соединения строк со вставкой пробела (или любого другого символа между ними) воспользоваться свойством подстановки значения скалярной переменной в строку, ограниченную двойными кавычками: \$s1 = "one_string"; \$s2 = "two_string"; \$s = "\$s1 \$s2"; # Значение \$s: "one_string two_string"

Можно использовать операцию конкатенации строк последовательно в одном выражении для соединения нескольких строк:

```
$s1 = "one";
$s2 = "two";
$s3 = "three";
$s = $s1.$s2.$s3; # Значение $s: "onetwothree"
```

Операцию конкатенации можно применять и к числовым литералам, и к числовым данным, хранящимся в скалярных переменных. Результатом будет строка, содержащая символьные представления двух чисел:

```
$n1 = 23.5;
$n2 = 3e01;
$n = $n1.$n2; t Значение $n: "23.530"
$n = 23.5.3e01; # Значение $n: '"23.530"
```

Заметим, что последний оператор выглядит несколько экзотично и его семантика не определяется с первого взгляда.

Для работы со строками в языке Perl предусмотрена еще одна операция – *повторение строки* x (просто символ строчной буквы "x"). Эта бинарная операция создает новую строку, в которой строка, заданная левым операндом, повторяется определяемое правым операндом количество раз:

```
"aA" x 2; # Результат: "aAaA"
10.0 x "3"; # Результат: "101010"
101e-1 x 3; # Результат: "101010" $n = 010;
$n x 2; # Результат: "88"
```

```
10.1 x 3.9; # Результат: "10.110.110.1"
"101e-1" x 2; # Результат: "101e-1101e-1"
```

Обратим внимание, что в качестве левого операнда можно использовать и числовые литералы, и переменные, содержащие числовые данные. Правым операндом, задающим число повторений, может быть любое число или строка, содержащая правильное десятичное число.

Эта операция удобна, если надо напечатать или отобразить на экране монитора повторяющийся символ или последовательность символов. Например, следующий оператор выведет на экран монитора строку, целиком состоящую из символов подчеркивания: `print "_" x 80;`

Левым операндом этой операции может быть список, заключенный в круглые скобки. В этом случае операция повторения x работает как повторитель списка, т. е. ее результатом будет список, в котором список левого операнда повторяется заданное правым операндом количество раз:

```
(1) x 3; # Результат: (1, 1, 1) (1, 2) x 2; # Результат: (1, 2, 1, 2)
```

Это пример использования операции Perl в разных контекстах: скалярном и списковом (о контекстах мы поговорим ниже в этой же главе). Операция повторения в списковом контексте удобна для задания массива скаляров с одинаковыми значениями элементов или групп элементов:

```
@аггай = ("а", "б") x 2; # Результат: @аггай = ("а", "б", "а", "б") @аггай = ("а") x 3; #
Результат: @аггай = ("а", "а", "а")
```

Аналогично, эту операцию можно использовать для инициализации хеш-массива одинаковыми значениями:

```
@keys = ( one, two, three); # Определение ключей хеш-массива. @hash{@keys} = ("а") x @keys; #
Инициализация значений хеш-массива.
```

В последнем операторе присваивания в правой части массив скаляров `@keys` используется в списковом контексте и представляет список своих значений, тогда как в левой части он используется в скалярном контексте и имеет значение, равное числу своих элементов.

Знак операции повторения x 'следует отделять пробелами от operandов, так как иначе он может быть воспринят интерпретатором, как относящийся к лексеме, а не представляющий операцию повторения. Например, при синтаксическом разборе строки

```
$пх$m;
```

интерпретатор определит, что в ней идут подряд две переменные `$пх` и `$м`, а не операция повторения содержимого переменной `$п`, что приведет к синтаксической ошибке.

Операции отношения

Для сравнения скалярных данных или значений скалярных переменных язык Perl предлагает набор бинарных операций, вычисляющих отношения равенства, больше, больше или равно и т. п. между своими operandами, поэтому эту группу операций еще называют *операциями отношения*. Для сравнения числовых данных и строковых данных Perl использует разные операции. Все они представлены в табл. 4.1.

Таблица 4.1. Операции отношения

Операция	Числовая	Строковая	Значение
Равенство	<code>==</code>	<code>eq</code>	Истина, если operandы равны, иначе ложь
Неравенство	<code>!=</code>	<code>ne</code>	Истина, если operandы не равны, иначе ложь
Меньше	<code><</code>	<code>lt</code>	Истина, если левый operand меньше правого, иначе ложь
Больше	<code>></code>	<code>gt</code>	Истина, если левый operand больше правого, иначе ложь
Меньше или равно	<code><=</code>	<code>le</code>	Истина, если левый operand больше правого или равен ему, иначе ложь
Больше или равно	<code>>=</code>	<code>ge</code>	Истина, если правый operand больше левого или равен ему, иначе ложь
Сравнение	<code><=></code>	<code>cmp</code>	0, если operandы равны 1, если левый operand больше правого -1, если правый operand больше левого

Результатом операций отношения (кроме последней сравнения) является Истина, значение 1, или Ложь, пустая строка "".

Замечание

Значение истина в арифметических операциях интерпретируется как число 1, а в строковых как строка "1". Значение ложь в арифметических операциях интерпретируется как число 0, а в строковых как пустая строка " ".

Числовые операции отношения

Числовые операции отношения применяются к числовым данным, причем один или оба operandы могут задаваться строкой, содержащей правильное десятичное число. Если в числовых операциях отношения какой-либо из operandов задан строкой, содержимое которой не представляет правильное десятичное число, то его значение принимается равным 0 и отображается предупреждение о некорректном

использовании операнда в числовые операции отношения (если включен режим отображения предупреждений интерпретатора Perl). Смысл операций отношения для числовых данных соответствует обычным математическим операциям сравнения чисел (пример 4.7).

```
123 > 89; # Результат: 1 (истина)
123 < 89; # Результат: "" (ложь)
123 <= 89; # Результат: "" (ложь)
123 != 89; # Результат: 1 (истина)
89 <= 89; # Результат: 1 (истина)
23 >= 89; # Результат: "" (ложь)
23 <=> 89; # Результат: -1 (правый операнд больше левого)
89 <=> 23; # Результат: 1 (правый операнд больше левого)
```

Применение числовых операций сравнения не представляет сложности, однако при сравнении на равенство десятичных чисел с плавающей точкой могут проявиться эффекты округления, связанные с ограниченным количеством значащих цифр в мантиссе представления действительных чисел в компьютере и приводящие к "неправильной", с точки зрения пользователя работе операций сравнения. Пример 4.8 иллюстрирует подобную ситуацию.

```
#! perl -w
$z = 0.7;
$zz = 10+0.7-10; # Переменная $zz содержит число 0.7
# Печать строки "z равно zz", если равны значения переменных $z и $zz print "z равно zz\n" if
($z == $zz);
```

При попытке выполнить пример 4.8 мы с удивлением обнаружим, что наша программа ничего не напечатает. В чем же дело? Разгадка лежит в операторе вычисления значения переменной \$zz. При выполнении арифметических операций в результате ошибок округления получается значение 0.699999999999999 (можете вставить оператор печати переменной \$zz и убедиться в этом), хотя и близкое к 0.7, но не равное ему в точности. Следовательно, операция сравнения отработала верно!

Совет

Не используйте операцию сравнения на равенство вещественных чисел, ее результат может не соответствовать ожидаемому с точки зрения математики. Если необходимо проверить равенство двух вещественных чисел, то лучше сравнивать абсолютное значение их разности с некоторым очень маленьким числом (в зависимости от требуемой точности):

```
abs($a-$b) <= 0.0000001; # Проверка равенства
```

Строковые операции отношения

Сравнение строковых данных базируется на их упорядочении в соответствии с таблицей кодов ASCII, т. е. символ с меньшим кодом ASCII предшествует символу с большим кодом. Сравнение строк осуществляется посимвольно слева направо. Это означает, что если равны первые символы строк, то сравниваются вторые и если они равны, то сравниваются третьи и т. д. Причем, если строки разной длины, то в конец строки меньшей длины добавляется недостающее для равенства количество символов с кодом 0. Следует отметить, что в отличие от некоторых других языков программирования в Perl замыкающие строку пробельные символы являются значимыми при сравнении строк. В примере 4.9 показаны сравнения строк, иллюстрирующие изложенные правила.

```
"A" lt "a";          # Результат: истина (код "A" - \101, код "a" - \141)
```

```
"a" lt "aa";        # Результат: истина (к строке "a" добавляется символ
                      # с кодом \000, который меньше кода \141
                      # второго символа "a" строки правого операнда)
```

```
"a" lt "a ";        # Результат: истина (к строке "a" добавляется символ
                      # с кодом \000, который меньше кода \040
                      # замыкающего пробела строки правого операнда)
```

```
"12" lt "9";        # Результат: истина (код "1" - \061, код "9" - \071)
```

```
" 9" eq "09";       # Результат: ложь (код " " - \040, код "0" - \060)
```

Обратим внимание на две последние операции сравнения строковых литералов. Содержимое их operandов может быть преобразовано в правильные числа, и поэтому к ним применимы аналогичные числовые операции отношения. Однако их результат будет существенно отличаться от результата выполнения строковых операций отношения. При использовании операции < в предпоследнем выражении результат будет Ложь, а если в последнем выражении применить операцию ==, то результат будет Истина. Об этом всегда следует помнить, так как Perl автоматически преобразует символьные данные в числовые там, где это необходимо.

Логические операции

Рассмотренные в предыдущем параграфе операции сравнения используются в условном операторе if (онем и других операторах Perl в следующей главе) для организации ветвления в программе. Однако, иногда желательно проверять одновременно результаты нескольких операций сравнения и предпринимать соответствующие алгоритму действия. Можно подобную ситуацию запрограммировать с помощью вложенных операторов if, а можно в одном операторе использовать сложное выражение, результатом вычисления которого будет, например, истинность двух или более каких-либо операций сравнения. Для формирования подобных проверок и служат *логические операции* языка Perl.

