Все о Perl 6

Справочное руководство

А Загацкий

Все о Perl 6: Справочное руководство

А Загацкий

Publication date 13.10.2010

Аннотация

Данная книга является сборником статей о perl6.

Содержание

Список таблиц

Глава 1. Предисловие

Perl 6 представляет собой спецификацию, для которой существует несколько реализаций в виде компиляторов и интерпретаторов, каждая из которых находится на разной степени завершенности. Все эти реализации являются основной движущей силой развития языка, указывая на слабые стороны и противоречия в дизайне Perl 6. С их помощью обнаруживается функционал, сложный в реализации и недостаточно важный. Благодаря своего рода "естественному отбору" среди реализаций происходит процесс эволюции, который улучшает связанность и целостность спецификации языка Perl 6.

Perl 6 универсален, интуитивен и гибок. Он охватывает несколько парадигм таких как процедурное, объектно-ориентированное и функциональное программирование, а также предлагает мощные инструменты для обработки текста.

Perl должен оставаться Perl

Perl 6 по прежнему остается Perl. Что это означает ? Конечно же это не значит, что Perl 6 обладает такой же функциональностью или синтаксически совместим с Perl 5. В таком случае это была бы очередная версия Perl 5. Perl является философией и оба языка, Perl 5 и Perl 6, разделяют ее. Согласно этой философии существует больше одного способа достичь результата , а также простые вещи должны оставаться простыми, а сложные - возможными. Эти принципы связаны с прошлым, настоящим и будущим Perl и определяют фундаментальное предназначение Perl 6. В Perl 6 легкие вещи стали более легкими, а трудные - более возможными.

Об этой книге

Идея написания данной книги появилась, когда стало известно о выпуске первой стабильной версии "Rakudo star", реализации Perl 6 для виртуальной машины parrot. К этому моменту спецификация языка Perl 6 стала стабильной и изменения в нее стали не настолько кардинальными. Выпуск реализации Perl 6, пригодной для разработки программ, окончательно подтвердил факт - Perl 6 становиться реальным языком разработки.

К тому же написание книги - хороший способ изучить язык. Мое первое знакомство с языком произошло в 2005 году, благодаря книге "Perl 6 и Parrot: справочник", издательства "Кудиц-образ". Сейчас, спустя столько лет, произошло много изменений в стандарте языка и вероятно предстоит заново изучить его.

Основная задача этой книги - стать полезным источником знаний о языке Perl 6 для всех желающих изучить этот язык или просто интересующихся Perl 6. Данная книга - открыта для авторов и если вам интересно участвовать в написании этой книги, присылайте материалы в виде статей или патчей.

Исходные тексты этой книги располагаются по адресу http://github.com/zag/ru-perl6-book. Формат статей этой книги - Perldoc Pod. Частично материалы, описывающие этот формат на русском языке, размещены на страницах блога http://zag.ru. Если вы не хотите изучать Perldoc Pod, просто высылайте статьи в их оригинальном виде на адрес me(at)zag.ru. Они будут приведены к нужному формату.

Основным источником материалов для этой книги, на данный момент является английская версия. Ее пишут разработчики наиболее динамично развивающейся реализации Perl 6

- rakudo. Их книга располагается по адресу: http://github.com/perl6/book. Однако, надеюсь, по мере роста интереса к Perl 6 появятся желающие написать свои главы в этой книге.

Реализации Perl 6

Являясь спецификацией, Perl 6 подразумевает неограниченное количество реализаций. Любая из реализаций, успешно проходящая тесты, может назвать себя "Perl 6". Примеры, приведенные в книге, могут быть выполнены как с помощью компилятора *Rakudo Perl 6 (наиболее развитой на момент написания книги)*, так и любой другой.

Установка Rakudo и запуск программ

Подробные инструкции по установке Rakudo доступны по адресу http://www.rakudo.org/how-to-get-rakudo. Доступны как исходные тексты для сборки, так и уже предварительно скомпилированный пакет для Windows: http://sourceforge.net/projects/parrotwin32/files/.

Если вы являетесь пользователем FreeBSD, то для установки достаточно выполнить команду:

```
pkg_add -r rakudo
```

Проверить правильность установки Rakudo можно с помощью команды:

```
perl6 -e 'say "Hello world!"'
```

В случае неудачи, проверьте наличие пути для запуска perl6 в переменной `PATH`. Есть так же переменная `PERL6LIB`, с помощью которой можно использовать дополнительные модули для Perl 6. Для этого необходимо указать пути к ним в вашей системе аналогично `PERL5LIB` для Perl 5.

Дополнительные источники информации

Если вы хотите принять участие в развитии языка Perl 6, поделится своим опытом воспользуйтесь следующими ресурсами:

World Wide Web	Отправной точкой ресурсов, посвященных Perl 6, является домашняя страница языка : `http://perl6.org/`.
IRC	Задать вопросы о Perl 6 можно на канале `#perl6` по адресу `irc.freenode.net`.
Списоки рассылки	Для получения помощи о Perl 6 достаточно отправить письмо по адресу `perl6-users@perl.org`. По вопросам относящимся к спецификации Perl 6 или компиляторам можно обратиться по следующим адресам соответственно: `perl6-language@perl.org`, `perl6-compiler@perl.org`.

Глава 2. Базовый синтаксис

Изначальным предназначением Perl была обработка текстовых файлов. Это предназначение по прежнему является важным, однако Perl 5 также является мощным языком программирования общего назначения. Perl 6 является еще более развитым.

Представьте, что вы устраиваете турнир по настольному теннису. Рефери сообщают результаты соревнований в формате `Player 1 vs Player 2 | 3:2`, то есть участник `Player 1` выиграл у `Player 2` три сета против двух. Для определения победителя создадим скрипт, который просуммирует количество выигранных матчей и сетов для каждого игрока.

Входные данные выглядят следующим образом:

```
Beth Ana Charlie Dave
Ana vs Dave | 3:0
Charlie vs Beth | 3:1
Ana vs Beth | 2:3
Dave vs Charlie | 3:0
Ana vs Charlie | 3:1
Beth vs Dave | 0:3
```

Первая строка содержит список игроков, а каждая последующая - результаты матчей.

Один из способов получить ожидаемый результат с помощью Perl 6 следующий:

```
use v6;

my $file  = open 'scores';
my @names = $file.get.split(' ');

my %matches;
my %sets;

for $file.lines -> $line {
    my ($pairing, $result) = $line.split(' | ');
    my ($p1, $p2)          = $pairing.split(' vs ');
    my ($r1, $r2)          = $result.split(':');

    %sets{$p1} += $r1;
    %sets{$p2} += $r2;

    if $r1 > $r2 {
        %matches{$p1}++;
    } else {
        %matches{$p2}++;
    }
}

my @sorted = @names.sort({ %sets{$_} }).sort({ %matches{$_} }).reverse;
```

```
for @sorted -> $n {
    say "$n has won %matches{$n} matches and %sets{$n} sets";
}
```

На экран будет выведен следующий результат:

```
Ana has won 2 matches and 8 sets
Dave has won 2 matches and 6 sets
Charlie has won 1 matches and 4 sets
Beth has won 1 matches and 4 sets
```

Каждая программа на Perl 6 начинается с `use v6;`. Эта строка указывает компилятору необходимую версию Perl. Благодаря ей, при случайной попытке выполнить файл с помощью Perl 5, появиться полезное сообщение об ошибке.

В программе на Perl 6 может быть как ни одной, так и произвольное количество команд (утверждений). *Команда* завершается точкой с запятой или фигурной скобкой в конце строки:

```
my $file = open 'scores';
```

В данной строке `my` определяет лексическую переменную. Лексическая переменная доступна только в границах текущего блока. Если границы не определены, то видимость распространяется до конца файла. Блок - любая часть кода ограниченная фигурными скобками { }.

Имя переменной начинается с *сигила* - символа (значка), обладающего по утверждению wikipedia *(и тут я полностью согласен)* определенной магической силой. В Perl 6 к сигилам относятся такие символы, как `$`, `@`, `%` и `&` (изредка встречающийся в виде пары двоеточий `::`).

Сигилы наделяют переменную особыми характеристиками, наподобие возможности хранения простого или составного значения. После сигила следует идентификатор, состоящий из букв, цифр и символов подчеркивания. Между буквами возможно использование дефиса - или апострофа `'`, поэтому `isn't` и `double-click` являются допустимыми именами.

Сигил `$` указывается перед *скалярной* переменной. Эти переменные могут хранить одиночное значение.

Встроенная функция `open` открывает файл с именем *scores* и возвращает *дескриптор файла* - объект ассоциированный с указанным файлом. Знак равенства =*присваивает* дескриптор переменной слева и является способом сохранения дескриптора файла в переменной `$file`.

`'scores'` является *строковым литералом* . Строка является участком текста, в строковый литерал - строкой объявленной непосредственно в программе. В следующей строке строковый литерал указан в качестве аргумента для функции `open`.

```
my @names = $file.get.split(' ');
```

В данной строке виден правосторонний способ вызова *методов* - именованного набора команд. Так у хранящегося в переменной `$file` дескриптора файла вызывается метод `get`. Метод `get` читает и возвращает строку из файла, удаляя символ конца строки *(я предполагаю, что это перевод каретки)*. Далее следует вызов метода `split`. Он вызывается для строки, возвращаемой `get`. Эту строку называют *инвокантом (invocant)*. Метод `split` используется для разбиения строки-инвоканта на части, используя в качестве разделителя шаблон. Шаблон передается через аргумент. В нашем случае в качестве аргумента `split` получает строку, состоящую из символа пробела.

Таким образом строка из нашего примера `'Beth Ana Charlie Dave'` будет преобразована в список небольших строк: `'Beth', 'Ana', 'Charlie', 'Dave'`. А затем сохранена *(присвоена)* в массив `@names`. Сигил `@` маркирует указанную переменную как `Array` *(Массив)*. Массивы хранят упорядоченные списки.