В языке определены бинарные операции логического сравнения | 1 (ИЛИ), `s &` (И) и унарная операция логического отрицания !. Их действие аналогично действию соответствующих математических операций исчисления предикатов. Результатом операции || (логическое ИЛИ) является Истина, если истинен хотя бы один из operandов, в остальных случаях она возвращает Ложь (остальные случаи представляют единственный вариант, когда оба операнда ложны). Операция логического И && возвращает в качестве результата Истину, только если оба операнда истинны, в противном случае ее результат Ложь. Операция логического отрицания ! работает как переключатель: если ее operand истинен, то она возвращает Ложь, если operand имеет значение Ложь, то ее результатом будет Истина.

Замечание

В языке Perl нет специальных литералов для булевых значений Истина и Ложь. В качестве значения Истина принимается любое скалярное значение, не равное нулевой строке "" или числу 0 (а также его строковому эквиваленту "0"). Естественно, нулевая "" строка и 0 (вместе с его строковым эквивалентом "0") представляют значение Ложь.

Начиная с Perl 5.001, в язык были введены логические операции `og`, `and`, `not` и `xog`. Первые три полностью аналогичны логическим операциям ||, && и !, тогда как операция `xog` реализует исключающее ИЛИ:

```
Истина xog Истина = Ложь
Истина xog Ложь = Истина
Ложь xog Истина = Истина
Ложь xog Ложь = Ложь
```

Единственное отличие этих логических операций от рассмотренных ранее заключается в том, что они имеют наименьший приоритет при вычислении сложных выражений.

(Старшинство, или приоритет операций при вычислении сложных выражений, рассматривается в разделе 4.14.2 этой главы.)

В Perl вычисление логических операций ИЛИ и И осуществляется по "укороченной схеме". Это непосредственно связано со смыслом этих операций. Если при вычислении операции ИЛИ определено, что значение ее первого операнда Истина, то при любом значении второго операнда результатом всей операции будет Истина, поэтому нет смысла вообще вычислять второй operand. Аналогично для операции логического И: если значение первого операнда Ложь, то результат всей операции Ложь вне зависимости от значения второго операнда. В отличие от операций отношения, результатом которых может быть 0 (или пустая строка "") или 1, соответствующие булевым значениям Ложь и Истина, результатом логических операций является значение последнего вычисленного операнда. Пример 4.10 иллюстрирует вычисление логических операций.

```
$op1 = 0; $op2 = "s"; $op3 = ""; $op4 = 25; $op5 = "0";
$op4 II $op2; # Результат: истина. Значение: 25.
$op2 I| $op4; # Результат: истина. Значение: "s".
$op1 && $op2; # Результат: ложь. Значение: 0.
$op2 && $op4; # Результат: истина. Значение: 25.
!$op2; # Результат: ложь. Значение: "".
not $op3; # Результат: истина. Значение: 25.
$op4 and $op5; # Результат: ложь. Значение: "".
```

Свойство логических операций языка Perl вычисляться по "укороченной схеме" можно использовать для управления некоторыми исключительными ситуациями, возникающими в программе в процессе вычислений. Например, можно достаточно элегантно избежать деления на нуль с помощью операции логического ИЛИ:

```
($x = 0) II ($t = 1/$x);
```

При вычислении результата этой операции сначала вычисляется левый операнд, который сравнивает значение переменной \$x с нулем. Если это значение действительно равно нулю, то результатом операции сравнения будет Истина, а поэтому второй operand операции логического ИЛИ не вычисляется, так его значение не влияет на результат выполнения логической операции, и не возникает ситуации деления на нуль. Если значение переменной \$x не равно нулю, то результатом вычисления первого операнда операции | | будет Ложь, и обязательно будет вычисляться ее второй operand, в котором осуществляется деление на не равную нулю переменную \$x.

Побитовые операции

Данные в компьютере представляются в виде последовательности битов. В языке Perl определены бинарные операции *побитового логического сравнения* целых чисел и строк: & (И), | (ИЛИ) и ^ (исключающее ИЛИ), а также унарная операция логического отрицания ~. Результат их вычисления зависит от того, к данным какого типа они применяются: числовым или строковым. Эти операторы различают числовые данные и строки, содержимое которых может быть преобразовано в число.

Кроме логических операций побитового сравнения, две операции сдвигают влево («) и вправо (») биты в представлении целых чисел. Эти операторы не работают со строками.

Числовые операнды

Если хотя бы один operand в бинарных побитовых операциях является *числом*, то содержимое второго операнда также должно быть *числом*. Операнд, являющийся строкой символов, преобразуется в *числовое значение*. В случае несоответствия содержимого строки десятичному числу ее значение принимается равным 0 и отображается предупреждающее сообщение, если установлен соответствующий режим работы интерполятора. Все числовые операнды преобразуются к целым числам простым отбрасыванием дробной части, никакого округления не происходит.

Чтобы понять сущность побитовых операций над числовыми данными, необходимо представлять, как хранятся в программе целые числа. При задании чисел мы можем использовать одно из трех представлений: десятичное, восьмеричное или шестнадцатеричное. Однако в компьютере числа не хранятся ни в одном из указанных представлений. Они переводятся в двоичные числа – числа с основанием 2, цифры которых называются битами. Двоичные числа представляют собой запись чисел в позиционной системе счисления, в которой в качестве основания используется число 2. Таким образом, двоичные цифры, или биты, могут принимать значения только 0 или 1. Например, десятичное число 10, переведенное в двоичное, представляется в виде `1010`. Для обратного перевода этого числа в десятичную форму представления следует, в соответствии с правилами позиционной системы счисления, произвести следующие действия:

$$1 * (2^{**3}) + 0 * (2^{**2}) + 1 * -(2^{**1}) + 0 * (2^{**0})$$

Язык Perl гарантирует, что все целые числа имеют длину 32 бит, хотя на некоторых машинах они могут представляться и 64 битами. Именно с двоичными представлениями целых чисел и работают все побитовые операции, преобразуя и отрицательные, и положительные числа к целому типу данных.

Операция `&` побитового логического И сравнивает каждый бит правого операнда с соответствующим битом левого операнда. Если оба сравниваемых бита имеют значение 1, то соответствующий бит результирующего значения операции устанавливается равным 1, в противном случае значением бита результата будет 0. В качестве примера рассмотрим следующее выражение

`45.93 & 100`

Прежде всего, правый operand преобразуется к целому 45, двоичное представление которого будет

`0000000000000000000000000000101101`

Двоичное представление левого операнда 100 будет иметь вид

`00000000000000000000000000001100100`

Результатом побитового логического И будет следующая последовательность битов

`0000000000000000000000000000100100`

Она соответствует десятичному целому числу 36. Следовательно, значением выражения `45.93 & 100` будет целое число 36.

Замечание

В этом примере мы специально использовали 32-битное представление целых чисел, хотя для бинарных побитовых операций лидирующие нули не имеют значения. Они существенны только при операциях побитового логического отрицания и сдвигов.

При побитовой операции логического ИЛИ | бит результата устанавливается равным 1, если хотя бы один из сравниваемых битов равен 1. Операция побитового логического ИЛИ для тех же двух чисел

```
45.93 | 100
```

даст результат равный юэ, так как при применении побитового логического ИЛИ к operandам

```
00000000000000000000000000000000101101 (десятичное 45)
```

и

```
000000000000000000000000000000001100100 (десятичное 100)
```

дает следующую цепочку битов

```
000000000000000000000000000000001101101 (десятичное 109:
```

```
2**6+2**5+2**3+2**2+2**1)
```

Побитовое исключающее ИЛИ Δ при сравнении битов дает значение 1 тогда, когда *точно один* из operandов имеет значение равное 1. Следовательно, $1\Delta 1=0$ и $0\Delta 0=0$, в остальных случаях результат сравнения битов равен 0. Поэтому для тех же чисел результатом операции 45.93 Δ юо будет десятичное число 73.

Операция логического отрицания ~ является унарной и ее действие заключается в том, что при последовательном просмотре битов числа все значения 0 заменяются на 1, и наоборот. Результат этой операции существенно зависит от используемого количества битов для представления целых чисел. Например, на 32-разрядной машине результатом операции ~ 1 будет последовательность битов

```
11111111111111111111111111111110
```

представляющая десятичное число $4294967294 = 2^{31} + 2^{30} + \dots + 2^1$, тогда как на 16-разрядной машине эта же операция даст число $6534 = 2^{15} + 2^{14} + \dots + 2^1$.

Бинарные операции побитового сдвига осуществляют сдвиг битов целого числа, заданного левым operandом, влево («<») или вправо («>») на количество бит, определяемых правым целочисленным operandом. При сдвиге вправо недостающие старшие биты, а при сдвиге влево младшие биты числа дополняются нулями. Биты, выходящие за разрядную сетку, пропадают. Несколько примеров операций сдвига представлено ниже:

```
# Битовое представление числа 22: (000000000000000000000000000010110)
```

```
22 >> 2 # Результат: (0000000000000000000000000000101) = 5
```

```
22<<2 # Результат: (0000000000000000000000000000101100) = 88
```

Все перечисленные операции работают и с отрицательными целыми числами, только при их использовании следует учитывать, что они хранятся в дополнительном коде. Двоичная запись неотрицательного целого числа называется прямым кодом. Обратным кодом называется запись, полученная поразрядной инверсией прямого кода. Отрицательные целые числа представляются в памяти компьютера в дополнительном коде, который получается прибавлением единицы к младшему разряду обратного кода. Например, представление числа -1 получается следующим образом:

```
00000000000000000000000000000001 # положительное число 1 111111111111111111111111111111110 #
обратный код числа 1 11111111111111111111111111111111 # добавляем к младшему разряду 1
# и получаем представление числа -1
```

Именно с этим кодом числа -1 будут работать все побитовые операции, если оно будет задано в качестве операнда одной из них.