Разделение по пустому символу не оптимально, не дает ожидаемого результата при наличии пробелов в конце строки или больше одного пробела в столбце данных наших соревнований. Для подобных задач наиболее подойдут способы извлечения данных в разделе посвященном регулярным выражениям.

```
my %matches;
my %sets;
```

Указанные две строки кода определяют два хэша. Сигил `%` помечает каждую из переменных как `Hash` *(Хэш)*. Хэш представляет собой неупорядоченный набор пар ключей и значений. В других языках программирования можно встретить другие названия для данного типа: *hash table*, *dictionary* или *map*. Получение из хэш-таблицы значения соответствующего запрашиваемому ключу `$key` производиться посредством выражения `%hash{$key}`.

:сноска В отличие от Perl 5, в Perl 6 сигил остается неизменным при обращении к массива или хэшам с использованием `[ ]` or `{ }`. Данная особенность называется *постоянство сигила (sigil invariance)*.

В программе расчета рейтингов матча, `%matches` хранит число выигранных матчей каждым игроком. В `%sets` запоминаются выигранные каждым игроком сеты.

Сигилы указывают на метод доступа к значениям. Переменные с сигилом `@` предоставляют доступ к значениям по номеру позиции; переменные с сигилом `%` - по строковому ключу. Сигил `$`, обычно, ассоциируется с общим контейнером, которым может содержать что угодно и доступ к данным так же может быть организован любым образом. Это значит, что скаляр может даже содержать составные объекты `Array` и `Hash`; сигил `$` указывает на тот факт, что данная переменная должна интерпретироваться как одиночное значение, даже в контексте где ожидаются множественные (как например `Array` и `Hash`).

```
for $file.lines -> $line {
    ...
}
```

Оператор `for` создает цикл, выполняющий *блок* кода, ограниченный фигурными скобками содержащий `...`, для каждого элемента в списке. Перед каждой итерацией переменная `$line` устанавливается в очередную строку, прочитанную из файла. `$file.lines` возвращает список строк из файла *scores*, начиная со строки, следующей за последней прочитанной `$file.get`. Чтение продолжается пока ну будет достигнут конец файла.

При первой итерации, `$line` будет содержать строку `Ana vs Dave | 3:0`. При второй - `Charlie vs Beth | 3:1`, и так далее.

```
    my ($pairing, $result) = $line.split(' | ');
```

С помощью `my` можно определить сразу несколько переменных одновременно. В правой части присвоения снова встречаем вызов вызов метода `split`, но в этот раз в качестве разделителя используется вертикальная черта с пробелами вокруг. Переменная `$pairing` получает значение первого элемента возвращаемого списка, а `$result` - второе.

В нашем примере, после обработки первой строки `$pairing` будет содержать строку `Ana vs Dave` и `$result` - `3:0`.

Следующие пару строк демонстрируют тот же прием:

```
    my ($p1, $p2) = $pairing.split(' vs ');
    my ($r1, $r2) = $result.split(':');
```

В первой строке извлекаются и сохраняются имена двух игроков в переменные `$p1` и `$p2`. В следующей строке примера результаты для каждого игрока сохраняются в переменные `$r1` и `$r2`.

После обработки первой строки файла переменные принимают следующие значения:

Таблица 2.1. Содержимое переменных

Переменная	Значение
`$line`	`'Ana vs Dave | 3:0'`
`$pairing`	`'Ana vs Dave'`
`$result`	`'3:0'`
`$p1`	`'Ana'`
`$p2`	`'Dave'`
`$r1`	`'3'`
`$r2`	`'0'`

Программа подсчитывает количество выигранных сетов каждым игроком в следующих строках:

```
    %sets{$p1} += $r1;
    %sets{$p2} += $r2;
```

Приведенные строки кода представляют собой сокращенную форму более общей:

```
%sets{$p1} = %sets{$p1} + $r1;
%sets{$p2} = %sets{$p2} + $r2;
```

Выражение `+= $r1` означает *увеличение значения в переменной, расположенной слева, на величину $r1* . Предыдущее значение суммируется с `$r1` и результат сохраняется в переменную слева. При выполнении первой итерации `%sets{$p1}` имеет особое значение и по умолчанию оно равно специальному значению `Any`. При выполнении арифметических операций `Any` трактуется как число со значением 0.

Перед указанными выше двух строк кода, хэш `%sets` пуст. При операциях сложения, отсутствующие ключи в хэше создаются в процессе выполнения, а значения равны 0. Это называется *автовивификация (autovivification)*. При первом выполнении цикла после этих двух строк `%sets` содержит `'Ana' => 3, 'Dave' => 0`. (Стрелка `=>` разделяет ключ от значения в `Паре` *(Pair)*.)

```
if $r1 > $r2 {
    %matches{$p1}++;
} else {
    %matches{$p2}++;
}
```

Если `$r1` имеет большее значение чем `$r2`, содержимое `%matches{$p1}` увеличивается на единицу. Если `$r1` не больше чем `$r2`,увеличивается на единицу `%matches{$p2}`. Также как в случае с `+=`, если в хэше отсутствовал ключ, он будет атовивифицирован *(это слово приходится даже проговаривать вслух, чтобы написать)* оператором инкремента.