Внимание

В языке Perl, как отмечалось в гл. 3, в арифметических операциях используется представление всех чисел в виде чисел с плавающей точкой удвоенной точности. Там же говорилось, что целое число можно задавать с 15 значащими цифрами, т.е. максимальное положительное целое число может быть 999 999 999 999 999. Но это число не имеет ничего общего с *представлением* целых чисел в компьютере, для которых может отводиться 64, 32 или 16 битов, в зависимости от архитектуры компьютера. Во всех побитовых операциях можно предсказать результат только если операндами являются целые числа из диапазона $-2^{32}-1 \dots 2^{32}-1$, так как ясен алгоритм их представления. Вещественные числа, не попадающие в этот диапазон, преобразуются к целым, но алгоритм их преобразования не описан авторами языка.

Строковые операнды

Если *оба* операнда являются строковыми литералами или переменными, содержащими строковые данные, то операции побитового логического сравнения сравнивают соответствующие биты кода каждого символа строки. Для кодирования символов используется таблица ASCII-кодов. Если строки разной длины, то при сравнении полагается, что строка меньшей длины содержит необходимое число недостающих символов с восьмеричным кодом \ooo. Например, результатом сравнения двух строк "++" и "з" с помощью операции | побитового логического ИЛИ будет строка ",-+". Операнды этой операции представляются следующими битовыми последовательностями (каждый символ представляется 8 битами):

```
00101011 00101011 # Восьмеричный код символа "+" равен 053. 00110011 # Восьмеричный код
символа "3" равен 063.
```

Вторая строка дополняется восемью нулевыми битами и последовательно для каждой пары соответствующих бит двух строк одинаковой длины выполняется операция логического ИЛИ. Результатом выполнения этой процедуры является битовая последовательность

```
00111011 00101011
```

При ее интерпретации как строки символов получается последовательность двух символов с восьмеричными кодами 07 3 и 053, которые соответствуют символам ";" и "+"

Следует отметить, что в случае двух строковых операндов, содержимое которых можно преобразовать в числовые данные, подобное преобразование не происходит, и побитовые операции логического сравнения выполняются как с обычными строковыми данными:

```
45.93 I 100 # Результат: число 109.  
"45.93" I 100 # Результат: число 109.  
45.93 I "100" # Результат: число 109.  
"45.93" I "100" # Результат: строка "55>93".
```

В первых трех операциях этого примера строки преобразуются в числа, а потом выполняется соответствующая операция побитового логического ИЛИ двух чисел.

Выполнение операции побитового логического отрицания `~` для строки ничем не отличается от соответствующей операции отрицания для чисел с той лишь разницей, что применяется она к битовому представлению символов строки:

```
~"1" # Результат: "+".  
~"ab" t Результат: "ЮЭ".
```

Операции присваивания

Присваивание переменной какого-либо значения, определенного литералом, или присваивание одной переменной значения другой переменной является наиболее часто выполняемым действием в программе, написанной на любом языке программирования. В одних языках это действие определяется с помощью оператора, а в других – с помощью операции. Отличие заключается в том, что в языках, где присваивание является операцией, оно может использоваться в выражениях как его составная часть, так как любая операция вычисляет определенное значение, тогда как оператор всего лишь производит действие. В языке Perl присваивание является операцией, которая возвращает правильное `Ivalue`. Что это такое, мы разъясним буквально в следующих абзацах.

Операция присваивания `=`, с которой читатель уже немного знаком, является бинарной операцией, правый operand которой может быть любым правильным выражением, тогда как левый operand должен определять область памяти, куда операция присваивания помещает вычисленное значение правого operand. В этом случае и говорят, что левый operand должен быть правильным `Ivalue` (от английского `left value` – левое значение). А что мы можем использовать в программе для обозначения области памяти? Правильно, переменные. Следовательно, в качестве левого operandа операции присваивания можно использовать переменную любого типа или элемент любого массива. (В языке Perl существуют и другие объекты, которые можно использовать в качестве левого operand)

операции присваивания, но об этом в свое время.) Следующая операция простого присваивания `$a = $b+3;`

вычислит значение правого операнда и присвоит его переменной `$a`, т.е. сохранит в области памяти, выделенной для переменной `$a`. Возвращаемым значением этой операции будет адрес области памяти переменной `$a` (правильное *Ivalue*), или говоря проще, имя скалярной переменной `$a`, которое снова можно использовать в качестве левого операнда операции присваивания. Таким образом, в языке Perl следующая операция присваивания

```
($a = $b) = 3;
```

является синтаксически правильной и в результате ее вычисления переменной `$a` будет присвоено значение 3, так результатом вычисления операции присваивания `$a = $b` будет присвоение переменной `$a` значения переменной `$b`, а возвращаемым значением можно считать *переменную \$a*, которой в следующей операции присваивается значение 3. Читатель спросит: "А зачем городить такие сложности, если тот же самый результат можно получить простой операцией присваивания `$a = 3`?". Действительно, замечание справедливо. Но на этом примере мы показали, как можно использовать операцию присваивания в качестве правильного *Ivalue*. Более интересные примеры мы покажем, когда определим составные операции присваивания, заимствованные из языка C.

Синтаксические правила языка Perl позволяют осуществлять присваивание одного и того же значения нескольким переменным в одном выражении:

```
$var1 = $var2 = $var1[0] = 34;
```

Очень часто при реализации вычислительных алгоритмов приходится осуществлять разнообразные вычисления с использованием значения некоторой переменной и результат присваивать этой же переменной. Например, увеличить на 3 значение переменной `$a` и результат присвоить этой же переменной `$a`. Это действие можно реализовать следующей операцией присваивания:

```
$a = $a + 3;
```

Однако, язык Perl предлагает более эффективный способ решения подобных проблем, предоставляя в распоряжение программиста бинарную операцию *составного присваивания* `+=`, которая прибавляет к значению левого операнда, представляющего правильное *Ivalue*, значение правого операнда и результат присваивает переменной, представленной левым операндом. Таким образом, оператор составного присваивания

```
$a += 3; # Результат: $a = $a + 3
```

эквивалентен предыдущему оператору простого присваивания. Единственное отличие заключается в том, что его реализация эффективнее реализации простого присваивания, так как в составном операторе присваивания переменная `$a` вычисляется один раз, тогда как в простом ее приходится

вычислять дважды. (Под вычислением переменной понимается вычисление адреса представляемой ею области памяти и извлечение значения, хранящегося в этой области памяти.)

Для всех бинарных операций языка Perl существуют соответствующие составные операции присваивания. Все они, вместе с примерами их использования, собраны в табл. 4.2.

Таблица 4.2. Составные операции присваивания

Операция	Пример	Эквивалент с операцией простого присваивания
<code>**=</code>	<code>\$a **= 3;</code>	<code>\$a = \$a ** 3;</code>
<code>+=</code>	<code>\$a += 3;</code>	<code>\$a = \$a + 3;</code>
<code>-=</code>	<code>\$a -= 3;</code>	<code>\$a = \$a - 3;</code>
<code>.=</code>	<code>\$a .= "a";</code>	<code>\$a = \$a . "a";</code>
<code>*=</code>	<code>\$a *= 3;</code>	<code>\$a = \$a * 3;</code>
<code>/=</code>	<code>\$a /= 3;</code>	<code>\$a = \$a / 3;</code>
<code>%=</code>	<code>\$a %= 3;</code>	<code>\$a = \$a % 3;</code>
<code>x=</code>	<code>\$a x= 3;</code>	<code>\$a = \$a x 3;</code>
<code>&=</code>	<code>\$a &= \$b;</code>	<code>\$a = \$a & \$b;</code>
<code> =</code>	<code>\$a = 3;</code>	<code>\$a = \$a 3;</code>
<code>^=</code>	<code>\$a ^= 3;</code>	<code>\$a = \$a ^ 3;</code>
<code><<=</code>	<code>\$a <<= 3;</code>	<code>\$a = \$a << 3;</code>
<code>>>=</code>	<code>\$a >>= 3;</code>	<code>\$a = \$a >> 3;</code>
<code>&&=</code>	<code>\$a &&= \$b > 1;</code>	<code>\$a = \$a && \$b > 1;</code>
<code> =</code>	<code>\$a = \$b == 0;</code>	<code>\$a = \$a \$b == 0;</code>

Возвращаемым значением каждой из составных операций присваивания, как и в случае простого присваивания, является переменная левого Операнда (правильное *lvalue*), поэтому их можно использовать в любом операнде других операций присваивания (пример 4.11).

<code>\$b = 1;</code>	
<code>\$a = (\$b += 3);</code>	# Результат: \$a = \$b = 4

```
$a += ($b += 3); # Результат: $a = $a+$b+3
( ($a += 2) **= 2) -= 1; # Результат: $a = ($a+2)**2-1
```

Замечание

При использовании операции присваивания (простой или составной) в качестве левого операнда другой операции присваивания обязательно ее заключение в круглые скобки. Иначе может сгенерироваться синтаксическая ошибка, или выражение будет интерпретировано не так, как задумывалось. При наличии нескольких операций присваивания в одном выражении без скобок интерпретатор Perl начинает его разбор *справа*. Например, если последнее выражение примера 4.11 записать без скобок

```
$a += 2 **= 2 -= 1;
```

то при его синтаксическом анализе интерпретатор сначала выделит операцию присваивания

```
2 -= 1;
```

и сообщит об ошибке, так как ее синтаксис ошибочен (левый operand не является переменной или элементом массива).

Ссылки и операция разыменования

При выполнении программы Perl она, вместе с используемыми ею данными, размещается в оперативной памяти компьютера. Обращение к данным осуществляется с помощью символьических имен – переменных, что является одним из преимуществ использования языка высокого уровня типа Perl. Однако иногда необходимо получить непосредственно адрес памяти, где размещены данные, на которые мы ссылаемся в программе с помощью переменной. Для этого в языке определено понятие *ссылки*, или *указателя*, который содержит адрес переменной, т. е. адрес области памяти, на которую ссылается переменная. Для получения адреса переменной используется операция ссылка, знак которой "\" ставится перед именем переменной:

```
$t = '5;
$pt = \$t; # Ссылка на скалярную величину
```

Ссылки хранятся в скалярных переменных и могут указывать на скалярную величину, на массив, на хеш и на функцию:

```
@aggrav = (1,2,3);
$raggrav = \@aggrav; # Ссылка на массив скаляров
%hesh = (one=>1, two=>2, three=>3);
$pshesh = \%hesh; # Ссылка на массив скаляров
```

Если распечатать в программе переменные-ссылки `$pm`, `$ragray` и `$phash`, то мы увидим строки, подобные следующим:

```
SCALAR(0x655a74) ARRAY (0x655b10) HASH(0x653514)
```

В них идентификатор определяет тип данных, а в скобках указан шестнадцатеричный адрес области памяти, содержащей данные соответствующего типа.

Для получения содержимого области памяти, на которую ссылается переменная-указатель, требуется выполнить операцию *разыменования ссылки*. Для этого достаточно перед именем такой переменной поставить символ, соответствующий типу данных, на который ссылается переменная (`$`, `@`, `%`):

```
@keys = keys(%$phash); # Массив ключей хеша @values = values(%$phash); # Массив значений хеша
print "$$pm \n@$ragray \n@keys \n@values";
```

Этот фрагмент кода для определенных в нем переменных-ссылок на скаляр, массив и хеш напечатает их значения:

```
5 # Значение скалярной переменной $t
123 # Значения элементов массива скаляров @аггай
three two one # Ключи хеша %hash
321 # Значения хеша %hash
```

Использование описанной выше простой операции разыменования может приводить к сложным, трудно читаемым синтаксическим конструкциям при попытке получить значения элементов сложных конструкций: массива массивов, массива хешей и т. п. Для подобных целей в языке Perl предусмотрена бинарная операция `->`, левым операндом которой может быть ссылка на массив скаляров или хеш-массив, а правым операндом индекс элемента массива или хеша, значение которого необходимо получить: `print "$ragray->[0], .$ragray->[1], .$ragray->[2]\n"; print "$phash->{one}, $phash->{two}, $phash->{three}\n";`

Эти операторы напечатают значения элементов массива `@аггай` и хеша `%hash`. (Более подробно ссылки и операции разыменования рассматриваются в главе 9.)

Операции связывания

Операции сопоставления с образцом, используемые многими утилитами обработки текста в Unix, являются мощным средством и в языке Perl. Эти операции с регулярными выражениями включают поиск (`//`), подстановку (`s//`) и замену символов (`tr///`) в строке. По умолчанию они работают со строкой, содержащейся в системной переменной `$_`. Операции `=~` и `\~` связывают выполнение сопоставления с образцом над строкой, содержащейся в переменной, представленной левым операндом этих операций:

```

$_ = "It's very interesting!";
s/very/not/; # Переменная $_ будет содержать строку
# "It's not interesting!"
$m = "my string";
$m =~ s/my/our/; i Переменная $m будет содержать строку
tt "our string"

```

Возвращаемым значением операции `=~` является Истина, если при выполнении соответствующей ей операции сопоставления с образцом в строке была найдена последовательность символов, определяемая регулярным выражением, и Ложь в противном случае. Операция `! ~` является логическим дополнением к операции `=~`. Следующие два выражения полностью эквивалентны:

```
$m !~ m/my/our/; not $m =~ m/my/our/;
```

(Более подробно регулярные выражения и операции связывания рассматриваются в главе 10.)

Именованные унарные операции

В языке Perl определено большое количество встроенных функций, выполняющих разнообразные действия. Некоторые из них, с точки зрения синтаксического анализатора языка, на самом деле являются унарными операциями, которые и называют именованными унарными операциями, чтобы отличить их от унарных операций со специальными символами в качестве знаков операций (например, унарный минус `"-"`, операция ссылки `"\"`, логического отрицания `"!"` и т. д.). Некоторые из именованных унарных операций перечислены ниже:

```
chdir, cos, defined, goto, log, rand, rmdir, sin, sqrt, do, eval, return (Является ли функция унарной операцией, можно определить в приложении 1.)
```

К именованным унарным операциям относятся также все операции проверки файлов, синтаксис которых имеет вид:

```
-символ [имя_файла\дескриптор_файла]
```

Например, для проверки существования файла определена операция `-e`, выяснить возможность записи в файл можно операцией `-w`.

(Более подробно операции проверки файлов рассматриваются в главе 7.)

Операции ввода/вывода

Для взаимодействия и общения с внешним окружением в любом языке программирования предусмотрены операции ввода/вывода. Perl не является исключением. В нем определен ряд операций, обеспечивающих ввод и вывод данных в/из программы.

Операция `print`

С этой операцией вывода мы уже немного знакомы. Операция `print` – унарная операция, правым операндом которой служит задаваемый список значений, которые она отображает по умолчанию на экране монитора. Операцию, операндом которой является список, называют списковой операцией. Внешне ее можно задать как вызов функции, заключив ее operand в круглые скобки. Следующие операции эквивалентны:

```
print "@m", "\n", $m, "\n"; print("@m", "\n", $m, "\n");
```

(Более подробно эта операция рассматривается в главе 6.)

Выполнение системных команд

Операция заключения в обратные кавычки – это специальная операция, которая передает свое содержимое на выполнение операционной системы и возвращает результат в виде строковых данных:

```
$command = ~dir; # Переменная $command после выполнения операционной . # системой  
КОMaHfibi'dir' содержит результат ее # выполнения.
```

Содержимое строкового литерала в обратных кавычках должно быть, после подстановки значений переменных, которые могут в нем присутствовать, правильной командой операционной системы.

(Более подробно эта операция рассматривается в главе 6.)

Операция <>

При открытии файла с помощью функции `open()` одним из ее параметров является идентификатор, называемый дескриптором файла, с помощью которого можно в программе Perl ссылаться на файл.

Операция *ввода из файла* осуществляется заключением в угловые скобки его дескриптора `<дескриптор_файла>`. Результатом вычисления этой операции является строка файла или строки файла в зависимости от скалярного или спискового контекста ее использования. Следующие операторы иллюстрируют эту особенность данной операции:

```
open( MYFILE, "data.dat"); tt Открытие файла "data.dat" и назначение ему  
# дескриптора MYFILE  
$firstline = <MYFILE>; # Присваивание первой строки файла @remainder = <MYFILE>; # Оставшиеся  
строки файла присваиваются  
# элементам массива скаляров.
```

Дескриптор файла можно использовать и в операции `print`, организуя вывод не на стандартное устройство вывода, а в файл, представляемый дескриптором:

```
print MYFILE @m;
```

Особый случай представляет операция чтения из файла с пустым дескриптором `o`: информация считывается либо из стандартного файла ввода, либо из файлов, заданных в командной строке.

(Более подробно операции ввода/вывода из/в файл рассматриваются в главе 6.) (Работа с файлами более подробно рассматривается в главе 7.)

Разные операции

В этом параграфе собраны операции, которые не вошли ни в одну из рассмотренных нами групп операций. Две из них упоминались при описании массивов и хешей (операции диапазон и запятая), а третья является единственной тернарной операцией языка Perl (операция выбора).

Операция диапазон

Бинарная операция *диапазон* "..." по существу представляет две различных операции в зависимости от контекста, в котором она используется.

В *списковом* контексте, если ее operandами являются числа (числовые литералы, переменные или выражения, возвращающие числовые значения), она возвращает список, состоящий из последовательности увеличивающихся на единицу целых чисел, начинающихся со значения, определяемого левым operandом, и не превосходящих числовое значение, представленное правым operandом. Операцию диапазон часто используют для задания значений элементов массивов и хешей, а также их фрагментов (*см. главу 3*). Она удобна для организации циклов *for* и *foreach*:

```
# Напечатает числа от 1 до 5, каждое на новой строке, foreach $cycle (1..5){ print "$cycle\n";
}
# Напечатает строку "12345".
for(1..5){
print; '
```

(Операторы цикла рассматриваются в главе 5.)

Замечание

Если значение какого-либо operand'a не является целым, оно приводится к целому отбрасыванием дробной части числа.

Если левый operand больше правого operand'a, то операция диапазон возвращает пустой список. Подобную ситуацию можно отследить с помощью функции *defined* 0, возвращающей истину, если ее параметр определен, или простой проверкой логической истинности массива, элементам которого присваивались значения с помощью операции диапазон:

```
$min = 2; .
$max = ~2;
@array = ($min .. $max); # Массив не определен.
```

```
print "Эаггай аггау\n" if defined(@array); # Печати не будет!
print "Саггай аггау\n" if Эаггай; # Печати не будет!
```

Замечание

В операции диапазон можно использовать и отрицательные числа. В этом случае возвращается список отрицательных чисел, причем значение левого операнда должно быть меньше значения правого операнда:

```
(-5..5) # Список чисел: (-2, -1, 0, 1, 2). (-5..-10) # Пустой список.
```

Если operandами операции диапазон являются строки, содержащие буквенно-цифровые символы, то в списковом контексте эта операция возвращает список строк, расположенных между строками operandов с использованием лексикографического порядка:

```
@a = ("a".."d"); # Массив @a: "a", "b", "c", "d"
@a = ("01".."31"); # Массив @a: "01", "02", ..., "30", "31"
@a = ("a1".."d4"); # Массив Эа: "a1", "a2", "a3", "a4"
```

Если левый operand меньше правого, с точки зрения лексикографического порядка, то возвращается единственное значение, равное левому operandу.