`$thing++` - эквивалентен выражениям `$thing += 1` или `$thing = $thing + 1`, и представляет собой более краткую их форму, но с небольшим исключением: он возвращает значение `$thing` *предшествующее* увеличению на единицу. Если, как во многих языках программирования, используется `++` как префикс, то возвращается результат, т.е. увеличенное на единицу значение. Так `my $x = 1; say ++$x` выведет на экран `2`.

```
my @sorted = @names.sort({ %sets{$_} }).sort({ %matches{$_} }).reverse;
```

Данная строка содержит три самостоятельных шага. Метод массива `sort` возвращает отсортированную копию содержимого массива. Однако, по умолчанию сортировка производиться по содержимому. Для нашей задачи необходимо сортировка не по имени игроков, а по их победам. Для указания критерия сортировки методу `sort` передается *блок*, который преобразует массив элементов (в данном случае имена игроков) в данные для сортировки. Имена элементов передаются в *блок* через *локальную переменную*.

Блоки встречались и ранее: в цикле `for` использовался `-> $line { ... }`, а также при сравнении `if`. Блок - самодостаточный кусок кода Perl 6 с необязательной сигнатурой (а именно `-> $line` в примере для `for`). Подробнее описано в разделе посвященном сигнатурам.

Наиболее простым способом сортировки игроков по достигнутым результатам будет код `@names.sort({%matches{$_} })`, который сортирует по выигранным матчам. Однако Ana и Dave оба выиграли по два матча. Поэтому, для определения победителей в турнире, требуется анализ дополнительного критерия - количества выигранных сетов.

Когда два элемента массива имеют одинаковые значения, `sort` сохраняет их оригинальный порядок следования. В компьютерной науке данному поведению соответствует термин *устойчивая сортировка (stable sort)*. Программа использует эту особенность метода `sort` языка Perl 6 для получения результата, применяя сортировку дважды: сначала сортируя игроков по количеству выигранных сетов (второстепенный критерий определения победителя), а затем - по количеству выигранных матчей.

После первой сортировки имена игроков располагаются в следующем порядке: `Beth Charlie Dave Ana`. После второго шага данный порядок сохраняется. Связано с тем, что количество выигранных сетов связаны в той же последовательности, что и числовой ряд выигранных матчей. Однако, при проведении больших турниров возможны исключения.

`sort` производит сортировку в восходящем порядке, от наименьшего к большему. В случае подготовки списка победителей необходим обратный порядок. Вот почему производится вызов метода `.reverse` после второй сортировки. Затем список результатов сохраняется в `@sorted`.

```
for @sorted -> $n {
    say "$n has won %matches{$n} matches and %sets{$n} sets";
}
```

Для вывода результатов турнира, используется цикл по массиву `@sorted`, на каждом шаге которого имя очередного игрока сохраняется в переменную `$n`. Данный код можно прочитать следующим образом: "Для каждого элемента списка sorted: установить значение переменной `$n` равное текущему элементу списка, а затем выполнить блок". Команда `say` выводит аргументы на устройство вывода *(обычно это - экран)* и завершает вывод переводом курсора на новую строку. Чтобы вывести на экран без перевода курсора в конце строки, используется оператор `print`.

В процессе работы программы, на экране выводится не совсем точная копия строки, указанной в параметрах `say`. Вместо `$n` выводится содержимое переменной `$n` - имена игроков. Данная автоматическая подстановка значения переменой вместо ее имени называется *интерполяцией*. Интерполяция производится в строках, заключенных в двойные кавычки `"..."`. А в строках с одинарными кавычками `'...'` - нет.

```
my $names = 'things';
say 'Do not call me $names'; # Do not call me $names
say "Do not call me $names"; # Do not call me things
```

В заключенных в двойные кавычки строках Perl 6 может интерполировать не только переменные с сигилом `$`, но и блоки кода в фигурных скобках. Поскольку любой код Perl может быть указан в фигурных скобках, это делает возможным интерполировать переменные

с типами `Array` и `Hash`. Достаточно указать необходимую переменную внутри фигурных скобок.

Массивы внутри фигурных скобок интерполируются в строку с одним пробелом в качестве разделителя элементов. Хэши, помещенные в блок, преобразуются в очередность строк. Каждая строка содержит ключ и соответствующее ему значение, разделенные табуляцией. Завершается строка символом новой строки *(он же перевод каретки, или newline)*

```
say "Math: { 1 + 2 }"                    # Math: 3
my @people = <Luke Matthew Mark>;
say "The synoptics are: {@people}"       # The synoptics are: Luke Matthew Mark

say "{%sets}";                           # From the table tennis tournament

# Charlie 4
# Dave    6
# Ana     8
# Beth    4
```