Замечание

Операция диапазон не работает со строками, содержащими символы национальных алфавитов. В этом случае она всегда возвращает единственное значение, соответствующее левому operandу.

В *скалярном* контексте операция диапазон возвращает булево значение Истина или Ложь. Она работает как переключатель и эмулирует операцию запятая "," пакетного редактора sed и фильтра awk системы Unix, представляющую диапазон обрабатываемых строк этими программами.

Каждая операция диапазон поддерживает свое собственное булево состояние, которое изменяется в процессе ее повторных вычислений по следующей схеме. Она ложна, пока ложным остается значение ее левого operand. Как только левый operand становится истинным, операция диапазон переходит в состояние Истина и находится в нем до того момента, как ее правый operand не станет истинным, после чего операция снова переходит в состояние Ложь. Правый operand не вычисляется, пока операция находится в состоянии Ложь; левый operand не вычисляется, пока операция диапазон находится в состоянии Истина.

В состоянии Ложь возвращаемым значением операции является пустая строка, которая трактуется как булева Ложь. В состоянии Истина при повторном вычислении она возвращает следующее порядковое число, отсчет которого начинается с единицы, т. е. как только операция переходит в состояние Истина, она возвращает 1, при последующем вычислении, если она все еще находится в состоянии

Истина, возвращается 2 и т. д. В момент, когда операция переходит в состояние Ложь, к ее последнему возвращаемому порядковому числу добавляется строка "ко", которая не влияет на возвращаемое значение, но может быть использована для идентификации последнего элемента в диапазоне вычислений. Программа примера 4.12 и ее вывод, представленный в примере 4.13, иллюстрируют поведение оператора диапазон в скалярном контексте. Мы настоятельно рекомендуем внимательно с ними ознакомиться, чтобы "почувствовать", как работает эта операция.

```
#!/usr/bin/perl -w

$left = 3; # Операнд1 $right = 2; # Операнд2
# Заголовок таблицы
print "\$i\t\$left\t\$right\n";
print '-' x 48, "\n";
# Тест операции
for($i=1; $i <= 10; $i++) {
    $s = $left..$right;
    print "$i\t$s\t$left\t$right\n";
    $s10 = 3 if $i==5; # Когда переменная цикла $i равна 5, # $s10 устанавливается равной 3.
    if ($right==0) {} else {-$right}; # Уменьшение $right на 1, пока
    # $right не достигла значения 0.
    -$left; }
```

Замечание

В целях экономии времени мы не объясняем смысл незнакомых операторов примера 4.12, надеясь, что читатель сможет понять их смысл. Если все же это окажется для него сложным, мы рекомендуем снова вернуться к этой программе после прочтения главы 5.

\$1	Диапазон	Операнд1	Операнд2
1	1..0	3	2
2	1..0	2	1
3	1	1	0
4	2	0	0
5	3	-1	0
6	4..0	-2	2
7	1..0	-3	1

8	1	-4	0
9	2	-5	0
10	3	-6	0

Сделаем замечания относительно работы программы примера 4..12. На первом шаге цикла левый операнд операции диапазон истинен, следовательно сама операция находится в состоянии Истина и возвращает первое порядковое число (*i*). Но правый operand становится также истинным (*\$right* = 2), следовательно она переходит в состояние Ложь и к возвращаемому ей значению добавляется строка "E0". На втором шаге цикла левый operand истинен (*\$ieft* = 2) и операция переходит в состояние Истина, возвращая значение д, к которому опять добавляется строка "E0", так как истинный правый operand (*\$right* = 1) переводит операцию в состояние Ложь.

На третьем шаге операция становится истинной (*\$ieft* = 1), возвращая *i*, и правый operand со значением Ложь (*\$right* = 0) не влияет на ее состояние. На следующих шагах 4 и 5 правый operand остается ложным, а операция возвращает соответственно следующие порядковые числа 2 и з. На шаге 6 операция находится в состоянии Истина и возвращает 4, но правый operand, принимая значение Истина (*\$right* = 0), переводит ее в состояние Ложь, в котором к возвращаемому значению добавляется строка "E0" и т. д.

Подобное поведение, связанное с переходом из состояния Истина в состояние Ложь и одновременным изменением возвращаемого значения (добавлением строки "E0") эмулирует поведение операции запятая фильтра awk. Для эмуляции этой же операции редактора sed, в которой изменение возвращаемого значения осуществляется при следующем вычислении операции диапазон, следует вместо двух точек в знаке операции ".." использовать три точки "...". Результаты вывода программы примера 4.12, в которой осуществлена подобная замена, представлены в примере 4.14.

\$i	Диапазон	Операнд1	Операнд2
1	1	3	2
2	2E0	2	1
3	1	1	0
4	2	0	0
5	3	-1	0
6	4E0	-2	2
7	1	-3	1
8	2	-4	0

9	3	-5	0
10	4	-6	0

Еще одно достаточно полезное свойство операции диапазон в скалярном контексте, используемое при обработке строк файлов, заключается в том, что если какой-либо операнд этой операции задан в виде числового литерала, то он сравнивается с номером прочитанной строки файла, хранящейся в специальной переменной `$_`, возвращая булево значение Истина при совпадении и Ложь в противном случае. В программе примера 4.15 иллюстрируется такое использование операции диапазон. В ней осуществляется пропуск первых не пустых строк файла, печатается первая строка после пустой строки и после этого завершается цикл обработки файла.

```
#! perl -w
open{POST, "file.txt"} or die "Нельзя открыть файл file.txt!";
LINE:
while(<POST>) {
    $temp = 1../^$/; # Истина, пока строка файла не пустая.
    next LINE if ($temp); # Переход на чтение новой строки,
    # если $temp истинна.
    print $_; # Печать первой строки файла после не пустой
    last; # Выход из цикла
}
close(POST);
```

В этой программе для нас интересен оператор присваивания возвращаемого значения операции диапазон переменной `$temp`. Прежде всего отметим, что эта операция используется в скалярном контексте, причем ее левый operand представлен числовым литералом. На первом шаге цикла читается первая строка файла и специальной переменной `$_` присваивается ее номер который сравнивается со значением левого операнда операции диапазон. Результатом этого сравнения является булево значение Истина, и операция диапазон переходит в состояние Истина, в котором она и остается, если первая строка файла не пустая, так как операция поиска по образцу `/$/` возвращает в этом случае Ложь. Операция `next` осуществляет переход к следующей итерации цикла, во время которой читается вторая строка файла. Операция диапазон остается в состоянии Истина, если прочитанная строка файла пустая, увеличивая возвращаемое значение на единицу. Далее операция `next` снова инициирует чтение новой строки файла. Если строка файла пустая, то операция поиска по образцу возвращает значение Истина, переводя тем самым операцию диапазон в состояние Ложь, которое распознается при следующем ее вычислении, поэтому считывается еще одна строка файла (первая после пустой). Теперь операция `next` не выполняется, так как операция диапазон возвращает Ложь, печатается первая не пустая строка и операция `last` прекращает дальнейшую обработку файла. 1, л

Операция запятая

Бинарная операция запятая "," ведет себя по-разному в скалярном и списковом контексте.

В списковом контексте она является всего лишь разделителем между элементами списка:

```
@a = (1, 2); # Создается массив скаляров  
# и его элементам присваиваются значения 1 и 2.
```

В скалярном контексте эта операция полностью соответствует аналогичной операции языка С: вычисляется левый операнд, затем правый операнд, вычисленное значение которого и является возвращаемым значением этой операции. Если в предыдущем примере заменить массив @a скалярной переменной \$a, то ей будет присвоено значение 2:

```
$a = (1, 2); # Переменной $a присваивается значение 2.
```

Замечание

Надеемся, читателю теперь стало ясно, почему конструкторы массивов, о которых мы рассказывали в гл. 3, в скалярном и в списковом контексте ведут себя по-разному.

Для операции запятая в языке Perl существует удобный синоним – операция =>, которая полностью идентична операции запятая и удобна при задании каких-либо величин, которые появляются парами, например, ключ/значение в ассоциированных массивах. Правда, эта операция обладает еще одним свойством, достаточно удобным для ее использования при задании ассоциированных массивов: любой идентификатор, используемый в качестве ее левого операнда, интерпретируется как строка (см. раздел 3.4 главы 3).

Операция выбора

Единственная тернарная операция выбора

```
операнд1 ? операнд2 : операнд3
```

полностью заимствована из языка С и работает точно так же, как и ее двойник. Если операнд1 истинен, то возвращается значение операнд2, в противном случае операнда:

```
($п = 1) ? $a : $array;
```

Скалярный или списковый контекст, в котором используется эта операция, распространяется и на возвращаемое значение этой операции:

```
$a = $yes ? $b : @b; tt Скалярный контекст. Если возвращается  
# массив @b, то присваивается количество его t элементов.
```

Операцию выбора можно использовать в качестве левого операнда операции присваивания, если и второй, и третий ее operandы являются правильными lvalue, т. е. такими значениями, которым

можно присвоить какое-либо значение, например, именами переменных:

```
($a = $yes ? $b : @b) = @c;
```

В связи с тем, что результатом операции выбора может оказаться правильное lvalue, следует использовать скобки для уточнения ее operandов. Например, если в следующем выражении

```
($a % 3) ? ($a += 2) : ($a -= 2);
```

опустить скобки вокруг operandов

```
$a % 3 ? $a += 2 : $a -= 2;
```

то оно будет откомпилировано следующим образом

```
((($a % 3) ? ($a += 2) : $a). -= 2;
```

Списковые операции

Списковая операция – это операция над списком значений, причем список не обязательно заключать в круглые скобки.

Мы уже знакомы с одной из таких операций – операцией вывода на стандартное устройство `print`. Иногда мы говорили об этой операции как о функции – и это справедливо, так как всесписковые операции Perl выглядят как вызовы функций (даже их описание находится в разделе "Функции" документации по Perl), и более того, они позволяют заключать в круглые скобки список их параметров, что еще сближает их с функциями. Таким образом, списковую операцию (или функцию) `print` можно определить в программе любым из следующих способов:

```
print $a, "string", $b; # Синтаксис списковой операции, print($a, "string", $b); # Синтаксис вызова функции.
```

(Встроенные функции более подробно рассматриваются в главе П.)

Замечание

Описание многих встроенных функций (списковых операций) можно найти в главах, в которых вводятся понятия языка, для работы с которыми и предназначены определенные встроенные функции. Например, функции `print`, `printf` и `write` описаны в главе 6.

Операции заключения в кавычки

Кавычки (одинарные, двойные и обратные) в Perl мы используем для задания строковых литералов, причем получающиеся результирующие строковые данные существенно зависят от используемого типа кавычек: символы строки в одинарных кавычках трактуются так, как они в ней заданы, тогда как некоторые символы (\$, @) или даже последовательности символов (\n, \t) в строке в двойных кавычках выполняют определенные действия. Всё дело в том, что в Perl кавычки – это всего лишь

удобный синтаксический эквивалент определенных операций, выполняемых над символами строки. В языке, кроме трех перечисленных операций заключения в кавычки, определен еще ряд операций, выполняющих определенные действия со строковыми данными и внешне похожих на операции заключения в кавычки, на которые мы будем в дальнейшем ссылаться так же, как на операции заключения в кавычки.

Все операции *заключения в кавычки* представлены в табл. 4.3 с эквивалентным синтаксисом (если такой существует) и кратким описанием действий, выполняемых при их выполнении.

Таблица 4.3. Операции *заключения в кавычки*

Общая форма	Эквивалентная форма	Значение	Возможность подстановки
<code>q{ }</code>	<code>* i</code>	Строковый литерал	Нет
<code>qq{ }</code>	<code>it ii'</code>	Строковый литерал	Да
<code>qx{ }</code>		Команда системы	Да
<code>qw { }</code>	<code>0</code>	Список слов	Нет
<code>m{ }</code>	<code>//</code>	Поиск по образцу	Да
<code>qr{ }</code>		Образец	Да
<code>s { } { }</code>		Подстановка	Да
<code>tr{ }{ }</code>	<code>y///</code>	Транслитерация	Нет

При использовании общей формы операции *заключения в кавычки* вместо фигурных скобок {}, представленных в табл. 4.3, можно использовать любую пару символов, выбранную в качестве разделителя. Если выбранный символ не является какой-либо скобкой (круглой, угловой, квадратной или фигурной), то он ставится в начале и в конце строки, к которой должна быть применена соответствующая операция, тогда как в случае использования скобок-разделителей сначала используется открывающая скобка, а в конце закрывающая. Между знаком операции и строками в символах-разделителях может быть произвольное число пробельных символов. Обычно в качестве разделителя программистами Perl используется косая строка "/", хотя это и не обязательно.

В табл. 4.3 в последнем столбце также указывается, осуществляет ли соответствующая операция подстановку значений скалярных переменных и массивов, а также интерпретацию управляющих символов.

В этом параграфе мы остановимся только на первых четырех операциях заключения в кавычки. Остальные операции, как непосредственно связанные с регулярными выражениями, будут подробно рассмотрены в главе 10.

Операция q{}

Эта операция аналогична заданию строкового литерала в одинарных кавычках. В нем каждый символ строки представляет самого себя, подстановка значений переменных не выполняется. Единственное исключение – обратная косая черта, за которой следует символ-разделитель или еще одна обратная косая черта. Эти последовательности символов позволяют ввести непосредственно в строку символ разделителя или обратную косую черту (хотя обратная косая черта и так представляет саму себя).

Несколько примеров:

```
q-дескриптор \<FILE>; # Стока символов: Дескриптор <FILE>
д!Каталог \\bin\usr\п!; # Стока символов: Каталог \bin\usr\п
'Каталог \\bin\usr\п'; # Эквивалентно предыдущей операции
```

Операция qq{}

Эта операция аналогична заданию строкового литерала в двойных кавычках. При ее выполнении осуществляется подстановка в строку значений скалярных переменных, начинающихся с символа \$, и переменных массивов скаляров, начинающихся с символа @, а также осуществляется интерпретация управляющих последовательностей. После выполнения указанных действий будут сформированы строковые данные. Для задания в строке символа разделителя, используемого в этой операции, можно воспользоваться обратной косой чертой перед этим символом.

Несколько примеров: (см. главу 3).

```
qq(print\\() - функция вывода); # Стока символов:
# print() - функция вывода $t = 123; qq/4еное\t$t\п/; # Стока символов:
# Целое 123 "Ifеноe\t$t\п"; # Эквивалентно предыдущей операции
```

Операция qx{}

Эта операция аналогична заданию строкового литерала в обратных кавычках. При ее вычислении сначала осуществляется подстановка значений скалярных переменных и переменных массивов скаляров (если такие присутствуют) в строку, заданную между разделителями операции, а затем полученная строка, как некая команда, передается на выполнение командному интерпретатору операционной системы и результат ее выполнения подставляется в формируемое операцией qx{} окончательное

строковое значение. Таким способом можно ввести в программу Perl результаты выполнения определенных команд или пользовательских программ.

Несколько примеров:

```
$file = "file.tmp";
qx{rm $file}; # Удаление файла с именем file.tmp
$rez = qx{prog1 -a}; # Переменная
$rez содержит результаты вывода
# на экран программы prog1
$rez = 'prog1 -a'; # Эквивалентно предыдущей операции
```

Операция qw{}

Эта операция возвращает список слов, выделенных из строки, заданной между разделителями операции. Разделителями между словами считаются пробельные символы:

```
@m = qw( one two ); # Эквивалентно: $m[0] = 'one'; $m[1] = 'two'; Действие операции qw{СТРОКА}
эквивалентно действию встроенной функции
split ' ', qf{СТРОКА});
```

(Описание функции split см. в главе 10.)

Наиболее часто встречающаяся ошибка при использовании этой операции – отделить слова запятыми. При включенном режиме отображения предупреждений -w будет сгенерировано сообщение о том, что, возможно, запятая используется для разделения слов, а неходит в состав слова.

Операция "документ здесь"

В Perl реализована еще одна интересная возможность "ввода" строковых данных в программу, которая основана на синтаксисе командного интерпретатора shell системы UNIX. Она позволяет определить в программе строковые данные большого объема, расположенные в нескольких последовательных строках текста программы, и использовать их в качестве operandов разных операций: присваивания, печати и т. п.

Ее синтаксис прост: после знака операции < задается завершающий идентификатор, который служит признаком окончания задания строковых данных. Это означает, что все строки данных, расположенные между текущей строкой, содержащей операцию "документ здесь" и строкой, содержащей завершающий идентификатор, рассматриваются как единый фрагмент строковых данных:

```
$multi_line_string = <LINES; строка 1 строка 2 LINES
```

В приведенном фрагменте кода скалярная переменная \$multi_line_string будет содержать строку "строка 1\nстрока 2\n". Как видим, введенные нами с клавиатуры символы перехода на новую строку сохраняются при использовании операции "документ здесь". По умолчанию операция "документ здесь"

интерпретирует содержимое всех строк программы до завершающего идентификатора как строковые данные, заключенные в двойные кавычки, сохраняя в них символ перехода на новую строку "\n". Perl позволяет явно указать, как будут интерпретироваться данные при этой операции, заключив в двойные кавычки завершающий идентификатор операции. Следующие две операции "документ здесь" эквивалентны:

```
print «m;
line 1
m
print "m" <<;
line 1
m
```

Идентификатор можно задавать и в одинарных кавычках, и в обратных кавычках. В этом случае содержимое последующих строк программы до строки, содержащей завершающий идентификатор, трактуется как строковые данные в соответствующих кавычках.

В строке, физически ограничивающей данные операции "документ здесь", завершающий идентификатор задается без каких-либо кавычек.

При задании завершающего идентификатора в кавычках на строковые данные распространяются все правила подстановок переменных и управляющих символов, применяемые к строкам, ограниченным соответствующим типом кавычек. Пример 4.16 демонстрирует использование различных кавычек в операции "документ здесь".

```
#! perl -w
$var = "Александр"; print «FIRST; # Отобразит: Пользователь: # Пользователь: \t$var #
Александр FIRST #
print «'FIRST; # Отобразит: 1
Пользователь: I Пользователь:
\t$var # \t$var
FIRST #
$coml = "echo Alex";
print «FIRST'; # Отобразит: 4
$coml i Alex FIRST # .
```

Обратите внимание, что в примере 4.16 использовался одинаковый завершающий идентификатор FIRST. Это не приводит к двусмысленностям и ошибкам компиляции, так как компилятор ищет первую после операции « строку с завершающим идентификатором. Главное, чтобы завершающий идентификатор в строке завершения был задан именно так, как он задан в самой операции.

Замечание

При использовании операции "документ здесь" с завершающим идентификатором в обратных кавычках в некоторых операционных системах может возникнуть проблема с обработкой потока команд, определяемого в нескольких строках. Не все командные интерпретаторы могут обрабатывать несколько строк команд. Обычно они ориентированы на ввод команды в строке ввода, выполнения ее и ожидания следующего ввода команды. Некоторые командные оболочки могут обрабатывать несколько команд, заданных в одной строке через разделитель, например, командный интерпретатор cmd системы Windows NT, в котором разделителем служит символ &.