Когда переменные с типом массив или хэш встречаются непосредственно в строке, заключенной в двойные кавычки, но не в внутри фигурных скобок, они интерполируются, если после имени переменной находится postcircumfix - скобочная пара следующая за утверждением. Примером может служить обращение к элементу массива: `@myarray[1]`. Интерполяция производится также, если между переменной и postcircumfix находятся вызовы методов.

```
my @flavours = <vanilla peach>;

say "we have @flavours";           # we have @flavours
say "we have @flavours[0]";        # we have vanilla
# so-called "Zen slice"
say "we have @flavours[]";         # we have vanilla peach

# method calls ending in postcircumfix
say "we have @flavours.sort()";    # we have peach vanilla

# chained method calls:
say "we have @flavours.sort.join(', ')";
                                # we have peach, vanilla
```

Упражнения

1. Входной формат данных для рассмотренного примера избыточен: первая строка содержит имена всех игроков, что излишне. Имена участвующих в турнире игроков можно получить из последующих строк.

Как изменить программу если строка с именами игроков отсутствует ? Подсказка: `%hash.keys` возвращает список всех ключей `%hash`.

Ответ: Достаточно удалить строку `my @names = $file.get.split(' ');`, и внести изменения в код:

```
my @sorted = @names.sort({ %sets{$_} }).sort({ %matches{$_} }).reverse;
```

... чтобы стало:

```
my @sorted = %sets.keys.sort({ %sets{$_} }).sort({ %matches{$_} }).reverse;
```

2. Вместо удаления избыточной строки, ее можно использовать для контроля наличия всех упомянутых в ней игроков среди результатов матча. Например, для обнаружения опечаток в именах. Каким образом можно изменить программу, чтобы добавить такую функциональность ?

Ответ: Ввести еще один хэш, в котором хранить в качестве ключей правильные имена игроков, а затем использовать его при чтении данных сетов:

```
...
my @names = $file.get.split(' ');
my %legitimate-players;
for @names -> $n {
    %legitimate-players{$n} = 1;
}

...

for $file.lines -> $line {
    my ($pairing, $result) = $line.split(' | ');
   my ($p1, $p2)          = $pairing.split(' vs ');
    for $p1, $p2 -> $p {
        if !%legitimate-players{$p} {
            say "Warning: '$p' is not on our list!";
        }
    }

    ...
}
```

Глава 3. Операторы

Операторы обеспечивают простой синтаксис для часто используемых действий. Они обладают специальным синтаксисом и позволяют манипулировать значениями. Вернемся к нашей турнирной таблице из предыдущей главы. Допустим вам потребовалось графически отобразить количество выигранных каждым игроком сетов в турнире. Следующий пример выводит на экран строки из символов X для создания горизонтальной столбчатой диаграммы:

```
use v6;

my @scores = 'Ana' => 8, 'Dave' => 6, 'Charlie' => 4, 'Beth' => 4;

my $screen-width         = 30;

my $label-area-width = 1 + [max] @scores».key».chars;
my $max-score            = [max] @scores».value;
my $unit                 = ($screen-width - $label-area-width) / $max-scor

for @scores {
   my $format = '%- ' ~ $label-area-width ~ "s%s\n";
   printf $format, .key, 'X' x ($unit * .value);
}
```

На экран будет выведен следующий результат:

```
Ana         XXXXXXXXXXXXXXXXXXXXX
Dave        XXXXXXXXXXXXXXX
Charlie XXXXXXXXXX
Beth        XXXXXXXXXX
```

Строка в примере:

```
my @scores = 'Ana' => 8, 'Dave' => 6, 'Charlie' => 4, 'Beth' => 4;
```

... содержит три разных оператора: =, =>, и ,.

Оператор = является *оператором присваивания* . Он берет значения, расположенные справа, и сохраняет их в переменной слева, а именно в переменной `@scores`.

Как и в других языках, основанных на синтаксисе C, Perl 6 допускает сокращенные формы для записи обычных присвоений. То есть вместо `$var = $var op EXPR` использовать `$var op= EXPR`. Например, ~ (тильда) - оператор строковой конкатенации (объединения); для добавления текста к концу строки достаточно выражения `$string ~= "text"`, которое является эквивалентом `$string = $string ~ "text"`.

Оператор `=>` (**=>** - *толстая стрелка*) создает `Пару` (pair) объектов. Пара содержит один ключ и одно значение; ключ располагается слева от оператора `=>`, а значение - справа. Этот оператор имеет одну особенность: парсер интерпретирует любой идентификатор в левой части выражения как строку. С учетом этой особенности строку из примера можно записать следующим образом:

```
my @scores = Ana => 8, Dave => 6, Charlie => 4, Beth => 4;
```

И наконец, оператор `,` создает `Парселы` (`Parcel`) - последовательности объектов. В данном случае объектами являются пары.

Все три рассмотренные оператора являются *инфиксными*, то есть располагаются между двумя *термами* (terms). Термом может быть литерал, например `8` или `"Dave"`, или комбинация других термов и операторов.