Если завершающий идентификатор в операции « задан без кавычек, то он должен следовать за знаком операции без каких-либо пробелов. Если такое случается, то Perl интерпретирует эту операцию с завершающим идентификатором пустая строка "" и ищет в тексте программы первую пустую строку, ограничивающую строковые данные этой операции:

```
$var = "Александр";
print « x2; i Отобразит 2 раза следующие 3 строки:
Пользователь: # Пользователь:
\t$var # Александр
x2 1x2
# Пустая строка завершает операцию «
```

В этом фрагменте кода ошибочно поставлен пробел перед завершающим идентификатором x2. Компилятор разобрал строку с операцией печати print следующим образом: строковые данные, вводимые операцией -«, завершаются пустой строкой, после чего они просто повторяются 2 раза (последовательность символов x2 понимается как операция повторения строки x с правым операндом равным 2).

Этот пример подобран специально таким образом, чтобы он нормально откомпилировался. Если вместо идентификатора x2 поставить, например FIRST, то компилятор сгенерирует ошибку.

Результат выполнения операции "документ здесь" можно использовать в качестве операнда строковых операций. Можно даже использовать несколько операций « в одном операторе, расположив строки их данных последовательно друг под другом, не забыв, конечно, строки с завершающим идентификатором:

```
$stack = «"ONE". «"TWO";
Первый
операнд
ONE
Второй
```

операнд

TWO

Значением скалярной переменной \$stack будет следующая строка:

"Первый\поперанд\пВторой\поперанд"

Выражения

С помощью операций в программе можно выполнить определенные действия над данными. Если для реализации алгоритма решения поставленной задачи необходимо произвести несколько действий над определенными данными, то это можно осуществить последовательным выполнением операций, сохраняя при необходимости их результаты во временных переменных, а можно построить одно выражение, составленное из последовательности операций и их operandов, и получить необходимый результат.

Итак, можно представить как последовательность operandов, соединенных одной или более операциями, результатом выполнения которой является единственное скалярное значение, массив или хеш. Operandы, в свою очередь, сами могут быть выражениями, что приводит к заданию в программе достаточно сложных выражений, вычисление которых начинается с их синтаксического разбора.

Когда компилятор начинает синтаксический разбор выражения Perl, он прежде всего выделяет в нем элементарные члены, называемые а потом по определенным правилам соединяет их знаками операций, заданными в выражении, т. е. определяет последовательность выполнения операций над термами в соответствии с определенным в языке приоритетом операций. После такого разбора выражение вычисляется в соответствии с полученной последовательностью термов, соединенных знаками операций.

Таким образом, выражение можно мыслить как последовательность термов, соединенных знаками операций.

Термы

Чтобы понять, как вычисляется выражение, следует знать, что представляет собой терм – элементарный член арифметического или логического выражения. В Perl является любой литерал, любая переменная, выражение в круглых скобках, любая строка символов, к которой применена операция заключения в кавычки, а также любая функция с параметрами, заключенными в круглые скобки. В конечном итоге только терм может быть operandом вычисляемой операции в выражении.

Из всех перечисленных объектов языка, которые рассматриваются как термы, требует некоторого пояснения последний – функция с параметрами в круглых скобках.

В действительности в Perl отсутствуют истинные функции, понимаемые в смысле, например, языка С, в котором регламентировано обращение в программе к функции указанием ее имени с параметрами, заданными в круглых скобках. По существу, "функции" языка Perl являются списковыми и унарными именованными ведущими себя как функции, так как синтаксис языка позволяет заключать их

параметры в круглые скобки. Вызывая в программе функцию (с заключенными или не заключенными в скобки параметрами), мы выполняем операцию (списковую или унарную именованную). Причем следует иметь в виду, что коль скоро выполняется операция, то она не только выполняет предписанные ей действия, но и возвращает определенный результат, который используется при вычислении выражения. Например, функция `print` возвращает Истину, если ее вывод завершен успешно, и Ложь в противном случае. Что напечатается при вычислении следующей операции?

```
print print "0";
```

Ответ – строка 01, так как первая операция `print` должна напечатать результат вычисления второй операции `print`, которая успешно выводит на экран монитора 0. Следовательно, ее результат Истина, а она представляется числом 1.

При синтаксическом разборе выражений и операторов (о них в следующем разделе) как термы рассматриваются конструкции `do{}` и `eval`, вызовы подпрограмм и методов объектов, анонимные конструкторы массивов скаляров [] и хешей {}, а также все операции ввода/вывода. Все эти понятия будут рассмотрены в последующих главах книги, но мы сочли необходимым, в целях полноты изложения термов, просто привести их полный список.

Приоритет операций

После выделения и вычисления термов выражение разбирается с целью выявления последовательности выполнения операций в выражении: какая из них должна быть выполнена раньше другой. Это достаточно ответственная процедура, так как порядок выполнения операций существенно влияет на результат вычисления всего выражения. Например, результатом вычисления выражения

```
4+3*2
```

будет 14, если сначала выполнить сложение, а потом умножение, и 10, если сначала выполнить умножение, а потом сложение. Дабы избежать подобных двусмысленностей в языках программирования, вводится или который учитывается при вычислении выражения. Приоритет операции умножения выше приоритета сложения, а поэтому наше арифметическое выражение будет однозначно вычислено равным 10.

В табл. 4.4 представлены все операции Perl в порядке убывания их приоритета, в ней также определен порядок выполнения операций с одинаковым приоритетом (столбец Сочетаемость).

Таблица 4.4. Приоритет и сочетаемость операций Perl

Приоритет	Операция	Сочетаемость
1	Вычисление термов и левосторонних списковых операций	Слева направо

2	->	Слева направо
3	++ --	Не сочетаются
4	* *	Справа налево
5	! ~ \ унарные + и -	Справа налево
6	=~ !=	Слева направо
7	* / % x	Слева направо
8	+ - .	Слева направо
9	« »	Слева направо
10	Именованные унарные операции	Не сочетаются
11	<><=>= lt gt le ge	Не сочетаются
12	== != <=> eq ne cmp	Не сочетаются
13	&	Слева направо
14	I \l	Слева направо
15	&&	Слева направо
16	II	Слева направо
17	Не сочетаются
18	? ;	Справа налево
19	= **= += -= .= *= /= %= x= \\$= = \l=	Справа налево

`<= >= % % =`

20	<code>,</code> \Rightarrow	Слева направо
21	Правосторонние списковые операции	Не сочетаются
22	<code>not</code>	Справа налево
23	<code>and</code>	Слева направо
24	<code>or xor</code>	Слева направо

Некоторые операции, приведенные в табл. 4.4, требуют пояснения. И первым в этом ряду стоят операции с наивысшим приоритетом: термы и левосторонние списковые операции. Термы мы определили в предыдущем разделе и там же разъяснили, что списковые операции и унарные именованные операции рассматриваются компилятором Perl как термы, если список их параметров заключен в круглые скобки. Так как умножение имеет больший приоритет, чем унарная именованная операция `sin`, то следующие операции вычисляются так, как указано в комментариях к ним:

```
use Math::Trig; # В пакете определена константа
# pi = 3.14159265358979
sin I * pi; # sin( 1 * pi) = 1.22460635382238e-016 sin (1) * pi; f (sin 1) * pi =
2.64355906408146
```

В последнем выражении `sin (i)` рассматривается как так как после имени операции первой распознаваемое лексемой стоит открывающая круглая скобка, а если это терм, – то и вычислять его надо в первую очередь, как операцию с наивысшим приоритетом.

Можно чисто визуально в тексте программы списковую или унарную именованную операцию с параметрами в круглых скобках сделать не похожей на вызов функции, поставив префикс `+` перед списком ее параметров:

```
sin +(1) * pi; # (sin 1} * pi = 2.64355906408146
```

Этот префикс не выполняет никакой семантической роли в программе, даже не преобразует параметр в числовой тип данных. Он просто служит для акцентирования того факта, что `sin` не является функцией, а представляет собой унарную именованную операцию.

Если в списковой операции отсутствуют скобки вокруг параметров, то она может иметь либо наивысший, либо самый низкий (ниже только логические операции `not`, `and`, `or` и `xor`) приоритет.

Это зависит от того, где расположена операция относительно других операций в выражении: слева или справа. Все операции в выражении, расположенные от списковой операции (сама операция расположена от них), имеют более высокий приоритет относительно такой списковой операции, и вычисляются, естественно, раньше нее. Именно это имелось в виду, когда в табл. 4.4 вносились строки с правосторонними списковыми операциями. Следующий пример иллюстрирует правостороннее расположение списковой операции:

```
$t = $n || print "Нуль, пустая строка или не определена!";
```

По замыслу программиста это выражение должно напечатать сообщение, если только значение переменной \$n равно нулю, пустой строке или не определено. На первый взгляд, кажется, так и должно быть: выполнится операция присваивания и возвратит присвоенное значение. Если оно не равняется нулю, пустой строке или значение переменной \$n не определено, то в булевом контексте операции логического ИЛИ (||) оно трактуется как Истина, а поэтому второй operand этой логической операции (операция печати) не вычисляется, так как мы помним, что логическое ИЛИ вычисляется по укороченной схеме. Однако реально переменной \$t будет присваиваться 1, правда сообщение будет печататься именно тогда, когда переменная \$n равна нулю, пустой строке или не определена.