В предыдущей главе были использованы также другие типы операторов. Они сдержали инструкцию `%games{$p1}++;`. *Постциркумфиксный* (postcircumfix) оператор `{...}` указан после (*post*) терма, и содержит два символа (открывающую и закрывающую фигурные скобки), которые окружают (*circumfix*) другой терм. После postcircumfix оператора следует обычный *постфиксный* оператор `++`, который инкрементирует (увеличивает на единицу) переменную слева. Не допускается использование пробела между термом и его постфиксными (postfix) или постциркумфиксным (postcircumfix) операторами.

Еще один тип операторов - *префиксный* (prefix). Они указываются перед термом. Примером такого оператора служит `-`, который инвертирует указанное числовое значение: `my $x = -4`.

Оператор `-` еще означает вычитание, поэтому `say 5 - 4` напечатает `1`. Чтобы отличить префиксный оператор `-` от инфиксного `-`, парсер Perl 6 отслеживает контекст: ожидается ли в данный момент инфиксный оператор или терм. У терма может отсутствовать или указано сколько угодно префиксных операторов, то есть возможна следующее выражение : `say 4 + -5`. В нем, после `+` (инфиксного оператора), компилятор ожидает терм, и поэтому следующий за ним `-` интерпретируется как префиксный оператор для терма `5`.

Следующая строка содержит новые особенности :

```
my $label-area-width = 1 + [max] @scores».key».chars;
```

Начинатся указанная строка с безобидного определения переменной `my $label-area-width` и оператора присвоения. Затем следует простое операция сложения `1 + ...`. Правая часть оператора `+` более сложная. Инфиксный оператор `max` возвращает большее из двух значений, то есть `2 max 3` вернет 3. Квадратные скобки вокруг оператора дают инструкцию Perl 6 применить указанный в них оператор к списку поэлементно. Поэтому конструкция `[max] 1, 5, 3, 7` эквивалентна `1 max 5 max 3 max 7`, а результатом будет число `7`.

Также можно использовать `[+]` для получения суммы элементов списка, `[*]` - произведения и `[<=]` для проверки отсортирован ли список по убыванию.

Следующим идет выражение `@scores».key».chars`. Так же, как `@variable.method` вызывает метод у `@variable`, `@array».method` производит вызовы метода для каждого элемента в массиве `@array` и возвращает список результатов.

» представляет собой *гипер оператор*. Это также Unicode символ. В случае невозможности ввода данного символа, его можно заменить на два знака больше (`>>`) . За неимением Ubuntu под рукой следующее решение привожу в оригинале: *Ubuntu 10.4: In System/ Preferences/Keyboard/Layouts/Options/Compose Key position select one of the keys to be the "Compose" key. Then press Compose-key and the "greater than" key twice.*

Результатом `@scores».key` является список ключей пар в `@scores`, а `@scores».key».chars` возвращает список длин ключей в `@scores`.

Выражение `[max]@scores».key».chars` выдаст наибольшее из значений. Это так же идентично следующему коду:

```
@scores[0].key.chars
   max @scores[1].key.chars
   max @scores[2].key.chars
   max ...
```

Предваряющие выражение *(circumfix)* квадратные скобки являются *редукционным мета оператором*, который преобразует содержащийся в нем инфиксный оператор в оператор, который ожидает список (*listop*), а также последовательно осуществляет операции между элементами каждого из списков.

Для отображения имен игроков и столбцов диаграммы, программе необходима информация о количестве позиций на экране, отводимом для имен игроков. Для этого вычисляется максимальная длина имени и прибавляется 1 для отделения имени от начала столбца диаграммы. Полученный результат будет длиной подписи к столбцу диаграммы (*с одним уточнением: столбцы - горизотальные*).

Следующий текст определяет наибольшее количество очков:

```
my $max-score = [max] @scores».value;
```

Область диаграммы имеет ширину `$screen-width - $label-area-width`, равную разнице ширины экрана и длины подписи для данного столбца. Таким образом для каждой строки рейтинга потребуется вывести на экран :

```
my $unit = ($screen-width - $label-area-width) / $max-score;
```

... количество символов `X`. В процессе вычислений используются инфиксные операторы `-` и `/`.

Теперь вся необходимая информация известна и можно построить диаграмму:

```
for @scores {
   my $format = '%- ' ~ $label-area-width ~ "s%s\n";
   printf $format, .key, 'X' x ($unit * .value);
```

```
}
```

Данный код циклически обходит весь список `@scores`, связывая каждый из элементов со специальной переменной `$_`. Для каждого элемента используется встроенная функция `printf` , которая печатает на экране имя игрока и строку диаграммы. Данная функция похожа на `printf` в языках C и Perl 5. Она получает строку форматирования, которая описывает каким образом печатать следующие за ней параметры. Если `$label-area-width` равна 8, то строка форматирования будет `"%-8s%s\n"`. Это значит, что строка `%s` занимает 8 позиций (`'8'`) и выравнена по левому краю, за ней следует еще строка и символ новой строки `'\n'`. В нашем случае первой строкой является имя игрока. второй - строка диаграммы.

Инфиксный оператор `x`, или *оператор повторения*, формирует строку столбца. Он возвращает строку, состоящую из левого операнда, повторенного число раз, заданное правым операндом. То есть `'ab' x 3` вернет строку `'ababab'`. `.value` возвращает значение текущей пары, `($unit * .value)` умножает его на `$unit`, и `'X' x ($unit * .value)` возвращает строку с требуемым количеством символов.

Приоритетность

Объяснения примера в данной главе содержат важный момент, который не полностью очевиден. В следующей строке:

```
my @scores = 'Ana' => 8, 'Dave' => 6, 'Charlie' => 4, 'Beth' => 4;
```

.. в правой части присваивания определен список (согласно оператору `,`), состоящий из пар (благодаря `=>`), а затем присваивается переменной-массиву. Глядя на данное выражение вполне можно придумать другие способы интерпретации. Например Perl 5 интерпретирует как :

```
(my @scores = 'Ana') => 8, 'Dave' => 6, 'Charlie' => 4, 'Beth' => 4;
```

... так что в `@scores` будет содержаться только один элемент. А остальная часть выражения воспринимается как список констант и будет отброшена.

Правила приоритетности определяют способ обработки строки парсером. Правила приоритета в Perl 6 гласят, что инфиксный оператор `=>` имеет более сильную связь с аргументами чем инфиксный оператор `,`, который в свою очередь имеет больший приоритет чем оператор присваивания `=`.

На самом деле существует два оператора присваивания с разными приоритетом. Когда в правой части указан скаляр, используется *оператор присваивания еденичного значения* с высоким приоритетом. Иначе используется *списочный оператор присваивания*, который имеет меньший приоритет. Это позволяет следующим выражениям `$a = 1, $b = 2` и `@a = 1, 2` означать ожидаемое от них: присвоение значений двум переменным в списке и присвоение списка из двух значений одной переменной.

Правила приоритетов в Perl 6 позволяют сформулировать много обычных операций в естественном виде, не заботясь о их приоритетности. Однако если требуется изменить приоритет обработки, то достаточно взять в скобки выражение и данная группа получить наиболее высокий приоритет:

```
say 5 - 7 / 2;          # 5 - 3.5  = 1.5
say (5 - 7) / 2;        # (-2) / 2 =  -1
```

В приведенной ниже таблице приоритет убывает сверху вниз.

Таблица 3.1. Таблица приоритетов

Пример	Имя
`(), 42.5`	term
`42.rand`	вызовы методов и postcircumfixes
`$x++`	автоинкремент и автодекремент
`$x**2`	возведение в степень
`?$x, !$x`	логический префикс
`+$x, ~$x`	префиксные операторы контекстов
`2*3, 7/5`	мультипликативные инфиксные операторы
`1+2, 7-5`	инфиксные операторы сложения
`$x x 3`	оператор репликации (повторитель)
`$x ~ ".\n"`	строковая конкатенация
`1&2`	коньктивный AND (оператор объединения)
`1|2`	коньктивный OR (оператор объединения)
`abs $x`	именованный унарный префикс
`$x cmp 3`	non-chaining binary operators
`$x == 3`	chaining binary operators
`$x && $y`	бинарный логический инфикс AND
`$x || $y`	бинарный логический инфикс OR
`$x > 0 ?? 1 !! -1`	оператор условия
`$x = 1`	присванивание
`not $x`	унарный префикс отрицания
`1, 2`	запятая
`1, 2 Z @a`	инфиксный список
`@a = 1, 2`	префиксный список, присваивание списка
`$x and say "Yes"`	инфикс AND с низким приоритетом
`$x or die "No"`	инфикс OR с низким приритетом
`;`	завершние выражения

Сравнения и "Умное" сопоставление

Есть несколько способов сравнения объектов в Perl. Можно проверить равенство значений используя инфиксный оператор ===. Для неизменных (*immutable*) объектов (значения которых нельзя изменить, литералов. Например литерал `7` всегда будет `7`) это обычное

сравнение значений. Например `'hello'==='hello'` всегда верно потому, что обе строки неизменны и имеют одинаковое значение.

Для изменяемых объектов `===` сравнивает их идентичность. `===` возвращает истину, если его аргументы являются псевдонимами одного и того же объекта. Или же двое объектов идентичны, если это один и тот же объект. Даже если оба массива `@a` и `@b` *содержат* одинаковые значения, если их контейнеры разные объекты, они будут иметь различные идентичности и *не* будут тождественны при сравнении `===`:

```
my @a = 1, 2, 3;
my @b = 1, 2, 3;

say @a  === @a;         # 1
say @a  === @b;         # 0

# здесь используется идентичность
say 3   === 3;          # 1
say 'a' === 'a';   # 1

my $a = 'a';
say $a === 'a';          # 1
```

Оператор `eqv` возвращает `Истина` если два объекта одного типа *и* одинаковой структуры. Так для `@a` и `@b` указанных в примере, `@a eqv @b` истинно потому, что `@a` и `@b` содержат одни и те же значения. С другой стороны `'2' eqv 2` вернет `False`, так как аргумент слева строка, а справа - число, и таким образом они разных типов.