В чем дело? Программист забыл о приоритете правосторонних списковых операций! В этом выражении списковая операция print расположена справа от всех остальных операций, поэтому она имеет самый низкий приоритет. Выражение будет вычисляться по следующему алгоритму. Сначала будет вычислен левый operand операции i i. Если он имеет значение Истина (переменная \$n имеет значение, не равное нулю или пустой строке), то второй operand этой операции (print) вычисляться не будет, а переменной \$t будет присвоена Истина, т. е. 1. Если первый operand вычисляется как Ложь (переменная \$n равна нулю, пустой строке или не определена), то вычисляется второй operand, выводящий сообщение на экран монитора. Но так как возвращаемым значением операции печати является Истина, то именно она и присваивается переменной \$t.

Правильное решение – использовать низкоприоритетную операцию or логического ИЛИ:

```
$m = $n or print "Нуль, пустая строка или не определена!; "или скобками изменить порядок выполнения операций:
```

```
($m = $n) I I print "Нуль, пустая строка или не определена!";
```

Теперь обратимся к случаю, когда списковая операция стоит от других операций в выражении. В этом случае, в соответствии с табл. 4.4, она имеет наивысший приоритет и что стоит справа от нее, она рассматривает как список своих параметров. Рассмотрим небольшой пример. Предположим, что необходимо удалить из массива @a все элементы, начиная со второго, и вставить их в создаваемый массив @t после второго элемента. Списковая операция splice со списком параметров @a, 1 удаляет из массива @a все элементы, начиная с элемента с индексом 1, т. е. со второго элемента до конца

массива, и возвращает список удаленных элементов. Ее можно использовать в конструкторе нового массива для решения поставленной задачи: слева (левосторонняя списковая операция).

```
@a = ("a1", "a2", "a3", "a4");
@m = ("m0", "m1", splice @a, 1, "t2", "t3"); ,
```

В конструкторе массива мы специально задали параметры операции `splice` без скобок. Если выполнить этот фрагмент и распечатать значения элементов массивов, то результат будет следующим:

```
@m: m0 m1
@a: a1 t3 a2 a3 a4
```

Совершенно не то, что нам надо: в массив `@t` не вставлен фрагмент массива `@a`, да и из него самого не удалены элементы, начиная со второго. Все дело в том, что операция `splice` в этом выражении левосторонняя, и весь расположенный справа от нее список рассматривает как список своих параметров: `@a, i, "m2", "t3"`. Ее третьим параметром должно быть число, определяющее количество удаляемых из массива элементов, начиная с элемента, индекс которого определен вторым параметром. В нашем случае третий параметр не является числовым, и функция завершается с ошибкой, возвращая Ложь. Исправить положение помогут опять скобки:

```
@m = ("m0", "m1", (splice @a, 1), "t2", "t3");
ИЛИ "''' @t = ("m0", "m1", splice (@a, 1), "t2", "t3");
```

Завершая разговор о приоритете выполнения операций, следует объяснить свойство и его практическое применение. Сочетаемость важна при вычислении выражений, содержащих операции с одинаковым приоритетом, и определяет порядок их вычисления. Рассмотрим выражение:

```
$t += $n += 1;
```

Как следует его понимать? Как $(\$t += \$n) += 1$ или как $\$t += (\$n += 1)$? Ответ дает правило сочетаемости. Смотрим в табл. 4.4 и видим, что все операции присваивания сочетаются справа налево. Это означает, что сначала должно выполниться присваивание $\$n += 1$, а потом результат увеличенной на единицу переменной $\$n$ прибавляется к переменной $\$t$. Следовательно, это выражение эквивалентно следующему:

```
$t += ($n += 1);
```

Аналогично применяется правило сочетаемости и к другим операциям языка Perl:

```
$a > $b < $c; # Эквивалентно: ($a > $b) < $c; Сочетаемость: слева направо. $a ** $b ** $c; # Эквивалентно:
$a ** ($b ** $c); Сочетаемость: справа налево.
```

Скобки изменяют порядок вычислений, определяемый по правилу приоритетов и сочетаемости. Любое, заключенное в скобки подвыражение, будет вычисляться с наивысшим приоритетом, так как Perl рассматривает его как терм, имеющий наивысший приоритет.

Контекст

Наш разговор о выражениях Perl был бы не полным, если бы обошли стороной такое понятие, как Каждая операция и каждый терм вычисляются в определенном контексте, который определяет поведение операции и интерпретацию возвращаемого ею значения. Существует два основных контекста: и В главе 3 мы уже немного познакомились с ними, когда определяли поведение конструктора массива и переменной массива в правой части оператора присваивания. Их можно "определить" так: если для выполнения операции требуются скалярные данные, то действует скалярный контекст, если необходимы массивы скаляров, то программа находится в списковом контексте. Например, если левый операнд операции присваивания, скалярная переменная, то действует скалярный контекст, если же в левой части задан массив, хеш или фрагмент массива или хеша, то вычисления в правой части осуществляются в списковом контексте. Присваивание списку скалярных переменных также инициирует списоковый контекст для вычислений, осуществляемых в правой части операции. контекст. скалярный списоковый.

Некоторые операции Perl распознают контекст, в котором они вычисляются, и возвращают список в списковом контексте и скалярное значение в скалярном контексте. Обладает ли операция подобным поведением, можно всегда выяснить из ее описания в документации. Например, можно рассматривать префикс @ перед идентификатором как унарную операцию объявления массива скаляров, распознающую контекст, в котором она вычисляется. В списковом контексте она возвращает список элементов массива, а в скалярном – число элементов массива.

Некоторые операции поддерживают списоковый контекст для своих operandов (в основном это списковые операции), и это также можно узнать из описания их синтаксиса, в котором присутствует СПИСОК (LIST). Например, известная уже нам операция создания фрагмента массива splice поддерживает списоковый контекст для своих параметров, поэтому ее можно вызывать вот таким образом:

```
@s = (1, 2);
splice @m, @s; # Эквивалентно: splice @m, 2; @s,
```

Скалярный контекст можно подразделить на и Если скалярный и списоковый контекст некоторыми операциями распознается, то ни одна операция не может определить, вычисляется ли она в числовом или строковом скалярном контексте. Perl просто при необходимости преобразует возвращаемые операцией числа в строки и наоборот. В некоторых случаях вообще не важно, возвращается ли операцией число или строка. Подобное, например, происходит при присваивании переменной какого-либо значения. Переменная просто принимает подтип присваиваемого значения. Такой контекст называется безразличным. числовой, строковый безразличный.

В языке существует контекст – специальный тип скалярного контекста, в котором вычисленное значение выражения трактуется только как Истина или Ложь. Как уже отмечалось ранее, в Perl нет специального булева типа данных. Здесь любая скалярная величина трактуется как Истина, если только она не равна пустой строке "" или числу 0. (Строковый эквивалент ложности – строка из одного нуля "0"; любая другая строка, эквивалентная нулевому значению, считается в булевом контексте истинной, например, "oo" или "o.o".)

Другим специфическим типом скалярного контекста является Он не только не заботится о подтипе возвращаемого значения – скалярный или числовой, но ему и не надо никакого возвращаемого значения. Этот контекст возникает при вычислении выражения без побочного эффекта, т. е. когда не изменяется никакая переменная программы. Например, следующие выражения вычисляются в void-контексте: void-контекст.

```
$n; "текст";
```

Этот контекст можно "обнаружить", если установить ключ -w компилятора Perl. Тогда можно получить предупреждающее сообщение следующего типа:

```
Useless use of variable in void context at D:\P\EX.PL line 3. a
```

(Бесполезное использование переменной в, void-контексте в строке 3 программы D:\P\EX.PL) x

Завершит наш рассказ о контексте (*interpolative context*), в котором вычисляются операции заключения в кавычки (кроме заключения в одинарные кавычки). В этом контексте вместо любой переменной, заданной в строке, в нее подставляется значение этой переменной, а также интерпретируются управляемые последовательности.

* * *

В этой главе мы изучили практически все скалярные операции языка Perl, чуть-чуть коснулись операций сопоставления по образцу, создания ссылок и операций ввода\вывода, познакомились с основами работы со списковыми и унарными именованными операциями. Узнали, что такое выражение, а также в каком порядке вычисляются в нем операции на основе их приоритета и сочетаемости. Научились выделять термы в выражениях и выяснили, в каких контекстах могут вычисляться выражения.

Вопросы для самоконтроля

1. Какую роль выполняют операции в программе?
2. Какие основные группы операций существуют в Perl?
3. Объясните "укороченную схему" вычисления логических операций. Где она используется?
4. Что такое выражение?
5. Определите понятие терм. Что считается термом в языке Perl?

6. Что такое приоритет операций и как он применяется при вычислении выражений?
7. Когда необходимо применять свойство сочетаемости операции?
8. Объясните понятие "контекст". Какие два основных типа контекста используются в языке Perl?

Упражнения

1. Что будет отображено на экране монитора при вычислении выражения

```
print print 1;
```

2. Определите результат вычисления следующих выражений:

```
print "0" I I print "1"; print "0" or print "1";
```

3. Что будет отображено на экране монитора и каковы будут значения элементов массива @t в результате выполнения следующей операции присваивания:

```
@t = (print "p\n", 2, print 3, 4);
```

4. Определите результат выполнения следующих операторов:

```
$var0 = 2;
```

```
$var1 = 1;
```

```
$rez1 = $var0 ** 3 * 2 !I 4 + $var1, $var1++;
```

```
$rez2 = ($var1++, $var0 ** 3 * 2 || 4 + $var1, "6");
```

```
@rez3 = ($var1++, $var0 ** 3 * 2 || 4 + $var1, "6");
```

5. Что напечатает следующий фрагмент программы при вводе числа или строки и почему:

```
$input = <STDIN>; $hello = "Hello "; $hello += $input; print $hello;
```

6. Найдите ошибку в программе:

```
$first_number =34;
```

```
$second_number = 150;
```

```
if( $first_number It $second_number ) { print $first_number; }
```