Сравнения чисел

Вы можете узнать, равны ли числовые значения двух объектов с помощью инфиксного оператора `==`. Если один из объектов не числовой, Perl произведет его преобразование в число перед сравнением. Если не будет подходящего способа преобразовать объект в число, Perl будет использовать `0` в качестве значения.

```
say 1 == 1.0;           # 1
say 1 == '1';           # 1
say 1 == '2';           # 0
say 3 == '3b'           # 1
```

Операторы `<`, `<=`, `>=`, и `>` - являются числовыми операторами сравнения и возвращают логическое значение сравнения. `!=` возвращает `True` *(Истина)*, если числовые значения объектов различны.

Если сравниваются списки или массивы,то вычисляется количество элементов в списке.

```
my @colors = <red blue green>;

if @colors == 3 {
```

```
    say "It's true, @colors contains 3 items";
}
```

Сравнение строк

Так же как == преобразует свои аргументы в числа, инфиксный оператор `eq` сравнивает равенство строк, преобразуя аргументы в строки при необходимости.

```
if $greeting eq 'hello' {
   say 'welcome';
}
```

Другие операторы сравнивают строки лексикографически.

Таблица 3.2. Операторы и сравнения

Числовые	Строковые	Значение
==	eq	равно *(equals)*
!=	ne	не равно *(not equal)*
!==	!eq	не равно *(not equal)*
<	lt	меньше чем *(less than)*
<=	le	меньше или равно *(less or equal)*
>	gt	больше чем *(greater than)*
>=	ge	больше или равно *(greater or equal)*

Например, `'a' lt 'b'` вернет истину, так же как `'a' lt 'aa'`.

`!=` на самом деле более удобная форма для `!==`, который в свою очередб представляет собой объединеие метаоператора `!` и инфиксного оператора ==. Такое же обяснение приминительно к `ne` и `!eq`.

Three-way сравнение

Операторы three-way сравнения получают два операнда и возвращают `Order::Increase`, если операнд слева меньше, `Order::Same` - если равны, `Order::Decrease` - если операнд справа меньше (`Order::Increase`, `Order::Same` и `Order::Decrease` являются перечислениями *(enums)*; см. подтипы). Для числовых сравнений используется оператор <=>, а для строковых это `leg` (от англ. *l*esser, *e*qual, *g*reater). Инфиксный оператор `cmp` также является оператором сравнения, возвращающий три ре-

зультата сравнения. Его особенность в том, что он зависит от типа аргументов: числа сравнивает как <=>, строки как leg и (например) пары сначала сравнивая ключи, а затем значения (если ключи равны).

```
say 10   <=> 5;           # +1
say 10   leg 5;           # because '1' lt '5'
say 'ab' leg 'a';         # +1, lexicographic comparison
```

Типичным применением упомянутых three-way операторов сравнения является сортировка. Метод .sort в списках получает блок или функцию, которые сравнивают свои два аргумента и возвращают значения отрицательные если меньше, 0 - если аргументы равны и больше 0, если первый аргумент больше второго. Эти результаты затем используются при сортировке для формирования результата.

```
say ~<abstract Concrete>.sort;
# output: Concrete abstract

say ~<abstract Concrete>.sort:
       -> $a, $b { uc($a) leg uc($b) };
# output: abstract Concrete
```

По умолчанию используется сортировка чувствительная к регистру, т.е. символы в верхнем регистре "больше" символов в нижем. В примере используется сортировка без учета регистра.

"Умное" сопоставление

Разные операторы сравнения приводит свои аргументы к определённым типам перед сравнением их. Это полезно, когда требуется конкретное сравнение, но типы параметров неизвестны. Perl 6 предоставляет особый оператор который позволяет производить сравнение "Делай Как Надо" *(Do The Right Thing)* с помощью ~~ - оператора "умного" сравнения.

```
if $pints-drunk ~~ 8 {
   say "Go home, you've had enough!";
}

if $country ~~ 'Sweden' {
   say "Meatballs with lingonberries and potato moose, please."
}

unless $group-size ~~ 2..4 {
   say "You must have between 2 and 4 people to book this tour.";
}
```

Оператор "умного" сопоставления всегда решает какого рода сравнение производить в зависимости от типа значения в правой части. В предыдущих примерах эти сравнения были числовым, строковым и сравнением диапазонов соответственно. В данной главе была продемонстрирована работа операторов сравнения: чисел - == и строк `eq`.Однако нет оператора для сравнения диапазонов. Это является частью возможностей "умного" сопоставления: более сложные типы позволяют реализовывать необычные идеи сочетая сравнения их с другими.

"Умное" сопоставление работает, вызывая метод `ACCEPTS` у правого операнда и передавая ему операнд слева как аргумент. Выражение `$answer ~~ 42` сводится к вызову `42.ACCEPTS($answer)`. Данная информация пригодится, когда вы прочитаете последующие главы, посвященные классам и методам. Вы тоже напишите вещи, которые смогут производить "умное" сопоставление, реализовав метод `ACCEPTS` для того, чтобы "работало как надо".